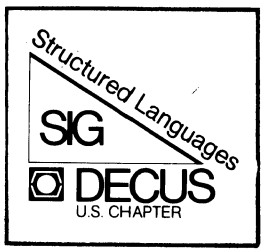


STRUCTURED LANGUAGES SIG

NEWSLETTER

Volume 4 Number 2 September 1980

Chairman's message:

Pascal SIG becomes the Structured Languages SIG

At the Spring DECUS Symposium in Chicago, the Pascal SIG was reorganized to include all structured languages in its interest area. This happened because there has been an increasing interest among members of the Pascal SIG in languages similar to Pascal. A group of users interested in the programming language C were the ones who got the change started. Hal Morris, who is our new symposia coordinator, wanted to see more emphasis on C at DECUS symposia. However, the proliferation of language SIGs would not be in the best interests of DECUS members, as it drains the supply of volunteer leaders who may have some common interests. We thought that the reorganization would increase the activity in the Pascal SIG by recognizing that we do in fact work with languages other than Pascal. With the continued interest in ADA, Concurrent Pascal, and Modula, and the new interest in the languages PRAXIS and C, we felt that the Pascal SIG should take these new users officially under its wing and build an even stronger and more active SIG.

An interesting part of the reorganization was the selection of a new name. We could not continue to be the Pascal SIG, as it would offend our new members. At a rather large meeting of potential leaders, we tossed several names about and voted on the best names. Our choices were "Structured languages," "Modern languages," "Technical languages," and just plain "Languages." We first chose the name Languages SIG, as it included everybody who did not have a home in any other languages related SIG. However, the COBOL and BASIC SIGs did not approve of that name. They were probably worried that we would eventually suck them up as well. The second most popular choice was the Structured Languages SIG, as it conveyed what we have been trying to impress upon the users of other languages such as FORTRAN, COBOL, BASIC, etc. for sometime, that there is a class of languages, which may be called Structured Languages, which are better for developing programs in all programming environments. We have also recognized that there may not be one all encompassing language which satisfies everybody's needs all of the time.

With the reorganization, we have set some new goals. First of all, we would like to see efficient programming environments available on all DIGITAL computers. It may be possible to do this with DIGITAL operating systems, or users may have to go to other operating systems to get the most out of their hardware. In either case, the cooperation of the users who are convinced that structured languages are better than non-structured languages, should be able to benefit the overall programming environment. Within the efficient

programming environment, the second goal of the SIG will be to see that the appropriate languages are available for use on DIGITAL computers. We do not have many qualms about who supplies the language translators for these languages. As in the past, if it is necessary for the SIG to produce an appropriate translator, every effort will be made to get it done. We will also keep continued pressure upon DIGITAL for them to provide the language translators we need. I know that many users cannot utilize unsupported products when implementing their software systems. DIGITAL is aware of this, and may yet come out with a decent programming language for the PDP-11 computer systems.

One of the most critical items the SIG must handle, is the compatability of the various languages and tools that programmers must use to implement their systems. We will be working towards a common set of software tools for the production of software, and towards common run-time systems for the various language translators that the SIG may support.

I would like to make you aware of who has agreed to work in the Structured Languages SIG. We are by no means completely staffed, and will accept more volunteers at any time. There will definitely be several meetings in San Diego concerning the future structure of the SIG, and who will be doing the work. I already mentioned that Hal Morris has agreed to be the symposia coordinator. In addition, Roy Touzeau will continue to be the newsletter editor, but he will have the help of several feature editors who will supply him with articles on languages other than Pascal. We currently have the following feature editors:

Steven McGready - C
Kockums Industries
P.O. Box 575
Tualatin, OR 97062
(503) 638-9811

Gray Beckman - RATFOR
Joint Center for Radiation Therapy
50 Binney Street
Boston, MA 02115
(617) 732-8508

I expect that we will soon get people interested in ADA, Modula, and PRAXIS. Hal Morris can be reached at:

Hal Morris
Prindle and Patrick
121 W. Oakland Avenue
Columbus, Ohio 43201
(614) 228-3233

We are also going to be revamping how the library functions. Bill Heidebrecht has been doing a great job with the symposia SIG tape copy project, but would like to pass his function on to someone else in the near future. The librarian will have to become aware of the various operating system dependent tape formats, as we are serving RT-11, RSTS, RSX, IAS, and Unix users.

I cannot emphasize enough that it takes volunteer efforts to make SIG work properly. We are going to have several business and organizational meetings in San Diego, so if you are truly interested in improving your programming environment, participate!

John R. Barr, Chairman

The fall symposium promises to be one of the best we have had, with many very interesting and informative sessions and pre-symposium seminars. Unfortunately, one problem continues to plague us - conflicts in scheduling. A glance at the preliminary program shows many interesting sessions scheduled for the same time, in spite of the valiant efforts of our new symposia coordinator Hal Morris. We regret this but were unable to avoid conflicts because of the large number of related sessions.

If you are able to take in pre-symposium seminars, there are a couple which look like they will be excellent. The SIG is sponsoring a seminar on the programming language C. C is a high level language which has been used for implementing compilers and operating systems as well as many utilities and applications programs. For example, it has been used to implement most of the UNIX operating system and most of the software that runs under it.

Another seminar which would be of interest is "Introduction to Pascal", sponsored by the DIGITAL Educational Services Group. This seminar is being taught by Kathleen Jensen who worked with Niklaus Wirth during the early seventies (when Pascal was being developed) and is co-author with Wirth of the Pascal Users Manual and Report. Many of you may have heard of the talk Ms. Jensen gave a year ago and wished you had not missed it. Here is your chance.

The regular symposium schedule begins at 9 am on Tuesday with the SIG roadmap. Try to attend this session if possible as it will be used to identify SIG related activities throughout the symposium. Information will also be presented about the various languages which have come under the wing of the SIG to enable you to better judge which sessions may be of interest to you.

In the afternoon there are several sessions on structured programming and the language APL. The concurrent languages panel in the evening will feature Modula, Ada and Concurrent Pascal. Those interested in Modula should also notice the session (easily missed when scanning the program) at 8 pm on Thursday describing the use of Modula to write data acquisition software in a biomedical laboratory.

Beginning the day on Wednesday is the session on the new language PRAXIS. Unfortunately this session conflicts with the session on VAX-11 PL/1 but if you decide to go to the PL/1 session you may at least pick up the PRAXIS workshop later.

At the end of the day is an introductory session on the language C (hope you are not too tired by then). Two more sessions are scheduled Thursday afternoon describing the use of C in systems programming and other applications. The comparison between C and Bliss at 8 pm should be interesting.

If you make it to Friday be sure to get to the symposium planning session at 2 pm. It is a difficult job planning the sessions and we need all the help we can get. Join us and help make the SIG a real "force" in DECUS.

The preliminary schedule has a good overview of the various sessions, organized by subject. Look under the "Structured Languages SIG" section. It will fill in some of the gaps I have left.

