

# Developing CSUBs for HP BASIC/UX 6.2



HP Part No. E2040-90003  
Printed in USA

---

## **Notice**

The information contained in this document is subject to change without notice.

Hewlett-Packard Company (HP) shall not be liable for any errors contained in this document. HP MAKES NO WARRANTIES OF ANY KIND WITH REGARD TO THIS DOCUMENT, WHETHER EXPRESS OR IMPLIED. HP SPECIFICALLY DISCLAIMS THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. HP shall not be liable for any direct, indirect, special, incidental, or consequential damages, whether based on contract, tort, or any other legal theory, in connection with the furnishing of this document or the use of the information in this document.

## **Warranty Information**

A copy of the specific warranty terms applicable to your Hewlett-Packard product and replacement parts can be obtained from your local Sales and Service Office.

## **Restricted Rights Legend**

Use, duplication or disclosure by the U.S. Government is subject to restrictions as set forth in subparagraph (c)(1)(ii) of the Rights in Technical Data and Computer Software clause of DFARS 252.227-7013.

Use of this manual and magnetic media supplied for this product are restricted. Additional copies of the software can be made for security and backup purposes only. Resale of the software in its present form or with alterations is expressly prohibited.

Copyright © Hewlett-Packard Company 1987, 1988, 1989, 1990, 1991

This document contains information which is protected by copyright. All rights are reserved. Reproduction, adaptation, or translation without prior written permission is prohibited, except as allowed under the copyright laws.

Copyright © AT&T Technologies, Inc. 1980, 1984, 1986

Copyright © The Regents of the University of California 1979, 1980, 1983,  
1985-86

This software and documentation is based in part on the Fourth Berkeley  
Software Distribution under license from the Regents of the University of  
California.

---

## **Printing History**

First Edition - August 1991

# Contents

---

<b>1. Introduction</b>	
Prerequisites . . . . .	1-1
Notations Used in This Manual . . . . .	1-2
When to Use CSUBs . . . . .	1-3
The System Components . . . . .	1-4
<b>2. Writing Pascal CSUBs</b>	
Steps for Creating a Pascal CSUB . . . . .	2-1
An Example: Finding the String . . . . .	2-3
Step 1: Create a BASIC Program that Calls the CSUB . . . . .	2-3
Steps 2 and 3: Write, Compile, and Debug the CSUB . . . . .	2-4
Step 4: Generate a CSUB Object File . . . . .	2-5
Step 5: Generate a BASIC PROG File (rmbuildc) . . . . .	2-5
Steps 6 and 7: LOAD and RUN the CSUB . . . . .	2-6
A Closer Look at Pascal CSUB Components . . . . .	2-6
A Closer Look at Parameter Passing . . . . .	2-7
Passing Parameters by Reference . . . . .	2-8
Parameter Types . . . . .	2-9
REAL . . . . .	2-11
COMPLEX . . . . .	2-11
INTEGER . . . . .	2-12
Strings . . . . .	2-12
I/O Paths . . . . .	2-13
Arrays . . . . .	2-14
Defining BASIC and Pascal Arrays . . . . .	2-17
Dimensioning an Array . . . . .	2-17
Redimensioning an Array . . . . .	2-17
Declaring the Value Area of a String . . . . .	2-17
Useful TYPE Declarations . . . . .	2-18
Optional Parameters . . . . .	2-19

Accessing BASIC COM from a CSUB . . . . .	2-20
Accessing Global Variables . . . . .	2-22
A Closer Look at Linking CSUB Object Files . . . . .	2-22
Using the Linker . . . . .	2-22
Specifying the librmb.a Library . . . . .	2-23
Resolving External References . . . . .	2-23
A Closer Look at Executing rmbbuildc . . . . .	2-24
Procedure for Using rmbbuildc . . . . .	2-24
Step 1: Executing rmbbuildc . . . . .	2-24
Step 2: Entering a PROG File Name . . . . .	2-25
Step 3: Naming CSUB Object Files . . . . .	2-25
Step 4: Specifying CSUB Interfaces . . . . .	2-25
rmbbuildc Errors . . . . .	2-28
A Closer Look at Managing CSUBs from BASIC . . . . .	2-31
Loading CSUBs into a BASIC Program . . . . .	2-31
Deleting CSUBs . . . . .	2-32
Example of Deleting CSUBs . . . . .	2-32
Handling CSUB Run-time Errors . . . . .	2-33
Trapping Errors . . . . .	2-33
Reporting Errors to BASIC . . . . .	2-34
Accessing System Resources . . . . .	2-35
Allocating Dynamic Memory . . . . .	2-35
Simple Keyboard and Printer I/O . . . . .	2-36
Device I/O . . . . .	2-38
BASIC File I/O . . . . .	2-38

**3. Writing C and Assembly CSUBs**

Steps for Creating a C and Assembly CSUB . . . . .	3-1
An Example: Passing Parameters . . . . .	3-2
Step 1: Create a BASIC Program that Calls the CSUB . . . . .	3-3
Steps 2 and 3: Write, Compile, and Debug the C CSUB . . . . .	3-3
Steps 2 and 3: Write, Compile, and Debug the Assembly CSUB . . . . .	3-4
Step 4: Generate a CSUB Object File . . . . .	3-5
Step 5: Generate a BASIC PROG File (rmbbuildc) . . . . .	3-5
Steps 6 and 7: LOAD and RUN the CSUB . . . . .	3-6
A Closer Look at C CSUB Components . . . . .	3-6
A Closer Look at Parameter Passing . . . . .	3-7

Passing Parameters by Reference . . . . .	3-7
Parameter Types . . . . .	3-8
REAL . . . . .	3-10
COMPLEX . . . . .	3-10
INTEGER . . . . .	3-10
Strings . . . . .	3-11
I/O Paths . . . . .	3-12
Arrays . . . . .	3-13
Defining BASIC and C Arrays . . . . .	3-15
Dimensioning an Array . . . . .	3-15
Redimensioning an Array . . . . .	3-15
Declaring the Value Area of a String . . . . .	3-15
Useful type Declarations . . . . .	3-16
Optional Parameters . . . . .	3-17
Accessing BASIC COM from a CSUB . . . . .	3-18
Defining a C Structure . . . . .	3-18
A Closer Look at Linking CSUB Object Files . . . . .	3-20
Using the Linker . . . . .	3-20
Specifying the librb.a Library . . . . .	3-21
Resolving External References . . . . .	3-21
A Closer Look at Executing rmbuildc . . . . .	3-22
Procedure for Using rmbuildc . . . . .	3-22
Step 1: Executing rmbuildc . . . . .	3-22
Step 2: Entering a PROG File Name . . . . .	3-23
Step 3: Naming CSUB Object Files . . . . .	3-23
Step 4: Specifying CSUB Interfaces . . . . .	3-23
rmbuildc Errors . . . . .	3-26
A Closer Look at Managing CSUBs from BASIC . . . . .	3-29
Loading CSUBs into a BASIC Program . . . . .	3-29
Deleting CSUBs . . . . .	3-30
Example of Deleting CSUBs . . . . .	3-30
Handling CSUB Run-time Errors . . . . .	3-31
Trapping Errors . . . . .	3-31
Reporting Errors to BASIC . . . . .	3-32
Accessing System Resources . . . . .	3-33
Allocating Dynamic Memory . . . . .	3-33
Simple Keyboard and Printer I/O . . . . .	3-33
Device I/O . . . . .	3-34

BASIC File I/O . . . . .	3-35
<b>4. Writing FORTRAN CSUBs</b>	
Steps for Creating a FORTRAN CSUB . . . . .	4-1
An Example: Finding the String . . . . .	4-2
Step 1: Create a BASIC Program that Calls the CSUB . . . . .	4-3
Steps 2 and 3: Write, Compile, and Debug the CSUB . . . . .	4-3
Step 4: Generate a CSUB Object File . . . . .	4-4
Step 5: Generate a BASIC PROG File (rmbbuilc) . . . . .	4-4
Steps 6 and 7: LOAD and RUN the CSUB . . . . .	4-5
A Closer Look at FORTRAN CSUB Components . . . . .	4-5
A Closer Look at Parameter Passing . . . . .	4-6
Passing Parameters by Reference . . . . .	4-6
Parameter Types . . . . .	4-6
REAL . . . . .	4-7
COMPLEX . . . . .	4-7
INTEGER . . . . .	4-8
Strings . . . . .	4-8
I/O Paths . . . . .	4-10
Arrays . . . . .	4-10
Defining BASIC and FORTRAN Arrays . . . . .	4-13
Dimensioning an Array . . . . .	4-13
Redimensioning an Array . . . . .	4-13
Declaring the Value Area of a String . . . . .	4-14
Optional Parameters . . . . .	4-14
Accessing BASIC COM from a CSUB . . . . .	4-15
Using the basic_com Routine . . . . .	4-15
Defining a FORTRAN COM Block Structure . . . . .	4-16
A Closer Look at Linking CSUB Object Files . . . . .	4-17
Using the Linker . . . . .	4-18
Specifying the libmb.a Library . . . . .	4-18
Resolving External References . . . . .	4-19
A Closer Look at Executing rmbbuilc . . . . .	4-19
Procedure for Using rmbbuilc . . . . .	4-20
Step 1: Executing rmbbuilc . . . . .	4-20
Step 2: Entering a PROG File Name . . . . .	4-20
Step 3: Naming CSUB Object Files . . . . .	4-20
Step 4: Specifying CSUB Interfaces . . . . .	4-21



rmbuildc Errors . . . . .	4-23
A Closer Look at Managing CSUBs from BASIC . . . . .	4-26
Loading CSUBs into a BASIC program . . . . .	4-26
Deleting CSUBs . . . . .	4-27
Example of Deleting CSUBs . . . . .	4-27
Handling CSUB Run-time Errors . . . . .	4-28
Trapping Errors . . . . .	4-28
Reporting Errors to BASIC . . . . .	4-29
Accessing System Resources . . . . .	4-29
Simple Keyboard and Printer I/O . . . . .	4-30
Device I/O . . . . .	4-31
BASIC File I/O . . . . .	4-31

## 5. CSUB Prototyper Utility

Why Use the CSUB Prototyper? . . . . .	5-1
Creating CSUBs . . . . .	5-1
Calling CSUBs Dynamically . . . . .	5-3
Using the Prototyper to Create a CSUB . . . . .	5-3
Writing CSUB Routines in C . . . . .	5-3
Step 1: Using the CSUB Prototyper in a BASIC program . . . . .	5-4
Step 2: Exiting BASIC to HP-UX . . . . .	5-5
Step 3: Writing C Subroutines . . . . .	5-6
Step 4: Generating a Relocatable Object File . . . . .	5-7
Step 5: Running the BASIC Program . . . . .	5-7
Deciding Whether or Not to Create a Standard CSUB . . . . .	5-8
Reasons for Choosing the First Option . . . . .	5-8
Reasons for Choosing the Second Option . . . . .	5-8
Passing Parameters . . . . .	5-9
Parameter Passing Conventions . . . . .	5-9
Mapping of Parameter Types . . . . .	5-11
Handling Prototyper Errors . . . . .	5-14

**6. Porting Pascal Workstation Assembly CSUBs**

Prerequisites . . . . .	6-1
Using atrans . . . . .	6-2
Copying a Pascal Workstation CSUB . . . . .	6-2
Executing the atrans Command. . . . .	6-3
Modifying the Translated CSUB . . . . .	6-5
What Was Changed? . . . . .	6-6
Completing Your Assembly CSUB . . . . .	6-7
Copy the Calling BASIC Program to HP-UX . . . . .	6-7
Execute Steps 4 through 7 of the Pascal CSUB Procedure . . . . .	6-8

**A. File Access Reference**

FAL_CLOSE . . . . .	A-2
FAL_CREATE . . . . .	A-4
FAL_CREATE_ASCII . . . . .	A-5
FAL_CREATE_BDAT . . . . .	A-6
FAL_EOF . . . . .	A-7
FAL_LOADSUB_ALL . . . . .	A-9
FAL_LOADSUB_NAME . . . . .	A-10
FAL_OPEN . . . . .	A-11
FAL_POSITION . . . . .	A-13
FAL_PURGE . . . . .	A-15
FAL_READ . . . . .	A-16
FAL_READ_BDAT_INT . . . . .	A-18
FAL_READ_STRING . . . . .	A-20
FAL_WRITE . . . . .	A-22
FAL_WRITE_BDAT_INT . . . . .	A-24
FAL_WRITE_STRING . . . . .	A-26

**B. Keyboard and CRT I/O Reference**

CLEAR_SCREEN . . . . .	B-2
CONTROLCRT . . . . .	B-3
CONTROLKBD . . . . .	B-4
CRTREADCHAR . . . . .	B-5
CRTSCROLL . . . . .	B-6
CURSOR . . . . .	B-8
DISP_AT_XY . . . . .	B-10
READ_KBD . . . . .	B-12

SCROLLDN . . . . .	B-13
SCROLLUP . . . . .	B-14
STATUSCRT . . . . .	B-15
STATUSKBD . . . . .	B-16
SYSTEMD . . . . .	B-17

**Index**

## Figures

---

2-1. REAL, INTEGER, or COMPLEX Array Dimension Record . . . . .	2-15
2-2. String Array Dimension Record . . . . .	2-16
3-1. REAL, INTEGER, or COMPLEX Array Dimension Structure . . . . .	3-13
3-2. String Array Dimension Structure . . . . .	3-14
4-1. REAL, INTEGER, or COMPLEX Array Dimension Record . . . . .	4-11
4-2. String Array Dimension Record . . . . .	4-12

# Tables

---

1-1. CSUB Utilities Provided . . . . .	1-4
2-1. Equivalent Pascal and BASIC Parameter Types . . . . .	2-10
2-2. rmbuildc Errors . . . . .	2-29
2-3. Error Numbers . . . . .	2-34
2-4. Keyboard and CRT I/O Routines . . . . .	2-37
2-5. File Access Routines . . . . .	2-39
3-1. Equivalent C and BASIC Parameter Types . . . . .	3-9
3-2. rmbuildc Errors . . . . .	3-27
3-3. Keyboard and CRT I/O Routines . . . . .	3-34
3-4. File Access Routines . . . . .	3-36
4-1. Equivalent FORTRAN and BASIC Parameter Types . . . . .	4-7
4-2. rmbuildc Errors . . . . .	4-24
4-3. Keyboard and CRT I/O Routines . . . . .	4-30
4-4. File Access Routines . . . . .	4-32
5-1. Comparison of CSUB Creation Procedures . . . . .	5-2
5-2. Mapping Between Actual and Formal Parameters of a CSUB . . . . .	5-11
5-3. Mapping Between BASIC Return Value Types and Prototyper execute CSUBs or Functions . . . . .	5-12
5-4. Prototyper CSUB Errors . . . . .	5-14
6-1. Comparison of Program Segments . . . . .	6-6



## Introduction

---

A compiled subprogram (CSUB) is a routine written in Pascal, C, FORTRAN, or assembly language on HP-UX and transformed into a subprogram callable from BASIC. Either a single CSUB or a library of CSUBs may be generated using this technique. In BASIC, the CSUB is loaded into memory using the LOAD or LOADSUB command and called like any other BASIC subprogram.

---

### Prerequisites

This manual assumes you are familiar with the BASIC language, editor, data types, and subprograms.

To create CSUBs, you should also know how to use HP-UX to write and compile programs. This includes:

- a good working knowledge of an editor such as `vi`.
- an understanding of the different compilers and assemblers of the CSUB languages that you are going to be using (`pc` for Pascal, `cc` for C, `fc` for FORTRAN, and `as` for assembly).



To link the necessary modules and libraries into a CSUB object file, you should understand the linking procedures associated with the `ld` command.

Learn these things before proceeding with this manual.

---

## Notations Used in This Manual

The following are notations you should be familiar with:

**literal** This font is used for listing example programs, specifying file names, CSUB programming language variables and types, and output from commands. It is also used for specifying computer input that you must type in character for character. For example if you are told to type `rmb` , you would type the characters `rmb` and press the  key.

*variable* This font is used to specify variables for which you must supply a value. For example,

**MSI "dir"**

is a command to change directories (explained later). The variable "*dir*" is a directory whose name you have to supply for the command to work.

**KEYWORD** This font indicates keywords that are used in BASIC commands or statements. For example, **LOADSUB** is the BASIC command used to load compiled subprograms (CSUBs) into memory.

When the term BASIC is used in this manual, it is referring to the implementation of BASIC on HP-UX.



---

## When to Use CSUBs

CSUBs are used to fill a number of needs:

- The compiled or assembled routines are in 68000-family machine code and have a speed advantage over BASIC interpreted code. Computational routines are faster.
- CSUBs are in object code on the disk, and therefore, source code does not have to be released with them. This provides a greater measure of security for programs employing them.
- CSUBs are useful when libraries of routines are needed for both the CSUB language and BASIC environments. By writing the routines in Pascal, for example, you only need to write the routine once as a CSUB rather than once in BASIC and once in Pascal.

For example, when a set of benchmark programs were run, performance using CSUBs was far superior to BASIC. In these benchmarks, REAL and INTEGER loops, transcendental (e.g., functions such as `sin(x)`, `log(x)`, etc.) and non-transcendental math, quick and shell sorts, and matrix multiplication were compared. For the quick sort, the CSUB ran many times faster than a similar BASIC routine. For transcendental math, the CSUB was found to be marginally faster.

The price paid for this speed is the size of the routines. CSUB code will always be larger than comparable BASIC code. The reason for this size increase is that BASIC source code is represented internally in a tokenized RPN (Reverse Polish Notation) form. This results in space savings, even compared against the binary code of a CSUB.

---

## The System Components

The CSUB utility consists of several different system components, a library, an include file, and a program.

The library, `librmb.a`, contains the necessary code to link CSUB code with BASIC and any CSUB utilities provided.

**Table 1-1. CSUB Utilities Provided**

File	Contents
<code>csfa.o</code>	miscellaneous BASIC file I/O routines
<code>kbdcrct.o</code>	miscellaneous BASIC keyboard and crt I/O routines
<code>csubdecl.o</code>	useful parameter type declarations for Pascal CSUBs

The include file, `csubdecl.h`, contains useful parameter type declarations for C language CSUBs.

The program, `rmbbuildd`, interactively prompts you for information about the subprograms (CSUBs) being generated. It accepts a fully linked CSUB object file and generates as output a BASIC PROG file.

## Writing Pascal CSUBs

---

This chapter covers the process of writing Pascal CSUBs. When it is appropriate, information on porting existing CSUBs from Pascal Workstations to HP-UX is also included.

When you are developing a system that involves the use of CSUBs, you should:

- write the BASIC program.
- determine what the CSUB should do, the parameters to be passed, and which variables should be accessible (*global*) to both the BASIC program and the CSUB.
- develop the CSUB; a listing of the BASIC program can be very helpful as reference during this task.

---

### Steps for Creating a Pascal CSUB

The following steps present an overview of the process needed to create a CSUB and the results of those steps. Some porting information is also included (for those who already have Pascal Workstation CSUBs written). The CSUB related steps will be described in detail in later sections.

1. Enter BASIC, create and store the program that will call the CSUB. This program will contain CALLs to the CSUB, but the latter need not be implemented as it will be loaded later. You only need to decide what the subprogram will do and design the interface (parameters, COM, etc.) between BASIC and the CSUB.
2. Leave BASIC, enter HP-UX, and write the CSUB. You might, for example, use the vi editor to create the CSUB. See the sections “A Closer Look at Pascal CSUB Components” and “A Closer Look at Parameter Passing”

for details on how to organize your Pascal modules and define your CSUB parameters.

3. Compile and debug the CSUB as much as possible by writing a Pascal test program. You may want to use the Pascal debugger, `pdb`, for this testing task.

If you are porting an existing CSUB module, you will likely get compiler errors because of invalid compiler directives or missing libraries. These errors will necessitate some changes to the source code of the modules. For a list of the compiler directives supported by the HP-UX Pascal compiler and for the procedure on how to build libraries, refer to the “HP Pascal Language Reference” manual. You should also see the “Specifying the `librmb.a` Library” section for additional information on CSUB libraries.

Before attempting to execute any ported CSUB modules, carefully read the sections “A Closer Look at Parameter Passing” and “Accessing System Resources” for differences in the CSUB parameter types and the system calls supported. As examples of notable differences, a Pascal `REAL` type in HP-UX is now a 32-bit quantity, and the Pascal Workstation I/O library is no longer available from CSUBs (use HP-UX’s Device I/O Library (`DIL`) instead).

4. Use the `ld` command to generate a fully linked relocatable object file containing all the CSUBs and any necessary HP-UX library support routines. For example,

```
ld -rd -a archive csub.o -u _printf -lrmb -o csub
```

would create a fully linked relocatable object file called `csub` using the compiled file `csub.o` and the library `librmb.a`. See the section “A Closer Look at Linking CSUB Object Files” for details.

5. Execute `rmbbuildd` and answer its prompts. This program generates the final BASIC PROG file. See the section “A Closer Look at Executing `rmbbuildd`” for details on how to answer the prompts.
6. Enter BASIC and load the BASIC PROG file from the keyboard or from the BASIC program using `LOADSUB`. This statement generates the necessary statements in your BASIC program to call CSUBs. See the section “A Closer Look at Managing CSUBs from BASIC” for details.

7. RUN the BASIC program which calls the desired CSUBs using the BASIC CALL or implied CALL statement.

### An Example: Finding the String

This section goes through each step of creating an example Pascal CSUB and how the CSUB is used in a BASIC program. All files used in the HP-UX environment are read from and written to the current directory. In the BASIC environment, the CSUB is loaded from the MASS STORAGE IS device (directory).

This simple program fills an array of string variables and then calls a Pascal CSUB which determines 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. If the INTEGER variable **Yes** comes back with a value other than zero, it comes back pointing to the array element containing the matching string.

#### Step 1: Create a BASIC Program that Calls the CSUB

Enter BASIC, edit and store this program in a file named FSTR. This file can be found in the directory called /usr/lib/rmb/demo.

```

10  LOADSUB ALL FROM "FIND_STRING"
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$(5)="FAVORITE COLOR?"
100 File$(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

```

## Steps 2 and 3: Write, Compile, and Debug the CSUB

Enter HP-UX, edit, compile, and debug the following module called `test` and save it in the file named `string.p`. This file can be found in the directory called `/usr/lib/rmb/demo`.

```

MODULE test;

$SEARCH '/usr/lib/librmb.a'$

IMPORT csubdecl;

EXPORT
  TYPE
    string_type = RECORD
      len : shortint;
      c   : 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].c[j] = strx.c[j])
                DO j := j + 1;
              IF (filex[i].c[j] = strx.c[j]) THEN yes := i;
            END;
          IF yes = 0 THEN i := i + 1;
        END;
      END;
    END;
  END;

```

```
END;
END.
```

#### Step 4: Generate a CSUB Object File

Link the code file of the Pascal module with the BASIC CSUB library `librmb.a` to generate a fully linked relocatable CSUB object file. The HP-UX `ld` command should be used for this purpose, as follows:

```
ld -rd -a archive string.o -u _printf -lrmb -o string
```

#### Step 5: Generate a BASIC PROG File (rmbuildc)

Execute the `rmbuildc` program and answer the prompts as shown below. Notice that a stream file is generated by the response to the first prompt; you can use this file the next time you execute the program (i.e. `rmbuildc < stream`) to remove the need to interactively answer the prompts again.

RMB-UX Compiled Subprogram File Generator (Version 1.1)

```
Stream file name: stream
Output BASIC PROG file: FIND_STRING
CSUB object file name(s): string
Module name: test
  CSUB 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/C for Integer/Real/Complex): i
    Is this an array? (y/n): n
    Is this an optional parameter? (y/n): n
  Parameter name: yes
    Parameter type (I/R/C for Integer/Real/Complex): i
    Is this an array? (y/n): n
    Is this an optional parameter? (y/n): n
  Parameter name: Return
    Is there COM in this CSUB? (y/n): n

CSUB name: Return
```

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

### Steps 6 and 7: LOAD and RUN the CSUB

In this example, `FIND_STRING` is automatically loaded from the BASIC program. Therefore, you only need to re-enter the BASIC system, `LOAD "FSTR"`, and `RUN` the program. The output should be:

```
The string was found in number 4
```

---

## A Closer Look at Pascal CSUB Components

Certain Pascal concepts and constructs should be understood so that you can apply them in CSUBs. The CSUBs are contained in a **module** which can be compiled independently. The `EXPORT` section of the module defines the procedures that will be CSUBs. The `IMPORT` statement names all other modules on which the present one depends. If one of the modules specified is not found in the file of the module, the Pascal compiler will then search **library files** to satisfy the `IMPORT` declarations; the library files to be searched are specified by the `$SEARCH$` directive. Code for the CSUBs is found in the `IMPLEMENT` section of the module, as shown in the example below.

```
MODULE test;

$SEARCH '/usr/lib/librmb.a'$

IMPORT csubdecl;

EXPORT
  PROCEDURE find_string (file_dim : dumentryptr; VAR strx: bstringvaltype);

IMPLEMENT
  PROCEDURE find_string;
  BEGIN
    ...
  END;
END.
```

The Pascal procedures that will become CSUBs should be included in a Pascal `EXPORT` section. Other routines may be included in the `IMPLEMENT` section,

### 2-6 Writing Pascal CSUBs



but unless their names are included in the **EXPORT** section and specified to the **rmbuildc** program, the routines will not show up as separate **CSUB** entry points. The following example shows how two Pascal procedures are exported to make them available as **CSUBS**.

```

$SEARCH '/usr/lib/librmb.a'$

IMPORT csubdecl;

EXPORT
  PROCEDURE sqrit(x: binteger_parm; y: breal_parm);
  PROCEDURE readit(z: binteger_parm);

```

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 from **BASIC**.

Declarations of types and procedures from other modules may also be imported into the **CSUB** module. The libraries containing those modules should then also be listed in the **\$SEARCH\$** directive.

## A Closer Look at Parameter Passing

In order to be useful, a **CSUB** needs the ability to exchange data with the calling **BASIC** program. This section describes the different means of performing this data exchange. The primary method consists of defining the **CSUB** parameters to match those passed by **BASIC**. This method requires special attention to the different parameter types and formats of **BASIC** variables. The second method for a **CSUB** to exchange data with a **BASIC** program is via **COM** blocks. Again, this method necessitates the special handling of the variables defined in the blocks, according to their type and format.

## Passing Parameters by Reference

As far as Pascal is concerned, BASIC variables are always passed to a CSUB *by reference*. 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 passed by reference, so they can be used to *receive* BASIC parameters passed by reference.

For example, the following program passes a parameter by reference to obtain a pointer to a REAL variable `realvar`.

```
PROGRAM obvious;
VAR realvar: REAL;

PROCEDURE p(VAR r: REAL);
BEGIN
    r:=-31178.0;
END;

BEGIN
    p(realvar); {Compiler will emit code to pass a pointer}
END.
```

The following program accomplishes the same thing with a parameter passed by value. It passes the pointer to `realvar` by value.

```
PROGRAM not_so_obvious;
TYPE real_ptr = ^REAL;
VAR realvar: REAL;

PROCEDURE p2(rp: real_ptr); {Pass by value}
BEGIN
    rp^:=-31178.0;           {Must use the dereference symbol "^"}
END;

BEGIN
    p2(ADDR(realvar));     {Compiler passes the pointer by value}
END.
```

The two key points to remember are:

- BASIC always passes a pointer to a variable.

