CSUB Utility

for the HP Series 200 Computers

Manual Part No. 09800-10641

Disc Part No.	09800-10344	3½" External
Disc Part No.	09800-10544	5¼" External
Disc Part No.	09800-10644	5¼" Internal

Hewlett-Packard Desktop Computer Division
3404 East Harmony Road, Fort Collins, Colorado 80525

Printing History

New editions of this manual will incorporate all material updated since the previous edition. Update packages may be issued between editions and contain replacement and additional pages to be merged into the manual by the user. Each updated page will be indicated by a revision date at the bottom of the page. A vertical bar in the margin indicates the changes on each page. Note that pages which are rearranged due to changes on a previous page are not considered revised.

The manual printing date and part number indicate its current edition. The printing date changes when a new edition is printed. (Minor corrections and updates which are incorporated at reprint do not cause the date to change.) The manual part number changes when extensive technical changes are incorporated.

February 1983...First Edition

Table of Contents

HP Computer Museum
www.hpmuseum.net

For research and education purposes only.

Chapter 6: Advanced Topics

Appendix

Chapter 1
Introduction

A compiled subprogram (CSUB) is a routine that is written in either Pascal or assembly language on a Pascal 2.0 workstation and transformed into a routine that is callable from BASIC. Either a single routine or a library of routines may be generated using this technique. Once in the BASIC system, the CSUB is loaded into memory using the LOAD or LOADSUB command and is then callable like any other BASIC subprogram.

Before using CSUBs, you need to know how to use the Pascal workstation to generate programs. A working knowledge of the Editor, the Compiler (or Assembler), and the Librarian is necessary in order to generate CSUBs.

It will also be very helpful if you have an understanding of the Pascal language IMPORT feature. This will be used heavily to incorporate information from libraries.

If any of this background is missing, you should learn these things before proceeding with CSUBs. All the information can be obtained from the *Pascal 2.0 User's Manual* (Part No. 98615-90020) or the *BASIC Programming Techniques Manual* (Part No. 09826-90011).

When to Use CSUBs

CSUBs can be used to fill a number of needs. Because the compiled or assembled routines are in 68000 machine code, they have a speed advantage over BASIC interpreted code. Computational routines go much faster when written in Pascal. CSUBs are in object code on the disc, which means that source code does not have to be released with them. This provides a greater measure of security for programs employing them. Also, CSUBs are useful when libraries of routines are needed for both Pascal and BASIC environments. By writing the routines in Pascal, you only need to write the routine once.

Performance using CSUBs was found to be greatly superior to BASIC when a set of benchmark programs was run against both. In these benchmarks, REAL and INTEGER loops, transcendental and non-transcendental math, quick and shell sorts and matrix multiplication were compared. In the extreme case, the quick sort CSUB ran many times faster than a comparable BASIC routine. In the case of transcendental math, the CSUB was found to be only marginally faster.

The price that is paid for this speed is the size of the routines. CSUB code will always be larger than comparable BASIC code.

CSUB Limitations

There are some limitations in using CSUBs. Only INTEGER, REAL and STRING type parameters are allowed in a CSUB. Only a very limited I/O capability is available to a CSUB. Pascal READ(LN) and WRITE(LN) can be used in a CSUB but reading will always be done from the keyboard and writing will always be done to the PRINTER IS device. There is no file I/O capability. Routines must have all external references fully resolved before they can be used as CSUBs. This may lead to problems if calls to Pascal system routines generate unresolved external references. The "Linking the Elements" and "What Can Go Wrong" sections contain more information about this problem.

The Pascal LIBRARY is not supported.

The System Components

The CSUB utility consists of several different system components. There are three programs and two libraries that are used. The first program is called BUILDC. Its purpose is to interact with the user to gather information about the subprogram that is being generated and how it will look to BASIC. The other two programs are used by the GENC stream file. The first program to be executed in the stream file is RELDATA, which generates relocation information for BASIC. This allows programs of length greater than 32K to be used as CSUBs. The second program, BUILDLIF, takes the fully linked routine and reformats it to a LIF[1] or SRM[2] disc as a BASIC PROG file.

1 LIF - Logical Interchange Format.
2 SRM - Shared Resource Management.

The two libraries that are included are CSUBDECL and CSUBLIB. CSUBDECL's and CSUBLIB's declarations may be imported into your CSUB routines. CSUBDECL has one module by the same name which contains a number of type declarations to be used for passing BASIC parameters.

The file CSUBLIB contains the following modules:

Module	Contents
CSUBENTRY	contains code to facilitate entry and exit from CSUB (must always be present)
CSUBSYM	defines some entrypoints into BASIC and bootrom routines. It contains no code. (must always be present)
COMSTUFF	contains routines for accessing COM
HEAP	contains routines to support heap (requires COMSTUFF).
HPM	additional heap routines necessary when HEAP_DISPOSE compiler option is turned on (requires COMSTUFF and HEAP)
ALLREALS	REAL number arithmetic support
STRING	Pascal string support routines
MODIV	Pascal DIV and MOD routines
MISC	Miscellaneous Pascal support routines (check the DEF Table listing under "What Can Go Wrong")
SETSTUFF	support routines for Pascal sets
FS	support for string and INTEGER read and write
MFS	support for REAL read and write (requires ALLREALS and FS)

The GENC stream file must be modified so that the needed modules get linked to your Pascal module. This is explained in detail under "Linking the Elements". CSUBENTRY and CSUBSYM are linked automatically when GENC is streamed.

Chapter 2
Writing CSUBs

When you are developing a system that involves the use of CSUBs, the BASIC program should be written first. Determine what the CSUB should do, the parameters to be passed and the variables that should be global to both the BASIC program and the CSUB. When you have the BASIC program done, you can then begin developing the Pascal CSUB.

Pascal CSUBs must be contained in a module. This facilitates separate compilation and will not build a dummy main procedure segment.

A module body surrounds the Pascal routines. There is an EXPORT statement that defines the procedures that will be CSUBs. The code for the CSUBs is found in the IMPLEMENT section of code. There may be more procedures in the IMPLEMENT section than there are CSUBs.

Compiler Directives

It is necessary to turn off a number of compile time options in order to successfully create a CSUB. The options that must be specified are $STACKCHECK OFF$ and $IOCHECK OFF$. You cannot use $DEBUG ON$ which defaults to OFF.

Leaving the checking on may cause some system variables to be over-written, causing to system crash.

IMPORT/EXPORT Modules

Importing the module CSUBDECL allows the use of the types BINTEGER_PARM and BREAL_PARM as well as several others. These are used to simplify parameter passing to BASIC. They are used to pass a BASIC INTEGER and a BASIC REAL number into the CSUB, and return the value from the CSUB. They are both pointers to the values.

The procedure names that will be CSUBs must be included in a Pascal EXPORT statement. Other routines may be included in the IMPLEMENT section, but unless their names are included in the EXPORT section and the subprograms defined in the BUILDC program, the routines will not show up as separate CSUBs.

```
IMPORT CSUBDECL;

EXPORT
  PROCEDURE SQRIT(X: BINTEGER_PARM; Y: BREAL_PARM);
  PROCEDURE READIT(Z: BINTEGER_PARM);
```

Declarations of types and procedures in other user-created libraries may also be imported into the CSUB procedures. These additional libraries must then be added to the GENC stream file. Details of how this is done are found in the "Linking the Elements" section.

Parameter Passing

There are a few notes on parameter passing to and from BASIC. BASIC parameters are **always** passed by reference as far as Pascal is concerned, that is, a pointer to the actual value is passed. When you think you are passing a parameter by value, BASIC actually makes a copy of the value and passes a pointer to it. When you return to the calling program, the copy is destroyed. Pascal VAR parameters are "pass by reference", so they can be used to receive BASIC parameters.

The BASIC parameter types are not necessarily the same as their Pascal counterparts. It is important that the format of the parameters is correct so that BASIC can interface properly. Some of these differences were mentioned previously. This section will explain the formats in detail.

