"Use of ATTACHed Interrupts in the UCSD p-System"

By Fritz Whittington
Texas Instruments, Inc.
Presented at the TI-MIX 1982 National Symposium.

ABSTRACT

Version IV.0 of the UCSD p-System supports concurrent processes with the use of the SIGNAL and WAIT primitive operations on semaphores. By use of the ATTACH primitive, semaphores may also be signalled by asynchronous interrupts.

This allows for application programs written in Pascal to respond to real-time external events. A sample program which does terminal emulation with file transmit and receive functions will be presented, along with a program which illustrates control of hardware functions of bit-oriented devices from a high-level language.

The UCSD p-System provides a convenient basis for software development on a large variety of microcomputers. Highly interactive, user-friendly programs may be constructed to fill a variety of single-user, personal workstation computing needs. However, many useful applications of microcomputers involve response to real-time events, and asynchronous control of devices which do not fall into the usual categories of printers, consoles, or character-oriented devices. The control of bit-oriented and analog devices by programs running under the UCSD p-System has been difficult or impossible until the release of Version IV.0.

Even without entering the realm of real-time machine control, a very common problem facing the programmer of a p-System personal workstation has been the construction of a 'terminal emulator' program, to allow communications to a remote host computer through a modem or other serial communications link. Appendix 1 contains a listing of a UCSD Pascal program which solves this problem for the specific case of communicating with Telemail, an electronic mail service available on GTE Telenet. The program is not represented as being completely general-purpose and user-configurable, nor is it transportable to other p-System versions in object code form. However, modifications to the source text to fit other machines would be minor, provided the target implementation supports the attachment of external events to semaphore data structures.

For tutorial purposes, a much smaller and simplified version of the program is listed in Appendix 2. The basic algorithm is as follows:

```
repeat
    {if a key is struck on the keyboard, send the character out to the modem}
    {if a character is received from the modem, display the character on the screen}
until finished;
```

A first attempt to code this in Pascal might look something like this:

```
repeat
    read(keyboard,ch);
    write(remote,ch);
    read(remote,ch);
    write(output,ch);
until finished;
```

. . .which is quickly found to be of little use in practical time-sharing applications, since only one character from the host can be displayed for every character from the keyboard.

A solution which works on Version II.1 uses the Unit_Status system intrinsic to find out how many characters are in the input queue for the console and remote units:

```
repeat
  Unit_Status(1,digits,1);
  if digits[0] <> 0
  then
        begin
              for i := 1 to digits[0] do
              begin
                    read(keyboard,ch);
                    write(remote,ch)
              end;
        end;

  Unit_Status(7,digits,1);
  if digits[0] <> 0
  then
        begin
              for i := 1 to digits[0] do
              begin
                    read(remote,ch);
                    write(output,ch)
              end;
        end;

until finished;
```

While this solution is adequate as presented, the overhead of the polling technique, combined with the desire to have more functions supported, make it difficult to keep up with a continuous 300 baud data stream. Practical terminal emulators with features such as file transmission and printer hardcopy became very large and complex, resorting to obscure Pascal coding tricks and to assembly-language external procedures in an attempt to sustain a 300 or 1200 baud data rate.

Version IV.0, as currently implemented on all TI 990 computers which support interrupts from character-oriented p-System devices, allows a 'full-function' terminal emulator program to be written with comparative ease, since language constructs are available which match the problem to be solved. This makes the main body of the terminal emulator appear effectively as:

```
repeat
    {Nothing}
until finished;
```

. . .since all the work is done by processes which are activated in response to external events.

These language extensions include a new data type (SEMAPHORE), a new block type (PROCESS), and five new pre-defined global procedures (ATTACH, SEMINIT, START, SIGNAL and WAIT). All except ATTACH are intended to be machine-independent constructs, and can be used on systems which do not support external asynchronous events. The use of these language features is described in the Version IV.0 Users' Guide, however, since the ATTACH intrinsic is somewhat machine-dependent, a description of the implementation on TI 990 computers is given here.

Just as a real processor needs to finish (or at least, bring to some orderly suspension point) the current machine instruction before handling an external interrupt, so must the p-machine. In practice, all of the p-machine interrupts which were generated by the real machine during the interpretive execution of the prior p-machine instruction are recognized before the next p-machine instruction is fetched. P-machine interrupts are selectively enabled by ATTACHing a SEMAPHORE to an interrupt level, and disabled by ATTACHing the value NIL to an interrupt level. The priority of the p-machine interrupt is determined by the priority of the PROCESS which is WAITing on the SEMAPHORE. In the current 990 implementation, 32 p-machine interrupts are supported; 64 will be supported in the IV.1 release in an effort to standardize the use of ATTACH on all p-System processors.

It is important to note the difference between a p-machine interrupt and an interrupt generated by hardware and handled by the real processor. For example, the 990/10 receives a hardware interrupt on level 5 every 1/120th of a second. The machine code which handles this interrupt maintains a 32-bit integer which represents the time of day, but only on the full 1-second intervals does it generate the p-machine interrupt level 16, and then only if level 16 has been enabled by having had a SEMAPHORE ATTACHed to it. As a consequence, the DS990 Model 1 can run exactly the same Pascal PROCESS for handling the p-machine level 16 interrupt, even though the real-time clock on the Model 1 generates hardware level 2 interrupt every 250 milliseconds. These differences are handled in the machine code which services the interrupt, so that the Pascal programmer sees a consistent p-machine interrupt structure.

Another advantage to this de-coupling of hardware and software is that several p-machine interrupts can be generated by the same hardware interrupt under different circumstances, for example, a 0.1 second and a 10 second p-machine interrupt. Or more than one hardware interrupt could signal the same p-machine interrupt. Obviously, every hardware interrupt must have some machine code in place to handle the interrupt, but the amount of processing required in machine code could be as simple as acknowledgement of the interrupt bit and a generation of the appropriate p-machine interrupt. External (machine code) procedures are easily written which can be called from a Pascal program to effect CRU I/O and TILINE I/O.