Roy Touzeau, Editor

Österreichische
Studiengesellschaft für Atomenergie Ges.m.b.H.
Lenaugasse 10 · A-1082 WIEN · Austria

Lenaugasse 10 · A-1082 WIEN · Austria

Department of Computer Science
c/o Dr. John R. Barr
University of Montana
Missoula, MT 59812
(4o6) 243-2883

Institut für Physik
Forschungszentrum Seibersdorf
Telefon: (02254) / 80 *
Telex: 014 / 353
Telegramm: austratom wien
Bankverbindungen
C A - Bankverein: 26-34343/02
E. ö. Spar-Casse: 012-10122
Österr. Länderbank: 106-100-432

Ihr Zeichen	Ihre Nachricht vom	Unser Zeichen	Sachbearbeiter	Telefon (Durchwahl)	Datum
		PH/Ma/Schw		*	198o-o7-o1

Betreff:

News from the PDP11/RSX Concurrent Pascal System

A new version of the Concurrent Pascal System under RSX11M is now available. The main improvements are:

- The kernel/interpreter is faster. No system directives are needed to protect process switching from interrupts (AST-routines).
- Generalized queue management:
  All queues (delay-,monitor entry-,IO-, and ready queue)
  are organized as doubly linked circular queues.
- EXTERNAL assembly written routines can be added by the user. This facility allows the user to implement new IO-routines or time-critical routines conveniently.
- Multiple DELAY's in a single queue variable:
  More than just one process can wait in a single queue variable. A CONTINUE operation resumes all waiting processes in the queue.
  This feature is extremly useful for programming time tables where all proces ses that are scheduled for a certain time can wait in a single queue variabl and are awakened all at a time by a clock process.

Those who have already received the RSX/RT11 Concurrent Pascal version can get a copy of the new version free of charge, if they send me a tape or any other distribution medium.

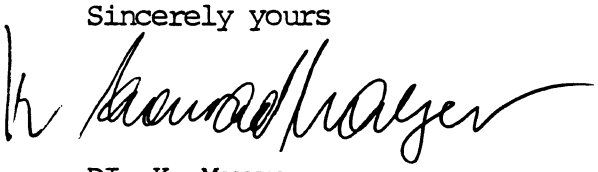
PASCAL-S for PDP-8

Prof. Stegbauer from the HTL Mödling (near Vienna) has implemented a compiler interpreter of Wirth's Pascal subset PASCAL-S on a 28k PDP8 for teaching purposes. The operating system is OS/8. All messages are in German.
You can receive a free copy if you send me an OS/8 formatted DEC tape.

A MODULA-2 like Extension of OMSI Pascal V1.2

We are now working on a runtime system for OMSI Pascal V1.2 that extends Pascal with MODULA-2 like concurrent programming features. If you work on similar problems or if you are interested in this work please write to me.

