From: JBONDY Subj: Virtus. Segments Article

Since I wrote the S/W for Version I.5, I have discovered how to get it working on Version III.0 (and IV.0 is similar to III.0 in this regard).

On the Microengine version IV.0 there are no segment tables to load, so I played with the Segment Information Blocks (SIBs). These are described in the IV.0 manual on page 34, and the format for the Microengine is given below:

```
SIB = record
   Segbase : ^integer;
   Seglenq : integer;
   xxx     : integer;
   Segaddr : integer;
   Sequnit : integer;
   Segref  : integer;
   SeqSP   : integer;
 end;
```

There is a p-machine register on the Microengine which is a pointer to an "array[segrange] of ^SIB", and the value of this register may be loaded into the variable "ptr" with
PMACHINE(^ptr,(-2),LPR,STO)
where LPR = 157 and STO = 196. One can then mess with the SIB for segment 'I' with ptr^[I].fieldname, and this is how I made Virtual Segments work on the Microengine; I assume that making it work under IV.0 would be similar.

Here is the article

Virtual Segment Procedures under UCSD Pascal

by
Jon Bondy
Box 148, Ardmore, Pa, 19003

This article is copyrighted by Microsystems (Box 1192, Mountainside, NJ, 07092) and is reprinted with their permission.

One of the nice features of UCSD Pascal is its support of segment procedures. A segment procedure is like any other Pascal procedure except that whenever it is called (except for recursive calls) it is loaded from disk into memory prior to being executed, and after it exits, the memory is reclaimed. In fact, the segment procedure is loaded onto the stack, since the pattern of memory use is nested in a very 'stack-like' manner.

Segment procedures allow the programmer to manage memory resources explicitly and conveniently, and really are a form of overlay. In large programs, it is not uncommon to dedicate a segment procedure to initialization, since that code need not reside in memory after the program starts. Segment procedures may have internal procedures and functions, all of which are loaded with the segment procedure code, allowing functional groups of routines to be brought into memory in a single operation, and executed.

Unfortunately, UCSD Pascal allows the programmer access to only six segment procedures under normal circumstances. This is fine for small programs (under 3000 lines), but when one starts to get serious about an application, one really needs more segment procedures. One reason for this is that the UCSD Pascal separate compilation construct (UNITs) uses one of these six segment procedure 'slots' even if it is not loaded into memory dynamically, wasting this scarce resource. This situation should improve very soon (perhaps by the time this article is printed) because Softech is planning to announce features in UCSD Pascal Version IV.0 to solve some of these problems.

I work for a company (Energy Data Systems) which has been trying to do some fairly complex applications in UCSD Pascal, and we ran into the 'segment barrier' in the spring of this year. We considered modifying the UCSD operating system, since we had source for it, but decided to try to stick to solutions which required as few modifications to operating system code as possible. After some thought I came up with an interim means of ameliorating the problem, which I will describe in this article.

When a segment procedure is called by the UCSD Pascal p-machine, a special op-code is used to do so. This op-code first looks in an operating system 'segment table' to see if the indicated procedure has been called more than zero times (reference count, in case it is a recursive call and the code need not be reloaded). If it is already in memory (reference count > 0), it is executed like any other procedure; if not, the op-code looks in the segment table for the block number on the disk where it can find the code

Copyright 1981 by Microsystems

for the segment procedure, and the number of bytes in the code. It then loads the code onto the stack and calls the procedure, incrementing the reference count to one. Upon exit, the segment procedure return op-code decrements the reference count by one, and clears the stack back up to where it was prior to the call if the count has become zero.

The idea which I had was to somehow write data into the segment tables prior to each segment procedure call in such a way that when the above op-code was invoked it would find data describing different segment procedures each time; it would in fact be faked into loading different code segments into memory for each call, even though the same segment procedure was being called each time.

