Z·WORLD

Dynamic C® 32

for Zilog Z180 microprocessors

Version 6.x

Integrated C Development System

Function Reference

019-0082 • 020329-D

Dynamic C 32 v. 6.x Function Reference

Part Number 019-0082 • 020329 - D • Printed in U.S.A.

Copyright

Trademarks

Notice to Users

When a system failure may cause serious consequences, protecting life and property against such consequences with a backup system or safety device is essential. The buyer agrees that protection against consequences resulting from system failure is the buyer's responsibility.

This device is not approved for life-support or medical systems.

Company Address

Z-World, Inc.
2900 Spafford Street
Davis, California 95616-6800 USA

Telephone:	(530) 757-3737
Facsimile:	(530) 753-5141
Web Site:	http://www.zworld.com
E-Mail:	zworld@zworld.com

TABLE OF CONTENTS

About This Manual

Z-World customers develop software for their programmable controllers using Z-World's Dynamic C 32 development system running on an IBM-compatible PC. The controller is connected to a COM port on the PC, usually COM2, which by default operates at 19,200 bps.

Features which were formerly available only in the Deluxe version are now standard. Dynamic C 32 supports programs with up to 512K in ROM (code and constants) and 512K in RAM (variable data), with full access to extended memory.

The Three Manuals

Dynamic C 32 is documented with three reference manuals:

- Dynamic C 32 Function Reference.
- Dynamic C 32 Technical Reference
- Dynamic C 32 Application Frameworks

This manual contains descriptions of all the function libraries on the Dynamic C disk and all the functions in those libraries.

The Technical Reference manual describes how to use the Dynamic C development system to write software for a Z-World programmable controller.

The Application Frameworks manual discusses various topics in depth. These topics include the use of the Z-World real-time kernel, costatements, function chaining, and serial communication.

Please read release notes and updates for late-breaking information about Z-World products and Dynamic C.

Assumptions

Assumptions are made regarding the user's knowledge and experience in the following areas.

- Understanding of the basics of operating a software program and editing files under Windows on a PC.
- Knowledge of the basics of C programming. Dynamic C is not the same as standard C.

For a full treatment of C, refer to the following texts.

The C Programming Language by Kernighan and Ritchie
C: A Reference Manual by Harbison and Steel

- Knowledge of basic Z180 assembly language and architecture.

For documentation from Zilog, refer to the following texts.

Z180 MPU User's Manual
Z180 Serial Communication Controllers
Z80 Microprocessor Family User's Manual

Acronyms

Table 1 lists the acronyms that may be used in this manual.

Table 1. Acronyms

Acronym	Meaning
EPROM	Erasable Programmable Read-Only Memory
EEPROM	Electronically Erasable Programmable Read-Only Memory
LCD	Liquid Crystal Display
LED	Light-Emitting Diode
NMI	Nonmaskable Interrupt
PIO	Parallel Input/Output Circuit (Individually Programmable Input/Output)
PRT	Programmable Reload Timer
RAM	Random Access Memory
RTC	Real-Time Clock
SIB	Serial Interface Board
SRAM	Static Random Access Memory
UART	Universal Asynchronous Receiver Transmitter

Icons

Table 2 displays and defines icons that may be used in this manual.

Table 2. Icons

Icon	Meaning	Icon	Meaning
👓	Refer to or see	✏	Note
☎	Please contact	Tip	Tip
⚠	Caution	⚡	High Voltage
FD	Factory Default		

Conventions

Table 3 lists and defines typographic conventions that may be used in this manual.

Table 3. Typographical Conventions

Example	Description
`while`	Courier font (bold) indicates a program, a fragment of a program, or a Dynamic C keyword or phrase.
`// IN-01…`	Program comments are written in Courier font, plain face.
Italics	Indicates that something should be typed instead of the italicized words (e.g., in place of *filename*, type a file's name).
Edit	Sans serif font (bold) signifies a menu or menu selection.
. . .	An ellipsis indicates that (1) irrelevant program text is omitted for brevity or that (2) preceding program text may be repeated indefinitely.
[ ]	Brackets in a C function's definition or program segment indicate that the enclosed directive is optional.
< >	Angle brackets occasionally enclose classes of terms.
A | b | c	A vertical bar indicates that a choice should be made from among the items listed.

CHAPTER 1: GENERAL SUPPORT LIBRARIES

The libraries described in Chapter 1 include standard C string and math functions in addition to general support functions specific to Z-World's controllers.

Global Initialization

Global initialization is an important but unclassifiable topic, and is described here. Your program can initialize variables and take initialization action (of any complexity) if you do the following:

1. Incorporate **`_GLOBAL_INIT`** segments in your functions:

```
void init_ios();

int my_func( void* thing ){
   int table[10],j;
   float x,y;
      ...
   segchain _GLOBAL_INIT{
      for( j=0; j<10; j++ ){ table[j] = 10-j; }
      x = y = 0.781;
      init_ios();
   }
      ...
}
```

2. Make a call to the function chain **`_GLOBAL_INIT`** at the start of main.

When your program starts (from scratch or because of a hardware reset) the call to **`_GLOBAL_INIT`** performs the initialization for all **`_GLOBAL_INIT`** segments throughout your program (including libraries). The name **`_GLOBAL_INIT`** is not the name of a library function. However, there is a function **`GLOBAL_INIT`** in **`VDRIVER.LIB`**. If you call **`VdInit`**, i.e., you invoke the virtual driver, **`VdInit`** does global initialization for you. You need not do it yourself. The function **`uplc_init`** also calls **`_GLOBAL_INIT`**.

BIOS Functions

These functions reside in BIOS. The source code is provided for your convenience. To override BIOS function, use

`#kill` *functionname*

at the beginning of your user program and redefine the function.

- **`unsigned inport( unsigned port )`**

  Reads a value from the specified I/O port. This may be an internal Z180 register, or it may access external hardware. Refer to the controller reference manual for a list of I/O ports.

  The function returns the value from the I/O port in lower byte, and zero in upper byte.

- **`void outport( unsigned port, unsigned value )`**

  Writes **`value`** to I/O port. This may be an internal Z180 register, or it may access external hardware. Refer to your controller reference manual for a list of I/O ports.

- **`int ee_rd( int address )`**

  Reads value from EEPROM at specified address. The function returns EEPROM data (0–255) if successful. It returns a negative value if unable to read the EEPROM.

- **`int ee_wr( int address, char value )`**

  Writes value to EEPROM at specified address. The function returns 0 if successful, otherwise a negative value if unable to write the EEPROM.

- **`void di( void )`**

  Disables interrupts. Use **`DI`** for better efficiency.

- **`void DI( void )`**

  Disables interrupts. Dynamic C expands this call in-line.

- **`void ei( void )`**

  Enables interrupts. Use **`EI`** for better efficiency.

- **`void EI( void )`**

  Enables interrupts. Dynamic C expands this call in-line.

- **`int iff( void )`**

  Returns the state of the Z180 interrupt mask. If zero, interrupts are off. Otherwise, interrupts are on.

- **`unsigned bit( void* address, unsigned bit )`**

  Reads the value of the specified **`bit`** at memory address. The **`bit`** may be from 0 to 31. Use **`BIT`** (upper case) for in-line expansion of this call. This is equivalent to the following expression:

  ```
  (*(long *) address >> bit) & 1
  ```

  The function returns 1 if specified **`bit`** is set; 0 if **`bit`** is clear.

- **`unsigned BIT( void *address, unsigned bit )`**

  Reads the value of the specified **`bit`** at memory address. The bit may be from 0 to 31. Dynamic C will attempt to expand this call in-line. This is equivalent to the following expression:

  ```
  (*(long *) address >> bit) & 1
  ```

  The function returns 1 if specified **`bit`** is set, and 0 if **`bit`** is clear.

- **`void set( void *address, unsigned bit )`**

  Sets the specified **`bit`** at memory address to 1. The **`bit`** may be from 0 to 31. Use **`SET`** (upper case) for in-line expansion of this call. This is equivalent to the following expression:

  ```
  *(long *) address |= 1L << bit
  ```

- **`void SET( void *address, unsigned bit )`**

  Sets the specified **`bit`** at memory address to 1. The **`bit`** may be from 0 to 31. Dynamic C will attempt to expand this call in-line. This is equivalent to the following expression:

  ```
  *(long *) address |= 1L << bit
  ```

- **`void res( void *address, unsigned bit )`**

  Clears specified **`bit`** at memory address to 0. **`bit`** may be from 0 to 31. Use **`RES`** (upper case) for in-line expansion of this call. This is equivalent to the following expression:

  ```
  *(long*)address &= ~(1L << bit)
  ```

- **`void RES( void *address, unsigned bit )`**

  Clears specified **`bit`** at memory address to 0. **`bit`** may be from 0 to 31. Dynamic C will attempt to expand this call in-line. This is equivalent to the following expression:

  ```
  *(long *) address &= ~(1L << bit)
  ```

- **`unsigned IBIT( unsigned port, unsigned bit )`**

  Reads the I/O port and returns the value of the specified **`bit`**. The bit may be from 0 to 7. The port may be an internal Z180 register, or it may access external hardware. Refer to your controller reference manual for a list of I/O ports. The function returns 1 if the specified **`bit`** is set, and 0 if the **`bit`** is clear.

- **`void ISET( unsigned port, unsigned bit )`**

  Sets the specified **`bit`** of the I/O port to 1. The **`bit`** may be from 0 to 7. The port may be an internal Z180 register, or it may access external hardware. The function generates code like the following:

  ```
      in   a,(c)
      set  bit,a
      out  (c),a
  ```

  Refer to the controller reference manual for a list of I/O ports.

- **`void IRES( unsigned port, unsigned bit )`**

  Resets the specified bit of the I/O port to 0. The **`bit`** may be from 0 to 7. The **`port`** may be an internal Z180 register, or it may access external hardware. The function generates code like the following:

  ```
  in   a,(c)
  set  bit,a
  out  (c),a
  ```

  Refer to the controller reference manual for a list of I/O ports.

- **`void hitwd( void )`**

  "Hits" the watchdog timer, postponing a hardware reset for approximately 1.2–1.6 seconds (the value depends on hardware). Unless the watchdog timer is disabled, the program must call this function periodically. Otherwise, the controller resets automatically. This allows the controller to recover from errors that cause the program to enter an infinite loop. If the virtual driver is enabled, it will call **`hitwd`** in the background but provide virtual watchdogs in its place. See **`VdWdogHit`** for more information. For information about setting jumpers to enable/disable the watchdog (not available on all boards), refer to the controller reference manual.

- **`int wderror( void )`**

  Determines if the previous reset was caused by the watchdog timer. This feature is not available on all boards. Refer to the controller reference manual for more information.

  The function returns a positive nonzero value if the watchdog caused the last reset and zero if not. It returns a negative value if the feature is not supported.

- **`void intrmode_0( void )`**

  Sets Z180 interrupt mode to 0. The default mode for Dynamic C is Mode 2. Do not select another mode unless the interrupts for all peripheral devices using Mode 2 interrupts have been disabled.

- **`void intrmode_1( void )`**

  Sets Z180 interrupt mode to 1. The default mode for Dynamic C is Mode 2. Do not select another mode unless the interrupts for all peripheral devices using Mode 2 interrupts have been disabled.

  The function returns None.

- **`void intrmode_2( void )`**

  Sets Z180 interrupt mode to 2. This is the default mode for Dynamic C. Do not select another mode unless the interrupts for all peripheral devices using Mode 2 interrupts have been disabled.

- **`void runwatch( void )`**

  Allows Dynamic C to update watch expressions. Calling **`runwatch`** periodically enables evaluation of watch expressions while the program is running. Watch expressions are always evaluated when the program is stopped.

- **`int kbhit( void )`**

  Detects keystrokes in the Dynamic C **STDIO** window. The function returns nonzero if a key has been pressed, and zero otherwise.

- **`void exit( int exitcode )`**

  Stops the program and returns **`exitcode`** to Dynamic C. Dynamic C uses code values above 128 for run-time errors. When not debugging, this function causes a watchdog time-out if the watchdog is enabled.

  The function does not return.

- **`unsigned sysclock( void )`**

  Returns the system clock speed in units of 1200 Hz. Some common clock speeds and the corresponding **`sysclock`** values are listed below.

6.144 MHz	0x1400 (5120)	9.126 MHz	0x1E00 (7680)
12.288 MHz	0x2800 (10,240)	18.432 MHz	0x3C00 (15,360)

- **`int powerlo( void )`**

  It is possible for the supply voltage to drop low enough to generate a power-fail interrupt, but then return to normal without ever dropping low enough to reset the board. Call this routine from an NMI (power-fail) interrupt handler to determine if power has returned. Refer to the controller reference manual to find out whether this feature is supported. The function returns 1 if voltage is below the NMI level, and 0 otherwise.

COSTATE.LIB

These functions support cooperative multitasking.

- **`void CoBegin( CoData *cd )`**

  `CoBegin` initializes a **`CoData`** structure. The INIT flag is set, but the STOPPED flag is cleared.

- **`void CoReset( CoData *cd )`**

  `CoReset` resets a **`CoData`** structure. The STOPPED and INIT flags are *both* set.

- **`void CoPause( CoData *cd )`**

  `CoPause` pauses a **`CoData`** structure. The STOPPED flag is set, but the INIT flag is cleared.

- **`void CoResume( CoData *cd )`**

  `CoResume` resumes a **`CoData`** structure. The STOPPED and INIT flags are both cleared.

- **`int isCoDone( CoData *cd )`**

  The function **`isCoDone`** returns true (1) if both the STOPPED and INIT flags are set. It returns 0 otherwise.

- **`int isCoRunning( CoData *cd )`**

  The function **`isCoRunning`** returns true (1) if the STOPPED flag is not set. It returns 0 otherwise.

CTYPE.LIB

- **`int toupper( int c )`**

  Converts character **`c`** to its uppercase equivalent.

- **`int tolower( int c )`**

  Converts character **`c`** to its lowercase equivalent.

- **`int islower( int c )`**

  Returns nonzero if **`c`** is a lowercase character; zero otherwise.

- **`int isupper( int c )`**

  Returns nonzero if **`c`** is an uppercase character; zero otherwise.

- **`int isdigit( int c )`**

  Returns nonzero if **`c`** is an ASCII digit (0–9); zero otherwise.

- **`int isxdigit( int c )`**

  Returns nonzero if **`c`** is a hexadecimal digit (0–9, a–f,. A–F); zero otherwise.

- **`int ispunct( int c )`**

  Returns nonzero if **`c`** is a punctuation mark; zero otherwise.

- **`int isspace( int c )`**

  Returns nonzero if **`c`** is a blank, tab, new line, or form feed; zero otherwise.

- **`int isprint( int c )`**

  Returns nonzero if **`c`** is a printable character; zero otherwise.

- **`int isalpha( int c )`**

  Returns nonzero if **`c`** is an alpha character (A-Z, a-z); zero otherwise.

- **`int isalnum( int c )`**

  Returns nonzero if **`c`** is alphanumeric (A-Z, a-z or 0-9); zero otherwise.

- **`int isgraph( int c )`**

  Returns nonzero if **`c`** is a visible printing character; zero otherwise.

- **`int iscntrl( int c )`**

  Returns nonzero if **`c`** is a control character (less than 20_H); zero otherwise.

MATH.LIB

The Z-World standard library contains floating-point functions in addition to I/O functions. Normal mathematical limitations apply to these functions, and any function generating a value outside the accepted floating-point range (about 10^{38} to -10^{38}) will result in an overflow error. Infinity is defined as INF in **`DC.HH`**.

Trigonometric functions such as tan(x) generally accept arguments in radians. Certain trig functions may fail if their argument is too large. Any angle may be normalized to fall within the range $[-\pi, \pi]$ without loss of accuracy.

- **`int abs( int x )`**

  Computes the absolute value of an integer argument.

- **`float acos( float x )`**

  Computes the arccosine of **`x`**. The value of **`x`** must be between –1 and +1. If **`x`** is out of bounds, the function returns 0 and signals a domain error.

- **`float acot( float x )`**

  Computes the arccotangent of **`x`**. The value of **`x`** must be between –INF and +INF.

- **`float acsc( float x )`**

  Computes the arccosecant of **`x`**. The value of **`x`** must be between –INF and +INF.

- **`float asec( float x )`**

  Computes the arccosecant of **`x`**. The value of **`x`** must be between –INF and +INF.

- **`float asin( float x )`**

  Computes the arcsine of **`x`**. The value of **`x`** must be between –1 and +1. If **`x`** is out of bounds, the function returns 0 and signals a domain error.

- **`float atan( float x )`**

  Computes the arctangent of **`x`**. The value of **`x`** must be between –INF and +INF.

- **`float atan2( float y, float x )`**

  Computes the arctangent of **`y/x`**. If both **`y`** and **`x`** are zero, the function returns 0 and signals a domain error. Otherwise the result is returned as follows:

angle	$x \neq 0, y \neq 0$
PI/2	$x = 0, y > 0$
–PI/2	$x = 0, y < 0$
0	$x > 0, y = 0$
PI	$x < 0, y = 0$

- **`float ceil( float x )`**

  Returns the smallest integer greater than or equal to **`x`**.

- **`float cos( float x )`**

  Computes the cosine of **`x`**.

- **`float cosh( float x )`**

  Computes the hyperbolic cosine of **`x`**. If |**x**| > 89.8 (approx.), the function returns INF and signals a range error.

- **`float deg( float x )`**

  Returns angle in degrees for angle **`x`** given in radians.

- **`float rad( float x )`**

  Returns angle in radians for angle **`x`** given in degrees.

- **`float exp( float x )`**

  Returns the value of e^x. If **`x`** > 89.8 (approx.), the function returns INF and signals a range error. If **`x`** < –89.8 (approx.), the function returns 0 and signals a range error.

- **`float fabs( float x )`**

  Computes the absolute value of **`x`**. The function returns **`x`** if **`x`** ≥ 0; otherwise it returns –**`x`**.

- **`float floor( float x )`**

  Computes the largest integer less than or equal to the given number.

- **`float fmod( float x, float y )`**

  Returns the *remainder* of **x** with respect to **y**, that is, the remaining part of **x** after all multiples of **y** have been removed. For example, if **x** is 22.7 and **y** is 10.3, the integral division result is 2. Then the remainder = 22.7 – 2 × 10.3 = 2.1.

- **`float frexp( float x, int *n )`**

  This function splits **x** into a fraction and exponent ($f \times 2^n$). The function returns the exponent in the integer ***n** and the fraction (between 0.5 and 0.999...) as the function result.

- **`long labs( long x )`**

  Computes the absolute value of long integer **x**. The function returns **x** if $x \geq 0$; otherwise it returns –**x**.

- **`float ldexp( float x, int n )`**

  Computes **x***(radix****n**), where **n** is an integer and $0.5 \leq x < 1.0$.

- **`float log( float x )`**

  Computes the natural logarithm (base e) of **x**. The function returns –INF and signals a domain error when $x \leq 0$.

- **`float log10( float x )`**

  Computes the base 10 logarithm of **x**. The function returns –INF and signals a domain error when $x \leq 0$.

- **`float modf( float x, int *n )`**

  Splits **x** into an integer part and fractional part, $f + n$, where **n** is the integer and f satisfies $|f| < 1.0$. The function returns the integer part in ***n** and returns the fractional part as the function result.

- **`float poly( float x, int n, float c[] )`**

  Computes a polynomial value by Horner's method. The term **x** is the variable of the polynomial, **n** is the order of the polynomial, and **c** is an array containing the coefficients of each power of **x** . For example, for the fourth-order polynomial

  $10x^4 - 3x^2 + 4x + 6$

  n would be 4 and the coefficients would be

  ```
  c[4] = 10.0
  c[3] = 0.0
  c[2] = -3.0
  c[1] = 4.0
  c[0] = 6.0
  ```

- **`float pow( float x, float y )`**

  Returns x^y.

- **`float pow10( float x )`**

  Returns 10^x.

- **`float sin( float x )`**

  Computes the sine of **`x`**.

- **`float sinh( float x )`**

  If **`x`** > 89.8 (approx.), the function returns INF and signals a range error. If **`x`** < –89.8 (approx.), the function returns –INF and signals a range error.

- **`float sqrt( float x )`**

  Computes the square root of **`x`**.

- **`float tan( float x )`**

  Return the tangent of **`x`**, where $-8 \times PI \le$ **`x`** $\le +8 \times PI$. If **`x`** is out of bounds, the function returns 0 and signals a domain error. If the value of **`x`** is too close to a multiple of 90° (PI/2) the function returns INF and signals a range error.

- **`float tanh( float x )`**

  Returns the hyperbolic tangent of **`x`**. If **`x`** > 49.9 (approx.), the function returns INF and signals a range error. If **`x`** < –49.9 (approx.), the function returns –INF and signals a range error.

- **`float _pow10( int exp )`**

  Computes integral powers of 10 (10^{exp}).

- **`int getcrc( char *buffer, char count, int accum )`**

  Computes the CRC (cyclic redundancy check, or check sum) for **`count`** bytes (maximum 255) of data in **`buffer`**. Calls to **`getcrc`** can be "concatenated" using **`accum`** to compute the CRC for a large buffer. The function returns the integer CRC value.

STDIO.LIB

The following functions address the standard I/O window in Dynamic C, which is used for debugging.

- **`char *gets( char *s )`**

  This function waits for a string terminated by a **‹CR›** (carriage return) to be typed. It does not return until a ‹CR› is typed in the **STDIO** window. However, the string returned is null terminated. The function returns the typed string at the location identified by the pointer **`s`**. Make sure the storage is big enough for the string and that only one process calls this function at a time.

- **`char getchar( void )`**

  This function waits (in an idle loop) for a character to be typed from the **STDIO** window in Dynamic C. Make sure only one process calls this function at a time.

- **`int puts( char *s )`**

  This function writes the string, identified by pointer `s`, in the **STDIO** window in Dynamic C. The **STDIO** window will interpret any escape code sequences contained in the string. Make sure only one process calls this function at a time. The function returns 1 if successful.

- **`void putchar( int ch )`**

  Writes a single character (the lower 8 bits of `ch`) to **STDIO**. Make sure only one process calls this function at a time.

- **`int sprintf( char *buffer, char *format, ... )`**

  An analog of standard function `printf`, this function takes a "format" string (`*format`), and a variable number of value arguments to be formatted. It formats the arguments, places the formatted string in `*buffer` and returns the formatted string length. Make sure that:

  1. There are enough arguments after `format` to fill in the format parameters in the format string.
  2. The types of arguments after `format` match the format fields in `format`.
  3. `buffer` is large enough to hold the longest possible formatted string.

  For example,

  ```
  sprintf( buffer, "%s=%x", "Variable x", 256 )
  ```

  should put the string "Variable x=100" into `buffer`. This function is reentrant and can be called by processes of different priorities.

  The `printf` function is not reentrant. The `doprnt` function implements `printf` and `sprintf` using the character output functions `__qen` and `__qe`, respectively. These functions accept format strings and a variable number of parameters whose values are to be printed according to the format, for example,

  ```
  printf( "Summary for %s:\n", person );
  printf( "  Age: %d, Income: $%8.2f", age, income );
  ```

  The first statement prints a character string. The `%s` in the format tells the function where and how to print the character string.

  The second statement prints two numbers, an integer `age` and a float `income`. The `%d` in the format tells the function where and how to print the integer: as a decimal string, free-formatted.

The **`%8.2f`** in the format tells the function to print income as a floating value, with a field width of eight characters and two decimal places.

```
Summary for Sally Forth:
Age: 39, Income: $39587.02
```

The complete syntax of a field code is:

`%[+|-][`*`width`*`[.`*`precision`*`]][`*`lmodifier`*`]`*`letter`*

where, if a field width is specified, '+' makes the value right justified in its field and '-' makes the value left justified in its field.

width is the field width. If not specified, the field width varies according to the value.

precision for floating-point values, that is, the number of digits to the right of the decimal.

lmodifier modifies *letter* **`d`**, **`o`**, **`x`**, or **`u`** to expect the applicable one of either type **`long int`** or type **`unsigned long int`**.

letter selects the data's interpretation according to the following list.

`d`	decimal conversion (expects type **`int`**)
`o`	octal conversion (expects type **`int`**)
`x`	hex conversion (expects type **`int`**)
`u`	unsigned decimal conversion (expects type)
`c`	single **`char`** representation (expects type **`char`**)
`s`	string (expects zero terminated vector of type **`char *`**)
`e`	mantissa/exponent form of floating point (expects type **`float`**)
`f`	normal floating point (expects type **`float`**)
`g`	use the shorter of **`e`** or **`f`** conversion (expects type **`float`**)

- **`int snprintf( char *buffer, unsigned bufSize, char *fmt, ... )`**

  A length-limited version of **`sprintf`**, this reentrant function takes a maximum string length in addition to the "format" string (**`*fmt`**) and a variable number of value arguments to be formatted. It formats the arguments, places the length-limited formatted string in **`*buffer`** and returns the length of the formatted string as if **`bufSize`** were always large enough to never truncate the string. Refer to the description of **`sprintf`** for details on **`fmt`** and the variable argument list.

- **`int printf( char *fmt, ... )`**

  This standard function accepts a variable number of value arguments, composes a formatted string from the values, writes the formatted string to the **STDIO** window and returns the formatted string length. Refer to the description of **`sprintf`** for details. Only one process should use this function at any time.

- **`void doprnt(int(*put)(), char *fmt, void *arg1)`**

  This is the support routine behind all `..printf` routines. The function `put` must output one byte, it will be called whenever `doprnt` outputs a character. The term `fmt` is the format string that specifies the output. The term `arg1` points to the first parameter to be used by the formatted string. The interpretation of the parameters depends on the format fields in the format string. This routine causes many math functions to be compiled and downloaded. This routine can be called from processes of different priorities.

- **`char *gtoa( unsigned long num, char *ibuf )`**

  This function uses `_gltoa` to output an unsigned long integer, `num`, to the character array `*ibuf`. The function returns a pointer to `ibuf`.

- **`char *ltoa( long num, char *ibuf )`**

  This function uses `_gltoa` to output a signed long integer, `num`, to the character array `*ibuf`. The function returns a pointer to `ibuf`.

- **`int gtoan( unsigned long num )`**

  This function returns the number of characters required to display a unsigned long integer, `num`.

- **`int ltoan( long num )`**

  This function returns the number of characters required to display a signed long integer, `num`.

- **`void pint( char flag, char code, int width, int(*put)(), int value )`**

  Writes a short integer value as a decimal string according to the user-specified single-character output procedure `put`. The term `width` specifies field width. If zero is specified, the field will be as wide as needed to represent `value`. The `flag`, if `'-'`, indicates that the field is left-justified. Otherwise, it is right-justified. If `code` is `'d'`, the function treats `value` as a signed integer, otherwise as an eger. The function prints all asterisks if the value does not fit in the field specified.

- **`void plint( char left, char code, int n1, int(*put)(), long num )`**

  This function has the same effect as `pint`, but accepts and prints a long integer.

- **`int ftoa( float f, char *buf )`**

  Converts the floating pointer number `f` to a character string `*buf`. The string will be no longer than 12 characters long. The character string only displays the mantissa up to 12 digits, with no decimal points.

The function returns the exponent (base 10) that should be used to compensate for the missing decimal point. For example,

```
ftoa(1.0, buf)
```

generates the string "100000000" and returns –8. If **f** is 123.456, **ftoa** will generate the character string "123456000" and return the integer exponent –6, indicating 123456000 × 10^{-6}.

- **void plhex(char left, int nl, int(*put)(),
        long num)**

  Writes a long (signed or unsigned) integer in hex format. The term **left** specifies the padding character that goes to the left side of the actual number. If **left** is **'-'**, white space is used as a padding character. The term **nl** is the expected length of the output. Asterisks will be written if **num** requires more width than **nl**. Otherwise, the padding character **left** will be used to make up the remaining spaces. Pass a function (**put**) that will output one character. The function **put** should take a character argument. The term **num** is the number to be converted and output. This function can be called from processes of different priorities.

- **void phex(char left, int nl, int(*put)(),
        int num)**

  Similar to **plhex**. This function prints the hexadecimal representation of a short integer (signed or unsigned). Refer to the description of **plhex** for details.

- **void pflt(char flag, char code, int width,
        int digits, int(*put)(), float value,
        int prec)**

  Prints a formatted floating-point value using the specified single-character output procedure **put**.