INTEGERS. A variable defined as an INTEGER in Pascal is not the same as a BASIC INTEGER. BASIC INTEGERs are 16-bit. Pascal INTEGERs are 32-bit.

```
EXPORT
  PROCEDURE X (VAR Y: INTEGER); (WILL NOT WORK!!)
```

This will not bring the correct value into the Pascal routine or return the proper value to BASIC. Instead the code could be:

```
TYPE
  TWOBYTES = -32768..32767;
  BINTVALTYPE = TWOBYTES;

EXPORT
  PROCEDURE X (VAR Y: BINTVALTYPE);
```

If CSUBDECL is imported into the module, you need not include the BINTVALTYPE type declaration above.

```
IMPORT CSUBDECL;

EXPORT
  PROCEDURE X (VAR Y: BINTVALTYPE);
```

BINTEGER_PARM is defined in CSUBDECL as a pointer to BINTVALTYPE.

```
IMPORT CSUBDECL;

EXPORT
  PROCEDURE X (Y_PTR: BINTEGER_PARM);
```

If the "VAR" method is used, the value is accessed as "Y". If the latter method is used, the value is accessed as "Y_PTR^".

REALS. REAL numbers are the same in both BASIC and Pascal. This means that it is possible to define a procedure:

```
PROCEDURE X (VAR Y: REAL);
```

This is exactly the same as:

```
PROCEDURE X (Y_PTR: BREAL_PARM);
```

because BREAL_PARM is defined as a pointer to REAL.

STRINGS. Strings are different in BASIC and Pascal, both in their structure and the way they are passed. The structure of the Pascal STRING type is a one byte length field followed by the STRING characters. The BASIC STRING has a two byte length field followed by the STRING characters. The BASIC STRING is passed as two parameters. The first is a pointer to the "DIMension record", the second a pointer to the STRING value. The dimension record for a STRING contains a single value, a 16-bit INTEGER that has the dimensioned length of the STRING. "DIMENTRYPTR" is the pointer to the dimension record. The value area has the two-byte actual length of the STRING value, followed by the characters of the STRING. "BSTRING_PARM" is a pointer to the BASIC STRING structure.

This is the structure of a BASIC STRING "Str$[80]".

```
BSTRINGVALTYPE = RECORD
  len : shortint;
  c : PACKED ARRAY[1..stringlimit] of CHAR;
END;
```

Refer to "Useful TYPE Declarations" for the TYPE definitions.

An example of how this would look in a Pascal program would be:

```
IMPORT CSUBDECL;

EXPORT
    PROCEDURE GETSTRING (dim_len: DIMENTRYPTR; b: BSTRING_PARM);

IMPLEMENT
    PROCEDURE GETSTRING;  {Not necessary to duplicate parm list}
        VAR s : STRING[80];
            i : shortint;
        BEGIN
           WRITELN('Enter an 80 character or less string:');
           READLN(s);
           b^.len:=STRLEN(s);
           FOR i:=1 TO b^.len DO
              b^.c[i]:=s[i];
        END;
END.  {module m1}
```

Notice that the Pascal STRING value must be put into the proper BASIC STRING value before it is passed back.

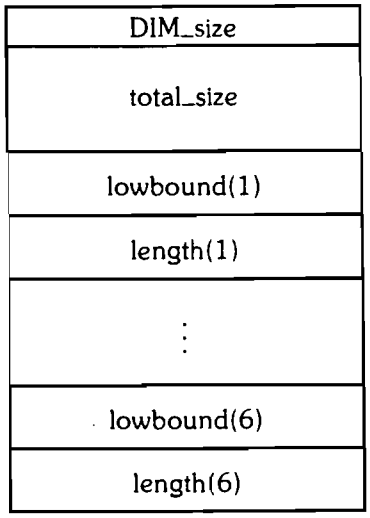
As far as BUILDC is concerned, this is still only one parameter of type STRING:

```
DIM Str$[80]

CALL Getstring(Str$)
```

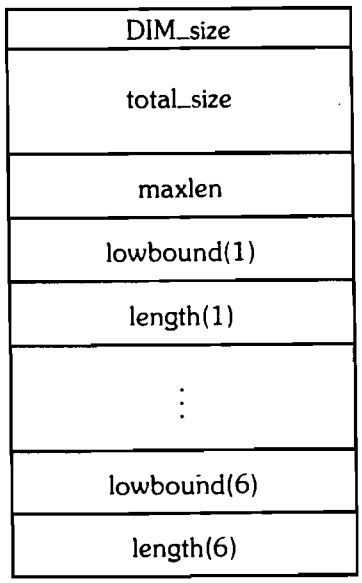
ARRAYs. Arrays are also passed as two parameters, a pointer to the dimension record and a pointer to the value area. The dimension record in this case, is more complicated. It is defined by the CSUBDECL module's DIMENTRY declaration. It is a variant record. Its structure is given in "Useful TYPE Declarations". For clarification, the variant record is shown below as several non-variant records:

```
bound_rec : RECORD
        lowbound : word;
        length : word;
    end;

REAL_dim_record : PACKED RECORD     {and INTEGER_dim_rec}
        DIM_size : byte;
        total_size : three_bytes;
        bounds : array [1..6] of bound_rec;
    end;

STRING_dim_record : PACKED RECORD
        DIM_size : byte;
        total_size :three_bytes;
        maxlen : word;
        bounds : array [1..6] of bound_rec;
    end;
```

REAL or INTEGER ARRAY



STRING ARRAY



"DIM_size" is a byte containing the number of dimensions, "total_size" is the number of bytes in the entire array, "lowbound(1)" is the lower bound of the leftmost dimension and "length" is the number of elements in that dimension. There are up to six lowbounds and lengths. That is the maximum number of dimensions in an array. In a STRING array, "maxlen" is the maximum length of any element in the array.

If this structure is needed, its declaration and the pointer to it (DIMENTRYPTR) are available in the CSUBDECL module. The dimensions of an array can be extracted from this structure. The element TYPEs should be known when the CSUB is written.

An example of receiving an INTEGER array from BASIC might be:

```
EXPORT
 TYPE
    arrstr = ARRAY [1..10] OF BINTVALTYPE;

  PROCEDURE X (d: DIMENTRYPTR; VAR arr: arrstr);
```

Again, this is a single parameter in BASIC. In order for this to be correct, BASIC would have to be sending it an array defined:

```
INTEGER Arr(1:10)

CALL X(Arr(*))
```

The Pascal array should be defined the same as the BASIC array. This is not mandatory, but helpful. Remember that an array [1..5] is the same as an array [6..10]. Both are five-element arrays. If a BASIC array defined as "INTEGER Arr(6:10)" is passed to a Pascal CSUB defined "ARRAY [1..5] OF TWO_BYTES;" that element "6" to the BASIC program is the same as element "1" in the Pascal CSUB.

The DIM statement should be used in conjunction with the Pascal array declaration to define the space for the array. The REDIM statement does not affect the size of that space. It does affect the BASE and SIZE functions and the length and lowbound values in the dimension record. The Pascal CSUB should check the dimension record to find the useful dimensions of the array. This is demonstrated later in the PAS_FIG example.

Optional Parameters

You can declare some or all af the CSUB parameters as optional. See "Subprograms" in *BASIC Programming Techniques* for information about optional parameters.

You must provide for all parameters in your Pascal subprogram as if they all were required parameters. BASIC passes a NIL pointer when an optional parameter is omitted.

Using I/O

The FS and MFS modules in file CSUBLIB have a set of routines that make it possible for you to do simple I/O from a CSUB. You are allowed to input characters from the keyboard and write to the PRINTER IS device. The mechanism for using these routines is to use Pascal READ, READLN, WRITE and WRITELN in the CSUB. Pascal STRINGs, CHARs, INTEGERs, REALs, PACKED ARRAY of CHARs can be read and written this way. The READ and WRITE routines will behave as they would in BASIC, which may be slightly different than in Pascal. For example, when reading a number, BASIC ignores blanks where Pascal treats them as separators. In addition, reads and writes of Pascal integers are restricted to the range of BASIC integers. When trying to read or write Pascal integers outside the range, an INTEGER OVERFLOW error will be reported.

