CET BASIC
UNIX User's Guide

CET BASIC UNIX/XENIX User's Guide

Published and printed in the United States of America.

First Edition	1986
Second Edition	1987
Third Edition	February, 1995
Revision A	June, 1995
Third Edition	July, 1996

TABLE OF CONTENTS

CHAPTER 1: CET BASIC COMPILER AND RUNTIME SYSTEMS 1

INTRODUCTION 1
INSTALLATION GUIDE 1
CREATING CET BASIC PROGRAMS 2
USING THE CET BASIC COMPILER 2
THE COMPILER FLAGS 3
C Object Output Flag (-c) 4
Define Symbol Flag (-D*symbol*) 4
Error Flag (-e) 4
Include Path Flag (-I*path*) 4
Library Flag (-l*name*) 5
Output File Flag (-o *executable*) 5
Preprocessor and Preprocessor Only Flags (-P and -PO) 5
Report Flag (-R*step*) 6
Strip Symbol Table Flag (-s) 6
Assembler Output Flag (-S) 6
Include Trace Code (-T[*trace-sub*]) 6
UX-BASIC Flag (-U) 7
Stack Error Flag (-X) 7
Line Selection Flag (-#) 7
Version Number Flag (-V) 8
COMPILER AND RUNTIME ENVIRONMENT VARIABLES 8
VARIABLES USED BY THE CET BASIC COMPILER 8
B_BB 8
B_CMDFLAGS 8
B_LIB 9
B_THCONV 9
VARIABLES USED BY THE CET BASIC RUNTIME SYSTEM 9
B_BRTM 9
Environment Flag (-E) 9
File and I/O-Related Environment Variables 9
B_BUFSIZ 10
B_CSHFCLOSE 10
B_COM 10
B_ESCDELAY 11
B_FPATH 11
B_LKPATH 11

B_NRTM 11
B_OPENNL 11
B_OPTLOCK 11
B_STDINDLM 12
UXFPATH 12
Variables Which Affect Printer Use and Control 12
B_FFPTR 12
B_LPR*n* 12
B_MAPPRT*n* 13
B_SPOOL 13
B_SPOOLCMD 13
Variables Which Affect Screen Display 13
B_PBUFSIZ 13
B_SYS (Version 2.2X and 5.XX Only) 13
B_TERMTYPE (Version 6.XX Only) 13
Variables that Provide Compatibility with THEOS 14
B_<*drive-code*>DRIVE 14
B_CSHFCLOSE 14
B_DBSLIU 14
B_DOSTAB 14
B_EMULATE 14
B_GINVERT 15
B_OPTLOCK 15
B_SEARCH 15
B_THLOCK 15
B_THPON 16
B_THVAL 16
Miscellaneous Environment Variables 16
B_4DYEAR 16
B_BSNONDS 16
B_DATEFORM 16
B_LANGUAGE 16
B_USER 16
B_TRACE 17
B_YR2000 17

CHAPTER 2: EXTERNALLY COMPILED BASIC SUBROUTINES 19

INTRODUCTION 19
COMPILING PROGRAMS THAT CALL BASIC SUBROUTINES 20
THE BASIC SUBROUTINE STATEMENTS 20
THE $MAIN STATEMENT 21

THE $SUB STATEMENT 21
THE $GLOBAL STATEMENT 21
THE $CLEAR STATEMENT 23
THE $EXIT STATEMENT 24
CONVERTING EXISTING CET BASIC APPLICATIONS 24
A SAMPLE CONVERSION OF A CET BASIC APPLICATION 25
DIFFERENCES FROM CONVENTIONAL CET BASIC PROGRAMS 26

CHAPTER 3: TERMINAL INDEPENDENCE 29

INTRODUCTION 29
THE BTERMGEN UTILITY 30
Modifying the Line-drawing Characters 31

CHAPTER 4: OPERATING SYSTEM COMMAND INTERFACE 33

INTRODUCTION 33
THE CSH/CSI STATEMENT AND CSH FUNCTION 33
COMMAND EXECUTION 34
EXAMPLES 36

CHAPTER 5: BUILT-IN C LANGUAGE ROUTINES 39

INTRODUCTION 39
USING C LANGUAGE ROUTINES 39
THE BUILT-IN C ROUTINES 40
File Related Functions 40
Environment Related Functions 43

CHAPTER 6: DEVELOPMENT UTILITIES 47

INTRODUCTION 47
THE BCHECK COMMAND 47
THE BCREATE COMMAND 48
THE BLIST COMMAND 49
THE BPRETTY COMMAND 50
THE BREBUILD COMMAND 50
THE BRENUM COMMAND 51

APPENDIX A: CET BULLETIN BOARD SYSTEM 53

APPENDIX B: CONSOLE BUFFERING 57

INTRODUCTION 57
THE PBON AND PBOFF FUNCTIONS 57

APPENDIX C: CET NETWORK RUNTIME SYSTEM 59

THE TRADITIONAL ENVIRONMENT 59
THE CLIENT-SERVER ENVIRONMENT 59
USING THE NETWORK RUNTIME SYSTEM 60

CHAPTER 1

CET BASIC Compiler and Runtime Systems

Introduction

CET BASIC is a modern, compiled language designed for developing professional business applications With CET BASIC you can support a wide variety of platforms with *one* set of source code. The system automatically detects when a network or multi-user system is in use so that record and file locking take place transparently within your program.

CET BASIC is a full-featured implementation of the BASIC language that provides easy access to operating system commands and externally compiled BASIC, C or assembly language subroutines.

The language is easy-to-use, yet extremely powerful. It is a high-level, programming language far exceeding the current ANSI BASIC standards. Complex notations can be expressed quickly and briefly. More than a hundred built-in functions are available for string handling, data conversion, screen control, output formatting and mathematics. Special libraries provide additional bar code and C File Access support. The Window System from Phase One Systems is available for adding visual appeal and functionality to your programs.

Installation Guide

The CET BASIC UNIX-based products are available for systems such as SCO XENIX and UNIX, AIX RS6000, DG AViiON and SOLARIS. For brevity, these operating systems are referred to as UNIX in this manual.

The specific software requirements and installation procedures for the CET BASIC UNIX-based products are covered in separate documents depending upon the operating system in use. Please contact your CET distributor if you need this information.

Creating CET BASIC Programs

The CET BASIC Compiler accepts source programs which are stored in ASCII format since text files are the easiest to port to and from other operating systems.

CET BASIC programs may be created using **vi** or any other text editor. Refer to the *THEOS Compatibility Guide* for specific information on porting programs from the THEOS operating system.

Here is a quick procedure for creating CET BASIC programs. For information on writing programs, please refer to the *CET BASIC Language Reference Manual.*

1. Make a directory for your BASIC programs and change to that directory. For example, if the acronym for your package is "MMS", you might:

   ```
   mkdir mms
   cd mms
   ```

2. Use an editor to create a program with or without line numbers. Note that the compiler requires that BASIC program files have a **.b** extension. For example:

   ```
   vi newpgm.b
   ```

Using the CET BASIC Compiler

The OB Compilation Manager is a multi-purpose program designed to compile a CET BASIC program, link the program to the CET libraries and create an executable file.

OB operates similarly to cc, the UNIX C language compiler. This syntax is supported to provide compatibility between the CET BASIC products for UNIX, DOS and Windows.

The syntax for the OB command is shown below. The square brackets indicate an optional item. The names of program source files and/or object modules are *italicized* to indicate that they are variables to be entered by the operator. Input the names in any order and in the appropriate case mode. UNIX is case sensitive and distinguishes between upper and lower-case names.

OB is referred to in upper case letters for documentation purposes only. It must be entered as "ob".

ob [–flags] [*pgm*] [B-*pgm*] [C-*pgms*] [A-*pgms*] [O-*pgms*]

–flags

is any valid combination of flags or switches used to control the operation of the compiler. Entering OB without any parameters will list the currently supported flags and their function to the screen. (The individual compiler flags are described in the next section.)

The following example will compile the **newpgm.b** program and create an executable with the name **newpgm** (without an extension). The name of the executable program must follow the -o flag.

```
ob -o newpgm newpgm.b
```

pgm

is the name to be used for the executable. If no name is specified, OB uses the default name **a.out**. For example, the following command will compile **newpgm.b** and produce **a.out**.

```
ob newpgm.b
```

B-pgm

is the name of the BASIC source file to be compiled. This must be an ASCII formatted file with a **.b** extension.

C-pgms

is the name of one or more C language programs to be compiled. To use this feature, the program must have a **.c** extension. (Using OB to compile C programs is not only convenient, but it eliminates having to remember another syntax.)

Note that the CET product comes with a variety of C language functions. These functions are covered in a separate chapter in this manual and in the *CET BASIC Library Manual.*

A-pgms

is the name of one or more assembly programs with a **.s** extension. (Refer to the -S compiler flag description.)

O-pgms

is the name of one or more object modules with a **.o** extension.

The Compiler Flags

Compiler flags control the operation of OB. Enter the command without any parameters to list the flags and environment variables that are currently supported. Each of the compiler flags are described in this section.

It is important to note that the compiler flags must be entered in exactly the same case mode as shown in the manual.

C Object Output Flag (-c)

The -c flag may be used to generate an object module and terminate. If this flag is omitted, the intermediate object module will be erased after the program is compiled. For example, the following command will terminate after producing the file **newpgm.o**:

```
ob -c newpgm.b
```

The next command will compile the C program **cpgm.c** and the BASIC program **bpgm.b**, then terminate after saving **cpgm.o** and **bpgm.o** in the current directory:

```
ob -c cpgm.c bpgm.b
```

If C language programs are to be compiled, CET BASIC expects to find a separate C compiler installed on the system.