  If **flag** is **'-'** then the output field is left-justified; if it is **'0'** then the field is right-justified and zero-filled; otherwise the field is right-justified and space-filled.

  The **code** can be **'e'**, **'f'**, or **'g'**. The **'e'** format displays a mantissa with "e" and an exponent. The **'f'** format displays standard decimal. The **'g'** format displays the shorter of the **'e'** or **'f'** formats.

  The term **width** is the field width. If zero is specified, the field will be as wide as needed to represent **value**. The terms **prec** and **digits** govern the number of significant digits to print. If **prec** is nonzero (true), the function prints **digits** significant digits. Otherwise, the function prints six significant digits. All asterisks are printed if the value does not fit in the field specified.

- **`char *itoa( int value, char *buf )`**

  Converts signed integer **`value`** to a character string in **`*buf`**, with a minus sign in first place, when appropriate. The function suppresses leading zeros, but leaves one zero digit for **`value`** = 0. The maximum value is 32767. The function returns a pointer to the end (the null terminator) of the string in **`*buf`**.

- **`char *utoa( unsigned value, char *buf )`**

  Converts eger **`value`** to a character string in **`*buf`**. The function suppresses leading zeros, but leaves one zero digit for **`value`** = 0. The maximum value is 65535. The function returns a pointer to the end (the null terminator) of the string in **`*buf`**.

- **`char *htoa( int value, char *buf )`**

  Converts integer **`value`** to hex character string in **`*buf`**. Leading zeros are not suppressed. The function returns a pointer to the end (null terminator) of the string in **`*buf`**.

- **`char *hltoa( long value, char *buf )`**

  Converts long integer **`value`** to hex character string in **`*buf`**. Leading zeros are not suppressed. The function returns a pointer to the end (null terminator) of the string in **`*buf`**.

- **`char outchrs( char c, int n, int(*put)() )`**

  Uses single-character output function **`put`** to output **`n`** times the character **`c`** . The function **`put`** should take a character parameter. The function returns the value of character **`c`**.

- **`char *outstr( char *buf, int(*put)() )`**

  Outputs the string **`*buf`** using calls to single-character output function **`put`**. The function **`put`** should take a character parameter. The function returns a pointer to the end (null terminator) of the string in **`*buf`**.

STRING.LIB

The following are standard C string functions.

- **`float atof( char *sptr )`**

  Returns the value resulting from conversion of a character string to a floating-point value. The initial “white space” is ignored.

- **`int atoi( char *sptr )`**

  Returns the value resulting from conversion of a character string to an integer value. The initial “white space” is ignored.

- **`int atol( char *sptr )`**

  Returns the value resulting from conversion of a character string to a long integer value. The initial "white space" is ignored.

- **`void *memset( void* dst, byte ch, unsigned n )`**

  Sets the memory starting at **`dst`** to **`n`** occurrences of the byte **`ch`**. The function returns a pointer to the address following the last byte written.

- **`char *strcpy( char *dst, char *src )`**

  Copies string **`*src`** to string **`*dst`**. The function copies at least one byte (the null). The function returns a pointer to **`*dst`**.

- **`char *strncpy(char *dst, char *src, unsigned n)`**

  Copies at most **`n`** characters from **`*src`** to **`*dst`**. May terminate earlier if null terminator is encountered in **`*src`** before **`n`** characters. The null terminator is not copied if **`n`** is encountered before null terminator (i.e., the programmer should take care of length-delimited cases). The function returns a pointer to **`*dst`**.

- **`char *strcat( char *dst, char *src )`**

  Concatenates string **`*src`** to the end of **`*dst`**. The destination string must be large enough to hold the additional characters. The function returns a pointer to **`*dst`**.

- **`char *strncat(char *dst, char *src, unsigned n)`**

  Concatenates up to **`n`** characters from **`*src`** to the end of **`*dst`**. A null terminator is appended to the end of **`*dst`** if **`n`** characters are copied before encountering the null terminator in **`*src`**. The function returns a pointer to **`*dst`**.

- **`int strcmp( char *a, char *b )`**

  Compares two strings. This function is useful for sorting. The function returns the relative difference between the first pair of differing characters, that is, the function result is

  0 if all bytes are equal
  < 0 if $a_i < b_i$
  > 0 if $a_i > b_i$

- **`int strncmp( char *a, char *b, unsigned n )`**

  Compares two strings up to **`n`** characters. The function return is similar to that of **`strcmp`**.

- **`char* strchr( char *src, char ch )`**

  Scans **`*src`** for the first occurrence of **`ch`**. The function returns a pointer to the first occurrence of **`ch`** in **`*src`**. It returns a null pointer if **`ch`** is not found.

- **`char* strrchr( char *src, int ch )`**

  Similar to **`strchr`**, except this function searches in reverse from the end of **`*src`** to the beginning. The function returns a pointer to the last occurrence of **`ch`** in **`*src`**. It returns a null pointer if **`ch`** is not found.

- **`unsigned strspn( char *src, char *set )`**

  Returns the length of the maximum initial segment of **`*src`**, which consists entirely of characters in **`*set`**.

- **`unsigned strcspn( char *src, char *set )`**

  Returns the length of the maximum initial segment of **`*src`**, which consists of characters not in **`*set`**.

- **`char* strpbrk( char *s1, char *s2 )`**

  Locates the first occurrence within **`*src`** of any character in **`*set`**. The function returns a pointer to the occurrence. The function returns a null pointer if none is found.

- **`void* memcpy(void *dst, void *src, unsigned n)`**

  Copies **`n`** characters from memory **`*src`** to memory **`*dst`**. Overlap is handled correctly. The function returns the **`*dst`** pointer.

- **`void* memchr( void* src, int ch, unsigned n )`**

  Searches up to **`n`** characters in buffer **`*src`** for character **`ch`**. The function returns a pointer to first occurrence of **`ch`** if found within **`n`** characters. Otherwise returns a null pointer.

- **`int strlen( char *s )`**

  Calculates the length of string **`*s`**, not including the terminating null. The function returns the number of bytes in the string.

- **`float strtod( char *s, char **tailptr )`**

  Converts a string to a floating-point value. The term **`*s`** is the string to convert, and **`**tailptr`** is a pointer to a pointer to a character. **`**tailptr`** is assigned the stopping point of conversion in **`*s`** (so continuation is possible at **`**tailptr`**). If no conversion takes place, **`**tailptr`** returns 0L. The initial "white space" is ignored. This function is ANSI compatible. The function returns the converted value.

- **`long strtol( char *s, char **tail, int base )`**

  Converts a string to a long integer value. The term **`*s`** is the string to convert, **`**tail`** is assigned the last position of the conversion, and **`base`** indicates the radix of conversion, which may be from 2 to 36.

  When **`base`** is 0, the function converts according to C syntax. For example, if the string starts with "**`0x`**," the function will interpret the string in hexadecimal format. This function skips initial "white space."

  When the conversion is successful, the function sets the tail pointer **`**tail`** to the character position at which the conversion finished and returns the converted value. The next conversion may resume at the location specified by **`**tail`**. Be careful with the double pointer! If the conversion fails (no conversion takes place) **`**tail`** is set to **`NULL`** and **`0L`** is returned.

- **`char *strtok( char *src, char *brk )`**

  Scans **`*src`** for tokens separated by delimiter characters specified in **`*brk`**. The first call takes a non-null **`*src`**. Subsequent calls with a null pointer for **`*src`** continue to search for tokens in the string. The function returns a pointer to the first character of the token. If a terminating delimiter is found, it is changed to a null character so that the token is terminated. A null pointer is returned if no token is found.

- **`char *strstr( char *string, char *target )`**

  Returns a pointer to the first occurrence of substring **`*target`** in **`*string`**. The function returns a null pointer if **`*target`** is not found in **`*string`**. The function returns the pointer string if the target is null.

- **`int memcmp( void *a, void *b, unsigned n )`**

  Compares two memory spaces **`a`** and **`b`** and returns the relative difference between the first pair of differing bytes, if any. The function stops comparing after **`n`** bytes. Thus, the function result is

$= 0$	if all bytes are equal
< 0	if $a_i < b_i$
> 0	if $a_i > b_i$

SYS.LIB

These are miscellaneous support functions.

- **`int setjmp( jmp_buf env )`**

  Stores the PC (program counter), SP (stack pointer) and other information about the current state into **`env`**. The saved information can be restored by executing **`longjmp`**. A typical program appears below.

```
switch(setjmp(e)){
   case 0:       // first time
      fx();      // fx() may take a longjmp
      break;     // if we get here, fx() was successful
// if we get here, fx() must have called longjmp
   case 1:
      do exception handling
      break;
// similar to case 1, but different exception code.
   case 2:
   ...
}
f(){
   g()
   ...
}                // Here, exception code 2 causes
g(){             // jump back to setjmp occurrence,
   ...           // but causes setjmp to return 2.
   longjmp(e,2); // Therefore, case 2 in the switch
}                // statement execute
```

  The function returns zero when it is executed. After **`longjmp`** is executed, the program counter, stack pointer, etc., are restored to the state when **`setjmp`** was executed the first time. However, this time, **`setjmp`** returns whatever value is specified by the **`longjmp`** statement.

- **`void longjmp( jmp_buf env, int value )`**
  Restores the stack environment saved in **`env`**. The integer value passed to **`longjmp`** is returned as the function result of **`setjmp`** when the long jump is taken. See the description of **`setjmp`** for usage.

- **`void *malloc( unsigned size )`**

  Allocates a dynamic block of **`size`** bytes. Call **`bfree`** before using **`*malloc`** (the compiler automatically calls **`bfree`** before **`main`** if some heap space is reserved in the logical memory options). Because **`*malloc`** uses a global free list pointer, **`*malloc`** must not be preempted by another **`*malloc`**. Heap space must be allocated using the logical memory option from the **Options** menu in order to use **`*malloc`**. (The default is a heap size of 0.) The function returns a pointer to the beginning of the allocated block, or a null pointer if space is unavailable.

- **`unsigned bfree( void *lo, void *hi )`**

  Defines a block of RAM, from **`*lo`** to **`*hi`** inclusive, as available for dynamic allocation. The function returns nonzero if successful, and zero if not.

- **`int free( void *f )`**

  Returns block (**`*f`**) of dynamically allocated RAM to the free list. The function returns nonzero if successful, and zero if not.

- **`int pack( void )`**

  Reduces fragmentation of dynamic memory by linking adjacent free blocks. The function returns the total number of free bytes.

- **`void *calloc( unsigned count, unsigned size )`**

  Allocates memory from the "heap" for a space of **`count`** elements of **`size`** bytes. The function finds a block of memory on the free list, trims it to the right size, and returns a pointer to the block. The function initializes the space to all zeros. The function returns a pointer to the allocated block, and returns a null pointer if it cannot find a block.

- **`void swap( byte a[], byte b[], int s )`**

  Swaps array **`a`** with array **`b`**, byte-for-byte, for the first **`size`** bytes.

- **`int qsort( void *base, unsigned n, unsigned s, int(*cmp)() )`**

  Performs a "quick sort" with center pivot, stack control, and easy-to-change comparison method. The term **`*base`** points to the base of an array (of fixed-size structures) to be sorted. The value **`n`** is the number of elements to be sorted, and **`s`** is the size of each element in the array. The programmer must supply a comparison function **`cmp`** that indicates the order of two structures.

  The comparison function takes pointers to two structures

  ```
  int cmp( void *p, void *q )
  ```

  and returns –1 if the first is *less* than the second, 0 if the structures are equal, and 1 if the first is *greater* than the second one.

  The **`qsort`** function returns zero if the operation is successful, and nonzero otherwise.

- **`char *realloc( void *ptr, unsigned size )`**

  Allocates a new block of size **`size`**, copies the data from the old block (**`*ptr`**) to the new block, frees the old block, and returns a pointer to the new block. If the function fails to allocate a new block, the function result is a null pointer.

- **`isr_ptr getvect( unsigned intrno )`**

  Gets the address of the handler of interrupt number **`intrno`**. For this function, number must be even and less than 255. The function returns the address of the handler. The type **`isr_ptr`** is a pointer to a function that returns void and takes no arguments.

- **`void setvect( unsigned intrno, isr_ptr isr )`**

  Sets a new handler **`isr`** for interrupt number **`intrno`**. The term **`intrno`** must be even and less than 255. The type **`isr_ptr`** is a pointer to a function that returns void and takes no arguments.

- **`int iff( void )`**

  Checks whether the interrupt flag is on. The function returns 1 if the interrupt flag is on, and 0 otherwise.

- **`void setireg( char value )`**

  Sets the Z180 interrupt register with the upper 8 bits of the specified 16-bit **`value`**.

- **`char readireg( void )`**

  Returns the value of the Z180 interrupt register as the upper 8 bits of the returned value. The lower 8 bits are set to zero.

- **`void _prot_init( void )`**

  Performs super initialization. The function initializes internal data needed for recovery of **`protected`** variables after a crash. To ensure that the protection mechanism works, call this function once in a program *before* any **`protected`** variables are set.

- **`void _prot_recover( void )`**

  Performs recovery of a partially completed assignment to a **`protected`** variable. Call this function after a power failure or a similar situation that does not lose memory.

- **`void reload_vec( unsigned vector, int(*isr)() )`**

  Loads an interrupt service routine to specified vector location at run time. **`vector`** is the interrupt vector to be served, **`*isr`** is the address of the interrupt service routine.

  `reload_vec` writes to the Flash EPROM memory when executed on a controller which is Flash EPROM equipped. Since Dynamic C 32 v. 6.30, **`reload_vec`** prevents rewriting existing Flash data because the Flash has a maximum life of about 10,000 writes. Exercise care to ensure that the application does not call **`reload_vec`** to write *different* data repeatedly to the same Flash address.

- **`int sysIsFlash( void )`**

  Returns nonzero if the controller is Flash EPROM equipped, zero otherwise.

- **`char sysDI( void )`**

  Disables interrupts then returns the previous interrupts enable state (IEF2) flag in bit 2. An interrupts disable instruction is always executed, regardless of the current interrupts enable state.

- **`void sysRestoreI( char PreviousState )`**

  Restores the interrupts enable state according to the supplied **`PreviousState`**, the result of a prior call to the **`sysDI`** function.

- **`int sysChk2ndFlash( struct _flashInfo *Info )`**

  Returns 0 and fills in the structure pointed to by **`Info`** if a 2nd Flash EPROM is found, otherwise returns a negative number. The **`_flashInfo`** structure is defined in **`SYS.LIB`** as follows:

  ```
  struct _flashInfo {
    unsigned sectorSize;     // size of one sector
    unsigned numSector;      // number of sectors
    char WEDelay;            // write enable delay
    unsigned ProgTO;         // programming timeout
    char wrMode;             // write mode
  };
  ```

- **`void sysRoot2FXmem( struct _flashInfo *Info,`**
  `void *Src, unsigned long Dest,`
  `unsigned Len)`

  Writes root memory data to the second Flash EPROM on a two-Flash EPROM equipped controller. The structure pointed to by **`Info`** should have been filled in by a previous call to **`sysChk2ndFlash`**. See the **`_flashInfo`** structure definition immediately above.

UTIL.LIB

These are general support functions for higher-level user functions.

- **`int IsZ80180( void )`**

  Checks Z180 CPU core type and returns non-0 if Z80180 or 0 if Z8S180. The method is undocumented, but deemed reliable by a Zilog designer. Required by functions in **`AASCZ0.LIB`** and **`AASCZ1.LIB`**. At the beginning of main() the application must call the appropriate one of **`_GLOBAL_INIT`**, **`uplc_init`** or **`VdInit`**.

XMEM.LIB

These are extended memory functions.

- **`unsigned long xmadr( void* address )`**

  Returns the physical address resulting from the conversion of logical address **`address`** according to the memory mapping registers. Uses the BBR, CBR and CBAR registers to determine the physical address.

- **`char xgetchar( long address )`**

  Gets a character whose address is specified by the physical **`address`** (20 bits). The function returns the character value.

- **`int xgetint( unsigned long address )`**

  Gets an integer whose address is specified by the physical **`address`** (20 bits). The function returns the integer value.

- **`unsigned long xgetlong( unsigned long address )`**

  Gets a long integer whose address is specified by the physical **`address`** (20 bits). The function returns an unsigned long integer value.

- **`float xgetfloat( unsigned long address )`**

  Gets a floating-point value whose address is specified by the physical **`address`** (20 bits). The function returns the floating-point value.

- **`void xputchar( long address, char value )`**

  Stores a character **`value`** at a physical **`address`** (20 bits).

- **`void xputint( long address, int value )`**

  Stores an integer **`value`** at a physical **`address`** (20 bits).

- **`void xputlong( long address, long value )`**

  Stores a long-**`value`** integer at a physical **`address`** (20 bits).

- **`void xputfloat( unsigned long address,`**
  `       float value )`

  Stores a float **`value`** at a physical **`address`** (20 bits).

- **`void xmem2root( unsigned long src, void *dst,`**
  `       unsigned n )`

  Copies a block of **`n`** bytes from extended memory **`src`** to root **`*dst`**. The address **`src`** is a physical **`address`** (20 bits).

- **`void root2xmem( void *src, unsigned long dst,`**
  `       unsigned n )`

  Copies a block of **`n`** bytes from root memory **`*src`** to extended memory **`dst`**. The address **`dst`** is a physical **`address`** (20 bits).

- **`unsigned xstrlen( unsigned long address )`**

  Returns the length of the string at the extended memory **`address`**. The address is a physical **`address`** (20 bits).

- **`unsigned x_makadr( unsigned long address )`**

  Computes the logical address from a physical **`address`**. The function also sets CBR to new page number and returns the logical address in HL. The old CBR is saved in **`af'`** (alternative register pair A and F). ***Never*** call this function from **`xmem`** functions. Z-World also recommends that this function not be called from C functions since it is easy to forget that a C function may be placed in **`xmem`** automatically.

- **`unsigned long a32_24( unsigned long address )`**

  Converts the 20-bit physical **`address`** (in a 32-bit integer) to a segmented (24-bit) address. Segmented addresses have the following structure.

8-bit CBR	16-bit Z180 address

- **`unsigned long a24_32( unsigned long address )`**

  Converts the 24-bit segmented **`address`** into a 20-bit physical address (in a 32-bit integer). The segment (second byte of the segmented address) is only effective if **`address`** is in **`xmem`**, that is, **`address`** $\geq$ 0xE000. Otherwise, the segment is ignored. Both the CBAR and BBR registers in the MMU are used to calculate the outcome. The function returns an unsigned long integer that holds the 20-bit physical address equivalent to the extended logical address supplied.

CHAPTER 2: MULTITASKING LIBRARIES

The multitasking libraries described in Chapter 2 include the real-time kernel, the simplified real-time kernel, the virtual driver, and the virtual watchdog libraries.

RTK.LIB

This library is the full real-time kernel. The simplified real-time kernel (SRTK) is described later.

- **`void request( unsigned tasknum )`**

  Requests the kernel to run the task specified by `tasknum` immediately. If a request for the task is pending, this call has no further effect. The specified task will be run on a future tick when priorities allow.

- **`void run_every( int tasknum, int period )`**

  Requests the kernel to run the task specified by `tasknum` every `period` ticks. The first request comes after `period` ticks. This is exact and no ticks will be gained or lost in the period.

- **`void run_after( int tasknum, long delay )`**

  Requests the kernel to run the task specified by `tasknum` after `delay` ticks have occurred.

- **`void run_at( int tasknum, void *time )`**

  Requests the kernel to run the task specified by `tasknum` when the time is greater than or equal to the time specified by the pointer `time`. The time pointer points to a 48-bit number (stored least significant byte first) that is the number of ticks since `init_kernel` was called.

- **`void run_cancel( int tasknum )`**

  Cancels any pending requests for the task specified by `tasknum`.

- **`void gettimer( void *time )`**

  Returns the current 48-bit time to the 6-byte area to which `time` points.

- **`void k_lock( void )`**

  Blocks task switches until the complementary `k_unlock` function is called. Should be called with interrupts disabled, and each `k_lock` function call must be paired with precisely one subsequent `k_unlock` function call.

- **`void k_unlock( void )`**

  Unblocks task switches that were blocked by a previous call to the complementary `k_lock` function. Should be called with interrupts disabled, and each `k_unlock` function call must be paired with precisely one prior `k_lock` function call.

- **`void run_timer( void )`**

  This function must be called by an interrupt routine between 10 and 500 times per second for the real-time kernel to operate. Each call to this function constitutes one kernel "tick," so all time values used by other kernel functions depend on the rate at which this function is called.

- **`int comp48( void *time1, void *time2 )`**

  Compares two 48-bit time values. The function returns

  –1 for **`time1`** < **`time2`**,

  0 for **`time1`** == **`time2`**, and

  +1 for **`time1`** > **`time2`**.

- **`void rkernel( void )`**

  This is the real-time kernel core, and is called by **`run_timer`**. This function will return immediately if there is no change to the task currently executing. If it decides to change tasks based on service requests such as **`run_every`** or **`run_after`**, then it will not return until the new task either returns or calls **`suspend`**.

- **`void suspend( unsigned ticks )`**

  This routine must be called only from within a given task. It allows the task to suspend itself for the specified number of ticks, after which it will continue to be requested automatically. Execution resumes at the statement following the call to **`suspend`**.

  If **`ticks`** is 0, then the suspension is for an indefinite period of time, until the task is again requested by some outside agent, such as a call to **`run_every()`**. Using a **`while`** statement is the usual method of using **`suspend`** to wait for an external event:

  ```
  while( !event() ) suspend(20);
  ```

  This example checks for the event every 20 ticks until the event takes place, at which point execution continues. The suspension can be up to 65,535 ticks.

- **`void init_kernel( void )`**

  Initializes the real-time kernel. This function takes no parameters. However, the calling program must contain certain definitions.

  Functions to be run as tasks must be declared with no parameters and return an integer. A global array of task pointers, **`Ftask`**, must be declared with the first task (**`Ftask[0]`**) given the highest priority and the last task the lowest priority. The application must

  `#define NTASKS` *number_of_tasks*

  to set the correct number of tasks. Then set up a periodic interrupt with a service routine that calls **`run_timer`**. An option is to

  `#define TASKSIZE_STORE` *task_storage_size*

  to be the size of the task storage area (this defaults to 50 if **`TASKSIZE_STORE`** is not defined in the application).

  All of the above definitions must occur in the source code before any reference to real-time kernel functions.

SRTK.LIB

These are the simplified real-time kernel functions.

- **`void srtk_hightask( void )`**

  This is the routine called every 25 ms by the SRTK to run high-priority tasks. The one in the library is a dummy routine.

  To have a user-defined SRTK high priority task, simply write one with the same name. Specify **`#nointerleave`** to guarantee that the user-defined high priority task is compiled.

- **`void srtk_lowtask( void )`**

  This routine is called every 100 ms by the SRTK to run low-priority tasks. The one in the library is a dummy routine.

  To have a user-defined SRTK low priority task, simply write one with the same name. Specify **`#nointerleave`** to guarantee that the user-defined low priority task is compiled.

- **`void init_srtkernel( void )`**

  Initializes the simplified real-time kernel. Once this is called, periodic interrupts will automatically invoke the SRTK high priority and low priority tasks. Initialize the virtual driver and

  `#define RUNKERNEL 1`

  before calling this function.

VDRIVER.LIB

These are the virtual driver functions. The virtual driver provides a number of different services, such as the virtual watchdog timers and a very high priority "fastcall" task. The virtual driver also provides delay routines for use by **`waitfor`** statements **`DelayMs`**, **`DelaySec`**, and **`DelayTick`**.

- **`void VdInit( void )`**

  Initializes the virtual driver. The Z180 PRT1 clocks the virtual driver every 1/1280 second. The virtual driver clocks the RTK or SRTK every 32 ticks (or 25 ms) if

  ```
  #define RUNKERNEL
  ```

  is defined in the application.

  For fastcall service, the virtual driver calls **`vd_fastcall`** every **`n`** ticks (1/1280 seconds) where 1 ≤ **`n`** ≤ 255. **`vd_fastcall`** must be defined and the definition will override the dummy version in the library. The application must

  ```
  #define VD_FASTCALL 1
  ```

  as well.

  `VdInit` must be called before the program can use the SRTK, virtual watchdogs, the **`waitfor`** delay routines or fastcall.

  `VdInit` makes a call to **`_GLOBAL_INIT`**. Therefore, a user-prepared program does not have to.

- **`int vd_initquickloop( int n )`**

  Initializes the "fastcall" feature of the virtual driver to run every **`n`** ticks. The value of **`n`** must be from 0 to 255, inclusive. If **`n`** = 0, it turns off **`vd_fastcall`**. In the application, use

  ```
  #define VD_FASTCALL 1
  ```

  next call **`VdInit`** (**`VdInit`** initializes **`vd_fastcall`** as *off*) and then call this function. The function returns 1 for success, 0 for a bad **`n`** value.

- **`void VdAdjClk( void )`**

  Synchronizes the software second timer used by **`DelaySec`** with the real-time clock. Call this function once a day or so to keep the clocks in sync.

- **`void vd_fastcall( void )`**

  Called by the virtual driver to run an ultra-fast thread every **`n`** ticks, where **`n`** is the argument to **`vd_initquickloop(n)`** and should be between 0 and 255, inclusive. In the application, use

  ```
  #define VD_FASTCALL 1
  ```

  to activate this thread. Calling **`vd_initquickloop(n)`** with **`n`** = 0 shuts off **`vd_fastcall`**.

VWDOG.LIB

These are the virtual watchdog timer functions. This library is automatically **`#use`**d by the **`VDRIVER.LIB`** virtual driver library.

- **`int VdGetFreeWd( char count )`**

  Returns a free virtual watchdog timer and starts it counting down from **`count`**. Virtual watchdog timers decrement every 25 milliseconds (32 virtual driver ticks). When a virtual watchdog reaches 0, it resets the processor. Once a virtual watchdog timer is active, the software should reset the timer periodically with a call to **`VdWdogHit`**. The function returns the integer ID of an unused virtual watchdog timer.

  If **`count`** ≤ 2, **`VdWdogHit`** must be called every 25 milliseconds. If **`count`** = 255, hit the watchdog at least every 6.375 seconds.

- **`int VdReleaseWd( int wd )`**

  Deactivates a virtual watchdog **`wd`** and returns it to the pool of watchdogs. The function returns 0 if **`wd`** is out of range, and 1 if successful.

- **`void VdWdogHit( int wd )`**

  Resets virtual watchdog timer to *n* counts where *n* was the argument to the call to **`VdGetFreeWd`** that obtained the virtual watchdog **`wd`**. The function returns 0 if **`wd`** is out of range, and 1 if successful.

CHAPTER 3: AASC LIBRARIES

The **`AASC.LIB`** Abstract Application-Level Serial Communication library and its low level support functions facilitate serial communication between controllers, and between a controller and another device such as a PC.

AASC.LIB

The Abstract Application-level Serial Communication (AASC) libraries allow the programmer to create buffered character streams (**`CHANNEL`**s) that perform input/output on serial ports in the communication devices. One principal library, **`AASC.LIB`**, contains all of the application level functions required for these tasks. The AASC libraries' function descriptions make use of the following

```
    typedef struct _Channel * CHANNEL;
```

which, along with the **`_Channel`** structure itself, is defined in **`AASC.LIB`**.

The high-level routines handle the bookkeeping for the connections between the low-level circular buffer and hardware driver libraries. This allows the same programming framework to use any applicable hardware.

- **`CHANNEL aascOpen(int Type, char CRTS,`**
  `long Param, void(*BRqFnc)())`

  If successful, initializes a serial device and returns the communication channel pointer required for all subsequent access to the serial device. If unsuccessful, returns NULL.

  `Type` is the type of communication device to open: **`DEV_Z0`** (ASCI0, "Z0"), **`DEV_Z1`** (ASCI1, "Z1"), **`DEV_UART`** (XP8700), **`DEV_ZNET`** (RS-485 "zNet" network), **`DEV_STDIO`** (Dynamic C STDIO window), **`DEV_SCC`** (Z85C30 SCC port A), **`DEV_SIO`** or **`DEV_SIOA`** (Z84C90 KIO's SIO port A).

  `CRTS` specifies whether CTS/RTS handshaking should be used: nonzero means yes, zero means no.

  `Param` specifies all of the other communication options. Z-World has defined the following macros.

