CSUB Utility Contents

CSUB Overview

The CSUB Toolkit allows one to build compiled subprograms for HTBasic. A compiled subprogram, or CSUB, runs directly on the processor hardware and has access to all of it's power and functionality. Many functions can be performed with CSUBs that otherwise would be impossible with BASIC programs.

Once built, a CSUB may be loaded into HTBasic with LOADSUB and deleted with DELSUB just as any other SUB program. CSUBs may be stored in a PROG file and loaded along with other subprograms.

Manual Organization

This section of the manual presents the basic information required to write, test, and build a CSUB routine. The following list explains what each chapter describes.

CSUB Overview, contains information about the required tools, supported CSUB development languages, development support options, special projects support, distribution media, and installation instructions.

CSUB Environment, describes the CSUB execution environment, error reporting conventions, elements of a CSUB, and CSUB resource restrictions.

CSUB Parameters, describes how HTBasic passes CALL parameters to the CSUB, data and dimension pointer types, the dimension table, how to handle OPTIONAL arguments, and the NPAR value.

Building a CSUB, presents the steps required to build a CSUB for use with the HTBasic for Windows version.

COM Variables, describes the *com_var* function, used to locate COM memory variables, the movable nature of COM data, and the use of COM statements in CSUB prototype definitions. Two program examples are presented that demonstrate how to use the *com_var* function.

Display and Keyboard Routines, describes the display and keyboard routines available to a CSUB.

C Examples, provides simple C language CSUB examples. Also included is an example that defines several CSUBs combined into one CSUB context.

Include File Listings, lists the C language include files.

CSUB Languages

CSUBs may be written in any language that can handle C type argument pointers. The CSUB object code is linked into a Windows DLL or ***dynamic linked library***. The HTBasic CSUB utility combines an HTBasic SUB definition file and the DLL control information to create a PROG file. This help file describes how to use the C language to create CSUBs.

Pascal CSUB's

Because of major differences in the HTBasic CSUB environment and computer resources, this CSUB Toolkit does not provide specific help in converting existing HP 9000 Workstation Pascal CSUBs.

Development Support

Creating an HTBasic Windows CSUB involves some knowledge of the Windows and C language runtime environments.

Windows CSUB Toolkit Files

The HTBasic Windows CSUB Toolkit consists of an include file, the HTBasic CSUB utility program and a C language example program.

Filename	**Description**
CSUB.EXE	Windows CSUB Builder Utility
CSUB.HLP	Windows CSUB Builder Help file
CSUBW.H	Windows CSUB include file
EXAMPLE	Windows C language example program directory

The Windows CSUB example program files are located in the \HTBWIN\CSUB\EXAMPLE directory.

Filename	**Description**
WINTEST.MAK	Visual C++ make file
WINTEST.DEF	Linker DEF
WINTEST.DLL	Windows DLL file
WINTEST.C	C source
WINTEST.PRO	CSUB Prototype PROG file
WINTEST.CSB	CSUB PROG file
RUNTEST.BAS	BASIC Test program for Wintest CSUB

CSUB Environment

This chapter describes the CSUB execution environment, error reporting conventions, elements of a CSUB, and resource restrictions.

CSUB routines run directly on the processor and have access to all of its instructions and power. They execute as if they were internal HTBasic routines using the HTBasic processor stack. Arguments are passed to the CSUB using the normal C language calling method.

The value returned from the CSUB by the C *return* statement is used as the BASIC error value. A zero value denotes no error. Error values should be limited to values in the range 1 through 32760 and should correspond to the HTBasic error values. If a non-zero value is returned the line number reported will be the line number of the CALL statement.

A floating point exception handler is set up to return control to HTBasic upon any floating point exception. It uses a *longjmp* function to return control to HTBasic's CALL routine and includes the floating-point error code. The line number reported with the error will be the line number of the CALL statement.

Care must be taken to insure that all user and C runtime library routines function correctly within the CSUB environment.

Dynamic Link Libraries

Dynamic Link Library CSUBs

HTBasic for Windows can execute certain types of Windows 32-bit DLLs (Dynamic Link Libraries) as CSUBs. This requires the addition of the DLL and calling information to the CSUB context.

CSUB Elements

A Windows CSUB is identical to a tokenized SUB program with the addition of Windows DLL information. The CSUB program context is made up of normal tokenized BASIC program lines with the supporting symbol, name, and dimension tables. There are no relocation table or object code sections. The DLL information section contains the name of the DLL and other CSUB calling information.

Dynamic Link Library Routines

The Windows 32-bit DLL is generated by a compiler and linker combination that makes DLLs compatible with the Win32 DLL format for Intel processors. The DLL can access all the standard C library runtime routines as well as those described in the Microsoft Win32 Programmer's Reference, with the exceptions noted in the Microsoft Win32s User's Guide. A DLL destined to be a CSUB can also use any of the routines documented in the *"Display and Keyboard Routines"* section to access the display and keyboard through HTBasic for Windows.

CSUB Call Interface

This chapter describes how HTBasic passes CALL arguments to the CSUB, how to define data and dimension table pointers, and how to handle OPTIONAL arguments.

CALL Actions

When a CALL statement is executed, the normal HTBasic parameter matching and expression evaluation occurs up to the point where the interpreter would execute the first tokenized SUB program line. At that point, the CSUB routine is called and it executes as if it were another internal HTBasic routine. After control returns to HTBasic the return value is tested for an error condition and the interpreter continues execution.

Pass by Reference

All CALL arguments are passed by ***reference***. That is, all user arguments are pointers to the data. Simple numeric variables generate one pointer that points to the variable data. Strings and Array variables generate two pointers. The first points to the first element of the data array and the second points to the array dimension table. The array elements are stored in row-major order. String elements are made up of the current string length followed by the string character data.