To bring these routines into the CSUB, it is necessary to include FS and MFS in the first link routine in the GENC stream file which is described in the "Linking the Elements" section. FS contains routines that do INTEGER and STRING I/O. MFS has routines to do REAL I/O. If MFS is needed, then the module ALLREALS and FS must also be linked to the file.

Chapter 3
Preparing CSUBs for BASIC

Generating Header and Jump Files

Once the Pascal or assembly language routine is in code form, you must execute the program called BUILDC. BUILDC prompts for information about the calling sequence of the CSUB. Remember that BUILDC is asking what the routines and parameters will look like to BASIC. This may be different from the way they are defined in Pascal. Determine how the CSUB statement should look to BASIC, and answer the same way.

BUILDC begins by prompting:

```
Compiled Subprogram Header Generator

Enter the name for a Stream File for BUILDC:
(or just press <ENTER> for no stream file  :
```

If you give a name for the stream file, all your responses will be recorded for future use. Next time you need only stream the file you create this time. See the Stream command in the Pascal User's Manual for more information on streaming files.

The next prompts are:

```
Enter name for the Header File:
Enter name for the Jump File:
```

These are two files that are generated by the BUILDC program. When the program is complete, the first one will contain the BASIC subroutine definition for the CSUB. The second one will contain information about where to find the beginning of the routines in the subroutine library. The names of both of these files will be asked for when the stream file GENC, that links all the pieces together, is streamed. For convenience, you may name these temporary files "HEADER" and "JUMP" respectively. The suffix ".CODE" will be added automatically to the file name.

The next prompt is:

```
Enter the name of the code file containing the compiled sub:
```

Enter the name of the code file that you have generated. The suffix ".CODE" will be added automatically to the file name. This code file must contain only one module. If you have more, they must first be linked together into one module before running BUILDC. It will open this file and look to see if any global variable space is required by the routines. If space is required the user will be asked:

```
Enter COM label for global variable space:
```

If there is no global variable space required, this prompt will not appear. Global/COM information is presented in the "Advanced Topics" section.

The next prompt will be:

```
Enter module name:
```

This refers to the name given to the Pascal module. If the CSUB is written in assembly language, there will not necessarily be a module name. In this case, just press [ENTER] in response to the prompt.

The next section asks for information about the procedures. It will be repeated until a null procedure name is entered by pressing [ENTER] in response to the prompt.

```
Enter procedure name:
```

The procedure name is the same name as the procedure in the Pascal module or the name defined by the Assembler's DEF statement.

The next prompts get information about the parameters:


```
Parameter name:
Parameter type (I/R/S for Integer/Real/String):
Is this an array? (y/n):
Is this an optional parameter? (y/n):
```

Remember, list parameters as they will look to the calling BASIC program; not as they look in Pascal.

The legal types are INTEGER, REAL and STRING. If the parameter name ends in a dollar sign ($) then the type is automatically assumed to be STRING. Otherwise answer with [I] or [R]. If the parameter name does not end in a dollar sign and the type entered is STRING, it will be treated as an error and you will be prompted again. If the parameter is an array, it is assumed to have dimension (*). That means it is defined in the calling program.

If a parameter is declared "optional" and it is not included in the CALL statement, a nil pointer is passed to the CSUB.

These prompts are repeated until a null parameter name is entered. Once a parameter has been specified as optional, all the following parameters default to optional and the prompt will not appear.

The next question about a procedure is:

```
Is there COM in this procedure? (y/n):
```

This is referring to accessing a BASIC COM from the Pascal module; not the Pascal global space. COM is described later.

All these prompts are made for each module. As soon as a null procedure name is entered, the following prompt appears:

```
Are there any more modules? (y/n):
```

As with all these yes/no prompts, a response other than [Y] is treated as a negative response. If the user responds positively to this prompt, the sequence of prompts start over with the "Enter procedure name:" prompt.

Linking the Elements

Once the header file and jump file have been created using the BUILDC program, the user will combine all the elements and create the BASIC subprogram by streaming the GENC file. Stream files are a way of batching up workstation commands to be done in sequence. In this case the GENC stream file has commands to link the user routine with library routines, execute the program that extracts relocation information, link the jump file and header file with the user routines and then execute a program that writes the completed routine to an SRM or LIF formatted disc.

A few words are needed here to explain how these programs should be on the disc to execute the stream file. In order to run the CSUB generating programs, only one disc drive need be available on the machine. However, the user may find it necessary to have two mass storage devices if the CSUB being generated is large. In some cases, you can use one device (the default volume) to hold all the CSUB code files, libraries, and stream file, and another to hold your code file, the header and jump files and the final CSUB. These are files whose volume name can be specified interactively, without having to modify the stream file. Remember that the System Volume must be on-line to stream a file. In this case, making the second volume the System Volume is the preferred alternative. When specifying the volume name for the temporary files, use "*".

In more extreme cases, it is necessary to modify the GENC stream file so that CSUB_TEMP includes a volume specifier, too. There are four occurences in the stream file where CSUB_TEMP must be modified. Just change "CSUB_TEMP" to "*CSUB_TEMP" using the Editor's Insert command.

If you have only one mass storage device, there must be enough RAM space available to create a memory volume and this should be the System Volume.

This is the easiest way to generate the CSUB once the Pascal module is written.

1. Begin by P-loading the Filer and Librarian so that they are resident in memory. (Provided you are not suffering memory restrictions.)
2. Set the default volume prefix to the unit that contains the CSUB Utility files.
3. Execute BUILDC putting the files it generates on either the default volume or the second volume.
4. Stream the file GENC by pressing [S] from the Main Command Level and entering the name "GENC". GENC asks for three file names:

   ```
   Input file name
   Header file name
   Jump file name
   ```

5. Type the file names (and volume names, if using two volumes). The suffix .CODE will be added automatically.

When all three names have been entered, the system will begin the stream. If there are errors in the stream, all processing will stop and the error will be displayed. The linking produces a file called CSUB_TEMP that contains the fully linked CSUB. This will be written to the default volume (unless you've added a volume prefix).

The Stream file then executes the program that writes the CSUB as a BASIC PROG file. When it is ready to output the CSUB, it prompts:

```
Enter the unit number or volume name of the
initialized media that is to contain the CSUB.
Just press <enter> for the default volume.

If needed, you can swap media so that the BASIC
program file (the CSUB) goes on a different media.
```

If you want the CSUB on the default volume, just press (ENTER). Alternatively, you may specify any volume you prefer.

If something should go wrong trying to create the BASIC program file, BUILDLIF does not purge the input file after it gets done using it. This allows you to re-execute BUILDLIF rather than re-streaming the entire GENC file. If you do this, you must provide the name of the input file, CSUB_TEMP.

When finished, you have a BASIC PROG file with the name of the first procedure in the Pascal library, truncated to 10 characters. All procedure names in the library are truncated to 15 characters. This library can then be loaded into your BASIC program.

The minimal stream file that links all the CSUB user components together looks like:

```
=IInput file name?
=HHeader file name?
=JJump file name?
=PEnter 'P' for linkmap.
LO CSUB_TEMP
L @P
I CSUBLIB
M CSUBENTRY
TM CSUBSYM
TI @I
  ALKQ
X RELDATA
CSUB_TEMP
CSUBRDATA

LO CSUB_TEMP
L @P
I @H
AI CSUB_TEMP
AI CSUBRDATA
AI @J
ALKQ

FR CSUBRDATA.CODE
Q

X BUILDLIF
CSUB_TEMP
```

If you need INTEGER and STRING I/O routines, you must add the following line to the first section of the stream file after "TM CSUBSYM" and before "TI @I".

```
TM FS
```

If you need REAL number math, add this line also.

```
TM ALLREALS
```

If you need REAL number I/O, add this line.

```
TM MFS
```

In order to link any of the modules listed in the first section, you must add a line similar to those above. The one exception is the HEAP module. Because it contains global variables, it must be linked to your Pascal module before you run BUILDC. Similarly, you can pre-link any module to your Pascal routines. If the module contains global variables, it must be linked before running BUILDC. Otherwise, the GENC stream file can be modified to do the linking for you. Pre-linking modules is described under "Using the Heap".