Appendix 3 describes the requirements of a software driver which takes the output of a text formatter program and controls a letter-quality printer. In this case, the printer has a 24-bit parallel interface. The encoding of the print-stream information was designed with an assembly-language driver program in mind, but the Pascal solution given was much easier to write, and could be transported with a minimum of trouble. Further, the same Pascal source can be easily modified to interface a variety of different output devices to the same print-stream protocol, including the H-P 7220A plotter which produced the overhead projection texts.

The preparation and production of this paper on a personal workstation computer running the UCSD p-System is a (somewhat needless) demonstration of the usefulness of the p-System in an office environment. Likewise, the fact that the entire p-System can be maintained and re-generated on the same machine is no surprise to system software developers who have used the p-System. However, the introduction of several new factors in Version IV.0 now makes possible the application of the p-System to a large subset of the problems which previously required assembly-language operating systems for real-time process control. Use of ATTACHed Pascal PROCESSes, coupled with machine-code interrupt handlers, external I/O primitive routines, and the use of the Native Code Generator to produce machine code from Pascal p-code, will allow many real-time applications to be easily implemented in a high-level, transportable language and Operating System.

Appendix 1

```
{-----------------------------------------------+
| Terminal emulator program for use with        |
| GTE Telenet and the Telemail service.         |
|                                               |
| Uses interrupts from the keyboard, remote,    |
| and 1-second interval timer, ATTACHed to      |
| SEMAPHOREs which are WAITed on by several     |
| PROCESSes.                                    |
|                                               |
| Fritz Whittington, Texas Instruments, Inc.    |
+-----------------------------------------------}

program telemail;

const
    clocksig = 16;
      keysig = 17;
      remsig = 18;

var
           ptr: integer;
          tick,
        godump,
       waiting,
      keyready,
      remready,
       eolecho,
         reply: semaphore;
           pid: processid;
             p: packed record
                  case integer of
                      1: (      a: packed array
                                      [0..80] of char);
                      2: (      s: string[80]);
                      3: (      k: integer);
                      4: (      x: integer;
                           online: boolean);
                  end;
         keych,
         remch,
           ret: packed array [0..1] of char;
            ch: char;
      dumping,
     quitting: boolean;
             i,
             j: integer;
         fname,
          mail,
      username,
      password,
       altuser,
       altpass,
      termtype,
        logoff: string;
            ff: file;              {the log file}
            gg: interactive;       {the file to transmit}
          buff: packed array [0..1023] of char;


process busy;

{-----------------------------------------------+
|  The OS does not tolerate a user program      |
|  that has every process totally blocked.      |
|  This process has lower priority than the     |
|  main program, and cannot be blocked itself,  |
|  since it doesn't wait on anything.  If it    |
|  is scheduled, it will be pre-empted sooner   |
|  or later by another task of higher priority. |
+-----------------------------------------------}

   begin
      while not quitting do {waste time};
      signal(reply);
   end   {busy};


process clocker;

{-----------------------------------------------+
|  This process has the highest priority, and   |
|  ensures that there will be an opportunity    |
|  for task switching at one-second intervals.  |
|  The signal(reply) ensures that the program   |
|  doesn't become deadlocked due to a lost      |
|  character echo.                              |
+-----------------------------------------------}

   begin
      while not quitting do
         begin
            wait(tick);
            signal(reply);
         end;
      attach(nil,clocksig);
   end   {clocker};


process dumper;

{-----------------------------------------------+
|  This process lies dormant until a file is    |
|  opened for transmission by procedure dof5.   |
|  It then transmits the text a line at a time, |
|  putting a local copy on the console, and waits|
|  for the echo of the carriage return.  (This  |
|  delay avoids overrunning the Telenet input   |
|  buffers, which were set up with human typists |
|  in mind.)  A possible deadlock could occur if |
|  the echo is lost, but can be cured by entering|
|  a carriage return from the keyboard.  During |
|  this process, the getrem process does not put |
|  echoed chars to the screen or log file.      |
+-----------------------------------------------}


   var
                s: string[255];

   begin
      repeat
         wait(godump);
         seminit(eolecho,0);
         if dumping then
            begin
               while not eof(gg) do
                  begin
                     readln(gg,s);
                     writeln(s);
{$R-}
                     moveright(s[1],s[2],length(s));
                     unitwrite(8,s[2],length(s),,
                        16396);
{$R+}
                     unitwrite(8,ret,1,,16396);
                     wait(eolecho);
                  end;
               close(gg,lock);
               writeln('~~~Closing File: ',fname,
                        '~~~');
               writeln;
               dumping := false;
            end;
      until quitting;
   end   {dumper};


procedure dof1;

   begin
      unitwrite(8,mail[2],(length(mail) - 1),,16396);
      unitwrite(8,ret,1,,16396);
   end   {dof1};


procedure dof2;

   begin
   end   {dof2};


procedure dof3;

   begin
   end   {dof3};


procedure dof4;

   begin
   end   {dof4};

procedure dof5;

   begin
      attach(nil,keysig);
      {-----------------------------------------------+
      |  We need to temporarily de-attach the keyboard |
      |  signal, in order to do an ordinary readln of  |
      |  the file name.                                |
      +-----------------------------------------------}
      writeln;
      j := 99;
{$I-}
      repeat
         write('Enter name of text file ',
               'to transmit, or <esc> <ret>  -->');
         readln(fname);
         if fname[1] <> chr(27)
         then
            begin
               fname := concat(fname,'.text');
               reset(gg,fname);
               j := ioresult;
               if j <> 0 then
                  writeln('No such file');
               close(gg);
            end
         else
            j := 0;
      until j = 0;
{$I+}
      if fname[1] <> chr(27) then
         begin
            reset(gg,fname);
            writeln('~~~Transmitting File: ',fname,
                     '~~~');
            dumping := true;
            signal(godump);
         end;
      {-----------------------------------------------+
      |  Now, we can re-attach the semaphore to the   |