Pointer Types

The include file *csub.h* defines the parameter pointer types and data structures. These parameter types and their pointer names are summarized in the following table.

BASIC	**Type C Type**
INTEGER	intptr
REAL	realptr
COMPLEX	cpxptr
Numeric Array	realptr or intptr and dimptr
String	strptr and dimptr
String Array	strptr and dimptr

Dimension Table

The dimension table is organized as follows:

Name	Size	Description
elen	2 bytes	Element Length
tae	4 bytes	NOD/Total Allocated Elements
cae	4 bytes	Current Allocated Elements
sbs[0]	4 bytes	Base and Size Elements (first subscript)
. . .	. . . .	
sbs[5]	4 bytes	Base and Size Elements (sixth subscript)

The ***elen*** item defines the element length. It is 2 for INTEGER, 8 for REAL, 16 for COMPLEX, or the string dimensioned length. Only the ***elen*** item is defined for simple string variables.

The ***tae*** item defines the total number of allocated elements in the lower 24 bits and the number of dimensions in the upper 8 bits. Use the NOD macro defined in *csub.h* to extract this value.

The ***cae*** item defines the current number or elements in use. This value can be different from the value of *tae* if the program has executed a REDIM statement.

The ***sbs*** item defines the ***base*** and ***size*** values for each dimension. Only the dimensioned number of ***sbs*** items are allocated and defined. For example: If an array is defined with one subscript, only one ***sbs*** item is allocated and defined.

INTEGER Arguments

For INTEGER arguments the CSUB routine requires a pointer to the integer data value. This is defined as follows:

```
100 SUB TEST1(INTEGER A)
```

```
int test1( int NPAR, intptr a)
```

Where *NPAR* is the number of arguments in the CALL and ***a*** is a pointer to the integer data.

REAL Arguments

For REAL arguments the CSUB routine requires a pointer to the real data value. This is defined as follows:

```
110 SUB TEST2(REAL B)
```

```
int test2( int NPAR, realptr b)
```

Where *NPAR* is the number of arguments in the CALL and ***b*** is a pointer to the real data.

COMPLEX Arguments

For COMPLEX arguments the CSUB routine requires a pointer to the complex data value. This is defined as follows:

```
120 SUB TEST3(COMPLEX C)
```

```
int test3( int NPAR, cpxptr c)
```

Where *NPAR* is the number of arguments in the CALL and ***c*** is a pointer to the complex data.

Array Arguments

For Array arguments the CSUB routine requires two pointers, one for the data and one for the array dimension table. This is defined as follows:

```
130 SUB TEST4(INTEGER A(*))
```

```
int test4( int NPAR, intptr a, dimptr ad )
```

Where *NPAR* is the number of arguments in the CALL, ***a*** is a pointer to the array data, and ***ad*** is a pointer to the array dimension table. The array data elements are stored in row-major order.

String Arguments

For string arguments the CSUB routine requires two pointers, one for the string data structure and one for the string dimension table. This is defined as follows:

```
140 SUB TEST5( C$ )
```

```
int test5( int NPAR, strptr c, dimptr cd )
```

Where ***c*** is a pointer to the string data, ***cd*** is a pointer to the string dimension table.

The string data is made up of two parts, the current string length followed by the string data. See the **csub.h** include file for detailed information about string data.

String Array Arguments

For string array arguments the CSUB routine requires two pointers, one for the string data structure and one for the string array dimension table. This is defined as follows:

```
150 SUB TEST6( D$(*) )
```

```
int test6( int NPAR, strptr d, dimptr dd)
```

Where ***d*** is a pointer to the string array data, and ***dd*** is a pointer to the string array dimension table. The string array elements are stored in row-major order.

Optional Arguments

The ***int value*** of NPAR is passed as the first item in the CSUB parameter list so the CSUB can know the number of arguments in the CALL statement. Omitted OPTIONAL arguments are passed as NULL pointers. The CSUB can use the value of NPAR to determine the number of passed arguments or it can test the argument pointer values before they are used. For example:

```
if( NPAR > 4 )
 *b = result;          /* OK for use */
else
 return( 143 );        /* missing OPTIONAL argument */

if( b )
 *b = result;          /* OK for use */
else
 return( 143 );        /* missing OPTIONAL argument */
```

This chapter described how HTBasic passes CALL arguments to the CSUB, how to define data and dimension table pointers, and how to handle OPTIONAL arguments.

Building a CSUB

This chapter presents the steps required to build a Windows 32-bit DLL CSUB for use with HTBasic for Windows.

There are six steps used to build a CSUB.

- Define the CALL interface.
- Create the CSUB prototype PROG file.
- Write the CSUB routines.
- Debug the CSUB in a stand-alone environment.
- Link the CSUB, helper routines, and CSUB header.
- Run the HTBasic CSUB utility.

Each of these steps is explained on the following pages.

Define the CALL Interface

The first step in building a CSUB is to define the CALL interface. This amounts to defining the number, type, and order of the CSUB arguments. They are defined using the SUB program line syntax and may be of type REAL, INTEGER, COMPLEX, or String. They may be simple variables or arrays.

Using the HTBasic SUB statement syntax, define the CSUB arguments that are required. INTEGER A, specifies a simple integer variable, whereas INTEGER A(*) specifies an integer array. A$, specifies a simple string variable, whereas A$(*) specifies a string array. This information will be used to create the CSUB prototype in the next step.

