BASIC SIG

THE NEWSLETTER FOR THE BASIC SPECIAL INTEREST GROUP

July 1982 **Vol. 3 No. 3**

BASIC-PLUS-2 DEVELOPMENT SUPPORT TEAM

PROGRAM SEGMENTATION IN BP2

Two types of segmentation exist in BASIC: the subprogram and CHAINing. When using subprograms, control passes from the main program to a subprogram within the same task. CHAINing passes control from one main program to an entirely different main program which is not contained in the task. BASIC-PLUS-2 allows subprograms to be written either in BASIC-PLUS-2 or MACRO-11.

Subprograms, like functions and subroutines, are ways for writing frequently used procedures. A subprogram also allows you to divide a large task into smaller, more manageable units. You compile subprograms separately from the main program and include them in the task through the BUILD command.

1.Ø BASIC-PLUS-2 SUBPROGRAMS

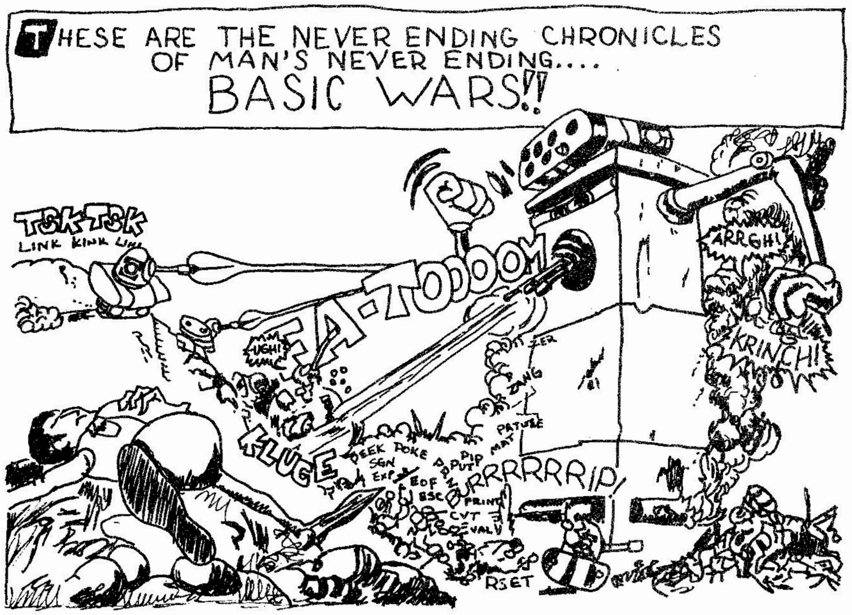
1.1 Transfer To And From

Transfer is passed from one program segment to another through the call statement. The call statement ha~ the format:

```
        CALL name [(argl,arg2,...arg8)]
```

where

name	is a unique one to six character name. Subprograms, MAPS, or COMMON statements cannot have the same name.
argl,arg2,...arg8	represent zero to eight arguments to be passed to the subprogram. These arguments will be referred to as the actual parameters.



Green Fungus & White Fuzz

by Clair W. Goldsmith
University of Texas Health Science Center

At the University of Texas Helth Science Center we had a DEC 20 that was having intermittent errors. Of course we couldn't find any reason for these errors, so we called those wonderful people from DEC. What tne heck, they built the thing, so they should be able to make it work, right.

The Data Center at the University of Texas Health Science Center is one story below ground level. The area had been prepared for a computer. The basement was dug out. A raised floor was installed. A false ceiling was put in. And all contact with outside air was sealed off, well almost all contact.

Those nice people from DEC tried everything they knew how to do to cause the same errors that the DEC 20 was having all by itself. Parts came out, parts went in until someone noticed something strange about a backplane. It was a funny color. It was a bit green. Could it be that when a computer gets sick it turns green.

It turns out that the green color came from a green fungus. It seems that not all outside air was kept from the computer. The fungus would grow on that backplane, make contact, cause an error and zap itself breaking contact.

Southwest Texas State was blessed with a deionzer. This deionzer had filters that had to be replaced by the crack maintenance crew of the University.

Continued page 36

The name used in the call can be either a quoted or unquoted string. The actual parameters must agree in type and number with the arguments used in the SUB statement of the subprogram. For example:

```
10      CALL 'LEED' (X,Y)

10      CALL LEED (X,Y)
```

Lines 10 above call the same subprogram. You cannot use string variables to call a subprogram.

```
05      NAM$ = 'SUBPGM'
10      CALL NAM$(X,Y)
```

In this case BASIC will try to call a routine with the name "NAM$" not a routine called "SUBPGM".

The CALL statement is valid in main programs, subprograms, and multi-line DEF's. Recursion is not allowed in the current version of BASIC-PLUS-2. When the CALL is executed, control transfers from the main program to the first line of the subprogram. This is the only entry point of a BASIC-PLUS-2 subprogram. The first line of a BP2 subprogram must be a SUB statement. The SUB statement has the form:

```
        SUB name [(arg1,arg2,...arg8)]
```

where

name	is the one to six character name used in the CALL statement. Again, note that SUBprograms, MAPs, or COMMONs cannot have the same name.
arg1,arg2,...arg8	represent zero to eight arguments passed to the subprogram by the CALLing program. These arguments will be referred to as the formal parameters.

The formal parameters must agree in number and type with the actual parameters.

Once control passes to the subprogram, execution continues until a SUBEND statement is encountered. The SUBEND statement marks the end of the SUBprogram. The format is:

```
        SUBEND
```

The SUBEND statement must be the last statement in the SUBprogram. When executed, SUBEND transfers control back to the calling program to the statement immediately following the CALL statement.

BASIC-PLUS-2 also has a SUBEXIT statement. SUBEXIT, like SUBEND is valid only in SUBprograms. When executed, SUBEXIT causes immediate transfer to the SUBEND of the current subprogram. The format of the SUBEXIT statement is:

```
        SUBEXIT
```

You can open and access files in the main program or the subprogram. Opened files remain opened until one of three things happens: the channel is closed by a CLOSE statement, another file is opened on the same channel, or the END statement in the main program is encountered.

The internal file pointer, which defines the current record, is the same for all modules. Thus each time you sequentially access the file, you get the next record, whether BASIC performs the access operation in the main program or a subprogram.

Main programs and subprograms can use the same variable names and line numbers. All variables are local to the subprogram unless they are:

- o Formal parameters
- o Elements of a COMMON or MAP

BASIC provides data communication between the calling program and the subprogram three different ways: passing parameters, COMMON and MAP areas, and virtual arrays (and files in general).

A function defined in a subprogram is local to that subprogram. You cannot use a variable name in the function definition if it is one of the SUB's formal parameters. For example:

```
  10    SUB SUBWDF(A$,B%,C)
  20          .
              .
              .
  15000 DEF FNDUM(A$,B%,C)
              .
              .
              .
  15020 FNEND
  32767 SUBEND
```

will result in an error message (?Illegal dummy argument at line 30 statement 1) at compile time because the parameters used in the definition of the function FNDUM are the same as the formal parameters passed to the subroutine.

Data statements are local to the subprogram also. READs in the subprogram do not affect the DATA pointer in the calling program. Each time the main program calls a subprogram, the data pointer returns to the beginning of the subprogram's data.

1.2 Data Communication

1.2.1 Parameters -

Paramaters are the arguments in the CALL and SUB statement. In the CALL statement they are referred to as the actual parameters. In the SUB statement they are referred to as the formal parameters. Parameters are handled by BP2 in two different ways.

The two ways that BASIC-PLUS-2 passes prarameters are by reference or by descriptor. If a parameter is passed by reference, its address is placed in the argument list. If it is passed by descriptor, the address of a descriptor is placed in the argument list. The descriptor contains the address of the storage assigned to the argument.

Parameters are of two types; modifiable and non-modifiable. For a modifiable parameter, the actual parameter is passed by one of the two mechanisms above. Thus, assigning a value to the parameter in the SUB changes the value of the argument in the main program. For non-modifiable parameters, a local copy is made of the parameter in the calling program. The local copy is then passed to the SUB by reference but the actual parameter in the calling program will never be modified.

BASIC treats several types of parameter as non-modifiable. They are constants, expressions, function calls (user defined as well as BP2 functions), and array elements. In the example below all parameters passed are non-modifiable. Notice that the actual parameters match the formal parameters in type and number.