      |  keyboard interrupt so that the getkey process |
      |  can handle characters.                       |
      +-----------------------------------------------}
```

```
      attach(keyready,keysig);
   end   {dof5};


procedure dof6;

   begin
      unitwrite(8,altuser[2],(length(altuser)-1),,16396);
      unitwrite(8,ret,1,,16396);
      for j := 1 to 2000 do {waste time};
      unitwrite(8,altpass[2],(length(altpass)-1),,16396);
      unitwrite(8,ret,1,,16396);
   end   {dof6};

procedure dof7;

   begin
      unitwrite(8,username[2],(length(username)-1),,16396);
      unitwrite(8,ret,1,,16396);
      for j := 1 to 2000 do {waste time};
      unitwrite(8,password[2],(length(password)-1),,16396);
      unitwrite(8,ret,1,,16396);
   end   {dof7};


procedure dof8;

   begin
      unitwrite(8,logoff[2],(length(logoff)-1),,16396);
      unitwrite(8,ret,1,,16396);
      quitting := true;
   end   {dof8};


process getkey;

{-----------------------------------------------+
|This process is activated by a character being |
|placed in the keyboard queue.  The specialkey  |
|function returns TRUE if the key is one of the |
|'F1' thru 'F8' keys on the 911 VDT, and trans- |
|lates the keys to ASCII '1' thru '8'.          |
+-----------------------------------------------}


   function specialkey: boolean;

      begin {specialkey}
         specialkey := false;
         if ord(ch) in [146..153] then
            begin
               ch := chr(ord(ch) - 97);
               specialkey := true;
            end;
      end   {specialkey};


   begin {getkey}
      repeat
         wait(keyready);
         if not quitting
         then
            begin
               {get from keyboard}
               unitread(2,keych,1,,12);
               ch := keych[0];
               if specialkey
               then
                  begin
                     case ch of
                        '1':  dof1;
                        '2':  dof2;
                        '3':  dof3;
                        '4':  dof4;
                        '5':  dof5;
                        '6':  dof6;
                        '7':  dof7;
                        '8':  dof8;
                     end {CASE}
                  end
               else {not specialkey}
                  begin
                     {put to remote}
                     unitwrite(8,keych,1,,16396);
                  end
            end
         else {if quitting}
            begin
               attach(nil,keysig);
            end;
      until quitting;
   end   {getkey};


process getrem;
```

```
{-----------------------------------------------+
|This process is activated by a character being |
|placed in the remote queue.  The echoed chars  |
|are normally placed in the logfile, except for |
|control chars, and during file transmission,   |
|since the echo is unreliable from Telenet, and |
|another copy of the text is not needed.        |
+-----------------------------------------------}


   var
         iord: integer;

   begin
      repeat
         wait(remready);
         if not quitting
         then
            begin
               {get from remin}
               unitread(7,remch,1,,16384);
               ch := remch[0];
               iord := ord(ch);
               if iord <> 10 then
                  begin
                     if not dumping then
                        begin
                           {put to screen}
                           unitwrite(1,remch,1);
                           signal(reply);
                        end;
                  end;
               if dumping and (iord = 13) then
                  signal(eolecho);
               if not dumping

               then
                  if (iord = 13)
                     or ((iord > 31) and (iord < 127))
                  then
                     begin
                        buff[ptr] := ch;
                        ptr := ptr + 1;
                        if ptr = 1024 then
                           begin
                              ptr := blockwrite(ff,
                                 buff,2);
                              ptr := 0;
                              fillchar(buff,1024,chr(0));
                           end;
                     end;
            end
         else {quitting}
            begin
               attach(nil,remsig);
            end;
      until quitting;
   end   {getrem};

procedure initialize;

{-----------------------------------------------+
|Initializes all the strings, and opens the     |
|next available logfile on the default volume.  |
|Unitclears the remote and console units.       |
+-----------------------------------------------}


   begin
      ret[0] := chr(13);
      {The four strings which follow need to be changed
       to the appropriate values for each user.  Note
       the unused ':' at the beginning, which is used to
       force word alignment for those PMEs that are
       sensitive to word alignment on unitwrites.}
      username := ':xxxxxxxx';
      password := ':xxxxxxxx';
       altuser := ':xxxxxxxx';
       altpass := ':xxxxxxxx';
        logoff := ':bye';
          mail := ':Mail';
      termtype := ':D1';
       dumping := false;
      quitting := false;
{$I-}
      i := 0;
      j := 0;
      repeat
         i := i + 1;
         p.s := '.x';
         p.s[2] := chr(i + 64);
         fname := concat('termlog',p.s,'.text');
         reset(ff,fname);
         j := ioresult;
         if j = 0
         then
```

```
            close(ff,lock)
         else
            close(ff);
      until j <> 0;
{$I+}
      rewrite(ff,fname);
      fillchar(buff,1024,chr(0));
      ptr := blockwrite(ff,buff,2);
      ptr := 0;
      unitclear(1);
      unitclear(7);
      unitclear(8);
   end  {initialize};


begin {main}

   initialize;

   write(chr(12));
   writeln('Waiting for connection...');
   writeln('214\748-0127 (300)...214\748-6371 (1200)');

   repeat
      p.online := false;
      unitstatus(7,p.a,1);
   until p.online;

   unitclear(7);
   writeln('Use F1 for ''Mail'' command');
   writeln('Use F5 to select transmit file');
   writeln('Use F6 for Alternate user');
   writeln('Use F7 for Username/password');
   writeln('Use F8 for Quitting');
   unitclear(1);

   seminit(remready,0);
   seminit(keyready,0);
   seminit(  godump,0);
   seminit(   reply,0);
   seminit(    tick,0);

   attach(    tick,clocksig);
   attach(remready,remsig);
   attach(keyready,keysig);

   start( getkey,pid,800,142);
   start( getrem,pid,500,140);
   start( dumper,pid,500,129);
   start(clocker,pid,500,145);
   start(   busy,pid,200,120);

   { Get Telenet's attention by sending 2 carriage returns
     at human speed, then define the video terminal}

   for i := 1 to 1000 do {waste time};
   unitwrite(8,ret,1,,16396);
   for i := 1 to 1000 do {waste time};
   unitwrite(8,ret,1,,16396);

   for i := 1 to 1000 do {waste time};
   unitwrite(8,termtype[2],2,,16396);
   unitwrite(8,ret,1,,16396);

   repeat
      wait(reply);
   until quitting;

   if ptr <> 0 then
      ptr := blockwrite(ff,buff,2);
   close(ff,lock);

   signal(keyready);
   signal(remready);
   signal(godump);

   attach(nil,clocksig);
   attach(nil,keysig);
   attach(nil,remsig);

   for i := 1 to 3000 do {waste time};
   writeln;
   writeln('Returning to PascalSystem');

end  {telemail}.
```