Create the CSUB Prototype

Once the CSUB arguments are defined, you can proceed to create the CSUB prototype PROG file. It provides the CSUB name and argument definitions that will be used later by the CSUB builder utility. It is created from within HTBasic using the EDIT mode as follows:

• Enter the SUB, any COM, and SUBEND program lines.
• Move back to the SUB line and press INSERT-LINE.
• Enter 1 END.

For example:

```
1  END
10 SUB Test12( INTEGER A(*), REAL B, C$ )
20 SUBEND
```

This creates a small main context and a SUB context named Test12. If you are defining several entry points in one CSUB context then continue to enter the additional SUB definitions after the first. For example:

```
30 SUB Test13( A$, INTEGER B, C )
40 SUBEND
```

After all the desired SUBs have been defined you must pre-run the program and then store it as a PROG file as follows:

• Press the STEP key or enter RUN to pre-run the SUB program.
• STORE the program to a PROG file:

STORE "your_routine.pro"

Remember that if you include COM statements in a SUB definition, you must enter the full COM definition in the main context to allow a prerun. For more information on using COM variables in CSUBs, see the "COM Variables" section.

Write the CSUB Routines

Once the CSUB prototype PROG file is created, you can proceed to write the CSUB routines. The basic outline of a CSUB routine in the C language is as follows:

```
/* Define Static Data Here */
static int saveit;
static char name[20];

int name( int NPAR, parameters.... ) {

          code to implement function

 return error;
}

static int other_routines( parameters.... ) {

          code to implement function

}
```

Remember that the C language routine receives the *value* of NPAR as the first argument and that it must return an error value. Define all global data and routines as ***static*** to make the DLL symbol table as small as possible.

CSUB Examples

Complete working C language CSUB examples are given later in this manual.

Routine Name

You may choose any routine name that is unique from any other DLL routine name in use in the Windows system.

Debug the CSUB

A Windows CSUB may be debugged using any debugger that can work with Windows 32-bit DLLs. Simple debugging may be done using the Windows sprintf and MessageBox functions.

For large amounts of debugging, it may be better to write a WinMain program and link it directly to the CSUB as a static object, and then debug the WinMain program with the integrated debugger available with most compilers.

Compiling the CSUB

The CSUB is compiled using a compiler capable of producing Windows 32- bit DLLs, such as Microsoft Visual C++ version 2.0 or above or Borland C++ version 4.0 or above. These 32-bit DLLs must be compatible with the Win32 calling standard.

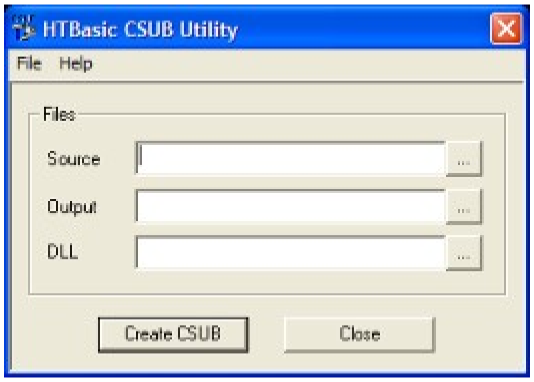
Link the CSUB

The routines in the CSUB are linked to become a Windows 32-bit DLL, by being made into a library and then converted to a DLL by combining the library and a definitions file using a DLL-builder. The definitions file should contain, as a minimum, a LIBRARY section naming the library and an EXPORTS section naming each CSUB entry point. See the example files TEST.DEF and TEST.MAK to see how this is done using the Microsoft 32-bit linker.

HTBasic CSUB Utility

The HTBasic CSUB utility accepts as input the prototype PROG file with one or more SUB program definitions and the name of the DLL that contains the routines. It produces an output PROG file that combines all the SUB program definitions and the DLL information into one CSUB context.

The CSUB utility is run from the Tools menu from within HTBasic.



Input:

Source file	.PRO - HTBasic PROG header file.
Output file	.CSB - file containing CSUB to be loaded into HTBasic for Windows.
DLL	DLL file name.

The CSUB is output to the PROG file by the utility. All the SUB prototype routines and the DLL information are combined and output as one CSUB context.

CSUB DLL Pathname

The CSUB DLL is stored in the CSUB definition and is used to load the DLL when the CSUB is executed. It is normally specified with a ***.DLL*** suffix. Use the exact case of the CSUB DLL file name.

When no path is specified, HTBasic will search for the DLL in the following sequence:

1) The home directory for HTBasic,
2) The current directory,
3) The Windows system directory
4) The Windows directory
5) The directories listed in the PATH environment variable.

Please refer to the *WIN32 Programmer's Reference* manual for more information on DLLs.

Loading and Storing the CSUB

The resulting CSUB can now be loaded into HTBasic memory and stored to a PROG file as if it were a normal SUB program. All CSUBs defined in one CSUB context will be treated as a group by the HTBasic LOADSUB and DELSUB statements. Use the name of the first CSUB to load or delete the entire CSUB context.

If you SAVE a program that contains a CSUB you will get just the CSUB program lines in the output file. When you attempt to GET the program, the CSUB program lines will be reported as syntax errors. To fix this problem, delete the CSUB lines and LOADSUB the CSUBs from a valid PROG file.

Remember that the Windows DLL file is now a part of the HTBasic program. It must be available on the computer system that is running the program that calls the CSUB.

Unexpected Operation

Once these steps have been completed, you are ready to test the new CSUB inside HTBasic. If the CSUB operates unexpectedly inside HTBasic, first check the link map for any library routines that might not operate correctly within the HTBasic CSUB environment. If you do not see any that look suspicious, perform further testing in a stand-alone environment to try and find the problem.

COM Variables

This chapter describes the *com_var* function, used to locate COM memory variables, the movable nature of COM data, and the use of COM statements in CSUB prototype definitions. Two example C programs are presented to demonstrate how to access COM variables.