Define Symbol Flag (-D*symbol*)

If you use preprocessor statements such as #ifdef in your BASIC program, this flag may be used to define a value for the preprocessor symbol. This feature is covered in the *CET BASIC Language Reference Manual.*

Error Flag (-e)

This flag will invoke the **vi** command automatically when a compiler error is detected. For example:

```
ob -e -o newpgm newpgm.b
```

If an error occurs, the compiler will display the line on which the error was detected and prompt you to edit the program, quit, or continue. If you decide to edit the program, you will be given the option of continuing the compilation process after leaving the editor.

Include Path Flag (-I*path*)

This flag may be used to specify an additional directory that should be searched to find the files referenced by the #include preprocessor statements.

For example, if **newpgm.b** contains the statement #include "getfld", use the -I flag to direct the compiler to search for **getfld** in /**include**:

```
ob -I/include -o newpgm newpgm.b
```

A separate C compiler is not required to use the C preprocessor statements. (Refer to the *CET BASIC Language Reference Manual* for information on this special feature.)

Library Flag (-l*name*)

Normally, the linker program searches the current directory before searching **/lib** and **/usr/lib** for the libraries it needs to resolve any external symbols referenced with CALL statements in the BASIC program. The library flag (a lower case "L") may be used to pass the name of additional libraries that contain the required functions to the linker.

For example, the C language functions used in the (optional) Phase One Window System are stored in **/usr/lib/libPosWin.a**. If you call these window functions in a BASIC program, direct the compiler to search this library by entering:

```
ob -o newpgm newpgm.b -lPosWin
```

Note that this is only meant as an example since OB will detect the window calls and search this library automatically.

Output File Flag (-o *executable*)

The -o flag may be used to create an executable program with a specific name. For example, the following command will compile **newpgm.b** and produce the executable **newpgm.**

```
ob -o newpgm newpgm.b
```

If this flag is omitted, the default name **a.out** will be assigned to the executable.

Preprocessor and Preprocessor Only Flags (-P and -PO)

The CET UNIX product does not require a separate C compiler in order to use the C preprocessor commands: #define and macro substitution, conditional compilation with #if, etc. (Refer to the *CET BASIC Language Reference Manual* for syntax and usage.)

The C preprocessor is invoked with the -P flag before the BASIC compilation is performed. If the -PO flag is specified, only preprocessing is done. The *expanded* text is stored in a file with the same name as the source file and the **.i** extension.

Variables may also be defined and assigned values on the command line as in:

```
ob -P -Dabc -Ddef=xyz newpgm.b
```

This command would be equivalent to entering the following lines at the top of the source file **newpgm.b**:

```
#define abc
#define def xyz
```

Report Flag (-R*step*)

The report flag may be used for debugging purposes when you need to know the exact command used in each of the steps invoked by OB. The letter following the flag indicates which step to report:

-RB	BASIC compilation	-RC	C compilation
-RA	assembly	-RL	link

During the compilation of a typical BASIC program, OB will invoke **/lib/Bb** to produce a **pgm.s** file, **/usr/bin/Basx** to produce a **pgm.o** and finally **/lib/Bld** to link the assembly program to produce an executable.

Strip Symbol Table Flag (-s)

The -s flag is identical in function to its corresponding flag in the cc command. Using this flag will strip the symbol table from the executable and thus decrease the size of the generated program.

Assembler Output Flag (-S)

Normally, OB will erase the intermediate assembly language program at the end of the compilation process. The -S flag may be used to inform the compiler to generate the assembly program and terminate. For example, the following command will terminate after creating **newpgm.s** in the current directory:

```
ob -S newpgm.b
```

Note that the -S flag will apply to all source files specified on the command line regardless of whether they are BASIC or C language programs.

Include Trace Code (-T[*trace-sub*])

The -T flag directs the compiler to include line number tracing code when generating the assembly code from a CET BASIC program that uses an ON ERROR statement. For example:

```
ob -T -o myprog myprog.b
```

When you run the compiled program with B_TRACE set, the number of each line executed will be displayed on the screen. (Compile with the -# flag to get the actual line number if there are some unnumbered lines in your source file.)

The -T flag may also be used with the name of a C subroutine. In the following example, **mysub.c** will be linked into the BASIC program **myprog** and automatically called after each BASIC statement.

```
ob -Tmysub -o myprog myprog.b mysub.c
```

If this flag is omitted, tracing may only be obtained with a CALL to Bdebug and setting B_TRACE on.

UX-BASIC Flag (-U)

This flag may be used to indicate that all data files used in the program are to be opened, read, and written in UX-BASIC compatibility mode. Specifying this flag is equivalent to entering the statement "OPTION MATIO 0" at the top of the program and opening each file with the option UX. The statement OPTION NEWLINE will also be in effect.

The Bconvux utility is provided to convert UX-BASIC files to a CET BASIC format. Please refer to the *THEOS Compatibility Manual* for documentation.

Stack Error Flag (-X)

The -X flag may be used to include special code in the program which will check the state of the stack after the execution of each BASIC statement to ensure that the runtime components are being executed correctly.

This feature may help determine if an unexplained error is due to a compiler problem. When this flag is used, an error during execution will be displayed in the following format:

```
STACK ERROR .....
CET BASIC Terminated with RC=<return code>, PID=<process id>
```

If this type of error occurs, please give your distributor all of the information displayed by the error processor; the return code, statement number, type of statement that was being executed, etc.

Line Selection Flag (-#)

The -# flag may be used to specify how runtime errors are reported. When used, ERL and the line report will return the number of the actual line where the error occurred. Otherwise, the default is to return the number of the last numbered line that was executed successfully.

This flag is particularly useful when a BASIC program has some lines without numbers. For example, the following program will terminate with ERL = 120 if the -# flag is not used and with ERL = 2 when it is used.

```
100     X = 0 \ GOTO 120
BADLINE: PRINT 1 / X
120     GOTO BADLINE
```

Version Number Flag (-V)

The -V flag displays the CET BASIC compiler version and serial number.

```
ob -V
```

Compiler and Runtime Environment Variables

Environment variables may be used to pass additional information to the compiler and/or modify the behavior of the CET BASIC Runtime System. Use the following syntax to set an environment variable. By convention, the variable names are entered in uppercase letters.

In the C shell:	setenv *VAR value*
In the Bourne or Korn shell:	*VAR=value; export VAR*

To list the environment variables that are available, execute the OB command without any parameters. They have been grouped here by function for your convenience.

Variables Used by the CET BASIC Compiler

The following environment variables may be used to modify the behavior of the CET BASIC Compiler:

B_BB

If the compiler module **/lib/Bb** has been renamed, set this variable to the full pathname of the new compiler so that OB may operate successfully. Note that you may move **/lib/Bb** to any directory in the PATH variable without having to set B_BB.

B_CMDFLAGS

Set to the compiler flags to be enabled for each compilation. For example, if you use the (optional) Phase One Window System, you could enter the library flag **-lPosWin** to indicate where the compiler can find the window routines. Separate multiple flags with a space as in:

```
setenv B_CMDFLAGS "-e lPosWin"
```

When this variable is used, OB will indicate that it is enabling the default flags and display their values. Note that "**-lPosWin**" is only intended as an example since OB will automatically search this library when specific window functions are detected in the BASIC program.

B_LIB

Set to the pathname of the library directory. By default, the compiler checks **/lib/libB.a** for the modules that may be linked with the BASIC program.

B_THCONV

Set to 1 to ensure that converted codes for international characters with an ASCII value greater than 128 are output by the assembler. This variable is automatically set when B_EMULATE=THEOS is in effect

Variables Used by the CET BASIC Runtime System

A wide variety of environment variables may be used to modify the behavior of the CET BASIC Runtime System.

B_BRTM

If the runtime module **/lib/Brtm** has been renamed, set this variable to the full pathname of the new runner so that your programs will operate successfully. For example, if you were using the CET Network Runtime System described in the *Appendix*, you would need to set:

```
setenv B_BRTM /lib/Nrtm
```

Note that you may move **/lib/Brtm** to any directory in the PATH variable without having to set B_BRTM.

Environment Flag (-E)

This flag may be specified to display the values of the CET BASIC environment variables that are in use when the program is executed. This flag is intended for debugging use only. The program will abort after the list is displayed. For example:

```
newpgm -E
```

For your convenience the other variables which are available have been grouped here according to their use.

File and I/O-Related Environment Variables

Note that whenever CET BASIC expects to find a file name, any of the following formats will be considered valid:

1. File names may be in a UNIX format. The complete path name may be specified as in /**pos\cp/data/custname** or relative to the current directory as in **./custname**.

Anytime a file name begins with a dot character or a '/', CET BASIC assumes that you are using a file name *native* to the operating system and no *filetype* translation will be done. (See #2.)

2. File names may also be specified in a THEOS format with periods separating each part of the name. By default, BASIC will use the following *filetype* method to *translate* the name of the file.

THEOS File Name	UNIX File Name
CUST	CUST
CUST.DATA	DATA/CUST
CP.DATA.CUST	CP/DATA/CUST

Note that CET BASIC will always upper case THEOS file names. UNIX formatted names must be used to reference files with names in lower case.

Unless the complete path name is specified, BASIC will search for the file in the current directory, and then along the path specified with the B_FPATH variable. If the file is still not found, a trappable error 30 is detected. (The B_FPATH setting will be searched to find all files excepts for those opened for OUTPUT SEQUENTIAL. CET BASIC always expects to find these files in the current working directory.)

B_BUFSIZ

Set to the maximum record buffer size in bytes (default 2304). If a program attempts to open a file with a record length greater than the value of B_BUFSIZ, a message will be displayed to indicate that the variable should be increased. When using indexed files, remember that the record length is the sum of the KEYL + RECL. (The key is written to the data record so that it may be rebuilt in case of a file corruption.)

Note that this variable only applies to binary records created with a WRITE statement. ASCII records created using PRINT may be up to 32 Kbytes long.

B_CSHFCLOSE

Set to 1 to close all files before performing a CSH/CSI statement to execute another program. This variable will override the default which is not to close open files. (B_CSHFCLOSE is automatically set with B_EMULATE.)