Sincerely yours



DI. K. Mayer

```
>PAS
KERNEL START
SOURCE:SYNCTST.CPA
LIST   :TI:
ERRORS:TI:
CODE   :SYNCTST.COD
CONCURRENT PASCAL? [Y/N]:Y

[1,54]CP1.COD

0001 (* PROGRAM SYNCTST.CPA *)
0002 (* TEST MULTIPLE DELAY,CONTINUE *)
0003
0004 (* IN THE NEW CONCURRENT PASCAL KERNEL IT IS POSSIBLE
0005    THAT MORE THAN ONLY ONE PROCESS CAN WAIT IN A QUEUE VARIABLE.
0006    A 'CONTINUE' OPERATION OF THIS VARIABLE RESUMES ALL WAITING
0007    PROCESSES IN THE QUEUE.
0008    THIS FEATURE IS EXTREMLY USEFUL FOR PROGRAMMING TIME TABLES
0009    WHERE ALL PROCESSES, THAT ARE SCHEDULED FOR A CERTAIN TIME,
0010    CAN WAIT IN A SINGLE QUEUE VARIABLE AND ARE AWAKEND ALL AT A TIME
0011    BY THE CLOCK PROCESS. *)
0012
0013 TYPE IOPARAM=RECORD OPERATION,STATUS,ARG:INTEGER END;
0014 CONST NL='(:10:)';
0015
0016 CONST PMAX= 26;
0017 TYPE PINDEX=1..PMAX;
0018
0019 TYPE MTYP= MONITOR;
0020 VAR Q:  QUEUE;
0021 PROCEDURE ENTRY DELAYPROC;
0022   BEGIN
0023    DELAY(Q);
0024   END;
0025 PROCEDURE ENTRY CONTINUEPROC;
0026   BEGIN
0027    CONTINUE(Q);
0028   END;
0029 BEGIN  END;
```

```
0030
0031 TYPE TERMINAL=MONITOR;
0032 VAR PARAM:IOPARAM;X:CHAR;
0033 PROCEDURE ENTRY WRITE(C:CHAR);
0034  BEGIN
0035    X:=C; IO(X,PARAM,0);
0036  END;
0037  BEGIN
0038   PARAM.OPERATION:=1;(*OUTPUT*)
0039   X:=' ';
0040  END;
0041
0042
0043 TYPE PTYP=PROCESS(C:CHAR;M:MTYP;T:TERMINAL);
0044 VAR X:CHAR;
0045 BEGIN
0046   X:=C;
0047  REPEAT
0048     M.DELAYPROC;
0049     T.WRITE(C);
0050     X:=SUCC(X);
0051  UNTIL X >='Z';
0052 END;
0053
0054 TYPE XTYP= PROCESS(LIMIT:INTEGER;M: MTYP; T:TERMINAL);
0055 VAR I:INTEGER;
0056 BEGIN
0057  FOR I:=1 TO LIMIT DO
0058  BEGIN
0059     T.WRITE(NL);
0060     M.CONTINUEPROC;
0061     WAIT;
0062   END;
0063 END;
0064
0065 VAR
0066     N: INTEGER;
0067     PN: ARRAY[1..PMAX] OF PTYP;
0068     M: MTYP; X: XTYP; T: TERMINAL;
0069 BEGIN
0070   WAIT;
0071   INIT M,T;
0072   FOR N:=1 TO PMAX DO INIT PN[N](CHR(ORD('A')+N-1),M,T);
0073   INIT X(10+PMAX,M,T);
[1,54]CP2.COD
[1,54]CP3.COD
[1,54]CP4.COD
[1,54]CP5.COD
[1,54]SCP6.COD
[1,54]SCP7.COD
COMPILATION OK
CODELENGTH:     2 PAGES
KERNEL END
>
```

```
>CER SYNCTST
*** CONCURRENT PASCAL KERNEL START ***

ZABCDEFGHIJKLMNOPQRSTUVWXY
XABCDEFGHIJKLMNOPQRSTUVW
WABCDEFGHIJKLMNOPQRSTUV
VABCDEFGHIJKLMNOPQRSTU
UABCDEFGHIJKLMNOPQRST
TABCDEFGHIJKLMNOPQRS
SABCDEFGHIJKLMNOPQR
RABCDEFGHIJKLMNOPQ
QABCDEFGHIJKLMNOP
PABCDEFGHIJKLMNO
NABDEFGHIJKLMOC
CNABDEFGHIJKLM
MCABDEFGHIJKL
LCABDEFGHIJK
KCABDEFGHIJ
JCABDEFGHI
ICABDEFGH
HCABDEFG
GCABDEF
FCABDE
ECABD
DCAB
BCA
AB
A

   PROGRAM TERMINATED AT LINE 00063. IN PROCESS 00028.
   PROGRAM HISTORY:
   ENTER MONITOR     IN PROCESS 00028. AT LINE 00059.
   IO START          IN PROCESS 00028. AT LINE 00035.
   EXIT  MONITOR     IN PROCESS 00028. AT LINE 00036.
   ENTER MONITOR     IN PROCESS 00028. AT LINE 00060.
   CONTINUE          IN PROCESS 00028. AT LINE 00027.
   EXIT  MONITOR     IN PROCESS 00028. AT LINE 00028.
   IO START          IN PROCESS 00028. AT LINE 00061.
   END PROCESS       IN PROCESS 00028. AT LINE 00063.
   *** CONCURRENT PASCAL KERNEL END ***
   >
```

```
>PIP TI:=EXTERN.MAC
        .TITLE  EXTERN
; EXTERNAL ROUTINES FOR CONCURRENT PASCAL PROGRAMS
;
; An external assembly written routine can be called in a Concurrent
; Pascal program via the new standard procedure EXTERNAL(n).
; 'n' is the number of the external routine. The table EXTTAB[n]
; contains the address of the routine. Every extension or modification
; of external routines needs recompilation of the module EXTERN.MAC
; and a new TKB run of the Concurrent Pascal kernel CER.
;
; New IO operations can easily added to the Concurrent Pascal
; system using this facility. The poor IO capabilities of the
; IO statement could be replaced by appropriate external routines.
; The IO statement is still implemented for compatibility reasons.
;
; Example of an external procedure declaration within a C.P. program:
;
; PROCEDURE ABC(X,Y:INTEGER; VAR Z: INTEGER);
; BEGIN EXTERNAL(2) END;
; ....
; ABC(3,2,I);(* CALL OF ABC *)
;
; How to write external routines:
;
; - Insert the address of the routine at the n'th position in
;   table EXTTAB. The first index of EXTTAB is 0.
;   If necessary, correct the number of EXTTAB table entries EXTN.
; - Do not modify resisters R3,R4,R5. Data pushed on the stack
;   must be removed before leaving the routine. Do not use
;   more than 32 words of stackspace.
;   R0,R1,R2 may be used as scratch resisters.
; - Terminate the external routine with
;       JMP     NEXTIN
; - Parameters can be accessed via R4.
;   The parameter list is stored in reverse order starting at 10.(R4).
;   Integers,chars,booleans,and scalars are stored as 1 word values.
;   reals are stored as 4 word and sets as 8 word(128 bits) values.
;   VAR parameters and all structured data types (ARRAYs,RECORDs)
;   are called by reference (= address stored in parameter list).
;   If the routine ABC computes the sum Z:=X+Y , the code would be:
; ABC:  MOV     14.(R4),R0      ; get first parameter
;       ADD     12.(R4),R0      ; add second parameter
;       MOV     R0,10.(R4)      ; store result
;       JMP     NEXTIN          ; return
;
```

```
; How to write external IO routines:
;
; - IO operations that can not be completed immediately must be
;   synchronized with other processes using AST routines.
; - The kernel routine IOSTART suspends the currently active process.
;   IOSTART expects in R2 the address of a 1 word empty (=0) queue
;   variable.Example:
;       MOV     #Q1,R2
;       CALL    IOSTART
;       QIO$C   ...
;       ...
; Q1:   .WORD   0
;
;   IOSTART must be called immediately before starting
;   the IO operation ( in most cases the QIO directive).
;   If IOSTART is called after the QIO directive and the
;   IO operation is completed before IOSTART suspends the
;   current process, then the linkage between the processes
;   will be destroyed and the C.P. kernel will crash.
;   On the other hand IOSTART should be called as late as possible
;   since IOSTART switches from the current process to the
;   next process in the READY queue and therefore changes the
;   complete process environment ( even the stack pointer !).
;   Consequently you may not store data on the stack or in
;   temporary registers when calling IOSTART !
;   The only way to save data for operations after IOSTART is
;   to store them in a private memory area.
;   Another way to circumvent this problem is to disable
;   AST recognition (DSAR$S) before executing the QIO directive
;   and to enable AST recognition (ENAR$S) after calling IOSTART.
;   Example:
;       DSAR$S
;       QIO$S ...
;       MOV     #Q1,R2
;       CALL    IOSTART
;       ENAR$S
;       JMP     NEXTIN
;   Note that this method takes more time than the former one !
;
; - The AST routine specified in the IO directive must save
;   registers R0,R1,R2 on the stack. The AST routine loads
;   the address of the queue variable in R2 and terminates with
;       JMP     ASTEXIT
;   Only R0,R1,R2 may be on the stack. AST specific parameters
;   must have been removed from stack.
;   ASTEXIT will resume the process waiting in the queue and
;   restore the registers.
;
; The following program gives an example of two external routines.
; The first routine performs a 'READ AFTER PROMT' operation on LUN 5.
; The second routine is a function and computes the maximum value
; of an array of N integers. The result is the index of the largest element.
;
```

```
; Declaration and call of the routines in Concurrent Pascal:
;
; TYPE AINDEX=1..5; ATYPE=ARRAY[AINDEX] OF INTEGER;
;      BINDEX=1..10; BTYPE=ARRAY[BINDEX] OF CHAR;
; VAR A: ATYPE; B: BTYPE; AMAX: INTEGER;
; PROCEDURE READAFTERPROMT(PRMT: BTYPE;PRMTLENGTH: BINDEX;
;                          VAR REPLY: BTYPE; REPLYLENGTH: BINDEX);
; BEGIN EXTERNAL(0) END;
;
; FUNCTION MAXIND(A: ATYPE; ALENGTH: AINDEX): AINDEX;
; BEGIN EXTERNAL(1) END;
;
; BEGIN
; ....
; AMAX:=A[ MAXIND(A,5) ]
; READAFTERPROMT('INPUT?: ',7,B,10);
; ....
; END.
;
        .DSABLE GBL
        .GLOBL  EXTERN,IOSTART,ASTEXIT,NEXTIN,IOERROR,HARDERROR
;(*)(*)(*)(*)(*)(*)(*)(*)(*)(*)(*)(*)(*)(*)(*)(*)(*)(*)(*)(*)(*)(*)(*)(*)(*)(*)
EXTERN::                        ; START OF EXTERNAL ROUTINE DISPATCHER
        MOV     (SP)+,R0        ; GET ROUTINE NUMBER
        CMP     R0,#EXTN        ; IN RANGE 0..EXTN-1 ?
        BLO     1$              ; O.K. IF LO
        JMP     IOERROR         ; TERMINATE WITH ERROR MESSAGE
1$:     ASL     R0              ; MAKE WORD OFFSET
        JMP     @EXTTAB(R0)     ; CASE N OF ....
;(*)(*)(*)(*)(*)(*)(*)(*)(*)(*)(*)(*)(*)(*)(*)(*)(*)(*)(*)(*)(*)(*)(*)(*)(*)(*)
;
EXTTAB:
        .WORD   RPRMT
        .WORD   MAXIND
EXTN=2                          ; NUMBER OF TABLE ENTRIES
;
; READ AFTER PROMT ROUTINE:
        .GLOBL  IO.RPR
        .MCALL  QIO$S,DSAR$S,ENAR$S
RPRMT:
        DSAR$S                  ; DISABLE AST RECOGNITION
        QIO$S   #IO.RPR,#5,,,,#RPRAST,<12.(R4),10.(R4),,16.(R4),14.(R4),#44>
        BCC     1$              ; DIRECTIVE O.K.?
        CALL    HARDERROR       ; FATAL EXIT
1$:     MOV     #QRPRMT,R2      ; LOAD ADDRESS OF QUEUE VARIABLE
        CALL    IOSTART         ; SUSPEND CURRENT PROCESS
                                ; AND SWITCH TO NEXT PROCESS
        ENAR$S                  ; ENABLE AST RECOGNITION
        JMP     NEXTIN
;DATA:
QRPRMT: .WORD   0
; AST ROUTINE:
RPRAST:
        TST     (SP)+           ; REMOVE AST PARAMETER
        MOV     R0,-(SP)        ; SAVE R0,R1,R2
        MOV     R1,-(SP)
        MOV     R2,-(SP)
        MOV     #QRPRMT,R2      ; ADDRESS OF QUEUE VARIABLE
        JMP     ASTEXIT
; END OF AST ROUTINE
```

```
;
; FUNCTION MAXIND(A:ARRAY[1..N] OF INTEGER;N:INTEGER): INTEGER;
MAXIND:
        MOV     12.(R4),R1      ; I:=1 (* ADDRESS OF A[1] *)
        MOV     (R1),R0         ; TEMPMAX:= A[I]
        MOV     R1,-(SP)        ; LASTI:=1 (* INDEX OF LAST MAXIMUM*)
        MOV     10.(R4),R2      ; LIM:=N
        TST     (R1)+           ; I:=I+1
1$:     DEC     R2              ; FOR I := 2 TO LIM DO
        BLE     10$             ; BEGIN
        CMP     (R1)+,R0        ; IF A[I]>TEMPMAX
        BLE     1$              ; THEN
        MOV     -(R1),R0        ; BEGIN TEMPMAX := A[I]; LASTI:=I END
        MOV     R1,(SP)         ;
        TST     (R1)+           ;
        BR      1$              ;
10$:                            ; END;
        SUB     12.(R4),(SP)    ; (* COMPUTE INDEX FROM ADDRESS *)
        ASR     (SP)
        INC     (SP)
        MOV     (SP)+,14.(R4)   ; MAXIND := LASTI (* FUNCTION RESULT *)
        JMP     NEXTIN          ; END
;
        .END    ; THAT'S ALL
>
>
```