Main program

```
10 DIM A$(5%),C(5%,5%)
20 FOR I% = 1% TO 5%              !INITIALIZE THE ARRAYS
30   A$(I%) = 'AAAA'              !INITIALIZE A$ TO BE 'AAAA'
50   FOR J% = 1% TO 5%            !INITIALIZE THE MATRIX TO BE 4.0
60      C(I%,J%) = 4.0
70   NEXT J%
80 NEXT I%
90 !    NOW PRINT THE VALUES OF PARAMTERS BEFORE THE CALL
100 PRINT A$(1%),C(1%,1%)
110 CALL SUB1(A$(1%),C(1%,1%))
120 PRINT A$(1%),C(1%,1%)
130 PRINT \PRINT 'THE SECOND CALL:'
140 PRINT A$(2%)+A$(2%), FNSQR(C(2%,2%))
150 CALL SUB1(A$(2%), FNSQR(C(2%,2%)))
170 PRINT A$(2%)+A$(2%), FNSQR(C(2%,2%))
1000 DEF FNSQR(DUMMY)
1010    FNSQR = DUMMY ** 2%
1020 FNEND
32767 END
```

Subprogram

```
10 SUB SUB1(STR.ING$, RE.AL)
20 PRINT 'THE VALUES OF THE FORMAL PARAMETERS ARE:'
30 PRINT 'STR.ING$ = '; STR.ING$
40 PRINT 'RE.AL  =  '; RE.AL
50 STR.ING$ = 'HI THERE'
55 RE.AL = 8.0
60 PRINT 'THE NEXT VALUES WILL BE PRINTED BY THE MAIN PROGRAM'
70 PRINT
32767   SUBEND
```

When executed gives:

```
AAAA           4
THE VALUES OF THE FORMAL PARAMETERS ARE:
STR.ING$ = AAAA
RE.AL   =   4
THE NEXT VALUES WILL BE PRINTED BY THE MAIN PROGRAM
```

```
AAAA           4

THE SECOND CALL:
AAAAAAAA       16
THE VALUES OF THE FORMAL PARAMETERS ARE:
STR.ING$ = AAAA
RE.AL   =   16
THE NEXT VALUES WILL BE PRINTED BY THE MAIN PROGRAM

AAAAAAAA       16
```

As you can see, the value of the array elements did not change during the call even though the values of the formal parameters that represent them did. This is what is meant by non-modifiable.

BASIC treats all other types of parameters as modifiable. This includes entire arrays of all types, simple string variables, simple integer variables, and simple real variables. These types of parameters may have their values changed while the subprogram is executing.

The way to pass entire arrays is to specify the name, parentheses, and a comma (if necessary), without the actual row (and column) number. For example:

Main program

```
  10     DIM A$(5%,5%), B%(2%)
  20     FOR I% = 1% TO 5%         !INITIALIZE A$
  30      FOR J% = 1% TO 5%
  40       A$(I%,J%) = 'AAA'
  50      NEXT J%
  60     NEXT I%
  70     B%(1%),B%(2%) = 32767%
  75     X$ = 'XXXXXXX'  \Y% = 9999%  \Z = 88.88
  80     PRINT 'HERE ARE THE VALUES BEFORE THE CALL:'
  90     MAT PRINT A$,  \PRINT
  100    PRINT B%(1%),B%(2%)
  105    PRINT X$,Y%,Z
  110    CALL BP2SUB(A$(,),5%,B%(),2%,X$,Y%,Z)
  120    PRINT \PRINT 'HERE ARE THE VALUES AFTER THE CALL:'
  130    MAT PRINT A$,   \PRINT
  140    PRINT B%(1%),B%(2%)
  150    PRINT X$,Y%,Z
  9000   END
```

Subprogram

```
  10     SUB BP2SUB(ARY$(,),D1%,INRY%(),D2%,STRG$,IN%,R)
  20     FOR I% = 1% TO D1%
  30       FOR J% = 1% TO D1%
  40             ARY$(I%,J%) = 'BBB'
  50       NEXT J%
  60     NEXT I%
  70     FOR I% = 1% TO D2%
  80       INRY%(I%) = -1%
  90     NEXT I%
  100    STRG$ = 'A STRING'
  110    IN% = 1234%
  120    R = 12E-4
  9000   SUBEND
```

Results in:

```
HERE ARE THE VALUES BEFORE THE CALL:
AAA            AAA            AAA            AAA            AAA

AAA            AAA            AAA            AAA            AAA

AAA            AAA            AAA            AAA            AAA

AAA            AAA            AAA            AAA            AAA

AAA            AAA            AAA            AAA            AAA

 32767          32767
XXXXXXX         9999           88.88

HERE ARE THE VALUES AFTER THE CALL:
BBB            BBB            BBB            BBB            BBB

BBB            BBB            BBB            BBB            BBB

BBB            BBB            BBB            BBB            BBB

BBB            BBB            BBB            BBB            BBB

BBB            BBB            BBB            BBB            BBB

-1             -1
A STRING        1234           .ØØ12
```

In the example above, BASIC passes the arrays A$ and B% by descriptor, the string X$ by descriptor and all others by reference. The constants 5% and 2% are the only parameters passed that are non-modifiable. The values of the arrays and simple variables are changed by the subprogram and the changes effect the actual parameters in the main program as well.

Be sure to note that the statements

```
1Ø      DIM A%(1ØØ%),B$(15%)
2Ø      CALL XXX(A%(1ØØ%),B$(15%))
```

result in only one element in each array being passed (non-modifiably) to the subprogram, not the whole arrays.

1.2.2 COMMON And MAPs -

COMMON and MAP is another method of data communication between the calling program and the subprogram. Special placement consideration must be given to these two statements when overlaying your programs. See the overlay section of this article for more information.

The COMMON and MAP statements define a named, shared area of memory called a COMMON block. This block contains values available for reading or changing by any BASIC subprogram with a COMMON or MAP of the same name.

The COMMON statement has the format:

```
COM[MON] [(name)] list
```

where:

name	can be from 1 to 6 characters long and must be different from any MAP in the same program module, or any subprogram names in the task.
list	specifies the variables whose values are stored in the COMMON area.

Variables stored in COMMON areas can be simple or subscripted. Simple numeric variables reserve fixed amounts of storage space:

- o Integers reserve 2 bytes
- o Floating-point numbers reserve 4 bytes for single precision systems or 8 bytes for double precision.

NOTE

Examples and explanations in this section assume single-precision.

String variables reserve fixed amounts of storage. Sixteen bytes is the default. Longer or shorter string lengths can be specified in the format:

```
{COMMON}
{ MAP }    str-variable-name = n%
```

where:

n%	is the number of bytes of storage you want the variable to reserve.

For example:

```
    Main Program                     Subprogram

1Ø COMMON(A1) A$,B$ = 1Ø%,C%    1Ø COMMON(A1) X$,Z$ = 1Ø%,Y%
```

Creates a COMMON block made up of:

o A 16 byte string field called A$ by the main program and X$ by the subprogram

- A 1Ø byte string field called B$ by the main program and Z$ by the subprogram
- A 2 byte integer field called C% by the main program and Y% by the subprogram

If the COMMON statement in the subprogram were:

```
    1Ø COMMON(A1) X$,Z$
```

the first variable, X$, references same storage as A$, because they are both 16-byte strings. The second variable, Z$, references the next 16 bytes of storage in the COMMON. Because the main program defined B$ as a 1Ø-byte string, the variable Z$ references these 1Ø bytes, plus the two bytes specified by the variable C%, plus the contents of the next four bytes of memory. This will result in Z$ containing garbage on entry into the subprogram.

Each element of a MAP or COMMON should start on a word boundary. When defining strings of odd lengths, you should add a " ,FILL$=1% " before defining any other variables, otherwise the compiler will generate an warning.

Areas in COMMON blocks can be subdivided. For example:

```
      Main Program

    1Ø COMMON A$ = 1Ø%,B%(1Ø%)

      Subprogram

    1Ø COMMON A$ = 5%, B$ = 5%, B1%(4%), B2%(5%)
```

In the main program, A$ is a 1Ø character string. In the subprogram, A$ is the first 5 characters and B$ is the next five characters of this same string. Arrays in COMMON allocate storage for row and column zero. Thus to access the array B% as B1% and B2% in the subprogram, you must account for the zero element of the array.

To align the variables correctly, use the FILL functions. They are place holders and do not place any values in the locations they hold. They move a pointer so that subsequent variables point to the correct values. See the FILL function in the file chapter of the Language Reference Manual for details.

BASIC dimensions arrays that appear in a COMMON statement. Therefore, you cannot also name them in a DIM statement. Specifying a DIM statement returns the compile-time warning:

```
    % Multiply allocated variable
```

A MAP is a fixed length area often used as a buffer for an I/O channel. It behaves much like a COMMON. The values in a MAP or COMMON are

available for access by any BASIC subprogram with a MAP or COMMON of the same name. Once a file is opened in a main or subprogram, any subprogram which defines a MAP of the same name can read data from it and place data in it. The subprogram can place data in the MAP either by performing a GET or by assigning values to the variables defined in the MAP.

There is a difference in the way BASIC allocates space for MAPs and COMMONs within a single program unit. BASIC concatenates storage for each COMMON area of the same name (that is places them end to end), while MAPs of the same name in a single program unit re-map the same storage (that is the MAPs ovedlay each other). The length of any MAP is the length of the longest single MAP statement with the same MAP name, while the length of a COMMON is the sum of the lengths of each individual COMMON. The order of elements in the COMMON list and the order of the COMMON statements, determines the order of values in the shared area. For example:

```
      Program with COMMON              Program with MAP

    1Ø COMMON(A) A$ = 1Ø             1Ø MAP(B) A$ = 1Ø
    2Ø COMMON(A) A%,B%,C%,D%,E%      2Ø MAP(B) A%,B%,C%,D%,E%
```

Both the MAP and COMMON statements reserve shared, named areas of memory. However, the COMMON statement program reserves a total of 2Ø bytes of storage: 1Ø bytes for strings, and 2 bytes for each of five integers. The MAP statement program reserves a total of 1Ø bytes: the 1Ø bytes for strings, and those same 1Ø bytes subdivided into 5 separately accessible 2-byte sections that the program can reference as integers.