In order to do this, I first had to find where the segment tables were located in memory. The first global variable declared in the UCSD Pascal operating system code is a pointer to a special record, called the system communication record, or SYSCOMREC, and the segment tables are a part of this record. There is a special 'switch' in the UCSD Pascal compiler (the 'U' switch), and when its value is '-', programs function very differently than usual. The 'main program' does not execute at all, but rather the first segment procedure declared in the program executes instead. Also, the variable definitions are 'aliased' on top of the definitions for the operating system (like an EQUIVALENCE in FORTRAN), allowing the program to read and write those variables. Usually, one uses exactly the same variable definitions as were used when the operating system was compiled, in order that one's program agree with the operating system definitions. In this case, however, I simply made my own definitions, since all I wanted was to determine the value of the pointer to SYSCOMREC. Since I.5 stores pointers as actual memory addresses, by printing the value of the pointer I could determine where in memory SYSCOMREC was stored.

I wrote the following program and was able to locate SYSCOMREC. It was part trial and error, since after I thought I had 'found' SYSCOMREC the first time, I was forced to look through dumps to figure out why I had been wrong... Anyway, the program below will tell you where in memory any SYSCOMREC is under Version I.5 (it is at location 718 [decimal] for my Z-80 version of I.5), and probably under Version II.0; I have not tried it with III.0.

```
{$U-}
program find;
  var
    i : integer; { aliased to ^syscomrec }

  segment procedure findsyscom;
    begin
    writeln('Syscomrec is located at address ',i,' decimal.');
    end;

begin
end.
```

I then needed to figure out how far from the start of SYSCOMREC the segment tables started. Fortunately, the UCSD Pascal operating system variable definitions, found in a file called GLOBALS.TEXT, were distributed with UCSD Pascal Versions I.4 and I.5, so I had them at my disposal. [The

Copyright 1981 by Microsystems

GLOBALS.TEXT file, along with all other UCSD Pascal source code, is copyright by the University of California at San Diego; some of the GLOBALS.TEXT file is presented in this article (the portions of Pascal source in capital letters) with the permission of Softech Microsystems Inc, their licensee.] I am informed that the SYSCOMREC data layout has not changed with the various UCSD Pascal Versions, so you should be able to locate the segment tables at 96 bytes past the start of your SYSCOMREC. This means that if one were to write to location 814 in my memory, one would be writing on the first byte of the segment tables.

The segment tables are defined as follows:

```
type

  segdesc = record
    diskaddr : integer; { absolute block number on disk }
    codeleng : integer; { in bytes }
    end;

  segtab = array [segrange] of record
    unit : unitnum; { disk unit number (an integer) }
    codedesc : segdesc; { as above }
    end;
```

where 'segrange' is the number of segment procedures defined for the UCSD system being used ('0..15' in the case of Version I.5). If one declared a pointer 'segptr' which pointed to a record of type 'segtab', one could refer to that record as 'segptr^', to the unit (disk drive) on which the code for the third segment procedure was located as 'segptr^[3].unit', and to the number of bytes in the code for that procedure as

segptr^[3].codedesc.codeleng . One could establish the correct value in
that pointer with the following record definition:

```
var

  alias : record case boolean of
    true : (i : integer);
    false: (p : ^segtab);
    end;
```

This record definition states that one will either use the storage for the record 'alias' as an integer (denoted 'alias.i') or as a pointer to a variable of type 'segtab' (denoted 'alias.p'). Since the same storage is used for both values, if one were to write into the integer part, one could then use that value as a pointer; one could 'fake' Pascal into thinking that a variable of type 'segtype' was being pointed to.

By stating 'alias.i := 718 + 96', I could access the actual operating system segment table as 'alias.p^'. I could then write into the segment table entries of any segment which I chose, forcing the system to load the code I wanted to when I called the appropriate procedure.

Suppose, for instance, that I knew that I wanted to execute each of a series of ten segment procedures which were located on 'unit[i]', had length 'length[i]' bytes, and started at disk block number 'block[i]'. I could write a procedure to perform a 'call' to the 'i-th' such segment

