BASIC SIG

THE NEWSLETTER FOR THE BASIC SPECIAL INTEREST GROUP

May 1982 Vol. 3 No. 2

BASIC-PLUS-2 DEVELOPMENT SUPPORT TEAM

1.0 BP2 TASK STRUCTURE

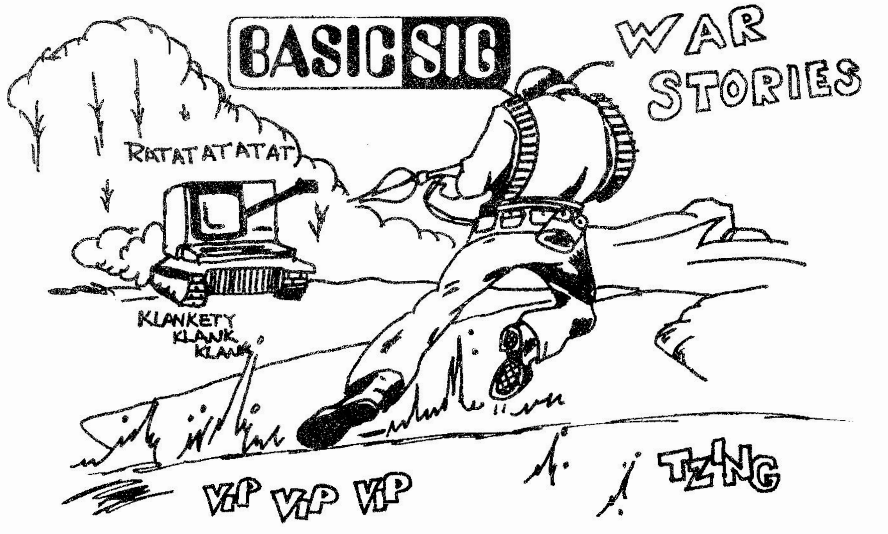
This article addresses the layout of a BP2 task in memory. Topics discussed include overall memory layout, psect usage, dynamic area allocation, and management for I/O buffer space and string space. The data covered in this article is for information only and may change in a future release.

If you take advantage of the physical layout of a task rather than the programing conventions used to create it; Murphy's law guarantees that something will change. This article provides background information to enhance your ability to diagnose suspected problems rather than teaching you how to modify the layout. This article takes a few liberties with the actual location of items and terms because of the differences across the PDP-11 operating systems. These cases are noted on first occurence and then ignored. Refer to the particular operating system manuals for the symbolic reference if you need more data about a particular feature.

2.0 GENERAL MEMORY LAYOUT

Any PDP-11 task may be divided into 4 areas in memory. These areas are shown below:

```
          --------------------------------
HIGH     |      un-used address space
ADDRESS  |
          --------------------------------
         |      CODE/DATA PSECTS
          --------------------------------
         |      STACK
          --------------------------------
LOW      |      LOW CORE/TASK HEADER
ADDRESS  |
          --------------------------------
```



These are the never ending chronicles of man's never ending wars with machines.

A PDP/11 IS NEVER DOWN UNTIL IT IS DOWN

by Keith Goodwin
Otero Junior College, Colorado

Back in 1977 Otero Junior College's Data Processing was done on a CDC 6400 located at Colorado State University. This was all right - except - Otero Junior College and Colorado State University are 200 miles apart. This was all right - except the higher education system of Colorado thought that it was saving money by putting Data Processing onto the State owned micro wave system. This was all right - except that one of the legs of the system that went between the two schools was maintained by the State of Colorado, and the other leg was maintained by (you guessed it) Ma Bell. If a bird took to the air anywhere between the two locations the microwave system went down, and the State of Colorado and Ma Bell would point the accusing finger at each other. This means that the system was down almost all of the time, and when it was up every Freshman at Colorado State University had priority over Otero Junior College - because, you know, they'er ONLY a Junior College.

All in all the Data Processing at Otero Junior College was depressing. I began to cast about for a alternative to the situation. Vendors were like wolves at my door. The DEC wolf assured me that I would have no trouble comming up on a PDP/11 34. And best of all -
"A PDP/11 IS NEVER DOWN UNTIL IT IS DOWN".
Who could argue with logic like that.

We signed on the dotted line and awaited the delivery of our new computer. DEC had arranged for us to lease time on Southern Colorado Power's PDP/11 70 until our own arrived. We thought that this would work well for the two or three months that we had to wait, because -
"A PDP/11 IS NEVER DOWN UNTIL IT IS DOWN".

Continued page 13

On RSTS/E systems the low address area is called low core while on RSX type system this area is called the task header. This area is detailed in the operating system manuals. The primary characteristic of this area is that it is read only storage (you should never modify it with your code). The area of interest to the BP2 OTS (object time system) is the location designated as $OTSV - OTS vector pointer. This pointer is one word of the set of 4 words called the Low Core Context area. The other three words of the low core context are: (1) .FSRPT - File Storage Region Pointer, (2) N.OVPT - Overlay Runtime System Pointer, and (3) $VEXT - Vector Extension area Pointer. All four pointers (which must be referenced symbolically) are set up at TKB time. Any modifications to these pointers at run time will be lost on the next context switch of the task. $OTSV points to the BP2 OTS work area. This work area contains pointers to the current task context as maintained by BP2; for example: pointers to the current line number, module name, I/O buffer area, string space, and free space.

The next major area of a task is the STACK area. This is the standard PDP-11 stack area. It is used by BP2 for calls, temporary storage, etc. During the running of a BP2 program the stack pointer should be at the original top of the stack at the beginning of each line/statement (not clause).

The next major area is the CODE/DATA PSECT area. If you are not familiar with psect's and their attributes, you should break out your copies of the MACRO-11 and TKB manuals and get busy. The psect's used in BP2 are described in a subsequent section of this article. At this introductory level, it is sufficient to say that all code and data is organized by usage of psect's. A task map is your best guide to where the individual psect's may be located within the in-core task image.

The last area designated above as UNUSED ADDRESS SPACE is used by BP2 as the dynamic area of a task. This area is "claimed" at either TKB time or at run time by the extend task directive. The dynamic area is used for string and I/O operations.

Another possible address area for a task may be located at the high end of the high address segment. This is the area for memory resident libraries (HISEG's, KEYBOARD MONITOR's, SHARED GLOBAL AREAS (SGA's), RUN TIME SYSTEMS, SHARED LIBRARIES, and so forth). During TKB time, space is allocated for the various segments of a task. The basic algorithm used by TKB is to allocate memory resident libraries from the top of the address space (APR 7) down while code/data sections from object modules are allocated from the bottom of the address space (APR Ø) upwards. (If you don't know about APR's, you should study the KT-11 hardware manuals, the memory management directives, and ask a friendly co-worker to explain them to you.) If you are using memory resident areas, the unused address space still exists between the top of the psect area and the bottom of the memory resident area. This means that the BP2 dynamic area is virtually allocated between two code areas.

The following diagrams illustrate a task in-memory. These illustrations should help you fix a mental image of a program as it exists at run time. This image will be expanded to include progressively more detail in the remainder of the article. The first diagram shows a single segment task, the second diagram shows a single segment task with a resident library, the third shows a multi-segment (overlayed) task. The fourth modifies the diagram for a multi-segment task with resident libraries. In all of these diagrams, the area shown as the unused address space (which is what it is at TKB time) becomes the task extension area which becomes the BP2 dynamic area at run time.

SINGLE SEGMENT TASK

```
                -------------------------------
   HIGH        |        unused address space
   ADDRESS     |
                -------------------------------
               |        CODE/DATA PSECTS
               |                (USER CODE)
               |                (OTS CODE)
                -------------------------------
               |        STACK
                -------------------------------
   LOW         |        LOW CORE/TASK HEADER
   ADDRESS     |
                -------------------------------
```

SINGLE SEGMENT TASK WITH RESIDENT LIBRARY

```
                -------------------------------
   HIGH        |        RESIDENT LIBRARY
   ADDRESS     |        (OTS PSECTS)
                -------------------------------
               |        unused address space
                -------------------------------
               |        CODE/DATA PSECTS
               |                (USER CODE)
               |                (OTS CODE)
                -------------------------------
               |        STACK
                -------------------------------
   LOW         |        LOW CORE/TASK HEADER
   ADDRESS     |
                -------------------------------
```

MULTI-SEGMENT TASK

```
              --------------------------------
  HIGH      |        unused address space
  ADDRESS   |
              --------------------------------
            |        OVERLAY LOAD AREA
            |        (CODE/DATA PSECTS)
              --------------------------------
            |        CODE/DATA PSECTS - ROOT
            |                (USER CODE)
            |                (OTS CODE)
              --------------------------------
            |        STACK
              --------------------------------
  LOW       |        LOW CORE/TASK HEADER
  ADDRESS   |
              --------------------------------
```

MULTI-SEGMENT TASK WITH RESIDENT LIBRARY

```
              --------------------------------
  HIGH      |        RESIDENT LIBRARY
  ADDRESS   |        (OTS PSECTS)
              --------------------------------
            |        un-used address space
              --------------------------------
            |        OVERLAY LOAD AREA
            |        (CODE/DATA PSECTS)
              --------------------------------
            |        CODE/DATA PSECTS - ROOT
            |                (USER CODE)
            |                (OTS CODE)
              --------------------------------
            |        STACK
              --------------------------------
  LOW       |        LOW CORE/TASK HEADER
  ADDRESS   |
              --------------------------------
```

By reviewing the preceding illustrations, you will see that we have begun dividing any view of a task into its primary areas and have further subdivided those areas into psect's. The next section defines the psect areas.

