The PASCAL Corner

Why PASCAL? Every programmer has his or her own favorite language, and most love to talk about them. At the risk of boring you, let me tell you why I stopped fighting and switched.

I was a diehard BASIC programmer for years. But once I went through the considerable difficulty of learning PASCAL at a local community college, it began to make sense.

True, BASIC-X has lots of nifty functions built in, such as the POS (position) and TIM (timed input) statements. But in PASCAL you can design and build your own. I do miss the XI mode, the execute immediate debugger so handy to programmers just starting out (like training wheels for bicycle riders). And waiting for a long program to compile to find one missing semicolon can be sheer agony. Here's what you get in return:

A. On the 8000, interpretive BASIC-X and compiled PASCAL have a tortoise-and-hare relationship. It's just no contest.

B. The block-on-block manner in which PASCAL programs have to be written forces you to clearly analyze what you're trying to do and how to get there. (Improvising as you go along is a lot of fun, but you wouldn't want to buy a car that was designed that way). Long variable names also help to keep things sorted out. "MidwesternYearlySalesTotals" is easier to understand than "T0", for instance.

C. Since the introduction of the BASIC-X "CALL" statement, it became possible to pass a parameter to a compiled program, and back again. In this way, PASCAL can expand your BASIC-X capabilities.

D. There simply is no way to do in BASIC-X some of the things you can do easily in PASCAL. Submitting a batch job is one common example. By using executive requests, which don't exist in BASIC-X, you can drop to the underworld of machine-language. It gives you much more control over how your computer behaves. Similarly, you can interface with the FMS (file management system), the most efficient database manager existing for the 8000.

(COBOL programmers can send their remarks to the COBOL CORNER, c/o this magazine. I'm sure there will be plenty.)

XREQs (pronounced Eks-Rex) fall into three general catagories: User Requests, File Requests, and Executive Requests. They are extremely useful, but a little confusing at first. When I was learning how to use them, I relied heavily on examples. In the issues to come, we'll be presenting some of the solutions we've found to common programming problems. As is usual with all our donations, no warranties or guarantees are implied.

We'll start with something relatively simple. How do I make a program pause for a given interval? Unlike the BASIC-X TIM function, PASCAL doesn't have a preset method. You could make it count to a million by ones. You could put it into a loop that continually examines the system universal time, and fall out when a given value is reached, or RUNPROGRAM(.SYS:WAIT,PauseStr, X). All of these would waste time and memory space.

There is a User Request available that puts your process at the bottom of the service queue--in effect, puts it into suspended animation.

```
PROGRAM STALL(INPUT,OUTPUT);

CONST URDELAY  =  16#140; {Delay process UREQ, page 3-90 of the manual.
                           The tic-tac-toe character tells the system that
                           the following number is hexadecimal, a handy
                           little trick in itself.}

VAR DummyValue, RegisterZero, Seconds : INTEGER;


PROCEDURE GoToSleep(SleepSeconds : INTEGER);
{Suspends program execution for given time (sleep seconds)}

BEGIN
    {We start by loading a register with the ID number of a
     machine-language subroutine.  In this case, a Hex 140.}
    RegisterZero := URDELAY;

    DummyValue := 0;     {This could be any number.}

{Here is the actual call to the routine.  Some XREQs use all five
CPU registers in combination as needed.  They are used also to pass
results of the operation back to your PASCAL program, when the
work is finished.}
    UREQ(RegisterZero, SleepSeconds, DummyValue, DummyValue, DummyValue);

{If everything went OK, the first register will be zeroed out.  If
something fouled up, it will contain the number of an error message.
Routines exist to analyze the error number and tell you in English
what the problem was.  We'll show you one later.}
    IF RegisterZero < 0
       THEN TellWhatsWrong(RegisterZero);

END; {PROCEDURE GoToSleep}
```

```
{Here's an example of how you might drive the procedure above.}

BEGIN {Main program, STALL}

{Here's another tip.  PASCAL sees your terminal as just another
 output file, so you can manipulate output as though it were going to
 a line printer.  In this case, we are preceeding the text with the
 splat character.  This tells the machine to suppress the CR/LineFeed
 sequence at the end of the line.  In BASIC-X, an equivalent state-
 ment would be:

                 10 PRINT "HOW MANY SECONDS?";
                 20 INPUT S                                        }

    FORMAT(OUTPUT, '*');
    WRITELN('How many seconds should I pause? ');
    READLN(Seconds);
    GoToSleep(Seconds);
    WRITELN('I am running again.');

END. {Main Program, STALL}
```

As promised, here's a routine to analyze XREQ error numbers.

```
PROCEDURE TellWhatsWrong(ErrorNum : INTEGER);
{Interprets the error status received from an XREQ (R0).  Page 9-4 of
 the PASCAL manual, although here it's set up as a procedure.}

CONST URERRORGET = 16#028; {Get the error type, Page 3-100 of the UREQ
                            manual.}

TYPE ERRORSTR = STRING[48];     {This will store the error message.}
     ERRORSTRPTR = ^ERRORSTR;   {Tells where the string is in memory.}

     {This is a 4-byte array, used as a function to extract a byte.
      --Only part of the error number is relevant.}
     BYTE = PACKED ARRAY[0..3] OF 0..255;

VAR ErrStrPtrVariable : ERRORSTRPTR;  {Define a pointer variable}
```

```
BEGIN
    {Create a dynamic variable}
    NEW(ErrStrPtrVariable);

    {Set length to 48 (longest error message).  Then fill it with blanks}
    ErrStrPtrVariable^[1..48] := ' ';

    RR0 := URERRORGET;

    {Here's the starting address of the string,
     string, plus one to skip length field}
    RR1 := ORD(ErrStrPtrVariable) + 1;

    RR2 := ErrorNum;

    {Here's the actual call to the machine-language subroutine.}
    UREQ(RR0, RR1, RR2, DummyVal, 48);

    IF RR0 >= 0
       THEN BEGIN
               {RR1 returns carrying the unused portion of the string}
               WRITELN('XREQ ERROR: ', BYTE(ErrorNum)[3]:4, ' "',
                                  ErrStrPtrVariable^[1..48 - RR1],'"');
            END

       { We just got an XREQ error while interpreting an XREQ error.}
       ELSE WRITELN('UREQ ERROR:', BYTE(RR0)[3],
                                     '--Cannot interpret error type.');

       {It's hopeless.  Quit and set the error flag.}
       STOPPROGRAM(1);

END; {PROCEDURE Tell what is wrong with the Xreq}
```

Some of this probably seems a little arcane. To get myself started, I just copied examples verbatim, changing variable names as needed with +SCREDIT. After working with them for a while, the logic behind it all gets clearer.

--Ray