- BASIC has no idea if a user-written CSUB has been properly coded to use that pointer.

Note that errors will occur if this distinction is overlooked.

## Parameter Types

The BASIC parameter types are not necessarily the same as their Pascal counterparts. It is important that the type of the parameters of a CSUB be correct so that BASIC and the CSUB can interface properly. This section will explain the types in detail. You should refer to the module `csubdecl` for the definition of the types used below. The definitions may be used to declare either reference or value parameters. The choice is dependent mainly on personal programming style.

The following table provides you with a quick reference to equivalent Pascal and BASIC parameter types.

Table 2-1. Equivalent Pascal and BASIC Parameter Types

BASIC Parameter Type	Pascal Parameter Type
REAL	IMPORT <i>csubdecl</i> : <i>brealvaltype</i> (by reference) or <i>breal_parm</i> (by value)
INTEGER	IMPORT <i>csubdecl</i> : <i>bintvaltype</i> (by reference) or <i>binteger_parm</i> (by value)
COMPLEX	IMPORT <i>csubdecl</i> : <i>bcplxvaltype</i> (by reference) or <i>bcomplex_parm</i> (by value)
<i>string_name</i> \$	IMPORT <i>csubdecl</i> : Two parameter types passed for strings: • <i>dimentrypnr</i> • <i>bstring_parm</i>
<i>array_name</i> [ <i>lower</i> : <i>upper</i> , <i>etc.</i> ] of one of the above numeric parameter types or <i>string_array</i> \$( <i>lower</i> : <i>upper</i> , <i>etc.</i> )[ <i>num_chars</i> ]	IMPORT <i>csubdecl</i> : Two parameter types passed for arrays: • <i>dimentrypnr</i> • <i>ARRAY</i> [ <i>lower</i> .. <i>upper</i> , <i>etc.</i> ] of one of the above numeric or string parameter types.
@ <i>io_path_name</i>	IMPORT <i>csfa</i> : <i>fc_b_type</i> (by reference) or <i>fc_b_ptr_type</i> (by value)

## REAL

A variable defined as a **REAL** in Pascal is not the same as a BASIC **REAL**. The latter is a 64-bit quantity while a Pascal **REAL** variable is only a 32-bit quantity. It is thus necessary to use the **brealvaltype** type for a BASIC **REAL** parameter. For example,

```
PROCEDURE x (VAR y: brealvaltype);
```

is exactly the same as:

```
PROCEDURE x (y_ptr: breal_parm);
```

because **breal\_parm** is defined as a pointer to **brealvaltype**.

If you are porting an existing Pascal Workstation CSUB module, you should review all CSUBs with **REAL** parameters since those parameters are no longer 64-bit quantities, as they were on the Pascal Workstation.

## COMPLEX

A variable defined in BASIC as **COMPLEX** is a floating point value with real and imaginary parts. There is no built-in **COMPLEX** type in Pascal.

A Pascal declaration for a **COMPLEX** value would be:

```
TYPE
  bcmplxvaltype = RECORD
    re : brealvaltype;
    im : brealvaltype;
  END;
```

Thus, you could use:

```
IMPORT csubdecl; {Exports the type bcmplxvaltype}

EXPORT PROCEDURE x (VAR y: bcmplxvaltype); {Pass by reference}
```

or use:

```
IMPORT csubdecl;

EXPORT PROCEDURE x (y_ptr: bcomplex_parm); {Pass pointer by value}
```

because **bcomplex\_parm** is defined as a pointer to **bcmplxvaltype**. If the **VAR** method is used, the values are accessed as **y.re** and **y.im**. If the latter method is used, the values are accessed as **y\_ptr^.re** and **y\_ptr^.im**.

## INTEGER

A variable defined as an **INTEGER** in Pascal is not the same as a BASIC **INTEGER**. The latter is a 16-bit quantity while a Pascal **INTEGER** variable is only a 32-bit quantity. Therefore, to receive a BASIC **INTEGER**, you should either use:

```
IMPORT csubdecl; {Exports the type BINTVALTYPE}

EXPORT PROCEDURE x (VAR y: bintvaltype); {Pass by reference}
```

or use:

```
IMPORT csubdecl;

EXPORT PROCEDURE x (y_ptr: binteger_parm); {Pass pointer by value}
```

because **binteger\_parm** is defined as a pointer to **bintvaltype**.

## Strings

Strings are different in BASIC and Pascal, both in their structure and the way they are passed. The structure of a variable of the Pascal type **STRING** is a one-byte length field followed by the characters of the string. The BASIC string has a two-byte length field followed by its characters.

BASIC passes its strings as two parameters:

- The first parameter is a pointer to a **dimension record**. This record contains information about arrays, strings, their maximum lengths, and their lower and upper bounds. It is expressed in Pascal as a *variant* record of type **dimentry** while its pointer type is **dimentryptr**. For the case of a scalar (non-array) string, the only field in this record is a short integer, a 16-bit quantity, expressing the maximum length of the string.
- The second parameter is a pointer to the string **value area**. This area contains the actual length of the string and its characters. The type of the pointer is **bstring\_parm**.

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

```
MODULE example;
IMPORT csubdecl;

EXPORT
```

```

    PROCEDURE getstring (dim_len: dinttryptr; b:bstring_parm);
IMPLEMENT
    PROCEDURE getstring;
    VAR
        s:STRING[80];
        i:shortint;

    BEGIN
        s:='a string';
        IF STRLEN(s)>dim_len^.maxlen THEN SETSTRLEN (s, dim_len^.maxlen);
        b^.len:=STRLEN(s);
        FOR i:=1 TO b^.len DO b^.c[i]:=s[i];
    END;
END.

```

The above procedure copies a Pascal string into a BASIC string. From this example, you should note how:

- the string parameter is declared,
- an explicit check has been made to insure that the length of the Pascal string is not greater than the maximum length of the BASIC string,
- the Pascal string value is put into the BASIC string value area.

Although a CSUB will receive two parameters for a BASIC string, as far as BASIC is concerned, there is only one actual parameter to be passed, as shown below.

```

...
10 DIM Str$[80]
20 CALL GETSTRING(Str$)
30 PRINT Str$
40 END

```

## I/O Paths

An I/O path is a block of storage used to keep track of the state of a file or I/O device. The size of this block is 190 bytes. See the section “BASIC File I/O” for details on how you would use an I/O path as a file control block with the file access library (`fal`) routines.

The following example shows how to receive an I/O path parameter from BASIC. The module `csfa` in the library `librmb.a` contains the necessary type

declarations to pass I/O path parameters to Pascal. The same parameters can then also be used with the `fal` routines.

```
$SEARCH '/usr/lib/librmb.a'$
```

```
IMPORT csfa; {Exports the type fcb_ptr_type}
```

```
EXPORT PROCEDURE x (y_ptr: fcb_ptr_type); {Pass pointer by value}
```

The typical use of the parameter `y_ptr` in a CSUB would then take the form:

```
fal_open ('MyFile', y_ptr);
```

It is also possible to use:

```
PROCEDURE x (VAR y : fcb_type);
```

but this requires the Pascal function `ADDR`:

```
fal_open ("MyFile", ADDR(y));
```

when using the I/O path.

## Arrays

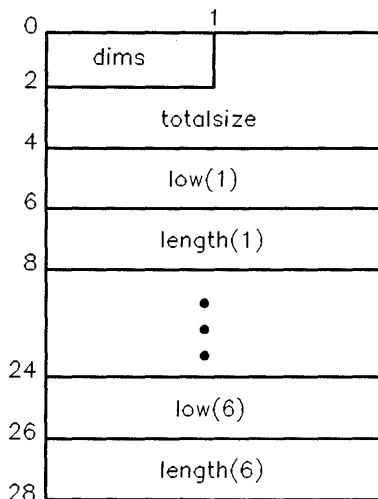
Arrays are passed as two parameters:

- a pointer to the dimension record. The fields in the dimension record, in this case, are more complicated. They are also defined by the `dimentry` record declaration.
- a pointer to the value area.

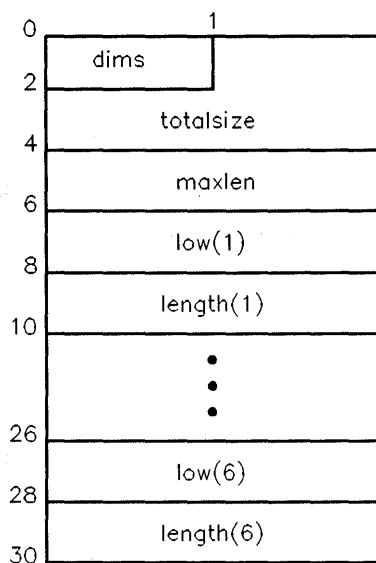


The actual dimension records for the REAL, INTEGER, COMPLEX, and string arrays are represented by Figures 2-1 and 2-2.

- dims            is a byte containing the number of dimensions
- totalsize       is the number of bytes in the entire array
- low(n)         is the lower bound of the nth dimension
- length(n)      is the number of elements in the nth dimension
- maxlen         is the maximum length of any element in a string array.



**Figure 2-1.**  
**REAL, INTEGER, or**  
**COMPLEX Array Dimension**  
**Record**



**Figure 2-2. String Array Dimension Record**

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

```
EXPORT
TYPE
    arrint = ARRAY [1..10] OF bintvaltype; {actual values, not pointers}

    PROCEDURE x (d: dimentryptr; VAR arr: arrint);
```

Again, this is a single parameter in BASIC. So, in this example, BASIC would send an array defined as:

```
INTEGER Arr(1:10)
CALL X(Arr(*))
```

## Defining BASIC and Pascal Arrays

The BASIC and Pascal arrays should be defined the same way. This is not mandatory, but helpful. You should remember that an array with bounds [1..5] is the same as an array with bounds [6..10]; both are five-element arrays. If a BASIC array defined as `INTEGER Arr(6:10)` is passed to a CSUB array parameter defined as `ARRAY [1..5] OF BINTVALTYPE;`, the sixth element in the BASIC array will correspond to the first element in the Pascal CSUB array. Array elements are stored in row-major order in both BASIC and Pascal.

## Dimensioning an Array

The DIM statement should be used in conjunction with the appropriate Pascal array declaration to define the space for the BASIC array. The REDIM statement does not affect the size of that space. It does however affect the BASE and SIZE functions and the `length` and `low` values in the dimension record. Note that a CSUB should therefore check the dimension record of an array parameter to find its current dimensions.

## Redimensioning an Array

If you use the REDIM statement on a multi-dimensional BASIC array, the Pascal declaration of the array will be invalid the next time the array is accessed in a CSUB. Therefore, if you accessed `arr(2,3)` in Pascal, you would not get the same value as `Arr(2,3)` from BASIC. To be immune from the effects of REDIM, you should declare a multi-dimensional BASIC array as a one-dimensional array in Pascal and do explicit subscript calculations based on the information in the dimension record of the array.

## Declaring the Value Area of a String

You should also note that the type `bstringvaltype` should not be used in specifying the value area for a string array since it defines a string with the maximum allowable number of characters. Instead, you should declare a different type, with the maximum allowable number of characters in the array strings set equal to the dimension of the equivalent BASIC string array. Thus, for the following BASIC definition of a string array,

```
DIM S$(1:10)[20]
```

you would define the type for a Pascal string in the array as:

```

TYPE
  string_type = RECORD
    len : shortint;
    c   : PACKED ARRAY [1..20] of CHAR; {Same maximum length as BASIC}
  END;

```

## Useful TYPE Declarations

The module CSUBDECL defines useful types which may be used in the declaration of Pascal CSUB parameters. These declarations are listed below.

```

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}
  brealvaltype = longreal;     {BASIC real}

  bcomplexvaltype = record      {BASIC complex type}
    re : brealvaltype;
    im : brealvaltype;
  end;

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

  valuetype = (breal, binteger, bstring, bsubstring,
               batname, bcomplex, spare2, spare3);

  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, bcomplex: {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 = ^brealvaltype;      {pointer to BASIC real}
bcomplex_parm = ^bcmplxvaltype;  {pointer to complex number}
bstring_parm = ^bstringvaltype;  {pointer to BASIC string}

```

## Optional Parameters

You can declare some or all parameters of a CSUB as *optional* through responses to `rmbuildc`. Optional parameters are those which are not required in the parameter list of the calling code. In a Pascal CSUB, however, there is no distinction between required and optional parameters as both types of parameters have to be listed in the declaration of the CSUB. See the section "Subprograms" in the "BASIC Programming Techniques" manual for more information about optional parameters.

BASIC will pass a NIL pointer to the CSUB when one of its parameters that has been declared as optional is omitted. The CSUB should therefore always make an explicit NIL check before trying to use the value of an optional parameter. Otherwise, a run-time error may occur if the CSUB was compiled with the option `$RANGE ON$`.

This NIL check can be performed in two ways. For example, if the BASIC declaration of a CSUB is:

```
100 CSUB My_csub(REAL R, OPTIONAL REAL Opt)
```

the variable `Opt` will have an address of `NIL` if it is not passed. In Pascal, the `CSUB` can either look like:

```
PROCEDURE my_csub(VAR required, optional: brealvaltype);
BEGIN
  IF ADDR(optional) <> NIL THEN ... {It was passed in}
```

or be defined as:

```
PROCEDURE my_csub(required, optional: breal_parm);
BEGIN
  IF optional <> NIL THEN ... {It was passed in}
```

## Accessing BASIC COM from a CSUB

Another way for a BASIC program and a Pascal `CSUB` to interchange data is via BASIC COM blocks. In order to access a BASIC COM block from a Pascal `CSUB`, you should use the function `find_com` declared in the module `csubdecl`. When passed the name of the COM block that is to be accessed, this function will return a pointer to the beginning of the value area of that block. Since upper and lower case letters are significant in a COM block name, they should be specified the same way BASIC does. To access an unlabeled COM block, you should specify a string with a single blank as the COM block name. When `find_com` cannot find the COM block requested, it returns a `NIL` pointer.

In order to read and store values in the variables of a COM block, you will need to define a Pascal record to map the variables into members of the record. This will require you to know the layout of the block in advance since there is no way of determining this layout from the Pascal `CSUB` at run-time.

In defining the Pascal record for accessing a COM block, you should know that the order of the members in the record should be the opposite from that of the variables in the block. For example, the first variable in the COM block should be mapped into the last member of the record. You should also know that there is nothing to prevent you from inadvertently corrupting the COM block by writing beyond its boundaries or by storing invalid values.

Also, you should note that only the value area of BASIC strings and arrays are stored in a COM block. Therefore, you should omit the specification of the dimension record for those variables in the Pascal record. Finally, you

should follow the same restriction previously mentioned for CSUB string array parameters in specifying the bounds of string arrays in COM blocks.

This is an example of accessing a BASIC COM block defined as:

```
COM /Num1/ INTEGER A,B(1:5),REAL C,D$[10]
```

To access this block from a CSUB, you could use:

```
$SEARCH '/usr/lib/librmb.a'$
...
IMPORT csubdecl;
EXPORT PROCEDURE x...

IMPLEMENT
PROCEDURE x...
TYPE
  strtype = RECORD
    len: shortint;
    c : PACKED ARRAY[1..10] OF CHAR;
  END;
  comtype = PACKED RECORD
    d: strtype;           {Do not use bstringvaltype!}
    c: brealvaltype;
    b: PACKED ARRAY [1..5] OF bintvaltype;
    a: bintvaltype;      {Note the reverse order of the members}
  END;

VAR
  comptr : ^comtype;
  vala : bintvaltype;

BEGIN
  comptr:=find_com('Num1'); {Observe same case of letters as BASIC does}
  vala:=comptr^.a;
  ...
```

A COM block is subject to being moved in memory at RUN, LOAD, and GET time. Therefore, to guarantee that you always get the current location of a block, you should not remember its address in a CSUB but always use the function `find_com` instead. Similarly, you should not attempt to remember the address of a variable defined in a COM block.

## Accessing Global Variables

All global variables declared in Pascal modules linked in CSUB object files are allocated in one contiguous area of memory. Previous implementations of BASIC provided the ability to access this area of memory from BASIC via a COM block. This feature is not supported in this implementation of BASIC because this area of memory is no longer allocated in a COM block.

---

## A Closer Look at Linking CSUB Object Files

After thoroughly testing your CSUBs in a Pascal program, you are now ready to generate a CSUB object file. This object file brings together the CSUB modules and the necessary libraries to satisfy the external references of those modules.

### Using the Linker

The HP-UX linker can be used to generate a fully linked, relocatable CSUB object file suitable for input to the `rmbuildc` program. The syntax for linking is as follows:

```
ld -rd -a archive csub_modules -u _printf -lrm [other_libraries] -o csub_obj_file
```

where:

- `-rd` are the options to retain relocation information in the output file for subsequent re-linking and to force the definition of common storage.
- `-a archive` tells the loader to use archive libraries instead of shared libraries.
- `csub_modules` are the `.o` files generated by the Pascal compiler. They contain the code for CSUBs and should always be followed in the above command line by the library `-lrm`.
- `-u _printf` links needed object files.



- other\_libraries* are libraries that are required to resolve all the references not satisfied by `librmb.a`; they may be your own or HP-UX libraries.
- `-o csub_obj_file` takes the output of the `ld` command and sends it to the file `csub_obj_file`.

## Specifying the librmb.a Library

In addition to providing useful system access routines for CSUBs, this library also specifies all the external symbols from HP-UX libraries which were used in the actual implementation of a specific version of BASIC. Specifying the library immediately after your CSUB modules in the `ld` command will thus ensure the sharing of those symbols between BASIC and your CSUBs. This sharing first removes the need for duplicating the code already available in BASIC in the CSUB object files. More importantly, it is also necessary for the proper operation of libraries, the routines of which maintain global variables which should not be duplicated in the CSUB object files. An example of such routines are the HP-UX memory management routines (`malloc 3C` includes: `calloc`, `free`, `mallopt`, `mallinfo`, and `realloc`).

## Resolving External References

In general, specifying the library `librmb.a` alone in your link command will be sufficient to resolve all external references from your CSUB modules; this is because BASIC uses most of the common HP-UX libraries in its implementation.

Nevertheless, to verify that your CSUB object file is indeed fully linked, you should use the following command, which will notify you of any unresolved external references (e.g., if a CSUB is calling a function in an HP-UX library not specified in your link (`ld`) command).

```
nm -u CSUB_object_file
```

In the event that you do have undefined external references, you should resolve them by first determining the libraries in which the symbols are defined and then specifying those libraries in your link command. Again, it is extremely important that you only list those libraries after the library `librmb.a`.

---

## A Closer Look at Executing rmbuildc

After successfully generating one or more CSUB object files without unresolved external references, you are now ready to use the `rmbuildc` program to create the associated BASIC PROG file for the CSUBs.

The purpose of `rmbuildc` is to generate a BASIC subprogram interface for each of the CSUBs. This interface consists of the:

- name of the subprogram
- name and type of its parameters
- optional specification of any COM block used.

The program will interactively prompt you for all the necessary information about a CSUB to generate its BASIC subprogram interface.

---

### Note



On all the yes/no prompts in `rmbuildc`, a response other than  is treated as a negative response. A null response for a prompt is specified by only pressing .

---

## Procedure for Using rmbuildc

The following sections contain steps that explain how to use `rmbuildc`.

### Step 1: Executing rmbuildc

In HP-UX, execute:

```
rmbuildc 
```

The `rmbuildc` program begins by prompting you for the name of an optional stream file.

```
RMB-UX Compiled Subprogram File Generator (Version 1.1)
```

```
Stream file name:
```

In this file, the program will record all your responses to its prompts so that you can use the file as its standard input in subsequent executions (e.g., `rmbuildc < file_name`). Note that you may give a null response if you do not want a stream file.

If you already have Pascal Workstation stream files for use with the `BUILDC` utility, you may use them with `rmbuildc` by removing the responses to the prompts for the Header and Jump file names, and adding a response for the output BASIC PROG file name prompt. The other prompts remain the same.

## Step 2: Entering a PROG File Name

The next prompt is:

Output BASIC PROG file name:

This prompt asks for the BASIC PROG file name that you will specify in a `LOADSUB` statement to load the desired CSUBs into a BASIC program.

## Step 3: Naming CSUB Object Files

To the prompt:

CSUB object file name(s):

list the names of the CSUB object files generated by the linking procedure. Note that each object file will correspond to a specific version of BASIC. The file names specified should be separated by one or more spaces.

## Step 4: Specifying CSUB Interfaces

The remaining prompts deal with the specification of each CSUB interface. For each Pascal module that you specify, you will be asked for the name and parameters of its CSUBs. The following steps explain the prompts that you will encounter during this specification.

1. Enter the name of a module containing CSUBs when the following prompt is given.

Module name:

2. Enter the name of a CSUB that is in the current module when the following prompt is given.

CSUB name:

This prompt will be repeated until a null CSUB name is specified by pressing the `Return` key.

3. The following four prompts are repeated for each parameter of the current CSUB until a null parameter name is entered. You should specify the name of the parameter, its type, whether it is an array, and whether it is optional.

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

Since you are declaring the interface of the BASIC subprogram for the current CSUB, the names of the parameters that you specify need not match that of the CSUB. However, the types of the parameters between the Pascal CSUB and its BASIC subprogram declaration should match. For example, the BASIC CSUB call may be similar to the following:

```
Read_result(INTEGER Value_ret)
```

and the actual parameter name in the CSUB may be as follows:

```
PROCEDURE read_result(VAR return_int: bintvaltype);
```

The legal BASIC types are INTEGER, REAL, COMPLEX, string, and I/O path. If the parameter name ends in a dollar sign (\$), then the type is automatically assumed to be string, and if it begins with an @, the type is assumed to be an I/O path. Otherwise, you should answer with , , or . An array parameter is assumed to have dimension (\*), which indicates that it will be defined in the calling BASIC program.

Once a parameter is specified as optional, all the following parameters default to being optional and the prompt will not reappear.

Since `rmbuildc` cannot check the parameter list of a CSUB to verify that it matches its BASIC declaration, it is your responsibility to make sure that the matching is done correctly. Otherwise, the CSUB will behave unpredictably when called from a BASIC program.

4. Indicate whether the current CSUB accesses a BASIC COM by responding with either  or  to the following prompt.

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

It is not required that you declare a COM that is accessed in the CSUB. By declaring it, however, the COM will remain in memory as long as the CSUB is present, even after the BASIC subprogram that defines it is deleted. If you answer  to the above prompt, the following prompts will appear.

Respond to them with the information that they request. Note that the number of COM blocks that you specify will depend on the number of COMs that you want to access. With the exception of the first prompt given below, all of the prompts are repeated for each COM block. For an unlabeled COM, the label will be null.

```

Number of COM blocks:
COM label:
  Item name:
  Item type (I/R/C for Integer/Real/Complex):
  Is this an array? (y/n):

```

The questions concerning name, type, and array are the same as for the parameters, with the exception that the bounds of an array in a COM block have to be explicitly specified. To specify these bounds, enter the appropriate information to the prompts given below.

```

Number of dimensions or *:
Dimension n lower bound:
Dimension n upper bound:

```

If the number of dimensions is entered as “\*” the other dimension prompts are not displayed. This option may be used only if the array is defined either in the calling BASIC routine or a previous CSUB. If there is an explicit number of dimensions, the lower and upper bounds need to be entered for each of the dimensions.

If the type of the item is string, the following prompt will appear. In response to this prompt, enter the DIMensioned or maximum string length allowed for this COM item.

```
String length:
```

The final question about the COM item is:

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

You should answer **(N)** unless you plan to use the variable as a buffer in a TRANSFER statement. Read the “Advanced Transfer Techniques” chapter of the “BASIC Interfacing Techniques” manual for details. This last prompt completes the set of prompts related to specifying COM blocks.

The next prompt to appear will be a request for the name of another CSUB in the current module. If you have another CSUB to specify, you should enter its name at this time and repeat steps 3 and 4 of this section.

Otherwise, simply press the  key. This enters a null CSUB name and the following prompt appears:

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

If you respond with  to this prompt, `rmbuildc` will prompt you for another module name and you will need to repeat steps 2 through 4 of this section. Otherwise, press  and `rmbuildc` will proceed to construct the output BASIC PROG file. To facilitate the BASIC coding process, it will also print to a file the COM declarations required by all the CSUBs in BASIC source form. This file will reside in the current directory and will have the PROG file's name with `_COM` appended to it.

## **rmbuildc Errors**

The following table describes all the errors generated by `rmbuildc`.

Table 2-2. rmbuildc Errors

Error	Description
Opening of <i>file_name</i> failed.	An attempt at creating, writing to, or reading from the specified file caused an error.
Maximum number of input files exceeded.	Too many CSUB object files were specified. The maximum number of files currently allowed is 10.
File <i>file_name</i> has unresolved references.	The specified CSUB object file still contains undefined symbols, which should be resolved by linking the necessary additional libraries to the object file.
File <i>file_name</i> has no CSUB version stamp.	The specified CSUB object file was not properly linked with the library <b>librmb.a</b> . You should verify that you have a valid version of this library.
File <i>file_name</i> has an invalid a.out format.	The specified CSUB object file does not conform to the format specified for an object file.
Bound must be between -32768 and 32767.	The lower and upper bounds for a dimension of a COM block array item must be within the specified range.
High bound value is less than low bound value.	The high bound value of an array dimension must be greater than or equal to the low bound value of the same dimension.
Maximum number of elements in a dimension exceeded.	The maximum number of elements in the dimension of an array has been exceeded.
No item was specified.	A COM block without any items was specified.

Table 2-2. `rmbuildc` Errors (continued)

Error	Description
Type must be integer, real, or complex.	The type of this parameter or COM item should be integer, real, or complex.
Value must be 6 or less, or be an asterisk.	The number of dimensions for an array must be within the limits specified.
Value must be between 1 and 32767.	The value for the requested parameter must be within the specified range.
Procedure <i>CSUB_name</i> was not found.	The definition for the specified CSUB was not found in one of the CSUB object files.
No procedure name was specified.	At least one CSUB should be specified during each execution of <code>rmbuildc</code> for the output BASIC PROG file to be meaningful.
Invalid specification of COM item <i>item_name</i> .	The amount of memory that would be required to store the elements of the specified COM item exceeds the system limits.
Memory overflow.	The program is unable to allocate the necessary memory for processing unusually large CSUB input object files. Re-execute <code>rmbuildc</code> with a single command line parameter specifying the estimated number of bytes needed to handle the large object files; the default is 1000000 bytes.



---

## A Closer Look at Managing CSUBs from BASIC

This section shows how to manage CSUBs in BASIC. Topics covered are:

- Loading CSUBs into a BASIC Program
- Deleting CSUBs
- Handling CSUB Run-Time Errors

### Loading CSUBs into a BASIC Program

Before you can call a CSUB from a BASIC program, you have to load it in BASIC memory with a command similar to this:

```
LOADSUB ALL FROM "file_name"
```

where *file\_name* is the name of a BASIC PROG file generated by `rmbuildc`. This command loads and lists all subprograms and CSUBs in the CSUB libraries (a CSUB library is a set of CSUBs generated by a single execution of `rmbuildc`) found in the PROG file. Because the CSUBs are now listed in your BASIC program, you have access to all of them.