3.Ø PSECT USAGE

Since this article is about a higher level language, this section will make a distinction between the psect's used by the OTS and the psect's generated by the compiler based on the source language input. Before the psect's are actually described let's review the definition of a psect. A PSECT (program section) is a block of code or data consisting of a name, a set of attributes and a length. A psect is the basic unit produced by a language processor. As input to TKB the psect definitions are used to determine the placement of code and data in the task image. A psect's name is used internally by the language processor or TKB to maintain tables which contain data about the psect attributes and length. The psect's attributes define the section's contents, its placement in the task image, and possibly the mode of access allowed (read-only or read-write). The program section (psect) length determines how much address space TKB must reserve for the section. Please refer to a TKB manual for more information about PSECT's.

4.Ø BP2 OTS PSECT USAGE

The BP2 OTS is built to run in the 'BP2OTS' psect. If you examine a task map for a BP2 task, you can observe that all modules extracted from the disk libraries for BP2 are contained in the BP2OTS psect. BP2 object modules mapped into a task by way of a resident library are included in a psect such as "???OTS" where the question marks are replaced by the CCL/MCR (3 character name used to invoke the compiler). The installation task "RESSTB" modifies the resident library psect because a disk modules and resident library modules may not contain identical psect's. The blank psect is no longer used by the BP2 object time system. All of this doesn't make much difference until you start trying to debug a program and you want to identify the module owner. If it is a BP2 module, you can now identify it by the psect in which it resides. The psect usage also comes in handy when you suspect that modules from the current version and the previous version are being intermixed by the user. This simple convention of psect usage will assure that BP2 V1.5 object module patches can not be applied to BP2 V1.6 and vice versa. The only other psect set by the BP2 OTS is ".99998". This is the patch space psect in resident libraries. The OTS psects are set with the attributes of read/write, instruction space, local, reloactable, and concatenated (RW,I,LCL,REL,CON).

5.Ø COMPILER GENERATED PSECT'S

When a user program is compiled BP2 normally uses 12 psect's to contain the results of the compilation. In addition, the user may indirectly generate psect's via the COMMON/MAP statements of the language. If the user program contains a source line such as the following:

```
        1Ø      COMMON  (JUNK)  A$=1ØØ
```

a psect definition for "JUNK" is generated by the compiler. You can observe the results of this generation process by the BP2 restrictions on COMMON/MAP names e.g. six character and special symbol limitations. The blank COMMON, indicated in a TKB map as the psect ".$$$$.", can be generated by BP2 if the user does not name COMMON areas. The 13 primary psect's generated are as follows:

PSECT NAME	GENERAL USAGE
$ARRAY	Array header storage
$CODE	Threaded code generated from user's source
$FLAGR	Recursion flag area
$FLAGS	Subroutine recursion flag
$FLAGT	Recursion flag area size indicator
$ICIO1	I/O Vector area
$ICIO2	I/O Impure (dirty) area
$IDATA	Impure OTS area and numeric variable area
$PDATA	Constant (pure) data area
$SAVSP	Initial stack pointer save area
$STRNG	String Header storage
$TDATA	Temporary array header storage area
$SYMTB	Symbol table storage for DEBUG

The compiler generated psect's have attributes set by the intended usage of the psect. The three primary uses are control psect's, code psect, and data psect's. The control psect's are $FLAGR, $FLAGT, and $FLAGS which have attributes of "RW, D, GBL, REL, CON" that is read/write, data, global, relocatable, and concatenated. The code psect, $CODE, contains the BP2 threaded code and has the attributes of "RW, I, LCL, REL, CON" that is read/write, instruction space, local, relocatable, and concatenated. The data psect's are divided into three additional categories: COMMON/MAP areas, OTS I/O areas and actual data. COMMON/MAP data areas are assigned the attributes of "RW, D, GBL, REL, OVR". The OTS I/O areas are $ICIO1 and $ICIO2 and have the attributes "RW, D, GBL, REL, OVR". The remaining psect's ($ARRAY, $IDATA, $PDATA, $SAVSP, $STRNG, $TDATA, and $SYMTB) have attributes of "RW, D, LCL, REL, CON". The psect attributes have significance when determining which routines may be placed in resident libraries and how data will be loaded when you overlay a task.

The I/O psect's deserve more detailed treatment than they were just given. These psect's ($ICIO1 and $ICIO2) control the I/O operations for BP2. $ICIO2 is a run time impure (dirty) area used for system specific data structures and RMS user (BP2) data structures. In other words, it contains a FAB block, a default device string, DPB (directive parameter block) and some miscellaneous data structures. $ICIO1 controls which I/O operations are valid from any given BP2 task. (That's right - this is where the control for the Illegal I/O operation message comes from.) $ICIO1 is organized by the classes of

I/O operations a BP2 task may perform. There are sixteen classes of I/O operations allowed. The following diagram illustrates the classes of I/O operations:

```
----------------------------
| INITIALIZATION CLASS
----------------------------
| OUTPUT CLASS
----------------------------
| INPUT CLASS
----------------------------
| OPEN CLASS
----------------------------
| CLOSE CLASS
----------------------------
| GET SEQUENTIAL CLASS
----------------------------
| GET RANDOM CLASS
----------------------------
| PUT SEQUENTIAL CLASS
----------------------------
| PUT RANDOM CLASS
----------------------------
| FIND SEQUENTIAL CLASS
----------------------------
| FIND RANDOM CLASS
----------------------------
| UPDATE CLASS
----------------------------
| DELETE CLASS
----------------------------
| RESTORE CLASS
----------------------------
| UNLOCK CLASS
----------------------------
| RESERVED
----------------------------
```

Each class of possible I/O operation is subdivided by the four primary types of file OPEN supported by BP2. The four types are terminal/virtual, sequential, relative, and indexed. Thus there are four types of I/O which may be performed on sixteen classes of I/O operations which indicates that the $ICIO1 psect is 128 bytes or 64 words or 64 entries in length (it's a jump table). This table is filled in by TKB and the BP2 task initialization code. The modules in the BASRMS object library of BP2 all contribute to the $ICIO1 psect. If you linked all of the BASRMS modules in your task the jump table would be completely filled in and you would also have all of the BP2/RMS code for all I/O operations. By using the BUILD switches of BP2 and the specialized ODL files, each task links in only the code to support the type(s) of I/O it is using at TKB time.

The remaining entries in $ICIO1 are zero at task inititalization time. The BP2 init-routine scans the table and inserts the address of the Illegal operation message generator in all table entries containing a zero. This mechanism reduces the run time error checking for some of the possible I/O combinations and could be by-and-large replaced by a link time weak reference facility. Since we have digressed from the task layout, let me leave you with one last thought about $ICIO1 and $ICIO2. A word of warning - ADJACENCY IS ASSUMED FOR $ICIO1 AND $ICIO2. If you use the sequential switch of TKB so that the psect's are not in alphabetic order, you can cause the BP2 I/O system to collapse. This is fairly difficult to do but a small potential for problems exists. Adjacency is also assumed for the flag psects: $FLAGR, $FLAGS and $FLAGT.

The $IDATA psect also contains storage for the OTS. $IDATA is used as storage for the line number table (for chaining) at task initialization time, as the primary storage area for the OTS work area at run time and as the argument address passing area for subroutines. On task initialization, the first word of $IDATA is used as the line number table existence indicator. If the task was chained to, then the address of the chain-to line number is looked up and stored until the end of task initialization. If there was no chain-to line number then the address of the first line is used. After this bit of housekeeping the line number storage area becomes the OTS work area.

In the beginning of this article reference was made to the symbol "$OTSV" as being the pointer to the OTS work area. $OTSV points within the $IDATA psect, to the actual physical location of the work area. The remainder of the $IDATA psect contains numeric variable data from the user's program. You can realize an immediate size benefit if you construct CHAIN with line number tasks as overlayed tasks. In this case the root module contains the dispatch control logic (based on line numbers) while the overlays perform the real work. In this way you can compile the overlays without line numbers and save that space in the root segment. You also save space in the root by reducing the size of the line table since you have reduced the number of lines.

The remaining psect's are used as indicated in the preceding diagram and are not explained any further in this article.

6.0 OTHER PSECTS IN IMAGE

Since this article discusses the task layout, it should at least reference the other psect's you may see when you examine a task map. The remaining psect's are not part of BP2 but are used by BP2-RMS-OPERATING SYSTEM to run your BP2 task image. Your primary guide to these psect's is the TKB manual for your particular operating system.

Below is a list of the psect's for operating system support of single segment but overlayed tasks.

PSECT NAME	GENERAL USAGE
$$ALER	Subroutine to handle autoload errors
$$MRKS	Mark segments routine
$$OVDT	Overlay data
$$RDSG	Read segments routine
$$RTS	Return instruction
$$SGD0	Start of task segment descriptors
$$SGD2	End of task segment descriptors

Below is a list of the psect's required for support of multi-segment tasks.