Copyright 1981 by Microsystems

procedure as follows:

```
program test;

  type
    segdesc = record
      diskaddr : integer; { absolute block number on disk }
      codeleng : integer; { in bytes }
      end;
    segtab = array [segrange] of record
      unit : unitnum; { disk unit number }
      codedesc : segdesc; { as above }
      end;

  var
    alias : record case boolean of
      true : (i : integer);
      false: (p : ^segtab);
      end;
    unit : array[1..10] of integer;
    length : array[1..10] of integer;
    block : array[1..10] of integer;
    i : integer;

  segment procedure virtual;
    begin { need not have any code, since it will never execute --
            the other ten segment procedures will execute instead }
    end; { virtual }

  procedure dovirtual(i : integer);
```

```
    begin
    { set up to call virtual segment }
    alias.p^[10].codeunit := unit[i];
    alias.p^[10].codedesc.codeleng := length[i];
    alias.p^[10].codedesc.diskaddr := block[i];
    { call virtual segment loaded above }
    virtual;
    end; { dovirtual }

begin
{ set up pointer to real segment table }
alias.i := 718 + 96;
{ call the ten virtual procedures }
for i := 1 to 10 do dovirtual(i);
end. { test }
```

The above program will actually work, but it has a few problems which make it a bit awkward to use. First off, procedure calls which previously looked like a nice name now are reduced to a cryptic statement like 'dovirtual(3)'. This can be taken care of by creating constants at the start of the program with values from 1 to 10, and calling 'dovirtual' with those constant values; a procedure to clear the screen might then be called as 'dovirtual(clrscreen)', a significant improvement.

The other problem is that it is NOT EASY to find out some of the information which I so casually stated would be found in the 'length' and

Copyright 1981 by Microsystems

'block' arrays. To do so involves reading the 'segment table' of the code file in which one of the virtual segments exists, in order to determine that information. Unfortunately, the disk address information stored in the segment tables of a code file is slightly different than that stored in the operating system's segment tables. The normal code file disk addresses are relative to the start of the entire code file; the system addresses are the absolute disk block number. This means that in order to convert the data in the code file's segment table into 'useful' information, we must also know the absolute address of the start of the code file. And in order to determine this, we must read (and understand) the directory of the disk. Whew!...

Taking it a step at a time, the format of a UCSD Pascal disk directory is given below. It is a portion of the UCSD GLOBALS.TEXT file mentioned earlier (and is copyright by UCSD).


```
CONST
  MAXUNIT = 12;          (*MAXIMUM PHYSICAL UNIT # FOR UREAD*)
  MAXDIR = 77;           (*MAX NUMBER OF ENTRIES IN A DIRECTORY*)
  VIDLENG = 7;           (*NUMBER OF CHARS IN A VOLUME ID*)
  TIDLENG = 15;          (*NUMBER OF CHARS IN TITLE ID*)
  FBLKSIZE = 512;        (*STANDARD DISK BLOCK LENGTH*)
  DIRBLK = 2;            (*DISK ADDR OF DIRECTORY*)
  MAXSEG = 15;           (*MAX CODE SEGMENT NUMBER*)

TYPE

  DATEREC = PACKED RECORD
    MONTH: 0..12;                (*0 IMPLIES DATE NOT MEANINGFUL*)
    DAY: 0..31;                  (*DAY OF MONTH*)
    YEAR: 0..100                 (*100 IS TEMP DISK FLAG*)
    END (*DATEREC*) ;

  UNITNUM = 0..MAXUNIT;
  VID = STRING[VIDLENG];

  DIRRANGE = 0..MAXDIR;
  TID = STRING[TIDLENG];

  FILEKIND = (UNTYPEDFILE,XDSKFILE,CODEFILE,TEXTFILE,
    INFOFILE,DATAFILE,GRAFFILE,FOTOFILE,SECUREDIR);

  DIRENTRY = RECORD
    DFIRSTBLK: INTEGER;                          (*FIRST PHYSICAL DISK ADDR*)
    DLASTBLK: INTEGER;                           (*POINTS AT BLOCK FOLLOWING*)
    CASE DFKIND: FILEKIND OF
      SECUREDIR,
      UNTYPEDFILE:                               (*ONLY IN DIR[0]...VOLUME INFO*)
        (DVID: VID;                              (*NAME OF DISK VOLUME*)
         DEOVBLK: INTEGER;                       (*LASTBLK OF VOLUME*)
         DNUMFILES: DIRRANGE;                    (*NUM FILES IN DIR*)
         DLOADTIME: INTEGER;                     (*TIME OF LAST ACCESS*)
         DLASTBOOT: DATEREC);                    (*MOST RECENT DATE SETTING*)
      XDSKFILE,CODEFILE,TEXTFILE,INFOFILE,
      DATAFILE,GRAFFILE,FOTOFILE:
        (DTID: TID;                              (*TITLE OF FILE*)
         DLASTBYTE: 1..FBLKSIZE;                 (*NUM BYTES IN LAST BLOCK*)
         DACCESS: DATEREC)                       (*LAST MODIFICATION DATE*)
    END (*DIRENTRY*) ;

var
  directory : array[dirrange] of direntry;
```