B_COM

Set to the list of tty device names (in quotes) to use for COM ports. A maximum of 9 devices may be specified.

B_ESCDELAY

Set to the number of milliseconds to wait before determining whether an Esc key was entered or whether the character was part of an escape sequence. Some terminals may need to use a setting between 30 to 50 milliseconds. Experiment to find which value to use.

Note that setting this variable to zero would mean that no function keys could be used because every Esc would be immediately processed.

B_FPATH

Set to the list of search paths for OPEN statements. For example, if the file **/data/inquiry** is specified, BASIC will search the current directory, by default. When the file is not found and the variable is set to **/apf**, the /**apf/data/inquiry** file will be opened.

Note that the B_FPATH variable will be searched to find all files except for those opened for OUTPUT SEQUENTIAL. CET BASIC always expects to find these files in the current directory.

B_LKPATH

Set to the list of paths to search for executables used in LINK, CHAIN and RUN statements. For example, when CHAIN "MYPROG" is encountered, the system will search for the program first in the current directory, then along the path specified by B_LKPATH.

B_NRTM

Set to the server name for remote access (e.g. server!). Refer to *Appendix* for information on Nrtm, the CET Network Runtime System.

B_OPENNL

Set to 1 to enable the NEWLINE option for the OPEN statement. This will indicate that the line-feed character has been used as the record delimiter instead of the default carriage-return.

If this variable is not set, the NEWLINE option may have to be specified in each program statement that opens an ASCII sequential file.

B_OPTLOCK

Set to 1 so that the file pointer is not moved after executing a RESUME from within a routine designed to handle a record lock condition. This way, (MAT)

READNEXT and (MAT) READPREV statements will attempt to reread the desired record.

Note that if a lock condition is encountered when this variable is not set, BASIC will move on to the next or previous record after a RESUME statement.

The B_OPTLOCK variable is automatically set with B_EMULATE.

B_STDINDLM

Set to the type of delimiter to be used for the console output; either CR for a carriage-return or NL for a linefeed character. CR is the default value.

This variable does not have to be set to NL to successfully redirect standard input. CET BASIC detects when the standard input file is not a terminal, and sets B_STDINDLM to NL automatically.

You may wish to set B_STDINDLM to NL so that CET BASIC will properly interpret the keystrokes that are typed just prior to the execution of the program. This manner of *turning on* type-ahead is necessary when an application will key in responses while the program is being loaded.

Note that setting B_STDINDLM to NL will effectively eliminate CET's ability to detect the entry of an intended linefeed or ↓ key, unless these keys are remapped with a C program. (See the chapter on *Terminal Independence*.)

UXFPATH

Same as B_FPATH.

Variables Which Affect Printer Use and Control

B_FFPTR

Set to 1 to have a form feed issued upon closing the printer. The default is to not generate a form feed.

B_LPR*n*

Set to the command string for PRINTER*n* if you need more control over the printed output than you get by simply setting B_SPOOL to the number(s) of the printers to be spooled. For example, when output is directed to PRINTER1, BASIC will append the name of the spooled file to the following string and pass it to the **lp** command:

```
B_LPR1 lp -c pan
```

Note that under some environments, the **lp** option -c may need to be used to copy files to the print spooler, otherwise nothing will be output.

B_MAPPRT*n*

Set to the name of the file that contains the control sequences for PRINTER*n*. The following example indicates that the escape sequences for the CRT functions to use for PRINTER1 can be found in the **hplaser.lpr** file.

```
B_MAPPRT1 hplaser.lpr
```

Refer to the *Formatted Output* chapter in the *CET BASIC Language Reference Manual* for information on using this feature.

B_SPOOL

Set to one or more spooled printer numbers separated by a colon. For example, to indicate that output directed to PRINTER1 and PRINTER3 is to be spooled, set:

```
B_SPOOL 1:3
```

B_SPOOLCMD

Set to the spooler command to use unless it has been previously set with the B_LPR*n* variable.

Variables Which Affect Screen Display

B_PBUFSIZ

Set to the screen buffer size in bytes. The default size is 2048 which should be large enough to hold all the characters necessary so that the screen display can be updated quickly.

For example, if you were to set this variable to 1, one character at a time would be output to the screen and displayed *very* slowly!

B_SYS (Version 2.2X and 5.XX Only)

Set to either UNIX or XENIX to specify how the CRT function should output color attributes. (This applies only to versions of CET BASIC prior to 6.14.)

B_TERMTYPE (Version 6.XX Only)

Set to the type of device in use. Acceptable values are currently ANSI, KIMTRON and WYSE for Wyse/Televideo type terminals. The B_TERMTYPE variable must be set in order for the CRT$ function for attributes such as CRT$("FCx") and CRT$("BCx") to work properly, especially on non-ANSI terminals.

Variables that Provide Compatibility with THEOS

The following variables may be set so that CET BASIC programs will behave as they do under the THEOS operating system. Setting B_EMULATE=THEOS automatically activates the other variables in this section with the exception of B_?DRIVE, B_CSHFCLOSE and B_SEARCH.

To turn off a default behavior, set the desired environment variable to null.

B_<*drive-code*>DRIVE

Set to affect the manner in which THEOS drive designations used in an OPEN statement are interpreted. For example, if B_SDRIVE is set to **/TREND**, then the following code could be used to access /**TREND/DMS/DATA/DMS01**. (THEOS formatted filenames are always in upper case.)

```
lib$ = "DMS"
drive$ = "S"
open #1: LIB$&".DATA.DMS01:"&DRIVE$, update direct
```

Setting this variable will override the default which is to search for the file in the current directory, and then along the path set by B_FPATH.

B_CSHFCLOSE

Set to 1 to close all files before performing a CSH/CSI statement to execute another program. This variable will override the default which is not to close open files.

B_DBSLIU

Set to 1 to enable the destructive backspace during LINPUT USING statements. Normally, the backspace character is not destructive. (Refer to the B_BSNONDS variable if you want the reverse behavior.)

B_DOSTAB

Set to 1 so the TAB character is returned as INP=9 in LINPUT USING statements. The default is to ignore the TAB.

B_EMULATE

Set to THEOS to emulate the following THEOS behavior:

1. Return THEOS INP values for LINPUT and INPUT
2. Use THEOS ASCII values for CRT$ function values
3. Map to THEOS characters for ASCII values greater than 160 decimal

(See also B_THCONV)

4. Enable default foreground/background display as per THEOS
5. Change print Esc character from 255 to 27
6. Change PAGE(0) and LINE(0) from 25->23 and 80->79 for normal screen
7. CRT("EU") erases foreground values 0,1,4,5,10,11,14,15
8. Return INP=9 for LINPUT statement on TAB, not insert mode
9. Fold unquoted or single quoted CMDARG() values to upper case

   Note that double quoted CMDARG() values must be preceded with a '\' to maintain case mode. For example:

   ```
   pgmname firstarg \"second arg\" thirdarg
   ```

Setting B_EMULATE=THEOS automatically sets the other variables in this section with the exception of B_?DRIVE and B_SEARCH.

B_GINVERT

Set to 1 to invert the meaning of intersection characters when used with the CRT$ function as in (CRT$("UI") and CRT$("DI").

B_OPTLOCK

Set to 1 so that the file pointer is not moved after executing a RESUME from within a routine designed to handle a record lock condition. This way, (MAT) READNEXT and (MAT) READPREV statements will attempt to reread the desired record.

Note that if a lock condition is encountered when this variable is not set, BASIC will move on to the next or previous record after a RESUME statement.

B_SEARCH

Set to the THEOS drive search sequence to use, by default SABCDEFG...Z

B_THLOCK

Set to 1 to perform multi-user record locking and suspend program operation when a lock condition is detected while accessing a record in a direct or indexed file. Since this is the typical THEOS behavior, this variable will be automatically set under B_EMULATE.

When a record is locked, the operator must wait until it is unlocked. Since a lock condition is not an error, normal error trapping can not be used. To overcome this problem, set B_THLOCK to null. Then, use an ON LOCK or ON ERROR statement to define the special handling for a lock condition.

For information on handling file/record lock conditions, refer to the *CET BASIC Error Handling System* chapter in the *CET BASIC Language Reference Manual.*

B_THPON

Set to 1 to invert the default meaning of PON/POFF so that a high-intensity foreground is used. CET BASIC normally displays low-intensity text which is typical in the UNIX environment.

B_THVAL

Set to 1 so that the VAL and NBR functions work as in THEOS. Refer to the *CET BASIC Language Reference Manual* for a definition of the standard method used by CET to handle these functions.

Miscellaneous Environment Variables

B_4DYEAR

Set to 1 to inform the DATE$ and DTE$ functions that a 4-digit year should be returned. Refer to the variable B_YR2000 if you wish to indicate which year should be the first one to be interpreted as 20XX.

B_BSNONDS

Set to 1 to disable the destructive backspace during LINPUT USING operations. This is the normal behavior unless B_EMULATE is set to THEOS.

B_DATEFORM

Set to 2 (European) or 3 (International) to override the default which is to display American-formatted dates. The BASIC OPTION DATEFORM may be used to change the format from within a program.

B_LANGUAGE

Set to a 2-character code to force the YESNO$ function to generate the proper text for English (EN) - the default, French (FR), Dutch (DU), German (GE), Swedish (SW), Finnish (FI), Spanish (SP), Portuguese (PO), or Italian (IT).

B_USER

Set to 1, 2, 3... to specify a unique user in the system. (See the *Built-in C Functions* chapter for an alternative method using BgetTtyMajMin.)

This feature is convenient when a program must create temporary work files in a multi-user environment. The following code segment illustrates how use the

value of B_USER to create an indexed file with a unique name for each user. The file will be stored in the /**tmp** directory.

```
call Bgetenv(addrof(user$),"B_USER")
filename$ = "/tmp/work"&user$
CALL Berase(filename$)
CALL Bcreate(filename$ " indexed reclen 100 keylen 10")
OPEN #1: filename$, update indexed
```

This procedure will work equally as well with a sequential file. In that case, instead of using the Bcreate function, use OPEN with the option OUTPUT to write the file.