You may access COM variables from within a CSUB. The *com_var* function is used to return a pointer to either the start of the specified COM variable data area, its dimension table, or the start of the specified COM data area. If the COM area is not defined or if the variable is not defined, or if the variable is not an array, a null pointer is returned. The function is defined as:

com_var Function

```
void *com_var(char *name, int varnum, int ptrtype);
```

The first argument is a pointer to the COM area name. The name is specified with the first character in upper case and the rest in lower case. The blank COM area name is specified as a single space.

The second argument is the COM variable number. Its value ranges from zero through the number of defined variables in the COM area. COM variables are numbered in left to right order starting with 1. If a zero is specified, the start of the COM data area is returned. If too large a variable number is specified, a null pointer is returned.

The third argument specifies the type of pointer to return. A zero specifies the variable data pointer and a one specifies the dimension table entry pointer. If a dimension table pointer is requested for a variable that is not a string or an array, a null pointer is returned.

Movable COM Data

Because a COM data area may be moved during a RUN, GET, or LOAD statement, you should always call *com_var* to get the current address before accessing the data area. Do not save the address for use in a later invocation of the CSUB.

Prototype COM Lines

Include COM program lines in the BASIC prototype definition only if you need to insure that the COM data area is left in memory between LOAD statements. COM program lines are specified after the SUB line and before the SUB END line. They will be displayed when the CSUB is listed. You must specify the full COM area definition in the Main program context to allow the SUB prototype to be prerun.

COM Examples

The following examples show how to access a COM area or individual COM variables from within a CSUB.

This example shows how to access a COM area from a C subprogram.

```
10  COM /Test/ B, INTEGER I(9), A$[100]
20  MAT I=(10)
30  CALL Tcom1
40  PRINT I(0)
50  END
60  SUB Tcom1
70   COM /Test/B,INTEGER I(*),A$
80   I(0)=I(2)*56
90  SUBEND
```

Prototype

```
1  COM /Test/ B,INTEGER I(9),A$[100]
2  END
10  SUB Tcom1
20    COM /Test/ B,INTEGER I(*),A$
30  SUBEND
```

C Program

```
#include "csub.h"

struct test             /* COM /Test/ data area def */
{
 T_FLT  b;             /* REAL B */
 T_INT  i[10];         /* INTEGER I(0:9) */
 T_SUBS l;             /* A$[100] current length */
 U_CHAR a[100];        /* A$ string data */
};

tcom1( npar )/* ACCESS FULL COM DATA AREA */
 int npar;             /* number of parameters */
{
 T_INT c;              /* working variable */
 struct test *t;       /* COM data area pointer */

 t = com_var("Test", 0, 0); /* get COM /Test/ data ptr */
 if( !t )              /* area found? */
   return( 47 );       /* no, area not found */
 c = t->i[2];          /* yes, get 3rd INTEGER element */
 t->i[0] = c * 56;     /* set value of first element */
 return( 0 );          /* no error */
}
```

The next example shows how to access individual COM variables and their array descriptions from within a C subprogram.

BASIC Program

```
10  COM /Test/ B, INTEGER I(9), A$[100]
20  MAT I=(1)
30  CALL TCOM2
40  PRINT I(0),I(1),I(2),A$
50  END
```

```
60  SUB TCOM2
70    COM /Test/ B, INTEGER I(*), A$
80    I(0)=RANK(I)
90    I(1)=BASE(I,1)
100   I(2)=SIZE(I,1)
110   A$="This is a test of strings"
120 SUBEND
```

Prototype

```
1  COM /Test/ B, INTEGER I(9), A$[100]
2  END
10  SUB Tcom2
20    COM /Test/ B, INTEGER I(*), A$
30  SUBEND
```

C Program

```
#include "csub.h"
#include <string.h>

tcom2( npar )   /* ACCESS SPECIFIC COM VARIABLES */
 int npar;             /* number of parameters */
{
 intptr  ip;           /* integer pointer */
 realptr dp;           /* real pointer */
 DIM    *ep;           /* dimension pointer */
 T_STR  *sp;           /* string pointer */

 dp = com_var("Test", 1, 0);   /* get B data pointer */
 ip = com_var("Test", 2, 0);   /* get I data pointer */
 ep = com_var("Test", 2, 1);   /* get I dim entry ptr */
 if( !dp || !ip || !ep)        /* variables found? */
   return( 47 );               /* no, variable not found */
 ip[0] = NOD(ep);              /* return number of subscripts */
 ip[1] = ep->sbs[0].base;      /* subscript base */
 ip[2] = ep->sbs[0].size;      /* and size values */

 sp = com_var("Test", 3, 0);   /* get A$ data pointer */
 ep = com_var("Test", 3, 1);   /* get A$ dim entry ptr */
 if( !sp || !ep )              /* variables found? */
   return( 47 );               /* no, variable not found */
 memcpy( sp->str, "This is a test of strings", 25);
 sp->clen = 25;                /* set the length */
 return( 0 );                  /* no error */
}
```

This chapter described the *com_var* function, used to locate COM memory variables, the movable nature of COM data, and the use of COM statements in CSUB prototype definitions. Two example C programs were presented that demonstrate how to access COM variables.

Display and Keyboard Routines

This chapter describes the internal HTBasic routines available to a CSUB to control the Display and Keyboard. The following table lists the display and keyboard support routines.