If you just want to list a particular CSUB in your BASIC program, you would use a command similar to this:

```
LOADSUB "sub_name" FROM "file_name"
```

where *sub\_name* is the CSUB that you want listed in your BASIC program, and *file\_name* is the BASIC PROG file where this CSUB is located. With this command, you will only have access to the specified CSUB and to those following it in its library. You should note, however, that just selecting one CSUB from a library to be listed does not save you memory because the entire library is always loaded into memory if one of its CSUBs is specified.

Once CSUBs are loaded into a BASIC program, they can be stored in a PROG file with the STORE command. Therefore, it is possible to have a PROG file consisting of several CSUBs generated at different times that were merged together using several LOADSUB and STORE commands. For example, consider a PROG file called *file\_name* with the following CSUB libraries:

```
(begin CSUB library A)
Csuba
Csubb
(end CSUB library A)
Subc
(begin CSUB library B)
Csubd
(end CSUB library B)
(begin CSUB library C)
Csube
Csubf
Csubg
(end CSUB library C)
Subh
Subi
```

If `Csubd` and `Csubf` are referenced in a program, executing `LOADSUB FROM "file_name"` will cause CSUB libraries B and C to be loaded in memory. Other CSUB libraries and subprograms are not brought in unless they are also referenced.

## Deleting CSUBs

All CSUBs belonging to the same CSUB library are listed contiguously and must remain in the order in which they were generated by `rmbuildc`. You cannot add BASIC program lines between CSUB declaration statements. However, you can delete CSUBs from a BASIC program by using the `DELSUB` command. Note that you can only delete the CSUB which comes first in a CSUB library. If you delete a CSUB not listed first in its library, BASIC will generate an error when you attempt to call any of the remaining CSUBs in the library.

### Example of Deleting CSUBs

In the following example, you may delete the CSUBs `Csube` and `Csubf`, in order, and you will still be able to call `Csubg`. However, you cannot delete `Csubb`, leaving `Csuba`, because BASIC will generate an error when you subsequently attempt to call `Csuba`.

```
(begin CSUB library A)
Csuba
Csubb
(end CSUB library A)
Subc
(begin CSUB library B)
Csubd
(end CSUB library B)
(begin CSUB library C)
Csube
Csubf
Csubg
(end CSUB library C)
Subh
Subi
```

## Handling CSUB Run-time Errors

Determining the cause of a CSUB run-time error is a difficult task because there is no BASIC debugger similar to the Pascal debugger which will let you step through your CSUB code. For this reason, you should thoroughly test your CSUBs in a test program before attempting to call them from BASIC. In situations when this approach is not practical, such as when using the BASIC file I/O routines, you may need to insert print statements in your CSUBs to monitor their execution in a BASIC program.

In Pascal, a CSUB run-time error may be generated by the Pascal system or by application code with an **ESCAPE** statement. You may handle the error condition by either trapping it within the CSUB or letting it propagate to the BASIC code. With either choice, there are precautions you should take to avoid corrupting the execution of BASIC.

### Trapping Errors

The **TRY/RECOVER** statement provides the mechanism to trap an error condition in Pascal. You may use it within a CSUB to process Pascal system errors or errors explicitly generated by your code with the **ESCAPE** statement.

You should refer to the "HP Pascal Language Reference" manual for a listing of the run-time errors that may be generated by the Pascal system.

## Reporting Errors to BASIC

After detecting a CSUB run-time error, you may decide to report the error to the BASIC calling code by simply ESCAPing out of a CSUB and causing a BASIC error condition. This condition may then be handled with an ON ERROR CALL/RECOVER statement. In this case, you need to be aware of the way BASIC will handle the error based on its ESCAPE code. The table below summarizes this information and will help you map Pascal ESCAPE codes to BASIC error numbers.

**Table 2-3. Error Numbers**

Pascal Errors	BASIC Definition
-32768 ... -891	SYSTEM ERROR
-890 ... -881	Reserved by operating system
-880 ... -32	SYSTEM ERROR
-33 ... -1	Pascal system error, ERROR (escapecode + 400)
0	Pascal exit, no error
1 ... 32767	User error, ERROR (escapecode MOD 1000)

In order to provide a consistent error recovery mechanism for CSUBs, BASIC has defined an error number for reporting all CSUB run-time errors. The basic idea is for a CSUB to exclusively use this error number for reporting all errors to BASIC; the parameterless routine `csub_error` in the module `kbdcr` should be used for this purpose.

The calling BASIC code can then recover from this error with an ON ERROR CALL/RECOVER statement and get the actual errors by calling an error CSUB or by reading some global error variables (e.g. COM block variables) also accessible to the reporting CSUB. This approach removes the need to map CSUB error numbers to BASIC error numbers, allows CSUB libraries from different sources to be shared within a program, and simplifies the task of recovering from a CSUB run-time error.

There may be situations when BASIC will not respond at all to user input or behave unpredictably during and after the execution of a CSUB due to

an unrecoverable error condition. In such situations, the current invocation of BASIC is likely to be corrupted and should be terminated with the **kill** command.

---

## Accessing System Resources

One of the motivations for using CSUBs with BASIC is the ability to access a rich set of HP-UX system libraries. This section covers the restrictions on the use of these libraries. The restrictions are due to the fact that BASIC also uses the libraries in its implementation.

- Allocating Dynamic Memory
- Simple Keyboard and Printer I/O
- Device I/O
- BASIC File I/O

### Allocating Dynamic Memory

When allocating memory from the heap, the only supported Pascal memory allocation statements allowed in CSUBs are **NEW** and **DISPOSE**. There are no special initialization procedures required to use these statements, but you are responsible for explicitly releasing any memory allocated within a CSUB. If you fail to do so, you may encounter an out-of-memory condition after repeated CSUB calls. In addition, you should use the compiler directive **\$HEAP\_DISPOSE ON\$** to activate the **DISPOSE** statement. The amount of heap memory available for CSUBs is determined by the:

- size of your BASIC process
- memory requirements of the calling BASIC program
- number of CSUBs loaded in memory.

You should note that the Pascal statements **MARK** and **RELEASE** are not supported in Pascal CSUBs. If you are porting Pascal Workstation CSUBs which make use of these statements, you should modify your code to:

- use the Pascal statements **NEW** and **DISPOSE**,

- remove the heap initialization call (`heap_init`) from the appropriate module,
- start taking advantage of the fact that the heap memory is guaranteed to remain at a fixed location during an execution of BASIC, since it is no longer allocated in a COM block.

## Simple Keyboard and Printer I/O

Operations on the standard I/O streams, writing to the screen and reading from the keyboard, for example, are not supported. Therefore, Pascal routines like `READLN` and `WRITELN` should only be used with an explicit file specifier. To allow a CSUB to input characters from the keyboard and to write to the PRINTER IS device, BASIC provides the module `kbdcr`t which implements procedures and functions for keyboard and CRT register access, character input and output, scrolling, and cursor manipulation. If you are porting Pascal Workstation CSUBs which rely on the routines `READ`, `READLN`, `WRITE`, and `WRITELN`, you should modify them to use this module.

Table 2-4. Keyboard and CRT I/O Routines

Procedure or Function	Description
<code>clear_screen</code>	clears the alpha CRT exactly as the <code>Clear display</code> key (or <code>CLEAR SCREEN</code> statement)
<code>controlcrt</code>	sends information to a CRT control register
<code>controlkbd</code>	sends information to a keyboard control register
<code>crtreadchar</code>	reads one character from the specified location on the CRT
<code>crtscroll</code>	scrolls the CRT area, from line <i>first</i> to line <i>last</i> , up or down one line
<code>cursor</code>	removes the previous cursor and writes a new cursor to any on-screen alpha location
<code>disp_at_xy</code>	allows text to be written to any alpha location on the CRT
<code>read_kbd</code>	returns the buffer contents trapped and held by <code>ON KBD</code> (same as <code>KBD\$</code> )
<code>scrolldn</code>	scrolls the <code>PRINT</code> area of the CRT down one line
<code>scrollup</code>	scrolls the <code>PRINT</code> area of the CRT up one line
<code>statuscrt</code>	returns the contents of a CRT status register
<code>statuskbd</code>	returns the contents of a keyboard status register
<code>systemd</code>	returns a string containing the results of calling the function <code>SYSTEM\$</code> for a given argument

## Device I/O

Device I/O in CSUBs is provided through the HP-UX device I/O library for HP-IB and GPIO interfaces, and through the standard HP-UX `termio` routines for the RS-232 interface. Pascal Workstation CSUBs which use the CSUB I/O library should be converted to call these new libraries. For more information, read the chapter “Device I/O Library (DIL)” in the *HP-UX Concepts and Tutorials: Device I/O and User Interfacing* manual.

## BASIC File I/O

BASIC provides the file access library module (`fal`) to allow operations on its file types. These file operations include:

- creating
- purging
- opening
- closing
- reading
- writing
- positioning.

Since CSUBs that use this module cannot be tested outside of a BASIC program, more time should be allocated for their implementation to minimize the number of debugging iterations.



**Table 2-5. File Access Routines**

Procedure or Function	Description
fal_create	creates an HP-UX file
fal_create_bdat	creates a BDAT file
fal_create_ascii	creates an ASCII file
fal_close	closes a file
fal_eof	writes an EOF at the current file position
fal_loadsub_all	loads all subprograms from the specified PROG file and appends them to the program in memory
fal_loadsub_name	loads the subprogram from the specified PROG file and appends it to the program in memory
fal_open	opens a file for reading and writing.
fal_position	positions the file pointer to a specified logical record number
fal_purge	purges a file
fal_read	reads data item(s) from a file
fal_read_bdat_int	reads a BASIC 16-bit integer from a BDAT file
fal_read_string	reads a string from an ASCII, BDAT, or HP-UX file
fal_write	writes data item(s) into a file
fal_write_bdat_int	writes a BASIC 16-bit integer to a BDAT file
fal_write_string	writes a string to an ASCII, BDAT, or HP-UX file



## Writing C and Assembly CSUBs

---

When you are developing a system that uses CSUBs, you should:

- write the BASIC program.
- determine what the CSUB should do, the parameters to be passed, and which variables should be accessible (*global*) to both the BASIC program and the CSUB.
- develop the CSUB; a listing of the BASIC program can be very helpful as reference during this task.

---

### Steps for Creating a C and Assembly CSUB

The following steps present an overview of the process needed to create a CSUB and the results of those steps. The CSUB related steps will be described in detail in later sections.

1. Enter BASIC, create and store the program that will call the CSUB. This program will contain CALLs to the CSUB, but the latter need not be implemented as it will be loaded later. You only need to decide what the subprogram will do and design the interface (parameters, COM, etc.) between BASIC and the CSUB.
2. Leave BASIC, enter HP-UX, and write the CSUB. You might, for example, use the `vi` editor to create the CSUB. See the sections “A Closer Look at C CSUB Components” and “A Closer Look at Parameter Passing” for details on how to organize your C functions and how to define your CSUB parameters.

3. Compile and debug the CSUB as much as possible by writing a C or Assembly test program. You may want to use the C debugger, `cdb`, or assembly debugger, `adb`, for this testing task.
4. Use the `ld` command to generate a fully linked relocatable object file containing all the CSUBs and any necessary HP-UX library support routines. For example,

```
ld -rd -a archive csub.o -u _printf -lrmb -o csub
```

would create a fully linked relocatable object file called `csub` using the compiled file `csub.o` and the library `librmb.a`. See the section “A Closer Look at Linking CSUB Object Files” for details.

5. Execute `rmbbuildc` and answer its prompts. This program generates the final BASIC PROG file. See the section “A Closer Look at Executing `rmbbuildc`” for details on how to answer the prompts.
6. Enter BASIC and load the BASIC PROG file from the keyboard or from the BASIC program using `LOADSUB`. This statement generates the necessary statements in your BASIC program to call CSUBs. See the section “A Closer Look at Managing CSUBs from BASIC” for details.
7. RUN the BASIC program which calls the desired CSUBs using the BASIC `CALL` or implied `CALL` statement.

## An Example: Passing Parameters

This section goes through each step of creating an example CSUB and how the CSUB is used in a BASIC program. All files used in the HP-UX environment are read from and written to the current directory. In the BASIC environment, the CSUB is loaded from the MASS STORAGE IS device (directory).

This simple program prompts you to enter your name (up to a maximum of 80 characters). It then passes your name in a string variable called `String$` to the CSUB called `parameters`. The CSUB changes the first character of you name to an asterisk (\*) and passes the name change back to the calling program along with the string length parameter called `int_val`. The CSUB also passes the real variable called `real_val` back to the calling program.

### Step 1: Create a BASIC Program that Calls the CSUB

Enter BASIC, edit and store this program in a file named `param_val`. This file can be found in the directory called `/usr/lib/rmb/demo`.

```
100 LOADSUB ALL FROM "Parm_vals"
110 DIM String$[80]
120 INTEGER Int_val
130 REAL Real_val
140 LINPUT "Enter your name and press [Return].",String$
150 Parameters(String$,Int_val,Real_val)
160 PRINT "Your name has been changed to: """;String$;""""
170 PRINT
180 PRINT "Your name contains ";
190 PRINT Int_val;
200 PRINT "characters."
210 PRINT
220 PRINT "Real_val = ";Real_val
230 DELSUB Parameters
240 END
```

### Steps 2 and 3: Write, Compile, and Debug the C CSUB

Enter HP-UX, edit, compile, and debug the following function called `parameters`, and save it in the file named `atest.c`. The file `atest.c` can be found in the directory called `/usr/lib/rmb/demo`. Note that, if you decide to change this C CSUB into an assembly CSUB, follow the steps provided in the next section.

```
#include <csubdecl.h>

typedef struct
{
shortint len;
char c[80];
} str_type;

parameters (str_dim, str_val, int_val, real_val)
dimentryptr str_dim;
str_type *str_val;
binteger_parm int_val;
breal_parm real_val;

{
```

```

str_val->c[0] = '*';
*int_val = str_val->len;
*real_val = 78.783;

}

```

3

### Steps 2 and 3: Write, Compile, and Debug the Assembly CSUB

Enter HP-UX, edit, compile, and debug the function called `parameters` that is provided in the previous section, and save it in a file named `atest.c`. The file `atest.c` can be found in the directory called `/usr/lib/rmb/demo`. To change the C source file to an assembly source file, type the following:

```
cc -S atest.c
```

The C compiler option `-S` creates an assembly code file called `atest.s` from the source C CSUB file called `atest.c`. Your assembly code file should look similar to the following:

```

        global  _parameters
_parameters:
        link.l  %a6,&LF1
        movm.l  &LS1, (%sp)
        mov.l   12(%a6), %a0
        movq    &42, %d0
        mov.b   %d0, 2(%a0)
        mov.l   12(%a6), %a0
        mov.l   16(%a6), %a1
        mov.w   (%a0), (%a1)
        mov.l   20(%a6), %a0
        mov.l   L12+0x4, 4(%a0)
        mov.l   L12, (%a0)
L11:
        unlk    %a6
        rts
        set     LF1, -0
        set     LS1, 0
        lalign  4
L12:
        long    0x4053b21c, 0xac083127
        data

```

Once the assembly source file has been created, you are ready to create an `.o` file out of it by typing:

### 3-4 Writing C and Assembly CSUBs

```
as atest.s
```

When you execute the HP-UX command `ls`, you will notice the file called `atest.o` is in your current working directory.

#### Step 4: Generate a CSUB Object File

Link the code file `atest.o` with the BASIC CSUB library `librmb.a` to generate a fully linked relocatable CSUB object file. The HP-UX `ld` command should be used for this purpose, as follows:

```
ld -rd -a archive atest.o -u _printf -lrmb -o atest
```

#### Step 5: Generate a BASIC PROG File (`rmbuildc`)

Execute the `rmbuildc` program and answer the prompts as shown below. Notice that a stream file is generated by the response to the first prompt; you can use this file the next time you execute the program (i.e. `rmbuildc < stream`) to remove the need to interactively answer the prompts again.

RMB-UX Compiled Subprogram File Generator (Version 1.1)

```
Stream file name: stream
Output BASIC PROG file: Parm_vals
CSUB object file names(s): atest
Module name: Return
CSUB name: parameters
  Parameter name: string$
    Parameter type is string
    Is this an array? (y/n): n
    Is this an optional parameter? (y/n): n
  Parameter name: int_val
    Parameter type (I/R/C for Integer/Real/Complex): i
    Is this an array? (y/n): n
    Is this an optional parameter? (y/n): n
  Parameter name: real_val
    Parameter type (I/R/C for Integer/Real/Complex): r
    Is this an array? (y/n): n
    Is this an optional parameter? (y/n): n
  Parameter name: Return
  Is there COM in this CSUB? (y/n): n

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

## Steps 6 and 7: LOAD and RUN the CSUB

In this example, `Parm_vals` is automatically loaded from the BASIC program. Therefore, you only need to re-enter the BASIC system, GET "param\_val", and RUN the program. The output should be:

```
Your name has been changed to: "*ohn J. Doe"
```

```
Your name contains 11 characters.
```

```
Real_val = 78.783
```

---

## A Closer Look at C CSUB Components

Any C function you implement may easily be transformed into a CSUB if you follow these guidelines:

- The C function must be defined as global; it therefore should not have the **static** attribute.
- The C function should not have a return value since there is no mechanism for getting this value from BASIC. Instead, it should use a reference parameter to store this return value.
- The C function's parameters should match the actual parameters that are passed when it is called from a BASIC program. For information on parameter type matching, see the table called "Equivalent C and BASIC Parameter Types."

The include file `csubdecl.h` was defined to facilitate this parameter matching between a BASIC program and C CSUBs. It provides all the necessary definitions to allow you to specify types for the formal parameters of C CSUBs. These types are limited to those that are supported for CSUBs and are described in detail in the next section.



---

## A Closer Look at Parameter Passing

In order to be useful, a CSUB needs the ability to exchange data with the calling BASIC program. This section describes the different way of performing this data exchange. The primary method consists of defining the CSUB parameters to match those passed by BASIC. This method requires special attention to the different parameter types and formats of BASIC variables. The second method for a CSUB to exchange data with a BASIC program is via COM blocks. Again, this method necessitates the special handling of the variables defined in the blocks, according to their type and format.

### Passing Parameters by Reference

As far as C is concerned, BASIC variables are always passed to a CSUB *by reference*. 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. Therefore, in order to receive a referenced BASIC parameter, a C CSUB needs to match it with a formal parameter declared as a pointer to the BASIC parameter.

The following example shows how the variable `realvar` is passed as a parameter by reference to a function using the address of the variable.

```
p(r)
double *r;
{
r=-31178.0;
}

main()
{
double realvar;

p(&realvar); /* Note the & symbol */
}
```

The two key points to remember are:

- BASIC always passes a pointer to a variable.
- BASIC has no idea if a user-written CSUB has been properly coded to use that pointer.

Note that errors will occur if this distinction is overlooked.

## Parameter Types

3 The BASIC parameter types are not necessarily the same as their C counterparts. It is important that the parameter types of a CSUB be correct so that BASIC and the CSUB can interface properly. This section will explain the types in detail. You should refer to the include file `csubdecl.h` for the definition of the types used below.

The following table provides you with a quick reference to equivalent C and BASIC parameter types.

Table 3-1. Equivalent C and BASIC Parameter Types

BASIC Parameter Type	Assembly Parameter Size	C Parameter Type
REAL	8 bytes	#include <csubdecl.h> breal_parm
INTEGER	2 bytes	#include <csubdecl.h> binteger_parm
COMPLEX	16 bytes	#include <csubdecl.h> bcomplex_parm
<i>string_name\$</i>	4 bytes for both parameters	#include <csubdecl.h> Two parameter types passed for strings: • dumentryptr • bstring_parm
<i>ary_nm[lower : upper, etc.]</i> of one of the above numeric parameter types or <i>str_ary\$(low : up, etc.)[n_chars]</i>	4 bytes for both parameters	#include <csubdecl.h> Two parameter types passed for arrays: • dumentryptr • one of the above numeric or string array types.
@ <i>io_path_name</i>	4 bytes	#include <csubdecl.h> fcb_ptr_type

## REAL

A variable defined as a `double` in C maps into a BASIC REAL. Therefore, a BASIC REAL parameter can be defined in a CSUB as:

```
#include <csubdecl.h>
x(y)
breal_parm y;
```

since `breal_parm` is defined as a pointer to `double`.

## COMPLEX

A variable defined in BASIC as COMPLEX is a floating point value with real and imaginary parts. There is no built-in COMPLEX type in C.

A C declaration for a COMPLEX value would be:

```
struct bcplxvaltype
{
    double re;
    double im;
}
```

Thus, you could use:

```
#include <csubdecl.h>
x(y)
bcomplex_parm y;
```

because `bcomplex_parm` is defined as a pointer to `bcplxvaltype`.

## INTEGER

A variable defined as an `int` in C is not the same as a BASIC INTEGER. The latter is a 16-bit quantity while a C integer is a 32-bit quantity. Therefore, to receive a BASIC INTEGER, you should use:

```
#include <csubdecl.h>
x(y)
binteger_parm y;
```

because `binteger_parm` is defined as a pointer to `bintvaltype`.

## Strings

Strings are different in BASIC and C, both in their structure and the way they are passed. The structure of a C string is a set of characters terminated by a NULL character while the BASIC string has a two-byte length field followed by the characters of the string.

BASIC passes its strings as two parameters:

- The first parameter is a pointer to a **dimension structure**. This structure contains information about arrays, strings, their maximum lengths, and their lower and upper bounds. It is expressed in C as a *union* of type `dimentry` while its pointer type is `dimentryptr`. For the case of a scalar (non-array) string, the only field in the union is a short integer, a 16-bit quantity, expressing the maximum length of the string.
- The second parameter is a pointer to the string **value area**. This area contains the actual length of the string and its characters. The type of this pointer is `bstring_parm`.

An example of how this would look in a C function is as follows:

```
#include <csbdecl.h>

static char *s="a string";

getstring(dim_len, b)
dimentryptr dim_len;
bstring_parm b;
{
short i;

if (strlen(s)>dim_len->maxlen) s[dim_len->maxlen]=0;
b->len=strlen(s);
for (i=0; i<b->len; i++) b->c[i]=s[i];
}
```

The above function copies a C string into a BASIC string. From this example, you should note how:

- the string parameter is declared,
- an explicit check has been made to insure that the length of the C string is not greater than the maximum length of the BASIC string,

- the C string value is put into the BASIC string value area.

Although a CSUB receives two parameters for a BASIC string, as far as BASIC is concerned, there is only one actual parameter to be passed, as shown below.

```
...
10 DIM Str$[80]
20 CALL GETSTRING(Str$)
30 PRINT Str$
40 END
```

### I/O Paths

An I/O path is a block of storage used to keep track of the state of a file or I/O device. The size of this block is 190 bytes. See the section “BASIC File I/O” for details on how you would use an I/O path as a file control block with the file access library (**fal**) routines.

The following example shows how a C CSUB receives an I/O path parameter from BASIC. The include file `csubdecl.h` contains the necessary type declarations to pass I/O path parameters to C. The same parameters can then be used with the **fal** routines.

```
#include <csubdecl.h>

x(y_ptr)
fcb_ptr_type y_ptr;
```

The typical use of the parameter `y_ptr` in a CSUB would then take the form:

```
csfa_fal_open(filename, y_ptr);
```

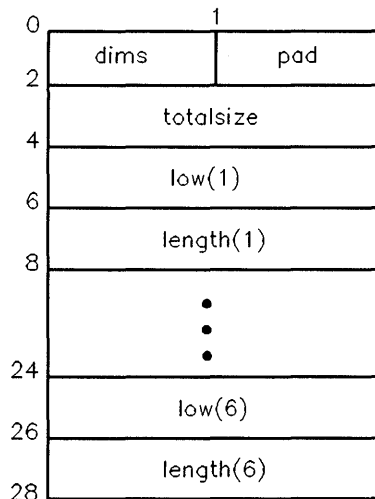
## Arrays

Arrays are passed as two parameters:

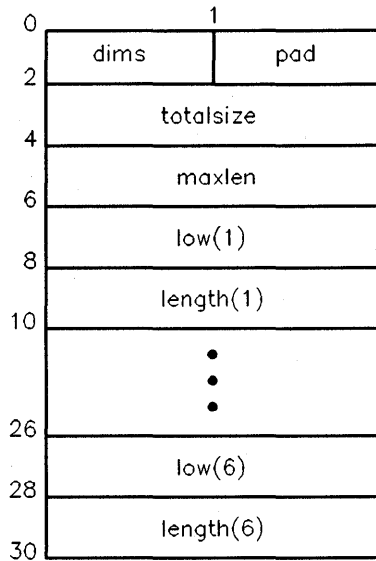
- a pointer to the dimension structure. The fields in the dimension structure, in this case, are more complicated. They are also defined by the `dimentry` type.
- a pointer to the value area.

The actual dimension structures for the `REAL`, `INTEGER`, `COMPLEX`, and string arrays are represented by Figures 3-1 and 3-2.

<code>dims</code>	is a byte containing the number of dimensions
<code>totalsize</code>	is the number of bytes in the entire array
<code>low(n)</code>	is the lower bound of the <i>n</i> th dimension
<code>length(n)</code>	is the number of elements in the <i>n</i> th dimension
<code>maxlen</code>	is the maximum length of any element in a string array.



**Figure 3-1.**  
**REAL, INTEGER, or COMPLEX Array Dimension Structure**



**Figure 3-2. String Array Dimension Structure**

An example of receiving an INTEGER array from BASIC is as follows:

```
x(d, arr)
dimentryptr d;
bintvaltype arr[];
```

Again, this is a single parameter in BASIC. So, in this example, BASIC would send an array defined as:

```
INTEGER Arr(1:10)
CALL X(Arr(*))
```



## Defining BASIC and C Arrays

The BASIC and C arrays should be defined the same way. This is not mandatory, but helpful. You should remember that an array with bounds [1..5] is the same as an array with bounds [6..10]; both are five-element arrays. If a BASIC array defined as `INTEGER Arr(6:10)` is passed to a CSUB array parameter defined as `bintvaltype arr[]`, the sixth element in the BASIC array will correspond to the first element in the C CSUB array. Array elements are stored in row-major order in both BASIC and C.

## Dimensioning an Array

The DIM statement should be used in conjunction with the appropriate C array declaration to define the space for the BASIC array. The REDIM statement does not affect the size of that space. It does however affect the BASE and SIZE functions and the `length` and `low` values in the dimension structure. Note that a CSUB should therefore check the dimension structure of an array parameter to find its current dimensions.

## Redimensioning an Array

If you use the REDIM statement on a multi-dimensional BASIC array, the C declaration of the array will be invalid the next time the array is accessed in a CSUB. Therefore, if you accessed `arr[2][3]` in C, you would not get the same value as `Arr(2,3)` from BASIC. To be immune from the effects of REDIM, you should declare a multi-dimensional BASIC array as a one-dimensional array in C and do explicit subscript calculations based on the information in the dimension structure of the array.

## Declaring the Value Area of a String

You should also note that the type `bstringvaltype` should not be used in specifying the value area for a string array since it defines a string with the maximum allowable number of characters. Instead, you should declare a different type, with the maximum allowable number of characters in the array strings set equal to the dimension of the equivalent BASIC string array. Thus, for the following BASIC definition of a string array,

```
DIM S$(1:10)[20]
```

you would define the type for a C string in the array as:

```

typedef struct
{
    shortint len;
    char c[20]; /* Same maximum length as BASIC */
} str_type;

```

3

## Useful type Declarations

The include file `csubdecl.h` defines useful types which may be used in the declaration of C CSUB parameters. These declarations are listed below.