B_TRACE

Set to 1 to display the information generated by compiling a program with the -T flag. The number of each line executed will be displayed on the screen. (Compile with the -# flag to get the actual line number if some or all of the lines in your source file are unnumbered.)

This feature is useful if you need to see which lines were executed (before the program failed).

B_YR2000

Set to the 2-digit year which is to be the last year interpreted as 20XX. This feature allows you to have dates in the 1900's and 2000's without using the complete 4-digit year.

For example, suppose you have no dates prior to January 1st, 1925. In that case, you could set

```
B_YR200O=24
```

Setting the above variable to the last year to be interpreted as 20XX would make the statements DAY("1/1/24") and DAY("1/1/2024") identical.

Using our example, the following dates will be interpreted as:

01/01/00	01/01/2000
12/31/24	12/31/2024
01/01/25	01/01/1925
12/31/99	12/31/1999

CHAPTER 2

Externally Compiled BASIC Subroutines

Introduction

The CET BASIC UNIX-based (and Windows) products allow you to compile a BASIC program that calls external BASIC subroutines. This feature may be used to integrate a number of CET BASIC programs into a single executable. The advantages of doing this are:

- Execution time is much faster, as all CHAIN and LINK operations are eliminated.
- The size of the resulting executable, though often large, is much smaller than the many executable BASIC programs that it replaces. This permits an entire application to be shipped on fewer diskettes than before.
- Some developers have found that calling externally compiled subroutines is a more natural way to program. Using subroutines instead of CHAINs and LINKs is similar to the way in which C, Pascal, COBOL and other applications are written.
- These executables, when run in a multi-user environment, often require much less total system memory, as every user can be running a program whose text space is shared.
- It is much easier to write structured code. Commonly used routines may be stored in external files, and then called as needed by simply entering the name of the subroutine in the main program. This feature saves valuable development time since code does not have to be constantly retyped.
- Maintenance is much easier. A BASIC subroutine may be modified to reflect the new requirements. Then, simply recompile all the programs that use the subroutine.

BASIC subroutines use a number of new statements. This chapter is intended to document these statements and point out how they affect the operation of a few of the conventional CET BASIC statements.

Compiling Programs that Call BASIC Subroutines

Programs that call BASIC subroutines are compiled with OB using the normal syntax and options. The only difference is that you can simultaneously compile several modules with one command. For example:

```
ob -o main main.b sub1.b sub2.b
```

It is also possible to compile each module separately and link them together, as:

```
ob -c sub1.b
ob -c sub2.b
ob -o main main.b sub1.obj sub2.obj
```

The ability to separately compile each BASIC subroutine, and then relink the entire program set, is an important feature. Recompilation is much quicker when you have just one change to make in a module.

The BASIC Subroutine Statements

The BASIC subroutine statements differ from conventional CET BASIC statements in two ways:

- Every statement begins with a dollar sign ($).
- With the exception of $EXIT and $CLEAR, the statements are not executable. Instead, they direct the compiler to treat the module and its constituent statements in a certain way.

The statements are:

$MAIN	Indicates that this module is the main routine
$SUB	Indicates that this module is a subroutine
$GLOBAL	Provides a list of variables that are global to all routines
$CLEAR	Clears all variables and dimensioned arrays in the subroutine except for those defined with $GLOBAL.
$EXIT	Exits from a BASIC subroutine

The $MAIN Statement

The $MAIN statement should be the first statement in the main module. The syntax is:

```
$MAIN
```

There can be only one module which uses the $MAIN statement. The main module is the program segment in which your application begins, and is the *root* of the execution tree for your application.

The $SUB Statement

A BASIC subroutine begins with a $SUB statement that defines the subroutine so that it can be called by another BASIC subroutine or program. The syntax is:

```
$SUB module-name
```

Module-names are case sensitive, therefore TESTsub, testSUB and TestSub are all different. For example:

```
$SUB  mysub
$SUB  WriteRpt
```

BASIC subroutines are invoked with the BASIC CALL statement. The calling statements associated with the $SUB statements above are:

```
CALL  mysub          used to call $SUB mysub
CALL  WriteRpt           used to call $SUB WriteRpt
```

The $GLOBAL Statement

Data can be shared between BASIC subroutines using either of the following methods:

1. Any variable referenced within the same source module or **.b** file refers to the same variable. In other words, all variable names are global between BASIC subroutines that reside in the same source file.
2. Any variables that are used in separate source modules are distinct unless they. are defined by the $GLOBAL statement. This is true even for variables with the same name.

This *name management* feature permits you to write and compile separate BASIC subroutines to be linked into an executable without worrying about conflicting variable names.

When the $GLOBAL statement is used, it must follow the $MAIN in the main module and the $SUB in the BASIC subroutine. The syntax for the statement is:

$GLOBAL *variable-list*

The *variable-list* is a list of BASIC variable names which refer to the same variable in all modules. The variable list is composed of simple scalar variable names, and/or matrix (array) names preceded by the keyword MAT.

If multiple variables are to be shared, they can be specified on separate $GLOBAL lines or all on the same line with commas separating the variable names. For example, a typical $GLOBAL statement might be:

```
$GLOBAL GVAR, MAT MYARRAY, ANYSTG$, I%, MAT XSTG$
```

Note that an array must be dimensioned with either the DIM or COMMON statement before it is passed to a BASIC subroutine.

As an example, consider the following main program and subroutine. There are three shared variables. The values for CNT% and the array ARRAY% are assigned in the main routine and passed to SumArray. In fact, these are exactly the same variables in both programs, and occupy the same address in memory.

```
REM Main program stored in MAIN.B
$MAIN
$GLOBAL CNT%, MAT ARRAY%, ANSWR%
DIM ARRAY%(10)
FOR I% = 1 TO 10
      ARRAY%(i%) = I%
NEXT
CNT% = 10
CALL SumArray
PRINT "The answer is: "; ANSWR%
END
```

The subroutine will sum the CNT% number of elements and return the result in ANSWR%.

```
REM Subroutine SumArray stored in SUM.B
$SUB SumArray
$GLOBAL CNT%, MAT ARRAY%, ANSWR%
ANSWR% = 0
FOR I% = 1 TO CNT%
      ANSWR% = ANSWR% + ARRAY%(I%)
NEXT
$EXIT
```

To compile an executable called **main.exe** you could enter the command:

```
ob -o main main.b sum.b
```

It is also possible to compile each module separately and link them together, as:

```
ob -c sum.b
ob -o main main.b sum.obj
```

To summarize, the important points to remember when using variables in programs that call BASIC subroutines are:

- Every variable which occurs in a $GLOBAL list is the same variable in every module in which it is used.
- If a variable is not in a $GLOBAL (or COMMON) variable list, it may be cleared upon entering or leaving a subroutine by using a $CLEAR statement.

The $CLEAR Statement

Variables are not automatically cleared upon entry into a subroutine. $CLEAR may be used when it is necessary to clear all variables and dimensioned arrays that are not defined as $GLOBAL or COMMON. The syntax of the statement is:

```
$CLEAR
```

The reason for clearing all variables derives from the fact that this is what normally happens when a CET BASIC program is first executed. Moreover, programs rely upon the variables in separate programs being distinct (unless they are in COMMON), even if they happen to have identical names.

$CLEAR is provided so that a *new* program that uses BASIC subroutines to combine a number of separate programs which were formerly integrated using CHAIN, LINK or RUN statements will operate the same way as before.

Unless a variable is in a $GLOBAL list (or held in COMMON), $CLEAR ensures that the following actions occur upon entry into a subroutine:

- Scalar integer and real variables are set to zero.
- Scalar string variables are set to a null string.
- Matrices are cleared. Every matrix is initialized to a one-dimensional array with 10 elements, each of which is zero or null, depending upon the data type.

The $EXIT Statement

$EXIT is an executable statement. When $EXIT is encountered, control passes back to the calling module, to the statement following the CALL. Internally, this statement operates like a RETURN. An $EXIT is implied at the end of a $SUB module if one does not exist. The syntax of the statement is:

```
$EXIT
```

The END, QUIT and STOP statements may not be used instead of $EXIT. If one of these statements is detected in any module, the entire program will terminate.

Converting Existing CET BASIC Applications

BASIC subroutines are often used to convert CET BASIC applications which are composed of a number of separate programs. The steps involved include:

1. Determine what the execution *tree* should look like for your application. For example, suppose you have an application with a main menu program which CHAINs or LINKs to other submenu programs.
2. Add $MAIN on the first line of your main (menu) program which uses CALL statements to invoke the next level of (submenu) programs (now BASIC subroutines), depending upon the conditions for their execution (i.e. user response, a fixed sequence or some other method of selection).
3. Define all previously shared COMMON variables with a $GLOBAL statement. Remember to use the MAT keyword in front of any matrix (array) defined as global. (Also make sure that all $GLOBAL variables are dimensioned.)
4. Change all occurrences of CHAIN and LINK to CALL statements. The precise way in which you do this, and the placement of the CALL statement depends upon the structure of your *execution tree*.
5. Use $CLEAR to clear all variables and arrays not defined with $GLOBAL before entering (or exiting) the subroutine so they will not contain values loaded during a previous CALL statement.
6. Modify the subroutines to return to the calling program with a $EXIT. It is no longer necessary to remember the name of the program for statements such as CHAIN "menu". $EXIT will ensure that control is automatically passed back to the $MAIN program which issued the CALL statement.
7. Terminate all GOSUB statements with a RETURN before a $EXIT is executed. Otherwise, the program will transfer control to the last GOSUB statement performed.

It is possible to CHAIN, LINK or RUN among sets of main program modules. In these cases, you may still want to use COMMON to communicate variables and arrays between the programs. COMMON statements should only be used in the $MAIN module. To share variables (COMMON or not) with a BASIC subroutine, you must use a $GLOBAL statement.