Routines

Routine Name	**Description**
CSB_VER	Version number of Table
kbdcrt_check	Check required Version
kbdcrt_clear_screen	CLEAR SCREEN
kbdcrt_controlcrt	CONTROL CRT
kbdcrt_controlkbd	CONTROL KBD
kbdcrt_crtreadstr	Read Output Area
kbdcrt_crtscroll	Scroll Display
kbdcrt_cursor	Control Cursor
kbdcrt_dispstr	Write to Display
kbdcrt_printstr	Print to Output Area
kbdcrt_readkbd	Read the KBD$ buffer
kbdcrt_scrolldn	Scroll Output Area Down
kbdcrt_scrollup	Scroll Output Area Up
kbdcrt_statuscrt	STATUS CRT
kbdcrt_statuskbd	STATUS KBD
kbdcrt_systemd	SYSTEM$ Function

These routines are called using the C language calling conventions. Most routines return either an HTBasic error code or a value of zero for no error. A description of each of the routines and a C language calling example are presented on the following pages.

Version

```
CSB_VER
```

Description

This macro returns the version number of the keyboard and display routine jump table. It can be used by the CSUB to determine if it is running on a version of HTBasic that includes the required defined entries.

Example

```
if( CSUB_VER < REQUIRED )
 return( 2009 );     /* report the error */
```

Check Version

```
void kbdcrt_check( );
```

Description

This routine checks the version number of the keyboard and display routine jump table. If the running HTBasic does not have the required version of the jump table it returns an error number 2009 to HTBasic.

Example

kbdcrt_check( );

Clear Screen

```
void kbdcrt_clear_screen( );
```

Description

This routine performs the same action as the CLR SCR key and the CLEAR SCREEN statement. The return value is either an HTBasic error code or a zero indicating no error.

Example

kbdcrt_clear_screen( );

CRT Control

```
int kbdcrt_controlcrt( int reg, int value );
```

Where:
  reg - register number
  value - value to write to control register

Description

This routine performs the same action as the CONTROL CRT statement. The return value is either an HTBasic error code or a zero indicating no error.

Example

int error;
error = kbdcrt_controlcrt(13, 15);

See Also

kbdcrt_statuscrt, kbdcrt_statuskbd

KBD Control

```
int kbdcrt_controlkbd( int reg, int value );
```

Where:
  reg   - register number
  value - value to write to control register

Description

This routine performs the same action as the CONTROL KBD statement. The return value is either a zero indicating no errors or the HTBasic error code.

Example

```
int error;
error = kbdcrt_controlkbd( 3, 50 );
```

See Also

kbdcrt_statuskbd, kbdcrt_statuscrt

Read Output Area

```
int kbdcrt_crtreadstr( char *buf, int max, int *len );
```

Where:
  buf - string buffer pointer
  max - length of string buffer
    0x10000000 - return characters and attributes flag
  len - number of bytes written to buf

Description

This routine performs the same action as ENTER CRT;A$. The non-blank 1, 12, 0);

See Also

kbdcrt_scrolldn, kbdcrt_scrollup

Cursor

```
int kbdcrt_cursor( int col, int line, int type );
```

Where:
  col - alpha character column number
  line - alpha character row number
  type - cursor type 0=Normal, 1=Insert, 2=Off

Description

This routine erases the current cursor and places a new cursor at the specified character location. The values of col and line are the same as specified by PRINT TABXY(col,line). The return value is either a zero for no errors or an HTBasic error code.

Any action generated by HTBasic which affects the cursor will move the cursor back to its normal location. You may wish to trap all keyboard input with ON KBD while using this routine.

Example

```
int error;
error = kbdcrt_cursor( 1, 10, 0 );
```

Write to Display

```
int kbdcrt_dispstr( int col, int line, char *buf, int len, int clr );
```

Where:
```
  col  - alpha character column number
  line - alpha character row number
  buf  - string pointer
  len  - string length
  clr  - color value or enhancement attributes buffer flag
        0-255 = text color value (See description)
          -1 = buf is a two byte string, first byte character,
               second byte color and enhancement attributes
               0x00FF - color value (See description)
               0x0100 - inverse video attribute bit
               0x0200 - blink attribute bit
               0x0400 - underline attribute bit
```

Description

This routine writes a string directly to the display at the specified text location in either the requested color or the specified attributes. Because this routine bypasses the normal Output Area buffer routines, characters cannot be read back with the **kbdcrt_crtreadstr** routine and can not be scrolled back onto the screen with the kbdcrt_scrolldn and **kbdcrt_scrollup** routines. Use the **kbdcrt_printstr** routine to enter characters into the extended Output Area buffer.

The values of col and line are the same as specified by PRINT TABXY(col,line). If the color value is -1 then the string contains the character to display in the first byte and the enhancement and color attributes in the second byte.

The return value is either a zero for no errors or an HTBasic error code.

Example

```
int error;
error = kbdcrt_dispstr( 10, 10, "This is a test", 14, 1 );
```

See Also

kbdcrt_printstr

Print String

```
void kbdcrt_printstr( char *buf, int len );
```

Where:
  buf - string pointer
  len - string length
    0x1000000 - Raw character and color/attribute flag

Description

This routine writes a string to the scrollable Output Area buffer just like a PRINT statement. The characters are displayed at the current PRINT position using the current color and enhancement attributes. Control characters and enhancement and color characters (128-143) are handled just like during a PRINT.

If the Raw flag is set in the string length then the string contains two bytes for each display character. The character is in the first byte and the color and enhancement attributes are in the second byte. This format is the same as used by the **kbdcrt_dispstr** routine. The string is copied directly into the Output Area buffer and the characters are displayed with the specified color and enhancement attributes.