The COMMON statement stores values accessed by different program modules. It also stores values that change from module to module. When a module changes the value in COMMON, all later references to that value return the changed value. For example, the following main program and subprogram access an array stored in a COMMON named ALPHA. The subprogram changes one of the elements in this array, and the main program then prints this element.

```
          MAIN PROGRAM

    1Ø COMMON(ALPHA) A%(5%,5%)
    2Ø FOR I% = 1% TO 5%
    3Ø      FOR J% = 1% TO 5%
    4Ø           Y% = Y% + 1%
    5Ø           A%(I%,J%) = Y%
    6Ø      NEXT J%
    7Ø NEXT I%
    8Ø MAT PRINT A%,
    9Ø PRINT "NOW TO THE SUBPROGRAM"
    1ØØ CALL SUB1
    1Ø5 PRINT \ PRINT "BACK IN MAIN PROGRAM"
    11Ø PRINT \ PRINT "CHANGED VALUE OF ELEMENT (3,3) IS ";A%(3%,3%)
    12Ø END
```

SUBPROGRAM

```
10 SUB SUB1
20 PRINT "IN SUBPROGRAM NOW"
30 COMMON(ALPHA) C%(5%,5%)
35 C%(3%,3%) = Ø
40 MAT PRINT C%,
60 SUBEND
```

The output is:

```
1              2              3              4              5
6              7              8              9              1Ø
11             12             13             14             15
16             17             18             19             2Ø
21             22             23             24             25
NOW TO THE SUBPROGRAM
IN SUBPROGRAM NOW
1              2              3              4              5
6              7              8              9              1Ø
11             12             Ø              14             15
16             17             18             19             2Ø
21             22             23             24             25

BACK IN MAIN PROGRAM

CHANGED VALUE OF ELEMENT (3,3) IS  Ø
```

1.2.3 VIRTUAL ARRAYS -

You can also pass data by DIMENSIONing virtual arrays in subprograms. DIGITAL strongly recommends that you do not pass virtual arrays as parameters. There is no need to do so because BASIC provides another means of access to virtual arrays.

To access a virtual array in a subprogram, use the DIMENSION statement with the same channel number used to define the virtual array in the main program. You do not need to open the file on that channel in the subprogram if the file is already open. You can also DIMENSION and open a virtual array in a suprogram, and then access this array in the calling program with two restrictions: (1) you cannot close the file before exiting the subprogram, and (2) you must DIMENSION the array in the main program on the same channel. You need not call the array by the same name in each program, and in any case, the virtual array must be opened BEFORE you access it. For example:

Main program

```
10     DIM #1%, A$(11%), A%(15%)
20     DIM #2%, B%(12%), B$(15%)
30     OPEN 'VIRFIL.DAT' FOR OUTPUT AS FILE #1%, VIRTUAL
```

```
40     A$(11%) = '11 12ST'
50     CALL VSUB1
60     B%(12%) = 12%
70     CLOSE  1%,  2%
32767 END
```

Subprogram

```
10     SUB VSUB1
20     DIM #1%, X$(11%), Z%(15%)
30     DIM #2%, CNT%(12%), ADR$(15%)
40     OPEN 'VFILE2.DAT' FOR INPUT AS FILE #2%, VIRTUAL
50     X$(3%) = ADR$(3%)
60     FOR I% = 1% TO 12%
70       Z%(I%) = CNT%(I%)
80     NEXT I%
32767 SUBEND
```

In this example, the main program cannot access the virtual arrays on channel 2 until control returns from the subprogram because the file is opened by the subprogram. The subprogram can access the arrays opened on channel 1 immediately because this file is already opened when VSUB1 is called. If VSUB1 had closed the file on channel 2 before returning to the main program, then the error '?Virtual array not yet opened' would be given at line 6Ø in the main program.

NOTE: Be sure that every virtual array you dimension is the same size and type as those in any other SUB or main program on the same channel.

1.3 Error Handling In BASIC Subprograms

Error handling in subprograms is similar to error handling in main programs. There are some error statements specifically related to subprograms, such as ON ERROR GO BACK and the ERN$ function.

The default error handler in the main program is ON ERROR GOTO Ø (i.e. all fatal errors abort execution). This is true for BASIC-PLUS-2 subprograms also. If there is an error handler in the main program, but none in the subprogram, and a fatal error occurs in the SUB, then execution will abort.

You may trap errors in a subprogram by using an error handler in the subprogram. ON ERROR GO BACK is a statement available in subprograms that is not available in main programs. This statement tells BASIC to return to the calling program's error handler when an error occurs in a subprogram. If this statement is executed when an error is pending then control will transfer immediately, otherwise the statement sets up the error handler for any future errors. You should explicitly state what you want your error handler to be in subprgrams, even if it is ON ERROR GOTO Ø. Future releases may change the default handler in subprograms to be ON ERROR GO BACK. By explicitly stating the type of error handling you want, you can avoid any compatability problems. The following example illustrates the use of ON ERROR GO BACK.

```
  10     SUB ERRTRP(X,Y)
  20     ON ERROR GOTO 19000
  30     INPUT 1%, G.LIN$                ! GET A LINE
                   .
                   .
                   .
  19000 PRINT ERR,ERL,ERN$
  19010 IF ERR = 11% AND ERL = 30% THEN &
                 RESUME 32767            &
        ELSE                             &
                 IF ERR = 51% THEN       &
                    ON ERROR GO BACK     &
                 ELSE                    &
                    ON ERROR GOTO 0
  32767 SUBEND
```

In this example, if the error is 11 and the error line is 30 then the error condition is cleared and the subprogram exits normally. If the error is 51 then control is transferred immediately to the error handler in the calling program with the error conditions still set. In all other cases, BASIC reports the fatal error and execution stops. ON ERROR GO BACK can appear anywhere in the SUB.

ON ERROR GO BACK allows you to do all of the error handling in the main routine no matter how deeply you nest your subprograms. If all subprograms in a task have ON ERROR GO BACK as their error handler then any errors will return to the error handler of the main routine. This is because when control returns from a subprogram, the calling program checks if an error is pending. If so, then that program's error handler is executed. If that error handler is ON ERROR GO BACK then control continues to return through the calling programs until one of two things happens: an error handler is executed that is not ON ERROR GO BACK, or no error handler is found. In the latter case BASIC reports the error and aborts execution. Example:

```
Main program
  10     ON ERROR GOTO 19000
  20     CALL ERSU1
  30     PRINT 'AFTER CALL.  SUCCESS'
  40     GOTO 32767
  19000 PRINT ERR,ERL,ERN$
  19010 RESUME 32767
  32767 END

Subprograms
  10     SUB ERSU1
  20     ON ERROR GO BACK
  30     CALL ERSU2
  40     PRINT 'SUCCESS AFTER CALL. IN ERSU1'
  32767 SUBEND

  10     SUB ERSU2
```

```
  20     ON ERROR GO BACK
  30     CALL ERSU3
  40     PRINT 'SUCCESS AFTER CALL. IN ERSU2'
  32767 SUBEND

  10     SUB ERSU3
  20     ON ERROR GO BACK
  30     A = VAL('12R')              !GENERATE ERROR
  32767 SUBEND
```

Gives the following as a result:

```
 52            30            ERSU3
```

Note that lines 40 in ERSU2 and ERSU1 do not get executed.

If you trap errors in subprograms be sure to RESUME before exiting the subprogram (unless you exit through ON ERROR GOTO 0 or ON ERROR GO BACK). If you just 'fall' through to the SUBEND statement, unexpected results can occur. It is always recommended that you RESUME from your error traps whether in the main program or a subprogram.

In applications where trapping errors is crucial, put the error handler on the same line number as the SUB statement. This will minimize the time between when the subprogram starts and when you can first trap an error. For control C trapping this is especially important. For example:

```
  10     SUB EXAMPL(DUM1)                      &
  \      ON ERROR GO BACK
                 . . .
```

2.0 MACRO SUBPROGRAMS

BASIC allows you to call subprograms written in MACRO as well as BASIC. This allows greater flexibility in sharing low level routines between several languages. The restrictions are listed in this section. Read the entire section before you start to write your subprogram.

2.1 Transfer To And From MACRO Subprograms

Transfer from a BASIC-PLUS-2 program to a MACRO subprogram is passed through the CALL statement. For MACRO subprograms there are two forms of the CALL. Their format is

```
        CALL name [(arg1,arg2,arg3,....)]
```

and

```
        CALL name BY REF (arg1,arg2,arg3,....)
```

where

name	is a unique 1 to 6 character name. Subprograms, MAPs, or COMMONs cannot have the same name. Name can appear as a quoted or unquoted string.
arg1,arg2,arg3,...	are the arguments from BASIC to the subprogram. These arguments will be referred to as the actual parameters.

BY REF affects the way data is passed and the way errors are handled in the MACRO subprogram. See the data communication section (sect 2.2) and error handling section (sect 2.3) for more details.