Appendix 2

```
{-----------------------------------------------+
| Terminal emulator program tutorial.           |
| Provides basic TTY emulation only.            |
|                                               |
| Uses interrupts from the keyboard, remote,    |
| and 1-second interval timer, ATTACHed to      |
| SEMAPHOREs which are WAITed on by several     |
| PROCESSes.                                    |
|                                               |
| Fritz Whittington, Texas Instruments, Inc.    |
+-----------------------------------------------}

program tutorial;

const
    clocksig = 16;
      keysig = 17;
      remsig = 18;

var
         tick,
     keyready,
     remready: semaphore;
          pid: processid;
        keych,
        remch,
           ch: char;
     quitting: boolean;

process busy;

{-----------------------------------------------+
| The OS does not tolerate a user program       |
| that has every process totally blocked.       |
| This process has lower priority than the      |
| main program, and cannot be blocked itself,   |
| since it doesn't wait on anything.  If it     |
| is scheduled, it will be pre-empted sooner    |
| or later by another task of higher priority.  |
+-----------------------------------------------}

   begin
      while not quitting do {waste time};
   end  {busy};


process clocker;

{-----------------------------------------------+
| This process has the highest priority, and    |
| ensures that there will be an opportunity     |
| for task switching at one-second intervals.   |
+-----------------------------------------------}

   begin
      while not quitting do
         wait(tick);
   end  {clocker};

process getkey;

{-----------------------------------------------+
|This process is activated by a character being |
|placed in the keyboard queue.                  |
+-----------------------------------------------}

   begin {getkey}
      repeat
         wait(keyready);
         {get from keyboard}
         unitread(2,keych,1,,12);
         {put to remote}
         unitwrite(8,keych,1,,16396);
      until quitting;
   end  {getkey};

process getrem;

{-----------------------------------------------+
|This process is activated by a character being |
|placed in the remote queue.                    |
+-----------------------------------------------}

   begin
      repeat
         wait(remready);
         {get from remin}
         unitread(7,remch,1,,16384);
         {put to screen}
         unitwrite(1,remch,1);
      until quitting;
   end  {getrem};

procedure initialize;
```

```
{--------------------------------------------+
|Unitclears the remote and console units.     |
+--------------------------------------------}

   begin
      quitting := false;
      unitclear(1);
      unitclear(7);
      unitclear(8);
   end   {initialize};

begin {main}

   initialize;


   seminit(remready,0);
   seminit(keyready,0);
   seminit(    tick,0);

   attach(    tick,clocksig);
   attach(remready,remsig);
   attach(keyready,keysig);

   start( getkey,pid,800,142);
   start( getrem,pid,500,140);
   start(clocker,pid,500,145);
   start(   busy,pid,200,120);

   repeat
      wait(tick);
   until quitting;

{Note that there is no provision for quitting, and that
 the (higher priority) process clocker is also waiting
 on tick.}

end   {tutorial}.
```

T2WPUNIT Interface Description

The text formatter program generates a print-stream which may be directed either to a file or to a printer. If the print-stream is directed to a file, it may subsequently be printed with the de-spooling program. Both of these programs interface with the physical printer by means of a unit T2WPUNIT which contains a signal public procedure WPINTERFACE(ch: char) to which the print-stream is sent byte-by-byte.

```
UNIT T2WPUNIT;
INTERFACE
        procedure wpinterface(ch: char);
IMPLEMENTATION
        var
                first_time: boolean;
        procedure wpinterface;
        begin
        if first_time then
                begin
                first_time := false;
                {here initialize printer}
                end;
        {here process print-stream byte ch}
        end {wpinterface};
BEGIN {T2WPUNIT}
first_time := true;
***;
END {T2WPUNIT}.
```

For documents printed in fixed-pitch mode, the print-stream is a simple ASCII stream. Each line of print is sent from left to right and terminated by a CR. The CR is usually followed by one of more LF's according to the amount of paper movement required between lines. However, where overprinting is required (e.g. for underlining), the CR is not followed by a LF. For fixed-pitch printers which are not capable of overprinting (because they have no non-advancing end-of-line function), the WPINTERFACE driver can throw away all characters after a CR until a LF is encountered. When single-forms mode is selected, the print-stream contains an ASCII group-separator character GS to signify that the driver should pause for a new sheet of paper to be positioned (or possibly should activate an automatic sheet feeder).