PSECT NAME	GENERAL USAGE
$$ALVC	Segment autoload vectors
$$AUTO	Overlay auto-load routine
$$OVRS	Overlay data
$$RGDS	Region descriptors
$$SGD1	Task segment descriptors
$$WNDS	Task window descriptors

7.0 EXPANDED MULTI-SEGMENT TASK WITH RESIDENT LIBRARY ILLUSTRATION

The following illustration shows the major areas (PSECT'S) discussed in this article. Two major items are omitted from this illustration. The first omission is the sizes for the psect shown. This is left to you and a task map because the sizes and even the inclusion of some of the psect's is highly variable by task. The second omission is the entire RMS structure.

```
          -------------------------------
HIGH     |     RESIDENT LIBRARY
ADDRESS  |     (OTS PSECTS)
         |             .99998 SPACE
         |             ???OTS SPACE
          -------------------------------
         |     unused address space
          -------------------------------
         |     OVERLAY LOAD AREA
         |     (CODE PSECTS)
         |             $$RTS   *
         |             $$ALVC SPACE
         |             $TDATA SPACE
         |             $STRNG SPACE
```

```
                  |            $PDATA SPACE
                  |            $IDATA SPACE
                  |            $FLAGT *
                  |            $FLAGS *
                  |            $FLAGR *
                  |            $CODE  SPACE
                  |            $ARRAY SPACE
                  |            BP2OTS SPACE
                  |     (DATA PSECTS)
                  |            .$$$$. *
                  |            (USER) *
                  | (* - may resolve to lower segment)
                  -------------------------------
                  |     CODE/DATA PSECTS - ROOT
                  |            .$$$$. SPACE
                  |            $$WNDS SPACE
                  |            $$VEX1 SPACE
                  |            $$VEXØ SPACE
                  |            $$SGD2 SPACE
                  |            $$SGD1 SPACE
                  |            $$SGDØ SPACE
                  |            $$RTS  SPACE
                  |            $$RGDS SPACE
                  |            $$RDSG SPACE
                  |            $$OVRS SPACE
                  |            $$OVDT SPACE
                  |            $$MRKS SPACE
                  |            $$AUTO SPACE
                  |            $$ALVC SPACE
                  |            $$ALER SPACE
                  |            $TDATA SPACE
                  |            $STRNG SPACE
                  |            $SAVSP SPACE
                  |            $PDATA SPACE
                  |            $IDATA SPACE
                  |            $ICIO2 SPACE
                  |            $ICIO1 SPACE
                  |            $FLAGT SPACE
                  |            $FLAGS SPACE
                  |            $FLAGR SPACE
                  |            $CODE  SPACE
                  |            $ARRAY SPACE
                  |            BP2OTS SPACE
                  |            . BLK. SPACE
                  -------------------------------
                  |      STACK
                  -------------------------------
LOW               |      LOW CORE/TASK HEADER
ADDRESS           |
                  -------------------------------
```

8.Ø BP2 DYNAMIC AREA COMMENTS

Throughou' this article several references have been made to the unused address space at run time and to the BP2 dynamic area. This section briefly discusses the dynamic area. First, let's consider how BP2 uses the dynamic area. To over-simplify, the dynamic area is used for dynamic string handling, I/O buffers, the program context, scratch buffers, etc. The mechanism BP2 uses to acquire this space is the extend task command at TKB time or extend task directive at run time. If you are not familiar with these mechanisms refer to the TKB manual and the system directives manual. After a task extension, the acquired space and the remaining unused address space may be illustrated as follows:

```
       BEFORE EXTENSION                    AFTER EXTENSION
                                     ---------------------------
              .                      | unused address space
  ------------------------           ---------------------------
  |                                  |        string space
  | unused address space             |
  |                                  ---------------------------
  |                                  |
  |                                  |        free space
  |                                  |
  |                                  ---------------------------
  |                                  |
  |                                  |        I/O buffer space
  |                                  |
  ------------------------           ---------------------------
  | code/data psects                 | code/data psects
  ------------------------           ---------------------------
```

String space expands down from the top of the task and is used for string storage. I/O buffer space expands up and is used for I/O buffers and data blocks associated with I/O. Free space is the amount of space not currently allocated to either I/O or string space. When an operation requires dynamic space, free space is checked to see if there is enough space currently available. If there is insufficient space, a string compression is performed to attempt to gather enough free space. If there is still insufficient space a task extension is performed for the amount of space needed for the operation (rounded to the next 32 word block boundry). If this task extension fails, the program aborts with a maximum memory exceeded error. Assuming the extension succeeded, the next operation is to move string space to the new top-of-memory, thereby creating a large enough free space area. Based on the operation which started this this process, free space is then reduced and either string space or I/O space is increased. The BP2 V1.6 USER'S GUIDE, chapter 3, section 1Ø, contains more information about the handling of I/O space. Also, a previous article of this series discussed the "$SIZE" routine which can be used to examine the allocations for the various areas.

This article has referenced several other manuals and resources that you can access and that you should be using. The intent of the article has been to provide a framework (albeit a BP2 framework) to enhance your understanding of the BP2 products. Other articles address other aspects of BP2. As stated in the first article of this series, we would appreciate your input and feedback.

War Stories (cont.)

Everything was going along smoothly until Friday - all of the terminals on campus went dead. We called up Southern Colorado Power to find out what was wrong. It seems that one of the local cowboys had started has Friday night partying a little early and ran his pick-up truck into the wall down at the power company. When he hit the wall he drove the computer room window mounted air conditioner right into the back of the PDP/11 70 and knocked the computer over. So the computer was down.

DEC field service came out and righted the computer, reset all ot the breakers and rebooted the system. And that son of a gun started right up. Which only goes to show you that A PDP/11 IS NEVER DOWN UNTIL IT IS DOWN.

THE LIGHTS ARE ON - SO IT MUST BE RUNNING

by Steve Harrison
Skidmore, Owens and Merrill, San Francisco

In the old days we had a PDP/11 45 with all of the lights in the front, you know one of the computers that you could tell was running because all of those lights blinked on and off.

One day it seemed that the computer went down, but it acted like it was up. A user could input to a terminal and it would echo. The lights were blinking on and off. But the darned things that didn't seem to work. As we watched the lights a pattern emerged - the lights were incrementing by one every time someone hit a key. And the machine told us it was up. We also found that if someone sat at the console and keyed the computer seemed to process.

DEC field service came out and told us that the clock had died, so even though the lights were on the machine was not running.

EXTENSIONS TO BASIC-PLUS-2

In response to a number of requests, we have developed several extensions to BASIC-PLUS-2. These extensions take the form of MACRO subroutines CALLable from a BASIC-PLUS-2 program. In anticipation of Version 2 syntax plus features, these extensions are unsupported. The routines have been checked for correctness, but are not be guaranteed that they will work under all conditions. Anyone who experiences a problem with them or needs further information should submit an information SPR to BP2.

BASIC-PLUS-2/RMS EXTENSIONS

To use these routines effectively you must have a copy of the RMS-11 MACRO Programmer's Reference Manual handy and be familiar with its contents.

A. RFA ACCESS

The ability to obtain the RFA (RMS Record's File Address) and subsequentially use it to retrieve the record is provided in four subroutines:

[Note: All numeric parameters must be integer.]

1. CALL RETRFA BY REF (A%, B%)

This routine returns the RFA of the record last accessed on channel A%. B% must be the first of three contiguous integers. It is recommended that B% be in COMMON or a MAP, because the next two words will be written into.

```
10   COMMON (RFA) B%, FILL$=4%
```

2. CALL GETRFA BY REF (A%, B%)

This routine will effect a "GET by RFA" on a file. The parameters are as above. RFA access is a form of random access. This routine set up the parameters necessary for RFA access and then jumps into the standard GET routine. Therefore, it is used in place of a GET and will process just like an ordinary GET returning, if necessary, the ordinary errors. In addition, a special error - #132 "Record has been deleted" will be returned if the record has been deleted since the RFA was saved, or the RFA was bad.

```
100 COM RFA%, RFA1%, RFA2%
110 GET #1%, KEY #1% EQ KEY$        &
 \  CALL RETRFA BY REF  (1%, RFA%)
     . . .
500 CALL GETRFA BY REF  (1%, RFA%)
```

B. INDEXED NULL KEY

To enable the NULL KEY feature, specify the NULKEY routine before the OPEN statement to which it applies:

> CALL NULKEY BY REF (NUMK%, KRF%, VAL$) where NUMK% is the total number of keys in the file, KRF% is the key number to which it applies, and VAL$ is a one-byte string containing the null key value. NULKEY cannot be used on the primary key.

C. SEGMENTED KEYS

To define a segmented key in BASIC-PLUS-2, one inserts the following call (not call by REF) before the OPEN statement:

> CALL SEGKEY (NUMK%, KRF%, A1$, A2$,...AN$) where NUMK% and KRF% are as above. A1$, etc., are the segments of the key. Up to eight (8) can be defined, and each <u>must</u> be part of the MAP statement used in the accompanying OPEN statement. Furthermore, in the OPEN statement one must specify a dummy key in the KEY field of the OPEN statement, and insure that the sum of segment lengths. If necessary, use a COMMON statement with the same name as the MAP to redefine variables.

To access Index Files with segmented keys use the multi MAP Facility with standard GET's and PUT's.