BASIC-PLUS-2 can call only those MACRO subprograms that use the PDP-11 standard R5 argument passing sequence as shown in Figure 4-1. There are other restrictions:

- o MACRO subprograms cannot perform I/O operations or execute monitor calls.
- o BASIC-PLUS-2 cannot call a FORTRAN subprogram.
- o A MACRO or FORTRAN subprogram cannot call BASIC-PLUS-2.

The MACRO instruction that returns control from the MACRO subprogram to the calling BASIC program is:

```
     RTS PC
```

where:

RTS is the "return from subprogram" instruction.

2.2 Data Communication.

2.2.1 Parameters -

The argument list of a CALL statement passes arguments from BASIC programs to MACRO subprograms. When the MACRO subprogram starts, register 5 (R5) contains the address of an argument list, as shown in Figure 4-1.

Figure 4-1: Argument List Format

```
R5----------------+
                  |
                  v
      ----------------------------------
      | UNDEFINED       |   NUMBER OF  |
      |                 |   ARGUMENTS  |
      ----------------------------------
      |     ADDRESS OF ARGUMENT 1      |
      |                                |
      ----------------------------------
      |     ADDRESS OF ARGUMENT 2      |
      |                                |
      ----------------------------------
                      .
                      .
                      .
      ----------------------------------
      |     ADDRESS OF ARGUMENT n      |
      |                                |
      ----------------------------------
```

A MACRO subprogram called by a BASIC-PLUS-2 program need not save registers. However, as is usual in MACRO code, register 6 (SP) must point to the same location on entry to, and exit from, the subprogram. Each "push" onto the stack must have a corresponding "pop" from the stack before the subprogram returns control to the BASIC-PLUS-2 program.

CALL BY REF affects only two types of parameters: simple string variables and entire arrays. CALL and CALL BY REF pass integer, real (single-precision), and double value (double-precision) arguments the same way, and string and array arguments in different ways. Refer to Appendix D of your Language Reference Manual for a description of data formats of strings and arrays.

The R5 argument list passes data formats as follows:

Integer	The R5 argument list contains the address of the word holding the integer value.
Real	The R5 argument list contains the address of the high-order word for the single-precision value.
Double	The R5 argument list contains the address of the high-order word for the double-precision value.
String	When CALL is used, the R5 argument list contains the address of a two word string header. The first word of this header is the address of the first byte in the string. The second word is the string's length in bytes.

When the program specifies a CALL BY REF, the R5 argument list contains the address of the first byte in the string; the string length is not available.

Array — When the program has a CALL, the R5 argument list contains the address of the second word in the array header. The array header contains subscript information and the address of the first array element.

When the program specifies CALL BY REF, the R5 argument list contains the address of the first element in the array; the array header is not available.

BASIC does not allow you to pass a string array in a CALL BY REF.

In MACRO subprograms, as in BASIC subprograms, BASIC still passes constants, expressions, function calls, and array elements as non-modifiable parameters. The addresses generated in the R5 list for non-modifiable arguments are the local-copy addresses in the calling program. The local-copy value can be changed by the MACRO routine, but this will never affect the actual parameters.

There are two other restrictions that should be noted.

- o You cannot pass virtual arrays to MACRO subprograms
- o MACRO subprograms cannot create strings nor change the length of existing strings.

To get around these restrictions, place strings and array elements in COMMON before passing them as parameters, then make updates and changes to the COMMON area known to the calling and subprogram (see section 2.2.2). On return to the calling program, you can move the updated COMMON variables into the virtual array or strings.

Consider the MACRO subprograms in Programs 1 and 2. Program 1 shows how to use CALL to pass arguments; Program 2 shows how to use CALL BY REF.

Program 1: CALL Statement

```
;               CALL INSRT(A$,B$,C%)
;
; INPUTS:       ARG1 = ADDRESS OF A$ STRING HEADER
;               ARG2 = ADDRESS OF B$ STRING HEADER
;               ARG3 = ADDRESS OF C%
;
; OUTPUTS:      C% =  Ø IF OPERATION WAS SUCCESSFUL
;                  = -1 IF OPERATION FAILED
;
; EFFECTS:      THIS subprogram OVERWRITES THE SUBSTRING B$ INTO THE
;               STRING A$ BEGINNING AT CHARACTER POSITION C%.
;               RETURNS  Ø IN C% IF THE OPERATION WAS SUCCESSFUL.
;               RETURNS -1 IN C% IF THE OPERATION FAILED.
```

```
;
INSRT::
        MOV     2(R5),RØ        ; RØ = ADDRESS OF A$ STRING HEADER
        MOV     4(R5),R1        ; R1 = ADDRESS OF B$ STRING HEADER
        MOV     @6(R5),R2       ; R2 = C%
        BLE     ERREX           ; BR TO ERROR IF C% <= Ø
        ADD     2(R1),R2        ; R2 = C% PLUS LENGTH OF B$
        DEC     R2              ; MAKE R2 A LENGTH
        CMP     R2,2(RØ)        ; WILL B$ FIT INTO A$ ?
        BGT     ERREX           ; BR TO ERROR IF B$ WON'T FIT INTO A$
        MOV     (RØ),RØ         ; RØ = ADDRESS OF A$
        MOV     @6(R5),R2       ; R2 = C%
        DEC     R2              ; R2 = C% MINUS ONE
        ADD     R2,RØ           ; RØ = ADDRESS OF 1ST CHAR TO BE REPLACED
        MOV     2(R1),R2        ; R2 = LENGTH OF B$
        BEQ     ERREX           ; BR TO ERROR IF LENGTH  OF B$ = Ø
        MOV     (R1),R1         ; R1 = ADDRESS OF B$
1$:     MOVB    (R1)+,(RØ)+     ;INSERT A CHARACTER INTO A$ FROM B$
        SOB     R2,1$
        CLR     @6(R5)          ; SET C% TO Ø (OPERATION SUCCESSFUL)
        RTS     PC

ERREX:  MOV     #-1,@6(R5)      ; SET C% TO -1 (OPERATION FAILED)
        RTS     PC
        .END
```

Program 2: CALL BY REF Statement

```
        .TITLE INSRT
;
;               CALL INSRT BY REF(A$,LEN(A$),B$,LEN(B$),C%)
;
; INPUTS:       ARG1 = ADDRESS OF A$
;               ARG2 = ADDRESS OF LENGTH OF A$
;               ARG3 = ADDRESS OF B$
;               ARG4 = ADDRESS OF LENGTH OF B$
;               ARG5 = ADDRESS OF C%
;
; OUTPUTS:      C% =  Ø IF OPERATION WAS SUCCESSFUL
;                  = -1 IF OPERATION FAILED
; EFFECTS:      THIS subprogram OVERWRITES THE SUBSTRING B$ INTO
;               THE STRING A$ BEGINNING AT CHARACTER POSITION C%.
;               RETURNS  Ø IN C% IF THE OPERATION WAS SUCCESSFUL.

;               RETURNS -1 IN C% IF THE OPERATION FAILED.
;
INSRT::
        MOV     @12(R5),R2      ; R2 = C%
        BLE     ERREX           ; BR TO ERROR IF C% <= Ø
        ADD     @1Ø(R5),R2      ; R2 = C% PLUS LENGTH OF B$
        CMP     R2,@4(R5)       ; WILL B$ FIT INTO A$ ?
        BGT     ERREX           ; BR TO ERROR IF B$ WON'T FIT INTO A$
        MOV     2(R5),RØ        ; RØ = ADDRESS OF A$
        MOV     @12(R5),R2       ; R2 = C%
        DEC     R2               ; R2 = C% MINUS ONE
        ADD     R2,RØ            ; RØ = ADDRESS OF A$ PLUS C%
        MOV     @1Ø(R5),R2       ; R2 = LENGTH OF B$
        BEQ     ERREX            ; BR TO ERROR IF LENGTH OF B$ = Ø
        MOV     6(R5),R1         ; R1 = ADDRESS OF B$
1$:     MOVB    (R1)+,(RØ)+      ;INSERT A CHARACTER INTO A$ FROM B$
        SOB     R2,1$
        CLR     @12(R5)          ; SET C% TO Ø (OPERATION SUCCESSFUL)
        RTS     PC

ERREX:  MOV     #-1,@12(R5)      ; SET C% TO -1 (OPERATION FAILED)
        RTS     PC
        .END
```

2.2.2 COMMON And MAPs -

BASIC allows you to access COMMONs and MAPs from subprograms written in MACRO. This enables you share large amounts of data between your BASIC programs and MACRO routines. Also you can use a MACRO program to initialize COMMON areas at program load time.

For each COMMON and MAP in the user program, BASIC generates a PSECT definition. The example below shows a piece of a BASIC program and the code generated by BASIC. Note the attributes that BASIC assigns to the PSECT.

```
SAMPLE.B2S
  1Ø  COMMON  (AAA) A$=6%, C%, D     ! 12. BYTE AREA
  2Ø  MAP     (BUF) ADRR$, SS.NO$=9%, FILL$=1%, AMNT ! 3Ø. BYTE AREA
```

Code generated

```
        .PSECT AAA,RW,D,GBL,REL,OVR
AAA:
        .PSECT BUF,RW,D,GBL,REL,OVR
BUF:
        .PSECT AAA
        .BLKW 6
        .PSECT BUF
        .BLKW 15
```

(numbers in the generated code are in decimal)

By defining a PSECT with the proper name and attributes you can access any COMMON or MAP from a MACRO subprogram.