Number of Data Bits	Number of Stop Bits	Number of Parity Bits
ASCI_PARAM_7D	**ASCI_PARAM_1STOP**	**ASCI_PARAM_NOPARITY**
ASCI_PARAM_8D	**ASCI_PARAM_2STOP**	**ASCI_PARAM_OPARITY**
		ASCI_PARAM_EPARITY
SCC_7DATA	**SCC_1STOP**	**SCC_NOPARITY**
SCC_8DATA	**SCC_2STOP**	**SCC_OPARITY**
		SCC_EPARITY

These macros apply to port Z0 of the Z180 or to the Serial Communication Controller. Refer to the Dynamic C driver descriptions or on-line help for additional macros.

Choose one macro from each column to bit-or or add together to describe the channel configuration, as shown below.

```
ASCI_PARAM_7D | ASCI_PARAM_1STOP | ASCI_NOPARITY
```

Two commonly used combination macros have also been defined.

`ASCI_PARAM_1200`—Basic quantum for bps rate. Multiply by the factor ***bps rate ÷ 1200*** (for example, 8 for 9600 bps).

`ASCI_PARAM_8N1`—Specifies 8 data bits, 1 stop bit and no parity.

For example, the Z0 channel in 8N1 format at 19,200 bps would have

```
Param = 16 * ASCI_PARAM_1200 | ASCI_PARAM_8N1
```

`BRqFnc` is a pointer to a function to be called by the **`Z0`** interrupt when a break request is detected. The return for **`BRqFnc`** is **`void`**.

- **`CHANNEL aascDLPReOpen(int Type, char CRTS, long Param)`**

  Like **`aascOpen`**, but used by a downloaded program (DLP) to reopen a channel that was previously opened by the download manager (DLM). It is necessary to use this instead of **`aascOpen`** if the two programs are to share a channel, as the DLM's **`BRqFnc`** handler must be preserved.

  Note that the DLP's **`aascDLPReOpen`** call's **`Type`**, **`CRTS`** and **`Param`** parameters must exactly match those in the DLM's respective **`aascOpen`** call. See **`aascOpen`** for details.

- **`void aascClose(CHANNEL Chan)`**

  Terminates a communication channel. The device dependent close routine is called first, then the storage associated with the channel is reattached to the free list. A downloaded program should not close a channel that it has reopened with an **`aascDLPReOpen`** call because it will disable the download manager's **`BRqFnc`** handler.

  `Chan` is a channel pointer, and must be the result of an **`aascOpen`** call.

- **`void aascSetReadBuf(CHANNEL Chan, char *Buf, unsigned Size)`**

  Designates a block of memory as the receive buffer for a communication channel.

  `Chan` is a channel pointer, the result of an **`aascOpen`** or an **`aascDLPReOpen`** call.

  `Buf` is the logical (root) address of the receive buffer.

  `Size` is the size of the receive buffer.

- **`void aascSetWriteBuf(CHANNEL Chan, char *Buf, unsigned Size)`**

  Designates the transmit buffer for a communication channel.

  `Chan` is a channel pointer, the result of an **`aascOpen`** call.

  `Buf` is the logical (root) address of the transmit buffer.

  `Size` is the size of the transmit buffer.

- **`void aascRxSwitch(CHANNEL Chan, char OnOff)`**

  Activates or deactivates a communication channel's receiver.

  `Chan` is a channel pointer, the result of an **`aascOpen`** call.

  `OnOff` sets the channel's receiver on (nonzero) or off (zero).

- **`void aascTxSwitch(CHANNEL Chan, char OnOff)`**

  Activates or deactivates a communication channel's transmitter.

  `Chan` is a channel pointer, the result of an **`aascOpen`** call.

  `OnOff` sets the channel's transmitter on (nonzero) or off (zero).

- **`unsigned aascReadChar(CHANNEL Chan, char *Dest)`**

  Reads a character from a communication channel, if available. Returns the actual number of characters read. The receiver will be enabled automatically if CTS/RTS flow control is enabled and the receive buffer has more than 16 bytes remaining after the read.

  `Chan` is a channel pointer, the result of an **`aascOpen`** call.

  `Dest` is a pointer to the buffer (one char) to read the character into.

- **`unsigned aascReadBlk(CHANNEL Chan, void *Dest, unsigned Len, char Flags)`**

  Reads a block of characters from a communication channel, if available. Returns the actual number of characters read. The receiver will be enabled automatically if CTS/RTS flow control is enabled and the receive buffer has more than 16 bytes remaining after the read.

  `Chan` is a channel pointer, the result of an **`aascOpen`** call.

  `Dest` is a pointer to the buffer to read the block of characters into.

  `Len` is the (optionally, maximum) number of bytes to read.

  `Flag` sets the block read mode. If zero then a maximum of Len characters will be read; if nonzero then either all **`Len`** characters will be read or none will be read.

- **`unsigned aascWriteChar(CHANNEL Chan, char Src)`**

  Writes a character to a communication channel, if possible. Returns the actual number of characters written. The transmitter is enabled automatically after the character is transferred.

  `Chan` is a channel pointer, the result of an **`aascOpen`** call.

  `Src` is the character to be written.

- **`unsigned aascWriteBlk(CHANNEL Chan, void *Src,`**
  `        unsigned Len, char Flags)`

  Writes a block of characters to a communication channel, if possible. Returns the actual number of characters written. The transmitter is enabled automatically after the characters are transferred.

  `Chan` is a channel pointer, the result of an **`aascOpen`** call.

  `Src` is a pointer to the block of character to be written.

  `Flag` sets the block write mode. If zero then a maximum of Len characters will be written; if nonzero then either all **`Len`** characters will be written or none will be written.

- **`unsigned aascPeek(CHANNEL Chan, void *Matchee,`**
  `        unsigned Len)`

  Returns the number of characters at the start of a communication channel's receive buffer that match the specified characters, up to a specified maximum length. Neither the buffer characters nor the match characters are presumed or required to be a zero-terminated "string." Note that this function does not read any characters from the channel.

  `Chan` is a channel pointer, the result of an **`aascOpen`** call.

  `Matchee` is a pointer to the match characters.

  `Len` is the maximum number of characters to attempt to match.

- **`unsigned aascScanTerm(CHANNEL Chan, char Term)`**

  Scans the receive buffer of a communication channel for the specified terminating character, and if found, returns the terminated packet's size. Note that this function does not read any characters from the channel. The receiver will be enabled automatically if CTS/RTS flow control is enabled and the receive buffer has more than 16 bytes remaining.

  `Chan` is a channel pointer, the result of an **`aascOpen`** call.

  `Term` is the terminator character to search for.

- **`void aascPipe(CHANNEL Chan1, CHANNEL Chan2)`**

  Makes a relatively low overhead data pipe by connecting the input of one communication channel to the output of the other, and the output of one channel to the input of the other. Exactly one of the channels (it doesn't matter which one) should have receive and transmit buffers previously assigned. The application should periodically check each channels' receive and transmit status, and call **`aascRxSwitch`** or **`aascTxSwitch`** to enable the receiver or transmitter as appropriate for each channel to ensure that the pipe remains active.

  `Chan1` is a channel pointer, the result of an **`aascOpen`** call.

  `Chan2` is a channel pointer, the result of an **`aascOpen`** call.

- **`long aascGetError(CHANNEL Chan)`**

  Returns a communication channel's current bit-mapped error condition. See the AASC libraries and on-line help for specific error conditions.

  `Chan` is a channel pointer, the result of an **`aascOpen`** call.

- **`void aascClearError(CHANNEL Chan)`**

  Clears a communication channel's current error condition.

  `Chan` is a channel pointer, the result of an **`aascOpen`** call.

- **`long aascGetStatus(CHANNEL Chan)`**

  Returns a communication channel's current bit-mapped status condition. See the AASC libraries and on-line help for specific status conditions.

  `Chan` is a channel pointer, the result of an **`aascOpen`** call.

- **`long aascSetStatus(CHANNEL Chan,`**
  `long ToggleMask)`

  Selectively toggles the writable bits in the communication channel's current status and then returns the channel's updated status condition. See the AASC libraries and on-line help for specific status conditions.

  `Chan` is a channel pointer, the result of an **`aascOpen`** call.

  `ToggleMask` is a device dependent bit-mapped mask which is XORed with the writable bits in the channel's current status.

- **`unsigned aascReadBufLeft(CHANNEL Chan)`**

  Returns the number of characters available to be read from a communication channel's receive buffer.

  `Chan` is a channel pointer, the result of an **`aascOpen`** call.

- **`unsigned aascWriteBufLeft(CHANNEL Chan)`**

  Returns the number of characters remaining to be transmitted from a communication channel's transmit buffer.

  `Chan` is a channel pointer, the result of an **`aascOpen`** call.

- **`unsigned aascReadBufFree(CHANNEL Chan)`**

  Returns the available free space remaining (maximum number of characters that would still fit) in a communication channel's receive buffer.

  `Chan` is a channel pointer, the result of an **`aascOpen`** call.

- **`unsigned aascWriteBufFree(CHANNEL Chan)`**

  Returns the available free space remaining (maximum number of characters that would still fit) in a communication channel's transmit buffer.

  `Chan` is a channel pointer, the result of an **`aascOpen`** call.

- **`void aascFlush(CHANNEL Chan)`**

  Flushes a communication channel's receive and transmit buffers by discarding any character(s) that may be present in the buffers. If the channel is capable of CTS/RTS flow control and Rx data is thereby inhibited, the programmer should determine whether to explicitly reenable Rx data by calling **`aascRxSwitch`**.

  `Chan` is a channel pointer, the result of an **`aascOpen`** call.

- **`void aascFlushRdBuf(CHANNEL Chan)`**

  Flushes a communication channel's receive buffer by discarding any character(s) that may be present in the buffer. If the channel is capable of CTS/RTS flow control and Rx data is thereby inhibited, the programmer should determine whether to explicitly reenable Rx data by calling **`aascRxSwitch`**.

  `Chan` is a channel pointer, the result of an **`aascOpen`** call.

- **`void aascFlushWrBuf(CHANNEL Chan)`**

  Flushes a communication channel's transmit buffer by discarding any character(s) that may be present in the buffer.

  `Chan` is a channel pointer, the result of an **`aascOpen`** call.

- **`void aascPrintf(CHANNEL Chan, char *Fmt, …)`**

  Writes a **`printf`**-style formatted string to a communication channel.

  `Chan` is a channel pointer, the result of an **`aascOpen`** call.

  `Fmt` is a pointer to the **`printf`**-style format string for the information that is to be printed. See the description of **`STDIO.LIB`**'s **`sprintf`** function in this manual for details.

  ... (ellipsis) represents a variable number (zero or more) of arguments that should match the conversion specifiers found in the format string.

- **`void aascVPrintf(CHANNEL Chan, char *Fmt, void *FirstArg)`**

  Writes a **`printf`**-style formatted string to a communication channel.

  `Chan` is a channel pointer, the result of an **`aascOpen`** call.

  `Fmt` is a pointer to the **`printf`**-style format string for the information that is to be printed. See the description of **`STDIO.LIB`**'s **`sprintf`** function in this manual for details.

  `FirstArg` is a pointer to the first in a vector of arguments (often, as a stack frame) that should match the conversion specifiers found in the format string. It can be **`NULL`** if no arguments are required.

XModem Functions in AASC.LIB

The XModem protocol performs packet-based data transfers with CRC error detection. The packet structure for XModem transfer appears below.

Bytes	Description
1	Start Of Header
1	Packet Sequence Number
1	1's Complement of Packet Sequence Number
128 or 1024	DATA (binary or text)
2	CRC-CCITT (0x1021 divisor)

- **`void aascXMRdInitPhy(unsigned Where, unsigned Len, unsigned long XmemDstAddr)`**

  Specifies the root memory (one packet size) transfer buffer, the physical memory destination and the maximum number of bytes to be transferred in an **`aascReadXModem`** PC-to-target physical memory data transfer.

  `Where` is the root memory transfer buffer, where the data being transferred are temporarily placed.

  `Len` is the maximum number of bytes to transfer.

  `XmemDstAddr` is the transferred data's final destination in physical memory.

- **`void aascXMRdInitLog(unsigned Where,`**
  `        unsigned Len)`

  Specifies the logical (root) memory destination and the maximum number of bytes to be transferred in an **`aascReadXModem`** PC-to-target root memory data transfer.

  `Where` is the transferred data's final destination in root memory.

  `Len` is the maximum number of bytes to transfer.

- **`unsigned aascReadXModem(CHANNEL Chan,`**
  `        char *(*read_callback_loc)(),`
  `        void (*read_callback_mod)(),`
  `        char Init)`

  Performs PC-to-target XModem data transfer. Call this function once with **`Init`** set to 1, then call this function repeatedly with **`Init`** set to 0 until its return value is nonzero. If successful, the nonzero result will be **`XX_SUCCESS`**; in case of failure, see on-line help for information on other nonzero result codes.

  Before calling this function, call either **`aascXMRdInitPhy`** to set up a physical memory transfer or **`aascXMRdInitLog`** to set up a logical (root) memory transfer.

  `Chan` is a channel pointer, the result of an **`aascOpen`** call.

  `read_callback_loc` is a pointer to the callback function that is called by this function ***before*** each XModem packet is received.

  The **`aascRdCBackLocLg`** and **`aascRdCBackLocPh`** callback functions are provided to determine the root memory location of the packet's destination or transfer buffer, respectively.

  `read_callback_mod` is a pointer to the callback function that is called by this function ***after*** each XModem packet is received.

  The empty **`aascRdCBackCfmLg`** callback function, which performs no post-processing of the packet, is provided for logical (root) memory transfers. The **`aascRdCBackCfmPh`** callback function, which performs root-to-xmem transfer of the packet, is provided for physical memory transfers.

  `Init` is a flag which: if nonzero initializes the XModem state machine (only the first call); if zero receives XModem packet data (all subsequent calls).

- **`unsigned aascRdCBackLocPh(unsigned PackSize,`**
  `        char PackNum)`

  Returns the logical (root) memory one packet size transfer buffer address, or zero if the packet's position is outside of the buffer. This function depends on information set up by calling **`aascXmRdInitPhy`** prior to the first call to **`aascReadXModem`**.

  This is the standard XModem packet preprocessing callback function provided for **`aascReadXModem`** PC-to-target physical memory transfers. It can be replaced by a user-defined function in applications which require a different type of packet preprocessing.

  `PackSize` is the packet size being used, either 128 or 1024 bytes.

  `PackNum` is the sequence number of the current packet.

- **`unsigned aascRdCBackLocLg(unsigned PackSize,`**
  `        char PackNum)`

  Returns the logical (root) memory destination buffer address, or zero if the packet's position is outside of the buffer. This function depends on information set up by calling **`aascXmRdInitLog`** prior to the first call to **`aascReadXModem`**.

  This is the standard XModem packet preprocessing callback function provided for **`aascReadXModem`** PC-to-target logical (root) memory transfers. It can be replaced by a user-defined function in applications which require a different type of packet preprocessing.

  `PackSize` is the packet size being used, either 128 or 1024 bytes.

  `PackNum` is the sequence number of the current packet.

- **`void aascRdCBackCfmPh(unsigned PackSize,`**
  `        char PackNum, unsigned Status);`

  Copies a successfully received packet from the logical (root) memory transfer buffer to its final destination in physical memory. This function depends on information set up by calling **`aascXmRdInitPhy`** prior to the first call to **`aascReadXModem`**.

  This is the standard XModem packet post-processing callback function provided for **`aascReadXModem`** PC-to-target physical memory transfers. It can be replaced by a user-defined function in applications which require a different type of packet post-processing.

  `PackSize` is the packet size being used, either 128 or 1024 bytes.

PackNum is the sequence number of the current packet.

Status is a flag which: if zero indicates a bad packet and no post-processing is done; if nonzero indicates a good packet which is post-processed.

- **`void aascRdCBackCfmLg(unsigned PackSize, char PackNum, unsigned Status);`**

Empty function which performs no post-processing of a received packet.

This is the standard XModem packet post-processing callback function provided for **aascReadXModem** PC-to-target logical (root) memory transfers. It can be replaced by a user-defined function in applications which require a different type of packet post-processing.

PackSize is the packet size being used, either 128 or 1024 bytes.

PackNum is the sequence number of the current packet.

Status is a flag which: if zero indicates a bad packet; if nonzero indicates a good packet.

- **`void aascXMWrInitPhy(unsigned Where, unsigned Len, unsigned XmemSrcAddr)`**

Specifies the root memory (one packet size) transfer buffer, the physical memory source and the maximum number of bytes to be transferred in an **aascWriteXModem** physical memory target-to-PC data transfer.

Where is the root memory transfer buffer, where the data being transferred are temporarily placed.

Len is the maximum number of bytes to transfer.

XmemSrcAddr is the transferred data's source in physical memory.

- **`void aascXMWrInitLog(unsigned Where, unsigned Length)`**

Specifies the logical (root) memory source and the maximum number of bytes to be transferred in an **aascWriteXModem** root memory target-to-PC data transfer.

Where is the transferred data's source in root memory.

Len is the maximum number of bytes to transfer.

- **`int aascWriteXModem(CHANNEL Chan,`**
  `char Pak1K, char Init,`
  `unsigned(*write_callback)())`

  Performs target-to-PC XModem data transfer. Call this function once with **`Init`** set to 1, then call this function repeatedly with **`Init`** set to 0 until its return value is nonzero. If successful, the nonzero result will be **`XX_SUCCESS`**; in case of failure, see on-line help for information on other nonzero result codes.

  Before calling this function, call either **`aascXMWrInitPhy`** to set up a physical memory transfer or **`aascXMWrInitLog`** to set up a logical (root) memory transfer.

  `Chan` is a channel pointer, the result of an **`aascOpen`** call.

  `Pak1K` is the XModem packet size flag: if zero use 128 byte packets; if nonzero use 1024 byte packets.

  `Init` is a flag which: if nonzero initializes the XModem state machine (only the first call); if zero sends XModem packet data (all subsequent calls).

  `write_callback` is a pointer to the callback function that is called by this function ***before*** each XModem packet is sent.

  The **`aascWrCallBackLg`** and **`aascWrCallBackPh`** callback functions are provided to determine the root memory location of the packet's source or transfer buffer, respectively.

- **`unsigned aascWrCallBackPh(unsigned PackSize,`**
  `char PackNum)`

  Returns the logical (root) memory one packet size transfer buffer address, or zero if the packet's position is outside of the buffer. This function depends on information set up by calling **`aascXmWrInitPhy`** prior to the first call to **`aascWriteXModem`**.

  This is the standard XModem packet preprocessing callback function provided for **`aascWriteXModem`** physical memory target-to-PC transfers. It can be replaced by a user-defined function in applications which require a different type of packet preprocessing.

  `PackSize` is the packet size being used, either 128 or 1024 bytes.

  `PackNum` is the sequence number of the current packet.

- **`unsigned aascWrCallBackLg(unsigned PackSize, char PackNum)`**

  Returns the logical (root) memory source buffer address, or zero if the packet's position is outside of the buffer. This function depends on information set up by calling **`aascXmWrInitLog`** prior to the first call to **`aascWriteXModem`**.

  This is the standard XModem packet preprocessing callback function provided for **`aascWriteXModem`** logical (root) memory target-to-PC transfers. It can be replaced by a user-defined function in applications which require a different type of packet preprocessing.

  `PackSize` is the packet size being used, either 128 or 1024 bytes.

  `PackNum` is the sequence number of the current packet.

CHAPTER 4: OTHER COMMUNICATION DRIVERS

These libraries contain both general and controller-specific functions for controller-to-controller and PC-to-controller communication. These libraries have been superseded by the AASC libraries in Chapter 3, which provide a consistent API for all Z-World controllers. The other Dynamic C communication libraries are still available, and are documented here in Chapter 4.

MODEM232.LIB

This is a modem functions support library for **Z0232.LIB**, **S0232.LIB**, **UART.LIB**, **NETWORK.LIB**, and **SCC232.LIB**.

- **`int Dget_modem_command( char *buffer )`**

  Scans **`buffer`** for a (Hayes-compatible) modem command terminated by **<CR>**.

  RETURN VALUE:

–1—no command present	5—“CONNECT 1200”
0—“OK”	6—“NO DIALTONE”
1—“CONNECT”	7—“BUSY”
2—“RING”	8—“NO ANSWER”
3—“NO CARRIER”	9—“CONNECT 2400”
4—“ERROR”	10—“\n” *just a new line*

  A Hayes SmartModem or compatible modem is recommended. A ***null*** modem cable is needed between the controller or expansion board and the modem. Some modems require that the RTS, CTS, and DTR lines be tied together.

- **`void resetZ180int( void )`**

  This is a generic reset function that resets, or disables, interrupts for the DMA channels, Z180 serial channels 0 and 1, the PRT timers, the CSI/O, INT1, and INT2.

NETWORK.LIB

These are RS-485 network functions. They provide utilities for master-slave half-duplex RS-485 communication using the Opto22 binary 9th bit protocol. There must be exactly one master. There can be as many as 255 slaves.

- **`int check_opto_command( void )`**

  Checks for a valid and completed command or reply in the receive buffer.

  RETURN VALUE:

0	if there is no completed command or message available.
1	if there is a completed command or reply available.
–2	if the completed command or reply has a bad CRC check.

-
  ```
  int sendOp22( unsigned char dest,
           char *message, unsigned char len,
           int delay )
  ```

  The master sends a message to the slave and waits for a reply. The function puts the message in the following format:

```
[slave id] [len] [ ] [ ]...[ ] [CRC hi][CRC lo]
```

PARAMETERS:
dest is the slave destination (1–255).
message points to a byte array.
len is the length of the message. The maximum message length is 251 bytes.
delay is the number of delays to wait for the slave reply. Each delay is ~50 ms. However, if the RTK is in use, the delay is made using **suspend(2)**.

RETURN VALUE:
–1 if there is no reply from the slave.
–2 if a completed reply has a bad CRC.
 1 if there is a completed reply with a proper CRC.

The slave's reply is stored in the receive buffer initialized with **op_init_z1**.

- **void replyOpto22(char *reply,
  unsigned char count, int delays)**

The slave replies to the master's inquiry. The function puts the reply in the following format.

```
[len] [ ] [ ]...[ ] [CRC hi] [CRC lo]
```

PARAMETERS:
reply is the slave's reply string.
count is the length of the reply. Because two CRC bytes are appended at the end, the longest reply is 252 bytes.
delays is the number of delays before the message is transmitted back. Each delay is ~50 ms. However, if the RTK is in use, the delay is made using **suspend(2)**.

- **void adelay_50ms(void)**

Creates a 2 tick delay if the RTK is in use. Otherwise, it executes a 50 ms (approx.) software delay loop.

- **void op_init_z1(char baud, char *rbuf,
  unsigned char address)**

Initializes Z180 port 1 for RS-485 9th-bit binary communication. The data format defaults to 8 bits, no parity, 1 stop bit.

PARAMETERS:
baud selects the baud rate in multiples of 1200 bps (for example, specify 16 for 19,200 bps).
rbuf is the receive buffer.
address is the network address of the board: 0 for the master board, 1–255 for slaves.

- **`void op_kill_z1( void )`**

  Disables Z180 port 1 and disables the RS-485 driver.

PRPORT.LIB

These are printer port functions. This library provides routines for communicating between the PIO parallel ports and IBM PC-style printers or the printer port on computer. The **`prsend...`** functions communicate with a PC-style printer. The **`plink...`** functions communicate with the printer port on a computer by making the PIO port 0 appear to be a printer. **`clink_init`** initializes PIO port 0 for high-speed communications with a PC.

- **`int prsend0( char dat )`**

  Sends the character **`dat`** to the printer on PIO port 0.

  RETURN VALUE:
  0 if the character was sent successfully.
  1 if the printer is off-line.
  2 if the printer is out of paper.

  - **`void prsend0_init( void )`**

  Initializes PIO port 0 for sending data to an IBM PC-style printer. The printer bits are as follows.

bit 7	error	A low signals printer error condition
bit 6	slct	A pulse signals that the printer is selected
bit 5	+PE	A pulse indicates out of paper
bit 4	+busy	A pulse indicates the printer is busy
bit 3	–slctin	Printer indicates it is selected
bit 2	–int	Drive a negative pulse to reset printer
bit 1	–ack	A negative pulse is acknowledgment
bit 0	–strobe	A negative pulse indicates char ready

- **`int prsend1( char dat )`**

  Sends the character **`dat`** to the printer on PIO port 1.

  RETURN VALUE:
  0 if the character was sent successfully.
  1 if the printer is off-line.
  2 if the printer is out of paper.

- **`int prsend1_init( void )`**

  Initializes PIO port 1 for sending data to an IBM PC style printer.

- **`void doreti( void )`**

  Call this routine to perform a Z80-style **`reti`** instruction. Use it to prevent resetting the IUS latch on two peripheral devices with one **`reti`** if LIR is enabled on the BL1000 or the BL1100. Call **`doreti`** with the interrupts off to avoid the risk of executing other interrupt routines with the wait states set to a high value.

- **`void piolatch( void )`**

  Guarantees an LIR cycle to latch the PIO interrupt state.

- **`void setwaits( int mem, int io )`**

  Sets up the programmable wait state register in the Z180 without disturbing DMA control in the lower part of the register.

  PARAMETERS:
  `mem` specifies the number of memory wait states (0–3).
  `io` specifies the number of io wait states (0–3).

- **`void plink_init0( struct circ_buf *ptr,`**
  `char *buf, int amask )`

  Initializes functions to make the PIO device appear to be an IBM PC-style printer. Characters will be captured under interrupt into the circular buffer provided to this function. The **`plink_rdy0`** and **`plinkgetc0`** functions may be used to retrieve characters from the buffer. It may be desirable to load a TSR on the PC to insure that no characters will be lost.

  PARAMETERS:
  `ptr` points to a **`circ_buf`** structure used by the **`plink...`** routines.
  `buf` points to the circular buffer.
  `amask` is the buffer wrap mask. This value must be set to 2^{n-1}, where n is an integer between 2 and 16, and 2^n is the size of the buffer in bytes.

- **`int plink_rdy0( void )`**

  Checks for characters received in the circular buffer.

  RETURN VALUE:
  0 if buffer is empty.
  1 if buffer contains at least one character.

- **`int plink_getc0( int no_purge )`**

  Retrieves the next character from the circular buffer. This function must not be called if the buffer is empty. If **`no_purge`** is nonzero, the character will remain in the buffer after this call. Interrupts must be enabled when this function is called.

  RETURN VALUE: the first character in the buffer.

- **`void plinki0( void )`**

  Initializes the printer port for high-speed input/output.

- **`void plink_intr0( void )`**

  Interrupt to handle receipt of characters into the circular buffer passed to **`plink_init0`**.

- **`void clink_init( void )`**

  Initializes functions for high-speed communications with the parallel port of an IBM PC. This function takes no parameters; however, the calling program must contain certain definitions: several data buffers must be defined, **`bufptrs`** must be declared as an array of pointers to the buffers, and **`NBUFS`** must be defined to be the number of buffers. These definitions must all occur in the source code before any reference to **`clink_init`**. All communications after this call are entirely driven by the PC, so the actual size and number of buffers depends on the PC application.

  After the link is established, the PC may initiate communication by sending a command packet to the target. This packet will be either 4 or 6 bytes long, depending on the command. The first three bytes of the packet are the command, the device address (unused), and the buffer number. Four-byte packets contain a check sum in the fourth byte; six-byte packets contain a 2 byte count in the fourth and fifth bytes, and a check sum in the sixth byte. The buffer number identifies the buffer (in the **`bufptrs`** array) to perform the operation on. Possible command values are 0x11, 0x22, 0x33, 0x44, and 0x55. Commands 0x11 (receive block) and 0x22 (send block) require 6 bytes; the other commands require 4 bytes.

  Command 0x11 (receive block) forces the target to receive count bytes of data and a check sum byte into the specified buffer. The target will respond with 0xAA if successful, 0xCC if one of the check sums failed.

  Command 0x22 (send block) forces the target to first acknowledge the command packet (0xAA for success; 0xCC for fail), and then send count bytes of data from the specified buffer, followed by a check sum byte. If the initial acknowledgment is negative (0xCC), no data will be sent.

  Command 0x33 (set) sets the first byte of the specified buffer to 1. The target will respond with 0xAA if successful, or 0xCC if the command packet check sum fails.

Command 0x44 (clear) clears the first byte of the specified buffer to 0. The target will respond with 0xAA if successful, or 0xCC if the command packet check sum fails.