D. BUCKET FILL FACTORS

To make use of bucket fill factors on indexed files, one inserts the following before the OPEN:

> CALL BKTFIL BY REF (NUMK%, KRF%, DATF%, INDXF%)) where NUMK% and KRF% are as above. DATF% expresses the data bucket fill factor in bytes and ranges from Ø to the bucketsize. IDNXF% is the corresponding factor for index buckets. This call only sets the numbers; in order to cause it to take effect, one must set a bit in the ROP field - see SETROP for details.

E. RECORD PROCESSING OPTIONS

During the execution of the program, it may be necessary to set and clear certain record processing options. This can be accomplished by two calls:

```
   CALL SETROP BY REF (CHAN%, BITS%)
   CALL CLRROP BY REF (CHAN%, BITS%)
```

where CHAN% is the channel number of the file used and BITS% is the value to be SET or CLEARED. The values once set stay set until explicitly cleared with the CLRROP call or the file is

closed. One may only use the following values:

VALUE	MEANING	SYMBOLIC
16	Load by fill factor	RB$LOA
4Ø96	Fast delete	RB$FDL
8192	Update if	RB$UIF

For further details, see The RMS Manual. The use of any other value will result in unpredictable results! The SETROP with a value of 16 is necessary to trigger the bucket fill factors set up by the BKTFIL call.

F. DEFAULT ALLOCATION QUANTITY

To set the default file extension quantity use the following call before the OPEN statement to create the file.

```
   CALL DEFALQ BY REF (A%)
```

A% specifies the minimum disk file extension as the number of blocks.

G. DEFAULT FILE NAME STRING

A default file name string (RMS field DNA) may be set with the call (not CALL BY REF) below:

```
   CALL DEFFNA (NAME$)
```

H. SET RMS FOP FIELD

The following call may be used to set the RMS FOP field:

```
   CALL SETFOP BY REF (BIT%)
```

where BIT% is the value to be set. By using this call, you may set the deferred write bit (FB$DFW - 8192.) of the FAB.

I. Multiple record streams can be connected to a single RMS Indexed or Relative file by using a special form of the OPEN. First one OPEN's the file in a normal manner, say on Channel A%. Next one uses the following statement:

```
     OPEN A$ AS FILE # B%, INDEXED (RELATIVE)     &
                   , CONNECT A%                   &
                   , MAP BUF
```

The restrictions are:

a. The file name, A$, is a dummy for syntax commpleteness only.

b. The Channel number, B%, must not be the same as A% and must be a valid channel.

c. The record buffer referenced in the MAP should be different from that of the original.

d. No key specifications are to be used.

In general, the number of streams to be connected is limited only by the number of channels or memory space available. Each connected stream takes as much memory as a regularly opened file less 5Ø bytes. Files opened by this method can be accessed and CLOSED as normal files.

In general, all of the calls affecting OPENS can be called in any order and there can be as many of each as numbers of keys; however, all must come immediately before the open statement. SETROP must appear after an OPEN and before the first record operation to which it applies.

EXAMPLES

To set up bucket fill factor and use on an Indexed file, and to use segmented keys:

```
  1Ø  MAP (BUF) A$=4, B$=2, C$=4, D$=12, E$=8, FILL$=7Ø

  15  MAP (BUF) Q$=22, REC$=78
  2Ø  CALL SEGKEY (2%,Ø%,B$,D$,A$)

  25  CALL BKTFIL BY REF (2%,1%,256%,256%)

  3Ø  OPEN FIL$ FOR OUTPUT AS FILE #1%      &
           , INDEXED FIXED                  &
           , ACCESS WRITE, ALLOW NONE       &
           , MAP BUF                        &
           , PRIMARY Q$                     &
           , ALTERNATE E$

     o o o

 14Ø  CALL SETROP BY REF  (1%, 16%)

 15Ø  PUT  #1%

     o o o

 1ØØØ  CLOSE  #1%
```

In this example the MAP at line 1Ø defines the segments of the primary key while the MAP at line 15 redefines the record layout. The call at 2Ø sets up the primary key as a segmented key

consisting of three (3) segments (B$, D$, and A$) for a total length of 17 bytes. In statement 3Ø, there is a dummy key Q$ where length is greater than 18, so all is well. At line 25, the bucket fill factor is defined for the alternate key as 256 bytes. Since the bucket size at line 3Ø defaults to one block(512 bytes), the buckets will be only half filled when initially filling. The SETROP aat line 14Ø activates this partial filling.

The above routines are "hidden" in the BP2 object library on the standard distribution tape. So you need only to reference them and re-TKB to have them included in your Task. The entry points plus modules names are as follows:

ENTRY	MODULE
BKTFIL	XABSET
NULKEY	XABSET
SEGKEY	XABSET
CLRROP	FIXROP
SETROP	FIXROP
DEFALQ	FABSET
DEFFNA	FABSET
SSTFOP	FABSET
CLRROP	FIXROP
SETROP	FIXROP
GETRFA	GETRFA
RETRFA	GETRFA

BASIC-PLUS-2 DEBUGGING AIDS

A. Hidden Debugger Option.

The BP2 debugger has one hidden command - "BPT". This command is used to generate an unexpected breakpoint into ODT or a reserved instruction trap if ODT has not been included in the Task.

B. OTS Dynamic Area Size

CALL $SIZE BY REF (CORE%, IOBSZ%, STRSZ%, FRESZ%)

This routine returns the size in words for the current task size (CORE%), the current I/O buffer space allocation (IOBSZ%), the current string space allocation (STRSZ%), and the amount of free space (FRESZ%). The following calls may be used to return the parameters on an individual bases.

```
A.  CALL  $CORE BY REF (CORE%)
B.  CALL  $IOBSZ BY REF (IOBSZ%)
C.  CALL  $STRSZ BY REF (STRSZ%)
D.  CALL  $FRESZ BY REF (FRESZ%)
```

If you are running TRAX-11 in TST mode, the individual calls (not the $SIZE form) should be used to avoid a multiple definition between the BP2 routine and a parameter in TPSCOM. An alternate name for the $SIZE call in TRAX TST mode is $BPSIZ.

USEFUL FUNCTIONS

A. CALL RAD (A%, A$)

This routine uses a straight call not a CALL BY REF. The input string is limited to 3 upper case characters plus "$", ".", and blank. The output integer is the RAD50 conversion or zero if an error was encountered in the conversion.

MAP, MOVE, and FIELD Performance

The purpose of this article is to give some insight on the performance of the MAPs, MOVEs, and FIELDs statements in BASIC-PLUS-2, VAX-11 BASIC, BASIC-PLUS (when applicable) and also some helpful hints in transferring files across the different operating systems.

This first part of the article is a report on the perfomance testing that was conducted for the MAP, MOVE, and FIELD statements. The following conditions were present:

System: PDP-11/34, VAX 11/780
Operating systems: RSTS/E V7.Ø, RSX-11M V3.2, VAX/VMS V2.Ø
BASIC: BASIC-PLUS-2 V1.6(patched), BASIC-PLUS and VAX-11 BASIC V1.Ø (VB)
Math option: EIS instruction set (except VMS)
Switches: NOCHAIN, NOLINE, SINGLE, NODEBUG
Library: none

Note: Each test was done on a stand-alone machine

The following program was used to create the necessary BLOCK I/O files with 1Ø,ØØØ records and record sizes of 512 bytes.

```
1Ø   OPEN "TIME.DAT" FOR OUTPUT AS FILE #1%,VIRTUAL,RECORDSIZE 512%
2Ø   OPEN "TIMFLD.DAT" FOR OUTPUT AS FILE #2%,RECORDSIZE 512%
3Ø   FOR I%= 1% TO 1Ø%
4Ø      READ LAST$,FIRST$
5Ø      SSN$="999-99-9999"
6Ø      PRINT I%
7Ø      FOR X% =1% TO 1ØØØ%
8Ø         WRKHRS= INT(RND*5Ø%) + INT(RND*1Ø%)
9Ø         HOUWAG=INT(RND*12%) + INT(RND*1Ø%)
1ØØ        YTDHRS= INT(RND*1ØØØ%) + INT(RND*1Ø%)
11Ø        YTDWRK= INT(RND*1ØØØ%) + INT(RND*1Ø%)
12Ø        GROSS=Ø
13Ø        MOVE TO #1%,LAST$=16%,FIRST$=16%,SSN$=12%,WRKHRS,HOUWAG
14Ø        MOVE TO #1%,FILL$=52%,GROSS,YTDHRS,YTDWRK
15Ø        PUT #1%
16Ø        A$=CVTF$(WRKHRS) B$=CVTF$(HOUWAG)  C$=CVTF$(GROSS)
17Ø        D$=CVTF$(YTDHRS)e$=CVTF$(YTDWRK)
18Ø        MOVE TO #2%,LAST$=16%,FIRST$=16%,SSN$=12%,A$=4%,B$=4%
19Ø        MOVE TO #2%,FILL$=52%,C$=4%,D$=4%,E$=4%
2ØØ        PUT #2%
21Ø     NEXT X%
22Ø  NEXT I%
23Ø  CLOSE #1%,#2%
24Ø  DATA 'SMITH','STEVE','SMITH','BOB','SMITH','JOHN','SMITH'
25Ø  DATA 'FRED','SMITH','TOM','SMITH','BILL','SMITH','FRED'
26Ø  DATA 'SMITH','JOHN','SMITH','ME','SMITH','BILL'
27Ø  END
```

The next three programs test the different methods in which BASIC programs can assign fields in a buffer to variables. Each timing program reads a record in the file, accesses the fields in the buffer, keeps a running sum of some of the fields and then updates some of the fields in the buffer without actually updating the record.