If you need a module that is not contained in CSUBLIB, the GENC stream file must be modified differently. Rather than adding a "TM MODULE" line, you must break up and add lines to the last line of the linking section of the stream file.

```
LO CSUB_TEMP
L @P
I CSUBLIB
M CSUBENTRY
TM CSUBSYM
TM ANYMODULE
TI @I
ALKQ
```

The last line is replaced with these three lines:

```
AI <file name>
M <module name>
TLKQ
```

If you needed more than one module, the last line is replaced with these lines:

```
AI FILENAME
M MODULE1NAME
TM MODULE2NAME
TM MODULE3NAME
TLKQ
```

FILENAME is the name of the file containing the needed modules. MODULExNAME are the names of the needed modules. To understand what you're doing here, read about the Stream command and the Librarian.

Calling CSUBs from Basic

Before you can call a CSUB, you must load it into the program.

```
LOADSUB FROM <file>
```

This statement causes the entire library to be brought if one of it's subprograms is called.

Assume that we have the following PROG file consisting of several CSUB libraries generated at different times and merged together:

Main
Suba (library A)
(begin library B)
Subb
Subc
Subd
(end library B)
(begin library E)
Sube
Subf
Subg
(end library E)
Subh

For example, if Suba and Subc are referenced in a program, library A and library B are brought in. The other libraries are not brought in unless the contained procedures are referenced.

```
LOADSUB <sub> FROM <file>
```

This statement causes all the code in the library containing "sub" to be loaded from "file" but only procedures from "sub" to the end of the library are listed in the BASIC program. For example, if Subc is loaded, Subd is also loaded.

```
LOADSUB ALL FROM <file>
```

This statement causes all subprograms in "file" to be loaded.

Editing CSUBs

CSUB libraries are generated by a single execution of the BUILDC program. All CSUBs belonging to the same library are contiguous and must remain in the order in which they were generated. The system will let you add statements between or below CSUBs, but this will modify the CSUB above the entry. For CSUBs to execute properly, they may not be modified in any way. Once a CSUB has been modified, you must reload to execute it.

Any number of CSUB entry points may be deleted from the beginning of the library, but not from the end or the middle.

Chapter 4
What Can Go Wrong

Unresolved External References

The most common problem encountered with CSUBs is "unresolved external references". These get reported by either the RELDATA program or the BUILDLIF program. When this happens, you must determine the source of the references, and link the missing modules to your CSUBs. To find out what all your unresolved external symbols are, use the librarian to list the externals of file CSUB_TEMP. See Pascal 2.0 User's Manual for details. If any of these externals are generated by the Compiler, you need to find out which modules in the CSUBLIB contain these entry points and link them in.

The tables below list the entry points in the CSUBLIB modules. If you have an unresolved reference, use the tables to find the missing module.

If the missing module contains global variables, it must be manually pre-linked. Otherwise, the GENC stream file can be modified to do the linking for you. Add the "TM MODULENAME" to your GENC stream file.

DEF table of 'COMSTUFF':

```
COMSTUFF_COMSTUFF
COMSTUFF_FIND_COM
```

DEF table of 'HEAP':

```
ASM_NEWBYTES
ASM_NEWWORDS
HEAP
HEAP_HEAP
HEAP_HEAP_INIT
```

DEF table of 'HPM':

```
HPM
HPM_DISPOSE
HPM_HESTABLISH
HPM_HPM
HPM_MARK
HPM_NEW
HPM_RELEASE
HPM__BASE
```

DEF table of 'ALLREALS':

```
ASM_ARCTAN
ASM_BCDROUND
ASM_BCD_REAL
ASM_COS
ASM_EQ
ASM_EXP
ASM_FLOAT
ASM_GE
ASM_GT
 ASM_LE
ASM_LN
ASM_LT
 ASM_NE
ASM_RADD
ASM_RDIV
ASM_REAL_BCD
ASM_RMUL
ASM_ROUND
ASM_RSUB
ASM_SIN
ASM_SQRT
ASM_TRUNC
```

DEF table of 'STRING':

```
ASM_DELETE
ASM_INSERT
ASM_PSUBTOPSUB
ASM_PSUBTOSSUB
ASM_SAPPEND
ASM_SCOPY
ASM_SSUBTOPSUB
ASM_SSUBTOSSUB
ASM_STRLTRIM
ASM_STRRPT
ASM_STRRTRIM
```

DEF table of 'MODIV':

```
ASM_DIV
ASM_MOD
```

DEF table of 'MISC':

```
ASM_BINARY
ASM_FASTMOVE
ASM_HEX
ASM_MOVEL
ASM_MOVELEFT
ASM_MOVER
ASM_MOVERIGHT
ASM_OCTAL
ASM_PACK
ASM_SCAN
ASM_UNPACK
```

DEF table of 'SETSTUFF':

```
ASM_ADDSETRANGE
ASM_ADELEMENT
ASM_SETASSIGN
```

DEF table of 'FS':

```
FS
FS_BASICTYPECONST
FS_DEV_INIT
FS_FHPRESET
FS_FREADCHAR
FS_FREADINT
FS_FREADLN
FS_FREADSTR
FS_FREADWORD
FS_FS
FS_FWRITECHAR
FS_FWRITEINT
FS_FWRITELN
FS_FWRITEPAOC
FS_FWRITESTR
FS_FWRITEWORD
FS_IO_ENTERNUM
FS_IO_ENTERSETUP
FS_IO_ENTERTERM
FS_IO_OUTPUT
FS_IO_STRINPUT
FS_NUMERIC_DEFAULT
FS_PRT_INIT
FS_RESET
FS__BASE
IOENTRY_STRINPUT
ROUNDIT
```

DEF table of 'MFS':

```
MFS
MFS_FREADREAL
MFS_FWRITEREAL
MFS_MFS
MFS__BASE
```

CSUB Errors

BASIC uses error range 373 through 399 to report CSUB errors. There is no textual message displayed with these errors. To find out what these errors mean, first subtract 400 from the error number. The result will be an escapecode value which you can look up in the Errors appendix in the *Pascal 2.0 User's Manual* (Part no. 98615-90020) under "Operating System Run Time Error Messages".

System Stack Overflow

Another problem you may run into is overflowing the system stack. Unfortunately, there are no checks. for this condition in the system. You can find out how much space you have by calling the function STACKSPACE. You should try to maintain about 5K free stackspace at all times for the BASIC system to use for interrupts and other various needs.

STACKSPACE is contained in the module CSUBENTRY and is automatically linked to your module. It will return an INTEGER containing the number of bytes available on the system stack. If you overflow the stack, you will destroy some system information (most likely the user program) and this will result in unpredictable behavior. You will need to reload the system.

Parameter Mismatch

Since there is nothing in the system that checks the parameter list of a CSUB to make sure it matches the Pascal declaration, it is up to you to make sure that the matching is done correctly. See "Parameter Passing". If this is not done correctly, the stack will not be restored properly and unpredictable behavior will result.

Heap Initialization

If you attempt to use the heap without calling the HEAP_INIT procedure first, the heap pointer will not be initialized and unpredictable behavior will result.

Relocation Errors

The RELDATA program generates relocation information to be used by the CSUB anytime the CSUB is moved. It can only, however, relocate objects which are 32 bits wide. If it can't generate the relocation information, it will report "Unable to relocate object at nnn". The number 'nnn' is the offset of the object in the code block.

One example. Using the $CALLABS OFF$ Compiler option causes PC-relative addressing to be used to call procedures. If you are calling a routine 32768 or more bytes away, and you don't provide patch space at link time, the call will not reach and an error will be reported. Either use long absolute addressing or let the Librarian put in patch jumps at link time to correct this problem.