```

#define STRINGLIMIT 32767 /* maximum length of a string */
#define MAXDIM 6 /* maximum dimensions in an array */
#define MAXARRAYSIZE 16777215 /* maximum bytes in an array */

typedef unsigned char byte;
typedef short shortint; /* two byte integer */
typedef short bintvaltype; /* BASIC integer */
typedef double brealvaltype; /* BASIC real */

struct bcmplxvaltype /* BASIC complex type */
{
    brealvaltype re;
    brealvaltype im;
};

struct bstringvaltype /* BASIC string type */
{
    shortint len;
    char c[STRINGLIMIT];
};

struct boundentry /* describes array bound */
{
    shortint low; /* lower limit */
    shortint length; /* number of elements */
};

union dumentry /* dimension record union */
{
    shortint maxlen; /* string scalar */
    struct /* array */
    {
        byte dims; /* number of dimensions */
    };
};

```

```

byte pad;
short totalsize;          /* total size of an array */
union
{
    {                      /* numeric array */
    struct boundentry bound[MAXDIM]; /* dimension boundaries */
    struct                /* string array */
    {
        shortint maxlen;          /* maximum string length */
        struct boundentry bound[MAXDIM]; /* dimension boundaries */
    } strval;
    } arrval;
} arrdim;
};

typedef union dimentry *dimentryptr;          /* pointer to dimension union
*/
typedef bintvaltype *binteger_parm;          /* pointer to BASIC integer */
typedef brealvaltype *breal_parm;           /* pointer to BASIC real */
typedef struct bcomplexvaltype *bcomplex_parm; /* pointer to BASIC complex */
typedef struct bstringvaltype *bstring_parm; /* pointer to BASIC string */

typedef char *fcb_ptr_type;

```

## Optional Parameters

You can declare some or all parameters of a CSUB as *optional* through responses to `rmbbuildc`. Optional parameters are those which are not required in the parameter list of the calling code. In a C CSUB, however, there is no distinction between required and optional parameters as both types of parameters have to be listed in the declaration of the CSUB. See “Subprograms” in the “BASIC Programming Techniques” manual for more information about optional parameters.

BASIC will pass a NIL pointer to the CSUB when one of its parameters that has been declared as optional is omitted. The CSUB should therefore always make an explicit NIL check before trying to use the value of an optional parameter. Otherwise, a run-time error may occur when attempting to dereference the pointer.

For example, if the BASIC declaration of a CSUB is:

```
100 CSUB My_csub(REAL R, OPTIONAL REAL Opt)
```

the variable `Opt` will have an address of `NIL` if it is not passed. In C, the CSUB should perform the following test:

```
my_csub(required, optional)
breal_parm required, optional;
{
  if (optional)... /* It was passed in */
```

3

## Accessing BASIC COM from a CSUB

Another way for a BASIC program and a C CSUB to interchange data is via BASIC COM blocks. In order to access a BASIC COM block from a C CSUB, you should use the function `find_com`. When passed the name of the COM block that is to be accessed, this function will return a pointer to the beginning of the value area of that block. Since upper and lower case letters are significant in a COM block name, they should be specified the same way BASIC does. To access an unlabeled COM block, you should specify a string with a single blank as the COM block name. When `find_com` cannot find the COM block requested, it returns a `NIL` pointer.

## Defining a C Structure

In order to read and store values in the variables of a COM block, you will need to define a C structure to map the variables into members of the structure. This will require you to know the layout of the block in advance since there is no way of determining this layout from the C CSUB at run-time.

In defining the C structure for accessing a COM block, you should know that:

- The order of the members in the structure should be opposite from that of the variables in the block. For example, the first variable in the COM block should be mapped into the last member of the structure.
- There is nothing to prevent you from inadvertently corrupting the COM block by writing beyond its boundaries or by storing invalid values.
- The value areas of BASIC strings and arrays are stored in a COM block. Therefore, you should omit the specification of the dimension structure for those variables in the C structure.
- You should follow the same restriction previously mentioned for CSUB string array parameters in specifying the bounds of string arrays in COM blocks.

This is an example of accessing a BASIC COM block defined as:

```
COM /Num1/ INTEGER A,B(1:5),REAL C,D$(10)
```

To access this block from a CSUB, you could use:

```
#include <csubdecl.h>

typedef struct
{
    shortint len;
    char c[10];
} strtype;

typedef struct
{
    strtype d;           /* Do not use bstringvaltype! */
    brealvaltype c;
    bintvaltype b[5];
    bintvaltype a;     /* Note the reverse order of the members */
} *comtype;
extern comtype find_com();

x()
{
    comtype comptr;
    bintvaltype vala;

    comptr=find_com("Num1"); /* Observe same case of letters as BASIC does */
    vala=comptr->a;
    ...
}
```

3

A COM block is subject to being moved in memory at RUN, LOAD, and GET time. Therefore, to guarantee that you always get the current location of a block, you should not remember its address in a CSUB but always use the function `find_com` instead. Similarly, you should not attempt to remember the address of a variable defined in a COM block.

---

## A Closer Look at Linking CSUB Object Files

After thoroughly testing your CSUBs in a C program, you are now ready to generate a CSUB object file. This object file brings together the CSUB binaries and the necessary libraries to satisfy the external references of those binaries.

### Using the Linker

The HP-UX linker can be used to generate a fully linked, relocatable CSUB object file suitable for input to the `rmbbuildc` program. The syntax for linking is as follows:

```
ld -rd -a archive csub_binaries -u _printf -lrm [other_libraries] -o csub_obj_file
```

where:

- `-rd` are the options to retain relocation information in the output file for subsequent re-linking and to force the definition of “common” storage.
- `-a archive` tells the loader to use archive libraries instead of shared libraries.
- `csub_binaries` are the `.o` files generated by the C compiler. They contain the code for CSUBs and should always be followed in the above command line by the library `-lrm`.
- `-u _printf` links needed object files.
- `other_libraries` are libraries that are required to resolve all the references not satisfied by `librm.a`; they may be your own or HP-UX libraries.
- `-o csub_obj_file` takes the output of the `ld` command and sends it to the file. `csub_obj_file`.

## Specifying the `librmb.a` Library

In addition to providing useful system access routines for CSUBs, this library also specifies all the external symbols from HP-UX libraries which were used in the actual implementation of a specific version of BASIC. Specifying the library immediately after your CSUB binaries in the `ld` command will thus ensure the sharing of those symbols between BASIC and your CSUBs. This sharing first removes the need for duplicating the code already available in BASIC in the CSUB object files. More importantly, it is also necessary for the proper operation of libraries, the routines of which maintain global variables which should not be duplicated in the CSUB object files. An example of such routines are the HP-UX memory management routines (`malloc 3C` includes: `calloc`, `free`, `mallopt`, `mallinfo`, and `realloc`).

## Resolving External References

In general, specifying the library `librmb.a` alone in your link command will be sufficient to resolve all external references from your CSUB binaries; this is because BASIC uses most of the common HP-UX libraries in its implementation.

Nevertheless, to verify that your CSUB object file is indeed fully linked, you should use the following command, which will notify you of any unresolved external references (e.g., if a CSUB is calling a function in an HP-UX library not specified in your link (`ld`) command).

```
nm -u CSUB_object_file
```

In the event that you do have undefined external references, you should resolve them by first determining the libraries in which the symbols are defined and then specifying those libraries in your link command. Again, it is extremely important that you only list those libraries after the library `librmb.a`.

---

## A Closer Look at Executing `rmbuildc`

After successfully generating one or more CSUB object files without unresolved external references, you are now ready to use the `rmbuildc` program to create the associated BASIC PROG file for the CSUBs.

3

The purpose of `rmbuildc` is to generate a BASIC subprogram **interface** for each of the CSUBs. This interface consists of the:

- name of the subprogram
- name and type of its parameters
- optional specification of any COM block used.

The program will interactively prompt you for all the necessary information about a CSUB to generate its BASIC subprogram interface.

---

### Note



On all the yes/no prompts in `rmbuildc`, a response other than **Y** is treated as a negative response. A null response for a prompt is specified by only pressing **Return**.

---

## Procedure for Using `rmbuildc`

The following sections contain steps that explain how to use `rmbuildc`.

### Step 1: Executing `rmbuildc`

In HP-UX, execute:

```
rmbuildc Return
```

The `rmbuildc` program begins by prompting you for the name of an optional stream file.

```
RMB-UX Compiled Subprogram File Generator (Version 1.1)
```

```
Stream file name:
```

In this file, the program will record all your responses to its prompts so that you can use the file as its standard input in subsequent executions (e.g., `rmbuildc < file_name`). Note that you may give a null response if you do not want a stream file.



## Step 2: Entering a PROG File Name

The next prompt is:

Output BASIC PROG file name:

This prompt asks for the BASIC PROG file name that you will specify in a LOADSUB statement to load the desired CSUBs into a BASIC program.

## Step 3: Naming CSUB Object Files

To the prompt:

CSUB object file name(s):

list the names of the CSUB object files generated by the linking procedure. Note that each object file will correspond to a specific version of BASIC. The file names specified should be separated by one or more spaces.

## Step 4: Specifying CSUB Interfaces

The remaining prompts deal with the specification of each CSUB interface. The following steps explain the prompts that you will encounter during this specification.

1. Press the **Return** key in response to the prompt below. This prompt is only relevant for Pascal language CSUBs and should always be handled as specified.

Module name:

2. Enter the name of a CSUB when the following prompt is given.

CSUB name:

This prompt will be repeated until a null CSUB name is specified by pressing the **Return** key.

3. The following four prompts are repeated for each parameter of the current CSUB until a null parameter name is entered. You should specify the name of the parameter, its type, whether it is an array, and whether it is optional.

Parameter name:

Parameter type (I/R/C for Integer/Real/Complex):

Is this an array? (y/n):

Is this an optional parameter? (y/n):

Since you are declaring the interface of the BASIC subprogram for the current CSUB, the names of the parameters that you specify need not match that of the CSUB. However, the types of the parameters between the C CSUB and its BASIC subprogram declaration should match. For example, the BASIC CSUB call may be similar to the following:

```
Read_result(INTEGER Value_ret)
```

and the actual parameter name in the CSUB may be as follows:

```
read_result(return_int)
binteger_parm return_int;
```

The legal BASIC types are INTEGER, REAL, COMPLEX, string, and I/O path. If the parameter name ends in a dollar sign (\$), then the type is automatically assumed to be string, and if it begins with an @, the type is assumed to be an I/O path. Otherwise, you should answer with **I**, **R**, or **C**. An array parameter is assumed to have dimension (\*), which indicates that it will be defined in the calling BASIC program.

Once a parameter is specified as optional, all the following parameters default to being optional and the prompt will not reappear.

Since `rmbuildc` cannot check the parameter list of a CSUB to verify that it matches its BASIC declaration, it is your responsibility to make sure that the matching is done correctly. Otherwise, the CSUB will behave unpredictably when called from a BASIC program.

4. Indicate whether the current CSUB accesses a BASIC COM by responding with either **Y** or **N** to the following prompt.

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

It is not required that you declare a COM that is accessed in the CSUB. By declaring it, however, the COM will remain in memory as long as the CSUB is present, even after the BASIC subprogram that defines it is deleted. If you answer **Y** to the above prompt, the following prompts will appear. Respond to them with the information that they request. Note that the number of COM blocks that you specify will depend on the number of COMs that you want to access. With the exception of the first prompt given below, all of the prompts are repeated for each COM block. For an unlabeled COM, the label will be null.

Number of COM blocks:

COM label:

Item name:

Item type (I/R/C for Integer/Real/Complex):

Is this an array? (y/n):

The questions concerning name, type, and array are the same as for the parameters, with the exception that the bounds of an array in a COM block have to be explicitly specified. To specify these bounds, enter the appropriate information to the prompts given below.

Number of dimensions or \*:

Dimension n lower bound:

Dimension n upper bound:

If the number of dimensions is entered as “\*” the other dimension prompts are not displayed. This option may be used only if the array is defined either in the calling BASIC routine or a previous CSUB. If there is an explicit number of dimensions, the lower and upper bounds need to be entered for each of the dimensions.

If the type of the item is string, the following prompt will appear. In response to this prompt, enter the DIMensioned or maximum string length allowed for this COM item.

String length:

The final question about the COM item is:

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

You should answer  unless you plan to use the variable as a buffer in a TRANSFER statement. Read the “Advanced Transfer Techniques” chapter of the “BASIC Interfacing Techniques” manual for details. This last prompt completes the set of prompts related to specifying COM blocks.

The next prompt to appear will be a request for the name of another CSUB. If you wish to specify another CSUB, you should enter its name at this time and repeat steps 3 and 4 of this section. Otherwise, simply press the  key. This enters a null CSUB name and the following prompt appears:

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

If you respond with  to this prompt, rmbuildc will prompt you for another module name and you will need to repeat steps 2 through 4 of this

section. Otherwise, press **(N)** and `rmbuildc` will proceed to construct the output BASIC PROG file. To facilitate the BASIC coding process, it will also print to a file the COM declarations required by all the CSUBs in BASIC source form. This file will reside in the current directory and will have the PROG file's name with `_COM` appended to it.

3

### **rmbuildc Errors**

The following table describes all the errors generated by `rmbuildc`.

**Table 3-2. rmbuildc Errors**

Error	Description
Opening of <i>file_name</i> failed.	An attempt at creating, writing to, or reading from the specified file caused an error.
Maximum number of input files exceeded.	Too many CSUB object files were specified. The maximum number of files currently allowed is 10.
File <i>file_name</i> has unresolved references.	The specified CSUB object file still contains undefined symbols, which should be resolved by linking the necessary additional libraries to the object file.
File <i>file_name</i> has no CSUB version stamp.	The specified CSUB object file was not properly linked with the library <b>librmb.a</b> . You should verify that you have a valid version of this library.
File <i>file_name</i> has an invalid a.out format.	The specified CSUB object file does not conform to the format specified for an object file.
Bound must be between -32768 and 32767.	The lower and upper bounds for a dimension of a COM block array item must be within the specified range.
High bound value is less than low bound value.	The high bound value of an array dimension must be greater than or equal to the low bound value of the same dimension.
Maximum number of elements in a dimension exceeded.	The maximum number of elements in the dimension of an array has been exceeded.
No item was specified.	A COM block without any items was specified.

**Table 3-2. rmbuildc Errors (continued)**

Error	Description
Type must be integer, real, or complex.	The type of this parameter or COM item should be integer, real, or complex.
Value must be 6 or less, or be an asterisk.	The number of dimensions for an array must be within the limits specified.
Value must be between 1 and 32767.	The value for the requested parameter must be within the specified range.
Procedure <i>CSUB_name</i> was not found.	The definition for the specified CSUB was not found in one of the CSUB object files.
No procedure name was specified.	At least one CSUB should be specified during each execution of <b>rmbuildc</b> for the output BASIC PROG file to be meaningful.
Invalid specification of COM item <i>item_name</i> .	The amount of memory that would be required to store the elements of the specified COM item exceeds the system limits.
Memory overflow.	The program is unable to allocate the necessary memory for processing unusually large CSUB input object files. Re-execute <b>rmbuildc</b> with a single command line parameter specifying the estimated number of bytes needed to handle the large object files; the default is 1000000 bytes.

3

---

## A Closer Look at Managing CSUBs from BASIC

This section shows to manage CSUBs in BASIC. Topics covered are:

- Loading CSUBs into a BASIC Program
- Deleting CSUBs
- Handling CSUB Run-Time Errors

### Loading CSUBs into a BASIC Program

Before you can call a CSUB from a BASIC program, you have to load it in BASIC memory with a command similar to this:

```
LOADSUB ALL FROM "file_name"
```

where *file\_name* is the name of a BASIC PROG file generated by `rmbuildc`. This command loads and lists all subprograms and CSUBs in the CSUB libraries (a CSUB library is a set of CSUBs generated by a single execution of `rmbuildc`) found in the PROG file. Because the CSUBs are now listed in your BASIC program, you have access to all of them.

If you just want to list a particular CSUB in your BASIC program, you would use a command similar to this:

```
LOADSUB "sub_name" FROM "file_name"
```

where *sub\_name* is the CSUB that you want listed in your BASIC program, and *file\_name* is the BASIC PROG file where this CSUB is located. With this command, you will only have access to the specified CSUB and to those following it in its library. You should note, however, that just selecting one CSUB from a library does not save you memory because the entire library is always loaded into memory if one of its CSUBS is specified.

Once CSUBs are loaded into a BASIC program, they can be stored in a PROG file with the STORE command. Therefore, it is possible to have a PROG file consisting of several CSUBs generated at different times that were merged together using several LOADSUB and STORE commands. For example, consider a PROG file called *file\_name* with the following CSUB libraries:

3

```
(begin CSUB library A)
Csuba
Csubb
(end CSUB library A)
Subc
(begin CSUB library B)
Csubd
(end CSUB library B)
(begin CSUB library C)
Csube
Csubf
Csubg
(end CSUB library C)
Subh
Subi
```

If `Csubd` and `Csubf` are referenced in a program, executing `LOADSUB FROM "file_name"` will cause CSUB libraries B and C to be loaded in memory. Other CSUB libraries and subprograms are not brought in unless they are also referenced.

## Deleting CSUBs

All CSUBs belonging to the same CSUB library are listed contiguously and must remain in the order in which they were generated by `rmbbuildc`. You cannot add BASIC program lines between CSUB declaration statements. However, you can delete CSUBs from a BASIC program by using the `DELSUB` command. Note that you can only delete the CSUB which comes first in a CSUB library. If you delete a CSUB not listed first in its library, BASIC will generate an error when you attempt to call any of the remaining CSUBs in the library.

### Example of Deleting CSUBs

In the following example, you may delete the CSUBs `Csube` and `Csubf` in order and you will still be able to call `Csubg`. However, you cannot delete `Csubb`, leaving `Csuba`, because BASIC will generate an error when you subsequently attempt to call `Csuba`.



```
(begin CSUB library A)
Csuba
Csubb
(end CSUB library A)
Subc
(begin CSUB library B)
Csubd
(end CSUB library B)
(begin CSUB library C)
Csube
Csubf
Csubg
(end CSUB library C)
Subh
Subi
```

## Handling CSUB Run-time Errors

Determining the cause of a CSUB run-time error is a difficult task because there is no BASIC debugger similar to the C debugger which will let you step through your CSUB code. For this reason, you should thoroughly test your CSUBs in a test program before attempting to call them from BASIC. In situations where this approach is not practical, such as when using the BASIC file I/O routines, you may need to insert print statements in your CSUBs to monitor their execution in a BASIC program.

In C, a CSUB run-time error is generated with a signal condition. You may either trap this condition with your own signal handler or let the signal propagate to the BASIC code. With either choice, there are precautions you should take to avoid corrupting the execution of BASIC.

### Trapping Errors

In order to trap a signal condition within a CSUB, you will need to install your own signal handler for the selected signal in the code of a CSUB. You should then make sure that the same code will restore the original handler for that signal when the CSUB terminates; otherwise, BASIC may behave unpredictably following the call to the CSUB. In order to assign a handler for a specific signal, you should use the system routine `sigvector(2)` exclusively.

## Reporting Errors to BASIC

3 If you decide to let a signal propagate to the BASIC code, you should verify that it is a valid signal, as defined by the ON EXT SIGNAL statement. You should always trap all other signals with a signal handler and handle them within your CSUB; BASIC may not handle invalid signals in a consistent fashion and may behave unpredictably when it receives them. On the other hand, the disposition of valid signals by BASIC is determined by whether a system signal event-initiated branch is in effect at the time the CSUB is called and is exactly specified by the ON EXT SIGNAL statement.

In order to provide a consistent error recovery mechanism for CSUBs, BASIC has defined an error number for reporting all CSUB run-time errors. The basic idea is for a CSUB to exclusively use this error number for reporting all signal and error conditions to BASIC; the parameterless routine `csub_error` should be used for this purpose.

The calling BASIC code can then recover from this error with an ON ERROR CALL/RECOVER statement and get the actual signals or errors by calling an error CSUB or by reading some global variables (e.g. COM block variables) also accessible to the reporting CSUB. This approach removes the need to map CSUB error numbers to BASIC error numbers, allows CSUB libraries from different sources to be shared within a program, and simplifies the task of recovering from a CSUB run-time error.

In situations where BASIC will not respond at all to user input or behaves unpredictably during and after the execution of a CSUB due to an unrecoverable error condition, the current invocation of BASIC is likely to be corrupted and should be terminated with the `kill` command.

---

## Accessing System Resources

One of the motivations for using CSUBs with BASIC is the ability to access a rich set of HP-UX system libraries. This section covers the restrictions on the use of these libraries. The restrictions are due to the fact that BASIC also uses the libraries in its implementation.

- Allocating Dynamic Memory
- Simple Keyboard and Printer I/O
- Device I/O
- BASIC File I/O

### Allocating Dynamic Memory

When allocating memory from the heap, you may use the standard memory allocation package `malloc` (3C or 3X). Although there is no initialization procedure required, you are still responsible for explicitly releasing any memory allocated within a CSUB. If you fail to do so, you may encounter an out-of-memory condition after repeated CSUB calls. The amount of heap memory available for CSUBs is determined by the:

- size of your BASIC process
- memory requirements of the calling BASIC program
- number of CSUBs loaded in memory.

### Simple Keyboard and Printer I/O

Operations on the standard I/O streams, writing to the screen and reading from the keyboard are not supported. Therefore, C routines like `printf` and `scanf` should not be used. To allow a CSUB to input characters from the keyboard and to write to the PRINTER IS device, BASIC provides a set of routines in the library `librmb.a` for keyboard and CRT register access, character input and output, scrolling, and cursor manipulation.

**Table 3-3. Keyboard and CRT I/O Routines**

Routine	Description
kdbcrt_clear_screen	clears the alpha CRT exactly as the <b>Clear display</b> key (or <b>CLEAR SCREEN</b> statement)
kdbcrt_controlcrt	sends information to a CRT control register
kdbcrt_controlkbd	sends information to a keyboard control register
kdbcrt_crtreadchar	reads one character from the specified location on the CRT
kdbcrt_crtscroll	scrolls the CRT area, from line <i>first</i> to line <i>last</i> , up or down one line
kdbcrt_cursor	removes the previous cursor and writes a new cursor to any on-screen alpha location
kdbcrt_disp_at_xy	allows text to be written to any alpha location on the CRT
kdbcrt_read_kbd	returns the buffer contents trapped and held by <b>ON KBD</b> (same as <b>KBD\$</b> )
kdbcrt_scrolldn	scrolls the <b>PRINT</b> area of the CRT down one line
kdbcrt_scrollup	scrolls the <b>PRINT</b> area of the CRT up one line
kdbcrt_statuscrt	returns the contents of a CRT status register
kdbcrt_statuskbd	returns the contents of a keyboard status register
kdbcrt_systemd	returns a string containing the results of calling the function <b>SYSTEM\$</b> for a given argument

## Device I/O

Device I/O in CSUBs is provided through the HP-UX device I/O library for HP-IB and GPIO interfaces and through the standard HP-UX **termio** routines for the RS-232 interface. For more information, read the chapter “Device I/O Library (DIL)” in the *HP-UX Concepts and Tutorials: Device I/O and User Interfacing* manual.

## **BASIC File I/O**

BASIC provides a set of routines found in the library ? `ibrmb.a` to allow operations on its file types. These file operations include:

- creating
- purging
- opening
- closing
- reading
- writing
- positioning.

Since CSUBs that use these routines cannot be tested outside of a BASIC program, more time should be allocated for their implementation to minimize the number of debugging iterations.

**Table 3-4. File Access Routines**

Routine	Description
csfa_fal_create	creates an HP-UX file
csfa_fal_create_bdat	creates a BDAT file
csfa_fal_create_ascii	creates an ASCII file
csfa_fal_close	closes a file
csfa_fal_eof	writes an EOF at the current file position
csfa_fal_loadsub_all	loads all subprograms from the specified PROG file and appends them to the program in memory
csfa_fal_loadsub_name	loads the subprogram from the specified PROG file and appends it to the program in memory
csfa_fal_open	opens a file for reading and writing.
csfa_fal_position	positions the file pointer to a specified logical record number
csfa_fal_purge	purges a file
csfa_fal_read	reads data item(s) from a file
csfa_fal_read_bdat_int	reads a BASIC 16-bit integer from a BDAT file
csfa_fal_read_string	reads a string from an ASCII, BDAT, or HP-UX file
csfa_fal_write	writes data item(s) into a file
csfa_fal_write_bdat_int	writes a BASIC 16-bit integer to a BDAT file
csfa_fal_write_string	writes a string to an ASCII, BDAT, or HP-UX file

3

## Writing FORTRAN CSUBs

---

When you are developing a system that involves the use of CSUBs, you should:

- write the BASIC program.
- determine what the CSUB should do, the parameters to be passed, and which variables should be accessible (*global*) to both the BASIC program and the CSUB.
- develop the CSUB; a listing of the BASIC program can be very helpful as reference during this task.

---

### Steps for Creating a FORTRAN CSUB

The following steps present an overview of the process needed to create a CSUB and the results of those steps. The CSUB related steps will be described in detail in later sections.

1. Enter BASIC, create and store the program that will call the CSUB. This program will contain CALLs to the CSUB, but the latter need not be implemented as it will be loaded later. You only need to decide what the subprogram will do and design the interface (parameters, COM, etc.) between BASIC and the CSUB.
2. Leave BASIC, enter HP-UX, and write the CSUB. You might, for example, use the vi editor to create the CSUB. See the sections “A Closer Look at FORTRAN CSUB Components” and “A Closer Look at Parameter Passing” for details on how to organize your FORTRAN subprograms and how to define your CSUB parameters.

3. Compile and debug the CSUB as much as possible by writing a FORTRAN test program. You may want to use the FORTRAN debugger, `fdb`, for this testing task.
4. Use the `ld` command to generate a fully linked relocatable object file containing all the CSUBs and any necessary HP-UX library support routines. For example,

```
ld -rd -a archive csub.o -u _printf -lrm -o csub
```

would create a fully linked relocatable object file called `csub` using the compiled file `csub.o` and the library `librm.a`. See the section “A Closer Look at Linking CSUB Object Files” for details.

5. Execute `rmbbuildc` and answer its prompts. This program generates the final BASIC PROG file. See the section “A Closer Look at Executing `rmbbuildc`” for details on how to answer the prompts.
6. Enter BASIC and load the BASIC PROG file from the keyboard or from the BASIC program using `LOADSUB`. This statement generates the necessary statements in your BASIC program to call CSUBs. See the section “A Closer Look at Managing CSUBs from BASIC” for details.
7. RUN the BASIC program which calls the desired CSUBs using the BASIC `CALL` or implied `CALL` statement.

## An Example: Finding the String

This section goes through each step of creating an example FORTRAN CSUB and how the CSUB is used in a BASIC program. All files used in the HP-UX environment are read from and written to the current directory. In the BASIC environment, the CSUB is loaded from the MASS STORAGE IS device (directory).

This simple program fills an array of string variables and then calls a FORTRAN CSUB which determines 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 FORTRAN CSUB. If the INTEGER variable `Yes` comes back with a value other than zero, it comes back pointing to the array element containing the matching string.



## Step 1: Create a BASIC Program that Calls the CSUB

Enter BASIC, edit and store this program in a file named FSTR. This file can be found in the directory called /usr/lib/rmb/demo.