By writing MACRO routines that define the same PSECT that is generated by the COMMON statement, you can pe-initialize COMMONs to whatever values you like, without executing a single statement. The values are simply put in the right place at load time.

Pre-defining your COMMON area can save time if the first thing your program does is assign values to variables in COMMON. This can also save space by eliminating code generated for the assignments, and storage allocated for constants that are used only once.

Suppose you have COMMON statements to hold data that is used by the main and subprograms. The data is constant and is used for such things as printing common error messages, checking maximum values and so on. Your main program might look something like this

```
1Ø      COMMON(FIXSTR)  OUT.STR$         = 1Ø%,  &
                        BAD.INFO$        = 24%,  &
                        ATLIN$           = 8%
2Ø      COMMON(FIXDAT)  MAXNUM%,                 &
                        MAXVAL,                  &
                        BADNUM%,                 &
                        FUN.STR$         = 6%
3Ø      OUT.STR$ =  "Output is "                 &
        BAD.INFO$ = "Bad information supplied"   &
        ATLIN$   =  " at line'
4Ø      MAXNUM%  = 1ØØ%                          &
        MAXVAL   = 2E6                           &
        BADNUM%  = -1%                           &
        FUN.STR$ = " FUNNY"
5Ø              ...
```

In this example seven statements are executed to initialize variables in COMMON. Each constant ("Output is", 1ØØ%, 2E6, etc) used is allocated storage that can not be recovered. If these constants are never used again this space is, in a sense, wasted. Now consider the same program with only lines 1Ø and 2Ø and a MACRO definition of the PSECTs which has the same effect as lines 3Ø and 4Ø, but requires no execution and generates no 'wasted space'.

```
1Ø      COMMON(FIXSTR)  OUT.STR$         = 1Ø%,  &
                        BAD.INFO$        = 24%,  &
                        ATLIN$           = 8%
2Ø      COMMON(FIXDAT)  MAXNUM%,                 &
                        MAXVAL,                  &
                        BADNUM%,                 &
                        FUN.STR$         = 6%
5Ø      ...

;       Macro routine to initialize FIXSTR and FIXDAT.
;
;       It should be linked with the main program like any other
;               object module.
;       Values take effect at load time.
;
;       Attributes for the PSECTs MUST be as shown.
;       They cannot be changed
;
        .ENABLE LC                              ; ENABLE LOWER CASE

        .PSECT  FIXSTR,RW,D,GBL,REL,OVR
        .ASCII   /Output is /                  ;OUT.STR$ len = 10
        .ASCII   /Bad information supplied/    ;BAD.INFO$ len= 24
        .ASCII   / at line/                    ;ATLIN$    len = 8

        .PSECT   FIXDAT,RW,D,GBL,REL,CVR
        .WORD    100.                          ; MAXNUM%
        .FLT2    2E6                           ; MAXVAL
        .WORD    -1                            ; BADNUM
        .ASCII   / FUNNY/                      ; FUN.STR$ LEN = 6

        .END
```

There are some pitfalls you should watch for.

o If you compile your BASIC program in double precsion ALL floating point numbers must be defined with .FLT4

o Check lengths of string and floating point numbers to be sure you line up data correctly

To include this routine in your task you can use the BUILD command and specify the routine as if it were a subprogram. Create the MACRO program with an editor then assemble it making sure there are no errors. The initialization can take place only once. If you change values of variables in COMMON, you cannot call this routine to re-initialize. It contains definitions, not executable code. Even if you add a subprogram to these PSECT definitions, the initialization takes place only once, regardless of how many times you call the routine.

To write MACRO subprograms that access COMMON you need to define the correct PSECTs as shown above. If you do not wish to initialize the COMMON then enter a ".BLKW n." where n is the number of words of the common you wish to reference. Using this method, the COMMON will have whatever values were last assigned by the CALLing program.

You will have to assign a label to each data item that you wish to reference. It is a good practice to make these names correspond to their names in the BASIC program. For an example we will use the program already started above.

```
10      COMMON(FIXSTR)  OUT.STR$         = 10%,
                        BAD.INFO$        = 24%,
                        ATLIN$           = 8%
20      COMMON(FIXDAT)  MAXNUM%,
                        MAXVAL,
                        BADNUM%,
                        FUN.STR$         = 6%
15      COMMON(DATUM)   M%,N%,X,Y
50      INPUT 'supply an integer to convert to octal'; M%
55      PRINT
60      PRINT 'Here are initial values of all COMMONs'
70      GOSUB 500
80      CALL SAMPLE               ! CALL THE SAMPLE MACRO SUB
85      IF N% <> 100% THEN PRINT BAD.INFO$; ATLIN$; 50
90      PRINT
```

```
95      PRINT 'The octal value of 328 is '; FUN.STR$
100     PRINT
110     PRINT 'Here are the values after the call'
120     GOSUB 500
130     GOTO 32767
500     PRINT OUT.STR$, BAD.INFO$, ATLIN$
510     PRINT MAXNUM%, MAXVAL, BADNUM%, FUN.STR$
520     PRINT M%,N%,X,Y
530     RETURN
32767   END


;
;       This is a sample routine it converts the first word of the
;       PSECT DATUM to an octal string and puts it in FUN.STR$.
;
;       There are no parameters, all data is shared through COMMON
;
        .ENABLE LC                                 ; ENABLE LOWER CASE

        .PSECT   FIXSTR,RW,D,GBL,REL,OVR
        .ASCII   /Output is /                      ;OUT.STR$ len = 10
        .ASCII   /Bad information supplied/        ;BAD.INFO$ len= 24
        .ASCII   / at line/                        ;ATLIN$    len = 8

        .PSECT   FIXDAT,RW,D,GBL,REL,OVR
        .WORD    100.                              ; MAXNUM%
        .FLT2    2E6                               ; MAXVAL
        .WORD    -1                                ; BADNUM
FUNST:  .ASCII   / FUNNY/                          ; FUN.STR$ LEN = 6
        .PSECT   DATUM,RW,D,GBL,REL,OVR
M:      .WORD    0                                 ; M%
N:      .WORD    0                                 ; N%
X:      .FLT2    0                                 ; X
Y:      .FLT2    0                                 ; Y

        .PSECT
SAMPLE::
        MOV      #FUNST,R0                     ; GET ADR OF STRING RESULT
        MOV      M,R1                          ; PICK UP INTEGER
        BGE      3$                            ; SKIP IF NEGATIVE
        MOVB     #'1,(R0)+                     ; SET HIGHEST BIT AND CONTINUE
        BR       2$                            ; SKIP
3$:     MOVB     #'0,(R0)+                     ; MAKE HIGH ORDER CHAR 0
2$:     MOV       -12.,R2                      ; START TO DIVIDE AT 8**4
1$:     MOV      R1,R3                         ; COPY NUMBER
        ASH      R2,R3                         ; SHIFT R3 BY R2
        BIC      #177770,R3                    ; GET LOW ORDER THREE BYTES
        ADD      #60,R3                        ; MAKE ASCII
        MOVB     R3,(R0)+                      ; MOV CHARACTER
        ADD      #3,R2                         ; DECREMENT DIVISOR
        BLE      1$                            ; CONTINUE
        MOV      #100.,N                       ; SHOW SUCCESS
        RETURN
        .END
```

Here is the output from the program.

```
Here are initial values of all COMMONs
Output is      Bad information supplied     at line
 100            .2E 07         -1            FUNNY
-2              0              0             0

The octal value of-2 is 177776

Here are the values after the call
Output is      Bad information supplied     at line
 100            .2E 07         -1           177776
-2              100            0             0
```

2.3 Error Handling

Errors are trapped in MACRO subprograms just as they are in BASIC subprograms. However, there is no mechanism for the user to trap the error in the subprogram itself; this must be done in tne calling program. If the calling program contains an error handler then any MACRO subprograms called will have their errors trapped in the calling program. The ERR function is set as usual. The ERL function contains the line of the CALL to the subprogram. If you CALLed BY REF then the ERN$ function contains the name of the calling program. If executed a CALL without BY REF then the ERN$ function contains the name of the subprogram you called.

If you get errors such as "?Memory Management violation" or "?Odd Address Trap", check that you are accessing the parameters correctly. Read the following section on overlays if you are writing MACRO routines. It contains general information on how BASIC allocates space and generates code as well as information on how to correctly overlay your programs.

2.4 OTHER LANGUAGES

BASIC-PLUS-2 can call subprograms written in COBOL V4.0, however BASIC and COBOL do not have the same types of data representation so it is up to the user to convert data as necessary. There is no support for performing I/O in any subprogram that BASIC-PLUS-2 calls unless that subprogram is written in BASIC also.

BASIC-PLUS-2 cannot call subprograms written in any other language (i.e. FORTRAN or FORTRAN IV+ etc.). At the current time no languages can call a BASIC-PLUS-2 subprogram.