hello world!

Supplement for Users of 'C' vol. 0-num.0

Welcome to hello word, the sub-newsletter for C language users. When the old PASCAL SIG was transformed into the Structured Languages SIG, we (C users) were taken under the wing of the new group. We hope to publish a regular supplement to the Newsletter containing items of interest specifically to users of the C language.

Contributions and comments should be addressed to:

Steven McGeady, c/o Kockums Industries
P.O. Box 575, Tualatin, OR 97062

Contributions are preferred in machine-readable form, on RT or RSX format floppy disk or 800 bpi DOS magtape, in RUNOFF format. Otherwise, contributions should be camera-ready copy, on one side of 8.5 x 11 inch paper, with one-inch margins.

Table of Contents

Items of Business

The name of this newsletter was the rather dictatorial choice of the editor, yours truly. Understanding that readers may have better ideas, I would like to open up a competition for a permanent name for this newsletter. The winner will have his/her name printed in our hallowed pages. The judge will be the editor, who must in all truthfullness admit a certain bias. Nevertheless, your participation is encouraged.

The intended functions of this newsletter are many and varied, and I will summarize them.

1. Software Trading - Users who have software written wholly or partially in C who wish to share it with other users are encouraged to send their name, address, phone number, and a "manual page" (described later) to the editor. It will be published in the first issue after it is received.

2. Compiler Reporting - Trading information on the various C compilers that are available or that become available, with emphasis on those that run on DEC computers, or which generate code for DEC computers. Part of this will be "wish list" presentations to various vendors, i.e. Whitesmith's or Yourdon. Also, some amount of pressure will be exerted on DEC to come up with a C compiler.

3. Hints and Kinks - Little secrets (or big secrets) that you have come across that would be off interest to other C programmers. These could be anything from brief notes on nifty usages of bitfields to full-size papers on particular applications. As we are somewhat limited in space, some editorial discretion will be used.

4. The CONROY Compiler - Although the editor knows very little about this compiler at present, it is hoped that the C sub-SIG will take an active role in supporting this public-domain compiler.

5. Distributions - It is hoped that we will be able to team up with either the PASCAL portion of the SLSIG, or the fledgling UNIX(tm) SIG (if it exists yet) in order to provide a tape distribution at the DECUS symposiums.

The list above is in no particular order, nor does it pretend to be a complete list of potential activities.

For the record, UNIX is a trademark of Bell Laboratories. PDP-11, RT-11, RSX-11, and almost anything with an '11' in it are trademarks of Digital Equipment Corp. C is not anybody's trademark.

Steven McGeady
Newsletter Editor

Fall Symposium in San Diego

The contingent of C users in the SLSIG is eagerly awaiting the Fall DECUS conference, as we are giving a handful of talks which promise to be very interesting. The will be a panel discussion on the topic of Low-Level Machine Control, which will include persons extolling the virtues of BLISS, C, MODULA, and perhaps other languages. Along the same lines, there will be a short talk on High-Level Languages in Process Control, which will be a case-study of one company's choice of a high-level language for real-time control applications. A talk will be given on C for Systems Programming, which will be an expose utilizing C which are useful for system applications.

Several introductory talks will be presented, one of which is What is C?, which will be a discussion of what the C language can do, what it can't do, and what it shouldn't be asked to do. This talk is oriented toward someone with no prior knowledge of C. There will be two talks on Structured Programming theory and practice, isolated from specifics of one language.

There will be two C workshops, an Applications Workskop, and a Technical Workshop, which will present views of C and C applications without and with the "gory details", respectively.

The PASCAL portion of the group will present many other talks, which will not be described here, and there will be talks on PRAXIS, BLISS, FORTH, and possibly other languages. There will be a UNIX workshop, and a discussion of UNIX-like systems.

There may be a one-day C Tutorial presented before the symposium which will teach high-level language programmers about C. This is still in the planning stage, and has not been finalized.

The San Diego conference promises to be full of discussion on all aspects of computing, including C and other high-level languages, and everyone is encouraged to attend.

Subroutine Linkage in C

I have been asked by several people to explain the C subroutine linkage scheme in writing, and I have decided to take this opportunity to do so.

The subroutine linkage convention used by the PDP-11 C compiler is a small and very elegant approach to the problem of generalized subroutine linkage.

To describe the workings of the linkage, we will step through the instructions one by one, following the processor as it executes them. The C call:

```
func(arg1, arg2, arg3);
```

appears in assembler as:

```
        mov     arg3, -(sp)     ; push 1st arg onto stack
        mov     arg2, -(sp)     ; 2nd arg onto stack
        mov     arg1, -(sp)     ; first arg onto stack
        jsr     pc, func        ; call function
        ...                     ; continue
```

This is the calling routine. The called routine would be:

```
func:   jsr     r5, csv         ; call save routine
        ...                     ; execute function
        jmp     cret            ; call return routine
```

The steps that are taken may be broken down into six major sections. We will examine these sections in the order in which they occur. Sections 1 and 2 occur in the calling routine, and sections 3 through 6 occur in the called, with execution resuming in the calling routine after the initial jsr.

1. Arguments - or parameters are pushed onto the stack, last argument first, first argument last. This is accomplished with mov xxx, -(sp) instructions. At the end of this operation, that stack pointer has been decremented by the integer number of words pushed onto the stack. Remember that bytes are passed as integers, and that double words are passed as two integers. Floating-point numbers are passed as two or four integers.

2. Call Routine - with the instruction jsr pc, func. This effectively pushes the address of the next instruction after the jsr onto the stack. This will be referred to as the return address. This is, of course, the address where execution of the current routine will resume when the called function is complete. The program counter (pc) is then loaded with the address of the beginning

of the function to be called (func). Execution proceeds at this address.