```
10  LOADSUB ALL FROM "FIND_STRING"
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$(5)="FAVORITE COLOR?"
100 File$(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
```

4

## Steps 2 and 3: Write, Compile, and Debug the CSUB

Enter HP-UX, edit, compile, and debug the following subprogram called find\_string, and save it in the file named string.f. This file can be found in the directory called /usr/lib/rmb/demo.

```
subroutine find_string(file_dim,filex,str_dim,strx,num_strs,yes)
character*30 file_dim
character*22 filex(*)
character*30 str_dim
character*22 strx
integer*2 num_strs
integer*2 yes

integer*2 i, j, size1, size2

yes=0
i=1

read(strx(1:2), '(A2)') size2
do while ((i.le.num_strs).and.(yes.eq.0))
  read(filex(i), '(A2)') size1
```

```

if (size1.eq.size2) then
  if (size2.eq.0) then
    yes=i+1
  else
    j=3
    do while ((j.lt.(size1+2)).and.(filex(i)(j:j).eq.strx(j:j)))
      j=j+1
    end do
    if (filex(i)(j:j).eq.strx(j:j)) yes=i
  endif
endif
if (yes.eq.0) i=i+1
end do
end

```

4

#### Step 4: Generate a CSUB Object File

Link the code file `string.o` with the BASIC CSUB library `librmb.a` and the necessary HP-UX libraries to generate a fully linked relocatable CSUB object file. The HP-UX `ld` command should be used for this purpose, as follows:

```
ld -rd -a archive string.o -u _printf -lrmb -lI077 -lF77 -lc -o string
```

#### Step 5: Generate a BASIC PROG File (`rmbbuildc`)

Execute the `rmbbuildc` program and answer the prompts as shown below. Notice that a stream file is generated by the response to the first prompt; you can use this file the next time you execute the program (i.e. `rmbbuildc < stream`) to remove the need to interactively answer the prompts again.

RMB-UX Compiled Subprogram File Generator (Version 1.1)

```

Stream file name: stream
Output BASIC PROG file: FIND_STRING
CSUB object file names(s): string
Module name: Return
CSUB 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/C for Integer/Real/Complex): i
    Is this an array? (y/n): n
    Is this an optional parameter? (y/n): n
Parameter name: yes
  Parameter type (I/R/C for Integer/Real/Complex): i
    Is this an array? (y/n): n
    Is this an optional parameter? (y/n): n
Parameter name: 
Is there COM in this CSUB? (y/n): n

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

4

## Steps 6 and 7: LOAD and RUN the CSUB

In this example, `FIND_STRING` is automatically loaded from the BASIC program. Therefore, you only need to re-enter the BASIC system, `LOAD "FSTR"`, and `RUN` the program. The output should be:

```
The string was found in number 4
```

---

## A Closer Look at FORTRAN CSUB Components

Procedures in FORTRAN can be grouped into two main categories:

- subroutine subprograms
- functions.

In order to implement a FORTRAN CSUB, you will need to use the structure of a subroutine subprogram. FORTRAN functions may not be used since they return a value to the calling program.

---

## A Closer Look at Parameter Passing

In order to be useful, a CSUB needs the ability to exchange data with the calling BASIC program. This section describes the different ways of performing this data exchange. The primary method consists of defining the CSUB parameters to match those passed by BASIC. This method requires special attention to the different parameter types and formats of BASIC variables. The second method for a CSUB to exchange data with a BASIC program is via COM blocks. Again, this method necessitates the special handling of the variables defined in the blocks, according to their type and format.

4

### Passing Parameters by Reference

As far as FORTRAN is concerned, BASIC variables are always passed to a CSUB *by reference*. 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. Since all FORTRAN subprogram parameters are passed by reference, you need to make sure that the formal parameters of a CSUB match those passed by BASIC in number and in type.

The two key points to remember are:

- BASIC always passes a pointer to a variable.
- BASIC has no idea if a user-written CSUB has been properly coded to use that pointer.

Note that errors will occur if this distinction is overlooked.

### Parameter Types

The BASIC parameter types are not necessarily the same as their FORTRAN counterparts. It is important that the type of the parameters of a CSUB be correct so that BASIC and the CSUB can interface properly. This section will explain the type differences in detail.

The following table provides you with a quick reference to equivalent FORTRAN and BASIC parameter types.

**Table 4-1. Equivalent FORTRAN and BASIC Parameter Types**

BASIC Parameter Type	FORTRAN Parameter Type
REAL	real*8
INTEGER	integer*2
COMPLEX	complex*16
<i>string_name</i> \$	Two parameter types passed for strings: <ul style="list-style-type: none"> <li>• character*30</li> <li>• character*n</li> </ul> where <i>n</i> is the DIM length of the string plus 2.
<i>array_name</i> [ <i>lower</i> : <i>upper</i> , <i>etc.</i> ] of one of the above numeric parameter types or <i>string_array</i> \$( <i>lower</i> : <i>upper</i> , <i>etc.</i> )[ <i>num_chars</i> ]	Two parameter types passed for arrays: <ul style="list-style-type: none"> <li>• character*30</li> <li>• one of the above numeric or string array types.</li> </ul>
@ <i>io_path_name</i>	character*190

4

## REAL

A variable defined as a **real\*8** in FORTRAN maps into a BASIC REAL. Therefore, a BASIC REAL parameter can be defined in a CSUB as:

```
subroutine x(y)
  real*8 y
```

## COMPLEX

A variable defined in BASIC as COMPLEX is a floating point value with real and imaginary parts. It maps directly into a FORTRAN variable of type **complex\*16**.

## INTEGER

A variable defined in BASIC as INTEGER is a 16-bit quantity. It is therefore equivalent to a FORTRAN variable of type `integer*2`.

## Strings

Strings are different in BASIC and FORTRAN, both in their structure and the way they are passed. The structure of a FORTRAN string is simply an array of characters while the BASIC string has a two-byte length field followed by its characters.

4 BASIC passes its strings as two parameters:

- The first parameter is a pointer to a record containing information about arrays, strings, their maximum lengths, and their lower and upper bounds. Since there is no aggregate type in FORTRAN, this record must be declared as an array of characters. For the case of a scalar (non-array) string, this array will simply contain the maximum length of the string, a 16-bit quantity. To access this parameter, you should use the FORTRAN statement `READ` to extract its value from the array into a variable. Similarly, you should use the statement `WRITE` to change its value.
- The second parameter is a pointer to another record specifying the string **value area**. This area contains the actual length of the string, a 16-bit quantity, and its characters. Again, this record is mapped into an array of characters and manipulated as described above. To determine the length of this array, simply add 2 to the dimensioned length of the string. Thus, for a BASIC string whose dimensioned length is 20, the length of the equivalent FORTRAN character array would be 22.

An example of how this would look in a FORTRAN CSUB is as follows:

```
subroutine getstring(dim_len, b)
character*30 dim_len
character*82 b

integer*2 i
character*20 s
integer*2 s_len, b_maxlen, b_len

s='a string'
s_len=8
read(dim_len(1:2), '(A2)') b_maxlen

if (s_len.gt.b_maxlen) s_len=b_maxlen
b_len=s_len
do i=1, b_len
  b(2+i)=s(i)
end do

write(b(1:2), '(A2)') b_len
end
```

4

The above subprogram copies a FORTRAN string into a BASIC string. From this example, you should note how:

- the string parameter is declared, paying attention to the length of the arrays,
- the maximum length of the string is retrieved from its dimension record using the `read` statement,
- an explicit check has been made to insure that the length of the FORTRAN string is not greater than the maximum length of the BASIC string,
- the FORTRAN string value is put into the BASIC string value area,
- the new length of the BASIC string is stored in its value area using the `write` statement.

Although a CSUB receives two parameters for a BASIC string, as far as BASIC is concerned, there is only one actual parameter to be passed, as shown below.

```
...
10 DIM Str$(80)
20 CALL GETSTRING(Str$)
30 PRINT Str$
40 END
```

## I/O Paths

An I/O path is a block of storage used to keep track of the state of a file or I/O device. In FORTRAN, this block is mapped into a 190-character array. See the section “BASIC File I/O” for details on how you would use an I/O path as a file control block with the file access library (**fal**) routines.

The following example shows how a FORTRAN CSUB receives an I/O path parameter from BASIC.

```
subroutine x(y_ptr)
character*190 y_ptr
```

The typical use of the parameter `y_ptr` in a CSUB would then take the form:

```
call csfa_fal_open(filename, y_ptr)
```

## Arrays

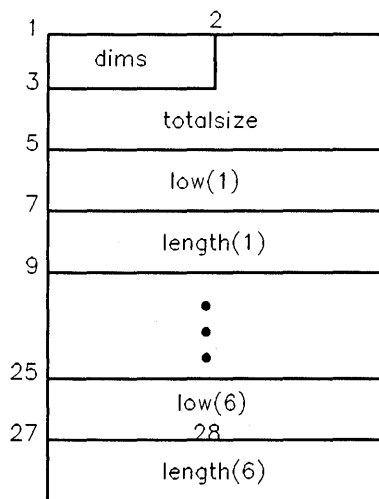
Arrays are passed as two parameters:

- a pointer to the dimension record. The fields in the dimension record, in this case, are more complicated. As with string arrays, this dimension record is mapped into an array of characters and is manipulated using the FORTRAN statements **read** and **write**.
- a pointer to the value area.

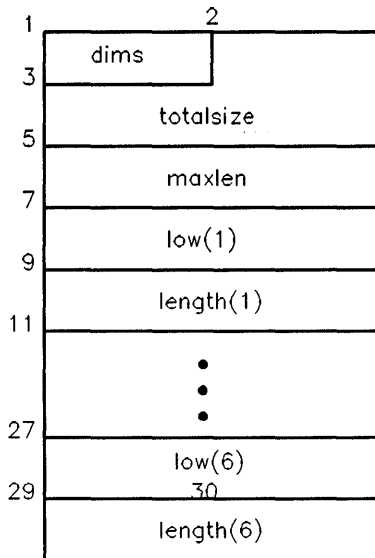


The actual dimension records for the REAL, INTEGER, COMPLEX, and string arrays are represented by Figures 4-1 and 4-2.

- dims            is a byte containing the number of dimensions
- totalsize       is the number of bytes in the entire array
- low(n)         is the lower bound of the nth dimension
- length(n)      is the number of elements in the nth dimension
- maxlen         is the maximum length of any element in a string array.



**Figure 4-1.**  
**REAL, INTEGER, or COMPLEX Array Dimension Record**



**Figure 4-2. String Array Dimension Record**

The above figures will help you determine the character array bound values to specify to the `read` and `write` statements when accessing or modifying a given field in the records. For example, to access the maximum length field (`maxlen`) of a string array parameter, you can refer to Figure 4-2 and use the following code:

```

subroutine x(s_dim, s)
character*30 s_dim
character*22 s(*)

integer*2 s_maxlen

read(s_dim(5:7), '(A2') s_maxlen

```

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

```
subroutine x(d, arr)
character*30 d
integer*2 arr(*)
```

Again, this is a single parameter in BASIC. So, in this example, BASIC would send an array defined as:

```
INTEGER Arr(1:10)
CALL X(Arr(*))
```

### Defining BASIC and FORTRAN Arrays

The BASIC and FORTRAN arrays should be defined the same way. This is not mandatory, but helpful. You should remember that an array with bounds [1..5] is the same as an array with bounds [6..10]; both are five-element arrays. If a BASIC array defined as INTEGER Arr(6:10) is passed to a CSUB array parameter defined as integer\*2 arr(\*), the sixth element in the BASIC array will correspond to the first element in the FORTRAN CSUB array.

4

### Dimensioning an Array

You should note that array elements are stored in row-major order in BASIC and in column-major order in FORTRAN. It is therefore best to declare a multi-dimensional BASIC array as a one-dimensional array in FORTRAN and do explicit subscript calculations based on the information in the dimension record of the array.

### Redimensioning an Array

The DIM statement should be used in conjunction with the appropriate FORTRAN array declaration to define the space for the BASIC array. The REDIM statement does not affect the size of that space. It does however affect the BASE and SIZE functions and the length and low values in the dimension record. Note that a CSUB should therefore check the dimension structure of an array parameter to find its current dimensions.

If you use the REDIM statement on a multi-dimensional BASIC array, the declaration of the array will be invalid the next time the array is accessed in a CSUB. To be immune from the effects of REDIM, it is again best to declare a multi-dimensional BASIC array as a one-dimensional FORTRAN array.

## Declaring the Value Area of a String

You should pay special attention to the definition of a BASIC string array to make sure that the value area of the array is properly dimensioned in FORTRAN. Each element of the array should have the same structure and size as a single string CSUB parameter with the same dimensions. For example, with the following BASIC definition,

```
DIM S$(1:10)[20]
```

you would define the equivalent FORTRAN string array as:

```
subroutine x(s_dim, s)
character*30 s_dim
character*22 s(10)
```

## Optional Parameters

You can declare some or all parameters of a CSUB as *optional* through responses to `rmbuildc`. Optional parameters are those which are not required in the parameter list of the calling code. In a FORTRAN CSUB, however, there is no distinction between required and optional parameters as both types of parameters have to be listed in the declaration of the CSUB. See “Subprograms” in the “BASIC Programming Techniques” manual for more information about optional parameters.

BASIC will pass a NIL pointer to the CSUB when one of its parameters that has been declared as optional is omitted. The CSUB should therefore always make an explicit NIL check before trying to use the value of an optional parameter. Otherwise, a run-time error may occur when attempting to access the parameter.

For example, if the BASIC declaration of a CSUB is:

```
100 CSUB My_csub(REAL R, OPTIONAL REAL Opt)
```

the variable `Opt` will have an address of `NIL` if it is not passed. In FORTRAN, the CSUB should perform the following test:

```
subroutine my_csub(required, optional)
real*8 required, optional

integer*4 i

i=%loc(optional)
if (i.ne.0) then
...
```

4

## Accessing BASIC COM from a CSUB

Another way for a BASIC program and a FORTRAN CSUB to interchange data is via COM blocks. This method allows you to map a BASIC COM block into a FORTRAN COM block in order to access BASIC variables from a CSUB. On entry of the CSUB, you would first copy the contents of a BASIC COM block into a FORTRAN COM block. After accessing or modifying the desired variables in the FORTRAN COM block, you would copy the contents of this block back into the BASIC COM block to update the values of the variables modified by the CSUB.

### Using the `basic_com` Routine

The routine provided for copying the contents of BASIC and FORTRAN COM blocks is named `basic_com`. It takes the four following parameters:

`character*80 com_name` is the name of the BASIC COM block for which the copy operation is to be performed. Since upper and lower case letters are significant in a COM block name, they should be specified the same way BASIC does. To access an unlabeled COM block, you should specify a string with a single blank as the COM block name. This string parameter should always be terminated with a null character.

<code>integer*4 com_start</code>	is the starting address of the FORTRAN COM block. It can be retrieved by using the <code>%loc</code> function with the first variable of the block as the argument to the function.
<code>integer*4 com_end</code>	is the last address of the FORTRAN COM block. It can be retrieved by using the <code>%loc</code> function with the last variable of the block as the argument to the function. A dummy variable should be declared in the COM block for this purpose.
<code>integer*4 copy_flag</code>	specifies the direction of the copy operation. A value of 0 will copy the contents of the specified BASIC COM block into the FORTRAN COM block while a value of 1 will copy the contents of the FORTRAN COM block into the BASIC COM block. The routine will also set this parameter to 0 for a successful copy operation and set the parameter to -1 if it was unable to perform the operation because of an invalid parameter value, such as an invalid BASIC COM name.

4

### Defining a FORTRAN COM Block Structure

In defining the FORTRAN COM block structure for accessing a BASIC COM block, you will need to know the layout of the BASIC block in advance since there is no way of determining this layout from the FORTRAN CSUB at run-time. You should also know that the order of the variables in the FORTRAN block should be the opposite from that of the variables in the BASIC block. For example, the first variable in the FORTRAN block should be mapped into the last variable in the BASIC block. You should also know that `basic_com` will not prevent you from inadvertently corrupting the COM blocks by writing beyond their boundaries or by storing invalid values.

Also, you should note that only the value area of BASIC strings and arrays are stored in a COM block. Therefore, you should omit the specification of the dimension record for those variables in the FORTRAN COM block.

Finally, you should always use the **+A** FORTRAN compiler option when compiling FORTRAN CSUBs with COM declarations. This option is necessary to guarantee the alignment of the variables in FORTRAN COM blocks and their BASIC counterparts.

This is an example of accessing a BASIC COM block defined as:

```
COM /Num1/ INTEGER A,B(1:5),REAL C,D$[10]
```

To access this block from a CSUB, you could use:

```
subroutine one

character*12 d
real*8 c
integer*2 b(5)
integer*2 a
common /some_com/ d,c,b,a,dummy

integer*4 flag

flag=0
call basic_com('Num1'//char(0), %loc(d), %loc(dummy), flag)

a=26
c=1214.61
...
flag=1
call basic_com('Num1'//char(0), %loc(d), %loc(dummy), flag)
end
```

4

---

## A Closer Look at Linking CSUB Object Files

After thoroughly testing your CSUBs in a FORTRAN program, you are now ready to generate a CSUB object file. This object file brings together the CSUB binaries and the necessary libraries to satisfy the external references of those binaries.

## Using the Linker

The HP-UX linker can be used to generate a fully linked, relocatable CSUB object file suitable for input to the `rmbbuildc` program. The syntax for linking is as follows:

```
ld -rd -a archive csub_binaries -u _printf -lrm [other_libraries] -o csub_obj_file
```

where:

- |                                      |   |
|--------------------------------------|---|
| <code>-rd</code>                     | are the options to retain relocation information in the output file for subsequent re-linking and to force the definition of “common” storage.  |
| <code>-a archive</code>              | tells the loader to use archive libraries instead of shared libraries.  |
| <code><i>csub_binaries</i></code>    | are the <code>.o</code> files generated by the FORTRAN compiler. They contain the code for CSUBs and should always be followed in the above command line by the library <code>-lrm</code> . |
| <code>-u _printf</code>              | links needed object files.  |
| <code><i>other_libraries</i></code>  | are libraries that are required to resolve all the references not satisfied by <code>librm.a</code> ; they may be your own or HP-UX libraries.  |
| <code>-o <i>csub_obj_file</i></code> | takes the output of the <code>ld</code> command and sends it to the file. <code><i>csub_obj_file</i></code> .   |

## Specifying the `librm.a` Library

In addition to providing useful system access routines for CSUBs, this library also specifies all the external symbols from HP-UX libraries which were used in the actual implementation of a specific version of BASIC. Specifying the library immediately after your CSUB binaries in the `ld` command will thus ensure the sharing of those symbols between BASIC and your CSUBs. This sharing first removes the need for duplicating the code already available in BASIC in the CSUB object files. More importantly, it is also necessary for the proper operation of libraries, the routines of which maintain global variables which should not be duplicated in the CSUB object files. An example of such



routines are the HP-UX memory management routines (malloc 3C includes: calloc, free, mallopt, mallinfo, and realloc).

## Resolving External References

In general, specifying the library `librmb.a` alone in your link command will be sufficient to resolve all external references from your CSUB binaries; this is because BASIC uses most of the common HP-UX libraries in its implementation.

Nevertheless, to verify that your CSUB object file is indeed fully linked, you should use the following command, which will notify you of any unresolved external references (e.g., if a CSUB is calling a function in an HP-UX library not specified in your link (`ld`) command).

```
nm -u CSUB_object_file
```

In the event that you do have undefined external references, you should resolve them by first determining the libraries in which the symbols are defined and then specifying those libraries in your link command. Again, it is extremely important that you only list those libraries after the library `librmb.a`.

---

## A Closer Look at Executing `rmbuildc`

After successfully generating one or more CSUB object files without unresolved external references, you are now ready to use the `rmbuildc` program to create the associated BASIC PROG file for the CSUBs.

The purpose of `rmbuildc` is to generate a BASIC subprogram **interface** for each of the CSUBs. This interface consists of the:

- name of the subprogram
- name and type of its parameters
- optional specification of any COM block used.

The program will interactively prompt you for all the necessary information about a CSUB to generate its BASIC subprogram interface.

---

**Note**

On all the yes/no prompts in `rmbuildc`, a response other than **Y** is treated as a negative response. A null response for a prompt is specified by only pressing **Return**.

---

## Procedure for Using `rmbuildc`

The following sections contain steps that explain how to use `rmbuildc`.

### Step 1: Executing `rmbuildc`

In HP-UX, execute:

```
rmbuildc Return
```

The `rmbuildc` program begins by prompting you for the name of an optional stream file.

```
RMB-UX Compiled Subprogram File Generator (Version 1.1)
```

```
Stream file name:
```

In this file, the program will record all your responses to its prompts so that you can use the file as its standard input in subsequent executions (e.g., `rmbuildc < file_name`). Note that you may give a null response if you do not want a stream file.

### Step 2: Entering a PROG File Name

The next prompt is:

```
Output BASIC PROG file name:
```

This prompt asks for the BASIC PROG file name that you will specify in a `LOADSUB` statement to load the desired CSUBs into a BASIC program.

### Step 3: Naming CSUB Object Files

To the prompt:

```
CSUB object file name(s):
```

list the names of the CSUB object files generated by the linking procedure. Note that each object file will correspond to a specific version of BASIC. The file names specified should be separated by one or more spaces.

#### Step 4: Specifying CSUB Interfaces

The remaining prompts deal with the specification of each CSUB interface. The following steps explain the prompts that you will encounter during this specification.

1. Press the **Return** key in response to the prompt below. This prompt is only relevant for Pascal language CSUBs and should always be handled as specified.

Module name:

2. Enter the name of a CSUB when the following prompt is given.

CSUB name:

This prompt will be repeated until a null CSUB name is specified by pressing the **Return** key.

3. The following four prompts are repeated for each parameter of the current CSUB until a null parameter name is entered. You should specify the name of the parameter, its type, whether it is an array, and whether it is optional.

Parameter name:

Parameter type (I/R/C for Integer/Real/Complex):

Is this an array? (y/n):

Is this an optional parameter? (y/n):

Since you are declaring the interface of the BASIC subprogram for the current CSUB, the names of the parameters that you specify need not match that of the CSUB. However, the types of the parameters between the FORTRAN CSUB and its BASIC subprogram declaration should match. For example, the BASIC CSUB call may be similar to the following:

```
Read_result(INTEGER Value_ret)
```

and the actual parameter name in the CSUB may be as follows:

```
subroutine read_result(return_int)
integer*2 return_int
```

The legal BASIC types are INTEGER, REAL, COMPLEX, string, and I/O path. If the parameter name ends in a dollar sign (\$), then the type is automatically assumed to be string, and if it begins with an @, the type is assumed to be an I/O path. Otherwise, you should answer with , R, or C. An array parameter is assumed to have dimension (\*), which indicates that it will be defined in the calling BASIC program.

Once a parameter is specified as optional, all the following parameters default to being optional and the prompt will not reappear.

Since `rmbuildc` cannot check the parameter list of a CSUB to verify that it matches its BASIC declaration, it is your responsibility to make sure that the matching is done correctly. Otherwise, the CSUB will behave unpredictably when called from a BASIC program.

4. Indicate whether the current CSUB accesses a BASIC COM by responding with either Y or N to the following prompt.

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

It is not required that you declare a COM that is accessed in the CSUB. By declaring it, however, the COM will remain in memory as long as the CSUB is present, even after the BASIC subprogram that defines it is deleted. If you answer Y to the above prompt, the following prompts will appear. Respond to them with the information that they request. Note that the number of COM blocks that you specify will depend on the number of COMs that you want to access. With the exception of the first prompt given below, all of the prompts are repeated for each COM block. For an unlabeled COM, the label will be null.

Number of COM blocks:

COM label:

Item name:

Item type (I/R/C for Integer/Real/Complex):

Is this an array? (y/n):

The questions concerning name, type, and array are the same as for the parameters, with the exception that the bounds of an array in a COM block have to be explicitly specified. To specify these bounds, enter the appropriate information to the prompts given below.

Number of dimensions or \*:

Dimension n lower bound:

Dimension n upper bound:

If the number of dimensions is entered as “\*” the other dimension prompts are not displayed. This option may be used only if the array is defined either in the calling BASIC routine or a previous CSUB. If there is an explicit number of dimensions, the lower and upper bounds need to be entered for each of the dimensions.

If the type of the item is string, the following prompt will appear. In response to this prompt, enter the DIMensioned or maximum string length allowed for this COM item.

String length:

The final question about the COM item is:

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

You should answer  unless you plan to use the variable as a buffer in a TRANSFER statement. Read the “Transfers and Buffered I/O” chapter of the *HP BASIC 6.2 Programming Guide* for details. This last prompt completes the set of prompts related to specifying COM blocks.

The next prompt to appear will be a request for the name of another CSUB. If you wish to specify another CSUB, you should enter its name at this time and repeat steps 3 and 4 of this section. Otherwise, simply press the  key. This enters a null CSUB name and the following prompt appears:

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

If you respond with  to this prompt, `rmbuildc` will prompt you for another module name and you will need to repeat steps 2 through 4 of this section. Otherwise, press  and `rmbuildc` will proceed to construct the output BASIC PROG file. To facilitate the BASIC coding process, it will also print to a file the COM declarations required by all the CSUBs in BASIC source form. This file will reside in the current directory and will have the PROG file’s name with `_COM` appended to it.

## rmbuildc Errors

The following table describes all the errors generated by `rmbuildc`.

**Table 4-2. rmbuildc Errors**

Error	Description
Opening of <i>file_name</i> failed.	An attempt at creating, writing to, or reading from the specified file caused an error.
Maximum number of input files exceeded.	Too many CSUB object files were specified. The maximum number of files currently allowed is 10.
File <i>file_name</i> has unresolved references.	The specified CSUB object file still contains undefined symbols, which should be resolved by linking the necessary additional libraries to the object file.
File <i>file_name</i> has no CSUB version stamp.	The specified CSUB object file was not properly linked with the library <b>librmb.a</b> . You should verify that you have a valid version of this library.
File <i>file_name</i> has an invalid a.out format.	The specified CSUB object file does not conform to the format specified for an object file.
Bound must be between -32768 and 32767.	The lower and upper bounds for a dimension of a COM block array item must be within the specified range.
High bound value is less than low bound value.	The high bound value of an array dimension must be greater than or equal to the low bound value of the same dimension.
Maximum number of elements in a dimension exceeded.	The maximum number of elements in the dimension of an array has been exceeded.
No item was specified.	A COM block without any items was specified.