Example

kbdcrt_printstr("This is a test", 14);

See Also

kbdcrt_dispstr

Read KBD$ Buffer

```
int kbdcrt_readkbd( char *buf, int max, int *len );
```

Where:
   buf - buffer pointer
   max - length of buf
   len - number of bytes read

Description

This routine returns the contents and then clears the KBD$ buffer. The characters are returned in the buffer pointed to by *buf*. It is not null terminated. The number of characters written is returned in *len*. The return value is either a zero indicating no errors or an HTBasic error code.

Specify the ON KBD statement before entering the CSUB. If you do not wish closure keys to block the keyboard input specify the ON KBD ALL option.

```
int  error;
int  len;
char buf[160];
error = readkbd( buf, 160, &len );
```

See Also

crtread

Scroll Output Down

```
void kbdcrt_scrolldn( );
```

Description

This routine scrolls the Output Area down one line. This also scrolls text from and to the extended Output Area.

Example

kbdcrt_scrolldn( );

See Also

kbdcrt_scrollup

Scroll Output Up

```
void kbdcrt_scrollup( );
```

Description

This routine scrolls the Output Area up one line. This also scrolls text from and to the extended Output Area.

Example

kbdcrt_scrollup( );

See Also

kbdcrt_scrolldn

CRT Status

int kbdcrt_statuscrt(int reg, int *value);

Where:
  reg   - register number
  value - pointer to location for status register value

Description

This routine performs the same action as the STATUS CRT statement. The current value of the specified register is returned. The return value is either a zero indicating no errors or an HTBasic error code.

Example

```
int error;
int value;
error = kbdcrt_statuscrt( 13, &value );
```

See Also

kbdcrt_controlcrt, kbdcrt_statuskbd

KBD Status

int kbdcrt_statuskbd(int reg, int *value);

Where:
  reg   - register number to read
  value - pointer to location for status register value

Description

This routine performs the same action as the STATUS KBD statement. The current value of the specified register is returned. The return value is either a zero indicating no errors or an HTBasic error code.

Example

```
int error;
int value;
error = kbdcrt_statuskbd( 3, &value );
```

See Also

kbdcrt_controlkbd, kbdcrt_statuscrt

SYSTEM$ Function

```
int systemd( char *arg, char *buf, int max, int *len );
```

Where:
  arg - pointer to null terminated request string
  buf - string buffer pointer
  max - maximum buffer length
  len - returned string length

Description

This routine performs a SYSTEM$( ) function and returns a null terminated response in the string buffer and sets *len* to the length of the response. The return value is either a zero, indicating no errors, or an HTBasic error code.

Example

```
int  error;
char buf[160];
int  len;
error = systemd( "VERSION:HTB", buf, 160, &len );
```

C Language Examples

This chapter provides C language CSUB examples for argument types, INTEGER, REAL, Arrays, Strings, and COM data, and an example that defines several CSUBs combined into one CSUB context. These examples are installed into CSUB Toolkit directory HTBWIN\CSUB\EXAMPLE

C Overview

C language CSUBs provide few, if any restrictions or requirements on the writer. The compiler takes care of most of the implementation details. The CSUB routine must return an error value. A value of zero is used for no error, non-zero values are interpreted as standard BASIC error values as given in Appendix A of the *HTBasic Reference Manual*.

By examining the following programs you can learn the required details to implement most CSUBs. If you have a large number of routines, make sure you examine the last example that demonstrates how to package several routines into one CSUB context.

INTEGER Variables

An INTEGER variable is defined as a *short int* in C. You are passed a pointer to the data location. This example shows how to access an INTEGER from a C routine.

BASIC Program

```
10 INTEGER I
20 I=256
30 CALL Imp3( I )
40 PRINT I
50 END
60 SUB Imp3( INTEGER A )
70   A = A * 3
80 SUBEND
```

Prototype

```
1  END
10 SUB Imp3( INTEGER A )
20 SUBEND
```

C Program

```
#include "csub.h"
imp3( npar, a )
 int npar;             /* number of parameters */
 intptr a;             /* integer data pointer */
{
 *a = *a * 3;          /* return a new value */
 return( 0 );          /* no errors */
}
```

REAL Variables

A REAL variable is defined as a *double* in C. You are passed a pointer to the data location. This example shows how to access a REAL variable from a C routine.

BASIC Program

```
10 REAL R
20 R=512
30 CALL Rdv4( R )
40 PRINT R
50 END
60 SUB Rdv4( A )
70   A = A / 4.0
80 SUBEND
```

Prototype

```
1  END
10 SUB Rdv4( A )
20 SUBEND
```

C Program

```
#include "csub.h"
rdv4( npar, a )
 int  npar;            /* number of parameters */
 realptr a;            /* real data pointer */
{
 *a = *a / 4.0;        /* return a new value */
 return( 0 );          /* no errors */
}
```

Numeric Arrays

A numeric array variable generates two pointers. The first points to the first element of the data array and the second points to the array dimension table. The array elements are stored in row-major order.

This example shows how to access an INTEGER array from a C routine.

BASIC Program

```
10  INTEGER I(20),S
20  MAT I=(2)
30  CALL Isum( I(*),S )
40  PRINT S
50  END
60  SUB Isum( INTEGER A(*), S )
70   S=0
80   FOR I=0 to 20
90     S=S+A(I)
100  NEXT I
110 SUBEND
```

Prototype

```
1  END
10 SUB Isum( INTEGER A(*), S )
20 SUBEND
```

C Program

```
#include "csub.h"
isum( npar, a, d, s )   /* sum integer array */
 int npar; /* number of parameters */
 intptr a;             /* array data pointer */
 dimptr d;             /* array dim entry pointer */
 intptr s;             /* integer sum pointer */
{
 int i;                /* element counter */
 int max = d->cae;    /* get number of elements */
 int sum = 0L;         /* clear the sum */

 for(i = 0; i < max; ++i) /* scan all elements */
   sum += a[i];
 *s = (T_INT)sum;      /* return the sum value */
 return( 0 );          /* no error */
}
```

String Variables

A string variable generates two pointers. The first points to the string data structure and the second points to the dimension table. For simple strings only the element length is defined in the dimension structure. It specifies the maximum dimensioned length of the string data.