The following programs use MAP or MOVE or FIELD to assign fields in the buffer to the variable. The first two programs are not used under BASIC-PLUS because it does not support MAP and MOVE statements and the tests results will reflect this.

```
TEST 1 - MAPS
   10   SUM, SUM1, SUM2=0
   20   MAP (TEST) LAST$, FIRST$, SSN$=12, WRKHRS, HOUWAG, FILL$=460%
   30   MAP (TEST) FILL$=52%, GROSS, YTDHRS, YTDWRK, FILL$=448%
   40   OPEN "TIME.DAT" FOR INPUT AS FILE #1%, VIRTUAL, MAP TEST
   50   A=TIME(0%)
   60   FOR I% = 1% TO 10000%
   70      GET #1%
   80      GROSS = WRKHRS * HOUWAG
   90      SUM = SUM + GROSS
   100     SUM1 = SUM1 + YTDHRS
   110     SUM2 = SUM2 + YTDWRK
   120     YTDHRS = YTDHRS + WRKHRS
   130     YTWRKS = YTDWRK + GROSS
   140  NEXT I%
   150  PRINT TIME(0%) - A, SUM, SUM1, SUM2
   155  CLOSE #1
   160  END

TEST 2 - MOVES
   10   SUM, SUM1, SUM2 = 0
   20   OPEN "TIME.DAT" FOR INPUT AS FILE #1%,VIRTUAL,RECORDSIZE 512%
   30   A = TIME(0%)
   40   FOR I% = 1% TO 10000%
   50      GET #1%
   60      MOVE FROM #1%, LAST$=16%, FIRST$=16%, SSN$=12%, WRKHRS, HOUWAG
   70      MOVE FROM #1%, FILL$=52%, GROSS, YTDHRS, YTDWRK
   80      GROSS = WRKHRS * HOUWAG
   90      SUM = SUM + GROSS
   100     SUM1 = SUM1 + YTDHRS
   110     SUM2 = SUM2 + YTDWRK
   120     TEMP1 = YTDHRS + WRKHRS
   130     TEMP2 = YTDWRK + GROSS
   140     MOVE TO #1%, FILL$=56%, TEMP1, TEMP2
   150  NEXT I%
   160  PRINT TIME(0%) - A, SUM, SUM1, SUM2
   165  CLOSE #1
   170  END

TEST 3 - FIELDS
   10   !EXTEND  USED ONLY FOR BASIC-PLUS
   20   SUM, SUM1, SUM2 = 0
   30   OPEN "TIMFLD.DAT" FOR INPUT AS FILE #1%, RECORDSIZE 512%
   40   A = TIME(0%)
```

```
   50   FOR I% = 1% TO 10000%
   60      GET #1%
   70      FIELD #1, 16% AS LAST$, 16% AS FIRST$, 12% AS SSN$, 4% AS A$
   80      FIELD #1, 48% AS Z$, 4% AS B$, 4% AS C$
   90      FIELD #1, 56% AS Z$, 4% AS D$, 4% AS E$
   100     WRKHRS = CVT$F(A$) \  HOUWAG = CVT$F(B$)
   110     YTDHRS = CVT$F(D$) \  YTDWRK = CVT$F(E$)
   120     GROSS = CVT$F(C$)
   130     GROSS= WRKHRS * HOUWAG
   140     SUM = SUM + GROSS
   150     SUM1 = SUM1 + YTDHRS
   160     SUM2 = SUM2 + YTDWRK
   170     LSET C$ = CVTF$(GROSS) \ LSET D$ = CVTF$(YTDHRS + WRKHRS)
   180     LSET E$ = CVTF$(YTDWRK + GROSS)
   190  NEXT I%
   200  PRINT TIME(0%) - A, SUM, SUM1, SUM2
   205  CLOSE #1
   210  END
```

The following results are found when running the above programs. The timing was done by using real time not CPU time.

O.S.	Lang.	File type	Time (seconds) FIELD	MOV	MAP
RSTS\E V7.0	BP	BLOCK I/O	188	-	-
RSTS\E V7.0	BP2	BLOCK I/O	188	186.4	182.8
RSX-11M V3.2	BP2	BLOCK I/O	292.8	189.5	178.8
VMS V2.0	VB	BLOCK I/O	57.4	51	50.8

The following program was used to create the necessary sequential files with 10,000 records and fixed record sizes of 512 bytes.

```
   10   OPEN "TIME.DAT" FOR OUTPUT AS FILE #1,SEQUENTIAL FIXED,RECORDSIZE 512
   20   OPEN "TIMFLD.DAT" FOR OUTPUT AS FILE #2,SEQUENTIAL FIXED,RECORDSIZE 512
   30   FOR I%= 1% TO 10%
   40      READ LAST$,FIRST$
   50      SSN$="999-99-9999"
   60      PRINT I%
   70      FOR X% =1% TO 1000%
   80         WRKHRS= INT(RND*50%) + INT(RND*10%)
   90         HOUWAG=INT(RND*12%) + INT(RND*10%)
   100        YTDHRS= INT(RND*1000%) + INT(RND*10%)
   110        YTDWRK= INT(RND*1000%) + INT(RND*10%)
   120        GROSS=0
   130        MOVE TO #1%,LAST$=16%,FIRST$=16%,SSN$=12%,WRKHRS,HOUWAG
   140        MOVE TO #1%,FILL$=52%,GROSS,YTDHRS,YTDWRK
   150        PUT #1%
   160        A$=CVTF$(WRKHRS) B$=CVTF$(HOUWAG)  C$=CVTF$(GROSS)
```

```
  17Ø        D$=CVTF$(YTDHRS)e$=CVTF$(YTDWRK)
  18Ø        MOVE TO #2%,LAST$=16%,FIRST$=16%,SSN$=12%,A$=4%,B$=4%
  19Ø        MOVE TO #2%,FILL$=52%,C$=4%,D$=4%,E$=4%
  2ØØ        PUT #2%
  21Ø     NEXT X%
  22Ø  NEXT I%
  23Ø  CLOSE #1%,#2%
  24Ø  DATA 'SMITH','STEVE','SMITH','BOB','SMITH','JOHN','SMITH'
  25Ø  DATA 'FRED','SMITH','TOM','SMITH','BILL','SMITH','FRED'
  26Ø  DATA 'SMITH','JOHN','SMITH','ME','SMITH','BILL'
  27Ø  END
```

The following three programs test the different methods in which a BASIC-PLUS-2 or VAX-11 BASIC program can assign fields in a buffer (using se- quential files) to a variable. Each timing program reads a record in the file, accesses the fields in the buffer, keeps a running sum of some of the fields, and then updates some of the fields in the buffer without actually updating the record.

The following programs use MAP or MOVE or FIELD to assign fields in the buffer to the variable. BASIC-PLUS was not used with this test set because that BASIC-PLUS does not support RMS files and this is reflected in the test results.

```
TEST 1 - MAPS
  1Ø   SUM, SUM1, SUM2=Ø
  2Ø   MAP (TEST) LAST$, FIRST$, SSN$=12, WRKHRS, HOUWAG, FILL$=452%
  3Ø   MAP (TEST) FILL$=52%, GROSS, YTDHRS, YTDWRK, FILL$=448%
  4Ø   OPEN "TIME.DAT" FOR INPUT AS FILE #1%, SEQUENTIAL FIXED, MAP TEST
  5Ø   A=TIME(Ø%)
  6Ø   FOR I% = 1% TO 1ØØØØ%
  7Ø      GET #1%
  8Ø      GROSS = WRKHRS * HOUWAG
  9Ø      SUM = SUM + GROSS
  1ØØ     SUM1 = SUM1 + YTDHRS
  11Ø     SUM2 = SUM2 + YTDWRK
  12Ø     YTDHRS = YTDHRS + WRKHRS
  13Ø     YTWRKS = YTDWRK + GROSS
  14Ø  NEXT I%
  15Ø  PRINT TIME(Ø%) - A, SUM, SUM1, SUM2
  155  CLOSE #1
  16Ø  END

TEST 2 - MOVES
  1Ø   SUM, SUM1, SUM2 = Ø
  2Ø   OPEN "TIME.DAT" FOR INPUT AS FILE #1%,VIRTUAL,RECORDSIZE 512%
  3Ø   A = TIME(Ø%)
  4Ø   FOR I% = 1% TO 1ØØØØ%
  5Ø   GET #1%
  6Ø      MOVE FROM #1%, LAST$=16%, FIRST$=16%, SSN$=12%, WRKHRS, HOUWAG
  7Ø      MOVE FROM #1%, FILL$=52%, GROSS, YTDHRS, YTDWRK
  8Ø      GROSS = WRKHRS * HOUWAG
  9Ø      SUM = SUM + GROSS
```

```
  1ØØ     SUM1 = SUM1 + YTDHRS
  11Ø     SUM2 = SUM2 + YTDWRK
  12Ø     TEMP1 = YTDHRS + WRKHRS
  13Ø     TEMP2 = YTDWRK + GROSS
  14Ø     MOVE TO #1%, FILL$=56%, TEMP1, TEMP2
  15Ø  NEXT I%
  16Ø  PRINT TIME(Ø%) - A, SUM, SUM1, SUM2
  165  CLOSE #1
  17Ø  END

TEST 3 - FIELDS
  1Ø   !EXTEND  USED ONLY FOR BASIC-PLUS
  2Ø   SUM, SUM1, SUM2 = Ø
  3Ø   OPEN "TIMFLD.DAT" FOR INPUT AS FILE #1%, RECORDSIZE 512%
  4Ø   A = TIME(Ø%)
  5Ø   FOR I% = 1% TO 1ØØØØ%
  6Ø      GET #1%
  7Ø      FIELD #1, 16% AS LAST$, 16% AS FIRST$, 12% AS SSN$, 4% AS A$
  8Ø      FIELD #1, 48% AS Z$, 4% AS B$, 4% AS C$
  9Ø      FIELD #1, 56% AS Z$, 4% AS D$, 4% AS E$
  1ØØ     WRKHRS = CVT$F(A$) \  HOUWAG = CVT$F(B$)
  11Ø     YTDHRS = CVT$F(D$) \  YTDWRK = CVT$F(E$)
  12Ø     GROSS = CVT$F(C$)
  13Ø     GROSS= WRKHRS * HOUWAG
  14Ø     SUM = SUM + GROSS
  15Ø     SUM1 = SUM1 + YTDHRS
  16Ø     SUM2 = SUM2 + YTDWRK
  17Ø     LSET C$ = CVTF$(GROSS) \ LSET D$ = CVTF$(YTDHRS + WRKHRS)
  18Ø     LSET E$ = CVTF$(YTDWRK + GROSS)
  19Ø  NEXT I%
  2ØØ  PRINT TIME(Ø%) - A, SUM, SUM1, SUM2
  2Ø5  CLOSE #1
  21Ø  END
```