4

**Table 4-2. rmbuildc Errors (continued)**

Error	Description
Type must be integer, real, or complex.	The type of this parameter or COM item should be integer, real, or complex.
Value must be 6 or less, or be an asterisk.	The number of dimensions for an array must be within the limits specified.
Value must be between 1 and 32767.	The value for the requested parameter must be within the specified range.
Procedure <i>CSUB_name</i> was not found.	The definition for the specified CSUB was not found in one of the CSUB object files.
No procedure name was specified.	At least one CSUB should be specified during each execution of <b>rmbuildc</b> for the output BASIC PROG file to be meaningful.
Invalid specification of COM item <i>item_name</i> .	The amount of memory that would be required to store the elements of the specified COM item exceeds the system limits.
Memory overflow.	The program is unable to allocate the necessary memory for processing unusually large CSUB input object files. Re-execute <b>rmbuildc</b> with a single command line parameter specifying the estimated number of bytes needed to handle the large object files; the default is 1000000 bytes.

---

## A Closer Look at Managing CSUBs from BASIC

This section shows how to manage CSUBs in BASIC. Topics covered are:

- Loading CSUBs into a BASIC Program
- Deleting CSUBs
- Handling CSUB Run-Time Errors

### Loading CSUBs into a BASIC program

Before you can call a CSUB from a BASIC program, you have to load it in BASIC memory with a command similar to this:

```
LOADSUB ALL FROM "file_name"
```

where *file\_name* is the name of a BASIC PROG file generated by `rmbuildc`. This command loads and lists all subprograms and CSUBs in the CSUB libraries (a CSUB library is a set of CSUBs generated by a single execution of `rmbuildc`) found in the PROG file. Because the CSUBs are now listed in your BASIC program, you have access to all of them.

If you just want to list a particular CSUB in your BASIC program, you would use a command similar to this:

```
LOADSUB "sub_name" FROM "file_name"
```

where *sub\_name* is the CSUB that you want listed in your BASIC program, and *file\_name* is the BASIC PROG file where this CSUB is located. With this command, you will only have access to the specified CSUB and to those following it in its library. You should note, however, that just selecting one CSUB from a library does not save you memory because the entire library is always loaded into memory if one of its CSUBS is specified.

Once CSUBs are loaded into a BASIC program, they can be stored in a PROG file with the `STORE` command. Therefore, it is possible to have a PROG file consisting of several CSUBs generated at different times that were merged together using several `LOADSUB` and `STORE` commands. For example, consider a PROG file called *file\_name* with the following CSUB libraries:



```
(begin CSUB library A)
Csuba
Csubb
(end CSUB library A)
Subc
(begin CSUB library B)
Csubd
(end CSUB library B)
(begin CSUB library C)
Csube
Csubf
Csubg
(end CSUB library C)
Subh
Subi
```

If `Csubd` and `Csubf` are referenced in a program, executing `LOADSUB FROM "file_name"` will cause CSUB libraries B and C to be loaded in memory. Other CSUB libraries and subprograms are not brought in unless they are also referenced.

## Deleting CSUBs

All CSUBs belonging to the same CSUB library are listed contiguously and must remain in the order in which they were generated by `rmbuildc`. You cannot add BASIC program lines between CSUB declaration statements. However, you can delete CSUBs from a BASIC program by using the `DELSUB` command. Note that you can only delete the CSUB which comes first in a CSUB library. If you delete a CSUB not listed first in its library, BASIC will generate an error when you attempt to call any of the remaining CSUBs in the library.

### Example of Deleting CSUBs

In the following example, you may delete the CSUBs `Csube` and `Csubf` in order and you will still be able to call `Csubg`. However, you cannot delete `Csubb`, leaving `Csuba`, because BASIC will generate an error when you subsequently attempt to call `Csuba`.

4

```
(begin CSUB library A)
Csuba
Csubb
(end CSUB library A)
Subc
(begin CSUB library B)
Csubd
(end CSUB library B)
(begin CSUB library C)
Csube
Csubf
Csubg
(end CSUB library C)
Subh
Subi
```

## Handling CSUB Run-time Errors

Determining the cause of a CSUB run-time error is a difficult task because there is no BASIC debugger similar to the FORTRAN debugger which will let you step through your CSUB code. For this reason, you should thoroughly test your CSUBs in a test program before attempting to call them from BASIC. In situations when this approach is not practical, such as when using the BASIC file I/O routines, you may need to insert print statements in your CSUBs to monitor their execution in a BASIC program.

In FORTRAN, a run-time error not explicitly handled will cause a program to terminate. You should not allow this condition to happen in a FORTRAN CSUB since it will also cause the invocation of BASIC to abnormally terminate. Instead, you should anticipate all possible causes of CSUB run-time errors and provide code to handle those errors. The error handling code can then report the same errors to BASIC.

## Trapping Errors

In order to trap a FORTRAN run-time error, a CSUB will need to use the `IOSTAT=ios`, `ERR=label`, and the `END=label` specifiers. These specifiers will determine the disposition of an error arising from using one of the language statements. You should note that there exist run-time errors that can not be handled with the above specifiers and that will always cause a program

termination. You should make sure that those errors cannot occur in the final version of your CSUBs.

## Reporting Errors to BASIC

In order to provide a consistent error recovery mechanism for CSUBs, BASIC has defined an error number for reporting all CSUB run-time errors. The basic idea is for a CSUB to exclusively use this error number for reporting all signal and error conditions to BASIC; the parameterless routine `csub_error` should be used for this purpose.

The calling BASIC code can then recover from this error with an `ON ERROR CALL/RECOVER` statement and get the actual signals or errors by reading some global variables (e.g. COM block variables) accessible to the reporting CSUB or by calling another CSUB. This approach removes the need to map CSUB error numbers to BASIC error numbers, allows CSUB libraries from different sources to be shared within a program, and simplifies the task of recovering from a CSUB run-time error.

In situations when BASIC will not respond at all to user input or behave unpredictably during and after the execution of a CSUB due to an unrecoverable error condition, the current invocation of BASIC is likely to be corrupted and should be terminated with the `kill` command.

---

## Accessing System Resources

One of the motivations for using CSUBs with BASIC is the ability to access a rich set of HP-UX system libraries. This section covers the restrictions on the use of these libraries. The restrictions are due to the fact that BASIC also uses the libraries in its implementation.

- Simple Keyboard and Printer I/O
- Device I/O
- BASIC File I/O

## Simple Keyboard and Printer I/O

Operations on the standard I/O streams, writing to the screen and reading from the keyboard are not supported. Therefore, FORTRAN statements like `READ` and `WRITE` should not be used with the preconnected logical unit numbers. To allow a CSUB to input characters from the keyboard and to write to the `PRINTER IS` device, BASIC provides a set of routines in found in the library `librmb.a` for keyboard and CRT register access, character input and output, scrolling, and cursor manipulation.

**Table 4-3. Keyboard and CRT I/O Routines**

Routine	Description
<code>kbdcr_t_clear_screen</code>	clears the alpha CRT exactly as the <code>Clear display</code> key (or <code>CLEAR SCREEN</code> statement)
<code>kbdcr_t_controlcrt</code>	sends information to a CRT control register
<code>kbdcr_t_controlkbd</code>	sends information to a keyboard control register
<code>kbdcr_t_crtreadchar</code>	reads one character from the specified location on the CRT
<code>kbdcr_t_crtscroll</code>	scrolls the CRT area, from line <i>first</i> to line <i>last</i> , up or down one line
<code>kbdcr_t_cursor</code>	removes the previous cursor and writes a new cursor to any on-screen alpha location
<code>kbdcr_t_disp_at_xy</code>	allows text to be written to any alpha location on the CRT
<code>kbdcr_t_read_kbd</code>	returns the buffer contents trapped and held by <code>ON KBD</code> (same as <code>KBD\$</code> )
<code>kbdcr_t_scrolldn</code>	scrolls the <code>PRINT</code> area of the CRT down one line
<code>kbdcr_t_scrollup</code>	scrolls the <code>PRINT</code> area of the CRT up one line
<code>kbdcr_t_statuscrt</code>	returns the contents of a CRT status register
<code>kbdcr_t_statuskbd</code>	returns the contents of a keyboard status register
<code>kbdcr_t_systemd</code>	returns a string containing the results of calling the function <code>SYSTEM\$</code> for a given argument

## Device I/O

Device I/O in CSUBs is provided through the HP-UX device I/O library for HP-IB and GPIO interfaces and through the standard HP-UX `termio` routines for the RS-232 interface. For more information, read the chapter “Device I/O Library (DIL)” in the *HP-UX Concepts and Tutorials: Device I/O and User Interfacing* manual.

## BASIC File I/O

BASIC provides the a set of routines in the library `librmb.a` to allow operations on its file types. These file operations include:

- creating
- purging
- opening
- closing
- reading
- writing
- positioning.

Since CSUBs that use this module cannot be tested outside of a BASIC program, more time should be allocated for their implementation to minimize the number of debugging iterations.

**Table 4-4. File Access Routines**

Routine	Description
csfa_fal_create	creates an HP-UX file
csfa_fal_create_bdat	creates a BDAT file
csfa_fal_create_ascii	creates an ASCII file
csfa_fal_close	closes a file
csfa_fal_eof	writes an EOF at the current file position
csfa_fal_loadsub_all	loads all subprograms from the specified PROG file and appends them to the program in memory
csfa_fal_loadsub_name	loads the subprogram from the specified PROG file and appends it to the program in memory
csfa_fal_open	opens a file for reading and writing.
csfa_fal_position	positions the file pointer to a specified logical record number
csfa_fal_purge	purges a file
csfa_fal_read	reads data item(s) from a file
csfa_fal_read_bdat_int	reads a BASIC 16-bit integer from a BDAT file
csfa_fal_read_string	reads a string from an ASCII, BDAT, or HP-UX file
csfa_fal_write	writes data item(s) into a file
csfa_fal_write_bdat_int	writes a BASIC 16-bit integer to a BDAT file
csfa_fal_write_string	writes a string to an ASCII, BDAT, or HP-UX file

## CSUB Prototyper Utility

---

The CSUB prototyper saves time in the creation of CSUBs. It consists of CSUBs and BASIC functions that let you call CSUB routines from BASIC in their “native” language (FORTRAN, Pascal, C, or assembly). In order to call CSUB routines in this manner, prototyper functions convert actual parameter types of the BASIC calling routine to the formal parameter types of the CSUB routine being called.

Before continuing, you should review the chapter in this manual that pertains to your particular CSUB language.

---

### Why Use the CSUB Prototyper?

The CSUB prototyper has two functions:

- To simplify the creation of CSUBs.
- To provide a means for dynamically calling CSUBs at run-time.

### Creating CSUBs

The following table gives a comparison between the procedural steps for creating a CSUB using the standard method and using the prototyper.

**Table 5-1. Comparison of CSUB Creation Procedures**

Step	Standard CSUB Procedure	Prototyper CSUB Procedure
1	Create the BASIC program that is to call the CSUB(s).	Same.
2	Exit BASIC and enter the HP-UX environment.	Same.
3	Select an editor (vi for example) and write a program that contains your CSUBs. This program can be written in FORTRAN, Pascal, C, or assembly language. Run the program and debug it until it works.	Same.
4	Provide the proper interface for the CSUB(s) by defining the formal parameters of the CSUB(s).	<i>This step is not required.</i>
5	Link the necessary libraries to your compiled CSUB to generate a CSUB object file.	Same.
6	Execute <i>rmbuildc</i>	<i>This step is not required.</i>
7	Load the CSUB(s) by executing the LOADSUB command from the keyboard or the BASIC program that is calling the CSUB(s).	Load the CSUB(s) by specifying the CSUB object file to the appropriate prototyper CSUB.

The table shows that the CSUB prototyper shortens the steps required to create a CSUB. This will save time, but you will need to have enough memory to hold the object code for an entire program that contains the CSUB(s).

**5-2 CSUB Prototyper Utility**



## Calling CSUBs Dynamically

The CSUB prototyper also opens up new possibilities for a BASIC application whose requirements for compiled language routines can only be determined at run-time. In this situation, you need not anticipate the usage requirement of the application by defining a CSUB interface for all of the possible routines that may be invoked. Instead, the selected routines may be called by name with the CSUB prototyper. By carefully restricting this method of accessing CSUBs to those routines that are not time critical, you can use this feature to give an application greater flexibility in the area of dynamic code loading without impairing its performance.

---

## Using the Prototyper to Create a CSUB

This section develops a complete example that illustrates all the major steps involved in using the CSUB prototyper. You may find it useful to refer to the section "Steps for Creating a CSUB" in chapter 2, 3, or 4 of this manual, depending on the language you are using to create your CSUB. In our example, we will use the C language.

5

## Writing CSUB Routines in C

The procedure for writing CSUB routines consists of five steps.

1. Using the CSUB prototyper in a BASIC program.
2. Exiting BASIC to HP-UX.
3. Writing C subroutines.
4. Generating a relocatable object file.
5. Running the BASIC program.

## Step 1: Using the CSUB Prototyper in a BASIC program

This BASIC program shows a typical session with the CSUB prototyper.

```
100 REAL A           ! Real variable file descriptor.
110 REAL J
120 REAL R
130 DIM S$(30)
140 ! Load the CSUB prototyper library.
150 LOADSUB ALL FROM "CPR"
160 ! Load the object file containing the CSUB routines.
170 Cprload(A,"example")
180 ! Call procedure "sub1"
190 Cpr(FNInit(A,"_sub1"),FNI(50),FNL(J,1),FNS(S$,1))
200 PRINT "J is",J
210 PRINT "S$ is",S$
220 ! Call function "sub2" which returns a 64-bit floating
230 ! point value.
240 R=FNCpr64(FNInit(A,"_sub2"),FND(45))
250 PRINT "R is",R
260 ! Unload the object file.
270 Cprunload(A)
280 END
```

5

The output of this program is as follows:

```
J is      50
S$ is     sub1
R is      90
```

This program illustrates the major operations provided by the CSUB prototyper:

- Line 100 declares a REAL variable file descriptor called A.
- Line 150 loads the CSUB prototyper library called CPR.
- Line 170 loads the object file containing the desired compiled language routines (`_sub1` and `_sub2`) into memory. The prototyper command called `Cprload` is used to load the object files. Note that the REAL variable A is assigned as the file descriptor for the object file called `example`. File descriptor A will be used when you reference the compiled routines `_sub1` and `_sub2`.
- Line 190 selects the prototyper *execute* CSUB called `Cpr`. Note that this *execute* CSUB is type *void*, which is the same type as the compiled routine

called `_sub1`. A list of *execute* CSUBs and functions can be found in the table entitled “Mapping Between BASIC Return Value Type and Prototyper *execute* CSUBs or Functions.” The parameters for this *execute* CSUB are as follows:

- `FNInit(A,"_sub1")`—selects the compiled routine that is to be called.
- `FNI(50)`—passes the value 50 as a 16-bit INTEGER to `_sub1`.
- `FNL(J,1)`—returns the 32-bit REAL value J to the calling routine.
- `FNS(S$,1)`—returns the string S\$ to the calling routine.
- Line 240 selects the prototyper function called `FNCpr64`. Note that this function is type REAL (64-bit floating-point compiled routine return value), which is the same type as the compiled routine called `_sub2`. A list of *execute* CSUBs and functions can be found in the table entitled “Mapping Between BASIC Return Value Type and Prototyper *execute* CSUBs or Functions.” The function parameters for this function are as follows:
  - `FNInit(A,"_sub2")`—selects the compiled routine to be called.
  - `FND(45)`—passes the value 45 as a 64-bit floating point REAL number to `_sub2`.
- Line 270 uses the unload command called `Cprunload` to remove the object files from memory.

5

## Step 2: Exiting BASIC to HP-UX

To exit the BASIC environment, execute the following statement:

```
QUIT Return
```

The BASIC default window should disappear. The window that the `rmb` command was executed in will appear with your HP-UX shell prompt.

### Step 3: Writing C Subroutines

This step shows an example C program called `example.c` that contains two CSUB routines. There are two important things to remember during this step:

- You should heed the CSUB limitations pertaining to library usage since the routines may eventually be converted to actual BASIC CSUBs.
- You should verify that the routines only use formal parameters with types supported by the prototyper. The section “Parameter Passing” defines those types.

Here is a listing of the program `example.c`, which defines two simple C routines:

```
void sub1(i, j, str)
short i;
int *j;
char *str;
{
char *s="sub1";
*j=i;
strncpy(str, s, strlen(s)+1);
}

double sub2(r)
double r;
{
return r*2;
}
```

The first routine called `sub1` simply assigns the value of its first numeric parameter to its second numeric parameter, a variable parameter, and copies characters into its string parameter. The second routine called `sub2` is a function which doubles the value of its input parameter and returns the result. You should note that, unlike CSUBs, these routines can have both value and variable formal parameters.

After compiling and testing the routines in C, you are now ready to generate a relocatable object file so that you can call the two routines from BASIC, using the prototyper.

#### Step 4: Generating a Relocatable Object File

The process of generating a relocatable object file as input to the prototyper is the same as that used for creating a CSUB object file. In other words, you use the HP-UX `ld` command to link your object files with the required libraries. The syntax for this command is as follows:

```
ld -rd -a archive example.o -u _printf -lmb -o example
```

The above command generates the fully linked relocatable object file `example`, which will be used as input to the prototyper.

#### Step 5: Running the BASIC Program

Before running the BASIC program called `test_csubs`, enter the BASIC environment by typing:

```
rmb Return
```

Now load the program called `test_csubs` (found in `/usr/lib/rmb/demo`) by executing the following command:

```
GET "test_csubs" Return
```

Note that you may have to use the statement `MASS STORAGE IS` to move to the directory that contains `test_csubs`.

To run the program `test_csubs`, type:

```
RUN Return
```

or press **RUN** (**f3**).

---

## Deciding Whether or Not to Create a Standard CSUB

After verifying the execution of compiled routines using the CSUB prototyper, you have two options to finalize the implementation of your program:

1. You can invoke the compiled routines using the prototyper.
2. You can convert the prototyper calls into direct CSUB calls.

### Reasons for Choosing the First Option

It makes sense to invoke the compiled routines using the prototyper if:

- You are satisfied with the performance of the prototyper calls to the compiled routines.
- You are satisfied with the memory utilization of the prototyper for storing the object files.
- You need the ability to invoke compiled routines sporadically or in non time-critical situations.
- You have no prior knowledge of the names of the compiled routines.

### Reasons for Choosing the Second Option

By converting the prototyper calls into direct CSUB calls, you can optimize the access time required to load the compiled routines with a CSUB interface. This option is also useful if you want to STORE a BASIC program with all of its CSUBs to make it self contained. The task of providing a CSUB interface is most easily done by defining a procedural interface for each of the selected compiled routines, as shown below for the current example:

```
void csub1(i, j, dim, str)
short *i;
double *j;
dimentryptr dim;
bstring_parm str;

{
    int jj;

    jj>(*j);
    sub1(*i, &jj, str->c);
}
```

```

    str->len=strlen(str->c)+1;
    *j=jj;
}

double csub2(result, r)
double *result;
double *r;

{
    *result=(*r)*2;
}

```

The main concern during this conversion process is to provide a CSUB interface with BASIC parameters that are compatible with those of the formal parameters of the compiled routines. This requirement may necessitate some parameter processing, as shown above, before and after the call to the compiled routines.

---

## Passing Parameters

This section explains the parameter passing conventions of the prototyper and lists its parameter type mappings.

In order to correctly use the CSUB prototyper, it is necessary that you know its parameter passing conventions and the mappings that it allows between the types of BASIC parameters used to call compiled routines and the types of formal parameters and return values defined by the compiled routines.

### Parameter Passing Conventions

The prototyper *execute* CSUB and functions can accept up to 16 parameters which correspond to the formal parameters of a given compiled routine. The BASIC variables or constants which are specified as parameters to these *execute* CSUBs may necessitate some type conversions before being passed to a compiled routine. Thus, you need to follow these conventions:

- Always nest BASIC variables or constants in a call to one of the prototyper parameter functions that can be found in the table “Mapping Between Actual and Formal Parameters of a CSUB.”

- Specify whether you are passing by reference or by value when you call a parameter function. This is done by setting the value of the optional parameter of each parameter function to 1 if you want to pass by reference, or 0 if you want to pass by value (the default). Some examples are:

`FNI(J,1)`    *passes an integer by reference*

`FNI(J,0)`    *passes an integer by value*

Again, this specification allows the prototyper to perform any necessary type conversions on parameters after the call to the compiled routine.

- Choose the appropriate prototyper *execute* CSUB and functions to invoke a particular routine based on the return value type of this routine. For example, this C CSUB routine accepts an integer and does not have a return value:

```
void sub1(i)
short i;
```

Requires that you use the following *execute* CSUB:

```
Cpr(FNInit(A,"_sub1"), FNI(I,1))
```

This C CSUB routine returns an integer and does not have a parameter:

```
short sub2()
```

Requires that you use the following *execute* CSUB:

```
I=FNCpr16(FNInit(A,"_sub2"))
```



## Mapping of Parameter Types

This section presents the mapping between the actual and formal parameters of compiled routines and mapping between BASIC return types and prototyper *execute* CSUBs. The following tables present this information.

**Table 5-2.**  
**Mapping Between Actual and Formal Parameters of a CSUB**

CSUB Formal Parameter Type	Parameter Function	Variable Parameter Supported	BASIC Actual Parameter Type	Languages Supported
8-bit character	FNC	Yes	INTEGER	all
16-bit integer	FNI	Yes	INTEGER	all
32-bit integer	FNI	No	INTEGER	all
32-bit integer	FNL	Yes	REAL	all
32-bit floating-point	FNR	Yes	REAL	all
64-bit floating-point	FND	Yes	REAL	all
32-bit pointer	FNP	Yes	REAL	Pascal, C
string	FNS	Yes	string	all
aggregate	FNA	Yes	REAL	Pascal, C

5

You should note that the language of the routine can affect these specifications, and that some parameter types have restrictions on the use of variable parameters.

**Table 5-3.**  
**Mapping Between BASIC Return Value Types and**  
**Prototyper execute CSUBs or Functions**

Compiled Routine Return Value Type	Execute CSUB or Function	BASIC Return Value Type
void	Cpr	
8-bit character	FNCpr8	INTEGER
16-bit integer	FNCpr16	INTEGER
32-bit integer	FNCpr32l	REAL
32-bit floating-point	FNCpr32r	REAL
64-bit floating-point	FNCpr64	REAL
32-bit pointer	FNCpr32p	REAL

5

In addition to the information in these tables, each formal parameter and return value of a compiled routine may have some considerations that need to be discussed. These considerations are as follows:

- 8-bit character      The value of the INTEGER actual parameter should be within the range of a byte. The prototyper takes care of making the proper format conversions before and after the call to the compiled routine.
- 32-bit integer      The INTEGER actual parameter may not be a variable parameter since INTEGER variables are only 16 bits wide. In order to have a variable parameter, you should use a REAL variable with a value in the range of a 32-bit integer. The prototyper takes care of making the proper format conversions before and after the call to the compiled routine.
- 32-bit floating-point      The value of the REAL actual parameter should be within the range of a 32-bit floating-point return value. The prototyper takes care of making the proper format

conversions before and after the call to the compiled routine.

32-bit pointer

Because BASIC does not define 32-bit scalar variables, the prototyper stores pointer values in REAL variables. You should not attempt to print or use such variables in BASIC since their values are only meaningful when passed to compiled routines. The prototyper takes care of making the proper format conversions before and after the call to the compiled routine.

String

The maximum length of a string actual parameter should always be greater than the current length of the string by one because the prototyper converts the storage for the string to a format suitable for the language of the compiled routine. This means that a string literal may not be used as a parameter to the string parameter function. Furthermore, variable string parameters should have a maximum length large enough to accommodate any new string value set by the compiled routine.

Aggregate

An aggregate, a Pascal record or a C struct/union, is specified by passing the pointer to it and its storage size. The prototyper takes care of passing the correct information to the compiled routine. There are two ways you can create an aggregate in BASIC: use a dynamic memory allocation routine in one of the compiled languages, or get a pointer to a named BASIC COM whose definition mirrors that of the aggregate.

---

## Handling Prototyper Errors

Prototyper CSUBs signal a run-time error with a BASIC CSUB run-time error message. In order to retrieve the actual error number, you should call the error CSUB (Cprerr) provided with the CSUB prototyper library (CPR). This CSUB returns a positive error number for each error generated by one of the prototyper CSUBs. The error numbers for the prototyper CSUBs are listed in the following table.

**Table 5-4. Prototyper CSUB Errors**

<b>CSUB or Function</b>	<b>Error Number</b>	<b>Description</b>
Cpload	1	Out of memory
Cpload	2	Object file access error
Cpload	3	Relocation error
FNInit	1	Compiled routine not found in object file
FNS	1	Maximum string length too small

## Porting Pascal Workstation Assembly CSUBs

---

Assembly language CSUBs written on the Pascal Workstation can be ported to the HP-UX operating system for use as BASIC/UX CSUBs. This is accomplished by translating the Pascal Workstation assembly language source file using the HP-UX command called *atrans* and then making any necessary changes to the program. This chapter provides you with a process for doing this, however, it requires a good background in assembly language programming with the 68000 microprocessor.

---

### Prerequisites

As previously mentioned, you will need a good understanding of assembly language programming on a 68000 microprocessor in order to translate a Pascal Workstation assembly language program into one for use on HP-UX. Note that the MC68010 is not supported on BASIC/UX. The following are prerequisites for porting your Pascal Workstation assembly language program:

- You should have a known working assembly language CSUB (one created on the Pascal Workstation) that you can copy over to the HP-UX system.
- You should have a knowledge of how data is pushed onto and pulled off of the stack on both the Pascal Workstation and the HP-UX operating system. For information on this read:
  - The section “How Pascal Programs Use the Stack” in the chapter “The Pascal Compiler” found in the *Pascal Workstation Systems Manual, Vol. 1*.
  - The appendix “Interfacing Assembly Routines to Other Languages” found in the *HP-UX Assembler Reference Manual and ADB Tutorial*.

---

## Using atrans

This section explains how to translate your Pascal Workstation assembly language CSUB for use on the HP-UX system. Topics covered are as follows:

- Copying a Pascal Workstation CSUB
- Executing the atrans Command
- Modifying the Translated CSUB.

### Copying a Pascal Workstation CSUB

If you have an assembly language CSUB file called `paw_file` located on a Pascal Workstation disk, it can be copied over to your HP-UX system using the `COPY` command. For information on how to do this, refer to the *HP BASIC 6.2 Porting and Globalization* manual. However, to simplify things, the assembly language file named `paw_file` can be found in the HP-UX examples directory called `/usr/lib/rmb/demo`. You will be using this file to learn how `atrans` works. The following listing shows the contents of this file:

```
*          Define some handy mnemonics

front      equ    a0      Pointer to the front half of the string
back       equ    a1      Pointer to the back half of the string
result     equ    a2      Pointer to a 16 bit INTEGER
return     equ    a3      The return address

temp       equ    d0      Used for calculations and comparisons

*          Define the entry point

def        palindrome

*          Go for it.  First unload the stack

palindrome equ *
    movea.l (sp)+,return
    movea.l (sp)+,result
    movea.l (sp)+,front
    addq.l #4,sp          The dimentryptr is not used

*          Get the current string length and use this to set up
*          the front and back pointers.  Remember, a BASIC string is
```

```

*      a 16-bit integer followed by a packed array of characters.

      clr.l   temp
      move.w  (front),temp           Get string length
      addq.l  #2,front              The first character is always here
      lea    0(front,temp),back     Back points to last character+1

*      Use the pre-increment and post-decrement of the address
*      registers to compare characters and update pointers.

loop   move.b  -(back),temp         Get next "back" character;
      cmp.b   (front)+,temp        compare it to the "front";
      bne.s   false                quit if there is a mismatch

      cmpa   front,back           If "back" is still less than...
      blt    loop                 ...keep trying

true   move.w  #1,(result)         Set it true
      jmp    (return)

false  clr.w   (result)
      jmp    (return)

      end

```

## Executing the *atrans* Command.

The HP-UX command *atrans* is used to partially translate a Pascal Workstation assembly language program for use on the HP-UX system. The reason the translator partially translates the assembly program is there are symbol definition pseudo-ops and registers that cannot be used in your HP-UX assembly CSUB.