3.0 OVERLAY CONSIDERATIONS

3.1 BASIC-PLUS-2 Modules

When designing your BASIC-PLUS-2 application, it would be a great help to know just how large the task is or how much overlaying it will require. Since this is usually impossible, you should plan for the overlay stage of development in general ways. By designing your application in functional, modular segments you can make overlaying as painless as possible.

3.1.1 Function -

By keeping your program segments small and functional you allow yourself greater flexibility when it comes time to overlay. You can also put off designing the overlay until your program is working correctly by testing each small module individually. Small modular segments keep the overlay from forcing a design on the program. Instead, you should design the program and then work the overlay into the design.

Many segments result in more possible overlays and more choices to make, but this is better than realizing you must recode a segment into two or three smaller ones to make the overlay structure work. When it does come to overlay time, you will be able to put in the root the commonly used modules, without including initialization code and other routines that execute only once. Put these seldomly executed routines in an overlay where they will not waste space. Small segments also allow you to complete the work in stages and test at each stage.

3.1.2 Testing -

Testing programs that are laden with overlays is difficult. It can be avoided to some extent by testing at intermediate stages. Test each subprogram by writing small drivers that feed the subprograms the same range of data that the caller normally would.

Once convinced that your code is producing expected results, start to overlay in stages. Work out the entire design before by using maps. This lets you get approximate sizes of object modules. As you build the overlay, stop at logical points to test. Be sure that you know where global symbols will be resolved (see section 3.3) and when overlays will be brought into core at run-time. If you get errors such as "?Odd Address Trap" or "Memory Management violation", check COMMON and MAP statements for alignment, and be sure that parameters being passed match in number and type.

Check calling sequences to see if overlays are inadvertently being brought into memory. Returning from a subprogram into the wrong overlay can be done easily. How this can happen is discussed in section 3.3. Prove that errors are not due to the overlay structure by stringing out the program.

The following sections on BASIC PSECTs and threaded code will give a better understanding of how the generated code is affected by overlays.

3.2 Compiler Outputs

The compiler generates a kind of object code known as threaded code. To get a better look at what type of code this is, compile a program into MACRO (i.e. COM/MAC). Use the generated MACRO code for reference as you read this section. The code is made up of a series of PSECTs and global symbols called threads. Compiling into object code (COM/OBJ) produces the same code as MACRO in its object form.

3.2.1 PSECTS -

BASIC generates a number of PSECTs for each main and subprogram it compiles. Some of them and their functions are mentioned here. Those mentioned are included in each program, though some may be empty depending on your code.

BASIC allocates PSECTs with certain attributes. These should never be changed. BASIC assigns the overlay attribute to all COMMONs and MAPs, and two PSECTs that BASIC uses for I/O: $ICIO1 and $ICIO2. All other PSECTs generated by BASIC have the concatenate attribute. This is important in understanding how BASIC resolves addresses in the main programs and subprograms.

When subprograms are used, you may see a reference in the main program's threaded code to the same offset of the same PSECT in the subprogram's threaded code. Example:

```
Main program
        MOF$MS  ,$PDATA+16      ; #12
Subprogram
        MOS$MP  ,$PDATA+16      ; "ZZZZ"
                 $IDATA+4       ; Z$
```

This is a typical line of threaded code. The MOF$MS is the thread to move a floating point number from memory to the stack. $PDATA+16 is an address which the thread will use. The comment tells what variable or constant the address refers to. In the example above, both the main and subprogram refer to $PDATA+16 but have different values for what the address refers to. This does not mean that the same location is used for both a string and a floating point number. In fact, the subprogram thread moves a string

constant from memory ($PDATA+16), through a pointer, to some other address.

How can a main program and a subprogram both refer to $PDATA+16? When the task builder is linking the object code, it checks the attributes of PSECTs as it allocates space. The attributes referred to are RW,I, LCL,REL,CON. The concatenate attribute (CON) forces the task builder to concatenate PSECTs of the same name that are supplied by different object modules. Thus, when the task builder see a definition to $PDATA in the main program and a definition to $PDATA in a subprogram, it concatenates the two by adding the length of the first definition to all references in the subprogram.

The PSECT $CODE contains the BP2 threaded code. More will be said about threaded code in the next section. $IDATA is the PSECT from which real and integer variables are allocated. $IDATA, in the main program, is also where BASIC's work area is allocated, therefore all references to $IDATA in the threaded code of the main program will usually start at 800; the first 800 bytes being used for the work area. BASIC does not use static allocation of work space in subprograms.

$STRNG is the PSECT from which string headers are allocated. See appendix D of the Language Reference Manual for a decription of string headers.

$PDATA is the PSECT from which all constants (string, real, and integer) are allocated. Also, some (but not all) array headers are allocated in this PSECT along with the contents of all DATA statements. A description of array headers is given in appendix D of the Language Reference Manual.

$TDATA and $ARRAY are used for array headers. The array headers included in these PSECTs are those that can be redimensioned. $SAVSP is a PSECT to hold initial value of the stack. It is allocated only in the main program.

The PSECTs $ICIO1 and $ICIO2, and those PSECTs generated by BASIC for COMMONs and MAPs are the only PSECTs generated by BASIC that have the overlay attribute (OVR). This means that references to $ICIO1 and $ICIO2 in subprograms are referencing the same locations that the main program does. Also, it is this overlay attribute that allows data in COMMON and MAPs to be shared across program segments.

COMMONs and MAPs generate PSECTs using the name you specify. Thus when a COMMON or MAP is used in a main and subprogram the PSECT generated by the subprogram overlays the one generated by the main program.

3.2.2 Threaded Code -

Threaded code is a series of global symbols and addresses. These symbols are names of routines that contain code to execute user programs. The global symbol may optionally have arguments following it. These arguments are picked up by the routine and used to reference data from the user program. All threaded code is placed in the concatenated PSECT $CODE. For example the line of code:

```
20      PRINT A%
```

generates

```
L20:    LIN$    ,20             ; #20
        CLI$S
        IPT$
        MOI$MS  ,$IDATA+804     ; A%
        PVI$SI  ,0              ; #0
        EOL$
```

L20: is a label that signifies the start of each line. LIN$ is a thread the line number. The threads are generally mnemonic in nature. The fourth character of a thread is always a dollar sign and two optional characters follow. The comment section, when present, provides a description of the argument. Other articles in this series will provide a more thorough description of the threads.

At task build time the task builder resolves all of the threads (global symbols) to a shared library if present or the BASIC object library. To start execution of threaded code at run-time the compiler generates one executable instruction:

```
        JSR     R4,@#$INITM
```

for main programs, and

```
        JSR     R4,@#$INITS
```

for BASIC-PLUS-2 subprograms.

This is the first instruction executed. In both cases the instruction causes a jump to an initializing routine of BASIC. This has the effect of making R4 point to the first argument. Each routine uses all of the arguments and leaves R4 pointing to the next thread. When done, the current routine exits by doing a JMP @(R4)+ , and the cycle begins again. This continues until BASIC executes the END$ thread which exits the threaded code and returns control to the system.

For further discussion of threaded code see chapter 15 of "Computer Engineering" by BELL, MUDGE, and McNAMARA. This chapter is an article by R.F. BRENDER entitled "Turning Cousins Into Sisters".

3.3 Task Build Address Resolution

This section describes how BASIC-PLUS-2 global symbol references are resolved at task build time. It is assumed that the reader has read the ODL chapter of the Task Builder Refernce Manual. You should be familiar with the tree description of a task as well as terms such as path, path loading, and task segment. This section discusses specifics of BASIC-PLUS-2 and is not meant as a general tutorial for overlaying any

type of program.

The global symbols that BASIC generates and that the task builder must resolve are thread names and names of subprograms. Each call to a subprogram generates an unresolved global symbol which is the name of the subprogram.

First, an example to show how references are generated and resolved in a nonoverlayed task. Below is a list of the calling sequence of a BASIC program. Also included are the CMD and ODL files generated by a BUILD command on an RSX system. They will differ slightly for each system.

```
Main program          SUB2          SUB1 and SUB3
        CALL SUB1     CALL SUB1       no calls
        CALL SUB2     CALL SUB3
        CALL SUB3
```

```
BUILD MAIN,SUB1,SUB2,SUB3

SY:MAIN/CP/FP,SY:MAIN/-SP=SY:MAIN/MP
LIBR=BASIC2:RO
UNITS = 14
ASG = TI:13
ASG = SY:5:6:7:8:9:10:11:12
EXTTSK= 512
//

        .ROOT USER
USER:   .FCTR SY:MAIN-SUB1-SUB2-SUB3-LIBR
LIBR:   .FCTR LB:[1,1]BASIC2/LB
        .END
```

Notice that a resident library is used. Always check your CMD file to see if you are linked to a resident library (LIBR=BASIC2:RO or RESLIB=BASICS/RO) or a runtime system (HISEG=BASIC2). This is very important in understanding where global symbols will be resolved. From the ODL we can see that all of the BASIC routines are concatenated (i.e. there is no overlaying). The following is a list of the order in which the task builder will search through the object modules to resolve undefined global references. The order holds for overlaid as well as non-overlayed tasks.