These results are found when running the above programs. The timing was done by using real time and not CPU time.

			Time (seconds.)		
O.S.	Lang.	File type	FIELD	MOV	MAP
RSTS\E V7.Ø	BP	SEQUENTIAL	-	-	-
RSTS\E V7.Ø	BP2	SEQUENTIAL	188	187.4	183
RSX-11M V3.2	BP2	SEQUENTIAL	268.8	189.6	182.4
VMS V2.Ø	VB	SEQUENTIAL	56	51	51

Notes: It is recomended that when using the FIELD statement,the file that is to be fielded should have a record size of 512. Mapping variables was con- sistently faster then moving variables from a dynamic buffer. The slowest method was FIELDing of variables from a buffer. The FIELD statements times would vary depending on the number of CVT's that were done.

The rest of this article is a step by step approach to transfer files across the different systems.

File Transfer RSTS/E TO VAX/VMS or RSX based systems

TERMINAL FORMAT FILES

1. Create DOS tape on RSTS with files and programs

   ```
   ASSIGN MMØ:.DOS
   
   PIP MMØ:(1,2)=*.*
   ```

2. Mount magnetic tape on VAX or RSX-11M+ using the foreign switch.

   ```
   MOUNT MTØ: /FOR
   ```

3. Using FLX, transfer files with the /RS switch on output and /DO switch on input.

   ```
   FLX>SY:/RS=MTØ:[1,2]*.*/DO
   ```

4. If the error message "FMTD ASCII RECORD FORMAT BAD" occurs, then use the /IM switch in FLX. This error should only occur when the terminal format file has a record greater then 512.

   ```
   FLX>SY:/RS=MTØ:[1,2]*.*/DO/IM
   ```

   To access these files a user must create a formatting program that reads the file as sequential fixed 512 and outputs it as a terminal format file.

VIRTUAL ARRAYS

1. Create DOS tape on RSTS with files in an account VAX can read.

   ```
   ASSIGN MMØ:.DOS PIP MMØ:(1,2)=*.*
   ```

2. Mount magnetic tape on VAX or RSX-11M+ using the foreign switch.

   ```
   MOUNT MTØ: /FOR
   ```

3. Using FLX, transfer files with the /RS switch on output and the /DO and /IM switches on input.

   ```
   FLX>SY:/RS=MTØ:[1,2]*.*/DO/IM
   ```

4. OPEN the file ORGANIZATION VIRTUAL.

RMS FILES

SEQUENTIAL (VARIABLE or FIXED), RELATIVE, and INDEXED FILES

Use RMS backup (RMSBCK) and RMS restore (RMSRST) utilities. See RMS-11 User's Guide, Sections 9.1 and 9.6.

If the error message "RST -- CLOSE ERROR ON FILE <filename>, ERROR CODE = 177760" occurs in RMSRST then use the switch "fR". See RMS-11 Users's Guide, Sections 9.1 and 9.6 for information on the switch.

BASIC-PLUS programs

1. Do the same as was done to Terminal Format files

2. If the program was written in NOEXTEND mode then run the FORMAT program, which can be found on the BP2 Re-build kit or the first floppy of the VAX BASIC kit. This will insure that if you run the program through the transla- tor on VMS, the program will be read correctly.

File Transfer RSX Systems to VAX

The only problem that could happen is if the file to be transfered is a RMS sequential stream. This is because that VAX-11 RMS does not have sequential stream files. To get around the problem used the RMSDEF in conjunction with RMSCNV, on VMS, to convert the files to sequential variable or write a BP2 program that reads the program as sequential stream and writes it out as se- quential variable file.

INCREASING BASIC-PLUS-2 COMPILER SPEED

This article is one of a series of articles on Basic Plus-2 V1.6. This article describes how to increase compilation speed of the Basic Plus-2 compiler by relocating the compilers work files. Any problems encountered with the information in this article should be reported via an FYI SPR only.

The Basic Plus-2 compiler uses two scratch files when compiling a program. One file contains the compiler hash tables, intermediate language, symbol table, etc. The second file contains a temporary copy of the source program being compiled. These files are normally opened in the current directory. Compile speed may be increased by placing these scratch files on a high speed disk such as an RS04 fixed head disk, or by placing them on a disk that is not highly used, thus avoiding disk contention for these files. The disk chosen must have a minimum number of free blocks according to the formula:

```
free blocks = (number of currently executing copies of the compiler)
               *(128. + (number of blocks of largest 'OLDed' program))
```

Example:

An RS04 fixed head disk holds a maximum of 2000 blocks (Files-11). Therefore, if the largest program to be 'OLDed' is 72 blocks, a maximum of ten (10) copies of the Basic Plus-2 compiler may be executing at any one time.

Since the Basic Plus-2 compiler is written in itself, these files are opened with a standard Basic Plus-2 'OPEN' statement. Redirecting the LUNS used for these files to a device other than SY: will not work unless the device contains a directory for every possible account the compiler will be run from. Although this method may be used to avoid protection problems described later in this article, it may not be feasible to create a directory for every possible account. A better way of moving these files is to specify a device and a directory. Every executing version of the compiler will then use this device and directory no matter what account it is running under. There is no conflict between the work files opened in the same directory because a unique file name based on terminal number (RSX) or job number (RSTS/E) is used when these files are opened.

On RSX based operating systems, these work files are opened as 'temporary'. There is no directory entry made for them and they are automatically deleted when the compiler task ends execution. When the disk

to be used is initalized, the correct protection (/PRO=[RWED,RWED,RWED,RWED]) must be used. This will enable the compiler running under any UIC to access this disk.

On RSTS/E systems, the RUN command uses these files to store vital compiler information before executing the user program. This data is then restored from these work files when the user program completes execution. Therefore, these files are not opened as temporary and they remain in the account when the compiler task exits. An exception to this is using the EXIT command to exit the compiler, which will cause these files to be deleted. These files are opened with 'logout' names, thus causing them to always be deleted when logging out if they are in the current account. If they have been opened in another directory or disk, they will not be deleted, possibly causing that disk to become cluttered with unused scratch files.

Due to the protection method used for non-privileged users on RSTS/E systems, patch number 3.5.7 (Allowing cross account creations) must be applied if the compiler is to be used from a non-privileged account. This patch will allow the work files to be opened in another directory provided it has the same project number as the account the compiler is running under.

In order to move these scratch files to another disk or directory, the compiler task image must be patched. The field that must be changed is 14 bytes long and contains a device and directory only. The starting address of this field is:

```
2:22030   on RSTS/E
2:33474   on RSX-11M/RSX-11M PLUS
3:33254   on IAS
```

Below is sample patching procedure for each of these systems. This patch will change the location of the compiler work files from SY: to DS0:[1,11]. These patches should be modified to reflect the device/account to be used for a particular installation.