To test the *atrans* command on a Pascal Workstation assembly language CSUB, copy the file called *paw\_file* to your current working directory and execute the following command:

```
atrans paw_file > paw_file.s
```

This command redirects the output of *atrans* to the file called *paw\_file.s*.  
 The contents of your file should look like this:

```

#           Define some handy mnemonics

           set     front,%a0   #Pointer to the front half of the string
           set     back,%a1    #Pointer to the back half of the string
           set     result,%a2  #Pointer to a 16 bit INTEGER
           set     return,%a3  #The return address

           set     temp,%d0    #Used for calculations and comparisons

#           Define the entry point

           global  palindrome

#           Go for it.  First unload the stack

           set     palindrome,.
           movea.l (%sp)+,return
           movea.l (%sp)+,result
           movea.l (%sp)+,front
           addq.l  &4,%sp      #The dimentryptr is not used

#           Get the current string length and use this to set up
#           the front and back pointers.  Remember, a BASIC string is
#           a 16-bit integer followed by a packed array of characters.

           clr.l   temp
           move.w  (front),temp      #Get string length
           addq.l  &2,front          #The first character is always here
           lea    0(front,temp),back #Back points to last character+1

#           Use the pre-increment and post-decrement of the address
#           registers to compare characters and update pointers.

loop:     move.b  -(back),temp      #Get next "back" character;
           cmp.b  temp,(front)+    #compare it to the "front";
           bne.b  false            #quit if there is a mismatch

           cmpa   back,front        #If "back" is still less than...
           blt    loop              #...keep trying

true:     move.w  &1,(result)      #Set it true
           jmp    (return)

```



```

false:  clr.w   (result)
        jmp    (return)

#       end

```

## Modifying the Translated CSUB

Since *atrans* does not fully translate a Pascal Workstation assembly program to code usable on the HP-UX system, you will have to manually complete the translation of the assembly language program called *paw\_file.s*. This section covers the changes you need to make to the *paw\_file.s* file so you can create an object code file out of it. When your file is completely translated, it should look like this:

```

#       Define some handy mnemonics

#       Define the entry point

        global  _palindrome
_palindrome:

#       Go for it.  First unload the stack

        movea.l (%sp)+,%a3
        addq.l  &4,%sp          #The dimentryptr is not used
        movea.l (%sp)+,%a0
        movea.l (%sp)+,%a2

#       Get the current string length and use this to set up
#       the front (%a0) and back (%a1) pointers.  Remember, a BASIC string is
#       a 16-bit integer followed by a packed array of characters.

        clr.l   %d0
        move.w  (%a0),%d0        #Get string length
        addq.l  &2,%a0          #The first character is always here
        lea    0(%a0,%d0),%a1   #%a1 points to last character+1

#       Use the pre-increment and post-decrement of the address registers
#       to compare characters and update pointers.

loop:   move.b  -(%a1),%d0       #Get next "back" (%a1) character;
        cmp.b  %d0,(%a0)+      #compare it to the "front" (%a0);

```

```

        bne.b   false          #quit if there is a mismatch

        cmpa   %a1,%a0         #If "back" is still less than...
        blt    loop           #...keep trying

true:   move.w  &1,(%a2)       #Set it true
        jmp    (%a3)

false:  clr.w   (%a2)
        jmp    (%a3)

#       end

```

### What Was Changed?

Here are the changes that were made to the file called `paw_file.s`:

- The symbol definition pseudo-op `set` cannot be used in an HP-UX assembly CSUB, therefore, all `set` pseudo-ops that were used to equate registers with identifier names need to be removed. All occurrences of the identifier names should be replaced as follows:

```

front   replaced with %a0
back    replaced with %a1
result  replaced with %a2
return  replaced with %a3
temp    replaced with %d0

```

- Change the order in which the stack is unloaded. This can be done by exchanging lines 18 and 20 of the assembly language program that was translated using `atrans`. The table given below shows the lines before and after the exchange of lines.

**Table 6-1. Comparison of Program Segments**

Lines 18 and 20 Before Exchange.	Exchanged Lines.
<pre> movea.l (%sp),%a2 movea.l (%sp),%a0 addq.l  &amp;4,%sp </pre>	<pre> addq.l  &amp;4,%sp movea.l (%sp),%a0 movea.l (%sp),%a2 </pre>

- Prefix an underscore (“\_”) to the `global` pseudo-op’s identifier name. It should look like this:

```
global _palindrome
```

- Add a label called `_palindrome` just under the `global` pseudo-op. Your label should look like this:

```
global _palindrome
_palindrome:
```

Note that all of the above changes are the types of changes that you will need to make to your Pascal Workstation assembly language programs and CSUBs when you translate them using the HP-UX command *atrans*.

---

## Completing Your Assembly CSUB

To complete the creation of your assembly language CSUB and to test it, you need to complete these steps:

- Copy the Calling BASIC Program to HP-UX.
- Execute Steps 4 through 7 of the Pascal CSUB Procedure.

### Copy the Calling BASIC Program to HP-UX

You will want to use the original BASIC program that you created on the BASIC Workstation to test you assembly CSUB. To do this, use the `COPY` command. For information on how to use the `COPY` command to transfer files to the HP-UX operating system, refer to the *HP BASIC 6.2 Porting and Globalization* manual. To save you time, there is a file named `PAL_PROG` located in the HP-UX examples directory called (`/usr/lib/rmb/demo`). The contents of the file are as follows:

```

100  LOADSUB ALL FROM "PAL_DRO"
110  DIM Test$[80]
120  INTEGER Result
130  DISP "This is a test for palindromes."
140  WAIT 2
150  LINPUT "Enter a word and press [Return].",Test$
160  Palindrome(Test$,Result)
170  IF Result=1 THEN
180      PRINT "The word you entered is a palindrome."
190  ELSE
200      PRINT "The word you entered is not a palindrome."
210  END IF
220  DELSUB Palindrome
230  END

```

The above program when executed loads the PROG file called PAL\_DRO which contains the CSUB called Palindrome. It then asks you to enter a word that can be either a palindrome or a non-palindrome. A palindrome is a word or phrase that reads the same backward or forward. The word you type in is passed as a string to the palindrome CSUB. The CSUB then returns a 1 in the variable Result if the word you entered is a palindrome; otherwise, a 0 is returned. A message stating that the word you entered is either a palindrome or not a palindrome is then displayed.

6

## Execute Steps 4 through 7 of the Pascal CSUB Procedure

The remaining steps for completing your CSUB are the same as steps 4 through 7 for creating a Pascal CSUB. These steps can be found in the section "Steps for Creating a Pascal CSUB" found in the chapter "Writing Pascal CSUBs." Note that the stream file that you created on the Pascal Workstation, for the original CSUB, can be copied over to the HP-UX system. This file, with minor modifications, can then be used with the HP-UX *rmbuildc* command to create the PROG file that your BASIC/UX program will call. To use this stream file, you would type a command similar to the following:

```
rmbuildc < paw_stream_file
```

where *paw\_stream\_file* is the modified Pascal Workstation stream file.

# A

## File Access Reference

---

This appendix is a reference that describes each routine in the file access library. There is a one-to-one functionality between `fal` routines and corresponding BASIC statements. For example, `FAL_CREATE_BDAT` creates a BDAT file from the CSUB the same way the BASIC `CREATE BDAT` statement creates a BDAT file.

The routines described in this appendix are implemented in HP Pascal for the Pascal Workstation and HP-UX. If you wish to call them from another language, you will need to determine the parameter types in that language which match the types of the formal parameters of the routines. Note that in HP Pascal an `idtable` is a packed array of 190 bytes. Therefore, in this appendix where it mentions an `idtable` is 148 bytes, it is referring to an `idtable` for a Pascal Workstation.

A

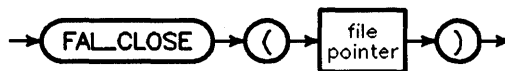
---

## FAL\_CLOSE

IMPORT: CSFA

This procedure closes a file. Files must be explicitly closed since the system does not know about `fal` files and cannot close them automatically at subroutine exit and stop.

### Syntax



Item	Description	Range
file pointer	pointer of TYPE <code>fc_b_ptr_type</code> (pointer to <code>idtable</code> )	—

### Example Procedure Calls

```
FAL_CLOSE(idptr)
```

```
FAL_CLOSE(Ptr)
```

A

### Semantics

The “file pointer” (`idptr`) parameter is a pointer to an `idtable`.

You must allocate an `idtable` for use by all routines that have an `idptr` as a parameter. An `idtable` is a file control block. From the user’s point of view, an `idtable` can be a packed array of 190 bytes. When a file is opened, file control information is written into the `idtable`. When a file is accessed, the `idtable` is consulted for information regarding file location, type, etc. You should never change any data in the `idtable` except by `FAL` procedures.

An `idtable` uniquely identifies a file. The same `idtable` must be used to access the file once it is open. Several files can be open at the same time; each

must have it's own idtable. Access to one file with two different idtables at the same time is not supported.

**A** 

## FAL\_CLOSE

---

## FAL\_CREATE

IMPORT: CSFA

This procedure creates an HP-UX file.

### Syntax



Item	Description	Range
file specifier	string expression of TYPE STRING[160] (file_name_type)	any valid file specifier
number of bytes	numeric expression of TYPE INTEGER	1 thru $2^{31} - 1$

### Example Procedure Calls

```
FAL_CREATE(File_spec,N_bytes)
```

```
FAL_CREATE("NewFile:,700",128)
```

### Semantics

The "file specifier" (`File_spec`) parameter may include a directory specifier (if the file is to be created on a volume that supports hierarchical directories), and a mass storage unit specifier (`msus`).

The "number of bytes" (`N_bytes`) parameter specifies how many bytes are to be allocated to the file.



## FAL\_CREATE\_ASCII

IMPORT: CSFA

This procedure creates an ASCII file.

### Syntax



Item	Description	Range
file specifier	string expression of TYPE STRING[160] (file_name_type)	any valid file specifier
number of records (256 bytes)	numeric expression of TYPE INTEGER	1 thru $2^{31} - 1$

### Example Procedure Calls

```
FAL_CREATE_ASCII(File_spec,N_recs)
```

```
FAL_CREATE_ASCII("NewFile:,700",128)
```

### Semantics

The “file specifier” (**File\_spec**) parameter may include a directory specifier (if the file is to be created on a volume that supports hierarchical directories), and a mass storage unit specifier (msus).

The “number of records” (**N\_recs**) parameter specifies how many records are to be allocated to the file. For ASCII files, one record is 256 bytes.

A

## FAL\_CREATE\_ASCII

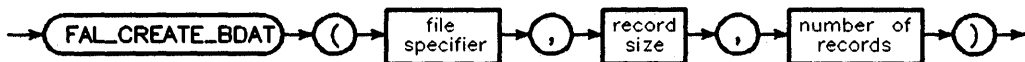
---

## FAL\_CREATE\_BDAT

IMPORT: CSFA

This procedure creates a BDAT file.

### Syntax



Item	Description	Range
file specifier	string expression of TYPE STRING[160] (file_name_type)	any valid file specifier
record size	numeric expression of TYPE INTEGER	1 thru $2^{31} - 1$
number of records	numeric expression of TYPE INTEGER	1 thru $2^{31} - 1$

### Example Procedure Calls

```
FAL_CREATE_BDAT(File_spec,Rec_size,N_recs)
```

```
FAL_CREATE_BDAT("NewFile:",700",20,128)
```

### Semantics

The "file specifier" (**File\_spec**) parameter may include a directory specifier (if the file is to be created on a volume that supports hierarchical directories), and a mass storage unit specifier (**msus**).

The "record size" (**Rec\_size**) parameter specifies the size of **logical records** to be used with the file. If the BDAT file is *not* to be used with **random access** operations, then use a record size of 256 (default value in BASIC).

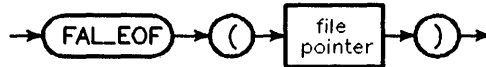
The "number of logical records" (**N\_recs**) parameter specifies how many logical records are to be allocated to the file.

## FAL\_EOF

IMPORT: CSFA

This procedure writes an end of file at the current file position. The file must be open.

### Syntax



Item	Description	Range
file pointer	pointer of TYPE <code>fc_b_ptr_type</code> (pointer to <code>idtable</code> )	—

### Example Procedure Calls

```
FAL_EOF(idptr)
```

```
FAL_EOF(Ptr)
```

### Semantics

The “file pointer” (`idptr`) parameter is a pointer to an `idtable`.

You must allocate an `idtable` for use by all routines that have an `idptr` as a parameter. An `idtable` is a file control block. From the user’s point of view, an `idtable` can be a packed array of 190 bytes. When a file is opened, file control information is written into the `idtable`. When a file is accessed, the `idtable` is consulted for information regarding file location, type, etc. You should never change any data in the `idtable` except by FAL procedures.

An `idtable` uniquely identifies a file. The same `idtable` must be used to access the file once it is open. Several files can be open at the same time; each must have its own `idtable`. Access to one file with two different `idtables` at the same time is not supported.

## **FAL\_EOF**

In a BDAT file, this updates the end of data pointer. In an ASCII file, it writes a -1 at the current location. If the user has been using **FAL\_READ** and **FAL\_WRITE**, he takes responsibility for actually being at the end of a record when he writes the -1 (else it will not be seen as an end-of-data).

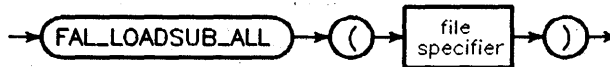
**A**

## FAL\_LOADSUB\_ALL

IMPORT: CSFA

This procedure loads all subprograms from a specified PROG file and appends them to the program in memory.

### Syntax



Item	Description	Range
file specifier	string expression of TYPE STRING[160] (file_name_type)	any valid file specifier

### Example Procedure Calls

```
FAL_LOADSUB_ALL(File_spec)
```

```
FAL_LOADSUB_ALL("NewFile:,700")
```

### Semantics

The “file specifier” (**File\_spec**) parameter may include a directory specifier (if the file is to be created on a volume that supports hierarchical directories), and a mass storage unit specifier (msus).

A

## FAL\_LOADSUB\_ALL

---

## FAL\_LOADSUB\_NAME

IMPORT: CSFA

This procedure loads the specified subprogram from the specified PROG file and appends it to the program in memory.

### Syntax



Item	Description	Range
file specifier	string expression of TYPE STRING[160] (file_name_type)	any valid file specifier
subprogram name	string expression of TYPE STRING[160]	any valid subprogram name

### Example Procedure Calls

```
FAL_LOADSUB_NAME(File_spec,Sub_name)
```

```
FAL_LOADSUB_NAME("NewFile:,700","Test1")
```

A

### Semantics

The “file specifier” (**File\_spec**) parameter may include a directory specifier (if the file is to be created on a volume that supports hierarchical directories), and a mass storage unit specifier (**msus**).

The “subprogram name” (**Sub\_name**) is the name of the subprogram to be loaded. It must appear exactly as it would in the program.

## FAL\_OPEN

IMPORT: CSFA

This procedure opens a file for reading and writing. It can be an ASCII, BDAT or HP-UX file.

### Syntax



Item	Description	Range
file specifier	string expression of TYPE STRING[160] (file_name_type)	any valid file specifier
file pointer	pointer of TYPE fcb_ptr_type (pointer to idtable)	—

### Example Procedure Calls

```
FAL_OPEN(File_spec, idptr)
```

```
FAL_OPEN("NewFile:,700", ptr)
```

### Semantics

The “file specifier” (**File\_spec**) parameter may include a directory specifier (if the file is to be created on a volume that supports hierarchical directories), and a mass storage unit specifier (msus).

The “file pointer” (**idptr**) parameter is a pointer to an **idtable**.

You must allocate an **idtable** for use by all routines that have an **idptr** as a parameter. An **idtable** is a file control block. From the user’s point of view, an **idtable** can be a packed array of 190 bytes. When a file is opened, file control information is written into the **idtable**. When a file is accessed, the **idtable** is consulted for information regarding file location, type, etc.

A

## FAL\_OPEN

An `idtable` uniquely identifies a file. The same `idtable` must be used to access the file once it is open. Several files can be open at the same time; each must have its own `idtable`. Access to one file with two different `idtables` at the same time is not supported.

---

### Note



As part of opening a file, the system will close any I/O path which is open using the same `idtable`. Since Pascal variables are not cleared when they are allocated, your `idtable` may initially contain data which will appear to the system to be an open I/O path. Errors and incorrect behavior may occur if the system tries to close an I/O path based on this random data. To prevent this, you must zero the `idtable` before you use it the first time in `FAL_OPEN`. Once you are using the `idtable`, never change any data in it except by using `FAL` procedures.

---

Files must be explicitly closed using `FAL_CLOSE`. They will not be closed at subroutine exit or stop.

`BDAT` and `HP-UX` files are opened in `FORMAT OFF`.



## FAL\_POSITION

IMPORT: CSFA

This procedure positions the file to a specified logical record number. The file must be open and must be a BDAT or HP-UX file.

### Syntax



Item	Description	Range
file pointer	pointer of TYPE <code>fc_ptr_type</code> (pointer to <code>idtable</code> )	—
logical record	numeric expression of TYPE <code>INTEGER</code>	1 thru $2^{31} - 1$

### Example Procedure Calls

```
FAL_POSITION(idptr,logrecno)
```

```
FAL_POSITION(Ptr, 1)
```

### Semantics

The “file pointer” (`idptr`) parameter is a pointer to an `idtable`.

You must allocate an `idtable` for use by all routines that have an `idptr` as a parameter. An `idtable` is a file control block. From the user’s point of view, an `idtable` can be a packed array of 190 bytes. When a file is opened, file control information is written into the `idtable`. When a file is accessed, the `idtable` is consulted for information regarding file location, type, etc. You should never change any data in the `idtable` except by FAL procedures.

An `idtable` uniquely identifies a file. The same `idtable` must be used to access the file once it is open. Several files can be open at the same time; each

A

## **FAL\_POSITION**

must have it's own `idtable`. Access to one file with two different `idtables` at the same time is not supported.

The `logrecno` (**logical record**) is the logical record for BDAT files or byte number for HP-UX files at which the file pointer is to be positioned. The beginning of a file is at logical record number 1.

**A**

## FAL\_PURGE

IMPORT: CSFA

This procedure purges a file. The file must be closed.

### Syntax



Item	Description	Range
file specifier	string expression of TYPE STRING[160] (file_name_type)	any valid file specifier

### Example Procedure Calls

```
FAL_PURGE(File_spec)
```

```
FAL_PURGE("NewFile:,700")
```

### Semantics

The “file specifier” (`File_spec`) parameter may include a directory specifier (if the file is to be created on a volume that supports hierarchical directories), and a mass storage unit specifier (`msus`).

FAL\_PURGE will detect attempts to purge files which are opened in BASIC (i.e., by assigning an *I/O path name* (`@name`) to the file). It will not detect attempts to purge files which are opened in the CSUB (i.e., by assigning a *file control block* (equivalent to the value area for an I/O path) to the file). This is also true for the BASIC PURGE statement. A method for avoiding this problem is given in the “Advanced Topics” chapter.

A

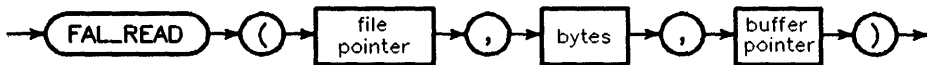
## FAL\_PURGE

## FAL\_READ

IMPORT: CSFA

This procedure reads a file. The file must be open. It can be an ASCII, BDAT or HP-UX file.

### Syntax



Item	Description	Range
file pointer	pointer of TYPE <code>fc_b_ptr_type</code> (pointer to <code>idtable</code> )	—
bytes	numeric expression, of TYPE <code>INTEGER</code>	0 to $2^{31} - 1$
buffer pointer	pointer to a buffer ( <code>anyptr</code> )	—

### Example Procedure Calls

```
FAL_READ(idptr,nbytes,bufptr)
```

```
FAL_READ(Ptr,70,Tobuf)
```

### Semantics

The “file pointer” (`idptr`) parameter is a pointer to an `idtable`.

You must allocate an `idtable` for use by all routines that have an `idptr` as a parameter. An `idtable` is a file control block. From the user’s point of view, an `idtable` can be a packed array of 190 bytes. When a file is opened, file control information is written into the `idtable`. When a file is accessed, the `idtable` is consulted for information regarding file location, type, etc. You should never change any data in the `idtable` except by `FAL` procedures.

An **idtable** uniquely identifies a file. The same **idtable** must be used to access the file once it is open. Several files can be open at the same time; each must have its own **idtable**. Access to one file with two different **idtables** at the same time is not supported.

The “bytes” (**nbytes**) parameter defines the number of bytes to read.

The “buffer pointer” (**bufptr**) is a pointer to the buffer.

This procedure provides direct access to the data bytes in the file. Therefore, the user must keep track of the record structure in a ASCII file. (Each item (logical record) in an ASCII file is preceded by a two-byte length header and contains an even number of bytes. If necessary a null byte, **CHR\$(0)**, is added to the item to make an even number of bytes in each record.)

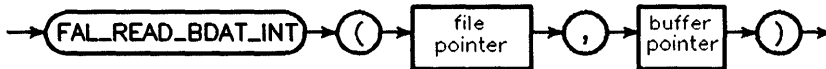
## FAL\_READ

### FAL\_READ\_BDAT\_INT

IMPORT: CSFA

This procedure reads a BASIC 16-bit integer from a BDAT file. This file must be open and must be a BDAT file.

#### Syntax



Item	Description	Range
file pointer	pointer of TYPE <code>fc_b_ptr_type</code> (pointer to <code>idtable</code> )	—
buffer pointer	pointer to a 16-bit <code>INTEGER</code> ( <code>anyptr</code> )	—

#### Example Procedure Calls

```
FAL_READ_BDAT_INT(idptr, intbufptr)
```

```
FAL_READ_BDAT_INT(Ptr, Froms)
```

A

#### Semantics

The “file pointer” (`idptr`) parameter is a pointer to an `idtable`.

You must allocate an `idtable` for use by all routines that have an `idptr` as a parameter. An `idtable` is a file control block. From the user’s point of view, an `idtable` can be a packed array of 190 bytes. When a file is opened, file control information is written into the `idtable`. When a file is accessed, the `idtable` is consulted for information regarding file location, type, etc. You should never change any data in the `idtable` except by `FAL` procedures.

An `idtable` uniquely identifies a file. The same `idtable` must be used to access the file once it is open. Several files can be open at the same time; each

## **FAL\_READ\_BDAT\_INT**

must have it's own idtable. Access to one file with two different idtables at the same time is not supported.

The intbufptr (buffer pointer) is a pointer to a 16-bit integer.

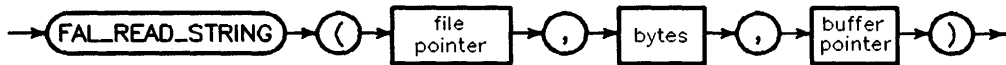
**A**

## FAL\_READ\_STRING

IMPORT: CSFA

This procedure reads a string from an ASCII, BDAT or HP-UX file. This file must be open.

### Syntax



Item	Description	Range
file pointer	pointer of TYPE <code>fcg_ptr_type</code> (pointer to <code>idtable</code> )	—
bytes	numeric expression, rounded to an integer	0 to $2^{31} - 1$
buffer pointer	pointer to a buffer ( <code>anyptr</code> )	—

### Example Procedure Calls

```
FAL_READ_STRING(idptr,nbytes,bufptr)
```

```
A FAL_READ_STRING(Ptr,70,Tobuf)
```

### Semantics

The “file pointer” (`idptr`) parameter is a pointer to an `idtable`.

You must allocate an `idtable` for use by all routines that have an `idptr` as a parameter. An `idtable` is a file control block. From the user’s point of view, an `idtable` can be a packed array of 190 bytes. When a file is opened, file control information is written into the `idtable`. When a file is accessed, the `idtable` is consulted for information regarding file location, type, etc. You should never change any data in the `idtable` except by FAL procedures.



## FAL\_READ\_STRING

An **idtable** uniquely identifies a file. The same **idtable** must be used to access the file once it is open. Several files can be open at the same time; each must have its own **idtable**. Access to one file with two different **idtables** at the same time is not supported.

The “bytes” (**nbytes**) parameter defines the maximum number of bytes to read.

The “buffer pointer” (**bufptr**) is a pointer to the string buffer. The length of the string in the file is determined from the file itself; the next 4-bytes for a BDAT file and the next 2-bytes for an ASCII file. If the length is odd, then it is increased by 1. If the result of that increase is greater than the **nbytes** specified, an escape is generated. Otherwise, the next length bytes are read into the buffer. Note that, if you specify **nbytes** greater than the buffer size you may write over system information.

In an HP-UX file, a string is terminated by a null character. There is no length field at the beginning of the string.

A

## FAL\_READ\_STRING

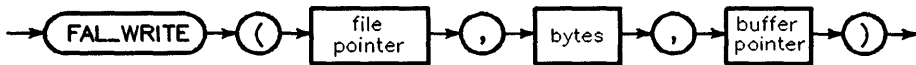
---

## FAL\_WRITE

IMPORT: CSFA

This procedure writes a file. The file must be open. It can be an ASCII, BDAT or HP-UX file.

### Syntax



Item	Description	Range
file pointer	pointer of TYPE <code>fc_b_ptr_type</code> (pointer to <code>idtable</code> )	—
bytes	numeric expression, rounded to an integer	0 to $2^{31} - 1$
buffer pointer	pointer to a buffer ( <code>anyptr</code> )	—

### Example Procedure Calls

```
FAL_WRITE(idptr,nbytes,bufptr)
```

```
FAL_WRITE(Ptr,70,Tobuf)
```

### Semantics

The “file pointer” (`idptr`) parameter is a pointer to an `idtable`.

You must allocate an `idtable` for use by all routines that have an `idptr` as a parameter. An `idtable` is a file control block. From the user’s point of view, an `idtable` can be a packed array of 190 bytes. When a file is opened, file control information is written into the `idtable`. When a file is accessed, the `idtable` is consulted for information regarding file location, type, etc. You should never change any data in the `idtable` except by `FAL` procedures.

An **idtable** uniquely identifies a file. The same **idtable** must be used to access the file once it is open. Several files can be open at the same time; each must have its own **idtable**. Access to one file with two different **idtables** at the same time is not supported.

The “bytes” (**nbytes**) parameter defines the number of bytes to write.

The “buffer pointer” (**bufptr**) is a pointer to the buffer.

This procedure provides direct access to the data bytes in the file. Therefore, the user must keep track of the record structure in a ASCII file. (Each item (logical record) in an ASCII file is preceded by a two-byte length header and contains an even number of bytes. If necessary a null byte, **CHR\$(0)**, is added to the item to make an even number of bytes in each record.)