3. Call CSV - with the instruction jsr r5, csv. CSV is the C register saving routine. This instruction moves the contents of register 5 onto the stack, and puts the return address into register 5. At this point the stack frame consists of: our calling routine return address; the arguments to the called routine; and the contents of register 5. The CSV routine first moves r5 into r0. (Registers 0 and 1 are reserved for return values from subroutines. Thus, at this point r0 and r1 are scratch registers.) The contents of the stack pointer (i.e. the address of the top of the stack) is moved into r5. Now, r0 contains the address of the beginning of the called routine, and r5 points to the top of our "stack frame". At this point, registers r4, r3, and r2 are pushed onto the stack, with mov rr, -(sp) instructions. These are followed by a clr -(sp) instruction, which reserves an empty scratch area at the bottom of the stack. Finally, the called routine is reentered with the jmp (r0) instruction, which transfers program control to the instruction after the jsr to csv.

4. Allocate Local Variables - by subtracting the size (in bytes) of the local data area from the stack pointer. Notice at this point that we still have r5 pointing at our original stack frame, so we can access the passed paramters with 4(r5) for the first argument, and 6(r5), 10(r5), etc. for succesive arguments. These addresses do not change no matter what we do to the stack. This is an aid in both compiler-writing and assembly language programming because the compiler (or assembly language programmer) need not perform complicated calculations to determine where things are on the stack every time the stack pointer is changed. Local variables are addressed either by there offsets from the stack pointer, or by negative offset from r5.

5. Execute Called Routine - using, if desired, temporary space on the stack. There is no obligation to clean off the stack after using temporary space. If the routine is to return an integer value to the caller, that value is left in register 0. If a long value is to be returned, it is left in the register pair r(0,1). Floating-point values are returned in a variety of ways, depending on the compiler and the presence of floating-point hardware.

6. Return to calling routine by jumping to the C register restore routine, CRET. This is accomplished with the instruction jmp cret. CRET first moves the stack frame

pointer, r5, into register 2, then restores registers r4, r3, and r2 with mov -(r2), rr instructions. Notice that register 2 is the last to be restored, so that it is overwritten only after it is no longer needed. R5 was not used for this because it is now moved to the stack pointer, effectively removing all local data that was placed on the stack by the called routine. The original (calling) r5 can now be restored by mov (sp)+, r5. At this point, only one word remains on the stack, that being the return address placed there by the original jsr pc. This is now loaded back into the PC, and the stack incremented, by the rts pc instruction. Execution continues in the calling routine at the instruction following the jsr instruction.

The actual code for the CSV and CRET routines is:

```
csv:
        mov     r5, r0
        mov     sp, r5
        mov     r4, -(sp)
        mov     r3, -(sp)
        mov     r2, -(sp)
        clr     -(sp)
        jmp     (r0)

cret:
        mov     r5, r2
        mov     -(r2), r4
        mov     -(r2), r3
        mov     -(r2), r2
        mov     r5, sp
        mov     (sp)+, r5
        rts     pc
```

The original UNIX V6 C compiler used r1 in CRET, but if long integers are to be returned in the register pair r(0,1), r1 must be preserved through CRET. Also, at least one compiler (Whitesmith's) generates a subset of the CSV routine in-line when no registers are used in the routine. Whitesmith's also has a variant CRET which does not restore the registers on return. I am not acquainted with the Yourdon or Conroy linkage methods, but it is presumed that they are similar to the UNIX method, described above.

The above explanation is, by design, quite short. To aid understanding, it is recommended that the reader first read the description of the jsr and rts instructions, and also draw pictures of the stack frame at various points. Also it is important to realize that the stack grows down, that is, toward location zero. Thus, when decremented, it grows, and it shrinks when incremented.

NAME: doc - A Documentation Convention

SYNOPSIS: (see below)

DESCRIPTION:

Doc is a description of a documentation convention that is particularly well suited for concisely documenting programs and subroutines. It consists of the following sections:

1. NAME - The name of the program or subroutine, followed by a dash, then a short phrase describing its function. The phrase should be worded such that it can be easy indexed by an automatic indexing program.

2. SYNOPSIS - The is a short and extremely concise description of the syntax of the call to the program or subroutine. If you are describing a program, the line should be a description of the syntax of the command line. For example:

```
mac [out][,list]=in1[,in2,...]

type file[/p1:n][/ou:file][/vt]
```

These are typical synopsis lines. The square brackets indicate optional portions of the command line. In the first example, "/sw" is generic for all possible switches. In the second example, each switch is given with its syntax. A function call is defined somewhat differently:

```
int main(argc, argv)
int argc;
char **argv;
```

3. DESCRIPTION - The is the guts of the documentation. It normally consists of one paragraph describing what the program or subroutine does, optionally followed by a paragraph elaborating on this. The next section is a list of either the command line arguments (parameters) and switches (flags), or, in the case of the subroutine, the calling parameters, and side-effects. This section is fairly free-form.

4. EXAMPLES - The section contains examples, if they are desired. The heading is omitted if no examples are given.

5. FILES - If the program uses some special files, they should be listed here.

6. DIAGNOSTICS - A description of all non-self-explanatory diagnostics.

7. BUGS - A list of actual bugs, limitations, unimplemented features, and unexpected side-effects.

8. SEE ALSO - A vector to more information about the item. The "manual page" should not be a tutorial. This section should vector the user to other documentation should he/she need help. This section should also refer to related programs or routines. If the manual page describes a subroutine, all external routines which it requires should be listed here.

9. AUTHOR - The section should contain your name, affiliation, address and phone number.

Other sections can be added at will if they are necesary. The main idea, however, is to keep the manual pages terse and concise. For a complex program, a separate user manual should be written.

BUGS:

The manual pages can sometimes be too concise, and sometimes too verbose. The format can hamper some persons who are not familier with the format. Mostly, it is not the be-all and end-all.

SEE ALSO:

The format was stolen from UNIX. One could refer to the UNIX documentation for some good examples.

C and Systems Programming: Introduction

This paper is intended to demonstrate, largely through examples, the potential of the C programming language for interfacing with an operating system, CPU, or peripheral hardware. This capability is not restricted to use with any particular system, and in fact, the examples are for use with RT-11, the only PDP-11 operating system which I have used extensively. I am using the C compiler from Whitesmith's Ltd., which runs on RT-11, RSX, RSTS, VMS, and UNIX. While it would be out of place to evaluate a commercial product here, I think people would like to know that it exists. For the record, I think that C is more than just a "systems programming language", and for a small shop doing a mixture of systems and applications work, it would do a good job. This runs counter to the maxim that whenever you do a job, you should first chose the best language to do it in, that maxim often proves impractical. One class of applications in which C excels is any kind of text manipulation from sorting to text-editing to compilers.

But regardless of how good or bad C may be for applications, most people with sensible reasons for using assembler (ignoring those who use it for sentimental reasons) would do better using C in at least 90% of their code. I know of much work being done in C, mostly with RSX or RT-11 systems, in process control, image processing, and the control of exotic hardware. C allows almost the complete machine control and the efficiency of assembler languages, while providing data structuring and the structured control flow mechanisms ("while", "if-then-else", and a few others which the purists may not like).

Kernighan and Ritchie's The C Programming Language is one of the better computor science textbooks around, and I recommend reading Chapter 1: "A Tutorial Introduction" and parts of chapter 2, which describes the operators. The book's index is very complete, and starts with all the operators (in case you wonder how they are alphabetized), so you could look up the operators as you need them. Part of the audience of the paper is someone in process control or wishing to do laboratory data acquisition who wants to quickly see if C will do their job. Such a person may get impatient reading through all of Kernighan and Ritchie, which does not use pointers for half the book, nor macros for 95% of the book.