Command 0x55 (test) test the value of the first byte in the specified buffer. If the value is nonzero, a response of 0xBB will be sent. If the value is zero, the response will be 0xAA. If the command packet check sum fails, 0xCC will be sent.

- **`int checksum( char *buf, int len )`**

  Performs a check sum on the data contained in **`buf`**. **`len`** is the number of bytes that will be included in the check sum.

  RETURN VALUE:
  the check sum value (also stored in the variable **`summer`**).

- **`void pioint( void )`**

  Interrupt routine to handle high-speed communications with the PC parallel port. This routine will process an entire packet (send or receive) from the PC before it returns.

- **`int fastblock( char *buf, unsigned int cnt2 )`**

  Receives a block of data from the PC parallel port. This function will not return until the entire block is received, or a time-out occurs.

  PARAMETERS:
  `buf` specifies the location to store the data.
  `cnt2` specifies the size of the block to receive in words (1/2 the number of bytes to receive).

  RETURN VALUE:
  0 if successful.
  1 if a time-out occurred.

- **`int sendfast( char *buf, int cnt4 )`**

  Sends a block of data to the PC parallel port. This function will not return until the entire block is sent, or a time-out occurs. This function uses port B. On entry PB1, PB2–PB7 are outputs, other bits are inputs.

  PARAMETERS:
  `buf` points to the data to send.
  `cnt4` is the size of the block in bytes.

  RETURN VALUE:
  0 if data sent successfully.
  1 if a time-out occurred.

SCC232.LIB

This library contains the serial drivers for SCC serial ports A and B. Interrupts are generated via the Z180's INT1. The library also contains definitions for the PIO ports on the BL1300.

- **`int Dinit_sca( void *rbuf, void *tbuf,`**
  `        int rsize, int tsize, char mode,`
  `        char baud, char ismodem, char isecho )`

  Initializes SCC port A for communication. This function uses circular receive and transmit buffers, which are allocated by the programmer. This function tells the software what the setup is.

  PARAMETERS:
  `rbuf` is a pointer to the receive buffer.
  `tbuf` is a pointer to the transmit buffer
  `rsize` is the size, in bytes, of the receive buffer.
  `tsize` is the size, in bytes, of the transmit buffer.
  `mode` selects communication criteria as follows.

bit 0	0 = 1 stop bit 1 = 2 stop bits
bit 1	0 = no parity 1 = with parity
bit 2	0 = 7 data bits 1 = 8 data bits
bit 3	0 = even parity 1 = odd parity
bit 4	0 = no CTS/RTS control 1 = CTS/RTS enabled

  `baud` selects the baud rate in multiples of 1200 bps. Valid multipliers are 1, 2, 4, 8, 16, 24, 32, 48 and 64. Pass a value of 8 to get 9600 bps.
  `ismodem` if 1, modem communication is supported. Otherwise is 0.
  `isecho` if 1, every character is echoed. Otherwise is 0.

  If CTS/RTS handshaking is selected, transmission from the sender is disabled (by raising RTS) when the receive buffer is 80% full. The software lowers RTS (enabling the sender to transmit) when the receive buffer falls below 20% of capacity. In a similar manner, a remote system can prevent transmission of data by SCC port A by asserting its RTS (connected to the SCC port A CTS).

  RETURN VALUE: always 1.

- **`void Dreset_scarbuf( void )`**

  Resets the receive buffer.

- **`void Dreset_scatbuf( void )`**

  Resets the transmit buffer and stops transmission.

- **`int Dinit_scb( void *rbuf, void *tbuf,`**
  `int rsize, int tsize, char mode,`
  `char baud, char ismodem, char isecho )`

  Initializes SCC port B for communication. This function uses circular receive and transmit buffers, which are allocated by the programmer. This function tells the software what the setup is.

  PARAMETERS:
  `rbuf` is a pointer to the receive buffer.
  `tbuf` is a pointer to the transmit buffer
  `rsize` is the size, in bytes, of the receive buffer.
  `tsize` is the size, in bytes, of the transmit buffer.
  `mode` selects communication criteria as follows.

bit 0	0 = 1 stop bit 1 = 2 stop bits
bit 1	0 = no parity 1 = with parity
bit 2	0 = 7 data bits 1 = 8 data bits
bit 3	0 = even parity 1 = odd parity
bit 4	0 = no CTS/RTS control 1 = CTS/RTS enabled

  `baud` selects the baud rate in multiples of 1200 bps. Valid multipliers are 1, 2, 4, 8, 16, 24, 32, 48 and 64. Pass a value of 8 to get 9600 bps.
  `ismodem` if 1, modem communication is supported. Otherwise is 0.
  `isecho` if 1, every character is echoed. Otherwise is 0.

  If CTS/RTS handshaking is selected, transmission from the sender is disabled (by raising RTS) when the receive buffer is 80% full. The software lowers RTS (enabling the sender to transmit) when the receive buffer falls below 20% of capacity. In a similar manner, a remote system can prevent transmission of data by SCC port A by asserting its RTS (connected to the SCC port B CTS).

  RETURN VALUE: always 1.

- **`void Dreset_scbrbuf( void )`**

  Resets the receive buffer.

- **`void Dreset_scbtbuf( void )`**

  Resets the transmit buffer and stops transmission.

- **`void Drestart_scamodem( void )`**

  Restarts a modem during start-up or because of abnormal operation in SCC port A.

- **`void Drestart_scbmodem()`**

  Restarts a modem during start-up or because of abnormal operation in SCC port B.

  A Hayes SmartModem or compatible modem is recommended. A ***null*** modem cable is needed between the Z-World controller or expansion board and the modem. Some modems require that the RTS, CTS, and DTR lines be tied together.

- **`void interrupt reti sccint( void )`**

  This is an interrupt service routine for the SCC serial channels via the INT1 of the Z180. The interrupt routine is automatically triggered when **`Dinit_sca`** or **`Dinit_scb`** is called.

- **`void scabinaryset( void )`**

  Puts the serial receiver in BINARY mode. This means that ***all*** received characters are placed in the receive buffer.

- **`void scabinaryreset( void )`**

  Places the serial receiver in ASCII mode, where the BACKSPACE character (0x08) is parsed out of the receive buffer. Character echo also resumes if it was selected.

- **`int scamodemstat( void )`**

  Returns the status of the modem.

  RETURN VALUE:
  1 if the modem is in command mode.
  0 if the modem is in data mode (i.e., open to communication).

- **`int scamodemset( void )`**

  Returns information about modem selection.

  RETURN VALUE:
  1 if the modem option is selected.
  0 otherwise.

- **`void Dscasend_prompt()`**

  Places CR, LF and > in the transmit buffer.

- **`int Dwrite_sca( char *buffer, int count )`**

  Copies **`count`** bytes from **`buffer`** to the transmit buffer. If SCC port A is not already transmitting, the function initiates transmission.

RETURN VALUE:
0 if the transmit buffer did not have space for **`count`** bytes.
1 if the write is successful.

- **`int Dread_sca( char *buffer, char terminate )`**

  Checks the receive buffer for a message terminated with the character **`terminate`**. The message is copied to **`buffer`**. The terminating character is discarded and the message in the buffer is terminated with a null character according to the C convention.

  RETURN VALUE:
  0 if no message was found with the specified terminating character.
  1 if a message has been extracted successfully from the buffer.

- **`int Dread_sca1ch( char *data )`**

  Reads a character from the serial receive buffer.

  PARAMETER:
  `data` is pointer to a character.

  RETURN VALUE:
  0 if the buffer is empty.
  1 if a byte has been extracted successfully from the buffer.

- **`int Dwrite_sca1ch( char ch )`**

  Places character **`ch`** in the transmit buffer. If SCC port A is not already transmitting, the function initiates transmission.

  RETURN VALUE:
  0 if the transmit buffer did not have space for **`ch.`**
  1 if the write was successful.

- **`void Dscamodem_chk( char *buffer )`**

  Checks the **`buffer`** for valid modem commands. The function takes the appropriate response to the modem command if it finds a valid modem command.

  RETURN VALUE:
  0 if a valid modem command is found.
  –1 if an invalid modem command is found.

- **`int Dxmodem_scadown( char *buffer, int count )`**

  Sends (downloads) **`count`** 128-byte blocks in **`buffer`** using the XMODEM protocol.

  RETURN VALUE:
  0 timed-out (no transfer).
  1 successful transfer.
  2 transfer canceled by receiver.

- **`int Dxmodem_scaup( unsigned long address,`**
  `        int *pages, int dest, int(*parser)() )`

  Receives (uploads) a file using the XMODEM protocol.

  PARAMETERS:
  `address` is the physical address in RAM where the received data are to be stored. If the receive buffer is allocated by **`xdata`**, then the name of the array may be used for the **`address`** argument. If, however, the data area is allocated using "normal" C, you must first convert the logical address of the buffer to a physical address using the library function **`phy_adr`**.
  `pages` is the number of 4K blocks of data that have been transferred.
  `dest` If 0, the upload is intended for the master in an RS-485 master-slave network. If **`dest`** is nonzero, the upload is intended for the designated slave (1–255).
  `parser` is the function that handles parsing of the uploaded data.

  RETURN VALUE:
  0 timed-out (no transfer).
  1 successful transfer.
  2 transfer canceled by sender side.

- **`void scbbinaryset( void )`**

  Puts the serial receiver in BINARY mode. This means that ***all*** received characters are placed in the receive buffer.

- **`void scbbinaryreset( void )`**

  Places the serial receiver in ASCII mode, where the BACKSPACE character (0x08) is parsed out of the receive buffer. Character echo also resumes if it was selected.

- **`int scbmodemstat( void )`**

  Returns the status of the modem.

  RETURN VALUE:
  1 if the modem is in command mode.
  0 if the modem is in data mode (i.e., open to communication).

- **`int scbmodemset( void )`**

  Returns information about modem selection.

  RETURN VALUE:
  1 if the modem option is selected.
  0 otherwise.

- **`void Dscbsend_prompt( void )`**

  Places CR, LF and > in the transmit buffer.

- **`int Dwrite_scb( char *buffer, int count )`**

  Copies **`count`** bytes from **`buffer`** to the transmit buffer. If SCC port B is not already transmitting, the function initiates transmission.

  RETURN VALUE:
  0 if the transmit buffer did not have space for **`count`** bytes.
  1 if the write was successful.

- **`int Dread_scb( char *buffer, char terminate )`**

  Checks the receive buffer for a message terminated with the character **`terminate`**. The message is copied to **`buffer`**. The terminating character is discarded and the message in the buffer is terminated with a null character according to the C convention.

  RETURN VALUE:
  0 if no message was found with the specified terminating character.
  1 if a message has been successfully extracted from buffer.

- **`int Dread_scb1ch( char *data )`**

  Reads a character from the serial receive buffer.

  PARAMETER:
  `data` is pointer to a character.

  RETURN VALUE:
  0 if the buffer is empty.
  1 if a byte has been extracted successfully from the buffer.

- **`int Dwrite_scb1ch( char ch )`**

  Places character **`ch`** in the transmit buffer. If SCC port B is not already transmitting, the function initiates transmission.

  RETURN VALUE:
  0 if the transmit buffer did not have space for **`ch`**.
  1 if the write was successful.

- **`void Dscbmodem_chk( char *buffer )`**

  Checks the **`buffer`** for valid modem commands. The function takes the appropriate response to the modem command if it finds a valid modem command.

  RETURN VALUE:
  0 if a valid modem command is found.
  –1 if an invalid modem command is found.

- **`int Dxmodem_scbdown( char *buffer, int count )`**

  Sends (downloads) **`count`** 128-byte blocks in **`buffer`** using the XMODEM protocol.

  RETURN VALUE:
  0 timed-out (no transfer).
  1 successful transfer.
  2 transfer canceled by receiver.

- **`int Dxmodem_scbup( unsigned long address,`**
  `        int *pages, int dest, int(*parser)() )`

  Receives (uploads) a file using the XMODEM protocol.

  PARAMETERS:
  `address` is the physical address in RAM where the received data are to be stored. If the receive buffer is allocated by **`xdata`**, then the name of the array may be used for the **`address`** argument. If, however, the data area is allocated using "normal" C, you must first convert the logical address of the buffer to a physical address using the library function **`phy_adr`**.
  `pages` is the number of 4K blocks of data that have been transferred.
  `dest` If 0, the upload is intended for the master in an RS-485 master-slave network. If **`dest`** is nonzero, the upload is intended for the designated slave (1–255).
  `parser` is the function that handles parsing of the uploaded data.

  RETURN VALUE:
  0 timed-out (no transfer).
  1 successful transfer.
  2 transfer canceled by sender side.

SERIAL.LIB

These are serial driver functions.

- **`void ser_init_z1( char mode, char baud )`**

  Initializes the driver for Z180 serial port 1. To use this driver, you must use **`z1_ser_int`** as the interrupt handler for Z180 port 1. This function uses circular receive and transmit buffers, which are allocated by the programmer. This function tells the software what the setup is.

  PARAMETERS:
  `mode` selects the operation mode as follows.

bit 0	0 = 1 stop bit 1 = 2 stop bits
bit 1	0 = no parity 1 = with parity

bit 2	0 = 7 data bits 1 = 8 data bits
bit 3	0 = even parity 1 = odd parity
bit 4	0 = no CTS/RTS control 1 = CTS/RTS enabled

`baud` selects the baud rate in multiples of 1200 bps. Valid multipliers are 1, 2, 4, 8, 16, 24, 32, 48 and 64. Pass a value of 8 to get 9600 bps.

Refer to **`ser_send_z1`** for sending information, **`ser_rec_z1`** for receiving information, and **`ser_kill_z1`** for aborting operations.

- **`void ser_send_z1( char *buf, char *count )`**

  Initializes driver to begin sending information.

  PARAMETERS:
  `buf` points to an array that contains the information to be sent.
  `count` points to a count variable that counts how many bytes remain to be sent. When **`*count`** becomes zero, the transmission is finished. The program should poll **`*count`** periodically to check whether the transmission is finished. A time-out mechanism is recommended to detect transmission failure.

- **`void ser_rec_z1( char *buf, char *count )`**

  Initializes driver to begin receiving information.

  PARAMETERS:
  `buf` points to an array where the information will be stored.
  `count` points to a count variable that counts how many bytes remain to be received. When **`*count`** becomes zero, the transmission is finished. The program should poll **`*count`** periodically to check whether the reception is finished. A time-out mechanism is recommended to detect reception failure.

- **`void ser_kill_z1( void )`**

  Aborts all operations for Z180 serial port 1. This function stops the receiver and the transmitter, but does not reset the counters associated with the transmitter and receiver.

- **`void ser_init_z0( char mode, char baud )`**

  Similar to **`ser_init_z1`**, but handles serial port 0 on the Z180.

- **`void ser_send_z0( char *buf, char *count )`**

  Similar to **`ser_send_z1`**, but handles serial port 0 on the Z180.

- **`void ser_rec_z0( char *buf, char *count )`**

  Similar to **`ser_rec_z1`**, but handles serial port 0 on the Z180.

- **`void ser_kill_z0( void )`**

  Similar to **`ser_kill_z1`**, but handles serial port 0 on the Z180.

- **`void ser_init_s0( char mode, char baud )`**

  Similar to **`ser_init_z1`**, but handles SIO port A.

- **`void ser_send_s0( char *buf, char *count )`**

  Similar to **`ser_send_z1`**, but handles SIO port A.

- **`void ser_rec_s0( char *buf, char *count )`**

  Similar to **`ser_rec_z1`**, but handles SIO port A.

- **`void ser_kill_s0()`**

  Similar to **`ser_kill_z1`**, but handles SIO port A.

- **`void ser_init_s1( char mode, char baud )`**

  Similar to **`ser_init_z1`**, but handles SIO port B.

- **`void ser_send_s1( char *buf, char *count )`**

  Similar to **`ser_send_z1`**, but handles SIO port B.

- **`void ser_rec_s1( char *buf, char *count )`**

  Similar to **`ser_rec_z1`**, but handles SIO port B.

- **`void ser_kill_s1()`**

  Similar to **`ser_kill_z1`**, but deals with SIO port B.

S0232.LIB

These are RS-232 functions for the BL1100's KIO serial port A (first port on KIO). This library is only for the BL1100.

- **`void s0binaryset( void )`**

  Puts the serial receiver in BINARY mode. This means that ***all*** received characters are placed in the receive buffer.

- **`void s0binaryreset( void )`**

  Places the serial receiver in ASCII mode, where the BACKSPACE character (0x08) is parsed out of the receive buffer. Character echo also resumes if it was selected.

- **`int s0modemstat( void )`**

  Returns the status of the modem.

  RETURN VALUE:
  1 if the modem is in command mode.
  0 if the modem is in data mode (i.e., open to communication).

- **`int s0modemset( void )`**

  Returns information about modem selection.

  RETURN VALUE:
  1 if the modem option is selected.
  0 otherwise.

- **`void Ds0send_prompt( void )`**

  Places CR, LF and > in the transmit buffer.

- **`int Dinit_s0( void *rbuf, void *tbuf,`**
  `int rsize, int tsize, char mode,`
  `char baud, byte ismodem, byte isecho )`

  Initializes SIO port 0 for communication. This function uses circular receive and transmit buffers, which are allocated by the programmer. This function tells the software what the setup is.

  PARAMETERS:
  `rbuf` is a pointer to the receive buffer.
  `tbuf` is apointer to the transmit buffer
  `rsize` is the size, in bytes, of the receive buffer.
  `tsize` is the size, in bytes, of the transmit buffer.
  `mode` selects communication criteria as follows.

bit 0	0 = 1 stop bit 1 = 2 stop bits
bit 1	0 = no parity 1 = with parity
bit 2	0 = 7 data bits 1 = 8 data bits
bit 3	0 = even parity 1 = odd parity
bit 4	0 = no CTS/RTS control 1 = CTS/RTS enabled

  `baud` selects the baud rate in multiples of 1200 bps. Valid multipliers are 1, 2, 4, 8, 16, 24, 32, 48 and 64. Pass a value of 8 to get 9600 bps.
  `ismodem` if 1, modem communication is supported. Otherwise is 0.
  `isecho` if 1, every character is echoed. Otherwise is 0.

If CTS/RTS handshaking is selected, transmission from the sender is disabled (by raising RTS) when the receive buffer is 80% full. The software lowers RTS (enabling the sender to transmit) when the receive buffer falls below 20% of capacity. In a similar manner, a remote system can prevent transmission of data by SIO port 0 by asserting its RTS (connected to the SCC port 0 CTS).

RETURN VALUE: always 1.

- **`void Dreset_s0rbuf( void )`**

  Resets the receive buffer.

- **`void Dreset_s0tbuf( void )`**

  Resets the transmit buffer and stops transmission.

- **`int Dwrite_s0( char *buffer, int count )`**

  Copies **`count`** bytes from **`buffer`** to the transmit buffer. If KIO serial port 0 is not already transmitting, the function initiates transmission.

  RETURN VALUE:
  0 if the transmit buffer did not have space for **`count`** bytes.
  1 if the write is successful.

- **`int Dread_s0( char *buffer, char terminate )`**

  Checks the receive buffer for a message terminated with the character **`terminate`**. The message is copied to **`buffer`**. The terminating character is discarded and the message in the buffer is terminated with a null character according to the C convention.

  RETURN VALUE:
  0 if no message was found with the specified terminating character.
  1 if a message has been successfully extracted from buffer.

- **`int Dwrite_s01ch( char ch )`**

  Places character **`ch`** in the transmit buffer. If KIO serial port A is not already transmitting, the function initiates transmission.

  RETURN VALUE:
  0 if the transmit buffer did not have space for **`ch`**.
  1 if the write was successful.

- **`int Dread_s01ch( char *data )`**

  Reads a character from the serial receive buffer.

  PARAMETER:
  `data` is pointer to a character.

  RETURN VALUE:
  0 if the buffer is empty.
  1 if a byte has been extracted successfully from the buffer.

- **`void Dkill_s0( void )`**

  Disables SIO port 0.

- **`void Drestart_s0modem( void )`**

  Restarts the modem during start-up or because of abnormal operation in SIO serial port 0.

  A Hayes SmartModem or compatible modem is recommended. A ***null*** modem connection is needed between the BL1100 and the modem for the TX and RD lines since both the BL1100's serial port and the modem are data communication equipment (DCE). A commercial ***null*** modem would have its CTS and RTS lines tied together right away on both sides. Some modems require that the RTS, CTS, and DTR lines be tied together.

- **`int Ds0modem_chk( char *buffer )`**

  Checks the **`buffer`** for valid modem commands. The function takes the appropriate response to the modem command if it finds a valid modem command.

  RETURN VALUE:
  0 if a valid modem command is found.
  –1 if an invalid modem command is found.

- **`void Ds0_circ_int( void )`**

  This is an interrupt service routine for SIO port 0.

- **`int Dxmodem_s0down( char *buffer, int count )`**

  Sends (downloads) **`count`** 128-byte blocks in **`buffer`** using the XMODEM protocol.

  RETURN VALUE:
  0 timed-out (no transfer).
  1 successful transfer.
  2 transfer canceled by receiver.

- **`int Dxmodem_s0up ( unsigned long address,`**
  `int *pages, int dest, int(*parser)() )`

  Receives (uploads) a file using the XMODEM protocol.

  PARAMETERS:
  `address` is the physical address in RAM where the received data are to be stored. If the receive buffer is allocated by **`xdata`**, then the name of the array may be used for the **`address`** argument. If, however, the data area is allocated using "normal" C, you must first convert the logical address of the buffer to a physical address using the library function **`phy_adr`**.

pages is the number of 4K blocks of data that have been transferred.
dest If 0, the upload is intended for the master in an RS-485 master-slave network. If **dest** is nonzero, the upload is intended for the designated slave (1–255).
parser is the function that handles parsing of the uploaded data.

RETURN VALUE:
0 timed-out (no transfer).
1 successful transfer.
2 transfer canceled by sender side.

S1232.LIB

These are RS-232 functions for the BL1100's KIO serial port B. The functions in this library are analogous to the functions in **S0232.LIB**. Just replace the "**0**" in that library's function name with "**1**" to get the corresponding function for this library.

Z0232.LIB

These are drivers for Z180 port 0. Be sure to include the following call before initializing Z180 port 0 regardless of whether application development is through the SIB 2 or direct.

```
reload_vec(14,Dz0_circ_int);
```

Depending on your application, it may be desirable to delay initialization of the serial port to make sure your hardware is connected. Alternatively, an external trigger from a keypad or input port could signal the software to initialize the serial port.

- **void z0binaryset(void)**

  Puts the serial receiver in BINARY mode. This means that ***all*** received characters are placed in the receive buffer.

- **void z0binaryreset(void)**

  Places the serial receiver in ASCII mode, where the BACKSPACE character (0x08) is parsed out of the receive buffer. Character echo also resumes if it was selected.

- **int z0modemstat(void)**

  Returns the status of the modem.

  RETURN VALUE:
  1 if the modem is in command mode.
  0 if the modem is in data mode (i.e., open to communication).

- **`int z0modemset( void )`**

  Returns information about modem selection.

  RETURN VALUE:
  1 if the modem option is selected.
  0 otherwise.

- **`void Dz0send_prompt( void )`**

  Places CR, LF and > in the transmit buffer.

-
  ```
  int Dinit_z0( void *rbuf, void *tbuf,
          int rsize, int tsize, char mode,
          char baud, char ismodem, char isecho )
  ```

  Initializes Z180 port 0 for communication. This function uses circular receive and transmit buffers, which are allocated by the programmer. This function tells the software what the setup is.

  PARAMETERS:
  `rbuf` is a pointer to the receive buffer.
  `tbuf` is apointer to the transmit buffer
  `rsize` is the size, in bytes, of the receive buffer.
  `tsize` is the size, in bytes, of the transmit buffer.
  `mode` selects communication criteria as follows.

bit 0	0 = 1 stop bit
	1 = 2 stop bits
bit 1	0 = no parity
	1 = with parity
bit 2	0 = 7 data bits
	1 = 8 data bits
bit 3	0 = even parity
	1 = odd parity
bit 4	0 = no CTS/RTS control
	1 = CTS/RTS enabled

  `baud` selects the baud rate in multiples of 1200 bps. Valid multipliers are 1, 2, 4, 8, 16, 24, 32, 48 and 64. Pass a value of 8 to get 9600 bps.
  `ismodem` if 1, modem communication is supported. Otherwise is 0.
  `isecho` if 1, every character is echoed. Otherwise is 0.
  If CTS/RTS handshaking is selected, transmission from the sender is disabled (by raising RTS) when the receive buffer is 80% full. The software lowers RTS (enabling the sender to transmit) when the receive buffer falls below 20% of capacity. In a similar manner, a remote system can prevent transmission of data by Z180 port 0 by asserting its RTS (connected to the Z180 port 0 CTS).

  RETURN VALUE: always 1.

- **`void Dreset_z0rbuf( void )`**

  Resets the receive buffer.

- **`void Dreset_z0tbuf( void )`**

  Resets the transmit buffer and stop transmission.

- **`int Dwrite_z0( char *buffer, int count )`**

  Copies **`count`** bytes from **`buffer`** to the transmit buffer. If Z180 port 0 is not already transmitting, the function initiates transmission.

  RETURN VALUE:
  0 if the transmit buffer did not have space for **`count`** bytes.
  1 if the write was successful.

- **`int Dread_z0( char *buffer, char terminate )`**

  Checks the receive buffer for a message terminated with the character **`terminate`**. The message is copied to **`buffer`**. The terminating character is discarded and the message in the buffer is terminated with a null character according to the C convention.

  RETURN VALUE:
  0 if no message was found with the specified terminating character.
  1 if a message has been extracted successfully from the buffer.

- **`int Dwrite_z01ch( char ch )`**

  Places character **`ch`** in the transmit buffer. If Z180 port 0 is not already transmitting, the function initiates transmission.

  RETURN VALUE:
  0 if the transmit buffer did not have space for **`ch`**.
  1 if the write was successful.

- **`int Dread_z01ch( char *data )`**

  Reads a character from the serial receive buffer.

  PARAMETER:
  `data` is pointer to a character.

  RETURN VALUE:
  0 if the buffer is empty.
  1 if a byte has been extracted successfully from the buffer.

- **`void Dkill_z0( void )`**

  Disables Z180 port 0.

- **`void Drestart_z0modem( void )`**

  Restarts a modem during start-up or because of abnormal operation in Z180 port 0.

  A Hayes SmartModem or compatible modem is recommended. A ***null*** connection is also required for the TX and RD lines since both the controller's serial port and the modem are data communication equipment (DCE). A commercial NULL modem would have its CTS and RTS lines tied together right away on both sides. Some modems require that the RTS, CTS, and DTR lines be tied together on the modem side. The CTS and RTS lines on the controller side also have to be tied together.

- **`void Dz0modem_chk( char *buffer )`**

  Checks the **`buffer`** for valid modem commands. The function takes the appropriate response to the modem command if it finds a valid modem command.

  RETURN VALUE:
  0 if a valid modem command is found.
  –1 if an invalid modem command is found.

- **`void Dz0_circ_int( void )`**

  This is an interrupt service routine for Z180 port 0.

- **`int Dxmodem_z0down( char *buffer, int count )`**

  Sends (downloads) **`count`** 128-byte blocks in **`buffer`** using the XMODEM protocol.

  RETURN VALUE:
  0 timed-out (no transfer).
  1 successful transfer.
  2 transfer canceled by receiver.

- **`int Dxmodem_z0up ( unsigned long address,`**
  `        int *pages, int dest, int(*parser)() )`

  Receives (uploads) a file using the XMODEM protocol.

  PARAMETERS:
  `address` is the physical address in RAM where the received data are to be stored. If the receive buffer is allocated by **`xdata`**, then the name of the array may be used for the **`address`** argument. If, however, the data area is allocated using "normal" C, you must first convert the logical address of the buffer to a physical address using the library function **`phy_adr`**.
  `pages` is the number of 4K blocks of data that have been transferred.
  `dest` If 0, the upload is intended for the master in an RS-485 master-slave network. If **`dest`** is nonzero, the upload is intended for the designated slave (1–255).
  `parser` is the function that handles parsing of the uploaded data.

  RETURN VALUE:
  0 timed-out (no transfer).
  1 successful transfer.
  2 transfer canceled by sender side.