Another example. An assembly language routine using short absolute addressing (16-bit addressing) to jump to another routine in the CSUB will not be able to address the procedure and an error will be reported by RELDATA. Use long absolute addressing to correct this problem.

Debugging

There is no BASIC Debugger similar to the Pascal Debugger. You should test and debug your code on the Pascal workstation as much as possible. If this is not possible for some reason, and you run into problems with your code, you should insert WRITELNs in your program to print diagnostic information.

Chapter 5
Two Complete Examples

This section presents two examples which should serve to illustrate the concepts described thus far. The examples each include the BASIC program, the Pascal module containing the CSUB, the execution of the BUILDC program and the modification necessary to the GENC stream file.

Find the String

This example is written for a system that has a large mass storage device such as a hard disc or SRM. All files used in the Pascal environment are read from and written to this device. The default prefix is set to this volume. In the BASIC environment, the CSUB is loaded from the MSI device.

This simple program fills an array of strings with a few choice strings and then asks a Pascal CSUB if a particular string is found in the array. The BASIC program keeps track of how many valid strings are contained in the array and passes that information to the Pascal CSUB. The next example extracts this information from the dimension record parameter. If the INTEGER variable "Yes" comes back with a value other than zero, it comes back pointing to the array element of the matching string.

The BASIC Program

```
10    LOADSUB ALL FROM "FIND_STRIN"
20    DIM File$(1:10)[20]
30    DIM Str$[20]
40    INTEGER Num_strs,Yes
50    File$(1)="HELLO - HOW ARE YOU?"
60    File$(2)="I AM GREAT"
70    File$(3)="WHAT IS YOUR NAME?"
80    File$(4)="WHERE ARE YOU GOING?"
90    File$(5)="WHATS YOUR FAVORITE?"
100   File$(6)="I LIKE YOU"
110   Num_strs=6
120   Str$="WHERE ARE YOU GOING?"
130   Find_string(File$(*),Str$,Num_strs,Yes)
140   IF Yes<>0 THEN PRINT "The string was found in number ";Yes
150   IF Yes=0 THEN PRINT "The string was not found"
160   DELSUB Find_string
170   END
```

The Pascal CSUB

```
MODULE TEST;

$MODCAL$
$STACKCHECK OFF$
$RANGE OFF$
$OVFLCHECK OFF$

$SEARCH 'CSUBDECL'$

IMPORT CSUBDECL;

EXPORT
   TYPE string_type = RECORD
                len : shortint;
                chars : PACKED ARRAY [1..20] of CHAR;
              end;

        str_array  = ARRAY [1..10] of string_type;

   PROCEDURE Find_string (file_dim   : DIMENTRYPTR;
                          VAR FileX      : str_array;
                              str_dim    : DIMENTRYPTR;
                          VAR StrX       : string_type;
                          VAR Num_strs   : BINTVALTYPE;
                          VAR Yes        : BINTVALTYPE);

IMPLEMENT

   PROCEDURE Find_string ;

      VAR i, j : integer;

      BEGIN
         yes := 0;
         i:= 1;

         while (i <= num_strs) and (yes = 0) do
            begin
               if fileX[i].len = strX.len then
                  if strX.len = 0 then yes := i
                  else
                     begin
                        j := 1;
                        while (j < fileX[i].len) and
                          (fileX[i].chars[j] = strX.chars[j])
                             do j := j + 1;
                        if (fileX[i].chars[j] = strX.chars[j])
                             then yes := i;
                     end;
               if yes = 0 then i := i + 1;
            end;
      END;

END.
```

Running BUILDC

These are the responses given to the BUILDC program. Notice that a stream file is generated by the response to the first question. Next time, you can just Stream the file named "TEST1STR" which itself executes BUILDC and answers all the questions as they are answered when the stream file is created. This example assumes that all files are located on the default volume. If this were not the case, the Stream file would have to be modified as in the second example.

```
          Compiled Subprogram Header and Jump File Generator

Enter the name for a Stream File for BUILDC:
(or just press <ENTER> for no stream file  :TEST1STR

Enter name for the Header File: HEADER
Enter name for the Jump File: JUMP
Enter the name of the code file containing the compiled sub:TEST1

Enter module name:TEST
   Enter procedure name:FIND_STRING
      Parameter name:FILEX$
         Parameter type is string
         Is this an array? (y/n):Y
         Is this an optional parameter? (y/n):N
      Parameter name:STRX$
         Parameter type is string
         Is this an array? (y/n):N
         Is this an optional parameter? (y/n):N
      Parameter name:NUM_STRS
         Parameter type (I/R/S for Integer/Real/String):I
         Is this an array? (y/n):N
         Is this an optional parameter? (y/n):N
      Parameter name:YES
         Parameter type (I/R/S for Integer/Real/String):I
         Is this an array? (y/n):N
         Is this an optional parameter? (y/n):N
      Parameter name:

      Is there COM in this procedure?(y/n):N

   Enter procedure name:
               Are there any more modules ? (y/n):
```

The GENC Stream File

No modifications were necessay to the GENC stream file for this Pascal CSUB.

```
=IInput file name?
=HHeader file name?
=JJump file name?
=PEnter 'P' for linkmap.
LO CSUB_TEMP
L @P
I CSUBLIB
M CSUBENTRY
TM CSUBSYM
TI @I
ALKQ

X RELDATA
CSUB_TEMP
CSUBRDATA

LO CSUB_TEMP
L @P
I @H
AI CSUB_TEMP
AI CSUBRDATA
AI @J
ALKQ

FR CSUBRDATA.CODE
Q

X BUILDLIF
CSUB_TEMP
```

The Standard Deviation

This example is written for a system that makes use of two small flexible disc volumes (or one disc volume and one memory volume). This involves volume specifications for files named interactively and the addition of volume names to files named in the GENC stream file. The CSUB utility files are contained on one volume which is named "UTIL:". The volume prefix is set to "UTIL:". The Pascal module, the temporary files and the final CSUB file are all read from and/or written to a second volume named "MEM:". If you must create a memory volume for this operation, this should be the memory volume.

This program fills a 100-element array with random numbers between 0 and 1. It then calculates the high, mean and standard deviation values for the array.

A Pascal CSUB is then called to duplicate the calculations. Times for the calculations are kept so they may be compared. The BASIC program contains a REDIM statement so that just a part of the whole array may used. The REDIM statement can be used to change the size of the array without forcing you to modify the Pascal module each time. The Pascal module looks at the dimension record parameter to determine the size.

The BASIC Program

```
10    LOADSUB ALL FROM "PAS_FIG"
20    PRINTER IS 701
30    DIM Vals(1:100)
40    INTEGER I,N
50    REDIM Vals(1:60)
60    Lo=BASE(Vals,1)
70    Hi=BASE(Vals,1)+SIZE(Vals,1)-1
80    FOR I=Lo TO Hi
90      Vals(I)=RND
100   NEXT I
110   !
120   !      Given an array of real numbers,
130   !               find the HIGH, MEAN and ST. DEVIATION
140   ! **************
150   ! BASIC first
160   ! **************
170   !
180   T1=TIMEDATE
190   Bas_fig(Vals(*),High,Mean,St_dev)
200   T2=TIMEDATE


220   ! **************
230   ! Now Pascal
240   ! **************
250   !
260   T3=TIMEDATE
270   Pas_fig(Vals(*),High2,Mean2,St_dev2)
280   T4=TIMEDATE
290   !
300   ! **************
310   ! WHO'S FASTER?
320   ! **************
330   !
340   PRINT
350   PRINT "BASIC time for a ";Hi-Lo+1;
355   PRINT " element array is ";PROUND(T2-T1,-2)
360   PRINT "       HIGH                    MEAN              STANDARD DEV"
370   PRINT High,Mean,St_dev
380   PRINT
390   PRINT "Pascal time is ";PROUND(T4-T3,-2)
400   PRINT "       HIGH                    MEAN              STANDARD DEV"
410   PRINT High2,Mean2,St_dev2
420   PRINTER IS 1
430   DELSUB Pas_fig
440   END
450   SUB Bas_fig(Vals(*),High,Mean,St_dev)
460     Lo=BASE(Vals,1)
470     Hi=BASE(Vals,1)+SIZE(Vals,1)-1
480     High=0
490     Sum_sqr_vals=0
500     Sum_vals=0
510     FOR I=Lo TO Hi
520       IF Vals(I)>High THEN High=Vals(I)
530       Sum_sqr_vals=Sum_sqr_vals+(Vals(I)*Vals(I))
540       Sum_vals=Sum_vals+Vals(I)
550     NEXT I
560     Sum_val_sqrds=Sum_vals*Sum_vals
570     St_dev=SQR((Sum_sqr_vals-(Sum_val_sqrds/Hi))/(Hi-1))
580     Mean=Sum_vals/Hi
590   SUBEND
```