A Sample Conversion of a CET BASIC Application

The following programs illustrate what needs to be done to convert a main menu program which CHAINs or LINKs to other submenu programs.

BASIC Program menu.b

```
COMMON CONAME$

PRINT "Select Program"
PRINT "1. Accounts Receivable"
PRINT "2. Accounts Payable"
PRINT "3. General Ledger"
PRINT "4. EXIT"
INPUT ANS%
SELECT ANS%
      CASE 1
            CHAIN "ar"
      CASE 2
            CHAIN "ap"
      CASE 3
            CHAIN "gl"
      CASE 4
            QUIT
CEND
```

Converted Program menu.b

```
$GLOBAL CONAME$
WHILE 1
      PRINT "Select Program"
PRINT "1. Accounts Receivable"
      PRINT "2. Accounts Payable"
      PRINT "3. General Ledger"
      PRINT "4. EXIT"
      INPUT ANS%
      SELECT ANS%
            CASE 1
                  CALL ar
            CASE 2
                  CALL ap
            CASE 3
                  CALL gl
            CASE 4
                  QUIT
      CEND
WEND
```

BASIC Program ar.b

```
COMMON CONAME$
<main body of ar>
CHAIN "menu"
```

Converted Program ar.b

```
$SUB     ar
$GLOBAL CONAME$  \  $CLEAR
<main body of ar>
      $EXIT
```

BASIC Program ap.b	**Converted Program ap.b**
	$SUB ap
COMMON CONAME$	$GLOBAL CONAME$ \ $CLEAR
<main body of ap>	<main body of ap>
CHAIN "menu"	$EXIT

BASIC Program gl.b	**Converted Program gl.b**
	$SUB gl
COMMON CONAME$	$GLOBAL CONAME$ \ $CLEAR
<main body of gl>	<main body of gl>
CHAIN "menu"	$EXIT

Very few modifications were made to convert the conventional CET BASIC application. None of the code in the *main body* of the programs (which involves hundreds of lines) was modified. The CHAIN statements (back to the main menu) were replaced with $EXITs. COMMON statements were replaced with $GLOBALs, and $CLEAR was added to clear the other variables and arrays.

Differences from Conventional CET BASIC Programs

Error Handling

It is also important to note that each BASIC subroutine does its own error handling. Every subroutine where an error might occur should begin with an "ON ERROR GOTO" statement. This also applies to lock and interrupt key processing.

Also not that there are some CET BASIC statements that act differently than when compiled into separate program modules. The following sections address these differences:

RESTORE and READ with DATA Statements

Normally, the READ statement, when unaccompanied by a file number, will read through a list of data specified by one or more DATA statements. Each successive READ *consumes* the next item in the data list.

The *pointer* to the next data item may be changed by using RESTORE. This statement can *rewind* the pointer to the first DATA statement or change the current pointer to the DATA statement following a designated statement number.

When using BASIC subroutines, the following behavior should be noted:

- READ statements only read data items located in that module.
- RESTORE only refers to a DATA statement in the same module.

COMMON Statements

The COMMON statement is a powerful executable statement which adds a list of variables to the currently allocated common list. During a CHAIN or LINK, this common list is matched up with the COMMON statement(s) in the target program.

When using BASIC subroutines, it is not advisable to use the COMMON statement in any but the main ($MAIN) module. If access to common variables is required in other modules, a $GLOBAL statement should be used in each module in which you need to access the variables.

User Defined Functions

A user defined function, specified by the DEF statement, is only recognized in the module in which it is defined.

CHAPTER 3

Terminal Independence

Introduction

In an operating system such as UNIX where many different types of terminals may be attached to the same system, it is important that programs are able to function independently of the type of device in use.

The CET UNIX-based products rely on the termcap system for conventional terminal independence. The termcap library was chosen because of the absence of some terminal control such as CRT("FON") in terminfo. Moreover, termcap permits you to define new terminal control strings, which we have done for line-drawing characters and others.

With CET BASIC, you can design and code applications without having to consider what terminals may eventually be used. The BASIC system will translate the characters used to control the various console characteristics to the correct escape sequences for the device. This procedure is covered in the Appendix of the *CET BASIC Language Reference Manual.*

This chapter is included to provide you with the information you need to add or edit the terminal capabilities. Typically, the TERM environment variable must be set to add a terminal.

Using the C-shell:	setenv TERM *term-type*
Using the Bourne or Korn shell:	TERM=*term-type ;* export TERM

The name *term-type* must be one of the terminal types defined in the termcap library used by CET BASIC, and should reside in **/etc/termcap**.

If necessary, the entry for the *term-type* in the **/etc/termcap** file may be edited to include new termcap tokens to describe functions that use graphics sets, half-intensity, format on, etc.

The Btermgen Utility

The Btermgen utility is included as an easy-to-use, menu-driven means of adding or editing a termcap entry. The syntax of the command is:

Btermgen *term-name* [-f *output-file*] [-t *similar-entry*]

term-name

Specifies one of the abbreviations for a particular terminal. The first field of each entry in the **/etc/termcap** file contains the abbreviations that are recognized. Btermgen will use the environment variable TERMCAP, if present, instead of **/etc/termcap** as the name of the file to search for *term-name*.

-f output-file

Indicates that the modified entry should be written to the *output-file* instead of the termcap file from which it came.

-t similar-entry

Indicates that when the *term-name* is not found, a new entry with name *term-name* is to be created by using the specifications for the *similar-entry*.

Editing with Btermgen is simple and direct. The appropriate fields are displayed on the screen with their two-letter termcap code and a brief functional description. The value of each field is expressed as an ASCII string using all of the termcap conventions. Spaces are added between each character representation to enhance readability. For example:

Actual Entry:	\E[7m
Displayed Entry:	\E [7 m

Note that any spaces in the termcap entry are represented as '\040'.

Btermgen recognizes the following editing keys:

Editing Keys	Function
CTRL+J or <return>	Moves down to the next field.
CTRL+K	Moves up to the previous field.
CTRL+E	Quits Btermgen without saving the changes.
CTRL+F	Files the new entry and quits the program.
CTRL+H	Erases a character.
<space>	Deletes the entire field.

Entering any other character will put the editor into insert mode, where each character will be interpreted as part of the field's value. Press the Enter key to end insert mode. To exit insert mode without changing the field, first back up (Ctrl+H) to the beginning of the field and press Enter.

Most special keys can be entered by simply typing the key. For example, pressing the Esc key will produce the string "\E", which termcap-processing programs recognize as an Esc character.

Note that on some terminals, entering the ↑ arrow might produce the string “\E[A” instead of a CTRL+K. In that case, values may be entered in ASCII (e.g., a TAB written as the character '^' followed by the character 'I') or as octal equivalents (e.g., DELETE entered as the character string "\177").

Note that although Btermgen will create new termcap entries from scratch, it is not intended for that purpose and would prove very unsatisfactory if used in that fashion. Several terminal capabilities that are not required by CET BASIC are not included in the Btermgen menu.

Modifying the Line-drawing Characters

If you use line-drawing characters in your programs, you will need to modify the termcap entries so that these characters are displayed correctly.

For example, the values for “ansi” should be as follows:

ULC	Upper Left Corner	\332
URC	Upper Right Corner	\277
LRC	Lower Right Corner	\331
LLC	Lower Right Corner	\300
FWI	4-way Intersect	\305
LI	Left Intersection	\303
RI	Right Intersection	\264
UI	Up Intersection	\302
DI	Down Intersection	\301
HOR	Horizontal	\304
VERT	Vertical	\263
RULC	not supported - leave empty	
RURC	“	
RLRC	“	
RURC	“	
DULC	Double Upper Left Corner	\311
DURC	Double Upper Right Corner	\273
DLRC	Double Lower Right Corner	\274
DLLC	Double Lower Left Corner	\310
DFWI	Double 4-way Intersect	\316
DLI	Double Left Intersect	\314

DRI	Double Right Intersect	\271
DUI	Double Up Intersect	\313
DDI	Double Down Intersect	\312
DHORZ	Double Horizontal	\315
DVERT	Double Vertical	\272

CHAPTER 4

Operating System Command Interface

Introduction

The CSH/CSI statement provides a means of executing UNIX commands from within a CET BASIC program. The CSH function is an extension of the BASIC statement that will also pass the UNIX return code for the command executed back to the calling program.

The *CET BASIC Language Reference Manual* describes the operation of these features. This chapter is intended for those individuals who wish to understand the more advanced aspects and how they apply to UNIX in particular.

The CSH/CSI Statement and CSH Function

The CSH/CSI statement has the following syntax:

CSH *string-expression*

The CSH function operates similarly, and is often coded as:

***integer-variable* = CSH(*string-expression*)**

Note that under THEOS emulation, unquoted or single quoted CMDARG() values are folded to upper case. Any double quoted values must be preceded with a '\' to maintain case mode. For example:

```
pgmname firstarg \"second arg\" thirdarg
```

The advantage in using the CSH function is that a return code is passed back to the program where it can be used to take a corrective action, if the command executed by the function was unsuccessful.

Throughout the remainder of this chapter, the CSH function will be used in the examples.

Command Execution

The CSH function provides a means of executing commands the same way they would be executed from the command line. CSH performs three functions: shell invocation, signal processing, and terminal attribute set and reset.

Shell Invocation

The Bourne shell (**/bin/sh**) is invoked to process and execute the designated command. For this reason, commands can reference environment variables, specify redirection, or contain any shell metasymbols. The shell will look at every directory specified in the PATH variable and attempt to execute the designated command.

Another feature of the Bourne shell is the ability to initiate a process in background. CET BASIC provides the setup and restoration necessary for this feature to work properly. (See example 2 at the end of this chapter.)

Terminal Attribute Set and Reset

CSH sets terminal attributes (**stty** parameters) to the values which were in force in the calling program, and resets them when the CSH operation terminates.