Z1232.LIB

These are the drivers for Z180 port 1. The functions in this library are **almost** exactly analogous to the functions in **`Z0232.LIB`**. Replace the "`0`" in that library's function name with "`1`" to get the corresponding function names in this library. The only difference is that because CTS/RTS handshake is not possible on the Z1 port, any reference to it should be ignored.

CHAPTER 5: MODBUS SLAVE LIBRARIES

Modbus is the generic name for two serial communication protocols, Modbus ASCII and Modbus RTU, originally developed by Modicon, Inc., for communication between programmable logic controllers (PLCs). Both protocols are easily implemented on standard asynchronous serial hardware. Its ease of implementation has made Modbus the most accepted of the asynchronous protocols for industrial inputs/outputs.

Z-World's Modbus slave libraries allow existing and new applications to act as slaves on a Modbus network. Both the Modbus ASCII and Modbus RTU protocols are supported.

Getting Started

Z-World's Modbus Slave libraries consist of two files, **`MS.LIB`** and **`MSZ.LIB`**.

- **`MS.LIB`** is the library that actually performs most Modbus operations. Among other things, **`MS.LIB`** decodes command packets from the network master, dispatches read/write commands to user-definable C function stubs, and encodes replies for transmission to the network master.
- **`MSZ.LIB`** provides easy-to-use standard drivers for the serial ports of the Z180. In accordance with the typical usage of the Z180 UARTs on Z-World controllers, **`MSZ.LIB`** drivers for Serial Port Z0 implement full-duplex RS-232 communication, and drivers for Serial Port Z1 implement half-duplex (two-wire) RS-485 communication. **`MSZ.LIB`** supports both the ASCII and RTU protocols on each port.

Standard Modbus Slave Procedure

Use the standard procedure to implement the Modbus interface as RS-232 on Serial Port Z0 or as RS-485 on Serial Port Z1. **`MSZ.LIB`** allows relatively simple implementation of a Modbus interface. The following five steps will allow you to add Modbus slave support to an existing or new application.

Step 1: Use `MSZ.LIB`

The following line must appear near the beginning of your program (typically after the opening comments) in order to use **`MSZ.LIB`**.

```
#use "MSZ.LIB"
```

Note that **`MSZ.LIB`** automatically uses **`MS.LIB`**, so a second **`#use`** is not required.

Step 2: Call `_GLOBAL_INIT`

`MS.LIB` has several global initializations that must be performed and also uses costatements. Therefore, the function **`_GLOBAL_INIT`** must be called during the initialization of the controller. This can be done by placing the following statement within your initialization code.

```
_GLOBAL_INIT ();
```

Note that the initialization functions **`uplc_init`** and **`VdInit`** call **`_GLOBAL_INIT`** directly. Thus, **`_GLOBAL_INIT`** does not have to be called explicitly if either of these initialization functions are used.

Refer to the ***Dynamic C 32 Application Frameworks*** manual for more information on costatements.

Step 3: Initialize the Modbus Serial Port

`MSZ.LIB` contains four initialization functions. Call one of these during initialization. The choice of which function to select depends on the protocol (Modbus ASCII or Modbus RTU) and on the serial port (Z0 or Z1). Note that Serial Port Z0 is assumed to be RS-232 and Serial Port Z1 is assumed to be half-duplex (two-wire) RS-485. Table 1 lists the four initialization functions.

Table 1. MSZ.LIB Initialization Functions

Function	Protocol	Serial Port/Type
`msaZ0`	Modbus ASCII	RS-232 on Z0
`msrZ0`	Modbus RTU	
`msaZ1`	Modbus ASCII	RS-485 on Z1
`msrZ1`	Modbus RTU	

Each of the `msaZ0`, `msrZ0`, `msaZ1` and `msrZ1` functions use the same calling conventions, as shown in the example below for `msaZ0`.

- **`int msaZ0(unsigned Addr, unsigned long Baud, unsigned Mode)`**

  `Addr` is the Modbus address of this slave. It should be set for a value between 1 and 255, with 0 being reserved as the address of messages broadcast to all slaves on the network. It is your responsibility to ensure that no two nodes (Z-World controller or other) on the Modbus network share the same address.

  `Baud` is the desired baud rate of the Modbus interface. Many of the standard baud rates are supported, but 9600 bps and 19,200 bps are the most common. Reliable communications at baud rates beyond 19,200 bps cannot be guaranteed in applications with a high multitasking density or in environments where serial communication is subject to noise. Also note that limitations in the baud-rate generators of the Z180 restrict which baud rates (even common baud rates) are attainable. The initialization will fail if attempts are made to select illegal baud rates, such as 38,400 bps on a 9.216 MHz Z180.

  `Mode` sets the desired serial character frame parameters according to the following list. Modbus RTU requires 8 data bits, and unlisted values default to 8-N-1 for both ASCII and RTU. All serial character frames listed below begin with one start bit. Eight data bits with no parity and either one or two stop bits are the most common Modbus settings, and should be tried first on existing systems with unknown parity settings.

0 - 8 bit data, no parity, 1 stop bit (default, ASCII and RTU)
1 - 7 bit data, odd parity, 1 stop bit (ASCII only)
2 - 7 bit data, even parity, 1 stop bit (ASCII only)
3 - 7 bit data, no parity, 2 stop bits (ASCII only)
4 - 8 bit data, odd parity, 1 stop bit (ASCII and RTU)
5 - 8 bit data, even parity, 1 stop bit (ASCII and RTU)
6 - 8 bit data, no parity, 2 stop bits (ASCII and RTU)
7 - 7 bit data, odd parity, 2 stop bits (ASCII only)
8 - 7 bit data, even parity, 2 stop bits (ASCII only)
9 - 8 bit data, odd parity, 2 stop bits (ASCII and RTU)
10 - 8 bit data, even parity, 2 stop bits (ASCII and RTU)
11 - 7 bit data, no parity, 1 stop bit (ASCII only)

Each initialization function returns true (non-zero) if the function succeeds in initializing the serial port. False (zero) is returned if the initialization fails. This is usually the result of selecting an unattainable baud rate.

Step 4: Call `msRun` Periodically

The function **`msRun`** decodes command packets from the network master, dispatches read/write commands to user-definable C function stubs, and encodes replies for transmission to the network master. Since **`msRun`** controls the flow of data between the Z-World controller and the Modbus network, care must be exercised in how frequently **`msRun`** is called.

Modbus RTU has an additional timing requirement that Modbus ASCII does not have. In order to meet this timing requirement, the Modbus RTU drivers (**`msrZ0`** and **`msrZ1`**) use Programmable Reload Timer 0 (PRT0) of the Z180. Make sure this does not conflict with your existing application or with code you add to the application in the future. For more details, see the section later in this chapter on the high-resolution timer.

The first consideration is the integrity of the serial data. If serial drivers have little or no buffering, then received characters must be processed promptly, or incoming bytes will be lost. Modbus RTU also uses timing for packet delimiting—any gap of 3.5 or more characters in serial data (transmitted or received) is seen as a packet delimiter. In such circumstances, **`msRun`** should be called nearly constantly. If the Z-World controller has little else to do, this might provide an acceptable solution and would allow for a simple user-defined serial driver.

The serial drivers provided in **`MSZ.LIB`** are fully buffered. As such, delays of 25 ms, 100 ms, or even more can be tolerated between calls to **`msRun`** without a loss of serial integrity. It should be noted, however, that Modbus network masters commonly implement a simple timeout. Delaying a reply to a packet by not calling **`msRun`** frequently enough may result in an uncommonly large number of network errors.

Step 5: Write Modbus Slave Command Handlers

Once you reach this point successfully, your application should compile and run. However, every request by the Modbus master will still return an invalid address error. Why?

Z-World's Modbus Slave Driver allows elements of a C application to be arbitrarily mapped onto the Modbus registers. This is accomplished by Modbus handler functions such as **`msIn`**, **`msOut`**, and **`msRead`**. If these functions do not appear in your application, then default handlers in **`MS.LIB`** indicate that no valid registers of that particular register type are available on this slave.

For more information, check out the **BL15MS.C**, **BL17MS.C**, **LP31MS.C** and **PK22MS.C** sample programs and read the sections later in this chapter on the Modbus Registers and the Modbus Slave Command Handlers.

Advanced Modbus Slave Procedure

If you require something other than full-duplex RS-232 communication on Serial Port Z0 or half-duplex RS-485 communication on Serial Port Z1, Z-World's Modbus Driver allows you to implement Modbus on any serial port. Doing so requires almost the same steps that are required to use **`MSZ.LIB`**, but requires the additional step of writing a serial driver and possibly a timer for the driver in conjunction with the **`MS.LIB`** library.

Step 1: Use `MS.LIB`

The following line must appear near the beginning of your program (typically after the opening comments) in order to use **`MS.LIB`**.

```
#use "MS.LIB"
```

Steps 2–5

These steps are the same as for the Standard Modbus Slave Procedure.

Step 6: Write A Modbus Slave Compatible Serial Driver

`MS.LIB` uses a generic model for the serial device it uses to interface to the Modbus network. Any device supplying the required functions can be used to interface to a Modbus network. For more details, see the section later in this chapter on the Modbus Serial Interface.

Modbus Registers

In C, actions and manipulation of data are performed by functions, while data are stored in variables. This organization dominates most programming languages and is so logical as to be intuitive to C programmers.

PLCs, however, do not operate on this principle. In PLCs, actions are performed by manipulating coils and registers, which also serve as storage for data. This uniformity is initially counter-intuitive to C programmers, but actually make for a simple interface.

Modbus simplifies this even further by supporting only two data types. Coils (referring to the coils in mechanical relays) store true/false information, a “bit” in common computer technology. Registers store 16-bit unsigned numbers.

The Modbus protocol provides for five classes of objects that can be manipulated. Since each class is assigned a unique, nonoverlapping address range, these objects are often referred to by their address space.

0X References (Discrete Outputs)

These bits are readable and writable. Some are used to control the PLC outputs , some are used to store internal bits, and others perform special PLC operations.

1X References (Discrete Inputs)

These bits are read only. They are used mainly for the digital inputs and to check the PLC status.

3X References (Input Registers)

These registers are read only. They are used mainly for multi-bit inputs such as analog/digital readings and pulse measurement readings.

4X References (Holding Registers)

These registers are readable and writable. They are used primary to hold data and for multi-bit outputs (such as digital/analog).

6X References (Extended Memory)

These registers are not supported by Dynamic C’s Modbus Slave Driver nor are they supported by most Modbus-compatible devices. In fact, even Modicon’s use of 6X registers is so nonstandard as to make a general-purpose driver difficult to implement.

Modbus Slave Command Handlers

The last step in implementing the Modbus driver is to define the functions used to perform read and write operations on the Modbus registers. These functions are called as needed by the Modbus driver while processing commands from the Modbus master. The functions are used to map the Z-World resources (I/O, variables and functions) to the Modbus paradigm.

You may define only the functions as you need, and any functions left undefined will be handled by dummy stubs in the library and reported as errors. While each function has a unique set of parameters, all return a common set values:

- **`MS_BADADDR`** is returned when the register or coil address is unsupported.
- **`MS_BADDATA`** is returned if a write command supplies data that are illegal for the addressed register or coil.
- 0 is returned if the operation can be performed successfully.

The functions used to perform read and write operations are listed below.

- **`void msStart(void)`**

  `msStart` is called just before a received packet is processed. While this can be used for any purpose, it is mainly intended to "lock" Modbus resources so that data returned in one packet are atomic.

- **`void msDone(void)`**

  `msDone` is called after the Modbus command has been processed and just before the reply is sent. **`msDone`** is primarily intended to "unlock" Modbus resources locked by msStart.

- **`int msIn(unsigned Coil, int *State)`**

  `msIn` is called to read the specified input **`Coil`** (1X reference). The coil's current state (0 for off and 1 for on) is stored to the int pointed to by **`State`**. Note that **`State`** is a pointer, thus it is necessary to precede **`State`** with an asterisk when making an assignment (i.e., **`*State`** = 1;).

The following function treats the PK2200 inputs (1…16) as Modbus input coils (0…15).

```
int msIn(unsigned Coil, int *State) {
  if ((0 <= Coil) && (Coil <= 15)) {
    *State = up_digin(Coil + 1);
    return 0;
  }
  return MS_BADADDR;
}
```

- **`int msOutRd(unsigned Coil, int *State)`**

  `msOutRd` is called to read the specified output **`Coil`** (0X reference), and operates identically to **`msIn`**.

- **`int msOutWr(unsigned Coil, int State)`**

  `msOutWr` is called to write the specified output **`Coil`** (0X reference). **`State`** is 0 if the output is to be "off," and 1 if the output is to be "on."

The following function treats the PK2200 outputs (1…14) as Modbus output coils (0…13).

```
int msOutWr(unsigned Coil, int State) {
  if ((0 <= Coil) && (Coil <= 13)) {
    up_setout(Coil + 1, State);
    return 0;
  }
  return MS_BADADDR;
}
```

- **`int msInput(unsigned Reg, unsigned *Value)`**

  `msInput` is called to read the specified input register **`Reg`** (3X reference). The input's current value is stored to the integer pointed to by **`Value`**. Since **`Value`** is a pointer, precede **`Value`** with an asterisk when making an assignment (i.e., **`*Value`** = 1;).

The following function maps Modbus input registers to the universal inputs of the PK2100 (1…6). Input registers (0…5) return calibrated input values (0…10000 represent 0 V…10 V), and input registers (16…21) return raw input values (0…1024 represent 0…VRef).

```
int msInput(unsigned Reg, unsigned *Value) {
  if ((0 <= Reg) && (Reg <= 5)) {
    *Value = up_adcal(Reg + 1);
    return 0;
  }
  if ((16 <= Reg) && (Reg <= 21)) {
    *Value = up_adraw(Reg - 15);
    return 0;
  }
return MS_BADADDR;
}
```

- **`int msRead(unsigned  Reg, unsigned *Value)`**

  `msRead` is called to read the specified holding register **`Reg`** (4X reference). The holding register's current value is stored to the unsigned integer pointed to by **`Value`**. Since **`Value`** is a pointer, it is necessary to precede **`Value`** with an asterisk when making an assignment (i.e., **`*Value`** = 1;).

- **`int msWrite(unsigned Reg, unsigned Value)`**

  `msWrite` is called to write **`Value`** to the specified holding register **`Reg`** (4X reference).

The following functions map the variables x, y and z onto holding registers 10, 20 and 30.

```
int msRead(unsigned Reg, unsigned *Value) {
  switch (Reg) {
    case 10: *Value = x; break;
    case 20: *Value = y; break;
    case 30: *Value = z; break;
    default: return MS_BADADDR;
  }
  return 0;
}
int msWrite(unsigned Reg, unsigned Value) {
  switch (Reg) {
    case 10: x = Value; break;
    case 20: y = Value; break;
    case 30: z = Value; break;
    default: return MS_BADADDR;
  }
return 0;
}
```

Modbus Slave Serial Interface

If the serial driver supplied in **`MSZ.LIB`** is not suited to your application, it is fairly straightforward to write your own serial driver. **`MS.LIB`** uses a standard set of functions to interface to the Modbus network, so writing new functions as described below will enable **`MS.LIB`** to talk to your Modbus network.

- **`void msaInit(unsigned Addr)`**
- **`void msrInit(unsigned Addr, unsigned Timeout)`**
  Call **`msaInit`** to initialize the slave as a Modbus ASCII device, or call **`msrInit`** to initialize the slave as a Modbus RTU device. Both functions take the **`Addr`** parameter, which defines the slave's Modbus address. **`Addr`** is a value from 1 to 255, with 0 reserved as the address for broadcast messages.

  `msrInit` requires the additional parameter **`Timeout`**. This is the number of "RTU ticks" that constitute an RTU timeout, which is equal to the period of 3.5 bytes. For more information on RTU ticks, consult the description of the msTimer function description.

- **`void msError(void)`**

  Call **`msError`** whenever an error is detected on the serial port to abort processing of the current packet. **`msError`** affects only the HL and AF registers, and can be called from assembly language as well as from C.

- **`void msRecv(int Byte)`**

  Call **`msRecv`** for each byte received by the Modbus interface. **`msRecv`** affects only the HL and AF registers, and can be called from assembly language as well as from C. The value of the received byte should be passed in the L register (H is ignored).

- **`int msSend(char *Reply, unsigned Len)`**

  Your Modbus interface must supply a function, **`msSend`**, to send **`Len`** bytes from the **`Reply`** buffer to Modbus interface. Prior to the actually sending the reply, the Modbus driver calls **`msSend`** with a NULL value for the **`Reply`** parameter. This can be used to "reset" or "ready" the Modbus interface for sending a reply to the network. At the very minimum, **`msSend`** should ignore calls when **`Reply`** is NULL.

  The Modbus driver will call **`msSend`** as often as **`msRun`** is called until the reply has been sent completely. **`msSend`** does this by returning a false (zero) value until the reply packet has completed transmission, at which time **`msSend`** returns a true (non-zero) value.

- **`unsigned msTimer(void)`**

  If you call **`msrInit`**, Modbus RTU requires that some timing be performed on incoming bytes. This is because a 3.5-byte silence period delimits the packets. In order to provide this timing, you need to supply an **`msTimer`** function. This function returns a free-running 16-bit timer that does a full 16-bit count from 0x0000 to 0xFFFF. **`msTimer`** must count up (0, 1, 2, 3…65534, 65535, 0,…).

  Rather than force a timer on the user-defined Modbus interface, **`MS.LIB`** allows an arbitrary timer to be used. This is accomplished by forcing the user to not only supply the free-running timer, but to also define the 3.5-byte period (in their arbitrary time units) when the Modbus RTU driver is initialized. For example, if the baud rate is slow enough, it would be possible for someone to used the lower 16 bits of **`MS_TIMER`** (initialized and maintained by **`uplc_init`** and **`VdInit`**) to provide millisecond timing.

  The function **`msTimer`** should be written in assembly language because **`msTimer`** can only modify the HL and AF registers. Technically, it could be written in C by declaring the **`msTimer`** function as interrupt, but the overhead from this is prohibitive. The free-running 16-bit up count should be returned in the HL register.

`MSZ.LIB` uses a special 32-bit high resolution timer based on PRT0 of the Z180. This allows timing down to 20 system clocks, which is roughly 2.1 µs at 9.216 MHz.

High-Resolution Timer

If you're using the Modbus RTU drivers in **`MSZ.LIB`** (via **`msrZ0`** or **`msrZ1`**), then you have invoked a 32-bit free-running counter in the background using Programmable Reload Timer 0 (PRT0) of the Z180. While this timer is mandatory for the proper operation of the RTU drivers in **`MSZ.LIB`**, it can also provide a time base that is extremely useful for other purposes.

- **`void hrtInit(void)`**

  Initializes and zeroes the 32-bit high-resolution timer based on PRT0.

- **`unsigned long hrtRead(void)`**

  Returns the current value of the 32-bit high resolution timer based on PRT0. This timer is incremented once every 20 system clocks.

Modbus Slave Supported Commands

Modbus protocols were created by Modicon to communicate with their PLCs. As such, the Modbus protocols are filled with commands that are specific to Modicon products and are, therefore, not well-suited for general use. Thus, Z-World's Modbus slave libraries support only the following Modbus commands.

0x01 : Read Coil Status
0x02 : Read Input Status
0x03 : Read Holding Registers
0x04 : Read Input Registers
0x05 : Force Single Coil
0x06 : Preset Single Register
0x0B : Fetch Communication Event Counter
0x0F : Force Multiple Coils
0x10 : Preset Multiple Registers
0x16 : Mask Write 4X Register
0x17 : Read/Write 4X Registers

Modbus Slave Unsupported Commands

The following Modbus commands are ***not*** supported in Z-World's Modbus slave libraries. Please note that this is not an exhaustive list, as many vendors have added their own PLC-specific commands to the Modbus protocol.

0x07 : Read Exception Status
0x08 : Diagnostics
0x09 : Program 484
0x0A : Poll 484
0x0C : Fetch Communication Event Log
0x0D : Program Controller
0x0E : Poll Controller
0x11 : Report Slave ID
0x12 : Program 884/M84
0x13 : Reset Communication Link
0x14 : Read General Reference
0x15 : Write General Reference
0x18 : Read FIFO Queue

For more information on the Modbus protocol, check the ***Modicon Modbus Protocol Reference Guide*** (Modicon Document PI-MBUS-300). This can be found on the World Wide Web at the Modicon website at (http://www.modicon.com).

CHAPTER 6: MODBUS MASTER LIBRARIES

Modbus is the generic name for two serial communication protocols, Modbus ASCII and Modbus RTU, originally developed by Modicon, Inc., for communication between programmable logic controllers (PLCs). Both protocols are easily implemented on standard asynchronous serial hardware. Its ease of implementation has made Modbus the most accepted of the asynchronous protocols for industrial inputs/outputs.

Z-World's Modbus master libraries allow existing and new applications to be the master on a Modbus network. Both the Modbus ASCII and Modbus RTU protocols are supported.

Getting Started

Z-World's Modbus Master libraries consist of two files, **`MM.LIB`** and **`MMZ.LIB`**.

- **`MM.LIB`** implements the Modbus commands. Each supported command of the Modbus protocol has a function within **`MM.LIB`**. Each function creates a Modbus packet representing the command, sends it to the specified slave or broadcasts it to all slaves, and reads any reply returned by a slave. Commands also return diagnostic information, telling why commands in failed packets were not executed. Each function works equally well under Modbus ASCII or RTU.
- **`MMZ.LIB`** provides easy-to-use standard drivers for the serial ports of the Z180. It implements two separate serial interfaces, an RS-232 interface for serial port Z0 and a half-duplex (two-wire) RS-485 interface for serial port Z1. Since these are the common assignments of these ports on Z-World controllers, both drivers will work on most controllers and at least one will work on every controller. The serial drivers in **`MMZ.LIB`** support both Modbus ASCII and RTU protocols.

Standard Modbus Master Procedure

Use the standard procedure to implement the Modbus interface as RS-232 on serial port Z0 or as RS-485 on serial port Z1. The following four steps will allow you to add Modbus master support to an existing or new application. See the **`PK22MM.C`** sample program to test a selection of the Modbus commands and display information returned in Modbus slave replies. This sample program can be easily modified to run on any Z-World controller with a 2x20 LCD display.

Step 1: Use `MMZ.LIB`

The following line must appear near the beginning of your program (typically after the opening comments) in order to use **`MMZ.LIB`**.

```
#use "MMZ.LIB"
```

Note that **`MMZ.LIB`** automatically uses **`MM.LIB`**, so a second #use is not required.

Step 2: Call `VdInit or uplc_init`

The Modbus master functions in **`MM.LIB`** are based on costatements and use the **`DelayMs`** function to perform certain timeout procedures. As such, the appropriate one of either the **`VdInit`** function (in **`VDRIVER.LIB`**) or the **`uplc_init`** function (in **`CPLC.LIB`**) must be used to initialize the target controller.

Refer to the ***Dynamic C 32 Application Frameworks*** manual for more information on costatements.

Step 3: Initialize the Modbus Serial Port

`MMZ.LIB` contains four initialization functions, one of which must be called to initialize the Modbus master's serial port prior to executing Modbus commands. The function that is selected determines the Modbus master's serial port and standard (Z0/RS-232 or Z1/RS-485) as well as the protocol (ASCII or RTU). Table 6-1 lists the four initialization functions.

Table 6-1. MMZ.LIB Initialization Functions

Function	Protocol	Serial Port/Type
`mmaZ0`	Modbus ASCII	RS-232 on Z0
`mmrZ0`	Modbus RTU	
`mmaZ1`	Modbus ASCII	RS-485 on Z1
`mmrZ1`	Modbus RTU	

Each of the **`mmaZ0`**, **`mmrZ0`**, **`mmaZ1`** and **`mmrZ1`** functions use the same calling conventions, as shown in the example below for **`mmaZ0`**.

- **`int mmaZ0(unsigned long Baud, unsigned Mode)`**

  `Baud` specifies the communication rate (bps) for the serial port, and can be any value supported by the selected port. Communication rates of 9600 or 19200 bps are the most common, and rates above 19200 bps are generally unreliable in most industrial settings due to noise.

  `Mode` sets the desired serial character frame parameters according to the following list. Modbus RTU requires 8 data bits, and unlisted values default to 8-N-1 for both ASCII and RTU. All serial character frames listed below begin with one start bit. Eight data bits with no parity and either one or two stop bits are the most common Modbus settings, and should be tried first on existing systems with unknown parity settings.

  0 - 8 bit data, no parity, 1 stop bit (default, ASCII and RTU)
  1 - 7 bit data, odd parity, 1 stop bit (ASCII only)
  2 - 7 bit data, even parity, 1 stop bit (ASCII only)
  3 - 7 bit data, no parity, 2 stop bits (ASCII only)
  4 - 8 bit data, odd parity, 1 stop bit (ASCII and RTU)
  5 - 8 bit data, even parity, 1 stop bit (ASCII and RTU)
  6 - 8 bit data, no parity, 2 stop bits (ASCII and RTU)
  7 - 7 bit data, odd parity, 2 stop bits (ASCII only)
  8 - 7 bit data, even parity, 2 stop bits (ASCII only)
  9 - 8 bit data, odd parity, 2 stop bits (ASCII and RTU)
  10 - 8 bit data, even parity, 2 stop bits (ASCII and RTU)
  11 - 7 bit data, no parity, 1 stop bit (ASCII only)

Return Value of the initialization functions is always 1 (true), even if an unattainable baud rate has been selected. This is due to **`MMZ.LIB`**'s use of the serial drivers in **`Z0232.LIB`** and **`Z1232.LIB`**, whose initialization functions always return 1. Thus, checking the return code will only be useful in order to take advantage of possible future enhancements to the Modbus master libraries.

Step 4: Call Modbus Master Command Functions

Now that the Modbus master's serial port is initialized and ready, you are in control of the Modbus network. To read or write any Modbus register on any slave, just use one of the Modbus command functions such as **`mmInput`** or **`mmForceCoils`**. One such function exists for each of the Modbus commands supported in **`MM.LIB`**. Lists of supported and unsupported Modbus commands appear at the end of this chapter.

Each Modbus command function should be called within a loop or in a **`waitfor`** costatement, since each command contains an internal costatement which controls the process of sending the command and parsing the slave's reply, if any. When the network transaction has completed, the function will return a value indicating success or failure (and if so, why) as well as any valid data in the slave's reply.

Advanced Modbus Master Procedure

While **`MMZ.LIB`** will work fine for most Z-World applications, occasionally you'll need to use a port other than Z0 or Z1 as the Modbus master serial port. In order to facilitate this, **`MM.LIB`** interfaces to the Modbus serial port through a very simple pair of functions, **`mmRecv`** and **`mmSend`**.

Any implementation of these functions which matches the description in the Modbus Serial Interface section later in this chapter will allow **`MM.LIB`** to control the Modbus network to which these functions interface.

It should be noted that the Modbus command functions in **`MM.LIB`** assume that the serial port has already been initialized prior to the first attempt to execute a Modbus command. In addition to initializing the Modbus serial port, you must also call the appropriate one (and only one) of the Modbus master initialization functions, **`mmaInit`** or **`mmrInit`**:

- **`void mmaInit(void)`**

  `mmaInit` initializes the Modbus ASCII protocol by importing the appropriate version of the **`mmExec`** function and the **`mmLRC`** function from **`MM.LIB`**. It does not, in fact, contain any executable code.

- **`void mmrInit(unsigned long Baud)`**

  `mmrInit` initializes the Modbus RTU protocol by importing the appropriate version of the **`mmExec`** function and the **`mmCRC`** function from **`MM.LIB`**.

`Baud` is the serial communication rate (bps) used to set the Modbus RTU inter-packet gap (End Reply Time Out) value, unless overridden by a user-defined value.