This example sets a simple string variable from a C routine.

BASIC Program

```
10 DIM S$[40]
20 CALL Strint( S$ )
30 PRINT LEN(S$),S$
40 END
50 SUB Strint( A$ )
60   A$ = "This is a test of strings"
70 SUBEND
```

Prototype

```
1  END
10 SUB Strint( A$ )
20 SUBEND
```

C Program

```
#include "csub.h"
#include <string.h>

strint( npar, a, d )
 int npar;                     /* number of parameters */
 strptr a;                     /* string data pointer */
 dimptr d;                     /* string dimension pointer */
{
 if( d->elen < 25 )            /* check variable length */
   return( 18 );               /* too small, return error */
 memcpy( a->str, "This is a test of strings", 25);
 a->clen = 25;                 /* set the length */
 return( 0 );                  /* no error */
}
```

Note: if the strcpy( ) function is used, remember that it will append a NULL.

String Arrays

A string array generates two pointers. The first points to the string data area and the second points to the array dimension table. The string data area contains the string elements in row-major order.

A string element is made up of a *short int* current string length followed by the string data. Each string element is allocated with enough string data space to contain the maximum dimensioned string length rounded up to an even length.

This example demonstrates setting a string array element from a C routine.

BASIC Program

```
10 DIM S$(10)[40]
20 CALL Streint( S$(*), 4)
30 PRINT LEN(S$(4)),S$(4)
40 END
50 SUB Streint( S$(*), INTEGER E )
60   S$(E) = "This is a test of strings"
70 SUBEND
```

Prototype

```
1  END
10 SUB Streint( A$(*), INTEGER E )
20 SUBEND
```

C Program

```
#include "csub.h"
#include <string.h>

streint( npar, a, d, e )
 int npar;              /* number of parameters */
 strptr a;              /* string data pointer */
 dimptr d;              /* dimension pointer */
 intptr e;              /* element number to set */
{
 strptr t;              /* string element pointer */
 int maxsize = d->elen;/* get element size */

 if( *e > d->cae )      /* check element number */
   return( 18 );        /* too large, return error */

 if( maxsize < 25 )     /* check variable length */
   return( 18 );        /* too small, return error */

/* Notice the string element address calculation */
 if( maxsize & 1)       /* if odd length */
   ++maxsize;           /* round up to even length */
 t = (strptr)((U_CHAR *)a + ((maxsize + sizeof(T_SUBS)) * *e));

 memcpy( t->str, "This is a test of strings", 25);
 t->clen = 25;          /* set the length */
 return( 0 );           /* no error */
}
```

Note: Don't use the strcpy( ) function because it will append an un-wanted NULL.

Several Routines in one CSUB Context

This example shows how to combine several routines into one CSUB context. It shows five C routines and five HTBasic SUB routines all defined in one CSUB context. The entire CSUB context can be loaded by one LOADSUB and deleted by one DELSUB statement that mentions the first CSUB's name.

Prototype SUB program lines:

BASIC Program

```
10 ! prototype SUB program lines:
20    END
30    SUB Integ(INTEGER A)
40    SUBEND
50    SUB Real(A)
60    SUBEND
70    SUB Isum(INTEGER A(*),S)
80    SUBEND
90    SUB Strint(A$)
100   SUBEND
110   SUB Streint(A$(*),INTEGER E)
120   SUBEND
```

C Programs

```
#include "csub.h"
#include <string.h>

integ( npar, a )         /* INTEGER C SUB routine */
 int npar;              /* number of parameters */
 intptr a;              /* integer data pointer */
{
 *a *= 3;               /* return the new value */
 return( 0 );           /* no errors */
}
real( npar, a )       /* REAL C SUB routine */
 int npar;                      /* number of parameters */
 realptr a;                     /* real data pointer */
{
 *a /= 4.0;                     /* return a new value */
 return( 0 );                   /* no errors */
}


isum( npar, a, d, s )   /* Integer Array C SUB routine */
  int npar;                     /* number of parameters */
  intptr a;                     /* array data pointer */
  dimptr d;                     /* array dim pointer */
  intptr s;                     /* sum data pointer */
{
 int i;                         /* element counter */
 int max = d->cae;              /* get number of elements */
 int sum = 0L;                  /* clear the sum */

 for(i = 0; i < max; ++i)       /* scan all elements of array */
   sum += a[i];
 *s = (T_INT)sum;               /* return the sum value */
 return( 0 );                   /* no error */
}

strint( npar, a, d )    /* String C SUB routine */
 int npar;                     /* number of parameters */
 strptr a;                     /* string data pointer */
 dimptr d;                     /* string dimension pointer */
{
 if( d->elen < 25 )            /* check variable length */
   return( 18 );               /* too small, return error */
 memcpy( a->str, "This is a test of strings", 25);
 a->clen = 25;                 /* set the length */
 return( 0 );                  /* no error */
}

streint( npar, a, d, e )        /* String Array C SUB routine */
 int npar;                     /* number of parameters */
 strptr a;                     /* string data pointer */
 dimptr d;                     /* dimension pointer */
 intptr e;                     /* element number to set */
{
 strptr t;                     /* string element pointer */
 int maxsize = d->elen;        /* get element size */
 int zbe;                      /* zero base element number */

 zbe = *e - d->sbs[0].base;    /* zero based element number */
 if( zbe > d-cae )             /* check element number */
   return( 18 );               /* too large, return error */

 if( maxsize > 25 )            /* check variable length */
   return( 18 );               /* too small, return error */

/*  Notice the string element address calculation */
 if( maxsize & 1)              /* if odd length */
   ++maxsize;                  /* round up to next even length */
 t = (strptr)((U_CHAR *)a + ((maxsize + sizeof(T_SUBS)) * zbe));

 memcpy( t->str, "This is a test of strings", 25);
 t->clen = 25;                 /* set the length */
 return( 0 );                  /* no error */
}
```

All the preceding C language examples are on the distribution media. The files used to create and test the CSUB are included for each example. You may use these routines as a model to implement CSUB routines.