Results

```
BASIC time for a 60 element array is  .13
      HIGH                MEAN                STANDARD DEV
 .975691075891      .499808551992       .266905357957

Pascal time is  .05
      HIGH                MEAN                STANDARD DEV
 .975691075891      .499808551992       .266905357957
```

The Pascal Module

This module makes use of the dimension record parameter to determine the REDIMensioned size of the array.

```
MODULE TEST;

$STACKCHECK OFF$
$IOCHECK OFF$
$RANGE OFF$
$OVFLCHECK OFF$

$SEARCH 'CSUBDECL'$   {Assuming it's on the default volume.}

IMPORT CSUBDECL;

EXPORT
   TYPE vals_type      = ARRAY [1..100] of REAL;

   PROCEDURE Pas_fig (vals_dim    : dimentryptr;
                      VAR vals    : vals_type;
                      VAR high    : real;
                      VAR mean    : real;
                      VAR st_dev  : real);

IMPLEMENT

   PROCEDURE Pas_fig;

     VAR sum_val_sq, sum_sq_vals, sum_vals : REAL;
         i, hi                             : INTEGER;

       BEGIN
          high := 0;
          sum_sq_vals:=0;
          sum_vals:=0;
          hi := vals_dim^.bound[1].length;

          FOR i := 1 TO hi DO
             BEGIN
                IF vals[i] > high THEN high := vals[i];
                sum_sq_vals := sum_sq_vals +
                                       (vals[i] * vals[i]);
                sum_vals:= sum_vals + vals[i];
             end;

          sum_val_sq := sum_vals * sum_vals;
          st_dev := sqrt(
             (sum_sq_vals - (sum_val_sq / hi) ) / (hi-1) );
          mean := sum_vals / hi;
       END;
END.
```

Running BUILDC

When asked for names for the header, jump and input files, include the volume specifier "MEM:".

```
        Compiled Subprogram Header and Jump File Generator

Enter the name for a Stream File for BUILDC:
(or just press <ENTER> for no stream file  :TEST2STR

Enter name for the Header File: MEM:HEADER
Enter name for the Jump File: MEM:JUMP
Enter the name of the code file containing the compiled sub:MEM:TEST3

Enter module name:TEST
   Enter procedure name:F4S_FIG
      Parameter name:VALS
         Parameter type (I/R/S for Integer/Real/String):R
         Is this an array? (y/n):Y
         Is this an optional parameter? (y/n):N
      Parameter name:HIGH2
         Parameter type (I/R/S for Integer/Real/String):R
         Is this an array? (y/n):N
         Is this an optional parameter? (y/n):N
      Parameter name:MEAN2
         Parameter type (I/R/S for Integer/Real/String):R
         Is this an array? (y/n):N
         Is this an optional parameter? (y/n):
      Parameter name:ST_DEV2
         Parameter type (I/R/S for Integer/Real/String):R
         Is this an array? (y/n):N
         Is this an optional parameter? (y/n):N
      Parameter name:

      Is there COM in this procedure?(y/n):N

   Enter procedure name:
               Are there any more modules ? (y/n):N
```

The GENC Stream File

The line "TM ALLREALS" was added to the first section of the stream file because REAL number math is used in the CSUB. No I/O or other special Pascal support is needed. The CSUB_TEMP output file is changed to contain the "MEM:" volume specifier (in four places).

```
=IInput file name?
=HHeader file name?
=JJump file name?
=PEnter 'P' for linkmap.
LO LIF:CSUB_TEMP
L @P
I CSUBLIB
M CSUBENTRY
TM CSUBSYM
TM ALLREALS
TI @I
ALKQ

X RELDATA
LIF:CSUB_TEMP
CSUBRDATA

LO LIF:CSUB_TEMP
L @P
I @H
AI LIF:CSUB_TEMP
AI CSUBRDATA
AI @J
ALKQ

FR CSUBRDATA.CODE
Q

X BUILDLIF
LIF:CSUB_TEMP
```

Chapter 6
Advanced Topics

Pascal CSUBs

Accessing Pascal Global Space from BASIC

Any variables declared before the procedures are global variables. Here is an example of global variables in a CSUB:

```
EXPORT
   TYPE
        WORD = -32768..32767
   PROCEDURE ...

IMPLEMENT
   VAR
        X : WORD;
        Y : WORD;
        Z : WORD;

   PROCEDURE ...
```

The space that is needed for these is set up in the BASIC subprogram definition as a labeled COM. The BUILDC program asks the name of the code file that contains the user-defined procedures. It examines this file to determine if any global space is needed. If so, it asks for a COM name by which to identify the global variables.

By allowing you to name the COM, you can access it from BASIC. BUILDC determines how much space is needed for the global variables and generates a COM statement like the one below for the Pascal CSUB.

```
COM /<label>/ INTEGER Global(1:number)
```

The number is derived according to the following formula.

```
number = (global_bytes DIV 2) + (global_bytes MOD 2) + 7
```

The global_bytes are DIVided by 2 because 2 bytes fit in one BASIC INTEGER. The +7 is for system use. For example:

```
TYPE
  WORD = -32768..32767
VAR
  X : WORD;
  Y : WORD;
  Z : WORD;
```

BUILDC would determine that an equivalent BASIC COM statement would be:

```
COM /given_name/ INTEGER GLOBAL(1:10)
```

In order to access this space from BASIC, your BASIC program should have an equivalent statement. The variables are placed in the BASIC INTEGER array in reverse order. The Pascal variable Z is found in GLOBAL(1), Y is found in GLOBAL(2) and X is found in GLOBAL(3). If the Pascal declaration is "X, Y, Z : WORD;" the variables are not placed in reverse order. From Global(4) thru Global(10) is used by the system and should not be tampered with.

The elements of an array are not in reverse order.

A word of warning is in order about using global COM to store pointers. Both the COM value area and the BASIC subprograms are subject to being moved around in memory. If a global variable is used to store a pointer into a routine, a structured constant, or another global variable that is subsequently moved, the pointer is no longer valid. This rule also holds true for procedure variables, which are in effect pointers to procedures. COM is subject to being moved in memory at RUN, LOAD, and GET time. Basic subprograms are subject to move at RUN, LOAD, GET, and DELSUB time. The heap is implemented in COM and these restrictions apply to it also. Using the heap is described later in this section.

Accessing BASIC COM from Pascal

In order to access BASIC COM from the Pascal CSUB routine, it is necessary to call a function found in module COMSTUFF. The name of the function is FIND_COM and its calling sequence is shown below. By passing the name of the COM that is to be accessed, it will return a pointer to the beginning of the value area of that COM. Upper and lower case letters are significant in the COM name. It must be the same as BASIC would list it back, otherwise FIND_COM will not find the COM block. To access a blank COM, use a single blank for the COM name. If FIND_COM is unable to find the COM block requested, it will return a nil pointer.

The user is required to know the layout of this COM in advance. There is no way of determining this from the Pascal routine.