Also note that the following line must appear near the top of the application in order for it to use **`MM.LIB`**:

```
        #use "MM.LIB"
```

Modbus Master Timeouts

The default Modbus master ASCII timeouts are standard at 1 second for both Begin Reply Time Out (**`MM_BRTO`**) and End Reply Time Out (**`MM_ERTO`**). However, the default Modbus master RTU timeouts are a combination of arbitrary at 100 mS for **`MM_BRTO`** and standard at 3.5 character times for **`MM_ERTO`** (calculated based on the **`Baud`** value passed to the **`mmrInit`** function, assuming 11 bit serial character frames and rounded up to the next whole mS). Note that the default timeouts can be overridden by defining **`MM_BRTO`** and/or **`MM_ERTO`** millisecond values at the beginning of the application, eg:

```
        #define MM_BRTO 1000      // Modbus ASCII standard
        #define MM_ERTO 1000      // Modbus ASCII standard
```

or,

```
        #define MM_BRTO 100 // Modbus RTU default
        #define MM_ERTO 100 // Modbus RTU override
```

Note that these timeout values apply directly only to the Modbus master and may be adjusted to suit a particular Modbus network's conditions. Each Modbus slave has its own timeout values, which may differ from the master and from other slaves.

Modbus Registers

In C, actions and manipulation of data are performed by functions, while data are stored in variables. This organization is so logical as to be intuitive to C programmers. This organization dominates most programming languages.

PLCs, however, do not operate on this principle. In PLCs, actions are performed by manipulating coils and registers, which also serve as storage for data. This uniformity is initially counter-intuitive to C programmers, but actually make for a simple interface.

Modbus simplifies this even further by supporting only two data types. Coils (referring to the coils in mechanical relays) store true/false information, a "bit" in common computer technology. Registers store 16-bit unsigned numbers.

The Modbus protocol provides for five classes of objects that can be manipulated. Since each class is assigned a unique, nonoverlapping address range, these objects are often referred to by their address space.

0X References (Discrete Outputs)

These bits are readable and writable. Some are used to control the PLC outputs , some are used to store internal bits, and others perform special PLC operations.

1X References (Discrete Inputs)

These bits are read only. They are used mainly for the digital inputs and to check the PLC status.

3X References (Input Registers)

These registers are read only. They are used mainly for multi-bit inputs such as analog/digital and pulse measurement readings.

4X References (Holding Registers)

These registers are readable and writable. They are used primary to hold data and for multi-bit outputs (such as digital/analog).

6X References (Extended Memory)

These registers are not supported by Dynamic C's Modbus Slave Driver nor are they supported by most Modbus-compatible devices. In fact, even Modicon's use of 6X registers is so nonstandard as to make a general-purpose driver difficult to implement.

Modbus Master Command Functions

- **`int mmOutRd(unsigned Addr, unsigned Coil,`**
  `        unsigned Count, void *Coils)`

  `mmOutRd` reads one or more consecutive 0X output coils. Broadcast is not supported.

  `Addr` is the address (ID) of the Modbus slave whose coils are to be read. Valid addresses are 1 through 255, inclusive.

  `Coil` is the number of the first coil to be read. Valid coil numbers are 0 through 65534, inclusive. Since Modbus numbers the coils from 1 through 65535, **`Coil`** should be 0 to read coil 1.

  `Count` is the number of consecutive coils to be read. Limitations in the size of a Modbus packet reduce the number of coils that can be read at one command to just under 2048.

`Coils` is a pointer to the memory where coil results are stored, often a character array. Coil states are packed eight to a byte with lowest coil number represented in the LSbit of the first byte. The minimum number of bytes required is equal to `Count` coils divided by eight rounded up to the next whole number. Unused bits in the last required byte are zero-filled. If its address is cast to `void *`, an unsigned integer can store up to 16 coils and an unsigned long integer can store up to 32 coils, with the lowest numbered coil represented in the LSbit of each type.

Return Value is `MM_BUSY (0)` until the command is completed. If successful, returns `MM_OK (-1)` and stores `Coils` data in the memory pointed to by `Coils`. Otherwise, an error code is returned to indicate the reason for failure. See the Modbus Command Return Values section later in this chapter for more information.

- **`int mmIn(unsigned Addr, unsigned Coil,`**
  `unsigned Count, void *Coils)`

`mmIn` reads one or more consecutive 1X input coils. Broadcast is not supported.

`Addr` is the address (ID) of the Modbus slave whose coils are to be read. Valid addresses are 1 through 255, inclusive.

`Coil` is the number of the first coil to be read. Valid coil numbers are 0 through 65534, inclusive. Since Modbus numbers the coils from 1 through 65535, `Coil` should be 0 to read coil 1.

`Count` is the number of consecutive coils to be read. Limitations in the size of a Modbus packet reduce the number of coils that can be read at one command to just under 2048.

`Coils` is a pointer to the memory where coil results are stored, often a character array. Coil states are packed eight to a byte with lowest coil number represented in the LSbit of the first byte. The minimum number of bytes required is equal to `Count` coils divided by eight rounded up to the next whole number. Unused bits in the last required byte are zero-filled. If its address is cast to `void *`, an unsigned integer can store up to 16 coils and an unsigned long integer can store up to 32 coils, with the lowest numbered coil represented in the LSbit of each type.

Return Value is `MM_BUSY (0)` until the command is completed. If successful, returns `MM_OK (-1)` and stores coils data in the memory pointed to by `Coils`. Otherwise, an error code is returned to indicate the reason for failure. See the ***Modbus Command Return Values*** section later in this chapter for more information.

- **`int mmInput(unsigned Addr, unsigned Reg,`**
  `unsigned Count, void *Regs)`

  `mmInput` reads one or more consecutive 3X input registers. Broadcast is not supported.

  `Addr` is the address (ID) of the Modbus slave whose registers are to be read. Valid addresses are 1 through 255, inclusive.

  `Reg` is the number of the first register to be read. Valid register numbers are 0 through 65534, inclusive. Since Modbus numbers the registers from 1 through 65535, **`Reg`** should be 0 to read register 1.

  `Count` is the number of consecutive registers to be read. Limitations in the size of a Modbus packet reduce the number of registers that can be read at one command to just under 128.

  `Regs` is a pointer to the memory where register results are stored, often an array of unsigned integers. Registers are 16-bit unsigned values, so the minimum number of bytes required is equal to **`Count`** times two. If its address is cast to **`void *`**, an unsigned integer can store one register and an unsigned long integer can store up to two consecutive registers. Similarly, a Dynamic C float can be mapped to store two consecutive registers which use the IEEE-754 32-bit floating point representation, which fortunately is a common (but not mandatory) format in Modbus software that supports floats.

  Return Value is **`MM_BUSY (0)`** until the command is completed. If successful, returns **`MM_OK (-1)`** and stores registers data in the memory pointed to by **`Regs`**. Otherwise, an error code is returned to indicate the reason for failure. See the ***Modbus Command Return Values*** section later in this chapter for more information.

- **`int mmRead(unsigned Addr, unsigned Reg,`**
  `unsigned Count, void *Regs)`

  `mmRead` reads one or more consecutive 4X holding registers. Broadcast is not supported.

  `Addr` is the address (ID) of the Modbus slave whose registers are to be read. Valid addresses are 1 through 255, inclusive.

  `Reg` is the number of the first register to be read. Valid register numbers are 0 through 65534, inclusive. Since Modbus numbers the registers from 1 through 65535, **`Reg`** should be 0 to read register 1.

  `Count` is the number of consecutive registers to be read. Limitations in the size of a Modbus packet reduce the number of registers that can be read at one command to just under 128.

Regs is a pointer to the memory where register results are stored, often an array of unsigned integers. Registers are 16-bit unsigned values, so the minimum number of bytes required is equal to **Count** times two. If its address is cast to void *, an unsigned integer can store one register and an unsigned long integer can store up to two consecutive registers. Similarly, a Dynamic C float can be mapped to store two consecutive registers which use the IEEE-754 32-bit floating point representation, which fortunately is a common (but not mandatory) format in Modbus software which supports floats.

Return Value is **MM_BUSY** (0) until the command is completed. If successful, returns **MM_OK** (-1) and stores registers data in the memory pointed to by **Reg**. Otherwise, an error code is returned to indicate the reason for failure. See the Modbus Command Return Values section later in this chapter for more information.

- **int mmForceCoil(unsigned Addr, unsigned Coil, int State)**

  mmForceCoil forces a single 0X output coil to the on or off state. Broadcast is supported.

  Addr is the address (ID) of the Modbus slave whose coil is to be forced. Valid addresses are 0 (broadcast) and 1 through 255, inclusive.

  Coil is the number of the coil to be forced. Valid coil numbers are 0 through 65534, inclusive. Since Modbus numbers the coils from 1 through 65535, **Coil** should be 0 to force coil 1.

  State is 0 to force the specified coil off, non-zero to force the coil on.

  Return Value is **MM_BUSY** (0) until the command is completed. If successful, returns **MM_OK** (-1). Note that because there is no slave response to a broadcast command, it will always return **MM_OK** when completed. Otherwise, an error code is returned to indicate the reason for failure. See the Modbus Command Return Values section later in this chapter for more information.

- **int mmForceCoils(unsigned Addr, unsigned Coil, unsigned Count, void *Coils)**

  mmForceCoils forces one or more consecutive 0X output coils to the on or off state, individually. Broadcast is supported.

  Addr is the address (ID) of the Modbus slave whose coils are to be forced. Valid addresses are 0 (broadcast) and 1 through 255, inclusive.

  Coil is the number of the first coil to be forced. Valid coil numbers are 0 through 65534, inclusive. Since Modbus numbers the coils from 1 through 65535, **Coil** should be 0 to force coil 1.

`Count` is the number of consecutive coils to be forced. Limitations in the size of a Modbus packet reduce the number of coils that can be read at one command to just under 2048.

`Coils` is a pointer to the memory where the desired coil states reside, often a character array. Coil states are packed eight to a byte with lowest coil number represented in the LSbit of the first byte. The minimum number of bytes required is equal to `Count` coils divided by eight rounded up to the next whole number. Unused bits in the last required byte should be zero-filled. If its address is cast to void *, an unsigned integer can store up to 16 coils and an unsigned long integer can store up to 32 coils, with the lowest numbered coil represented in the LSbit of each type.

Return Value is `MM_BUSY` (0) until the command is completed. If successful, returns `MM_OK` (-1). Note that because there is no slave response to a broadcast command, it will always return `MM_OK` when completed. Otherwise, an error code is returned to indicate the reason for failure. See the Modbus Command Return Values section later in this chapter for more information.

- **`int mmPresetReg(unsigned Addr, unsigned Reg, unsigned Value)`**

  `mmPresetReg` forces a single 4X holding register to the specified value. Broadcast is supported.

  `Addr` is the address (ID) of the Modbus slave whose register is to be forced. Valid addresses are 0 (broadcast) and 1 through 255, inclusive.

  `Reg` is the number of the register to be forced. Valid register numbers are 0 through 65534, inclusive. Since Modbus numbers the registers from 1 through 65535, `Reg` should be 0 to force register 1.

  `Value` is any unsigned integer from 0 through 65535, inclusive.

  Return Value is `MM_BUSY (0)` until the command is completed. If successful, returns `MM_OK (-1)`. Note that because there is no slave response to a broadcast command, it will always return `MM_OK` when completed. Otherwise, an error code is returned to indicate the reason for failure. See the ***Modbus Command Return Values*** section later in this chapter for more information.

- **`int mmPresetRegs(unsigned Addr, unsigned Reg, unsigned Count, void *Regs)`**

  `mmPresetRegs` forces one or more consecutive 4X holding registers to specified values, individually. Broadcast is supported.

`Addr` is the address (ID) of the Modbus slave whose registers are to be forced. Valid addresses are 0 (broadcast) and 1 through 255, inclusive.

`Reg` is the number of the first register to be forced. Valid register numbers are 0 through 65534, inclusive. Since Modbus numbers the registers from 1 through 65535, **`Reg`** should be 0 to force register 1.

`Count` is the number of consecutive registers to be forced. Limitations in the size of a Modbus packet reduce the number of registers that can be forced at one command to just under 128.

`Regs` is a pointer to the memory where the desired register values reside, often an array of unsigned integers. Registers are 16-bit unsigned values, so the minimum number of bytes required is equal to **`Count`** times two. If its address is cast to **`void *`**, an unsigned integer can store one register and an unsigned long integer can store up to two consecutive registers. Similarly, a Dynamic C float can be mapped to store two consecutive registers which use the IEEE-754 32-bit floating point representation, which fortunately is a common (but not mandatory) format in Modbus software that supports floats.

Return Value is **`MM_BUSY (0)`** until the command is completed. If successful, returns **`MM_OK (-1)`**. Note that because there is no slave response to a broadcast command, it will always return **`MM_OK`** when completed. Otherwise, an error code is returned to indicate the reason for failure. See the ***Modbus Command Return Values*** section later in this chapter for more information.

- **`int mmRegRdWr(unsigned Addr, unsigned RdReg,`**
```
        unsigned RdCount, void *RdRegs,
        unsigned WrReg, unsigned WrCount,
        void *WrRegs)
```

`mmRegRdWr` forces one or more consecutive 4X holding registers to specified values, individually, then reads one or more consecutive 4X holding registers. Broadcast is not supported.

`Addr` is the address (ID) of the Modbus slave whose registers are to be read and then forced. Valid addresses are 0 (broadcast) and 1 through 255, inclusive.

`RdReg` is the number of the first register to be read. Valid register numbers are 0 through 65534, inclusive. Since Modbus numbers the registers from 1 through 65535, **`Reg`** should be 0 to read register 1.

`RdCount` is the number of consecutive registers to be read. Limitations in the size of a Modbus packet reduce the total number of registers that can be forced and read at one command to just under 128.

`RdRegs` is a pointer to the memory where register results are stored, often an array of unsigned integers. Registers are 16-bit unsigned values, so the minimum number of bytes required is equal to `RdCount` times two. If its address is cast to `void *`, an unsigned integer can store one register and an unsigned long integer can store up to two consecutive registers. Similarly, a Dynamic C float can be mapped to store two consecutive registers which use the IEEE-754 32-bit floating point representation, which fortunately is a common (but not mandatory) format in Modbus software that supports floats.

`WrReg` is the number of the first register to be forced. Valid register numbers are 0 through 65534, inclusive. Since Modbus numbers the registers from 1 through 65535, `WrReg` should be 0 to force register 1.

`WrCount` is the number of consecutive registers to be forced. Limitations in the size of a Modbus packet reduce the total number of registers that can be forced and read at one command to just under 128.

`WrRegs` is a pointer to the memory where the desired register values reside, often an array of unsigned integers. Registers are 16-bit unsigned values, so the minimum number of bytes required is equal to `WrCount` times two. If its address is cast to `void *`, an unsigned integer can store one register and an unsigned long integer can store up to two consecutive registers. Similarly, a Dynamic C float can be mapped to store two consecutive registers which use the IEEE-754 32-bit floating point representation, which fortunately is a common (but not mandatory) format in Modbus software that supports floats.

Return Value is `MM_BUSY (0)` until the command is completed. If successful, returns `MM_OK (-1)` and stores registers data in the memory pointed to by `RdRegs`. Otherwise, an error code is returned to indicate the reason for failure. See the ***Modbus Command Return Values*** section later in this chapter for more information.

- **`int mmRegMask(unsigned Addr, unsigned Reg, unsigned And, unsigned Or)`**

`mmRegMask`, in a single atomic operation, masks a single 4X holding register according to the following formula: `4X[Reg] = (4X[Reg] & And) | (Or & ~And)`. This formula allows any combination of bits in the register to be set, cleared or left unchanged without the chance of the register value being solved (changed) by the slave in between a read command and a subsequent preset command. Broadcast is supported.

`Addr` is the address (ID) of the Modbus slave whose register is to be masked. Valid addresses are 0 (broadcast) and 1 through 255, inclusive.

Reg is the number of the register to be masked. Valid register numbers are 0 through 65534, inclusive. Since Modbus numbers the registers from 1 through 65535, **Reg** should be 0 to mask register 1.

And is any unsigned integer from 0 through 65535, inclusive. If **And** is zero, the result is simply **4X[Reg] = Or**.

Or is any unsigned integer from 0 through 65535, inclusive. If **Or** is zero, the result is **4X[Reg] = 4X[Reg] & And**.

Return Value is **MM_BUSY (0)** until the command is completed. If successful, returns **MM_OK (-1)**. Note that because there is no slave response to a broadcast command, it will always return **MM_OK** when completed. Otherwise, an error code is returned to indicate the reason for failure. See the ***Modbus Command Return Values*** section later in this chapter for more information.

- **int mmFetchCommCnt (unsigned Addr, void *Count, void *Status)**

  mmFetchCommCnt reads the current value of the communication event counter and the status. The communications event count is commonly read before and after a critical broadcast command has been executed, to check each slave's response. The count can also be helpful when debugging Modbus installations. Broadcast is not supported.

  Addr is the address (ID) of the Modbus slave whose communication events count is to be read. Valid addresses are 1 through 255, inclusive.

  Count is a pointer to the memory where the communications event count is stored, often an unsigned integer. Each Modbus slave is supposed to maintain a communication event counter which starts at zero and is incremented once for each successfully executed Modbus command.

  Status is a pointer to the memory where the status word is stored, often an unsigned integer. The status word has all bits set (0xFFFF) if the slave is still busy processing a previous command, or all bits clear (0x0000) if the slave is not busy.

  Return Value is **MM_BUSY (0)** until the command is completed. If successful, returns **MM_OK (-1)** and stores count and status information in the memory pointed to by Count and Status, respectively. Otherwise, an error code is returned to indicate the reason for failure. See the ***Modbus Command Return Values*** section later in this chapter for more information.

- **`int mmRdExcStat ( unsigned Addr, void *Coils )`**

  `mmRdExcStat` reads the eight exception status coils. The assignment and meaning of each coil depends on the type of slave device and may be either predefined or programmable. Broadcast is not supported.

  `Addr` is the address (ID) of the Modbus slave whose coils are to be read. Valid addresses are 1 through 255, inclusive.

  `Coils` is a pointer to the memory where coil results are stored, often a single character. Coil states are packed eight to a byte with lowest coil number represented in the LSbit of the byte.

  Return Value is **`MM_BUSY (0)`** until the command is completed. If successful, returns **`MM_OK (-1)`** and stores coils data in the memory pointed to by **`Coils`**. Otherwise, an error code is returned to indicate the reason for failure. See the ***Modbus Command Return Values*** section later in this chapter for more information.

Modbus Master Serial Interface

If the default serial driver supplied in **`MMZ.LIB`** is not suited to your application, it is fairly straightforward to write your own serial driver. **`MM.LIB`** uses just two functions to interface to the Modbus network. The **`mmRecv`** function reads a byte from the Modbus network and the **`mmSend`** function sends a block of bytes to the Modbus network. Write your own versions of these functions according to the guidelines below to enable **`MM.LIB`** to talk to your Modbus network.

- **`int mmRecv(void)`**

  `mmRecv` attempts to read a single byte from the Modbus network. If a byte is read successfully, **`mmRecv`** should return the received value in the range of 0 through 255, inclusive. If no byte is available then -1 should be returned.

- **`int mmSend(unsigned char *Cmd, unsigned Len)`**

  `mmSend` transmits a block of **`Len`** bytes from the **`Cmd`** character array to the Modbus network. Rather than busy waiting while the processor is transmitting the specified bytes, **`mmSend`** is called repeatedly by the Modbus Master library command functions until the entire packet is transmitted. While transmitting, **`mmSend`** should return false (zero). When transmission is complete, **`mmSend`** should return true (non-zero).

Prior to the actual attempt to transmit data, the Modbus Master library command functions call **`mmSend`** with a **`NULL`** value for the **`Cmd`** parameter. This is a signal to **`mmSend`** that a new Modbus command transmission is about to take place, and any chores required to set up for the new command (such as clearing out remaining Rx and Tx data from a previous command that is being canceled) should be done at this time. At the very minimum, **`mmSend`** should ignore calls when **`Cmd`** is **`NULL`**. The Modbus Master library command functions ignore **`mmSend`**'s return value when **`Cmd`** is **`NULL`**.

Modbus Master Supported Commands

The following commands are supported in the Modbus Master library:

0x01 : Read Coil Status
0x02 : Read Input Status
0x03 : Read Holding Registers
0x04 : Read Input Registers
0x05 : Force Single Coil
0x06 : Preset Single Register
0x07 : Read Exception Status
0x0B : Fetch Communication Event Counter
0x0F : Force Multiple Coils
0x10 : Preset Multiple Registers
0x16 : Mask Write 4X Register
0x17 : Read/Write 4X Registers

Modbus Master Unsupported Commands

The following Modbus commands are ***not*** supported in Z-World's Modbus master libraries. Please note that this is not an exhaustive list, as many vendors have added their own PLC-specific commands to the Modbus protocol.

0x08 : Diagnostics
0x09 : Program 484
0x0A : Poll 484
0x0C : Fetch Communication Event Log
0x0D : Program Controller
0x0E : Poll Controller
0x11 : Report Slave ID
0x12 : Program 884/M84
0x13 : Reset Communication Link
0x14 : Read General Reference
0x15 : Write General Reference
0x18 : Read FIFO Queue

Modbus Master Command Function Return Values

Below are listed the possible return values from the Modbus master command functions in MM.LIB. Each is followed by a brief explanation.

It is worth noting that zero indicates an unfinished command, negative one indicates success, all other negative values indicate a failure in the master, and positive values generally indicate failure in the slave or network. This scheme was chosen to expand on the Modbus standard of using only positive unsigned byte values for slave exception codes.

`#define MM_NOBROAD (-9) // Broadcast Not Supported`

`MM_NOBROAD` indicates that the Modbus master has attempted to broadcast a command that is not supported in broadcast mode. An example of this is an attempt to read register contents simultaneously from all slaves, which otherwise would be ignored by all properly configured slaves and eventually result in a less informative **`MM_TIMEOUT`** error. Note that this error is returned immediately by the Modbus master command, and no traffic is generated on the Modbus network.

`#define MM_TIMEOUT (-8) // Response Timeout`

`MM_TIMEOUT` indicates that the Modbus command appears to have been transmitted successfully, but no reply has been received from the slave within an acceptable period of time. See the Modbus Master Timeouts section earlier in this chapter for information about and customization of the acceptable time period macros MM_BRTO and MM_ERTO. This error can indicate one (or more) of several conditions:

- The slave doesn't exist.
- The slave is currently non-operational.
- The command was sent, but serial garbage prevented the slave from validating the command.
- The command was received successfully by the slave, which has taken too long to process the response.

This last condition is particularly troublesome because the slave's late response will often interrupt subsequent commands from the master. If you suspect such a situation, changing timeouts to a high value (2000 mS or more) will often reveal the problem. If this proves to be the case, you can either leave the high timeout values (which will allow the slow slave to dominate and ultimately cripple network bandwidth) or you can have the slave repaired or replaced.

`#define MM_GARBAGE (-7) // Garbage In Response`

`MM_GARBAGE` indicates the received packet contained illegal data. This usually results from serial noise or data collisions.

`#define MM_TOOLONG (-6) // Response Exceeds Buffer Length`

`MM_TOOLONG` indicates that the slave's response does not fit in the Modbus master's reply buffer. In practice, a properly functioning slave should catch this error and return the `MS_BADRESP` / `MS_DEVFAIL` error code, so the most likely causes are serial noise, data collisions or a malfunctioning slave.

`#define MM_BADXRC (-5) // Bad CRC/LRC`

`MM_BADXRC` indicates that an otherwise well-formed packet was received, but the received and computed values for longitudinal redundancy check (LRC for Modbus ASCII) or cyclic redundancy check (CRC for Modbus RTU) do not match. Usually, this is the result of errors in a small number of serial bits, which is quite rare. Persistent MM_BADXRC errors are often the result of incorrectly coded Modbus slaves. Occasional errors are usually the result of faulty hardware or wiring. Typical serial noise is more likely to produce `MM_GARBAGE` or `MM_TIMEOUT` errors.

`#define MM_BADID (-4) // Unexpected Slave ID in Response`

`MM_BADID` indicates that the response packet is well-formed, but not from the expected slave device. This unlikely error can occur if the Modbus master sequentially issues the same command to a number of slaves, but one slave's response is delayed and results in a `MM_TIMEOUT` error. When the tardy response is finally received (after the command is issued to another slave, and assuming no data collisions) it is not from the expected slave device.

`#define MM_BADCODE (-3) // Unexpected Response Code`

`MM_BADCODE` indicates that the response packet is well-formed, but not a response to the expected command. This unlikely error can occur if a Modbus master command resulting in an `MM_TIMEOUT` error is followed by a different command to the same slave. When the tardy response to the first command is finally received (after the second command is issued, and assuming no data collisions) it is not a response to the expected (second) command.

`#define MM_RESCODE (-2) // Reserved Exception Code (zero)`

`MM_RESCODE` indicates that a well-formed exception response packet has been received, but its exception byte is zero (ie: **`MM_BUSY`**, a reserved return value). Note that the Modbus standard uses only positive unsigned byte values for slave exception codes.

`#define MM_OK (-1) // Success`

`MM_OK` indicates that the Modbus master command has completed successfully.

`#define MM_BUSY 0 // Master Still Processing`

`MM_BUSY` indicates that the command is still being processed. Since Modbus commands generally perform serial communications which do not complete instantly, each takes a fair amount of time. Rather than halting execution of the entire application while the Modbus command function is waiting on serial communications, each command will return a value of **`MM_BUSY`** until the command is finished. The function should be called periodically (as often as the programmer is able or wishes) until a value other than **`MM_BUSY`** is returned. The particular return value of zero was chosen because it is logically distinct from non-zero values (false vs. true), so it works well with C control structures and fits nicely with a costatement's waitfor requirements.

`#define MS_BADFUNC 0x01 // Illegal Function`

`MS_BADFUNC` is returned by the slave to indicate that while a valid packet did arrive, the opcode of the packet was not recognized by that slave. In general, this means that the slave does not support the specified command. This failure is quite common, since most devices only implement a subset of the Modbus protocol.

`#define MS_BADADDR 0x02 // Illegal Data Address`

`MS_BADADDR` is returned by the slave to indicate that an attempt was made to read or write a coil or register which is unsupported by that slave. While coil and register spaces span 16-bits (64K coils or 64K registers), slaves typically only support some limited subset.

This error often confuses customers because Modbus slaves will fail a whole packet if any portion of it is illegal. Consider Modbus slave 0x23, which implements only registers 40001 through 40008, inclusive. Using the following command will result in an **`MS_BADADDR`** error:

```
unsigned MyRegs[9];
int Err;
while(!(Err = mmRead(0x23, 0, 8, MyRegs)));
```

The attempt to read unsupported register 40009 generates an **MS_BADADDR** error despite the fact that registers 40001 through 40008 are supported. No data are returned because of the single breach.

#define MS_BADDATA 0x03 // Illegal Data Value

MS_BADDATA is returned by the slave to indicate that an attempt was made to force a coil or register with an illegal value. This error is rare and never encountered for coils and registers used for general purpose output. It is generally a response to an attempt to write an illegal value into a specialized register, for example, writing 13 to a slave's register which represents the current month.

#define MS_BADRESP 0x04 // Illegal Response Length (unrecoverable error)

#define MS_DEVFAIL 0x04 // Slave Device Failure (unrecoverable error)

MS_BADRESP and **MS_DEVFAIL** are synonymous errors indicating that the slave has had an unrecoverable error while attempting to perform the command. A common cause of **MS_BADRESP** / **MS_DEVFAIL** is a request for more information than can fit in a single Modbus packet. Different sources of Modbus literature refer to this error code using different terminology, so both names are defined in **MM.LIB** for the convenience of the user.

#define MS_ACK 0x05 // Acknowledge (long duration program)

MS_ACK is returned by the slave to indicate that the command has been accepted, but a long processing time is required. This response is returned to prevent a timeout error from occurring in the Modbus master. Currently, no command supported in the Modbus master library should result in this response from a slave.

#define MS_DEVBUSY 0x06 // Slave Device Busy (long duration program)

MS_DEVBUSY is returned by the slave to indicate that the command can not be accepted, because it is still processing a previous command. Currently, no command supported in the Modbus master library should result in this response from a slave.

#define MS_NACK 0x07 // Negative Acknowledge (reject program)

MS_NACK is returned by the slave to indicate that the command specifies a program function that the slave can not perform. Currently, no command supported in the Modbus master library should result in this response from a slave.

`#define MS_MEMPERR 0x08 // Memory Parity Error (read extended memory)`

`MS_MEMPERR` is returned by the slave to indicate that a parity error was detected while attempting to read extended memory (6X references). If this error is persistent the slave device may need service. Currently, no command supported in the Modbus master library should result in this response from a slave.