What I am presenting now, which should be close to my talk to be given at DECUS Fall '80 in San Diego, is roughly the 1st 3 chapters of the following outline:

1) Expressions Involving Bit Manipulation: This presents bit manipulation. Using the macro preprocessor to define parameterized expressions lets me present useable general purpose "functions" such as getbits(x,p,n), which is the n-bit field of x starting at bit p, building getbits up in a gradual, modular way.

2) Pointers both for Accessing Specific Addresses and Machine Independant Uses (My meta-title at least). Again, an interesting topic to the intended audience, and it is

convenient to bring it up early so example programs can be written the way they are written in practice (At least the esoteric things I avoid will not be the very things my stated audience is interested in.).

3) I/O Processing and Interrupts on the PDP-11: An application of the preceding sections, and hopefully for some novices like myself, a revelation about how easy it is to learn about UNIBUS devices and interrupts with the help of a reasonable language.

4) Why is C Self-sufficient in Terms of I/O Conversions?
   a) Internal<->external conversions like atof(), RAD50 stuff.
   b) Variable-sized argument lists and how to write them.
   c) Sheer efficiency.
   d) Meshes well with what little assembler you need.

5) Structures: Especially how C structures can mimick existing O.S. structures that you "have to live with". E.g. defining structures which give you a handle on the RT-11 directory. A program to mimick "DIR/OUT:file/COL:1/BLO" was written in about 6 hours (by a novice - me), and from this can easily be build a pseudo-spooler, or a subroutine to allow programs to use wildcard file descriptors.

Like The C Programming Language, and Software Tools, by Kernighan and Plauger, both of which I admire, I have developed and presented a series of very small examples which do in fact do something useful, or a piece of something useful, or illustrate a broad category of programming, like interrupt-driven I/O.

I expect most readers will understand a structured language, such as PL/1 or Pascal. I assume some vague knowlege of assembler languages and/or operating systems. I mean knowing for instance that computers generally have various shift operations, that "A = A + B" might become one binary instruction, while "A = B + C" probably would not; that "A == 0" (A is equal to 0) will probably produce faster more compact code than "A == B". It would help to have understood at one time what the parts of some processor status word or device register mean, even if you don't remember much about it now.

Expressions Involving Bit Manipulation:

Definition of some operators: The following discussion is offerred for those who do not have a copy of Kernighan and Ritchie handy. However, as they do such a good job of describing the operators, I have not sweated over it too much, so my recommendation is that you read their chapter 1, and at least skim chapter 2, then use it as a reference. What follows is a very dry listing which you might skip now, and refer back to when you see an operator you don't recognize.

'=' is used in the traditional, FORTRAN, etc. way to denote assignment.

"a == b" is the proposition "a is equal to b" and is used as in "if(a == b)...", "while(a == b)...".

The following are octal constants: "07, 0377, 0370" by virtue of their leading '0'.

a += b means a = a+b.
a *= b means a = a*b.
a /= b means a = a/b.
The pattern generalizes to all arithmetic and logical operators.

x++ means increment x by one after evaluating the expression it is in. E.g. "y = x++;" is equivalent to "y=x ; x=x+1 ;".

++x means increment x by one before evaluating the expression. E.g. "y = ++x;" is equivalent to "x=x+1 ; y=x ;".

x-- and --s similarly mean decrement before and after. These are tough to understand without examples.

Brackets are used around array subscripts: "x[i]", "x[1][2]" (which you might expect to be written x[1,2].

The data types int and unsigned represent integers, but an unsigned is always positive, the high bit being used for magnitude. For example, 0177777, the largest possible 16 bit unsigned, is internally the same as the int -1. Shift operations may affect unsigneds and ints differently.

"m >> n" is the value of m shifted right n bits. Hence 077 >> 3 == 070 >> 3 == 07. Floating point data types (float and double) can't be shifted. What happens on shifting a negative int right depends on the machine (and possibly the implementor). It is recommended (to implementors) that copies of the sign bit should roll in on the left so that it resembles division by a power of 2 (Though with improper rounding).

The & operator (in the right context, it has a 2nd meaning) is a bitwise and, so, for instance "m & MASK" has bits turned on in every position in which m and MASK both have a bit turned on. Or to turn off on a particular bit (for instance bit 7) without affecting the

rest, one can say "m &= BIT7_OFF", where BIT7OFF has all bits on except bit 7.

ints, unsigneds, and chars (act like ints or unsigneds depending on the machine, and have size about 8 bits) are what is used for logical propositions, and treated as "true" if and only if they are nonzero. This means an bitwise and is inappropriate for propositions, since 1 and 01 are both considered true, but 1 & 01 is not, therefore:

&& is provided, which is 1 if and only if its two operands are nonzero (would be considered true), and 0 otherwise.

Similarly, '|' and '||' are bitwise and logical or, respectively.

"^" is bitwise exclusive or,

"~" is bitwise complement,

"!" means truth value complement (and "!=" means does not equal).

"<<" means shift left.

"a % b" means remainder on dividing a by b

The expression "(cond) ? a : b" equals a if cond is true, or else it is b. A reasonable use of "? :" is:

```
output(printing_char(c) ? charformat : octalformat , c)
```

to print the character c in one of 2 ways depending on whether its ascii value is printable or not.

Examples Using Macros: The most important preprocessor command is #define. A macro definition consists of:

#define name-defined definition

which causes subsequent occurrences of name-defined to be replaced by definition in a modified copy of the program which is then sent to the compiler proper.

#define is frequently used to define constants. For example:

```
#define        YES     1
#define        NO      0
```

This is preferable to having two variables YES and NO initialized to 1 and 0, since there is no way, however bizarre, to alter their values; for instance "YES = NO" will be seen by the compiler as "1 = 0", and it will tell you politely that 1 does not belong on the left side of an assignment statement.

"name-defined" may have a list of arguments after it in parentheses, which makes things a little more involved. I hope that the examples will show how the arguments work.

Suppose I want LoByte(w) to be the value of the low order byte of w. The low order byte is "w & LO_MASK", where LO_MASK has just the bits of its low order byte turned on. The following, then, defines LoByte(w) as desired:

```
#define LO_MASK 0377
#define LoByte(x) ((x) & LO_MASK)
```

The parentheses around x in ((x) & LO_MASK) are a good idea since LoByte(x+3) will otherwise become (x+3 & LO_MASK), which looks like it might be mean the same as (x+(3 & LO_BYTE)) == (x + 3). In fact addition has a higher precedence than &, but LoByte(a | b) will get grouped wrong, and it seems best to me to just habitually put parentheses around all occurrences of the arguments in the body of a macro.

Finally, I want to develop some tools for creating and pulling apart dates in the RT-11 operating systems format which encodes them in one 16 bit word. This format has bits 0-4 (low order) representing a 5-bit integer y such that y + 1972 is the year, bits 5-9 represent the day of the month (in such a way that if you shift it right 5 bits and throw away the rest of the word, you have it in its ordinary form), and bits 10-13 have the number of the month in them (1 for January, ... , 12 for December). If day is the day of the month I want, then "day << 5" has the right thing in bits 5-9, or the 5 bits starting at bit 5 (day will overflow that if it's over 31, but then it's not a valid day of the month). Similarly, "month <<10" will produce a bitstring with month in bits 10,... and 0s elsewhere. And if the century is left off, "year-72" will have RT-11's representation of year in bits 0-5, and zeroes elsewhere (until 2004, when the format will cease to be valid. By oring these expressions

together, one should produce a date in the right format. Hence:

```
#define  MakRtdate(month, day, year)\
       ((month)<<10 | (day)<<5 | (year)-72)
       /* comment:  '\' means "continue onto next line. */
```

will cause a subsequent reference to "MakRtdate(4, 15, 80)" to produce the RT-11 internal representation of 4/15/80. Note that if all its arguments are constants, MakRtdate produces an expression containing only constants. The compiler notes this and does all the arithmetic at compile time, so when the resulting binary program runs, no shifting or oring result from this expression, and the expression can be used in places requiring a constant, such as variable initialization. For instance, the declaration:

```
extern int       taxday = MakRtdate(4, 15, 80) ;
```

generates one properly initialized word in memory, makes "taxday" refer to that word. It does not require any execution time.

```
#define TaxDayYr(yr)      MakRtdate(4, 15, (yr))
```

would make the declaration:

```
extern int       taxday = TaxDayYr(80) ;
```

compile identically to the previous declaration.