For documents printed in variable-pitch mode, the print stream contains tightly encoded information on horizontal printhead movement, vertical paper movement, print-element character selection, hammer energy, special print effects, and single-forms control. Such variable-pitch information is always introduced in the print-stream by the ASCII sequence NUL-NUL-SO to indicate to the WPRINTERFACE driver that variable-pitch mode is required. The driver must decode the print-stream and produce the requested printer actions. The following page defines the contents of the print-stream when in variable-pitch mode.

Appendix 3

Variable-Pitch Print-Stream Description

Drivers are instructed to enter variable-pitch mode by the sequence NUL-NUL-SO, and to leave variable-pitch mode by the sequence NUL-NUL-SI. Once in variable-pitch mode, the driver must respond to the following character sequences:

Type 1: (2 bytes)	1rrr rrrr	eeem mmmm
Type 2: (2 bytes)	01dd mmmm	mmmm mmmm
Type 3: (1 byte)	00cc cccc	

where:

rrrrrrr	= 7-bit character (code or rotation)
eee	= 3-bit hammer energy (zero if n/a)
m...m	= 5-bit or 12-bit movement distance:

in 1/120-inch increments for horizontal movement
in 1/48-inch increments for vertical movement

dd = 2-bit movement direction:

00 = right (forward tabulation)
01 = left (reverse tabulation)
10 = down (forward line feed)
11 = up (reverse line feed)

ccccc = 5-bit special action code:

NUL = no action
GS = pause after single form
BS = set direction = backward
HT = set direction = forward
CR = home printhead, set direction = forward
ESC = next Type-1 sequence defines underline
SI = return to fixed-pitch mode

Any Type-3 byte value from 30 to 3F hexadecimal sets special-effects:

byte = 0011 xdbu where:

u = underscore mode
b = boldface mode
d = double-strike mode
x = (not defined yet)