`#define MS_NOGPATH 0x0A // Gateway Path Unavailable (Modbus Plus)`

`MS_NOGPATH` has specialized use in conjunction with Modbus Plus network gateways to indicate that the gateway was unable to allocate the PATH required to process the request. It usually means that the gateway is misconfigured. Currently, the Modbus master library does not support the Modbus Plus protocol.

`#define MS_NOGRESP 0x0B // Gateway Target Device Failed to Respond (Modbus Plus)`

`MS_NOGRESP` has specialized use in conjunction with Modbus Plus network gateways to indicate that no response was obtained from the target device. It usually means that the device is not present on the network. Currently, the Modbus master library does not support the Modbus Plus protocol.

For more information on the Modbus protocol, check the ***Modicon Modbus Protocol Reference Guide*** (Modicon Document PI-MBUS-300). This can be found on the World Wide Web at the Modicon Web site at (http://www.modicon.com).

CHAPTER 7: GRAPHICS ENGINE SUPPORT LIBRARY

The **`GESUPRT.LIB`** Graphics Engine support library described in this chapter can be **`#use`**d by applications for any of Z-World's Z180-based controllers. It may also be helpful as a reference when integrating the Graphics Engine into other types of systems.

GESUPRT.LIB

These interface functions support the use of the **op7100ge.c** Graphics Engine sample program (found in the **SAMPLES\OP71XX** sub-folder of the main Dynamic C 32 installation folder) as an intelligent, on-the-fly customizable operator interface for any Z-World Z180-based controller application.

GESUPRT.LIB supports anywhere from one up to four Graphics Engine devices connected to a controller whose application defines one or more of the following four macros in any combination:

```
#define GE_USE_PORTZ0    // G.E. on port Z0
#define GE_USE_PORTZ1    // G.E. on port Z1
#define GE_USE_PORTA     // G.E. on port SCC A
#define GE_USE_PORTB     // G.E. on port SCC B
```

One or more of the following four macros will be available to the application after the respective **#define GE_USE_PORTx** macro definition has been made. Use the appropriate following macros for the Port parameter in all **GESUPRT.LIB** user-callable functions:

```
GE_PORTZ0
GE_PORTZ1
GE_PORTA
GE_PORTB
```

GESUPRT.LIB has both blocking and non-blocking versions of all user-callable functions. Blocking functions do not return until either completed or timed out; they return 1 on success and a negative error code on failure. Non-blocking functions immediately return 0 if the Graphics Engine's response is still pending; otherwise they return 1 on success and a negative error code on failure. The list of Graphics Engine error codes follows:

-1	CRC fail
-2	record less than 5 bytes
-3	command not recognized
-4	command not implemented yet
-5	option not recognized
-6	real time clock has failed or is not installed
-7	command queue full
-8	command not allowed in a macro
-9	error adding a macro to the list
-10	END_MACRO command when not building a macro
-11	a command target (add, delete) does not exist
-12	memory allocation error
-13	parameter out of range, or object not found
-14	command other than STOP_MACRO if a macro is running
-15	time out error (no response received from G.E. device)

- **`int GEInit(int Port, long BaudRate)`**

  This blocking function always returns 1 when complete. Sets up the serial port and buffers. It may be necessary to delay briefly after this function is called in order to permit the Graphics Engine device to power up and be ready to accept commands. This function must be called once before any other Graphics Engine commands are issued.

  `Port` is one of **`GE_PORTZ0`**, **`GE_PORTZ1`**, **`GE_PORTA`**, or **`GE_PORTB`**.

  `BaudRate` is the communication bps rate to operate at (eg: 9600, 19200, etc.). Maximum for the OP7100 is 57600.

- **`int GEHardResetWF(int Port)`**

  This non-blocking version of the **`GEHardReset`** function immediately returns 0 while the Graphics Engine's response is pending, and so is suitable for use in costatements' **`waitfor`** statements. See the following function description for all other information.

- **`int GEHardReset(int Port)`**

  If successful, instructs the Graphics Engine to perform a hard reset and returns 1; otherwise, returns a negative error code from the list at the beginning of this chapter.

  `Port` is one of **`GE_PORTZ0`**, **`GE_PORTZ1`**, **`GE_PORTA`**, or **`GE_PORTB`**.

- **`int GESendStatusWF(int Port, int *status,`**
  `        int *errCode)`

  This non-blocking version of the **`GESendStatus`** function immediately returns 0 while the Graphics Engine's response is pending, and so is suitable for use in costatements' **`waitfor`** statements. See the following function description for all other information.

- **`int GESendStatus(int Port, int *status,`**
  `        int *errCode)`

  If successful, instructs the Graphics Engine to report its current status and returns 1; otherwise, returns a negative error code from the list at the beginning of this chapter.

  `Port` is one of **`GE_PORTZ0`**, **`GE_PORTZ1`**, **`GE_PORTA`**, or **`GE_PORTB`**.

  `status` is a pointer to where the Graphics Engine's current status will be stored.

  `errCode` is a pointer to where the Graphics Engine's current error code (if any) will be stored.

- **`int GESendLastPushWF(int Port, char iFunction, int *iBtnVal)`**

  This non-blocking version of the **`GESendLastPush`** function immediately returns 0 while the Graphics Engine's response is pending, and so is suitable for use in costatements' **`waitfor`** statements. See the following function description for all other information.

- **`int GESendLastPush(int Port, char iFunction, int *iBtnVal)`**

  If successful, instructs the Graphics Engine to report its last button or cell push and returns 1; otherwise, returns a negative error code from the list at the beginning of this chapter. Note that after the last-pushed cell number or button ID is sent, the Graphics Engine resets the value to zero in order to prevent continually sending the same information.

  `Port` is one of **`GE_PORTZ0`**, **`GE_PORTZ1`**, **`GE_PORTA`**, or **`GE_PORTB`**.

  `iFunction` is a flag which: if 1, selects last cell push information; if 2, selects last button push information.

  `iBtnVal` is a pointer to where the last-pushed cell number or button ID will be stored.

- **`int GESendStringWF(int Port, char *String)`**

  This non-blocking version of the **`GESendString`** function immediately returns 0 while the Graphics Engine's response is pending, and so is suitable for use in costatements' **`waitfor`** statements. See the following function description for all other information.

- **`int GESendString(int Port, char *String)`**

  If successful, instructs the Graphics Engine to report the last string entered via the Virtual Keyboard or set by a SetString command and returns 1; otherwise, returns a negative error code from the list at the beginning of this chapter.

  `Port` is one of **`GE_PORTZ0`**, **`GE_PORTZ1`**, **`GE_PORTA`**, or **`GE_PORTB`**.

  `String` is a pointer to where the Graphics Engine's reported string will be stored.

- **`int GESendLongWF(int Port, long *lValue)`**

  This non-blocking version of the **`GESendLong`** function immediately returns 0 while the Graphics Engine's response is pending, and so is suitable for use in costatements' **`waitfor`** statements. See the following function description for all other information.

- **`int GESendLong(int Port, long *lValue)`**

  If successful, instructs the Graphics Engine to report the last long integer value entered via the Virtual Keyboard or set by a SetLong command and returns 1; otherwise, returns a negative error code from the list at the beginning of this chapter.

  `Port` is one of **`GE_PORTZ0`**, **`GE_PORTZ1`**, **`GE_PORTA`**, or **`GE_PORTB`**.

  `lValue` is a pointer to where the Graphics Engine's reported long integer value will be stored.

- **`int GESendFloatWF(int Port, float *fValue)`**

  This non-blocking version of the **`GESendFloat`** function immediately returns 0 while the Graphics Engine's response is pending, and so is suitable for use in costatements' **`waitfor`** statements. See the following function description for all other information.

- **`int GESendFloat(int Port, float *fValue)`**

  If successful, instructs the Graphics Engine to report the last float value entered via the Virtual Keyboard or set by a SetFloat command and returns 1; otherwise, returns a negative error code from the list at the beginning of this chapter.

  `Port` is one of **`GE_PORTZ0`**, **`GE_PORTZ1`**, **`GE_PORTA`**, or **`GE_PORTB`**.

  `fValue` is a pointer to where the Graphics Engine's reported float value will be stored.

- **`int  GESendCharWF(int Port, char *cVal)`**

  This non-blocking version of the **`GESendChar`** function immediately returns 0 while the Graphics Engine's response is pending, and so is suitable for use in costatements' **`waitfor`** statements. See the following function description for all other information.

- **`int  GESendChar(int Port, char *cVal)`**

  If successful, instructs the Graphics Engine to report the last character entered via the Virtual Keyboard or set by a SetChar command and returns 1; otherwise, returns a negative error code from the list at the beginning of this chapter. Note that after the last-entered character is sent, the Graphics Engine resets the value to zero in order to prevent continually sending the same information.

  `Port` is one of **`GE_PORTZ0`**, **`GE_PORTZ1`**, **`GE_PORTA`**, or **`GE_PORTB`**.

  `cVal` is a pointer to where the Graphics Engine's reported character will be stored.

- **`int GESendTodWF(int Port, struct tm *time)`**

  This non-blocking version of the **`GESendTod`** function immediately returns 0 while the Graphics Engine's response is pending, and so is suitable for use in costatements' **`waitfor`** statements. See the following function description for all other information.

- **`int GESendTod(int Port, struct tm *time)`**

  If successful, instructs the Graphics Engine to report its current date and time of day and returns 1; otherwise returns a negative error code from the list at the beginning of this chapter.

  `Port` is one of **`GE_PORTZ0`**, **`GE_PORTZ1`**, **`GE_PORTA`**, or **`GE_PORTB`**.

  `time` is a pointer to where the Graphics Engine's reported time structure will be stored.

- **`int GEClearBufferWF(int Port)`**

  This non-blocking version of the **`GEClearBuffer`** function immediately returns 0 while the Graphics Engine's response is pending, and so is suitable for use in costatements' **`waitfor`** statements. See the following function description for all other information.

- **`int GEClearBuffer(int Port)`**

  If successful, instructs the Graphics Engine to clear the Virtual Keyboard's string buffer and returns 1; otherwise returns a negative error code from the list at the beginning of this chapter.

  `Port` is one of **`GE_PORTZ0`**, **`GE_PORTZ1`**, **`GE_PORTA`**, or **`GE_PORTB`**.

- **`int GEDeleteBitMapWF(int Port, int BitmapID)`**

  This non-blocking version of the **`GEDeleteBitMap`** function immediately returns 0 while the Graphics Engine's response is pending, and so is suitable for use in costatements' **`waitfor`** statements. See the following function description for all other information.

- **`int GEDeleteBitMap(int Port, int BitmapID)`**

  If successful, instructs the Graphics Engine to delete the specified user-defined stored bit map and returns 1; otherwise returns a negative error code from the list at the beginning of this chapter.

  `Port` is one of **`GE_PORTZ0`**, **`GE_PORTZ1`**, **`GE_PORTA`**, or **`GE_PORTB`**.

  `BitmapID` is the ID number of a user-defined stored bit map, and can't be 0 or a reserved bit map ID (251 through 255, inclusive).

- **`int GELoadBitMapWF(int Port, int BitmapID,`**
  `        int Width, int Height,`
  `        unsigned long BmData)`

  This non-blocking version of the **`GELoadBitMap`** function immediately returns 0 while the Graphics Engine's response is pending, and so is suitable for use in costatements' **`waitfor`** statements. See the following function description for all other information.

- **`int GELoadBitMap(int Port, int BitmapID,`**
  `        int Width, int Height,`
  `        unsigned long BmData)`

  If successful, instructs the Graphics Engine to load and store the specified user-defined bit map and returns 1; otherwise returns a negative error code from the list at the beginning of this chapter. Note that this function call may require multiple Graphics Engine communication packets to finish.

  `Port` is one of **`GE_PORTZ0`**, **`GE_PORTZ1`**, **`GE_PORTA`**, or **`GE_PORTB`**.

  `BitmapID` is the ID number associated with the user-defined bit map, and can't be 0 or a reserved bit map ID (251 through 255, inclusive).

  `Width` is the horizontal size (width) of the bit map in pixels.

  `Height` is the vertical size (height) of the bit map in pixels.

  `BmData` is the physical address of the user-defined bit map source.

- **`int GEDeleteFontWF(int Port, int FontID)`**

  This non-blocking version of the **`GEDeleteFont`** function immediately returns 0 while the Graphics Engine's response is pending, and so is suitable for use in costatements' **`waitfor`** statements. See the following function description for all other information.

- **`int GEDeleteFont(int Port, int FontID)`**

  If successful, instructs the Graphics Engine to delete the specified user-defined font and returns 1; otherwise returns a negative error code from the list at the beginning of this chapter.

  `Port` is one of **`GE_PORTZ0`**, **`GE_PORTZ1`**, **`GE_PORTA`**, or **`GE_PORTB`**.

  `FontID` is the ID number of a user-defined stored font, and can't be 0 or a default font ID (1 through 3, inclusive).

- **`int GELoadFontWF(int Port, int FontID, int Width, int Height, int SChar, int EChar, unsigned long FontData)`**

  This non-blocking version of the **`GELoadFont`** function immediately returns 0 while the Graphics Engine's response is pending, and so is suitable for use in costatements' **`waitfor`** statements. See the following function description for all other information.

- **`int GELoadFont(int Port, int FontID, int Width, int Height, int SChar, int EChar, unsigned long FontData)`**

  If successful, instructs the Graphics Engine to load and store the specified user-defined font and returns 1; otherwise returns a negative error code from the list at the beginning of this chapter. Note that this function call may require multiple Graphics Engine communication packets to finish.

  `Port` is one of **`GE_PORTZ0`**, **`GE_PORTZ1`**, **`GE_PORTA`**, or **`GE_PORTB`**.

  `FontID` is the ID number associated with the user-defined font, and can't be 0 or a default font ID (1 through 3, inclusive).

  `Width` is the horizontal size of each font character in pixels.

  `Height` is the vertical size of each font character in pixels.

  `SChar` is the index (often the ASCII code) of the first font character.

  `EChar` is the index (often the ASCII code) of the last font character.

  `FontData` is the physical address of the user-defined font source.

- **`int GEDisableCellsWF(int Port, int CellID)`**

  This non-blocking version of the **`GEDisableCells`** function immediately returns 0 while the Graphics Engine's response is pending, and so is suitable for use in costatements' **`waitfor`** statements. See the following function description for all other information.

- **`int GEDisableCells(int Port, int CellID)`**

  If successful, instructs the Graphics Engine to disable either one or all touch-screen cells and returns 1; otherwise returns a negative error code from the list at the beginning of this chapter.

  `Port` is one of **`GE_PORTZ0`**, **`GE_PORTZ1`**, **`GE_PORTA`**, or **`GE_PORTB`**.

  `CellID` is the ID number of a single cell, or 0 to disable all cells.

- **`int GEEnableCellsWF(int Port, int CellID)`**

  This non-blocking version of the **`GEEnableCells`** function immediately returns 0 while the Graphics Engine's response is pending, and so is suitable for use in costatements' **`waitfor`** statements. See the following function description for all other information.

- **`int GEEnableCells(int Port, int CellID)`**

  If successful, instructs the Graphics Engine to enable either one or all touch-screen cells and returns 1; otherwise returns a negative error code from the list at the beginning of this chapter.

  `Port` is one of **`GE_PORTZ0`**, **`GE_PORTZ1`**, **`GE_PORTA`**, or **`GE_PORTB`**.

  `CellID` is the ID number of a single cell, or 0 to enable all cells.

- **`int GESuperResetWF(int Port)`**

  This non-blocking version of the **`GESuperReset`** function immediately returns 0 while the Graphics Engine's response is pending, and so is suitable for use in costatements' **`waitfor`** statements. See the following function description for all other information.

- **`int GESuperReset(int Port)`**

  If successful, instructs the Graphics Engine to clear all user-defined stored bit maps and fonts by initializing their storage structures and returns 1; otherwise returns a negative error code from the list at the beginning of this chapter. Note that this function must be called once before user-defined bit maps and fonts can be stored.

  `Port` is one of **`GE_PORTZ0`**, **`GE_PORTZ1`**, **`GE_PORTA`**, or **`GE_PORTB`**.

- **`int GEBeginMacroWF(int Port, int MacroID)`**

  This non-blocking version of the **`GEBeginMacro`** function immediately returns 0 while the Graphics Engine's response is pending, and so is suitable for use in costatements' **`waitfor`** statements. See the following function description for all other information.

- **`int GEBeginMacro(int Port, int MacroID)`**

  If successful, instructs the Graphics Engine to store subsequent commands in a macro, up to the required EndMacro command, and returns 1; otherwise returns a negative error code from the list at the beginning of this chapter.

  `Port` is one of **`GE_PORTZ0`**, **`GE_PORTZ1`**, **`GE_PORTA`**, or **`GE_PORTB`**.

  `MacroID` is the ID number associated with the user-defined macro, and can't redefine a macro ID number that is already in use.

- **`int GEEndMacroWF(int Port, int MacroID)`**

  This non-blocking version of the **`GEEndMacro`** function immediately returns 0 while the Graphics Engine's response is pending, and so is suitable for use in costatements' **`waitfor`** statements. See the following function description for all other information.

- **`int GEEndMacro(int Port, int MacroID)`**

  If successful, instructs the Graphics Engine to stop storing commands in a macro and returns 1; otherwise returns a negative error code from the list at the beginning of this chapter.

  `Port` is one of **`GE_PORTZ0`**, **`GE_PORTZ1`**, **`GE_PORTA`**, or **`GE_PORTB`**.

  `MacroID` is the ID number associated with the user-defined macro that is under construction.

- **`int GEClearScreenWF(int Port)`**

  This non-blocking version of the **`GEClearScreen`** function immediately returns 0 while the Graphics Engine's response is pending, and so is suitable for use in costatements' **`waitfor`** statements. See the following function description for all other information.

- **`int GEClearScreen(int Port)`**

  If successful, instructs the Graphics Engine to clear its display screen and returns 1; otherwise returns a negative error code from the list at the beginning of this chapter.

  `Port` is one of **`GE_PORTZ0`**, **`GE_PORTZ1`**, **`GE_PORTA`**, or **`GE_PORTB`**.

- **`int GESetBrushTypeWF(int Port, int Btype)`**

  This non-blocking version of the **`GESetBrushType`** function immediately returns 0 while the Graphics Engine's response is pending, and so is suitable for use in costatements' **`waitfor`** statements. See the following function description for all other information.

- **`int GESetBrushType(int Port, int Btype)`**

  If successful, instructs the Graphics Engine to set the brush type used for subsequent draw commands and returns 1; otherwise returns a negative error code from the list at the beginning of this chapter.

  `Port` is one of **`GE_PORTZ0`**, **`GE_PORTZ1`**, **`GE_PORTA`**, or **`GE_PORTB`**.

  `Btype` is the brush type, where 1 = CLEAR (white), 2 = SET (black), and 3 = XOR (change).

- **`int GEPutPixelWF(int Port, int Xcoord, int Ycoord)`**

  This non-blocking version of the **`GEPutPixel`** function immediately returns 0 while the Graphics Engine's response is pending, and so is suitable for use in costatements' **`waitfor`** statements. See the following function description for all other information.

- **`int GEPutPixel(int Port, int Xcoord, int Ycoord)`**

  If successful, instructs the Graphics Engine to draw a pixel using the current brush and returns 1; otherwise returns a negative error code from the list at the beginning of this chapter.

  `Port` is one of **`GE_PORTZ0`**, **`GE_PORTZ1`**, **`GE_PORTA`**, or **`GE_PORTB`**.

  `Xcoord` is the pixel's horizontal coordinate.

  `Ycoord` is the pixel's vertical coordinate.

- **`int GEPutLineWF(int Port, int Xcoord, int Ycoord, int XXcoord, int YYcoord)`**

  This non-blocking version of the **`GEPutLine`** function immediately returns 0 while the Graphics Engine's response is pending, and so is suitable for use in costatements' **`waitfor`** statements. See the following function description for all other information.

- **`int GEPutLine(int Port, int Xcoord, int Ycoord, int XXcoord, int YYcoord)`**

  If successful, instructs the Graphics Engine to draw a line using the current brush and returns 1; otherwise returns a negative error code from the list at the beginning of this chapter.

  `Port` is one of **`GE_PORTZ0`**, **`GE_PORTZ1`**, **`GE_PORTA`**, or **`GE_PORTB`**.

  `Xcoord` is the horizontal coordinate of the line's start point pixel.

  `Ycoord` is the vertical coordinate of the line's start point pixel.

  `XXcoord` is the horizontal coordinate of the line's end point pixel.

  `YYcoord` is the vertical coordinate of the line's end point pixel.

- **`int GEPutCircleWF(int Port, int FillOnOff, int Xcoord, int Ycoord, int Radius)`**

  This non-blocking version of the **`GEPutCircle`** function immediately returns 0 while the Graphics Engine's response is pending, and so is suitable for use in costatements' **`waitfor`** statements. See the following function description for all other information.

- **`int GEPutCircle(int Port, int FillOnOff, int Xcoord, int Ycoord, int Radius)`**

  If successful, instructs the Graphics Engine to draw a circle using the current brush and returns 1; otherwise returns a negative error code from the list at the beginning of this chapter.

  `Port` is one of **`GE_PORTZ0`**, **`GE_PORTZ1`**, **`GE_PORTA`**, or **`GE_PORTB`**.

  `FillOnOff` is a flag which: if 1, draws the circle filled; if 0, draws the circle outline.

  `Xcoord` is the horizontal coordinate of the circle's center point pixel.

  `Ycoord` is the vertical coordinate of the circle's center point pixel.

  `Radius` is the circle's radius in pixels.

- **`int GEPutRectangleWF(int Port, int FillOnOff, int Xcoord, int Ycoord, int XXcoord, int YYcoord)`**

  This non-blocking version of the **`GEPutRectangle`** function immediately returns 0 while the Graphics Engine's response is pending, and so is suitable for use in costatements' **`waitfor`** statements. See the following function description for all other information.

- **`int GEPutRectangle(int Port, int FillOnOff, int Xcoord, int Ycoord, int XXcoord, int YYcoord)`**

  If successful, instructs the Graphics Engine to draw a rectangle using the current brush and returns 1; otherwise returns a negative error code from the list at the beginning of this chapter.

  `Port` is one of **`GE_PORTZ0`**, **`GE_PORTZ1`**, **`GE_PORTA`**, or **`GE_PORTB`**.

  `FillOnOff` is a flag which: if 1, draws the rectangle filled; if 0, draws the rectangle outline.

  `Xcoord` is the horizontal coordinate of the rectangle's upper left corner point pixel.

  `Ycoord` is the vertical coordinate of the rectangle's upper left corner point pixel.

  `XXcoord` is the horizontal coordinate of the rectangle's lower right corner point pixel.

  `YYcoord` is the vertical coordinate of the rectangle's lower right corner point pixel.

- **`int GEPutPolygonWF(int Port, int FillOnOff, int NumSides, int X1, int Y1, int X2, int Y2, int X3, int Y3, ...)`**

  This non-blocking version of the **`GEPutPolygon`** function immediately returns 0 while the Graphics Engine's response is pending, and so is suitable for use in costatements' **`waitfor`** statements. See the following function description for all other information.

- **`int GEPutPolygon(int Port, int FillOnOff, int NumSides, int X1, int Y1, int X2, int Y2, int X3, int Y3, ...)`**

  If successful, instructs the Graphics Engine to draw a polygon using the current brush and returns 1; otherwise returns a negative error code from the list at the beginning of this chapter. Note that the count **`n`** of (**`Xi`**,**`Yi`**) coordinate pairs must equal **`NumSides`**, and the last side is automatically drawn from (**`Xn`**,**`Yn`**) back to (**`X1`**,**`Y1`**).

  `Port` is one of **`GE_PORTZ0`**, **`GE_PORTZ1`**, **`GE_PORTA`**, or **`GE_PORTB`**.

  `FillOnOff` is a flag which: if 1, draws the polygon filled; if 0, draws the polygon outline.

  `NumSides` is the number of sides in the polygon; a minimum of 3, and equal to the number of supplied (**`Xi`**,**`Yi`**) coordinate pair parameters.

  `X1` is the horizontal coordinate of the 1st vertex. Note that all subsequent parameters **`X2, ..., Xn`** are the horizontal coordinates of additional vertices.

  `Y1` is the vertical coordinate of the 1st vertex. Note that all subsequent parameters **`Y2, ..., Yn`** are the vertical coordinates of additional vertices.

- **`int GEBlankRegionWF(int Port, int FillOnOff, int Xcoord, int Ycoord, int XXcoord, int YYcoord)`**

  This non-blocking version of the **`GEBlankRegion`** function immediately returns 0 while the Graphics Engine's response is pending, and so is suitable for use in costatements' **`waitfor`** statements. See the following function description for all other information.

- **`int GEBlankRegion(int Port, int FillOnOff, int Xcoord, int Ycoord, int XXcoord, int YYcoord)`**

  If successful, instructs the Graphics Engine to blank (clear or set) a region on the display screen and returns 1; otherwise returns a negative error code from the list at the beginning of this chapter.

Port is one of **GE_PORTZ0**, **GE_PORTZ1**, **GE_PORTA**, or **GE_PORTB**.

FillOnOff is a flag which: if 1, draws the region set (filled); if 0, draws the region clear (empty).

Xcoord is the horizontal coordinate of the region's upper left corner point pixel.

Ycoord is the vertical coordinate of the region's upper left corner point pixel.

XXcoord is the horizontal coordinate of the region's lower right corner point pixel.

YYcoord is the vertical coordinate of the region's lower right corner point pixel.

- **`int GEInvertRegionWF(int Port, int Xcoord, int Ycoord, int XXcoord, int YYcoord)`**

  This non-blocking version of the **GEInvertRegion** function immediately returns 0 while the Graphics Engine's response is pending, and so is suitable for use in costatements' **waitfor** statements. See the following function description for all other information.

- **`int GEInvertRegion(int Port, int Xcoord, int Ycoord, int XXcoord, int YYcoord)`**

  If successful, instructs the Graphics Engine to invert a region on the display screen and returns 1; otherwise returns a negative error code from the list at the beginning of this chapter.

  Port is one of **GE_PORTZ0**, **GE_PORTZ1**, **GE_PORTA**, or **GE_PORTB**.

  Xcoord is the horizontal coordinate of the region's upper left corner point pixel.

  Ycoord is the vertical coordinate of the region's upper left corner point pixel.

  XXcoord is the horizontal coordinate of the region's lower right corner point pixel.

  YYcoord is the vertical coordinate of the region's lower right corner point pixel.

- **`int GEStoreRegionWF(int Port, int RegionID, int Xcoord, int Ycoord, int XXcoord, int YYcoord)`**

  This non-blocking version of the **GEStoreRegion** function immediately returns 0 while the Graphics Engine's response is pending, and so is suitable for use in costatements' **waitfor** statements. See the following function description for all other information.

- **`int GEStoreRegion(int Port, int RegionID, int Xcoord, int Ycoord, int XXcoord, int YYcoord)`**

  If successful, instructs the Graphics Engine to store a region on the display screen and returns 1; otherwise returns a negative error code from the list at the beginning of this chapter.

  `Port` is one of **`GE_PORTZ0`**, **`GE_PORTZ1`**, **`GE_PORTA`**, or **`GE_PORTB`**.

  `RegionID` is the ID number associated with the stored region, a number from 1 through 255, inclusive.

  `Xcoord` is the horizontal coordinate of the region's upper left corner point pixel.

  `Ycoord` is the vertical coordinate of the region's upper left corner point pixel.

  `XXcoord` is the horizontal coordinate of the region's lower right corner point pixel.

  `YYcoord` is the vertical coordinate of the region's lower right corner point pixel.

- **`int GERestoreRegionWF(int Port, int RegionID, int Xcoord, int Ycoord)`**

  This non-blocking version of the **`GERestoreRegion`** function immediately returns 0 while the Graphics Engine's response is pending, and so is suitable for use in costatements' **`waitfor`** statements. See the following function description for all other information.

- **`int GERestoreRegion(int Port, int RegionID, int Xcoord, int Ycoord)`**

  If successful, instructs the Graphics Engine to restore a stored region on the display screen and returns 1; otherwise returns a negative error code from the list at the beginning of this chapter. Note that the region does not have to be restored to its original location; but if **`Xcoord`** and **`Ycoord`** each equal -1 the region is restored to the stored coordinates.

  `Port` is one of **`GE_PORTZ0`**, **`GE_PORTZ1`**, **`GE_PORTA`**, or **`GE_PORTB`**.