This is an example of accessing BASIC COM. Suppose the COM is defined in BASIC as:

```
COM /Num1/ INTEGER A,B(5),REAL C
```

To access this from a CSUB:

```
$SEARCH 'CSUBLIB'$
IMPORT COMSTUFF;
EXPORT PROCEDURE X ...

IMPLEMENT
 PROCEDURE X...
  TYPE COMTYPE = PACKED RECORD
         C: REAL;
         B: PACKED ARRAY (0..5) OF BINTVALTYPE;
         A: BINTVALTYPE;            {Notice the reverse order}
       END;
 VAR COMPTR : ^COMTYPE;
       VALA : BINTVALTYPE;

  BEGIN
       COMPTR:=FIND_COM('Num1');
       VALA:=COMPTR^.A;
       ...
```

This technique may be used for both reading information from a COM area or putting new values into COM. It should also be stated that there is nothing to prevent you from destroying the COM inadvertently.

When you run BUILDC, it is not required that you declare the COM that will be accessed in the CSUB. By declaring it, however, the COM will remain in memory even if the BASIC routine that defined it is removed. If you wish to define the COM with the CSUB, answer [Y] when asked "Is there COM in this procedure?". The following prompts will appear:

```
How many COM blocks are there?:
    Enter COM label:
      Item name:
      Item type (I/R/S for Integer/Real/String):
      Is this an array? (y/n):
```

The number of COM blocks will depend on the number of labeled COMs that you want to access. The other prompts are repeated for each COM block. The COM label may be a name or could be blank. Remember that an unlabeled COM must be defined in the main routine and there may be only one. Other labeled COMs need not be defined in the main program. The questions concerning name, type and array are the same as for the parameters, with one exception. If this is an array, the bounds of that array must be specifically entered. The prompts for these are:

```
Enter number of dimensions or *:
   Dimension n lower bound:
   Dimension n upper bound:
```

If the number of dimensions is entered as "*" the rest of the dimension prompts are not displayed. This may be used only if the array is defined either in the calling BASIC routine or a previous procedure. If there is an explicit number of dimensions, the lower and upper bound must be entered for each one.

If the type of the item is STRING, the following prompt appears:

```
String length:
```

The user should enter the DIMensioned STRING length allowed for this COM entry.

The final question about the COM entry is:

```
Will this be used as a BUFFER? (y/n):
```

You should answer N unless you plan to use the vaiable as a buffer in a TRANSFER statement. Read the "Advanced Transfer Techniques" chapter of the *BASIC Programming Techniques* manual for details.

These prompts will be repeated until all the information about all the parameters in all the procedures is gathered.

Using the Heap

The Pascal statements NEW, MARK, RELEASE and DISPOSE are supported in CSUBs. Because they must operate in a BASIC environment, their implementation is different than in standard Pascal. The heap space is another labeled COM that is accessible from both BASIC and Pascal. This COM may be defined in either the BASIC main routine or may be defined in the subprogram. By defining it in the main program, it is possible to vary the size of the heap, without changing the CSUB.

Assuming the size of the heap is defined in the BASIC calling program, respond with "*" to the BUILDC prompt:

```
Enter number of dimensions or *:
```

The Pascal statements for accessing the heap are exactly as defined in Pascal. However, before these statements are used, the routine HEAP_INIT found in the CSUBLIB module HEAP must be called. This takes the name of the COM to be used for heap as a parameter.

If there is a heap overflow, a Pascal ESCAPE(-2) is generated by the heap routine, which is turned into BASIC error 398.

This is an example of a CSUB that uses the Pascal heap:

```
MODULE HEAPSTUFF;

$STACKCHECK OFF$
$IOCHECK OFF$

$SEARCH 'CSUBLIB'$

IMPORT HEAP;

EXPORT
  PROCEDURE USEHEAP...

IMPLEMENT
  PROCEDURE USEHEAP...
  TYPE
    myheap = RECORD...

  VAR
    x : ^myheap;

  BEGIN
    heap_init('Heap');     {'Heap' -- the BASIC COM block.}
    NEW(x);
    ...
  END;
END.  {module HEAPSTUFF}
```

There are two ways that you may include heap accessing statements in a CSUB. If NEW, MARK, and RELEASE are adequate to access heap, it is necessary to link the COMSTUFF and HEAP module to your Pascal module. This is because COM is used to implement heap.

When the Pascal statement DISPOSE is used in the CSUB it is necessary to do two things. First you must use the $HEAP_DISPOSE$ Compiler option. Second, the HPM module must be linked to the Pascal module.

Linking the HEAP module must be done differently from other links. It must be done even before the BUILDC program is done. It must be done this way because the module contains global variable space. This global space requirement must be known to BUILDC so that the proper amount of space in the global COM can be assured. To pre-link the HEAP module (or any module) to your Pascal module:

1. Give the Librarian command from the Main Command Level. Press [L].
2. Press [O] for Output and type:

   VOL:TEMP

   Where VOL: is your chosen volume name and TEMP is the name of the new file.
3. Press [L] to Link.
4. Press [I] for Input and type:

   VOL:CSUBFILE

   Where VOL: contains the CSUB file and CSUBFILE is the name of the file.
5. Press [A] to link All of the modules into the new file.
6. Press [I] for another Input file and type:

   VOL:CSUBLIB

   Where VOL: is the volume containing your CSUBLIB utilities.
7. Press [M] for Module and type:

   HEAP

8. Press [T] to Transfer (link) the HEAP module to your Pascal module.
9. Press [L], [K] and [Q] to Link, Keep and Quit.

When you're finished, use the Filer to rename the TEMP file or execute BUILDC and GENC giving them the TEMP file name.

A stream file can be set up to do this link. It might look like:

```
=UUser CSUB file?
=PEnter 'P' for linkmap:
=GEnter Filename if it applies.
*
* The parameters "P" and "G"  can be used to send the
* linkmap to an SRM spool printer.
* The user needs only to enter the spooler file name
* (eg. /PRINTER/MY_FILE ) and this
* stream file code will append the .ASC to it.
*
LO USER_TEMP
L@P@G.ASC
I @U
AI CSUBLIB
M HEAP
ALKQ
```

Afterwards, USER_TEMP is either renamed or used as the input file to BUILDC and GENC.

Trapping Errors

The Pascal ESCAPE statement may be used to do error reporting from a CSUB.

```
ESCAPE(some_number);
```

This statement causes one of three things to happen. If the number is a standard BASIC error number, the number will be displayed along with the error message. If the number is not a standard error, just the number will be displayed. If the number is an illegal BASIC error number, a SYSTEM ERROR will result.

The following table shows the possible error numbers and how they are interpreted by the system:

Pascal Escapes	BASIC Definition
-32767... -891	SYSTEM ERROR
-890... -881	reserved by OS
-880... -28	SYSTEM ERROR
-27... -1	ERROR (escape_code + 400)
0	Pascal exit, no error
1...32767	ERROR (escape_code MOD 1000)

Another error recovery technique is to use Pascal TRY-RECOVER sequence. TRY-RECOVER is documented in the *Pascal 2.0 User's Manual.*

Linking Other Modules and Libraries

It is important to understand the principles of linking in Pascal before attempting to link additional modules and libraries into a CSUB. Linking is described in detail in the *Pascal 2.0 User's Manual.* If this section has not already been read and understood, it is advisable to take the time to do so before proceeding.

Assembly Language CSUBs

CSUBs may be written in assembly language either as Pascal-type modules or as stand-alone routines. Pascal calling sequences for passing parameters must be observed.

Information about writing assembly language routines can be found in the *Pascal 2.0 User's Manual*. The Assembler chapter, particularly "The Programming System" section, should be read before attempting to write a routine.

Assembly language routines are somewhat limited in the same way as Pascal routines. They must be relocatable at link time. One difference to note is that routines running in the BASIC environment will be running in system mode rather than user mode. This should present no problem since the system mode is a superset of the user mode.