Include File Listings

This chapter lists the include files provided with the CSUB Toolkit. They define the various HTBasic data types, structures, keyboard and display routines. Include the csubw.h file. For example:

```
#include "csub.h"
```

For C language programs that require floating point math include the *math.h* file if creating a CSUB in the C program file. For example:

```
#include "cmath.h"
```

Windows C Include File

```
/* CSUBW.H — HTBasic Windows CSUB C Language Include File, 6.0 */
/* (c) Copyright 1989-2002 TransEra Corp. All Rights Reserved */

#define MX_STLEN 32767  /* maximum string length */
#define MX_SUBS  6      /* maximum number of subscripts */

typedef unsigned char U_CHAR;
typedef unsigned short int U_SHORT;
typedef unsigned long U_LONG;

typedef short int T_SUBS;/* subscript value */

typedef struct subs     /* SUBSCRIPT BOUNDS DATA */
{
 T_SUBS  base;         /* subscript lower bound value */
 T_SUBS  size;         /* number of elements */
} SUBS;

typedef struct dim      /* DIMENSION TABLE ENTRY */
{
 U_LONG  elen;         /* Element LENgth (string max len) (bytes) */
 U_LONG  tae;          /* Total Allocated Elements (upper byte nod) */
 U_LONG  cae;          /* Current Allocated Elements */
 SUBS sbs[MX_SUBS];    /* Subscript Bounds */
}   DIM;

/* The upper byte of dim.tae is used to store the number of dimensions */

#define NOD(d) ( ((d)->tae > 24) &0xFF) )  /* number of dimensions */
#define TAE(d) ( (d)->tae & 0x00FFFFFF )  /* total array elements */

typedef short int T_INT;        /* BASIC INTEGER value */
typedef double    T_FLT;        /* BASIC REAL value */

typedef struct t_cpx            /* BASIC COMPLEX value */
{
 T_FLT r;                      /* real part */
 T_FLT i;                      /*imaginary part */
}   T_CPX;

typedef struct t_str            /* BASIC STRING Data */
{
 T_SUBS clen;                  /* current string length */
 U_CHAR str[MX_STLEN];         /* string data */
}   T_STR;

typedef DIM    *dimptr;         /* pointer to Dimension Structure */
typedef T_INT  *intptr;         /* pointer to INTEGER Value */
typedef T_FLT  *realptr;        /* pointer to REAL Value */
typedef T_CPX  *cpxptr;         /* pointer to COMPLEX Value */
typedef T_STR  *strptr;         /* pointer to String Structure */

_declspec(dllexport) void *(*_fnd_var)(); /* COM memory finder */
#define com_var(n,v,p)  (*_fnd_var)( (char *)(n), (int)(v), (int)(p) )
```

```
/* CSUB Jump Table Definition */
_declspec(dllexport) int (** _cjmptbl)():  /* ptr to csub jump table */

#define CSB_VER ((long)_csbjtbl[ 0])

#define kbdcrt_check  if( CSB_VER < 16 ) return( 2009 )

/* Keyboard and Display Routines */

#define kbdcrt_clear_screen      (*_csbjtbl[ 2])
#define kbdcrt_controlcrt(r,v)   (*_csbjtbl[ 3])((int)(r),(int)(v))
#define kbdcrt_controlkbd(r,v)   (*_csbjtbl[ 4])((int)(r),(int)(v))
#define kbdcrt_crtreadstr(b,m,l) (*_csbjtbl[ 5]) \
                                  ((char*)(b),(int)(m),(int*)(l))
#define kbdcrt_crtscroll(f,l,d)  (*_csbjtbl[ 6]) \
                                  ((int)(f),(int)(l),(int)(d))
#define kbdcrt_cursor(c,l,t)     (*_csbjtbl[ 7]) \
                                  ((int)(c),(int)(l),(int)(t))
#define kbdcrt_dispstr(c,l,b,s,a) (*_csbjtbl[ 8]) \
                                  ((int)(c),(int)(l),(char *)(b), \
                                   (int)(s),(int)(a))
#define kbdcrt_printstr(b,l)     (*_csbjtbl[ 9]) \
                                  ((char*)(b),(int)(l))
#define kbdcrt_readkbd(b,m,l)    (*_csbjtbl[10]) \
                                  ((char*)(b),(int)(m),(int*)(l))
#define kbdcrt_scrolldn          (*_csbjtbl[11])
#define kbdcrt_scrollup          (*_csbjtbl[12])
#define kbdcrt_statuscrt(r,v)    (*_csbjtbl[13])((int)(r),(int*)(v))
#define kbdcrt_statuskbd(r,v)    (*_csbjtbl[14])((int)(r),(int*)(v))
#define kbdcrt_systemd(a,b,m,l)  (*_csbjtbl[15]) \
                                 ((char*)(a),(int)(b),(int)(m),(int*)(l))

/* end of CSUBW.H */
```

{ewl RoboEx32.dll, WinHelp2000, }