Type-1 sequences cause the print-head to move mmmmm 120ths of an inch to the left or right (whichever was set by the last Type-2 or Type-3 sequence), and then the character rrrrrrr to be printed with hammer energy eee.

```
{$N+}
UNIT T2WPUNIT;
{UNIT for the NEC Spinwriter with parallel card
 and using ABSOLUTE SPOKE mode}
INTERFACE
  PROCEDURE wpinterface(ch: char);
IMPLEMENTATION
const
wpcru    = 64;    {base address of interface card}
   {cru output bits}
   restore  = 12;    {sbo to restore, sbz to run}
   select   = 16;    {sbo to select}
   riblft   = 20;    {sbo for lower ribbon part (black)}
   halfsp   = 11;    {lsb of horizontal movement}
   waybit   = 10;    {sbz = right or down (head wrt paper)}
   lpback   = 13;    {sbo to read bits back for test}
   pwstb    = 18;    {print wheel strobe}
   pfstb    = 17;    {paper feed strobe}
   carstb   = 19;    {carriage strobe}
   {cru input bits}
   papout   = 22;    {true if paper out}
   ribout   = 21;    {true if ribbon out}
   pcheck   = 20;    {true if printer in check}
   pready   = 16;    {true if printer ready}
   pfready  = 17;    {true if paper feed ready}
   pwready  = 18;    {true if print wheel ready}
   carready = 19;    {true if carriage ready}
type
   onebit      =  0..1;
   twobits     =  0..3;
   threebits   =  0..7;
   nibble      =  0..15;
   fivebits    =  0..31;
   sixbits     =  0..63;
   sevenbits   =  0..127;
   ubyte       =  0..255;
   twelvebits  =  0..4095;
   bits        =  packed array [0..15] of boolean;
   typekind    =  (type3,type2,type1a,type1b);
   waykind     =  (right,left,down,up);  {direction of head wrt paper}
   t3kind      =  (action,s_effects);
   actionkind  =  (prefix,ud1,ud2,ud3,uv4,ud5,ud6,ud7,
                   setback,setnorm,ud10,ud11,ud12,home,entervp,exitvp,
                   ud16,ud17,ud18,ud19,ud20,ud21,ud22,ud23,
                   ud24,ud25,ud26,setus,ud28,holdit,ud30,ud31);
   word0       =  packed record          {for byte-swapping}
                  r_byte:       ubyte;
                  l_byte:       ubyte;
                  end;
   word1       =  packed record          {for TYPE 1 commands}
                  movement:     fivebits;
                  energy:       threebits;
                  rotation:     sevenbits;
                  istype1:      boolean;
                  end;
   word2       =  packed record          {for TYPE 2 commands}
                  movement:     twelvebits;
                  direction:    waykind;
                  whichtype:    typekind;
                  end;
   word3       =  packed record          {for TYPE 3 commands- actions}
                  res1:         ubyte;
                  t3action:     actionkind;
                  whicht3:      t3kind;
                  whichtype:    typekind;
                  end;
   word4       =  packed record          {for TYPE 3 commands- effects}
                  res1:         ubyte;
                  us_on:        boolean;
                  bf_on:        boolean;
                  ds_on:        boolean;
                  xx_on:        boolean;  {not currently used}
                  res2:         onebit;
                  whicht3:      t3kind;
                  whichtype:    typekind;
                  end;
  {word5 is just an integer and is denoted by i5}
   word6       =  packed record  {for 1355 Diablo in Rib. Opt. 2}
                  spoke:        sevenbits;
                  ribbon:       threebits;
                  energy:       twobits;
                  res1:         nibble;
                  end;
  {word7 is a packed array of bits}
   word8       =  packed record  {for NEC Spinwriter in Absolute mode}
                  spoke:        sevenbits;
                  abs:          boolean;
                  energy:       threebits;
                  res1:         fivebits;
                  end;
    urec       =  packed record
                  case integer of
                  0:  (w0:  word0);
                  1:  (w1:  word1);
                  2:  (w2:  word2);
                  3:  (w3:  word3);
                  4:  (w4:  word4);
                  5:  (i5:  integer);
                  6:  (w6:  word6);
                  7:  (bit: bits);
                  8:  (w8:  word8);
                  end;
var
inrec,outrec,usrec,cruword: urec;
intype1,intype2,inusdef,fixedpitch: boolean;
vdir,hdir: waykind;
bold,dubstrike,underlining: boolean;
vrtpos,horpos,nulcount: integer;
first_time: boolean;
PROCEDURE CPUIDLE; EXTERNAL;
PROCEDURE SETBASE(CRUBASE:INTEGER); EXTERNAL;
PROCEDURE CLEARBIT(BITNUM:INTEGER); EXTERNAL;
PROCEDURE SETBIT(BITNUM:INTEGER); EXTERNAL;
PROCEDURE LOADCRU(DATA,NUMBITS:INTEGER); EXTERNAL;
PROCEDURE STORECRU(VAR DATA:INTEGER; NUMBITS:INTEGER); EXTERNAL;
FUNCTION TESTBIT(BITNUM:INTEGER): BOOLEAN; EXTERNAL;
PROCEDURE wpinterface; (* PUBLIC PROCEDURE, PARAMETERS DECLARED ABOVE *)
procedure check_ready;
var
   ch: char;
   acked: boolean;
   pa:    packed array [0..10] of integer;
begin  {check_ready}
   cpuidle;          {allow cpu to idle until interrupt}
   if    (  (not testbit(pready))
                or
            (testbit(papout))
                or
            (testbit(ribout))
                or
            (testbit(pcheck))
   then
   begin
      unitclear(1);
      acked := false;
      while not acked
      do
      begin
         write(chr(13+128));    {Return without auto linefeed}
         write('Printer needs attention- press <space>',chr(7));
         cpuidle;
         pa[0] := 0;
         unitstatus(1,pa,1);   {get number of keys buffered for input}
         if pa[0] <> 0 then read(keyboard,ch);
         if ch = ' ' then acked := true;
      end;
      unitclear(1);
      gotoxy(0,100);
      writeln('Correct printer condition (paper, ribbon, cover)');
      writeln;
      writeln('Printing will resume when <enter> is pressed');
      repeat read(keyboard,ch) until ch = chr(160);
   end;
end;   {check_ready}
procedure hardinit;
var
   i: integer;
begin  {hardinit}
   cruword.i5 := 0;
   loadcru(cruword.i5,16);     {clear all data lines}
   setbit(select);             {select the printer}
   setbit(restore);            {tell it to restore}
   clearbit(pfstb);            {deactivate strobe line}
   clearbit(pwstb);            {deactivate strobe line}
   clearbit(carstb);           {deactivate strobe line}
   setbit(riblft);             {raise the ribbon}
   for i := 1 to 500 do;       {waste some time}
   clearbit(restore);          {let go of restore line}
   i := 1;
   while (i < 3000) and (testbit(pready) = false)
   do i := i + 1;
   if not testbit(pready)
   then
   begin
      writeln('Cannot open printer');
   end;
   check_ready;
end;   {hardinit}
procedure print_it(k: urec);
var
   timeout,i: integer;
begin  {print_it}
   check_ready;
   cruword.i5 := 0;
   cruword.w8.spoke := k.w1.rotation;
   cruword.w8.energy := k.w1.energy;
   cruword.w8.abs := true ;
   loadcru(cruword.i5,12);