For RSX-11M/RSX-11M PLUS:

```
RUN $ZAP
LB:[1,54]BASIC2.TSK
2:33474;0R              (address of INSPAR psect)
0,344"                  (offset into psect)
54523V                  (old contents SY)
51504                   (DS)
0,346"
20072V                  (old contents  :)
35060                   (0:)
0,350"
20040V
30533                   ([1)
0,352"
```

```
        20040V
        30454                   (,1)
        0,354"
        20040V
        56461                   (1])
        0,356"
        20040V
        20040                   (blank fill)
        0,360"
        20040V
        20040                   (blank fill)
        0,344"                  (check for correct device/account)
        X                       (exit ZAP)
```

For IAS:

```
        RUN LB:[11,1]ZAP
        LB:[11,1]BASIC2.TSK
        3:33254;0R              (address of INSPAR psect)
        0,344"                  (offset into psect)
        54523V                  (old contents SY)
        51504                   (DS)
        0,346"
        20072V                  (old contents  :)
        35060                   (0:)
        0,350"
        20040V
        30533                   ([1)
        0,352"
        20040V
        30454                   (,1)
        0,354"
        20040V
        56461                   (1])
        0,356"
        20040V
        20040                   (blank fill)
        0,360"
        20040V
        20040                   (blank fill)
        0,344"                  (check new device/account)
        X                       (exit ZAP)
```

For RSTS/E:

```
        RUN $ONLPAT
        Command file name? <LF>
        File to patch? LB:[1,2]BASIC2.TSK
        Base address? 2:22030
        Offset address? 344
         Base   Offset  Old     New?
        022030  000344  054523  ? 51504
        022030  000346  020072  ? 35060
```

```
        022030  000350  020040  ? 30533
        022030  000352  020040  ? 30454
        022030  000354  020040  ? 56461
        022030  000356  020040  ? 20040
        022030  000360  020040  ? 20040
        022030  000362  ??????  ? ^Z
        Offset address? ^Z
        Base address? ^Z
        File to patch? ^Z
```

Another optional patch which may be applied to the BASIC PLUS-2 compiler on RSTS/E will allow the compiler work file used for the hash and symbol tables to use the RSTS/E V7.0 Random Data Caching feature. This patch may be applied with or without the above patch to move the work files to another device and/or account. In order for the work file to be cached, the compiler must be run from a privileged account and Data Caching must have been selected durning the RSTS/E SYSGEN and must be enabled. If the compiler is not run from a privileged account, the file will not be cached and no error message will be produced. Please note that this patch is not available for any other operating systems.

```
        RUN $ONLPAT
        Command file name? <LF>
        File to patch? $BASIC2.TSK
        Base address? 36:35750-33150
        Offset address? 360
         Base   Offset  Old     New?
        002600  000360  000000  ? 23350+32
        002600  000362  000000  ? ^Z
        Offset address? ^Z
        Base address? 2:23350
        Offset address? 32
         Base   Offset  Old     New?
        023350  000032  000000  ? 400    (420 for contiguous and cached)
        023350  000034  000000  ? ^Z
        Offset address? ^Z
        Base address? ^Z
        File to patch? ^Z
```

A time trial was run using the above patches on a PDP-11/40 system with RK05J disks running RSTS/E V7.0. The VT5DPY CUSP was used and three cases were tried. The first case used a standard compiler without the above patches. The second case used a standard compiler with the first patch above applied to move the work files to a different disk and the compiler was moved off the system disk. The third case used a standard compiler with both patches above applied as well as flagging the compiler task image for sequential data caching and moving the compiler off the system disk. The system was used stand alone for the three cases. The results, as measured in wall clock time in seconds for OLD and COMPILE commands, were:

```
        Case 1  -  No Patches                          491
```

Case 2	-	Work files on a different disk	465
Case 3	-	Work files moved and Data Caching	434

A 1Ø% increase in compiler speed was realized from this time trial. On a system with faster disks and many more users, greater than 1Ø% speed increase may be realized.

PORTABILITY ISSUES

PDP-11 BASIC-PLUS-2 is currently distributed on RSTS/E, RSX-11M, RSX-11M+, IAS, TRAX, and compatibility mode on VMS. A great deal of effort has been made to make the language compatible across all systems, however, there are some features which are not compatible. For the most part, this is because these features take advantage of system capabilities which are unavailable on other systems.

If you must write applications to run on more than one system, or if you will be migrating from one system to another in the future, this article could prove helpful in providing a few hints on keeping your BASIC-PLUS-2 programs as transportable as possible. Some other suggestions are added to channel all BASIC programs being developed, towards a single, transportable BASIC, that will be compatible across operating systems, and with any future releases of BASIC.

First, a few general hints. If you do use system specific features (such as WINDOWSIZE, CLUSTERSIZE, RSTS/E SYS calls, calls to QIO routines), or "dying" features (such as FIELD and CVT), try to isolate them in subprograms. This will help to minimize change and prevent the entire program logic from being tied to these features.

For example:

instead of

```
10 A$ = SYS(CHR$(6%) + CHR$(22%) ...)
```

use

```
10 CALL PUTMSG(MESSAGE$,RECEIVER%)
```

Where PUTMSG handles the send/receive functions. In BASIC-PLUS you can isolate the SYS calls by placing them in the program at certain line numbers. This enables you to append other BASIC files to the program that override that line, replacing it with the code for the new system. To minimize the trouble in changing sources, designate certain line numbers of the program to always hold the incompatabilites. For example:

```
325Ø GOSUB 17Ø5Ø                    !INCOMPATIBLE OPEN
            .
            .
            .
17Ø5Ø       OPEN '(4,8)RSTS.DAT/FI:6' FOR INPUT AS FILE #1%, MODE
8%
17Ø51       RETURN
```

It is also convenient to make an object library of the system specific routines. Then you can extract the appropriate module for the target system at build time. If you are unsure whether a feature works the same across all systems, check your Language Reference Manual. The Language Reference Manual contains all the features and tells you which elements of the language are system specific. For more information on these system specific elements you should consult your User's Guide. Each system's User's Guide describes BASIC features that exist only on that system, or behave differently on that system. Also, be sure to read the BASIC Release Notes because they often contain important information and exceptions for each system.

Consider your future requirements carefully. Extra effort is required to create programs that are easily transported across systems, but this effort is trivial compared to that required to recode existing programs to be transportable.

The remainder of this article contains specific details on cross-system compatibility as well as recommendations for writing more readable and maintainable BASIC-PLUS-2 programs. Some of the suggestions may seem obvious, but they are included here to help you avoid poor programming habits. BASIC places no constraints on structure and style, therefore some effort is required to produce clear and concise code. Please keep the suggestions in mind as you code, they really can help.

When using services provided by the system, check to see if a BASIC feature will perform the same function. For example, the BASIC-PLUS-2 functions, CTRLO, CTRLC, RCTRLO, RCTRLC, NOECHO, ECHO, ONECHR, and statements, NAME AS, KILL, and SLEEP, all provide the same features on different operating systems. They can help you to avoid using system calls.

One of the more confusing compatibility issues is file I/O. There are three separate types of I/O to be addressed: terminal format files, device specific I/O, and RMS files. The next section of the article discusses the transportability aspects of these types of I/O.

TERMINAL FORMAT FILES

BASIC-PLUS-2 provides a type of file which treats I/O as if it were to a terminal. This is a terminal format file, and is accessed in BASIC-PLUS-2 with an OPEN statement with no ORGANIZATION clause specified. All I/O to terminal format files should be with PRINT, INPUT, and MAT I/O statements (except MAT READ). The files are sensitive to comma and semicolon formatting on PRINT statements. When terminal format files are to be used for INPUT, data should be formatted with commas, just as if the data were from the terminal. For example if you wish to say:

```
10      INPUT #1%, A,B,C
```

then your file should look like this

```
124,2786,.8
```

Therefore, to get this into the file using a BASIC program you must do this:

```
10      PRINT #1%, A; ','; B; ','; C
```

BASIC-PLUS-2 does not handle these files the same across all systems. The differences are discussed below.

On RSTS/E, all file and record operations for terminal format files are handled by the system. RMS is not used, therefore support is not linked into the task image for these files. However, on all other systems, terminal format files are implemented as sequential variable files, and all file and record I/O is done through RMS. Therefore, when using the BUILD command for BP2 programs using terminal format on non-RSTS/E systems, the "/SEQ" switch is required. To convert your RSTS/E stream files to so that thay can be recognized by an RSX (or VMS) based system, write a BASIC-PLUS-2 program to read the records in and then write them to a sequential variable RMS file. In fact, there is a short BP2 program on the rebuild kit, called FORMAT.B2S, which will do this for you.

Although terminal format files may be implemented as sequential variable files on some systems, you should never specify SEQUENTIAL VARIABLE in the OPEN statement if you really want a terminal format file. BASIC-PLUS-2 accesses the files differently.

The MODE clause in the OPEN statement is not fully supported on all systems. BP2, through RMS, supports 2 modes across all systems: MODE 16% and MODE 8192%. To write your OPEN statement transportably, use the BP2 keywords CONTIGUOUS and ACCESS READ, respectively for these MODES.

Programs using terminal format files are transportable to some extent but there are differences that should be known. TEMPORARY for terminal format files creates a file that is checked when closed. On RSTS/E, TEMPORARY for terminal format files is implemented as a TENTATIVE file (see the RSTS/E Programming Manual) and on RSX based systems, it is implemented as a TEMPORARY file (deleted on CLOSE). Note that with RMS files TEMPORARY is implemented the same across all systems (that is, the file is deleted when closed). On RSTS/E, you can write to a terminal format file when not at the end of the file (at your own risk!), however, on the other systems where BP2 uses RMS, you can write to the file only if you are at the end of the file.