This means that programs that perform their own terminal handling (i.e. other CET BASIC programs) will not interfere with any special terminal handling done in the current program.

The following example demonstrates a typical use of CSH:

```
rc% = CSH("ls > temp.out")
IF rc% = 0
      OPEN #1: "./temp.out", INPUT SEQUENTIAL NEWLINE
      WHILE NOT EOF(1)
            LINPUT #1: L$
            PRINT L$
      WEND
IFEND
```

The first statement will invoke the **ls** command, causing its output to be redirected to the file ./**temp.out**. The return code provided by **ls** is passed back to the program as the value of rc%. If the result of the operation was successful, then the subsequent statements will be executed.

Signal Processing

In UNIX nomenclature, abnormal events which can lead to termination are called signals. CSH initializes and resets signal trapping in conformance with UNIX conventions. CET BASIC normally traps all signals except for SIGKILL signal 9, which is untrappable. By trapping signals, the following services are provided:

- All files are closed prior to exit.
- The terminal port characteristics are reset to those which existed prior to entering a CET BASIC program.
- Control is passed to the ON ERROR or ON INTERRUPT statements, if they are currently active.

When a CET BASIC program is executed in background, all signals are set to the values at initial program execution. This generally means that all signals except SIGKILL are disabled.

Consider the CET BASIC program called **exbas** which may be executed in background by typing:

```
#exbas &
1294
#next-command
```

In this example, the "#" character is shown as the shell prompt. The underlined portion indicates what is typed in, and the remainder is the output of the shell. (1294 is the process id of **exbas**.) The program would run to completion, unless an error occurred or the process associated with **exbas** was terminated by entering:

```
kill -9 1294
```

If **exbas** were executed normally (without the background designator '&'), then entering the interrupt key (typically the DELETE key) will terminate the program unless ON ERROR or ON INTERRUPT is in effect.

Now, consider a situation in which **exbas** invokes the CSH function. The effect of the signals on **exbas** and the CSH executed program are summarized as:

- If **exbas** is run in background (**exbas** &), neither **exbas** nor the program executed by CSH will respond to signals other than SIGKILL.
- If **exbas** is run in foreground (**exbas**), then
    a. If the CSH is run in foreground (CSH("ls > temp")), then **exbas** and the program executed by the CSH will respond to signals.

b. If CSH is run in background as in CSH("ls > temp &"), then **exbas** will respond to signals, but the background program will not.

Examples

The following examples illustrate the proper use of the CSH function.

Example 1: Determining the Characteristics of a File

Perhaps the most common use of CSH is in determining the accessibility of pathnames, and the creation of direct and indexed files.

The standard UNIX utility, **test**, provides an easy means of determining the attributes of a file. The following code illustrates a typical use of the **test** command in conjunction with the CSH function.

```
IF CSH("test expression") = 0
        REM expression has tested TRUE
        appropriate action...
ELSE
        REM expression has tested FALSE
        alternative action...
IFEND
```

Some possible test conditions are:

Expression	Test Condition
-r *file*	*file* exists and is readable.
-w *file*	*file* exists and is writable.
-f *file*	*file* exists and is not a directory.
-d *file*	*file* exists and is a directory.

Complex expressions can be formed using the following operators:

Operator	Description
!	Negation
-a	AND
-o	OR
\(	Left parenthesis
\)	Right parenthesis

For example:

```
IF CSH("test -r afile") <> 0 THEN CSH("create ./afile direct reclen 10")
```

```
IF CSH("test \( -d " & f$ &" \) -o ! \( -r " & f$ &" \)")
        PRINT "File ";f$;" is directory or not readable"
        QUIT
IFEND
OPEN #1: f$, INPUT SEQUENTIAL
```

Example 2: Background Tasks

In some applications, it may be useful to begin a stand-alone program which runs concurrently with the primary application program.

One such situation occurs when the developer wishes to permit a file to be printed while normal application processing continues. Another example is the need to backup user files to tape. The following code demonstrates this concept:

```
LINPUT "Do you wish to backup your files", ans$
IF UCASE$(ans$)[1:1] = "Y"
        LINPUT "Hit Carriage-Return when you have inserted diskette", cr$
        CSH("tar cvF "& filename$ &" > tar.output &")
IFEND
```

In this example, the names of the files to be backed up to diskette are contained in *filename$.* The output generated by the v option of **tar** is redirected to the file **tar.output**. CSH returns immediately after beginning the **tar** and program execution is resumed while the backup is being done.

CHAPTER 5

Built-in C Language Routines

Introduction

CET BASIC provides you with the means to enter the unconstrained (and reasonably unprotected) UNIX environment and perform operations which would be difficult or inefficient to code using BASIC alone.

This chapter describes some of the C language routines that are available. See the *CET BASIC Library Manual* for information on the additional functions that are provided. Refer to the *C Interface Manual* if you wish to write your own C functions.

Using C Language Routines

The BASIC CALL statement is used to transfer control to an external C or assembly language routine. The names of the routines must be in exactly the same case mode as shown in this manual in order for the linker to resolve the reference during compilation.

While it is not required that you know how to program in C before using C functions, it is helpful to understand how the CALL statement operates.

There are two modes to the CALL statement. Mode 1 uses the syntax "CALL *routine"* to transfer control to a specific routine. No information is passed as arguments to the routine or back to the calling program. A few of the CET routines such as Becho use this syntax.

The majority of the C routines use mode 2 of the CALL statement to pass one or more arguments to the specified routine using the syntax "CALL *routine* (*arg-list*)". Note that when using this mode, it is important to pass the type and number of arguments in the expected order. Otherwise, the CET Compiler will detect an error.

For example, if a routine such as Bgetport expects an integer value (portno%), you must pass it an integer variable or constant and not a floating point.

The special function ADDROF may be used to pass the memory address of a variable to a C routine rather than the value itself. The routine will evaluate the address of the variable and assign it a new value. For example, the ADDROF an integer variable is passed to Bgetport.

```
CALL Bgetport(ADDROF(portno$))
```

The Bgetport routine will assign an index value to the variable portno$ so that it may be used by the BASIC program.

MATADDROF is a similar function that may be used to pass the address of an array variable.

For more information on the CALL statement, review the documentation on the statement in the *CET BASIC Language Reference Manual*.

The Built-in C Routines

The currently supported routines have been documented in this chapter. As developers need additional functionality, new C routines are provided. Check with your CET distributor if you do not find the feature you want.

For your convenience, the routines have been grouped here by functionality.

File Related Functions

The following functions have been provided to eliminate the need to perform a CSI/CSH operation. We recommend that you use these functions for improved speed and, in some cases, reliability. (Refer to Berase and Brename.)

Bchmod

The Bchmod function may be used to change the permission of a specific file. The syntax is:

```
CALL Bchmod( filename$, mode$ )
```

In the following example, the read/write permissions are set for the group, other and all for the indexed file cp/data/custname. Note that Bchmod only supports UNIX formatted file names.

```
call Bchmod ( "cp/data/custname.idx", "0666" )
call Bchmod ( "cp/data/custname.dat", "0666" )
```

Bchown

The Bchown function may be used to change the ownership of a specific file. The syntax is:

CALL Bchown(filename$, owner$,group$)

Note that only UNIX formatted file names are supported. For example, to change the owner and group name for the indexed file cp/data/custname:

```
call Bchown ( “cp/data/custname.idx”, “compass”, “compass” )
call Bchown ( “cp/data/custname.dat”, “compass”, “compass” )
```

Bcopy

The Bcopy function may be used to copy a file and eliminate having to perform a CSI statement to execute the UNIX cp command. The syntax of the CALL statement is:

CALL Bcopy(“*source-filename destination-filename*”)

Bcopy recognizes a *filename* in either a UNIX or THEOS format. If the destination file does not exist, it will be created with the same specifications as the source file.

Bcreate

Bcreate operates like the CET Bcreate utility except that it is executed from within a BASIC program by using a CALL statement instead of a CSI. The syntax of the statement is:

CALL Bcreate(“*create command line*”)

The name of the file to be created may be in either a UNIX or THEOS format. For example, both of the following will create the indexed file **data/custname** with a key length of 11 and a record length of 350.

```
CALL Bcreate(“data/custname indexed reclen 350 keylen 11”)
CALL Bcreate(“custname.data indexed reclen 350 keylen 11”)
```

The following function will clear the contents of the indexed file **data/test**. Both the keys in the **.idx** file and the associated data in the .**dat** file will be cleared.

```
CALL Bcreate(“data/test clear”)
```

For more information on usage, please refer to the description of Bcreate in the *Development Utilities* chapter.

Berase

Berase is designed to eliminate having to perform a CSI statement to execute the UNIX rm command. Note that the function must be used to erase indexed files to avoid possible C-ISAM problems. In any case, the file must be closed before it can be erased. The syntax of the statement is:

CALL Berase("*filename*")

The *filename* may be in either a UNIX or THEOS format. If the file is indexed, both the **.idx** and the associated **.dat** file will be erased.

Bfilestat

This function may be used to determine if a directory or file exists. Its syntax is:

CALL Bfilestat("*filename*",addrof(status$))

The parameters are defined as:

filename

Specifies the path or filename in a UNIX format such as ./**data/test** or a THEOS format such as **test.data**. If *filename* refers to an indexed file, then the .**idx** (or .**dat**) extension must be specified as in ./**data/test.idx**.

status

Specifies a string that returns a D for directory, R for a file, S for a special file or a null if the file does not exist.

Note that this function has been available in the past as filestatus. Although the function is still supported, filestatus only recognizes UNIX formatted names.

Bindinfo

The Bindinfo function may be used to get information about an open indexed file. It will return the number of records, the record length and the key length. The syntax of the statement is:

CALL Bindinfo(ch%,addrof(cnt),addrof(recl%),addrof(keyl%))

The parameters are defined as:

ch

Specifies the channel used to open the indexed file.

cnt

Returns the current number of records.

recl

Returns the record length which is the sum of the key plus the data.

keyl

Returns the length of the key.