   if testbit(papout) then check_ready;
   timeout := 1;
   while (timeout < 30000)
           and
          (not testbit(pwready))
   do
   begin
      check_ready;
      timeout := timeout + 1;
   end;
   setbit(pwstb);
   clearbit(pwstb);
end;   {print_it}
procedure move_carriage(howfar: integer; whichway: waykind);
var
   timeout,tempmove,bigmove,lsb,lead: integer;
begin  {move_carriage}
   if whichway = right then horpos := horpos + howfar
                       else horpos := horpos - howfar;
   check_ready;
   if not underlining
   then
   begin
      while howfar > 0
      do
      begin
         if howfar > 1023 then tempmove := 1023 else tempmove := howfar;
         howfar := howfar - 1023;
         bigmove := tempmove div 2;
         lsb := tempmove mod 2;
         if lsb = 1 then setbit(halfsp) else clearbit(halfsp);
         case whichway of
            right:  clearbit(waybit);
             left:    setbit(waybit);
         end;
         loadcru(bigmove,10);
         timeout := 1;
         while (timeout < 30000)
                 and
                (not testbit(carready))
         do
         begin
            check_ready;
            timeout := timeout + 1;
         end;
         setbit(carstb);
         clearbit(carstb);
      end;   {while howfar > 0}
   end    {of then clause on "if not underlining" }
   else   {if underlining}
   begin
      while howfar > 0
      do
      begin
         printit(usrec);
         case whichway of
            right:  clearbit(waybit);
             left:    setbit(waybit);
         end;
         bigmove := 0;      {that is 0/60ths}
         setbit(halfsp);    {+ 1/120th       }
         howfar := howfar - 1;
         loadcru(bigmove,10);
         timeout := 1;
         while (timeout < 30000)
                and
               (not testbit(carready))
         do
         begin
            check_ready;
            timeout := timeout + 1;
         end;
         setbit(carstb);
         clearbit(carstb);
      end;   {while howfar > 0}
   end;  {of the "if not underlining"}
end;   {move_carriage}
procedure move_paper(howfar: integer; whichway: waykind);
var
   timeout,tempmove: integer;
begin  {move_paper}
   if whichway = down then vrtpos := vrtpos + howfar
      else vrtpos := vrtpos - howfar;
   check_ready;
   setbit(halfsp);
   case whichway of       {DOWN means printhead down, paper UP}
      down:   clearbit(waybit);
      up:        setbit(waybit);
   end;
   while howfar > 0
   do
   begin
      if howfar > 1023 then tempmove := 1023 else tempmove := howfar;
      howfar := howfar - 1023;
      loadcru(tempmove,10);
      timeout := 1;
      while (timeout < 30000)
             and
            (not testbit(pfready))
      do
      begin
         check_ready;
         timeout := timeout + 1;
      end;
      setbit(pfstb);
      clearbit(pfstb);
   end;
   end;   {move_paper}
   procedure home_head;
   begin  {home_head}
      hdir := left;
      move_carriage(horpos,hdir);
      hdir := right;
      horpos := 0;
   end;   {home_head}
   procedure turn_vp;
   begin  {turn_vp}
      fixedpitch := false;
   end;   {turn_vp}
   procedure turn_fp;
   begin  {turn_fp}
      fixedpitch := true;
   end;   {turn_fp}
   procedure initialize;
   begin
      intype1 := false;
      intype2 := false;
      fixedpitch := true;
      nulcount := 0;
      inusdef := false;
      setbase(wpcru);
      vdir := down;
      hdir := right;
      usrec.i5 := 0;
      usrec.w1.istype1 := true;
      bold := false;
      dubstrike := false;
      underline := false;
      vrtpos := 0;
      horpos := 0;
      hardinit;
      turn_fp;
      home_head;
   end; {initialize}
   procedure ex_pause;
   var
      cg: char;
   begin  {ex_pause}
      writeln('Printing suspended by [single forms]');
      writeln('Printing will resume when <enter> is pressed');
      repeat read(keyboard,cg) until cg = chr(160);
   end;   {ex_pause}
   procedure ex_type1;
   var
      saveusf: boolean;
      bdir: waykind;
      howfar: integer;
   begin  {ex_type1}
      if inusdef
      then
      begin
         usrec.i5 := outrec.i5;
         inusdef := false;
         exit(ex_type1);
      end;
      howfar := outrec.w1.movement;
```

```
   move_carriage(howfar,hdir);
   print_it(outrec);
   if dubstrike and (not bold) then print_it(outrec);
   if bold
   then
   begin
      case hdir of
         right:  bdir := left;
         left:   bdir := right;
      end;
      saveusf := underlining;
      underlining := false;
      move_carriage(1,hdir);
      print_it(outrec);
      print_it(outrec);
      move_carriage(1,bdir);
      print_it(outrec);
      underlining := saveusf;
   end;
end;   {ex_type1}
procedure ex_type2;
var
   howfar: integer;
begin  {ex_type2}
   case outrec.w2.direction of
      right: hdir := right;
      left:  hdir := left;
      down:  vdir := down;
      up:    vdir := up;
   end;
   if (outrec.w2.direction = down) or (outrec.w2.direction = up)
   then
   begin
      howfar := outrec.w2.movement;
      move_paper(howfar,vdir);
   end
   else
   begin
      howfar := outrec.w2.movement;
      move_carriage(howfar,hdir);
   end;
end;   {ex_type2}
procedure ex_type3;
begin  {ex_type3}
   if nulcount = 2
   then
   begin
      nulcount := 0;
   end;
   if outrec.w4.whicht3 = action
   then
   begin
      case outrec.w3.t3action of
         prefix: nulcount := nulcount + 1;
         setback: hdir := left;
         setnorm: hdir := right;
         home:    home_head;
         entervp: turn_vp;
         exitvp:  turn_fp;
         setus:   inusdef := true;
         holdit:   ex_pause;
      end;  {case}
   end
   else
   begin
      if outrec.w4.bf_on then bold := true else bold := false;
      if outrec.w4.ds_on then dubstrike := true else dubstrike := false;
      if outrec.w4.us_on then underlining := true else underlining :=
false;
   end;
end;   {ex_type3}
procedure process_it;
var
   pa: packed array [0..1] of char;
begin  {process_it}
   if fixedpitch and (ch <> chr(0)) and (nulcount <> 2)
   then
   begin
      pa[0] := chr(ord(ch));
      unitwrite(6,pa,1,,12);
      exit(process_it);
   end;
   if intype1
   then
   begin
      outrec.w0.r_byte := ord(ch);
      intype1 := false;
      ex_type1;
      exit(process_it);
   end;
   if intype2
   then
   begin
      outrec.w0.r_byte := ord(ch);
      intype2 := false;
      ex_type2;
      exit(process_it);
   end;
   inrec.w0.l_byte := ord(ch);
   if inrec.w2.whichtype = type3
   then
   begin
      outrec.w0.l_byte := ord(ch);
      ex_type3;
      exit(process_it);
   end
   else
   begin
      if inrec.w1.istype1
      then
      begin
         outrec.w0.l_byte := ord(ch);
         intype1 := true
      end
      else
      begin
         outrec.w0.l_byte := ord(ch);
         intype2 := true;
```

```
       end;
    end;
end;     {process_it}
begin      {wpinterface}
   if first_time then
     begin