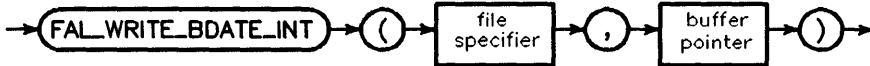
## FAL\_WRITE

### FAL\_WRITE\_BDAT\_INT

IMPORT: CSFA

This procedure writes a BASIC 16-bit integer to a BDAT file. This file must be open and must be a BDAT file.

#### Syntax



Item	Description	Range
file pointer	pointer of TYPE <code>fcg_ptr_type</code> (pointer to <code>idtable</code> )	—
buffer pointer	pointer to a 16-bit <code>INTEGER</code> ( <code>anyptr</code> )	—

#### Example Procedure Calls

```
FAL_WRITE_BDAT_INT(idptr,intbufptr)
```

```
FAL_WRITE_BDAT_INT(Ptr,Tosix)
```

#### Semantics

The “file pointer” (`idptr`) parameter is a pointer to an `idtable`.

You must allocate an `idtable` for use by all routines that have an `idptr` as a parameter. An `idtable` is a file control block. From the user’s point of view, an `idtable` can be a packed array of 190 bytes. When a file is opened, file control information is written into the `idtable`. When a file is accessed, the `idtable` is consulted for information regarding file location, type, etc. You should never change any data in the `idtable` except by `FAL` procedures.

An `idtable` uniquely identifies a file. The same `idtable` must be used to access the file once it is open. Several files can be open at the same time; each

## FAL\_WRITE\_BDAT\_INT

must have it's own `idtable`. Access to one file with two different `idtables` at the same time is not supported.

The `intbufptr` (`buffer pointer`) is a pointer to a 16-bit integer.

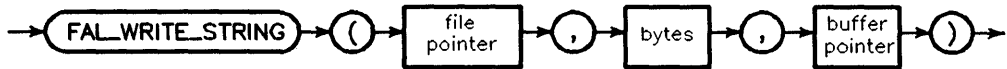
A

## FAL\_WRITE\_STRING

IMPORT: CSFA

This procedure writes a string to an ASCII, BDAT or HP-UX file. This file must be open.

### Syntax



Item	Description	Range
file pointer	pointer of TYPE <code>fc_b_ptr_type</code> (pointer to <code>idtable</code> )	—
bytes	numeric expression, rounded to an integer	0 to $2^{31} - 1$
buffer pointer	pointer to a buffer ( <code>anyptr</code> )	—

### Example Procedure Calls

```
FAL_WRITE_STRING(idptr,nbytes,bufptr)
```

```
FAL_WRITE_STRING(Ptr,70,Tobuf)
```

### Semantics

The “file pointer” (`idptr`) parameter is a pointer to an `idtable`.

You must allocate an `idtable` for use by all routines that have an `idptr` as a parameter. An `idtable` is a file control block. From the user’s point of view, an `idtable` can be a packed array of 190 bytes. When a file is opened, file control information is written into the `idtable`. When a file is accessed, the `idtable` is consulted for information regarding file location, type, etc. You should never change any data in the `idtable` except by FAL procedures.

## FAL\_WRITE\_STRING

An **idtable** uniquely identifies a file. The same **idtable** must be used to access the file once it is open. Several files can be open at the same time; each must have its own **idtable**. Access to one file with two different **idtables** at the same time is not supported.

The “bytes” (**nbytes**) parameter defines the number of bytes to write. When you write to a BDAT file a 4-byte binary length is written followed by the string, padded to the next even byte if necessary. When you write to an ASCII file a 2-byte binary length is written followed by the string, padded to the next even byte if necessary.

The “buffer pointer” (**bufptr**) is a pointer to the string buffer.

In an HP-UX file, a string is terminated by a null character. There is no length field at the beginning of the string.

A





# B

## Keyboard and CRT I/O Reference

---

This appendix is a reference that describes in detail the routines which provide access to the BASIC keyboard and CRT.

The CRT I/O routines access the CRT drivers at a fairly low-level. For example, the scrolling routines and `disp_at_xy` are below the high-level `PRINT` driver. Therefore, system variables defining the top and bottom of current text, and current print position are not updated by these routines. This leads to what appears to be abnormal behavior in arrow key scrolling and current print position. Consequently, it is recommended these routines not be used when mixing CSUB displays with normal BASIC displays.

The routines described in this appendix are implemented in HP Pascal for the Pascal Workstation and HP-UX. If you wish to call them from another language, you will need to determine the parameter types in that language which match the types of the formal parameters of the routines.

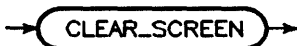
---

## CLEAR\_SCREEN

IMPORT: KBD CRT

This procedure clears the alpha CRT, which is exactly the same as the **Clear display** key on the keyboard, or the BASIC CLEAR SCREEN statement.

### Syntax



### Example Procedure Calls

```
CLEAR_SCREEN
```

```
IF finished THEN CLEAR_SCREEN
```

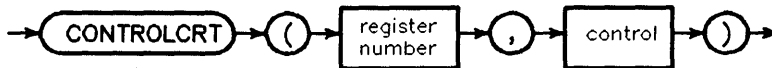
**B**

# CONTROLCRT

IMPORT: KBCRT

This procedure sends information to a CRT control register

## Syntax



Item	Description	Range
register number	numeric expression of TYPE <b>cbyte</b>	0 to 21 except for 3,6,7,9, and 19.
control	numeric expression of TYPE <b>cword</b>	depends on the register

## Example Procedure Calls

CONTROLCRT(**reg**, **cont**)

CONTROLCRT(1,10)

## Semantics

The “register number” (**reg**) parameter designates the register to be written.

The “control” (**cont**) contains the value to be written into the CRT control register. Refer to the “CRT Status and Control Registers” section in the *HP BASIC Language Reference* for more information.

This procedure will escape with a BASIC errorcode if an illegal argument is given. The value will be such that **errorcode MOD 1000 = 401** is true.

**B**

## CONTROLCRT

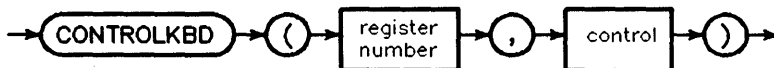
---

## CONTROLKBD

IMPORT: KBCRT

This procedure sends information to a keyboard control register

### Syntax



Item	Description	Range
register number	numeric expression of TYPE <b>cbyte</b>	0 to 17 except for 5,6,8,9 and 10
control	numeric expression of TYPE <b>cword</b>	depends on register

### Example Procedure Calls

```
CONTROLKBD(reg, cont)
```

```
CONTROLKBD(2,0)
```

### Semantics

The “register number” (**reg**) parameter designates the register to be written.

The “control” (**cont**) contains the value to be written into the keyboard control register. Refer to the “Keyboard Status and Control Registers” section in the *HP BASIC Language Reference* for more information.

This procedure will escape with a BASIC errorcode if an illegal argument is given. The value will be such that **errorcode MOD 1000 = 401** is true.

B

# CRTREADCHAR

IMPORT: KBD CRT

This function reads one character from the specified location on the CRT.

## Syntax



Item	Description	Range
x_coordinate	numeric expression of TYPE cbyte	1 to screen width
y_coordinate word	numeric expression of TYPE cbyte	1 to screen height

## Function Heading

```
FUNCTION CRTREADCHAR (X,Y:CBYTE): CHAR;
```

## Example Function Calls

```
One_chr := CRTREADCHAR(xcrd,ycrd)
```

```
IF CRTREADCHAR(24,22) = 'A' THEN foundit := TRUE
```

## Semantics

The “x\_coordinate” (xcrd) and “y\_coordinate” (ycrd) specify the on-screen alpha location from the upper left corner of the alpha CRT area (affected by ALPHA HEIGHT). Invalid values of x,y result in an escape with escapecode set to 1019.

**B**

## CRTREADCHAR

---

# CRTSCROLL

IMPORT: KBD CRT

This procedure scrolls the CRT area from line *first* to line *last* up or down one line.

### Syntax



Item	Description	Range
first line	numeric expression of TYPE <code>cbyte</code>	1 thru screen height
last line	numeric expression of TYPE <code>cbyte</code>	1 thru screen height
boolean	expression of TYPE <code>boolean</code>	true or false

### Example Procedure Calls

```
CRTSCROLL(first, last, dn)
```

```
CRTSCROLL(12, 22, tf)
```

### Semantics

The “first line” (`first`) parameter signifies the *first* line.

The “last line” (`last`) parameter signifies the *last* line. Lines *first* and *last* are relative to the current alpha screen (i.e., modified by `ALPHA HEIGHT`). Tests are made for  $0 < \text{first line} < \text{last line} \leq \text{screen height}$ , if they fail, an `escape(1019)` occurs.

These routines do *not* scroll text to/from the offscreen alpha buffer (if any) which the system uses. Instead, they clear the first or last line of the scrolling region.

## **CRTSCROLL**

The “down” (**dn**) parameter signifies whether to scroll up or down one line. That is, if down (**dn**) is false, it scrolls this area up one line.

**B**

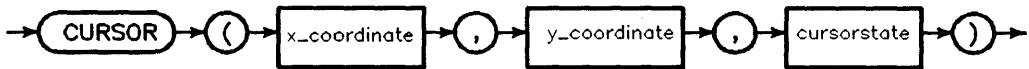
## CRTSCROLL

### CURSOR

IMPORT: KBD CRT

This procedure removes the previous cursor and writes a new cursor to any on-screen alpha location.

#### Syntax



Item	Description	Range
x_coordinate	numeric expression of TYPE <b>cbyte</b>	1 thru screen width
y_coordinate	numeric expression of TYPE <b>cbyte</b>	1 thru screen height
cursorstate	expression of TYPE <b>cursor</b>	<b>normal_cursor</b> , <b>insert_cursor</b> , or <b>off_cursor</b>

#### Example Procedure Calls

```
CURSOR(xcrd,ycrd,ctype)
```

```
CURSOR(24,22,normalcursor)
```

#### Semantics

The “x\_coordinate” (xcrd) and “y\_coordinate” (ycrd) specify the on-screen alpha location of the cursor from the upper left corner of the alpha CRT area (affected by ALPHA HEIGHT).

Any system action which moves the cursor will overwrite the action of this procedure. This includes any key press if not trapped by ON KBD. The programmer must take this into account in using this feature.



## CURSOR

The possible values of "cursorstate" (ctype) are `normal_cursor`, `insert_cursor`, and `off_cursor`.

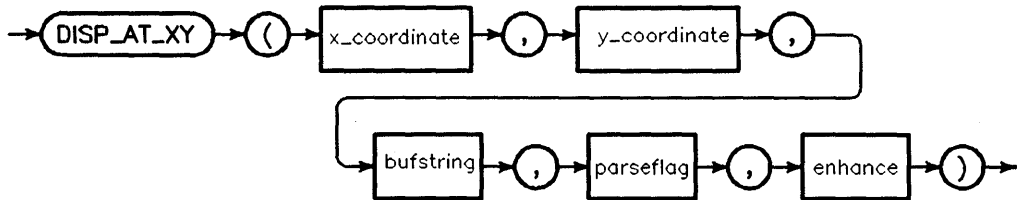
## CURSOR

### DISP\_AT\_XY

IMPORT: KBD CRT

This procedure writes text to any on-screen alpha location.

### Syntax



Item	Description	Range
<code>x_coordinate</code>	numeric expression of TYPE <code>cbyte</code>	1 thru screen width
<code>y_coordinate</code>	numeric expression of TYPE <code>cbyte</code>	1 thru screen height
<code>bufstring</code>	expression of TYPE <code>kbd_string = packed record</code> <code>  len: word;</code> <code>  c: packed array [1..256] of char;</code> <code>end;</code>	within range of <code>kbd_string</code>
<code>parseflag</code>	numeric expression of TYPE <code>boolean</code>	true or false
<code>enhance</code>	expression of TYPE <code>enh_type = packed record</code> <code>  color : cbyte;</code> <code>  hilite : cbyte;</code> <code>end;</code>	within range of <code>enhtype</code>

B

## Example Procedure Calls

```
DISP_AT_XY(xcrd, ycrd, var_str, parse, var_enhc)
```

```
DISP_AT_XY(22, 34, strrecord, tf, enhcrecord)
```

## Semantics

`disp_at_xy` can be used to put characters anywhere on the screen including the areas which are normally the keyboard lines, key labels, etc. This does not put those characters into the keyboard buffer, or any other internal buffer related to those areas. Specifically, it does *not* do an `OUTPUT KBD`. They will, however, be re-displayed by a `GCLEAR` on a bit-mapped display.

The “x\_coordinate” (`xcrd`) and “y\_coordinate” (`ycrd`) specify the on-screen alpha location of the cursor from the upper left corner of the alpha CRT area (affected by `ALPHA HEIGHT`). Characters will be wrapped at the alpha CRT width (not necessarily the `PRINTER IS;WIDTH`). A check is made for strings which would run off the end of the display. Out of range values of x and y are mapped to 1,1 and strings that are too long are truncated.

The “parseflag” (`parse`) parameter indicates whether characters in the range 128..143 are to be interpreted as highlights or displayed as characters. It also affects the display of characters 144..159; characters 0..31 are always displayed.

The “bufstring” (`var_str`) parameter contains the text to be written to the screen.

The “enhance” (`var_enhc`) parameter specifies the color and highlights for the text. It should be initialized to an appropriate value. However, internal color values for `ALPHA PEN 1..7` may be different from the user specified value. For `ALPHA PENS` above 7 the value of color is the pen number. The highlight values are also encoded. The simplest way to initialize `enhc` for highlights and for colors less than 8 is to set `parseflag` to true and print a string with characters in the range 128..143. Then the correct values will be returned in the `var` parameter `var_enhc`.

For monochrome displays, the color byte is ignored.

## READ\_KBD

IMPORT:     KBD CRT

This function returns the buffer contents trapped and held by ON KBD (same as KBD\$).

### Syntax



### Example Function Calls

READ\_KBD

### Semantics

You must execute an ON KBD statement before calling the CSUB. Then any calls to the function read\_kbd will return the same data as a call to the BASIC function KBD\$ would. Calling this function also clears the KBD\$ buffer. Use of this function allows you to poll the keyboard for input while in the CSUB. For the best results, include the ,ALL option in the ON KBD statement. Otherwise, a closure key, such as a typing-aid, causes the keyboard to block until an end-of-line occurs. This will not happen in the CSUB.

---

### Note



Closure keys are system-function keys (as opposed to ASCII-character keys) which are logged and subsequently processed by the BASIC system and therefore, normally “close” subsequent keyboard inputs to the system until fully processed. For further information, see the “Keyboard” chapter of the *HP BASIC 6.2 Interface Reference* manual.

---

B

---

## **SCROLLDN**

IMPORT: KBCRT

This procedure scrolls the PRINT area of the CRT down one line.

### **Syntax**



### **Example Procedure Calls**

SCROLLDN

**SCROLLDN**

---

## **SCROLLUP**

IMPORT: KBD CRT

This procedure scrolls the PRINT area of the CRT up one line.

### **Syntax**



### **Example Procedure Calls**

SCROLLUP

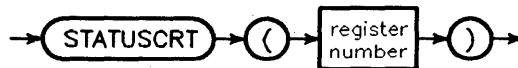
**B**

## STATUSCRT

IMPORT: KBCRT

This function returns the contents of a CRT status register.

### Syntax



Item	Description	Range
register number	numeric expression of TYPE cbyte	0 thru 21

### Function Heading

```
FUNCTION STATUSCRT (reg:cbyte) : cword;
```

### Example Function Calls

```
St_reg := STATUSCRT(reg_no);
```

```
IF STATUSCRT(13) = 9 THEN CONTROLCRT (13,18);
```

### Semantics

The “register number” (`reg_no`) parameter designates the register to be read. Refer to the “CRT Status and Control Registers” section in the *HP BASIC Language Reference* for more information.

This function will escape with a BASIC errorcode if an illegal argument is given. The value will be such that `errorcode MOD 1000 = 401` is true.

**B**

## STATUSCRT

---

## STATUSKBD

IMPORT: KBD CRT

This function returns the contents of a keyboard status register.

### Syntax



Item	Description	Range
register number	numeric expression of TYPE <b>cbyte</b>	0 thru 17 except for 3 and 4

### Function Heading

```
FUNCTION STATUSKBD (reg:cbyte) : cword;
```

### Example Function Calls

```
Str_reg := STATUSKBD(reg_no);
```

```
Fn_key := STATUSKBD(2);
```

### Semantics

The “register number” (**reg\_no**) parameter designates the register to be read. Refer to the “Keyboard Status and Control Registers” section in the *HP BASIC Language Reference* for more information.

This function will escape with a BASIC errorcode if an illegal argument is given. The value will be such that **errorcode MOD 1000 = 401** is true.

**B**

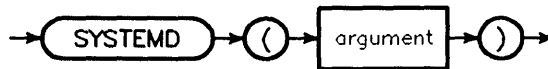


## SYSTEMD

IMPORT: KBD CRT

This function returns a string containing the same results (system status and configuration information) as executing `SYSTEM$` with the given argument.

### Syntax



Item	Description	Range
argument	string expression of TYPE STRING[18] (string_18)	any valid SYSTEM\$ argument

### Function Heading

```
FUNCTION SYSTEMD (Request: string_18): string_160;
```

### Example Function Calls

```
Some_thing := SYSTEMD (arg)
```

```
Or_other := SYSTEMD ("AVAILABLE MEMORY")
```

### Semantics

The “argument” (`arg`) is the string containing the argument for the BASIC `SYSTEM$` function. If the argument is invalid, the result returned is the string “ERROR 401” (i.e., the function does not escape). The value returned is of TYPE STRING[160].

This provides Security Module and ID PROM access plus many other capabilities. Refer to `SYSTEM$` in the *HP BASIC Language Reference* for further information.



# Index

---

## A

### Array Dimension Record

COMPLEX, 2-15, 4-11  
 INTEGER, 2-15, 4-11  
 REAL, 2-15, 4-11  
 String, 2-16, 4-12

### Array Dimension Structure

COMPLEX, 3-13  
 INTEGER, 3-13  
 REAL, 3-13  
 String, 3-14

### Array parameter type, 2-14, 3-13, 4-10

### Assembly CSUB, steps for creating an, 3-1

atest.c, 3-3

atrans, 6-2

## B

basic\_com routine, 4-15

BASIC file I/O, 2-38, 3-35, 4-31

bcmplxvaltype (by reference), 2-9

bcomplex\_parm, 3-8

bcomplex\_parm (by value), 2-9

binteger\_parm, 3-8

binteger\_parm (by value), 2-9

bintvaltype (by reference), 2-9

breal\_parm, 3-8

breal\_parm (by value), 2-9

brealvaltype (by reference), 2-9

bstring\_parm, 2-9, 3-8

## C

### C CSUB

components, 3-6

executing rmbuildc, 3-22

linking CSUB object files, 3-20

managing CSUBs from BASIC, 3-29

parameter passing, 3-7

steps for creating CSUBs, 3-1

character\*190, 4-6

character\*30, 4-6

character\*n, 4-6

clear\_screen, 2-36, B-2, B-3

COM blocks, accessing them from a CSUB,  
 2-20, 3-18, 4-15

Compiled subprogram (CSUB), 1-1

complex\*16, 4-6

COMPLEX parameter type, 2-11, 3-10,  
 4-7

controlcrt, 2-36

controlkbd, 2-36, B-4

CRT and keyboard I/O routines, 2-36

crtreadchar, 2-36, B-5

crtscroll, 2-36, B-6

csfa\_fal\_close, 3-35, 4-31

csfa\_fal\_create, 3-35, 4-31

csfa\_fal\_create\_ascii, 3-35, 4-31

csfa\_fal\_create\_bdat, 3-35, 4-31

csfa\_fal\_eof, 3-35, 4-31

csfa\_fal\_loadsub\_all, 3-35, 4-31

csfa\_fal\_loadsub\_name, 3-35, 4-31

csfa\_fal\_open, 3-35, 4-31

csfa\_fal\_position, 3-35, 4-31

**csfa\_fal\_purge**, 3-35, 4-31  
**csfa\_fal\_read**, 3-35, 4-31  
**csfa\_fal\_read\_bdat\_int**, 3-35, 4-31  
**csfa\_fal\_read\_string**, 3-35, 4-31  
**csfa\_fal\_write**, 3-35, 4-31  
**csfa\_fal\_write\_bdat\_int**, 3-35, 4-31  
**csfa\_fal\_write\_string**, 3-35, 4-31  
**csubdecl.h**, 1-4, 3-16  
**csubdecl** module, 2-18  
 CSUB interface, 2-25  
 CSUB Run-time Errors, Handling, 2-33  
 CSUBs, deleting, 4-27  
 CSUBs, Deleting, 2-32, 3-30  
 CSUBs, loading, 2-31, 3-29, 4-26  
 CSUB, steps for creating an assembly,  
     3-1  
 CSUBs, When to Use, 1-3  
 CSUB utilities  
     **csfa.o**, 1-4  
     **csubdecl.o**, 1-4  
     **kbcrt.o**, 1-4  
**cursor**, 2-36, B-8

## D

Device I/O, 2-38, 3-34, 4-31  
**dimentryp**, 2-9, 3-8  
**disp\_at\_xy**, 2-36, B-10  
 Dynamic memory allocation, 2-35, 3-33

## E

Errors, reporting, 2-34, 3-32, 4-29  
 Errors, **rmbuildc**, 2-28, 3-26, 4-23  
 Errors, trapping, 2-33, 3-31, 4-28  
 Example  
     finding the string, 2-3, 4-2  
     passing parameters, 3-2  
**EXPORT**, 2-6  
 External references, resolving, 2-23,  
     3-21, 4-19

## F

**fal**, 2-38, A-1  
**fal\_close**, 2-38, A-2  
**fal\_create**, 2-38, A-4  
**fal\_create\_ascii**, 2-38, A-5  
**fal\_create\_bdat**, 2-38, A-6  
**fal\_eof**, 2-38  
**FAL\_EOF**, A-7  
**fal\_loadsub\_all**, 2-38, A-9  
**fal\_loadsub\_name**, 2-38, A-10  
**fal\_open**, 2-38, A-11  
**fal\_position**, 2-38, A-13  
**fal\_purge**, 2-38, A-15  
**fal\_read**, 2-38, A-16  
**fal\_read\_bdat\_int**, 2-38, A-18  
**fal\_read\_string**, 2-38, A-20  
**fal\_write**, 2-38, A-22  
**fal\_write\_bdat\_int**, 2-38, A-24  
**fal\_write\_string**, 2-38, A-26  
**fc\_ptr\_type**, 3-8  
**fc\_ptr\_type** (by value), 2-9  
**fc\_type** (by reference), 2-9  
 File access reference, A-1  
 File access routines, 2-38, 3-35, 4-31  
**FORTTRAN COM** block structure, 4-16  
**FORTTRAN CSUB**  
     components, 4-5  
     executing **rmbuildc**, 4-19  
     linking CSUB object files, 4-17  
     managing CSUBs from BASIC, 4-26  
     parameter passing, 4-6  
     steps for creating CSUBs, 4-1  
**FSTR**, 2-3, 4-3

## G

Global variables, 2-22

## I

**IMPLEMENT**, 2-6  
**IMPORT**, 2-6  
**IMPORT csfa**

**fc***ptr*\_type (by value), 2-9  
**fc***type* (by reference), 2-9  
**integer**\*2, 4-6  
**INTEGER** parameter type, 2-12, 3-10, 4-8  
 I/O, BASIC file, 2-38  
 I/O, device, 2-38, 3-34, 4-31  
 I/O paths, 3-12  
 I/O Paths, 2-13, 4-10

## K

**kbd***clear*\_screen, 3-33, 4-30  
**kbd***ctrl*\_control*ctrl*, 3-33, 4-30  
**kbd***ctrl*\_control*kbd*, 3-33, 4-30  
**kbd***ctrl*\_crtreadchar, 3-33, 4-30  
**kbd***ctrl*\_crtscroll, 3-33, 4-30  
**kbd***ctrl*\_cursor, 3-33, 4-30  
**kbd***ctrl*\_disp\_at\_xy, 3-33, 4-30  
**kbd***ctrl*\_read\_kbd, 3-33, 4-30  
**kbd***ctrl*\_scrollup, 3-33, 4-30  
**kbd***ctrl*\_status*ctrl*, 3-33, 4-30  
**kbd***ctrl*\_status*kbd*, 3-33, 4-30  
**kbd***ctrl*\_system*dm*, 3-33, 4-30  
**kbd***ctrl*\_scroll*dm*, 3-33, 4-30  
 Keyboard and CRT I/O reference, B-1  
 Keyboard and CRT I/O routines, 2-36, 3-33, 4-30  
 Keyboard and printer I/O, 2-36, 3-33, 4-30

## L

**ld**, 2-2, 2-22, 3-20, 4-18  
**lib***rmb.a*, 1-4, 2-5, 2-23, 3-21, 4-18  
 Linker (HP-UX), 2-22, 3-20, 4-18

## N

**nm**, 2-23, 4-19

## O

Optional parameters, 2-19, 3-17, 4-14

## P

Palindrome, 6-8  
**PAL\_PROG**, 6-7  
 Parameters passed by reference, 2-8, 3-7  
 Parameter types, 4-6  
     comparison between C and BASIC, 3-8  
     comparison between FORTRAN and BASIC, 4-6  
     comparison between Pascal and BASIC, 2-9  
**param\_val**, 3-3  
 Pascal and BASIC parameter types, 2-9  
 Pascal CSUB  
     components, 2-6  
     executing **rmb***buildc*, 2-24  
     linking CSUB object files, 2-22  
     managing CSUBs from BASIC, 2-31  
     parameter passing, 2-7  
     steps for creating CSUBs, 2-1  
 Passing parameters by reference, 2-8, 4-6  
**paw\_file**, 6-2  
 Porting Pascal Workstation Assembly CSUBs, 6-1  
 Program  
     **atest.c**, 3-3  
     **FSTR**, 2-3, 4-3  
     **PAL\_PROG**, 6-7  
     **param\_val**, 3-3  
     **paw\_file**, 6-2  
     **string.f**, 4-3  
     **string.p**, 2-4

## R

**read\_kbd**, 2-36, B-12  
**real**\*8, 4-6  
**REAL** parameter type, 2-11, 3-10, 4-7

Reference, passing parameters by, 2-8,  
4-6

**rmbuildc**, 1-4, 2-2, 2-5, 2-24, 3-22,  
4-19

**rmbuildc** errors

when creating C CSUBs, 3-26

when creating FORTRAN CSUBs,  
4-23

when creating Pascal CSUBs, 2-28

Run-time errors, handling CSUB, 2-33,  
3-31, 4-28

## **S**

**scrolldn**, 2-36, B-13

**scrollup**, 2-36, B-14

**SEARCH** directive, 2-6

**statuscrt**, 2-36, B-15

**statuskbd**, 2-36, B-16

**string.f**, 4-3

**string.p**, 2-4

String parameter type, 2-12, 3-11, 4-8

System components, 1-4

**systemd**, 2-36, B-17

## **T**

Trapping errors, 2-33, 4-28

**TYPE** declarations, useful, 2-18

1



HP Part Number  
E2040-90003



Printed in U.S.A. E0891

E2040-90603 Manufacturing Number