1. The segment being processed
2. All segments on the sam_ branch towards the root (including the root itself)
3. All segments on the same branch away from the root
4. All cotrees
5. Any specified object library

In the above example, each object module of BASIC will generate threads that are global symbols for the task builder to resolve. The module MAIN.OBJ will also have three other global references to resolve, namely

SUB1, SUB2, and SUB3. The object module SUB2 will generate two additional global symbols to resolve: SUB1 and SUB2.

Each BASIC object module defines the global symbol by which it is named. SUB1 will define the global symbol SUB1 and so forth.

The task builder will find a definition for a thread in one of two possible places: a resident library (or runtime system), or the disk object library. Note that all resident libraries and runtime systems are given virtual address space from the root. The task builder tries to resolve threads in resident libraries and run time systems before searching the BASIC object libraries.

Using the above example and the scheme for resolving global symbols we see that references to SUB1, SUB2, and SUB3 will all be resolved in the object code produced by BASIC. The threads will be resolved either in the resident library or the disk object library BASIC2. The resident library is searched first, and all threads still undefined are resolved in the disk library. BASIC provides a library with every thread name defined in it. Therefore, if you do not link to a resident library (or run-time system) all global references will be resolved in the disk library.

With overlayed code this exercise in deciding where global symbols (especially your subprogram references) will be defined is very important and worth further dicussion. It is easy to get UNDEFINED or AMBIGUOUSLY DEFINED error messages from the task builder if care is not taken to overlay your subprograms correctly

Consider the following example of a BASIC program and its overlay tree and decription. The BASIC programs contain only calls to simplify the problem.

```
A.B2S                  B.B2S                    C2.B2S
 1Ø      CALL B          1Ø  CALL D1               1Ø     CALL D1
 2Ø      CALL C1         2Ø  CALL C1               2Ø     CALL D2
 3Ø      CALL C2                                   3Ø     CALL D3
```

C1.B2S, D1.B2S, D2.B2S, and D3.B2S make no calls.

```
         .ROOT USER
USER:    .FCTR SY:A-B-LIBR-*(BR1,BR2)
BR1:     .FCTR SY:C1-LIBR
BR2:     .FCTR SY:C2-LIBR-*(BR3,BR4,BR5)
BR3:     .FCTR SY:D1-LIBR
BR4:     .FCTR SY:D2-LIBR
BR5:     .FCTR SY:D3-LIBR
LIBR:    .FCTR LB:[1,1]BASIC2/LB
         .END
```

Note that each object module is concatenated with the object library search default. This means that for any global symbols not found on the paths or in the root, the object library will be searched. Only the object modules needed are retreived from the library. You should always concatenate any BASIC-PLUs-2 object modules with BASIC's disk library. This is done by adding the "-LIBR" to the BASIC modules before any overlays are specified.

Below is the tree structure for task as generated by the ODL above.

```
                                   D1        D2        D3
                                   !         !         !
                                   !         !         !
                                   -------------------
                                             !
        C1                                   C2
        !                                    !
        !                                    !
        --------------------------------------
                          !
                          B
                          A
```

Module A is the BASIC main program. All other modules could be BASIC or MACRO subprograms. There are 2 object modules in the root: A and B. Execution starts in A which calls B. B then calls D1. This is legal since there is a path between B and D1. This call brings in the module C2 even though it is not referenced. Therefore at this point in the execution the modules A, B, C2, and D1 are all in memory.

D1 returns to B which then calls C1. This brings in the C1 segment on top of C2 and D1. C1 processes data then returns to B, which returns to A.

A now calls C1 which is still in memory, therefore no overlays are brought in. Again C1 executes and this time returns to A. A calls C2, which brings in the C2 overlay. Now C2 calls D1, D2, and D3 in succesion, forcing a new overlay to be brought in for each call. When D3 finishes, control returns to C2, which finishes and then returns to A where processing stops.

Another way of looking at the task is shown below.

```
                        __________
|        |_______|  |_____________|
|        |   | D |  |
|--------| D | 2 |  D3 |
|        | 1 |   |     |
|        |   |   |     |
|   C1   |-------------|
|        |             |
|        |     C2      |
|        |             |
|________|______|______|____________________|
|        B             |
|______________________|____________________|
|        A             |
|______________________|____________________|
```

Looking at the task in this diagram allows you to see two things. By drawing a vertical line through the diagram at different points you can see exactly what can be in core at any given time. You can also see how big the task is at any given time. The highest horizontal line that crosses any vertical line (the time slice line) indicates how large the task is. If you have task built the program once, you can look at the map to get the actual sizes of the segments. Remember, however, that opening files and creating strings causes dynamic changes in the size of your task.

It is a good idea to trace through the actual runtime control, as was done here, to prove that the overlay structure will in fact work. Be aware that not all tasks that get task built without warnings or errors are properly overlayed. Consider the same overlay structure with the calling sequence as follows:

```
A.B2S                              B.B2S           C2.B2S
 10     CALL C1                    10 CALL C1      10       CALL B
 20     CALL C2                                    20       CALL D1
                                                   30       CALL D2
                                                   40       CALL D3
```

This task will task build without any error or warnings but on tracing through the control you can see that the program will abort with an error.

A will call C1 bringing in an overlay. C1 executes and returns to A, which then calls C2. C2 now calls B. This is legal since any module can call any other which lay on the same path.

B now calls C1 bringing in the C1 overlay on top of the C2 overlay. C1 finishes executing and returns to B. At this point the C1 overlay is still in memory where C2 was when it called B. When B returns it will try to return to C2. However C1 is still in C2's place. At this point you will usually get a "?Memory Management violation" or "Odd address trap". The reason is that when B executes the return to C2 it returns into some section of C1 which may not even be code.

Overlays are brought into your address space by auto-load vectors which are generated by CALL statements, but it is up to the user to be sure that the correct overlay is in memory when the called segment tries to return.

Another important point to consider is the allocation of MAPs and COMMONs in overlay sections. As was noted earlier, MAPs and COMMONs are generated into PSECT definitions by BASIC. The PSECTs have attributes that will cause any references to a COMMON or MAP of the same name in an overlay, to be allocated the same storage as any other COMMON or MAP on the same path below it. This is how the data is shared between the subprograms that reference the same COMMON or MAP. For example:

```
A.B2S
  10    COMMON (MST) A$,B$

B.B2S
  10    COMMON (MST) X$ = 30%, Y$ = 2%

C.B2S
  10    COMMON (MST) M$ = 32%
```

with the overlay structure

```
        B                        C
        |                        |
        |                        |
        --------------------------
                     |
                     A
```

In this case, all three modules use the exact same space allocated for the PSECT MST. It is allocated in the root and no space is allocated in the overlay.

Now consider the case where A does not define the COMMON MST. The task builder will allocate MST from EACH overlay. Thus, whenever an overlay is brought in, its own copy of MST will be brought in. Remember, overlays are not written back out to the disk. Any changes to MST just disappear when a new overlay is brought in. This new overlay has its own copy of MST with values unrelated to the MST across the tree. PSECTs with the overlay attribute are overlayed only if they are on the same path.

Any time COMMONs or MAPs are not allocated in the root, data contained in them will not be valid once a new overlay is brought in. Therefore, COMMONs and MAPs which are not defined in the root should only be used to hold temporary data that is needed only while the first overlay to define the COMMON or MAP remains in your address space.

When using co-trees (an overlay structure of more than one tree which does not share storage among trees; see TKB manual), be sure that you know ahead of time where ALL global symbols (BASIC's as well as your own) will be resolved. Notice that co-trees are searched BEFORE the default object library. This may lead to the resolution of BASIC threads (global symbols) in overlays of co-trees. The unwary user can call into a co-tree from overlay X. This co-tree has symbols defined in overlay Y which overlays overlay X. Execution continues until the cotree tries to return to overlay X, which is no longer in memory. At this point your task will probably abort with "?Odd Address Trap", or "?memory Management violation". Co-trees should be approached with caution and only when necessary.

3.4 Run-time Happenings

When a call is executed by a BASIC program, several things take place. BASIC saves the current values of error handlers and data pointers so that the context of the calling program can be restored on return. Arguments are pushed on to the argument list stack. R5 is made to point to the argument list and then the actual call is made.

CALL BY REF does not save the current context of error handlers and data pointers because this type of call can be used only for MACRO routines, which cannot change these pointers and data. It is for this reason that the ERN$ function does not return the name of the MACRO subprogram when an error occurs there. The name of the called module is not saved anywhere as it is for regular calls.

Calls to subprograms in a different overlay generate autoload vectors at task build time. When the actual call is made the overlay is brought into memory. No checking is done to be sure that the correct overlays are still in memory on return. It is impossible for the task builder to know to which routine you are returning, therefore returns do not generate autoload vectors. How this can lead to problems was discussed under section 3.3.

When a return is executed, (1) all the context information is recovered, (2) local strings created by the subprogram are destroyed, and (3) the space returned to free space. Again, a CALL BY REF does not need to restore its current context because it never changed in the first place. Also, because no strings can be changed or created by MACRO routines, no local strings need to be cleaned up.

4.Ø Calling keywords

This section will reveiw the different types of calling subprograms and parameter passing. A list of the VAX-11 BASIC calling conventions is included in the table below so that you can see the differences and the added flexibility of that implementation. First a quick review of the terms involved.