     first_time := false;
     initialize;
     end;
   process_it;
end;       {wpinterface}
BEGIN {T2WPUNIT}
first_time := true; {UNIT initialization}
***;
{no UNIT termination}
END {T2WPUNIT}.
;
; Machine code for the external functions
;
          .proc     machine
          limi      1                 ; To simulate the effect of a
          blwp      @8                ; Machine check (level 2 interrupt)
          b         *r11
          .proc     cpuidle
          idle
          b         *r11
          .proc     setbase,1
          .def      crubase,ws
;pascal declaration;
;  procedure setbase(i: integer); external;
;   where i is the crubase desired (decimal)
;
;990 instruction is:        li      r12,i
;
          mov       *r10+,crubase     ; Pop stack into private storage
          b         *r11              ; Return to pascal
crubase   .word     0
ws        .block    32                ; Workspace for the others
          .proc     setbit,1
          .ref      crubase,ws
;pascal declaration;
;  procedure setbit(i: integer); external;
;   where i is the bitnumber desired
;
;990 instruction is:        sbo     i
;
          mov       *r10+,ws+2        ; Pop stack into new r1
          blwp      start             ; Do the stuff
          b         *r11              ; Return to pascal
start     .word     ws,ep
inst      sbo       0                 ; Instruction mask
ep        mov       crubase,r12       ; Get current base from private storage
          mov       inst,r5           ; Op code for sbo
          andi      r1,00ffh          ; Insure parm passed is 00 in first byte
          soc       r1,r5             ; R5 now has proper sbo instruction
          x         r5                ; Execute it
          rtwp                        ; Done
          .proc     clearbit,1
          .ref      crubase,ws
;pascal declaration;
;  procedure clearbit(i: integer); external;
;   where i is the bitnumber desired
;
;990 instruction is:        sbz     i
;
          mov       *r10+,ws+2        ; Pop stack into new r1
          blwp      start             ; Do the stuff
          b         *r11              ; Return to pascal
start     .word     ws,ep
inst      sbz       0                 ; Instruction mask
ep        mov       crubase,r12       ; Get current base from private storage
          mov       inst,r5           ; Op code for sbz
          andi      r1,00ffh          ; Insure parm passed is 00 in first byte
          soc       r1,r5             ; R5 now has proper sbz instruction
          x         r5                ; Execute it
          rtwp                        ; Done
          .proc     loadcru,2
          .ref      crubase,ws
;pascal declaration;
;  procedure loadcru(i,j: integer); external;
;   where i is the 16-bit pattern desired, j is number of bits
;
;990 instruction is:        li    rx,i
;                           ldcr  rx,j
;
          mov       *r10+,ws+2        ; Pop # of bits into new r1
          mov       *r10+,ws+4        ; Pop data word into new r2
          blwp      start             ; Go do it
          b         *r11              ; Return to pascal
start     .word     ws,ep
inst      ldcr      r2,0              ; Instruction mask
ep        mov       crubase,r12       ; Get current base from private storage
          mov       inst,r5           ; Puts 'ldcr  r2,0' instruction in r5
          andi      r1,000fh          ; Mask # of bits to 0-15
          sla       r1,6              ; Shift to spot required in instruction
          soc       r1,r5             ; Or into r5
          x         r5                ; Execute it
          rtwp                        ; Done
          .proc     storecru,2
          .ref      crubase,ws
;pascal declaration;
;  procedure storecru(var i: integer; j: integer); external;
;   where i is the 16-bit result area, j is number of bits
;
;990 instruction is:        stcr  rx,j
;                           mov   rx,i
;
          mov       *r10+,ws+2        ; Pop # of bits into new r1
          mov       *r10+,ws+4        ; Pop address of data into new r2
          blwp      start             ; Go do it
          b         *r11              ; Return to pascal
start     .word     ws,ep
inst      stcr      *r2,0             ; Instruction mask
ep        mov       crubase,r12       ; Get current base from private storage
          mov       inst,r5           ; Puts 'stcr  *r2,0' instruction in r5
          andi      r1,000fh          ; Mask # of bits to 0-15
          sla       r1,6              ; Shift to spot required in instruction
          soc       r1,r5             ; Or into r5
          x         r5                ; Execute it
          rtwp                        ; Done
          .func     testbit,1
          .ref      crubase,ws
;pascal declaration;
;  function testbit(i: integer): boolean;  external;
;   where i is the bitnumber
;
;990 instruction is:        tb    i
;                           (returns true if bit is 1)
;
          mov       *r10,ws+2         ; Pop # of bits into new r1
          ai        r10,2             ; Point to word provided on stack
; Note- the stack pointer is left pointing to result word
          blwp      start             ; Go do it
          mov       ws,*r10           ; Push r0 of new ws
          b         *r11              ; Return to pascal
start     .word     ws,ep
inst      tb        0                 ; Instruction mask
ep        mov       crubase,r12       ; Get current base from private storage
          clr       r0                ; Assume bit is false
          andi      r1,00ffh          ; Insure displacement is in range
          mov       inst,r5           ; Get copy of instruction
          soc       r1,r5             ; Or in displacement
          x         r5                ; Execute test bit
          jne       $1                ; Jump if bit was false
          inc       r0                ; If bit was 1, r0 := 1
$1        rtwp                        ; All done
          .end
```

US

The Two Faces of UCSD Pascal

By Rich Gleaves
Volition Systems

Rich Gleaves of Volition Systems submits the slides from a talk of his. I (as usual) dropped out where and when the talk was. I have included the first few slides of his talk of which the outline was as follows:

THE TWO FACES OF UCSD PASCAL
UCSD PASCAL _ INDUSTRY IMPACT
UCSD PASCAL - HISTORY
UCSD PASCAL SYSTEM VERSIONS
UCSD PASCAL vs. STANDARD PASCAL
UCSD PASCAL EXTENSIONS
PROGRAM SEGMENTATION
SEPARATE COMPILATION - UNITS
UCSD I/O HIERARCHY
INTERACTIVE I/O
RANDOM ACCESS FILES
UNIT I/O
STRINGS
BYTE ARRAY MANIPULATION
DYNAMIC STORAGE
PROCEDURE TERMINATION
EVEN MORE TRICKS
RECORD AND ARRAY COMPARISON

If you would like to have the bodies for all these slides, please contact Volition Systems. ed.

THE TWO FACES OF UCSD PASCAL

- Friendly beginner's language

  Used at UCSD to teach introductory computer programming to non-science students. UCSD Pascal's