Next, I want to develop some tools to extract the extract the fields of an RT-11 date. To mask off the year (after which I will add 72), I only need to & it with a mask having just the low order 5 bits 1 and the rest 0. To get the low order byte of something, one needs to mask off the low order 8 bits of a word. It seems to be quite a general problem. Is there a general way to get a mask for the low order n bits? Such a method can be gotten from Kernighan and Ritchie, p. 45, although I will develop it in detail here and produce a macro rather than a function (or subroutine). The complement of zero, "~0", (any 0, whether 16 bits, 32 bits or what) is composed of all 1s. Shifting left always causes 0s to file in from the right, so shifting ~0 to the left n produces a word, "~0 << n", with its low order n bits all 0, and the other bits 1. This is the complement of what I want, so I complement it, or take "~(~0 << n)". So:

```
#define lowmask(n)        (~(~0 << (n)))
```

and to get the low order n bits of x:

```
#define lowbits(x, n) ((x) & lowmask(n))
```

and finally:

```
#define yearpart(date) (72 + lowbits(date, 5))
```

will extract the year in the date word date.

The other parts of the date word: day and month, are small integers occupying the 5 bits starting at bit 5, and the 4 bits starting at bit 10. How to get a numeric value from a bit field in the middle of a word? Suppose the field is in x, its starting bit is p, and its length is n. Bit p is the lowest order bit (rightmost, as generally pictured, consistently with "shift right", and "shift left"). If the whole word is shifted to the right p places, via "x >> p", in the result, the low order bit of the field is bit 0, anything of lower order is discarded, and we just need to get rid of anything above the low order n bits. But lowbits(z, n) does just this, so:

```
/* Extract the n bit field in x which starts at p. */
#define getbits(x,p,n)    lowbits((x) >> (p) , (n))
```

and what we were after becomes easy:

```
#define daypart(date)     getbits((date), 5, 5)
#define monthpart(date)   getbits((date), 10, 4)
```

so that the routine ptaxday():

```
[header]

extern int    taxday = MakRtdate(4, 15, 80)

ptaxday()
{
        printf("%d/%d/%d\n", monthpart(taxday), daypart(taxday),
                yearpart(taxday) ;
}
```

using a general formatted output routine, will print "4/15/80".

Using Specified Addresses

In communicating with peripheral devices, and to some extent, operating systems, it may be necessary to read and write to locations in memory which you specify. This is the way one communicates with devices on the UNIBUS. Partly for this purpose, C has a class of data types called pointer variables, of the form "pointer to this-type", where p is declared to be such a pointer by the declaration:

```
this-type *p ;
```

The '*' is what makes this declaration a pointer declaration. In subsequent code, after p is made to point to something, one can access what p points to as "*p". If y is a variable of type this-type, then

```
p = &y ;
```

makes p point to y. An example from Kernighan and Ritchie is:

```
       int     *p ;        /*   Declare p a pointer to int        */
```

```
int     x, y;       /*   x, y have type int.                */
...
p = &x ;            /*   p becomes pointer to x.            */
y  = *p;           /*   y gets contents of p, i.e.:  x.    */
```

which is a Rube Goldberg way of doing the same thing as:

```
y  = x  ;
```

In addition to the sort of thing done above, which is entirely portable, it is possible to have pointers point to an absolute location in computer memory. However, a pointer is more than just a location; it is "the integer at some address", or the "floating point" at some address, and this affects, among other things, whether "*p" is a 1, 2, 4, 8, or other byte quantity. Coercion is a method for converting things from one type to another, and should be used for converting integers which represent address into pointers of various types. For example, since the data types char, int, and double represent character, integer, and double-precision variables whose sizes are respectively 1, 2, and 8 bytes. The coercion operator is a type descriptor within parentheses, which comes before what is being coerced to that type. (char *), (int *), and (double *) convert what comes after them to address of a character, address of an integer, and address of a double-precision floating point, respectively. Consequently,

"* (char *) 0500"

refers to the byte at location 0500, taken as a character, while

*(double *) 0500

refers to the 8 bytes at location 0500, taken as a floating point number, and:

```
* (char   *) 0500 = * (char   *) 0600 ;
* (int    *) 0500 = * (int    *) 0600 ;
* (double *) 0500 = * (double *) 0600 ;
```

move, respectively, the 1, 2, and 8 bytes starting at location 0600 to the bytes starting at 0500. In the example programs to follow, I have used a couple of macros which would allow the first two statements above to be written as:

```
ByteAt(0500) = ByteAt(0600) ;
WordAt(0500) = WordAt(0600) ;
```

One of the macro definitions is:

```
#define ByteAt(address)   (* (char *) (address) )
```

which makes

"ByteAt(0500)" become "(* (char *) (0500))"

in the version of the program which the compiler sees.

The next sections will present programs which make use of specific locations and coercions fairly heavily.

Coercion can be used for purposes unrelated to pointers and addresses, such as converting floating point numbers to ints. For further reading, I suggest chapter 5 of Kernighan and Ritchie.
Example of Specified Addresses and Other Pointer Techniques:

The part of RT-11 called the Resident Monitor (or RMON) is so named because it always resides in memory, often alongside a running program. Some of the data in the RMON is thus available to programs, and several useful things are guaranteed to be certain distances from the start of the RMON. For instance, the system date is 0262 bytes from the start of the RMON (Recall that a number with a leading '0' is in octal.). So if an unsigned variable rmon contains the address of the start of the RMON, then "WordAt(rmon + 0262)" is the system date.

The address of the start of the Resident Monitor can vary for RT-11 systems with different options, but to make it easy to find, it is always kept in a fixed location, namely 054. It is placed there at bootstrap time. The following sequence puts the system date into rtdat.

```
unsigned  rmon, rtdat ;

...

rmon  = WordAt(054) ;
rtdat = WordAt(rmon + 0262) ;
```

rtdat may then be used any way you like, such as extracting the day, month, and year parts and printing them as in the "taxday" example for bitfields and the macro preprocessor.

Now, instead of using a formatted output routine from a standard library, as that example did, I would like to produce the output string with my own subroutine and output it using a system call. This will demonstrate two things. One is the ability to easily manipulate text in C, and particularly the role pointers play in text manipulation. This is, by the way, a system independent use of pointers, and these are at least as important as the system dependent uses. The second thing demonstrated is the advantage of having a language in which I/O conversion routines can be written, rather than one which has a lot of formatted I/O primitives built into it. Since formatted I/O is part of a support library (almost entirely written in C), and is not part of the language, the I/O built into a program can be as general or as specialized as I want. In this case I want to make it very specialized to obtain a tiny object program. The program occupies well under a block, which is 1/4K words (excluding a normal sized stack and the low memory area). The techniques shown are useful for putting a simple C program (with limited I/O needs) on a microcomputer with very little memory, possible a one-chip computer with the program burned into ROM. Since the compiler I use is available as a cross-compiler for "80 family" microcomputers, this is

a practical thought.