Given the above definitions, after a bit of studying we can see that each directory entry always contains the disk address (in blocks) of the first and last blocks of the file for which it is an entry. If the entry is a volume ID entry, it also contains the volume (diskette) name, the number of blocks on the volume ('deovblk'), the number of files on the volume ('dnumfiles') and the time (date) when it was last accessed. Normally one finds this entry as the first entry in the directory (or 'directory[0]'). If the entry is a normal file entry, it contains the file ID (name), the number of bytes in the last block which really contain data, and the 'date

Copyright 1981 by Microsystems

of the last modification to the file. The above definitions also tell us where to find the directory, namely at 'dirblk', or block two.

We can read the directory into memory with the statement

```
unitread(unitnum,directory,sizeof(directory),dirblk);
```

This statement uses two UCSD intrinsics, the 'sizeof' function and the 'unitread' procedure. The former returns the number of bytes in a given data structure; in our case, it is the number of bytes in the 'directory'. The 'unitread' procedure reads from unit (disk drive) 'unitnum' into memory at the address of the 'directory' structure for 'sizeof(directory)' bytes starting at absolute disk block 'dirblk'

With this information in memory, we can search the directory for an entry for a given file. Suppose that we wanted to see if the file 'sample.code' were in the directory. We could say

```
numfiles := directory[0].dnumfiles; { number of valid entries }
i := 0; { will index into directory }
done := false; { will become 'true' when we are done }
while (i < numfiles) and not done do begin
  i := i + 1;
  if (directory[i].dtid = 'sample.code') then done := true;
  end;
```

If we come out of this loop with 'done' having a value of 'true', then we have indeed found an entry for the required file; if not, the file was not present. If it was found, then the absolute address of the first block of the file on disk is simply 'directory[i].dfirstblk'. We have then found the first thing we needed, the absolute disk address of the code file. We now need to get the code length and relative disk address information from the code file's segment tables.

The format of the first block of a code file is as shown below, where the previous type definitions hold as before.

```
segtable : array[segrange] of segdesc;
```

If we wanted to find the necessary information about a particular code file, we could use the UCSD Pascal system intrinsic procedure 'unitread' to read the segment table data from that file into the above data structure. Then, if we wanted the information about the tenth segment procedure in a code file, we could simply use the tenth element of that array. For example, to determine the 'length' and 'block' data for segment procedure number ten in a file called 'sample.code' on unit five, one could do the following:


```
program find;
  var
    codefile : file;
    segtable : array[segrange] of segdesc; { 'segdesc' defined as before }
    i : integer;
  begin
  unit := 5;
  { read in segment table -- assume 'directory' read in as above first,
    and 'directory[i]' is data for file which is of interest to us }
  unitread(unit, segtable, sizeof(segtable), directory[i].dfirstblk);
  length := segtable[10].codeleng;
  block := segtable[10].diskaddr; { relative block number }
  block := block + directory[i].dfirstblk; { absolute block number }
  end. { find }
```

So, we finally have the entire ball of wax. We can read a UCSD disk directory to find out where on disk a file is stored; we can read and understand the information in the segment table of a code file; we can convert the relative block numbers in the code file segment tables to absolute block numbers; and we can use that information to invoke virtual segment procedures. To tie everything together, you will find below a single program which does it all. The procedure 'virtinit initializes an internal table of unit, length, and block information before the program really gets going. It does this by reading in the disk directory and calling 'virtlink' for each virtual segment procedure which it will call later. The 'virtlink' procedure looks in the directory for the code file requested and puts the required information in the internal table. The main program then invokes the virtual segments as a test. Following this program is a sample of one of the programs which defines a virtual segment procedure. Note that in both the program which calls the virtual segments and the program which defines one of them, the virtual segments are defined as the first executable code in the file. This is necessary, since the technique which I have shown requires that both segment procedures have the same 'segment numbers'.