Brename

The Brename function is designed to eliminate having to perform a CSI statement to execute the UNIX mv command. Note that the function must be used to rename indexed files to avoid possible C-ISAM problems. In any case, the file must be closed before it can be renamed. The syntax of the statement is:

CALL Brename("*original-filename new-filename*")

The *filename* may be in either a UNIX or THEOS format. If the file is indexed, both the **.idx** and the associated **.dat** file will be renamed at the same time.

Environment Related Functions

Becho and Bnoecho

These routines may be used to turn on or off the echoing of characters which are entered from the standard input file. This includes all INPUT, LINPUT and LINPUT USING statements which address the CONSOLE device.

No arguments are passed to the CALL statements. Their syntax is:

CALL Becho

CALL Bnoecho

Bgetenv

The Bgetenv function may be used to determine the current setting for an environment variable such as TERM, LOGNAME, and PATH. The syntax is:

CALL Bgetenv(addrof(setting$),env$)

The parameters are defined as:

setting

Returns the information stored in the environment variable or a null if the requested information is not available (i.e. the variable has not been set).

env

Specifies the environment variable.

For example, if TERM is entered as the *env*$, the value of *setting*$ would be "ansi" when the console is in use.

Note: If you have used the CET DOS product, are probably wondering why the Bputenv function has not been mentioned. Unfortunately, this feature is not available under UNIX. Although the environment variable setting may be changed with a C function, the variable will be reset to its original value as soon as control is transferred back to your BASIC program.

Bgetport

The Bgetport routine may be used to determine which port is currently in use. The feature is convenient in a multi-user environment where an application must create a unique work file for each user. In that case, the value returned by Bgetport may be appended to the name of the work file. (See BgetTtyMajMin for a similar function.) The syntax of the CALL statement is:

CALL Bgetport(addrof(portno$))

The value of portno$ is the port associated with the current terminal (/**dev**/**tty***n*). A zero is always returned for the (main) console. The following code illustrates how this feature may be used.

```
CALL Bgetport(ADDROF(portno$))
filename$="temp";str$(portno$)
open #1: filename$, update sequential
```

BgetTtyMajMin

The BgetTtyMajMin function may be used to identify the (terminal) device in use by returning its major and minor number. Since each session on a monitor has its own major and minor number, this function may need to be used instead of Bgetport when you need to know exactly which device is in use.

The syntax is:

CALL BgetTtyMajMin(addrof(majmin%))

For example:

```
CALL BgetTtyMajMin(ADDROF(majmin%))
maj%=majmin% / 256
min%=majmin% AND 255
mm$=FORMAT$(maj%,"999")&FORMAT$(min%,"999")
temp$="./cp/data/work"&mm$
```

The above example concatenates the maj% and min% values after they have been converted to strings and left-padded with zeros. The new variable is then appended to a file name to make it unique for each session (user).

Binton and Bintoff

These routines may be used to enable or disable the ability to recognize the interrupt key during program execution. Calling Bintoff will effectively disable ON INTERRUPT and ON ERROR processing for ERR = 1.

No arguments are passed to the CALL statements. Their syntax is:

CALL Binton

CALL Bintoff

Bsernum

Bsernum may be used to get the serial number from the CET KeyPlug. The syntax of the CALL statement is:

CALL Bsernum(addrof(serno%))

The parameter is defined as:

serno

Returns zero if the application is being run in demonstration mode with no KeyPlug. Otherwise, the serial number of the runtime system is returned to your program.

CHAPTER 6

Development Utilities

Introduction

The CET BASIC product includes a variety of utility programs that may be used in program development and data file recovery. These utilities are described in this chapter. To clarify their syntax, the following conventions have been used:

[ ] Square brackets indicate an optional parameter.

| A vertical bar indicates a choice between two parameters.

italics Italics indicate a variable name to be supplied by the user.

Since UNIX is case sensitive, all of the utilities must be entered exactly as shown in this manual.

The Bcheck Command

Bcheck may be used to rebuild a corrupted indexed file. This command differs from Brebuild in the following ways:

- Bcheck is a product of Informix, Inc. that is designed to check and fix indexed file inconsistencies.
- Bcheck may be used to check file consistency and list the b-tree entries without actually modifying the file.
- Bcheck will only accept UNIX formatted filenames.
- Bcheck may be used interactively or with flags. To list the flags, enter the command without any arguments.
- Bcheck rebuilds the specified file in place. Deleted records are not removed so the size of the file remains the same.

The Bcreate Command

The Bcreate command or its synonym CREATE may be used to create indexed and direct files or to clear the contents of an existing file. The syntax is:

Bcreate *filename* [*options*]

filename

Specifies the name of the file in either UNIX or THEOS format. Use a UNIX format to create files with names in lower case. Otherwise, the file name will be in upper case.

Bcreate recognizes the following options. The minimum abbreviations have been indicated with an underline character. If you have used a similar command under THEOS, it is important to note that UNIX does not allow a left parenthesis (before specifying the options) unless it is preceded with a backslash character.

indexed

Creates an indexed file which is actually two files: an **.idx** file for the record's keys and a **.dat** file for the data.

direct

Creates a direct (relative access) file.

reclen *n*

Allocates a record length of *n*.

keylen *n*

Allocates a key length of *n*. This option is ignored for direct files.

clear

Clears the contents of an existing file. The keylen and reclen will remain the same. Note that the file must be closed in order for this option to function successfully.

All CET BASIC files are dynamically allocated so the THEOS filesize option will be ignored, if used. If the command is used to create a THEOS keyed file, the program will create an indexed file instead.

For example, to create an indexed file with a key length of 11 and a record length of 350 enter either:

```
Bcreate data/custname indexed reclen 350 keylen 11
CREATE custname.data indexed reclen 350 keylen 11
```

The first command will create the **data/custname.idx** and **data/custname.dat** files in lower case letters since a UNIX formatted name was specified. The

second will create **DATA/CUSTNAME.idx** and **DATA/CUSTNAME.dat** because a THEOS filename was used.

The Blist Command

Blist may be used to display the contents of indexed, direct or sequential files. The data files may be in either binary or ASCII format. A banner will be displayed to identify the filename and filetype. The syntax of the command is:

Blist [-d] [-u] [-q*char*] [-f*char*] [-s*char*] *file*

-d

Displays the escape character '\' and the hexadecimal value of any non-printable characters found in the file. When this option is omitted, a period will be displayed by default.

-u

Specifies that the files are in UX-BASIC format.

-qchar

Defines the character to be used to surround the strings in the display.

-fchar

Defines the character to be used to surround the string fields.

-schar

Defines the character to be used to surround string subfields.

file

Specifies the name of one or more files to be listed. File names may be in either UNIX or THEOS format. For example:

./cp/data/custname CP.DATA.CUSTNAME

The contents of indexed files will be displayed in alphabetical order by key. Direct files will be listed in ascending numerical order. Sequential files may have either the <carriage return> or <line-feed> character as the record delimiter.

Note that Blist does not paginate. If this feature is needed, the output should be piped through the MORE utility.

The Bpretty Command

The Bpretty command may be used to rearrange the spacing and indentation of CET BASIC source files to reflect the program structure.

The syntax is as follows. Note that the options must be entered first, otherwise they will be interpreted as the names of the files to be used.

Bpretty [-*n*] [-t] [-i] [*source* [*destination*]]

-n

Indents n spaces. The default is 4.

-t

Replaces spaces with TAB characters wherever possible.

-i

Inserts text from #include files before performing any other operations.

Filenames must be in a UNIX format. If no destination file is specified, Bpretty will output to the screen. If no source file is specified, input is read from the keyboard.

Because this utility can read input from the keyboard and output to the screen, it may be used as a filter in conjunction with other UNIX or CET utilities to perform several tasks at once. For example:

```
Bpretty source.b | Brenum -a +5 > newfile.b
```

The Brebuild Command

Brebuild may be used to reconstruct an indexed file when disk problems or a file corruption has made a file completely or partially unusable. Since the program needs to analyze the data portion of the indexed file in order to rebuild the index, its capability is limited to how complete the data segment is.

This utility is also handy for reducing the size of a file by removing the records that have been marked as deleted during normal data entry operations.

The syntax of the command is:

Brebuild *filename*

filename

Specifies the UNIX filename, without the .dat or .idx extensions.

During execution, the program attempts to determine the length of the key and data record of the file being rebuilt. This is done by reading the **.idx** file, where this information is normally stored, and a number of records in the **.dat** file.

If the key length and record length cannot be determined, you will be prompted for this information. Your entry will be validated against the existing data records. If the lengths supplied and the existing records do not agree, then the file cannot be rebuilt.

When the file can be rebuilt, the program will rebuild the file in place. Any records marked as deleted will not be written and the size of the file will be adjusted accordingly.

The Brenum Command

Brenum may be used to renumber CET BASIC source programs with up to 32,767 lines. The syntax is:

Brenum [*source* [*destination* [-a] [-n *initial*] [-i | + *increment*]
[-b *begin*] [-e *end*]

Filenames must be in UNIX format. If no destination file is specified, the output will be directed to the screen. If no source file is specified, input will be read from the keyboard.

The command options are:

-a

Causes all lines within the given range to be renumbered. When omitted, the default is to renumber only the lines with numbers.

-n *initial*

Starts renumbering with the number *initial*. The default is 10.

-I *increment*

Increments the new line numbers by *increment*. The default is 10.

+ *increment*

The plus character will be treated as a -i *increment*.

-b *begin*

Begins renumbering at the *begin* line. When omitted, the default is to begin with the first line of the program.

-e *end*

Stops renumbering at the *end* line. When omitted, the default is to end with the last line of the program.