First, consider the relationship between arrays and pointers. In C, an array of type this-type differs from a pointer to this-type only in that the place to which the latter points cannot change, and that a certain amount of memory (as determined by the array's declaration) has been set aside there. So given the declarations:

```
char   buf[BUFSIZE] ;  /* NOTE: defines buf[0]..buf[BUFSIZE-1] */
char   *p = buf ;      /* initial value of p is buf */
```

p and buf are temporarily the same thing.

```
buf[2] = '0';     and    p[2] = '0';
```

both put the ascii value for the digit 0 in the same byte of memory.

*buf, buf[0], *p, p[0]

all mean the 0th element in the array of chars called buf.
and in general:

*(buf+n), buf[n], *(p+n), p[n] all mean the same thing.

The differences are that: (1) The declaration of buf is what actually caused space to be set aside for an array. (2) I can say "p = p+1;" or equivalently "p++;", which will make the above statements false. For instance, *p, or p[0], is now the same as *(buf+1), or buf[1]. However, "buf = buf+1;" (or "buf++;") is an illegal statement because buf is considered to be a constant. Note that buf is constant, not buf[i] for any index i.

Let me make a slight detour to pick up a couple of small tools. The following macros will pick out the digits of non-negative 1 or 2 digit numbers:

```
#define OnesPlace(n)    ((n) % 10)
        /* a % b is a modulo b, or remainder on dividing a by b */
#define TensPlace(n)    ((n) / 10)
        /*   13/10 == 1, 45/10 == 4.  Doesn't work for over     */
        /* 2-digit numbers.                                     */
#define ToDigit(ZeroToNine) (ZeroToNine + '0')
        /*   '0' is the same as 060, or ascii rep. of 0.        */
```

Now ToDigit(OnesPlace(27)) == '7' and ToDigit(TensPlace(27)) == '2'.

Now, the way to write a program to print the system date without these techniques is:

```
/*      D A Y . C   =   main
*/

#include      <c:std.h>

#define       WordAt(addr)   (* (unsigned *) (addr))

#define       lowmask(n)        (~(~0 << (n)))
#define       lowbits(x,n)      ((x) & lowmask(n))
#define       yearpart(date)    (72 + lowbits(date, 5))

#define       getbits(x,p,n)     lowbits((x) >> (p) , (n))
#define       daypart(date)      getbits((date), 5, 5)
#define       monthpart(date)    getbits((date), 10, 4)

#define       PutRtd(rtd)    putfmt("%i/%i/%i\n",\
                      monthpart(rtd),daypart(rtd),yearpart(rtd))

main()
{
      unsigned    rmon, rtdat;

      rmon  = WordAt(054) ;
      rtdat = WordAt(rmon + 0262) ;

      PutRtd(rtdat) ;
}
```

One way of rewriting it to avoid bringing in bulky standard I/O functions is:

```
/*      D A Y . C   =   main
*/

#include      <c:std.h>

#define       WordAt(addr)   (* (unsigned *) (addr))

_main()
{
      unsigned   rmon, rtdat;

      rmon  = WordAt(054) ;
      rtdat = WordAt(rmon + 0262) ;

      PutRtd(rtdat) ;
}
```

(The use of "_main()" instead of "main()" in my system prevents the facilities for I/O redirection being linked into the program. Since I am not using normal C I/O, they are useless.)

```
/*      P U T R T D . C   =   PutRtd(rtd)
*/

#include <c:std.h> /* Compile as if c:std.h were stuck in here. */

#define lowmask(n)       (~(~0 << (n)))
#define lowbits(x,n)     ((x) & lowmask(n))
#define yearpart(date)   (72 + lowbits(date, 5))

#define getbits(x,p,n)  lowbits((x) >> (p) , (n))
#define daypart(date)   getbits((date), 5, 5)
#define monthpart(date) getbits((date), 10, 4)

#define OnesPlace(n)     ((n) % 10)
#define TensPlace(n)     ((n) / 10)
#define ToDigit(n)       ((n) + '0')

#define RtPrint(msg)     emt(0351, msg)
        /* emt does most system calls; "0351" says which call */
PutRtd(rtd)
unsigned rtd;
{
     register unsigned m, y, d;
     char buf[9] ;

     m    = monthpart(rtd);
     d    = daypart(rtd);
     y    = yearpart(rtd);

     buf[0] = ToDigit(TensPlace(m));
     buf[1] = ToDigit(OnesPlace(m));
     buf[2] = '/';

     buf[3] = ToDigit(TensPlace(d));
     buf[4] = ToDigit(OnesPlace(d));
     buf[5] = '/';

     buf[6] = ToDigit(TensPlace(y));
     buf[7] = ToDigit(OnesPlace(y));
     buf[8] = NULL;

     RtPrint(buf);
}
```

This works by producing a NULL-terminated string, then printing it with a system call. NULL is defined to be 0 in C:STD.H.

There is one idiom which is very often used in C programs for text-processing. An example of it is:

```
*p++ = c;
```

where p is a char pointer and c is a char or int (whose content is small enough for a char). This adds c to a buffer if p points to the

next position to be filled in the buffer. Recall that

```
*p++ = c;
```

means the same thing as: *p = c; p = p+1;

So p starts as the pointer to "next position"; that position gets filled; then p gets incremented so it again points to "next position". For example:

```
char   buf[4], *p;
...
p = buf;
*p++ = 'o';
*p++ = 'u';
*p++ = 't';
*p   = NULL;
```

puts the NULL-terminated string "out" into the buffer buf.

```
register char  *from, *to;
...
while(*from != NULL)
      *to++ = *from ;
```

can be used to copy one NULL-terminated string to another. Finally, PutRtd() can be rewritten as (omitting parts that stay the same):

```
...
register char    *p;
...
p = buf;
*p++ = TensDigit(m); /* [a] (see below) */
*p++ = OnesDigit(m);
*p++ = '/';

*p++ = TensDigit(d); /* [b] */
*p++ = OnesDigit(d);
*p++ = '/';

*p++ = TensDigit(y);
*p++ = OnesDigit(y);
*p++ = NULL;
```

Now one needn't worry about exactly which character postion everything is going to; the characters are just tossed into the buffer in the right order.

The buffer-pointer technique provides some flexibility which it is otherwise quite awkward to get. For instance if I don't like having leading 0s, i.e., I want to get "6/2/80" instead of "06/02/80", I can replace [a] and [b] by:

```
*p = TensDigit(m);
if (*p != '0')
    p++;        /* otherwise, don't move; write on top of it */
...
*p = TensDigit(d);
if (*p != '0')
    p++;
```

With most languages, something like the 1st version of PutRtd() would be the only alternative, and to eliminate the trailing 0s would require switching from the constant indexes to a running index variable, which would make PutRtd() look a good bit more complex. Granted the "*p++ = c;" idiom may look complex (or bizarre or counter-revolutionary) if you're not used to it, but it is very analogous to something of the form "output-character(c);" except that I am outputting c to a string (very likely an output buffer) instead of directly to a terminal or file. By the way, switching to the pointer technique decreased the program size by 37 words. It could be improved quite a bit, but I think it is already pretty small for a high level language program.

(Hal Morris' discussion of C will be continued in the next issue with "I/O Processing and Interrupts on the PDP-11." - ed.)

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.