A Simple Routine

The purpose of the following example is to demonstrate the method of handling parameters after a BASIC call. As the BASIC call statement is scanned from left to right, the dimension record and value pointers are pushed onto the stack, each using four bytes. The return address is then pushed onto the stack. All entries should be popped from the stack before returning from the routine. If the return address is left on the stack, a RTS instruction should be used. If the return address is put in an address register, a JMP instruction should be used.

This example does not emulate a Pascal-type module. For this example, give BUILDC a null name (just ENTER) when asked for a module name.

```
        nosyms
        refa    sysglobals
        def     amatmul             (the procedure name)
*************************************************************
* This routine performs an array multiplication MAT R=S*T.
* The BASIC declaration for it is:
*     CSUB Amatmul(INTEGER S(*),T(*),R(*))
* The arrays are declared as:
*     INTEGER S(X,Y),T(Y,Z),R(X,Z)
* where X, Y, and Z can be arbitrary subscripts.
*************************************************************
* d0 - X
* d1 - Y
* d2 - Z
* d3 - K          counts down Z
* d4 - J          counts down Y
* d5 - Z*2        width of T
* d7 - A          result accumulator
* a0 - ^S
* a1 - ^T
* a2 - ^R
* a3 - ^S(I,*)
* a4 - ^T(*,K)
*
amatmul movea.l 8(sp),a2        a2 = dr  (dimentryptr for R)
        movea.l 16(sp),a1       a1 = d1  (dimentryptr for T)
        movea.l 24(sp),a0       a0 = ds  (dimentryptr for S)

        moveq   #2,d0           check if dims=2 for each array
        cmp.b   (a0),d0
        bne     err16
        cmp.b   (a1),d0
        bne     err16
        cmp.b   (a2),d0
        bne     err16
```

```
        move.w  6(a0),d0        d0 = X
        move.w  10(a0),d1       d1 = Y
        move.w  10(a1),d2       d2 = Z
        cmp.w   6(a1),d1        check if dims are consistent
        bne     err15
        cmp.w   6(a2),d0
        bne     err15
        cmp.w   10(a2),d2
        bne     err15

        movea.l 4(sp),a2        a2 = ^R  (result address)
        movea.l 20(sp),a0       a0 = ^S  (addr of first operand)


        move.w d2,d5
        ext.l d5
        asl.l #1,d5 d5 = Z*2 (for shortint offset)
        subq.w #1,d0
        subq.w #1,d1
        subq.w #1,d2

Iloop   move.w d2,d3            re-init K counter
        movea.l 12(sp),a1       a1 = ^T (multiplier address)

Kloop   movea.l a0,a3
        movea.l a1,a4
        move.w  d1,d4
        moveq   #0,d7

Jloop   move.w  (a3)+,d6        load S(I,J) element
        muls    (a4),d6         mult by T(J,K) element
        add.w   d6,d7           sum result in d7
        adda.l  d5,a4           add offset (Z*2) to ^T
        dbf     d4,Jloop

        move.w  d7,(a2)+        store in R(I,J)
        addq.l  #2,a1           set for next collumn
        dbf     d3,Kloop

        movea.l a3,a0           new position in S matrix
        dbf     d0,Iloop

        movea.l (sp)+,a0        pop return address
        adda.w  #24,sp          discard parameters
        jmp     (a0)            return

err15   move.w  15,sysglobals-2(a5)     escape(15);
        trap    #10         {report BASIC error 15}
err16   move.w  16,sysglobals-2(a5)     escape(16);
        trap    #10         {report BASIC error 16}

        end
```

An IMPORTable Module

By creating a module, you gain the ability to IMPORT and $SEARCH$ the routine in a Pascal program. Below, the example is modified so that it is a Pascal module. Essentially, the routine was given a module name and interface text. The MNAME and SRC psuedo-ops are used to accomplish this. You must also append the module name to the procedure name when labeling the entry point and DEFing the symbol. If you give BUILDC a module name, it assumes that the module name is appended to the procedure name (entry point) as in a Pascal module. This can be observed by checking the DEF table of any Pascal module. All the procedure entry points have the module name appended to the procedure names.

Since the only changes to the example occur at the top, only the top portion of the example is given.

```
          MNAME   ASMMOD
          SRC MODULE  ASMMOD;
          SRC IMPORT  CSUBDECL;
          SRC EXPORT
          SRC     TYPE ARRAYPTR = ^ARRAY[1..1000] OF BINTVALTYPE;
          SRC     PROCEDURE AMATMUL
                            (DS : DIMENTRYPTR; S : ARRAYPTR;
                             DT : DIMENTRYPTR; T : ARRAYPTR;
                             DR : DIMENTRYPTR; R : ARRAYPTR);
          SRC END;

          nosyms
          refa    sysglobals

          DEF     ASMMOD_AMATMUL      (formerly  "def   amatmul")

*************************************************************
* This routine performs an array multiplication MAT R=S*T.
* The BASIC declaration for it is:
*     CSUB Amatmul(INTEGER S(*),T(*),R(*))
* The arrays are declared as:
*     INTEGER S(X,Y),T(Y,Z),R(X,Z)
* where X, Y, and Z can be arbitrary subscripts.
*************************************************************
* d0 - X
* d1 - Y
* d2 - Z
* d3 - K          counts down Z
* d4 - J          counts down Y
* d5 - Z*2        width of T
* d7 - A          result accumulator
* a0 - ^S
* a1 - ^T
* a2 - ^R
* a3 - ^S(I,*)
* a4 - ^T(*,K)
*

ASMMOD_AMATMUL  movea.l 8(sp),a2      ( changed label )
                movea.l 16(sp),a1
                movea.l 24(sp),a0
```

Appendix
Useful TYPE Declarations

CSUBDECL is a library file containing one module by the same name. These are the constructs found in the declaration section of CSUBDECL:

```
module csubdecl;

export
  const stringlimit = 32767;     {maximum length of a string}
  maxdim = 6;                    {maximum dimensions in an array}
  maxarraysize = 16777215;       {maximum bytes in an array}

type
byte = 0..255;
  shortint = -32768..32767;     {two byte integer}
bintvaltype = shortint;        {BASIC integer}

  bstringvaltype = record       {BASIC string type}
    len : shortint;
  c   : packed array[1..stringlimit] of char;
  end;

  valuetype = (breal, binteger, bstring, spare1,
               spare2, spare3, spare4, spare5);

  dimtype = 0..maxdim;

  boundentry = record       {describes array bound}
    low : shortint;         {lower limit}
    length : shortint;      {number of elements}
  end;

  boundtype = array[1..maxdim] of boundentry; {array bounds}

  dimentry = packed record              {dimension record structure}
    case dimtype of
      0: (maxlen : shortint);            {string scalar}
      1,2,3,4,5,6:                       {array}
        (dims : byte;                    {number of dimensions}
         totalsize: 0..maxarraysize;     {total size of an array}
        case valuetype of
          breal, binteger:              {numeric array}
            (bound : boundtype);        {dimension boundaries}
          bstring:                      {string array}
            (stringarray : packed record
              maxlen : shortint;        {max string length}
              bound : boundtype;        {dimension boundaries}
            end))
end;

  dimentryptr = ^dimentry;          {pointer to dimension record}
  binteger_parm = ^bintvaltype;        {pointer to BASIC integer}
  breal_parm = ^real;                  {pointer to real number}
  bstring_parm = ^bstringvaltype;      {pointer to BASIC string}
```

The following declarations are found in module COMSTUFF:

```
module comstuff;

export
      type
        comlabeltype = string[18];

      function find_com       {finds the address of a given COM block}
        (seeked_label : comlabeltype)  {COM label name}
        : anyptr;
```

The following declarations are found in module HEAP:

```
module heap;

export
  type
    heapnametype = string[18];

  procedure heap_init          {initialize a heap}
    (heapname : heapnametype);  {COM block used for heap}
```

This is the STACKSPACE function declaration in module CSUBENTRY.

```
module csubentry;

export
  function stackspace : integer;  {return available stack space}
end;
```