Copyright 1981 by Microsystems

```
program vsegtest;

CONST
  MAXUNIT = 12;          (*MAXIMUM PHYSICAL UNIT # FOR UREAD*)
  MAXDIR = 77;           (*MAX NUMBER OF ENTRIES IN A DIRECTORY*)
  VIDLENG = 7;           (*NUMBER OF CHARS IN A VOLUME ID*)
  TIDLENG = 15;          (*NUMBER OF CHARS IN TITLE ID*)
  FBLKSIZE = 512;        (*STANDARD DISK BLOCK LENGTH*)
  DIRBLK = 2;            (*DISK ADDR OF DIRECTORY*)
  MAXSEG = 15;           (*MAX CODE SEGMENT NUMBER*)

TYPE

  DATEREC = PACKED RECORD
    MONTH: 0..12;                (*0 IMPLIES DATE NOT MEANINGFUL*)
    DAY: 0..31;                  (*DAY OF MONTH*)
    YEAR: 0..100                 (*100 IS TEMP DISK FLAG*)
    END (*DATEREC*) ;

  UNITNUM = 0..MAXUNIT;
  VID = STRING[VIDLENG];

  DIRRANGE = 0..MAXDIR;
  TID = STRING[TIDLENG];

  FILEKIND = (UNTYPEDFILE,XDSKFILE,CODEFILE,TEXTFILE,
    INFOFILE,DATAFILE,GRAFFILE,FOTOFILE,SECUREDIR);

  DIRENTRY = RECORD
    DFIRSTBLK: INTEGER;                          (*FIRST PHYSICAL DISK ADDR*)
    DLASTBLK: INTEGER;                           (*POINTS AT BLOCK FOLLOWING*)
    CASE DFKIND: FILEKIND OF
      SECUREDIR,
      UNTYPEDFILE:                               (*ONLY IN DIR[0]...VOLUME INFO*)
        (DVID: VID;                              (*NAME OF DISK VOLUME*)
         DEOVBLK: INTEGER;                       (*LASTBLK OF VOLUME*)
         DNUMFILES: DIRRANGE;                    (*NUM FILES IN DIR*)
         DLOADTIME: INTEGER;                     (*TIME OF LAST ACCESS*)
         DLASTBOOT: DATEREC);                    (*MOST RECENT DATE SETTING*)
      XDSKFILE,CODEFILE,TEXTFILE,INFOFILE,
      DATAFILE,GRAFFILE,FOTOFILE:
        (DTID: TID;                              (*TITLE OF FILE*)
         DLASTBYTE: 1..FBLKSIZE;                 (*NUM BYTES IN LAST BLOCK*)
         DACCESS: DATEREC)                       (*LAST MODIFICATION DATE*)
    END (*DIRENTRY*) ;

  SEGRANGE = 0..MAXSEG;
  SEGDESC = RECORD
    DISKADDR: INTEGER;     (* REL BLK IN CODE...ABS IN SYSCOM*)
    CODELENG: INTEGER      (*# BYTES TO READ IN*)
    END; (*SEGDESC*)

  vsegrange = 0..15; { virtual segment number }

  { segment table in syscomrec and in this program }
  segtabtype = array [segrange] of record
```