The flags may be arranged in any order provided that the parameters associated with the flags are entered in corresponding order. For example, the following command will renumber **source.b** and output the result in **dest.b**:

```
Brenum -a source.b dest.b
```

The following examples will all produce the same results: **source.b** will be renumbered from line 200 to line 300. The new line numbers will begin with 600 and increase in increments of 5. The result will be output to **dest.b.**

```
Brenum source.b dest.b -nibe 600 5 200 300
Brenum source.b -b 200 -e 300 -n 600 -i 5  dest.b
Brenum -n600 +5 -be 200 300 source.b dest.b
```

Because Brenum can read input from the keyboard and write to the video screen, it can be used as a filter in conjunction with other UNIX or CET utilities to perform several tasks at once. The redirection symbol is used to emphasize the destination file in the following example.

```
Bpretty source.b  | Brenum -a +5 > newfile.b
```

Note that the lines of code contained in the "#include" files (if any) will not be renumbered, even when the -a option is used. Renumbering portions of a file may cause those lines to be moved to preserve line number order.

APPENDIX A

CET Bulletin Board System

The CET Bulletin Board System (BBS) is an on-line bulletin board used to store current releases, release notes, and technical memoranda. You may browse through the BBS and download any software you need.

You may also leave messages or problem reports for the CET support staff. Programs illustrating a problem may be uploaded to the SUPPORT library. If you submit a problem report via the BBS, please include all the pertinent information regarding the problem along with your fax number so that we can respond as quickly as possible.

The CET BBS is available 24 hours a day, 7 days a week, except at 3 a.m. when the system is down for maintenance for approximately 30 minutes.

CET Bulletin Board Systems are located in several places around the world. The CET BBS in Boulder, Colorado, may be reached at:

(303) 530-9358

Contact your local CET distributor for the other numbers that are available.

Logging onto the BBS

Use a communications package such as PROCOMM PLUS which provides common download protocols, such as ZMODEM, Kermit and Y-MODEM. We suggest using a 14.4 Kbaud modem set with 1 stop bit, 8 data bits, and no parity. Do not use compression as this will tend to slow the download process. CET files are already compressed before they are uploaded.

Once you have entered the CET BBS, you must supply an account ID and password. If you do not have an account, enter "new". You will be asked to supply some information about your company and yourself, in addition to the ID and password you wish to use.

After you have finished the login procedure, you can traverse the system. Menus and on-line help simplify the process.

Downloading UNIX-based Files

Each of the UNIX-based products are stored in a separate library on the CET BBS. Currently, there are:

Library	Products
DGAVION	DG AViiON version
RS6000	AIX RISC 6000 versions
UNIX500	PC UNIX version 5.0
UNIX6XX	PC UNIX version 6.xx releases
UX22X	UNIX/XENIX version 2.2x

Although some of the files on the BBS are text files that can be simply copied onto the system, normally they will be one of the following types:

Extension	Type
z	A compressed executable such as **ob**.
az	A compressed library (**.a**) file.
taz	A compressed tar file.

Use the following procedure for downloading these types of files:

1. Download the files using a DOS communications package.
2. Copy the files to a DOS formatted diskette.
3. Change to the appropriate subdirectory:

   ```
   cd /tmp
   ```

4. Copy the files onto the UNIX-based system using the following syntax:

   ```
   doscp -r A:BB.Z   Bb.Z
   doscp -r A:LIBB.AZ  libB.a.Z
   doscp -r A:TK.TAZ  tk.taz
   ```

   Make sure that filenames are entered in the correct case mode. The name to use is specified in the short file description.

5. Decompress the files:

   ```
   compress -d Bb.Z
   compress -d libB.a.Z
   compress -d < tk.taz > tk.tar
   ```

6. Modify the file protection, when necessary, by changing the read and execute permission bits as in:

   ```
   chmod +x Bb
   chmod +r libB.a
   ```

7. Move the files to the appropriate directory. For example:

   ```
   mv Bb /lib/Bb
   mv libB.a /lib/libB.a
   ```

8. Tar the file:

   ```
   tar xvf tk.tar
   ```

If the **.taz** file is too large to copy to a DOS diskette, you may use the DOS tar utility (in the SHWARE library on the BBS). The syntax of the command is:

tar -S1440 -cvf b: *filename*

Note that the tar command is case sensitive. The flags must be entered in the same case mode used in the example above. The "1440" indicates a high-density 3.5" diskette.

Generally CET files are either executables, libraries or object modules. Many enhancements are implemented in the CET Link Libraries. When an enhancement or fix is made, it is normally possible to download just the object module affected, and then insert it in your library. For example, when a change is made to the CSH function, it will be implemented in the module **b_csh.o** and the following files would be uploaded to the BBS:

b_csh.o	The new object module.
libb.az	A compressed file that contains the complete library **/lib/libB.a** including the new **b_csh.o**.

While the compressed library file may be 200K, the decompressed object module may be only one or two Kbytes. If you choose to download **b_csh.o,** use the first 3 steps listed above to load the module on the UNIX-based system. Then, insert the module into the library. For example:

Under UNIX:	**ar rv /lib/libB.a b_csh.o**
Under XENIX:	**ar rv /lib/SlibB.a b_csh.o**
	ranlib /lib/SlibB.a

APPENDIX B

CONSOLE Buffering

Introduction

Since video memory is not available under UNIX, CET BASIC provides you with a method of buffering characters to increase the output rate to the console. Essentially the CRT$ functions PBOFF and PBON are used to collect screen output from a number of PRINT statements, and then output them to the screen all at once.

Note that the Window System, an optional package from Phase One Systems, may also be used to handle the screen display. In that case, all you need to do is initialize the system and all the PRINT statements will automatically pop up at the same time. A full range of other functions are available so you can add more visual appeal and functionality to your application. Contact your CET distributor if you need more information.

The PBON and PBOFF Functions

When a program *prints* the character CRT$("PBON"), all subsequent characters that are output to the console are held in an internal print buffer. CRT$("PBOFF") must be printed to send this buffer to the console. After that, any other characters that are printed will be output to the console immediately.

A typical usage of this function might be:

```
PRINT CRT$("PBON");CLS$;
PRINT a screen full of information, which will not start appearing until...
PRINT CRT$("PBOFF");CRT$("PBON"); CLS$;
REM now start constructing the next screen
```

APPENDIX C

CET Network Runtime System

The CET Network Runtime System (Nrtm) is an add-on product that has been released for further beta testing under SCO UNIX. This software is available from your CET distributor or from the UNIX6XX library on the BBS in Colorado as **nrtmbeta.taz**. This file contains the files **./lib/Nrtm** and **./lib/Brtm** which supports the new protocol.

The Traditional Environment

Traditionally, the UNIX-based CET BASIC applications have used Brtm to perform indexed and direct file operations and handle file and record locking. Brtm must be run as background process because of the nature of the C-ISAM file system and the fact that a new UNIX process is created and executed when a BASIC program performs a CHAIN, LINK or RUN. Although UNIX does maintain a great deal of information from one process to the next, it is not possible to retain the status of the internal tables for C-ISAM. The runtime system must be kept active to handle the indexed file I/O.

Normally, the CET UNIX-related products have been installed on a single computer system where the communication between a CET BASIC application and Brtm takes the form of packets. Over the years, CET has developed highly efficient code to pass these packets and to synchronize the operations between the two processes.

The Client-Server Environment

In the past, some resellers have set up CET applications to run in an NFS client-server environment. Unfortunately, they have not always been happy with the results when using the NFS implementation for SCO UNIX. They report that the NFS software is not only slow, but it does not always handle remote locking correctly.

The CET Network Runtime System (Nrtm) has been designed as an alternative to NFS when you need to centralize some or all of your application (database) on the file server. If a CET application detects that the database is not on the local drive, Brtm will then connect to a remote Nrtm which may be anywhere on the network. Nrtm will handle all file accesses to the server to eliminate file and record locking problems.

All Nrtm remote file accesses are routed via TCP/IP only (not NFS). This will greatly improve performance. The results of our preliminary tests writing records to an indexed file on the server (keyl=32 and recl=1024 bytes) show:

	1000 records	records per second
Using Nrtm	0 min. 25 sec.	40
Using NFS	3 min. 51 sec.	4 to 5

The results show that using Nrtm to update ISAM files with records of 1024 bytes or less (on the server) are quite acceptable. Nrtm performs these operations at a rate of 40 to 50 records/second. Unfortunately, tests with larger record sizes show that read and write operations are performed at the rate of 6 to 8 records/second. While this is still faster than with NFS, we will try to increase the speed of these operations in the next release.

Using Nrtm has an additional benefit. Since TCP/IP is a universal protocol, it is now possible to run Nrtm on one architecture such as an RS6000 machine running AIX and support requests from CET applications on other systems like SCO UNIX.

Using the Network Runtime System

The new versions of Nrtm and Brtm are designed to be compatible with CET applications compiled with version 6.0, although we recommend that you recompile with the current version that is available from the CET BBS or your CET distributor.

Nrtm currently supports a maximum of 32 IP connections. It is a daemon that is invoked the first time a CET application attempts to open a file on the server using the following format:

```
<server-name>!<file description>
```

For example, if the name of your server is "amsserver", then you would open the **/u/ams/data/custmast** file as:

```
amsserver!/u/ams/data/custmast
```

You can place the server-name in front of (all) file names automatically by setting the environment variable B_FPATH to the server-name as in the following examples:

```
B_FPATH=amsserver!
B_FPATH=amsserver!/u/ams/data
B_FPATH=amsserver!/u/ams/data:amsserver!/u/ams/work
B_FPATH=amsserver!/u/ams/data:/u/ams/work
```

Note that if all the data is stored on the server, you may set the environment variable B_NRTM to the server-name as in:

```
B_NRTM=amsserver!
```

To obtain debugging output from Nrtm or Brtm, set the variables TCP_DEBUG and/or B_DEBUG at both the workstation running the CET application and at the server.

A future release will include a utility that will monitor Nrtm and display all the connections on the network. The utility will also allow you to kill an inactive connection, if necessary.