Transportable:

```
10      OPEN 'X.X' FOR OUTPUT AS FILE #1%, &
        CONTIGUOUS
20      OPEN 'Y.Y' FOR INPUT  AS FILE #1%, &
        ACCESS READ
```

Non-transportable:

```
10        OPEN 'X.X' FOR OUTPUT AS FILE #1%, MODE 16%
20        OPEN 'Y.Y' FOR INPUT  AS FILE #1%, MODE 8192%
```

Device specific I/O is used for non-file structured I/O (for example, opening such devices as a terminal or paper tape reader). When opening a non-file structured device, BASIC-PLUS-2 passes all I/O directly to the operating system for processing with one exception: on RSX based systems, spooled devices (a device accessed through a queue manager), are processed through RMS, and programs accessing this device through BP2 must be built with RMS SEQUENTIAL support ("/SEQ"). In the general case, where BP2 uses the operating system for I/O, programmers should check their system manuals for how specific devices are handled.

Example:

```
10        OPEN 'LP:' FOR OUTPUT AS FILE #1%
```

requires RMS on RSX based systems if LP: is a spooled device.

```
20        OPEN 'PR:' FOR OUTPUT AS FILE #1%
```

I/O is handled by system.

RMS files and BASIC-PLUS-2 features that provide access to RMS are compatible, and should be used whenever transportability is crucial to your application.

Example:

```
10        OPEN 'BAR.FOO' FOR OUTPUT AS FILE #1%,    &
                  SEQUENTIAL, RECORDSIZE 132%,      &
                  ACCESS WRITE
20        OPEN 'FII.BAR' FOR INPUT AS FILE #1%,     &
                  RELATIVE, ACCESS MODIFY,          &
                  ALLOW NONE
```

These examples will perform the same across all systems.

Another recomendation for transportability is to use RSX file naming conventions. If you omit the version number, then these file names are accepted on RSTS/E also. Even nine character file names are allowable in the OPEN st-tement, although they are truncated to the first six on RSTS/E. On RSTS/E use "[1,3]" not "(1,3)"; RSX does not accept parentheses for brackets. Remember too, that RSTS/E uses decimal numbers for accounts, while RSX based systems use octal. This problem is easily avoided by using accounts which could exist on either system, but this could lead to some confusion when transferring files (be sure the files get put in the account you think they'll get put in).

Transportable:

```
10        OPEN 'DK0:[20,17]LONGFILES.NAM' FOR OUTPUT &
                  AS FILE #1%
```

Non-transportable

```
10        OPEN 'DK0:(20,17)FILNAM.DAT' FOR OUTPUT    &
                  AS FILE #1%
```

Avoid CHAINing when possible. Use subprograms to provide segmentation; they are better structured because there is only one entry and exit point. Subprograms allow more flexibility than CHAINing, because (1) you can call MACRO or BP2 subroutines, (2) you can pass up to eight paramaters to BP2 subprograms, and (3) there is no defined limit on the number of parameters that can be passed to MACRO routines. Through the use of COMMON, large amounts of data can be made available to subprograms. Subprograms are very transportable, whereas CHAINing has some significant differences across systems. CHAINing with a line number is not supported on RSX systems in the current release. On RSX and IAS the task you are CHAINing to must be installed. On IAS you must have real-time privileges (to execute RQST$), and no CHAINing is allowed in compatibility mode on VMS. If segmentation is desired, use subprograms; if CHAINing is a must, then CHAIN without line numbers.

Avoid default values and syntax. Each system may have different default values for good reasons and these can cause compatability problems. Data types should be explicit. Use integers for FOR loops and channel numbers. ERR and ERL return integer values and should be checked against integers to avoid unnecessary conversions. Never fall through your error traps into the end of a program. Exit error traps with RESUME. Never depend on the value of ERR, ERL, and ERN$ outside of an error handler. The following example illustrates some of these points.

Use

```
5         ON ERROR GOTO 19000
                  .
                  .
                  .
110       FOR I% = 1% TO 10%
115               INPUT #1%, A$
120               PRINT #2%, A$
130       NEXT I%
140               .
                  .
                  .
19000     IF ERR = 11% AND ERL = 115% THEN             &
                  RESUME 140                           &
          ELSE                                         &
                  ON ERROR GOTO 0
```

Not

```
5         ON ERROR GOTO 19000
                  .
                  .
                  .
110       FOR I% = 1 TO 10
```

```
115               INPUT #1, A$
120               PRINT #1, A$
130      NEXT I%
140               .
                  .
                  .
19000    IF ERR=11 AND ERL=115 THEN RESUME 140
```

Conversions can also be avoided by checking what data types functions return.

In PRINT and INPUT be sure to state the punctuation between variables explictly.

```
 Use
  10      INPUT 'three intgers'; A%,B%,C%
  20      PRINT 'Integer 1'; A%;  'Integer 2'; B%;    &
                                  'Integer 3'; C%
 Not
  10      INPUT 'three intgers'A%,B%,C%
  20      PRINT 'Integer 1'A%'Integer 2'B%'Integer 3'
```

Avoid other unintentional side effects. Close files explicitly as soon as you are done with them. This is a good practice on any system because, (1) closing files frees internal space which can then be used for buffers or string space, and (2) difficulties with files getting locked if the program exits abnormally can be avoided.

The BASIC language encourages haphazard program design. This does not mean you must code that way. You should not jump into or out of DEFs or into FOR loops. Eliminate the GOTOs that GOTO GOTOs. Use different names for integer, real, list and matrix variables. The statement

```
10      C = VAL(MID$(C$(C%,5%),C%(C%,2%),C%))
```

may work, but it is probably meaningless to any future maintainer. Have DEFs do their own error trapping. Use MAP and MOVE statements instead of FIELDs and the CVT functions; LINPUT instead of INPUT LINE followed by CVT$$ or EDIT$. Note that not all TIME functions are available everywhere. TIME(0%) which returns the clock time in seconds from midnight is supported on all systems but TIME(1%), TIME(2%), etc are not supported on the RSX based systems. However the DATE$() and TIME$() functions are supported across all systems; they return dd-mmm-yy and hh:mm AM/PM respectively.

Using a template as an outline for your BASIC programs can help in writing functional, readable code. By using the one provided in the back of your User's Guide, or developing your own, you encourage a program structure that is divided into functional sections. Limiting one statement to a line prevents "losing" statements that are imbedded.

```
20      AMNT%=12% \FL.TOT=OP1+OP2 \GOTO 40 &
```

```
                \TOT=VAL(SS$) \ B.TOT=0.0
```

Here the last two statements on line 20 are never executed.

Provided below is a template for function definitions. If the outline is followed, it will force the user to trap his/her own errors within the function itself as well as provide a description of what the function does. Programs are much easier to read when you know what to look for and where to look for it.

```
N     DEF FN....                                        &
\             ON ERROR GOTO N + 90                      &
              !                                         &
              !  Function description                   &
              !       ...                               &

N+1   Function Code ...                                 &

N+89  FNEXIT
N+90  ! Local function error handler                    &
      Error handler code...                             &
\             .                                         &
\             .                                         &
\             .                                         &
\     ON ERROR GOTO 0                                   &

N+99  FNEND
```

It is suggested that N (the start of your function definitions), begin at line 15100 and each new definition start at a line number which is a multiple of 100.

Always put data declaration statements (such as DIMENSION, COMMON, and MAP) before the referencing the variables contained in them. Note that the MAP, COMMON, and DIMENSION statements are NOT executable statements. Therefore:

```
20       IF STUDENT.RECORD% THEN GOSUB 10000
30          .
            .
            .
10000    MAP (ADDR) NUM%, STR.ADDRS$=32%, APT.NO$=4%, &
                        CITY$=18%
10005    DIM TO.TAL%(1500%), SUB.TOTAL%(4500%)
10010    COMMON (FIXSTR) OUT.LIN$=132%, SUB.LIN$=12%
10015    RETURN
```

makes no sense at all, in BASIC-PLUS-2. The MAP, arrays, and COMMON will be allocated regardless of STUDENT.RECORD% and nothing will be executed at run-time in the subroutine.

Initialize your variables before using them. This resolves any doubts a maintainer may have when supporting code he or she did not write. A manual update some time in the future will explain how to initialize variables in COMMONs and MAPs with no cost to execution time.

As BASIC continues to develop, a major goal is to provide cross-system functionality. In order to provide a single BASIC that is compatible across all systems, ambiguities will, from time to time, be removed from the language. To be sure that your programs do not take advantage of these ambiguities, specify options and values.

A future article will deal with those BASIC functions which can be used to increase the performance or functionality of BASIC on a particular system at the cost of transportability.

BASIC SIG DECUS U.S. CHAPTER

To Submit: remove form and return to:

BASIC SIG Editor
DECUS
MR2-3/E55
One Iron Way
Marlborough, MA 01752
USA

☐ **INPUT**
☐ **OUTPUT**
☐ **WAR STORY**
(check one)

A SIG Information Interchange

PLEASE REPRINT, THE FOLLOWING IN THE NEXT EDITION OF BASIC SIG

CAPTION: ____________________

MESSAGE: ____________________

____

____

____

____

____

____

____

____

CONTACT:

NAME ____________________

ADDRESS ____________________

____

TELEPHONE
IF THIS IS A REPLY TO A PREVIOUS I/O, WHICH NUMBER? ______
DATE
SIGNATURE ____________________

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

[redacted]

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.