Copyright 1981 by Microsystems


```
    codeunit : unitnum;
    codedesc : segdesc;
    end;

var
  alias : record case boolean of { allow manual setup of ^syscomrec }
    true : (i : integer);
    false: (p : ^segtabtype);
    end;
  disk : file; { file to read directory and seg tables from }
  { global in which to store processed seg table for later use }
  vsegs : segtabtype;

segment procedure virtual;
  begin writeln('I am the REAL segment procedure 10.');
  end; { virtual -- dummy }

procedure dovirtual(vsegnum : vsegrange);
  begin
  if (vsegs[vsegnum].codeunit = 0)  then begin
    writeln('Attempt to execute unlinked virtual seg number ',vsegnum,'.');
    exit(dovirtual); { not linked }
    end;
  { load segment register 10 with segment data from i'th segment procedure }
  alias.p^[10] := vsegs[vsegnum];
  virtual;
  end; { dovirtual }

procedure virtinit;
  { initialize tables for calling v segment procedures }
  const
    unum = 5; { unit number where v segment procs are found }
  var
    i : integer; { temp var }
    directory : array [dirrange] of direntry; { holds disk directory }
    numfiles : dirrange; { number of files in disk dir }

  procedure virtlink(fname : string; vsegnum : vsegrange);
    var
      i : integer;
      done : boolean;
      firstblk : integer; { blk number of first blk of fake seg code file }
      isegtable : array [segrange] of segdesc; { holds seg tbl from disk }
    begin
    { find file in directory }
    i := 0; done := false;
    while (i <= numfiles) and not done do begin
      i := i + 1;
      if (directory[i].dtid = fname) then done := true;
      end;
    if not done then begin
      writeln('unable to find file ',fname,'.');
      exit(virtlink);
      end;
    writeln('File ',fname,' found.');
```

Copyright 1981 by Microsystems

```
    { read in segment table }
    firstblk := directory[i].dfirstblk;
    unitread(unum,lseqtable,sizeof(lseqtable),firstblk);
    { enter data from segment 10 in found file into v seg table }
    vsegs[vsegnum].codeunit := unum;
    vsegs[vsegnum].codedesc.codeleng := lseqtable[10].codeleng;
    vsegs[vsegnum].codedesc.diskaddr := lseqtable[10].diskaddr + firstblk;
    writeln('Finished with association of file ',fname,'.');
    end; { virtlink }

  begin
  { initialize vseg table so that all units are zero (vseg undefined) }
  for i := 0 to 15 do vsegs[i].codeunit := 0;
  { set up pointer to syscomrec's seg table located by 'find' program }
  alias.i := 718 + 96; { 96 bytes in syscom rec before seg tables }
  { read directory into memory }
  unitread(unum,directory,sizeof(directory),dirblk,0);
  numfiles := directory[0].dnumfiles;
  writeln('Directory of unit ',unum,' read in.');
  { link file names with v seg numbers }
  virtlink('FAKE0.CODE',0);
  virtlink('FAKE1.CODE',1);
  virtlink('FAKE2.CODE',2);
  virtlink('FAKE3.CODE',3);
  virtlink('FAKE4.CODE',4);
  virtlink('FAKE5.CODE',5);
  virtlink('FAKE6.CODE',6);
  virtlink('FAKE7.CODE',7);
  end; { virtinit }

begin { main }
virtinit;
dovirtual(0);
dovirtual(1);
dovirtual(2);
dovirtual(3);
dovirtual(4);
dovirtual(5);
dovirtual(6);
dovirtual(7);
end.


program fake0;

  segment procedure fake0also;
    begin
    writeln('I think that I am virtual segment procedure 0.');
    end;

  begin { main }
  end.
```

They looked for me.

The wondered where I was,

    Was I operating a system,
    Was I processing a micro,
    Was I wearing hard;

They wondered where I was,

    Was I listing to one side,
    Was I seeking to the end,
    Was I tracking down a bug;

They wondered where I was,

    Was I terminally ill,
    Was I caught in a calm pile,
    Was I filing it away;

They wondered where I was,

    Was I designing a principle,
    Was I comparing a language,
    Was I analyzing an algorithim;

They wondered where I was,

    And they checked out all their sums,
    They ordered high and low,
    They searched down all their nodes;

And Still:

They wondered where I was,

    They looked in all their tables,
    They opened all their ports,
    They raised up all their flags,

        All their searching failed,
        Their scanning ran to end,
        I wasn't in their banks,
        I'd slipped off all their discs;

        They decided that I wasn't,
        They gave up all their hope,
        They dried out all their boots,
        And cleaned out all their bins;

        I wasn't marked upon their tapes,
        I was deleted in their files,
        I had no sectors on their disc,
        I wasn't on their heap;

        Free to smell the flower,
        Free to see the beach,
        Free with all the power,
        Of a wet-core security breach.

kasf1/20/781