To pass a parameter by immediate value means the value of the actual parameter is passed to the formal parameter. No update is made to the actual parameter during the subprogram or on return. It is not possible in the current version of BASIC-PLUS-2 to pass any parameters by value.

To pass a parameter by reference means to pass the address of the actual parameter to the formal parameter. The value of the actual parameter is updated when ever the formal parameter is because they point to the same value. BASIC presently uses this method for every type of parameter passed except two: strings and entire arrays.

To pass a parameter by descriptor means to pass the address of a descriptor of the parameter to the subroutine. This descriptor contains information that enables you to reference the actual parameter. This type of parameter passing is normally used when the actual parameter is not a

single element such as an array. BASIC currently uses this method to pass strings and entire arrays.

The table below includes all of the calling conventions for VAX-11 BASIC and PDP-11 BASIC. On VAX-11 BASIC the terms BY VALUE, BY REF, BY DESC are keywords that can be applied to the call or to each parameter individually to allow you to specify the method of parameter passing. On PDP-11 BASIC these terms are used only to explain how parameters are passed. Only in one case can you specify BY REF in PDP-11 BASIC (when calling MACRO subprograms), and this must be applied to the entire call. It affects only two types of parameters and has other side effects that were mentioned in the MACRO subprogram section of this article.

	VAX-11 BASIC			PDP-11 BASIC		
PARAMETERS	BY VALUE	BY REF	BY DESC	by value	by ref	by desc
NUMERIC DATA						
Variables	YES	*YES	YES	NO	*YES	NO
Constants	YES	*Local copy	Local copy	NO	*Local copy	NO
Expressions	YES	*Local copy	Local copy	NO	*Local copy	NO
non-virtual array elements	YES	*YES	YES	NO	*Local copy	NO
Virtual array elements	YES	*Local copy	Local copy	NO	*Local copy	NO
Non-virtual entire array	NO	YES	*YES	NO	++YES	*YES
Virtual entire array	NO	NO	NO	NO	NO	NO
STRING DATA						
Variables	NO	YES	*YES	NO	++YES	*YES
Constants	NO	YES	*YES	NO	++Local copy	*Local copy
Expressions	NO	Local copy	*Local copy	NO	++Local copy	*Local copy
Non-virtual array elements	NO	YES	*YES	NO	*Local copy	NO
Virtual array elements	NO	Local copy	*Local copy	NO	*Local copy	NO
Non-virtual entire arrays	NO	YES	*YES	NO	NO	*YES
Virtual entire arrays	NO	NO	NO	NO	NO	NO

* indicates the default parameter passing mechanism for BASIC programs.

In no case should you use a BY clause when calling a BASIC subprogram from a BASIC main program. The default parameter passing mechanisms for the CALL statement correspond precisely to the way a BASIC subprogram expects to receive the parameters. VAX-11 BASIC permits the use of the BY clauses only when the main and subprograms are written in different languages.

++ in PDP-11 BASIC, indicates those parameters for which you can specify BY REF when calling a MACRO routine.

Local Copy means that if a parameter is an expression, a function, or a virtual array element, then it is not possible to pass the parameters address. In these cases BASIC makes a local copy of the parameters value, and passes this local copy by reference. You can force BASIC to make a local copy of any parameter by enclosing it in parentheses.

Green Fungus & White Fuzz (Cont.)

The crack maintenance team replaced the filter, but with the wrong filter. This wrong filter caused the deionzer to blow salt water throughout the computer center. This salt water formed as a white fuzz all over the top of all the machines in the area. The white fuzz attacked the disc packs, destroying almost all of them.

Texas Southern has a building problem, an "old" building problem. And Texas Southern also has a crack maintenance crew.

The crack maintenance crew put a false ceiling in the computer center. There was a old water pipe that wasn't used anymore. Since the pipe wasn't to be used it should be OK to just cut the pipe off. And Since the pipe wasn't used the crack maintence crew didn't have to go the the trouble of threading and capping it.

Not to long after the remodling another member of the crack maintenance crew found a valve in the water system that was closed, and since there shouldn't be any valves in the water system that should be closed, he opened it - giving new meaning to floating point processor.

Hard as it is to believe, all university computer problems do not occur in Texas. At the University of Northern Colorado the main frame and the disc drives were provided and serviced by different vendors. The system used cards for input, and the Field Service people told the school that all of their card problems were because of static electricty because of the low humidity in the computer room.

The director of the center, beliving the Field Service people to always be right, had a humidifier installed. Then, all of a sudden, the main frame would not "talk" to the discs. Of course each vendor point a accusing finger at the other.

Each vendor trace their part of the system back to the controller, pronouncing their part as sound. It seems that the new dehumidifier was installed over the controller and as long as it was turned on the electrical disturbance was enough to disrupt the flow between the main frame and the discs.

BASIC-PLUS-2 TRANSLATOR ISSUES

The new BASIC Transportability Package consists of three components:

- o The BASIC Transportability Manual
- o The BASIC-PLUS Translator
- o The BASIC-11 Translator

The Manual discusses topics including "Tips for writing transportable BASIC programs", "Using the translators", "Overview of system differences influencing BASIC programming". It intended to help the non-novice BASIC programmer identify issues in moving to BASIC-PLUS-2, or in moving BP2 applications to VAX-11 BASIC.

The translators themselves are the same as those previously bundled with the BASIC-PLUS-2 distribution kit.

This article summarizes the major issues from a previous Small Buffer article concerning the translators, to assist in providing some know-how in tailoring the size and several functional characteristics of the translators.

A 28-K TRANSLATOR

The version of the BASIC-PLUS translator distributed with BASIC-PLUS-2 V1.6 is built to run in 24K. Because of this limitation, it is heavily overlayed, and execution speed suffers. It has been suggested that a 28K version of the translator would run more quickly on systems that support 28K tasks. Therefore a new ODL file, that produces a 28K translator, has been written.

The installation of this new translator requires rebuilding the old one. The procedure would be to copy the compiler object library, and the files TRANS.ODL, TRANS.CMD, UTLIC1.ODL, and BP2ØLB from the rebuild tape of the BASIC-PLUS-2 distribution into a rebuild account. Then the ODL file must be edited so that it looks like the one below. It is then necessary to change the command and ODL files to reflect: (1) the correct accounts (N: and OTS:), and (2) OTS object libraries (BAXRMS and BP2ØLB). Then the new translator may be task-built (TKB @TRANS), and moved to the desired system account.

This new translator should be able to run about 25 to 3Ø percent faster than the old one (the larger the file, the larger the amount saved).

THE ODL FILE FOR A 28K TRANSLATOR

```
             .ROOT BASIC2-RMSROT-USER,RMSALL
USER:        .FCTR <COMLIB>/LB:TRANS:TRNPAT-LIBR-*(A-B,C)
A:           .FCTR <COMLIB>/LB:TRNSIN:GETSYS:NAMSCN-LIBR
B:           .FCTR <COMLIB>/LB:TRNS1-LIBR
C:           .FCTR <COMLIB>/LB:TRNS2-LIBR-*(C1-C2-C3-C4-C7,C5-C6)
C1:          .FCTR <COMLIB>/LB:TRNBLI-LIBR
C2:          .FCTR <COMLIB>/LB:TRNLIN-LIBR
C3:          .FCTR <COMLIB>/LB:TRNPST-LIBR
C4:          .FCTR <COMLIB>/LB:TRNENC-LIBR
C5:          .FCTR <COMLIB>/LB:TRNSP1-LIBR
C6:          .FCTR <COMLIB>/LB:TRNSP2-LIBR
C7:          .FCTR <COMLIB>/LB:TRNCOM-LIBR-E-F-G
E:           .FCTR <COMLIB>/LB:TRNACT-LIBR-E1-E2-E3-E4
E1:          .FCTR <COMLIB>/LB:TRNBKT-LIBR
E2:          .FCTR <COMLIB>/LB:TRNDEL-LIBR
E3:          .FCTR <COMLIB>/LB:TRNREP-LIBR
E4:          .FCTR <COMLIB>/LB:TRNINS-LIBR
F:           .FCTR <COMLIB>/LB:TRNSCA-LIBR
G:           .FCTR <COMLIB>/LB:TRNSST-LIBR
LIBR: .FCTR   BP2ØLB/LB
@UTLIC1
@LB:[1,1]RMS11S
             .END
```

In the ODL file the symbol <COMLIB> is the compiler library and should be RTSLIB on RSTS, RSXLIB on RSX,and IASLIB on IAS.

The BASIC-PLUS Translator contained on the BASIC Transportability Kit is built to run in 28K, as is the BASIC-11 Translator. If you have these versions of the translators, you already have the speed of the 28K translator. However, if your system will not support 28K images, or if you wish a smaller translator image, the translator can be rebuilt to run in 24K. In this case copy the files TRANS.OLB, (or TRAN11.OLB), TRANS.CMD (or TRAN11.CMD), and TRANS.ODL (or TRAN11.ODL) off the transportability Tape, and edit the ODL file so it looks like the one below. Then task build the Translator, and move it to your system account.

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

ASSOCIATE [redacted]

[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.