  `RegionID` is the ID number associated with the stored region, a number from 1 through 255, inclusive.

  `Xcoord` is the horizontal coordinate of the region's upper left corner point pixel.

  `Ycoord` is the vertical coordinate of the region's upper left corner point pixel.

- **`int GEPutBitMapWF(int Port, int BitmapID,`**
  `        int Xcoord, int Ycoord)`

  This non-blocking version of the **`GEPutBitMap`** function immediately returns 0 while the Graphics Engine's response is pending, and so is suitable for use in costatements' **`waitfor`** statements. See the following function description for all other information.

- **`int GEPutBitMap(int Port, int BitmapID,`**
  `        int Xcoord, int Ycoord)`

  If successful, instructs the Graphics Engine to display a stored bit map on the screen and returns 1; otherwise returns a negative error code from the list at the beginning of this chapter.

  `Port` is one of **`GE_PORTZ0`**, **`GE_PORTZ1`**, **`GE_PORTA`**, or **`GE_PORTB`**.

  `BitmapID` is the ID number of a stored bit map.

  `Xcoord` is the horizontal coordinate of the displayed bit map's upper left corner point pixel.

  `Ycoord` is the vertical coordinate of the displayed bit map's upper left corner point pixel.

- **`int GEScrollRegionWF(int Port, int Direction,`**
  `        int Distance, int Xcoord, int Ycoord,`
  `        int XXcoord, int YYcoord)`

  This non-blocking version of the **`GEScrollRegion`** function immediately returns 0 while the Graphics Engine's response is pending, and so is suitable for use in costatements' **`waitfor`** statements. See the following function description for all other information.

- **`int GEScrollRegion(int Port, int Direction,`**
  `        int Distance, int Xcoord, int Ycoord,`
  `        int XXcoord, int YYcoord)`

  If successful, instructs the Graphics Engine to scroll a region on the display screen and returns 1; otherwise returns a negative error code from the list at the beginning of this chapter.

  `Port` is one of **`GE_PORTZ0`**, **`GE_PORTZ1`**, **`GE_PORTA`**, or **`GE_PORTB`**.

  `Direction` is the scroll movement code, one of: 1 (up), 2 (down), 3 (right), or 4 (left).

  `Distance` is the number of pixels that the region is to scroll.

  `Xcoord` is the horizontal coordinate of the region's upper left corner point pixel.

  `Ycoord` is the vertical coordinate of the region's upper left corner point pixel.

`XXcoord` is the horizontal coordinate of the region's lower right corner point pixel.

`YYcoord` is the vertical coordinate of the region's lower right corner point pixel.

- **`int GESetFontWF(int Port, int FontID)`**

  This non-blocking version of the **`GESetFont`** function immediately returns 0 while the Graphics Engine's response is pending, and so is suitable for use in costatements' **`waitfor`** statements. See the following function description for all other information.

- **`int GESetFont(int Port, int FontID)`**

  If successful, instructs the Graphics Engine to set the font used for subsequent PutText commands and returns 1; otherwise returns a negative error code from the list at the beginning of this chapter.

  `Port` is one of **`GE_PORTZ0`**, **`GE_PORTZ1`**, **`GE_PORTA`**, or **`GE_PORTB`**.

  `FontID` is the ID number associated with the stored font, and can't be 0. Default font IDs are 1 (small - 6x8), 2 (medium - 12x16), and 3 (large - 17x35).

- **`int GEPutTextWF(int Port, char *string,`**
  `        int Xcoord, int Ycoord)`

  This non-blocking version of the **`GEPutText`** function immediately returns 0 while the Graphics Engine's response is pending, and so is suitable for use in costatements' **`waitfor`** statements. See the following function description for all other information.

- **`int GEPutText(int Port, char *string,`**
  `        int Xcoord, int Ycoord)`

  If successful, instructs the Graphics Engine to display text using the currently selected font and returns 1; otherwise returns a negative error code from the list at the beginning of this chapter.

  `Port` is one of **`GE_PORTZ0`**, **`GE_PORTZ1`**, **`GE_PORTA`**, or **`GE_PORTB`**.

  `string` is a pointer to the displayed zero-terminated character string.

  `Xcoord` is the horizontal coordinate of the displayed text's upper left corner point pixel.

  `Ycoord` is the vertical coordinate of the displayed text's upper left corner point pixel.

- **`int GESetTextDirWF(int Port, int Direction)`**

  This non-blocking version of the **`GESetTextDir`** function immediately returns 0 while the Graphics Engine's response is pending, and so is suitable for use in costatements' **`waitfor`** statements. See the following function description for all other information.

- **`int GESetTextDir(int Port, int Direction)`**

  If successful, instructs the Graphics Engine to set the text printing direction used for subsequent PutText commands and returns 1; otherwise returns a negative error code from the list at the beginning of this chapter.

  `Port` is one of **`GE_PORTZ0`**, **`GE_PORTZ1`**, **`GE_PORTA`**, or **`GE_PORTB`**.

  `Direction` is the text printing movement code, one of: 0 (left-to-right), 1 (right-to-left), 2 (bottom-to-top), or 3 (top-to-bottom).

- **`int GEBackLightWF(int Port, int OnOff)`**

  This non-blocking version of the **`GEBackLight`** function immediately returns 0 while the Graphics Engine's response is pending, and so is suitable for use in costatements' **`waitfor`** statements. See the following function description for all other information.

- **`int GEBackLight(int Port, int OnOff)`**

  If successful, instructs the Graphics Engine to switch its display screen backlight on or off and returns 1; otherwise returns a negative error code from the list at the beginning of this chapter.

  `Port` is one of **`GE_PORTZ0`**, **`GE_PORTZ1`**, **`GE_PORTA`**, or **`GE_PORTB`**.

  `OnOff` is a flag which: if 1, switches the backlight on; if 0, switches the backlight off.

- **`int GESetContrastWF(int Port, int Contrast)`**

  This non-blocking version of the **`GESetContrast`** function immediately returns 0 while the Graphics Engine's response is pending, and so is suitable for use in costatements' **`waitfor`** statements. See the following function description for all other information.

- **`int GESetContrast(int Port, int Contrast)`**

  If successful, instructs the Graphics Engine to set the contrast on its display screen and returns 1; otherwise returns a negative error code from the list at the beginning of this chapter.

  `Port` is one of **`GE_PORTZ0`**, **`GE_PORTZ1`**, **`GE_PORTA`**, or **`GE_PORTB`**.

  `Contrast` is a level value from 0 through 63, inclusive; 0 sets the highest contrast.

- **`int GEBeepWF(int Port, int Duration)`**

  This non-blocking version of the **`GEBeep`** function immediately returns 0 while the Graphics Engine's response is pending, and so is suitable for use in costatements' **`waitfor`** statements. See the following function description for all other information.

- **`int GEBeep(int Port, int Duration)`**

  If successful, instructs the Graphics Engine to switch on its buzzer for a time and returns 1; otherwise returns a negative error code from the list at the beginning of this chapter.

  `Port` is one of **`GE_PORTZ0`**, **`GE_PORTZ1`**, **`GE_PORTA`**, or **`GE_PORTB`**.

  `Duration` is the count in milliseconds of buzzer-on time; 0 sets the default time of 100 milliseconds.

- **`int GESetCellActiveWF(int Port, char cellID, char flags)`**

  This non-blocking version of the **`GESetCellActive`** function immediately returns 0 while the Graphics Engine's response is pending, and so is suitable for use in costatements' **`waitfor`** statements. See the following function description for all other information.

- **`int GESetCellActive(int Port, char cellID, char flags)`**

  If successful, instructs the Graphics Engine to set the active enable/ disable levels for a touch-screen cell and returns 1; otherwise returns a negative error code from the list at the beginning of this chapter.

  `Port` is one of **`GE_PORTZ0`**, **`GE_PORTZ1`**, **`GE_PORTA`**, or **`GE_PORTB`**.

  `CellID` is the ID number of the cell.

  `flags` is the bit-coded cell activity enable/disable value where:
  if all bits are reset (0), cell is inactive;
  if bit 0 set (1), cell-push sends unsolicited response to host;
  if bit 1 set, cell-push produces a beep.

- **`int GEDefineButtonWF(int Port, int ButtonID, char BtnFlags, char BtnTL, char BtnBR, char BmID, char *string)`**

  This non-blocking version of the **`GEDefineButton`** function immediately returns 0 while the Graphics Engine's response is pending, and so is suitable for use in costatements' **`waitfor`** statements. See the following function description for all other information.

- **`int GEDefineButton(int Port, int ButtonID, char BtnFlags, char BtnTL, char BtnBR, char BmID, char *string)`**

  If successful, instructs the Graphics Engine to store a user-defined button and returns 1; otherwise returns a negative error code from the list at the beginning of this chapter. Note that the button is defined but not displayed by this function.

  `Port` is one of **`GE_PORTZ0`**, **`GE_PORTZ1`**, **`GE_PORTA`**, or **`GE_PORTB`**.

  `ButtonID` is the ID number of the button being defined.

  `BtnFlags` is the bit-coded button properties value where:
  if bit 0 set (1), button-push sends unsolicited response to host;
  if bit 1 set, button-push produces a beep;
  if bit 2 set, button is drawn with a frame;
  if bit 3 set, button frame has rounded corners (requires bit 2 set);
  if bit 4 set, button-press draws inverted button;
  if bit 5 set, button is normally drawn inverted;
  if bit 6 set, button text follows in **`string`** (requires bit 7 reset);
  if bit 7 set, button bit map set to **`BmID`** (requires bit 6 reset).

  `BtnTL` is the ID number of the button's top left touch-screen cell.

  `BtnBR` is the ID number of the button's bottom right touch-screen cell.

  `BmID` is the ID number of a stored bit map (0 if **`BtnFlags`** bit 6 set).

  `string` is a pointer to the button's zero-terminated text label (**`NULL`** if **`BtnFlags`** bit 7 set).

- **`int GEDeleteButtonWF(int Port, int ButtonID)`**

  This non-blocking version of the **`GEDeleteButton`** function immediately returns 0 while the Graphics Engine's response is pending, and so is suitable for use in costatements' **`waitfor`** statements. See the following function description for all other information.

- **`int GEDeleteButton(int Port, int ButtonID)`**

  If successful, instructs the Graphics Engine to delete a user-defined stored button and returns 1; otherwise returns a negative error code from the list at the beginning of this chapter.

  `Port` is one of **`GE_PORTZ0`**, **`GE_PORTZ1`**, **`GE_PORTA`**, or **`GE_PORTB`**.

  `ButtonID` is the ID number of the stored button being deleted.

- **`int GEDisplayButtonWF(int Port, int ButtonID)`**

  This non-blocking version of the **`GEDisplayButton`** function immediately returns 0 while the Graphics Engine's response is pending, and so is suitable for use in costatements' **`waitfor`** statements. See the following function description for all other information.

- **`int GEDisplayButton(int Port, int ButtonID)`**

  If successful, instructs the Graphics Engine to display and to enable a user-defined button and returns 1; otherwise returns a negative error code from the list at the beginning of this chapter. Note that the button's screen region is automatically saved before the button is displayed.

  `Port` is one of **`GE_PORTZ0`**, **`GE_PORTZ1`**, **`GE_PORTA`**, or **`GE_PORTB`**.

  `ButtonID` is the ID number of the stored button being displayed.

- **`int GERemoveButtonWF(int Port, int ButtonID)`**

  This non-blocking version of the **`GERemoveButton`** function immediately returns 0 while the Graphics Engine's response is pending, and so is suitable for use in costatements' **`waitfor`** statements. See the following function description for all other information.

- **`int GERemoveButton(int Port, int ButtonID)`**

  If successful, instructs the Graphics Engine to disable and to remove from display a user-defined button and returns 1; otherwise returns a negative error code from the list at the beginning of this chapter. Note that the button's screen region is automatically restored after the button is removed from display.

  `Port` is one of **`GE_PORTZ0`**, **`GE_PORTZ1`**, **`GE_PORTA`**, or **`GE_PORTB`**.

  `ButtonID` is the ID number of the stored button being removed from display (but not deleted).

- **`int GEDisableButtonWF(int Port, int ButtonID)`**

  This non-blocking version of the **`GEDisableButton`** function immediately returns 0 while the Graphics Engine's response is pending, and so is suitable for use in costatements' **`waitfor`** statements. See the following function description for all other information.

- **`int GEDisableButton(int Port, int ButtonID)`**

  If successful, instructs the Graphics Engine to disable a user-defined button and returns 1; otherwise returns a negative error code from the list at the beginning of this chapter. Note that the button's screen display is unchanged.

  `Port` is one of **`GE_PORTZ0`**, **`GE_PORTZ1`**, **`GE_PORTA`**, or **`GE_PORTB`**.

  `ButtonID` is the ID number of the stored button being disabled.

- **`int GEEnableButtonWF(int Port, int ButtonID)`**

  This non-blocking version of the **`GEEnableButton`** function immediately returns 0 while the Graphics Engine's response is pending, and so is suitable for use in costatements' **`waitfor`** statements. See the following function description for all other information.

- **`int GEEnableButton(int Port, int ButtonID)`**

  If successful, instructs the Graphics Engine to enable a user-defined button and returns 1; otherwise returns a negative error code from the list at the beginning of this chapter. Note that the button's screen display is unchanged.

  `Port` is one of **`GE_PORTZ0`**, **`GE_PORTZ1`**, **`GE_PORTA`**, or **`GE_PORTB`**.

  `ButtonID` is the ID number of the stored button being enabled.

- **`int GELinkCellToMacWF(int Port, int LinkOnOff, int CellID, int MacroID)`**

  This non-blocking version of the **`GELinkCellToMac`** function immediately returns 0 while the Graphics Engine's response is pending, and so is suitable for use in costatements' **`waitfor`** statements. See the following function description for all other information.

- **`int GELinkCellToMac(int Port, int LinkOnOff, int CellID, int MacroID)`**

  If successful, instructs the Graphics Engine to (un)link a cell-push to a user-defined macro and returns 1; otherwise returns a negative error code from the list at the beginning of this chapter.

  `Port` is one of **`GE_PORTZ0`**, **`GE_PORTZ1`**, **`GE_PORTA`**, or **`GE_PORTB`**.

  `LinkOnOff` is a flag which: if 1, links the cell-push to the macro; if 0, unlinks the cell-push from the macro.

  `CellID` is the ID number of the cell to be (un)linked.

  `MacroID` is the ID number of the macro to be (un)linked.

- **`int GELinkBtnToMacWF(int Port, int LinkOnOff, int ButtonID, int MacroID)`**

  This non-blocking version of the **`GELinkBtnToMac`** function immediately returns 0 while the Graphics Engine's response is pending, and so is suitable for use in costatements' **`waitfor`** statements. See the following function description for all other information.

- **`int GELinkBtnToMac(int Port, int LinkOnOff, int ButtonID, int MacroID)`**

  If successful, instructs the Graphics Engine to (un)link a user-defined button-push to a user-defined macro and returns 1; otherwise returns a negative error code from the list at the beginning of this chapter.

  `Port` is one of **`GE_PORTZ0`**, **`GE_PORTZ1`**, **`GE_PORTA`**, or **`GE_PORTB`**.

  `LinkOnOff` is a flag which: if 1, links the button-push to the macro; if 0, unlinks the button-push from the macro.

  `ButtonID` is the ID number of the button to be (un)linked.

  `MacroID` is the ID number of the macro to be (un)linked.

- **`int GEStopMacroWF(int Port)`**

  This non-blocking version of the **`GEStopMacro`** function immediately returns 0 while the Graphics Engine's response is pending, and so is suitable for use in costatements' **`waitfor`** statements. See the following function description for all other information.

- **`int GEStopMacro(int Port)`**

  If successful, instructs the Graphics Engine to stop playing a macro and returns 1; otherwise returns a negative error code from the list at the beginning of this chapter.

  `Port` is one of **`GE_PORTZ0`**, **`GE_PORTZ1`**, **`GE_PORTA`**, or **`GE_PORTB`**.

- **`int GESetStringWF(int Port, char *string)`**

  This non-blocking version of the **`GESetString`** function immediately returns 0 while the Graphics Engine's response is pending, and so is suitable for use in costatements' **`waitfor`** statements. See the following function description for all other information.

- **`int GESetString(int Port, char *string)`**

  If successful, instructs the Graphics Engine to set a default string for the Virtual Keyboard and returns 1; otherwise, returns a negative error code from the list at the beginning of this chapter.

  `Port` is one of **`GE_PORTZ0`**, **`GE_PORTZ1`**, **`GE_PORTA`**, or **`GE_PORTB`**.

  `string` is a pointer to the zero-terminated (ASCIIZ) string to be set.

- **`int GESetLongWF(int Port, long lValue)`**

  This non-blocking version of the **`GESetLong`** function immediately returns 0 while the Graphics Engine's response is pending, and so is suitable for use in costatements' **`waitfor`** statements. See the following function description for all other information.

- **`int GESetLong(int Port, long lValue)`**

  If successful, instructs the Graphics Engine to set a default long integer value for the Virtual Keyboard and returns 1; otherwise, returns a negative error code from the list at the beginning of this chapter.

  `Port` is one of **`GE_PORTZ0`**, **`GE_PORTZ1`**, **`GE_PORTA`**, or **`GE_PORTB`**.

  `lValue` is the long integer value to be set.

- **`int GESetFloatWF(int Port, float fValue)`**

  This non-blocking version of the **`GESetFloat`** function immediately returns 0 while the Graphics Engine's response is pending, and so is suitable for use in costatements' **`waitfor`** statements. See the following function description for all other information.

- **`int GESetFloat(int Port, float fValue)`**

  If successful, instructs the Graphics Engine to set a default float value for the Virtual Keyboard and returns 1; otherwise, returns a negative error code from the list at the beginning of this chapter.

  `Port` is one of **`GE_PORTZ0`**, **`GE_PORTZ1`**, **`GE_PORTA`**, or **`GE_PORTB`**.

  `fValue` is the float value to be set.

- **`int GESetCharWF(int Port, char cValue)`**

  This non-blocking version of the **`GESetChar`** function immediately returns 0 while the Graphics Engine's response is pending, and so is suitable for use in costatements' **`waitfor`** statements. See the following function description for all other information.

- **`int GESetChar(int Port, char cValue)`**

  If successful, instructs the Graphics Engine to append a character to the string used by the Virtual Keyboard and returns 1; otherwise, returns a negative error code from the list at the beginning of this chapter.

  `Port` is one of **`GE_PORTZ0`**, **`GE_PORTZ1`**, **`GE_PORTA`**, or **`GE_PORTB`**.

  `cValue` is the character to be appended.

- **`int GESetTodWF(int Port, struct tm *Time)`**

  This non-blocking version of the **`GESetTod`** function immediately returns 0 while the Graphics Engine's response is pending, and so is suitable for use in costatements' **`waitfor`** statements. See the following function description for all other information.

- **`int GESetTod(int Port, struct tm *Time)`**

  If successful, instructs the Graphics Engine to set the date and time of day to both its system and real time clock and returns 1; otherwise, returns a negative error code from the list at the beginning of this chapter.

  `Port` is one of **`GE_PORTZ0`**, **`GE_PORTZ1`**, **`GE_PORTA`**, or **`GE_PORTB`**.

  `Time` is a pointer to a time structure containing the Graphics Engine's new time to be set.

-
  ```
  int GEKeyBoardMacroWF(int Port, char *Prompt,
          int OnOff, int Flags, void *max,
          void *min)
  ```

  This non-blocking version of the **`GEKeyBoardMacro`** function immediately returns 0 while the Graphics Engine's response is pending, and so is suitable for use in costatements' **`waitfor`** statements. See the following function description for all other information.

-
  ```
  int GEKeyBoardMacro(int Port, char *Prompt,
          int OnOff, int Flags, void *max,
          void *min)
  ```

  If successful, instructs the Graphics Engine to (de)activate the built-in Virtual Keyboard macro and returns 1; otherwise, returns a negative error code from the list at the beginning of this chapter.

  `Port` is one of **`GE_PORTZ0`**, **`GE_PORTZ1`**, **`GE_PORTA`**, or **`GE_PORTB`**.

  `Prompt` is a pointer to the zero-terminated (ASCIIZ) prompt line string; may be **`NULL`** if **`OnOff`** is 0.

  `OnOff` is a flag which: if 1, activates the virtual keyboard macro; if 0, deactivates the virtual keyboard macro.

  `Flags` is the bit-coded virtual keyboard properties value where:
  if bit 0 set (1), ENTER key sends unsolicited response to host;
  if bit 1 set, ENTER key deactivates the virtual keyboard;
  if bit 2 set, password mode (echo stars);
  if bits 3-4 are 0-0, string data type (any entry is OK);
  if bits 3-4 are 0-1, long integer data type (may check limits);
  if bits 3-4 are 1-0, float data type (may check limits);
  bit 5 is reserved for future use;
  if bit 6 set, long integer or float data is subject to low limit check;
  if bit 7 set, long integer or float data is subject to high limit check.

  `max` is a pointer to the maximum (long integer or float) value accepted by the virtual keyboard; **`NULL`** if no upper limit is required.

  `min` is a pointer to the minimum (long integer or float) value accepted by the virtual keyboard; **`NULL`** if no lower limit is required.

- **`int GEPlayMacroWF(int Port, int MacroID)`**

  This non-blocking version of the `GEPlayMacro` function immediately returns 0 while the Graphics Engine's response is pending, and so is suitable for use in costatements' `waitfor` statements. See the following function description for all other information.

- **`int GEPlayMacro(int Port, int MacroID)`**

  If successful, instructs the Graphics Engine to start playing a user-defined macro and returns 1; otherwise returns a negative error code from the list at the beginning of this chapter.

  `Port` is one of `GE_PORTZ0`, `GE_PORTZ1`, `GE_PORTA`, or `GE_PORTB`.

  `MacroID` is the ID number of the macro to be played (run).

- **`int GEShiftMacroWF(int Port, int MacroID, int Xofst, int Yofst)`**

  This non-blocking version of the `GEShiftMacro` function immediately returns 0 while the Graphics Engine's response is pending, and so is suitable for use in costatements' `waitfor` statements. See the following function description for all other information.

- **`int GEShiftMacro(int Port, int MacroID, int Xofst, int Yofst)`**

  If successful, instructs the Graphics Engine to start playing a user-defined macro with shifted (offset) X,Y coordinates and returns 1; otherwise returns a negative error code from the list at the beginning of this chapter.

  `Port` is one of `GE_PORTZ0`, `GE_PORTZ1`, `GE_PORTA`, or `GE_PORTB`.

  `MacroID` is the ID number of the macro to be played (run).

  `Xofst` is the horizontal offset applied to the shifted macro's draw commands.

  `Yofst` is the vertical offset applied to the shifted macro's draw commands.

- **`int GEDeleteMacroWF(int Port, int MacroID)`**

  This non-blocking version of the `GEDeleteMacro` function immediately returns 0 while the Graphics Engine's response is pending, and so is suitable for use in costatements' `waitfor` statements. See the following function description for all other information.

- **`int GEDeleteMacro(int Port, int MacroID)`**

  If successful, instructs the Graphics Engine to delete a stored user-defined macro and returns 1; otherwise returns a negative error code from the list at the beginning of this chapter.

`MacroID` is the ID number of the macro to be deleted.

`Port` is one of **`GE_PORTZ0`**, **`GE_PORTZ1`**, **`GE_PORTA`**, or **`GE_PORTB`**.

- **`int GEDelayPlayWF(int Port, long mDelay)`**

  This non-blocking version of the **`GEDelayPlay`** function immediately returns 0 while the Graphics Engine's response is pending, and so is suitable for use in costatements' **`waitfor`** statements. See the following function description for all other information.

- **`int GEDelayPlay(int Port, long mDelay)`**

  If successful, instructs the Graphics Engine to delay for a time and returns 1; otherwise returns a negative error code from the list at the beginning of this chapter. Note that when this command is in a macro it will always produce a delay in the execution sequence; however, when this command is issued directly the following (subsequent) commands are processed without delay:

  ```
  BeginMacro, DefineButton, DeleteButton,
  DeleteMacro, DisableButton, DisableCells,
  EnableButton, EnableCells, LinkBtnToMac,
  LinkCellToMac, LoopMacro, SetCellActive,
  SetFont, SetTod, SendChar, SendFloat,
  SendLastPush, SendLong, SendStatus, SendString,
  SendTod, StopMacro, SuperReset.
  ```

  `Port` is one of **`GE_PORTZ0`**, **`GE_PORTZ1`**, **`GE_PORTA`**, or **`GE_PORTB`**.

  `mDelay` is the milliseconds delay time.

- **`int GELoopMacroWF(int Port)`**

  This non-blocking version of the **`GELoopMacro`** function immediately returns 0 while the Graphics Engine's response is pending, and so is suitable for use in costatements' **`waitfor`** statements. See the following function description for all other information.

- **`int GELoopMacro(int Port)`**

  If successful, instructs the Graphics Engine to replay from the beginning a stored user-defined macro and returns 1; otherwise returns a negative error code from the list at the beginning of this chapter.

  `Port` is one of **`GE_PORTZ0`**, **`GE_PORTZ1`**, **`GE_PORTA`**, or **`GE_PORTB`**.

- **`int GEDisplayButtonQWF(int Port, int btnID,`**
  `char fxn)`

  This non-blocking version of the **`GEDisplayButtonQ`** function immediately returns 0 while the Graphics Engine's response is pending, and so is suitable for use in costatements' **`waitfor`** statements. See the following function description for all other information.

- **`int GEDisplayButtonQ(int Port, int btnID,`**
  `        char fxn)`

  If successful, instructs the Graphics Engine to display (either normally or inverted) and to enable a user-defined button and returns 1; otherwise returns a negative error code from the list at the beginning of this chapter. Note that the button's screen region is automatically saved before the button is displayed.

  `Port` is one of **`GE_PORTZ0`**, **`GE_PORTZ1`**, **`GE_PORTA`**, or **`GE_PORTB`**.

  `btnID` is the ID number of the stored button being displayed.

  `fxn` is a flag which: if 1, sets normal button; if 2, sets inverted button.

- **`int GELockBufferWF(int Port, char LockUnlock)`**

  This non-blocking version of the **`GELockBuffer`** function immediately returns 0 while the Graphics Engine's response is pending, and so is suitable for use in costatements' **`waitfor`** statements. See the following function description for all other information.

- **`int GELockBuffer(int Port, char LockUnlock)`**

  If successful, instructs the Graphics Engine to (un)lock its screen display buffer and returns 1; otherwise returns a negative error code from the list at the beginning of this chapter.

  `Port` is one of **`GE_PORTZ0`**, **`GE_PORTZ1`**, **`GE_PORTA`**, or **`GE_PORTB`**.

  `LockUnlock` is a flag which: if 1, unlocks the buffer; if 0, locks the buffer.

- **`int GEGetPcktWF(int Port, char *cPacket)`**

  This non-blocking function operates identically to the **`GEGetPckt`** function. See the following function description for all information.

- **`int GEGetPckt(int Port, char *cPacket)`**

  If successful, copies a Graphics Engine's unsolicited response packet (less the CRC) to the specified buffer and returns 1; otherwise immediately returns 0 if a complete packet is not available, or returns a negative error code from the list at the beginning of this chapter. Note that both this function and the **`GEGetPcktWF`** function return 0 when a Graphics Engine's response is pending, and so are equally suitable for use in costatements' **`waitfor`** statements.

  `Port` is one of **`GE_PORTZ0`**, **`GE_PORTZ1`**, **`GE_PORTA`**, or **`GE_PORTB`**.

  `cPacket` is a pointer to where the Graphics Engine's unsolicited response packet (less the CRC) will be stored.

- **`int GEDisableBtnAreaWF(int Port, char CellTL, char CellBR)`**

  This non-blocking version of the **`GEDisableBtnArea`** function immediately returns 0 while the Graphics Engine's response is pending, and so is suitable for use in costatements' **`waitfor`** statements. See the following function description for all other information.

- **`int GEDisableBtnArea(int Port, char CellTL, char CellBR)`**

  If successful, instructs the Graphics Engine to disable all currently enabled user-defined buttons that are at least partially in the desgnated touch-screen area and returns 1; otherwise returns a negative error code from the list at the beginning of this chapter. Note that the buttons' screen display is unchanged.

  `Port` is one of **`GE_PORTZ0`**, **`GE_PORTZ1`**, **`GE_PORTA`**, or **`GE_PORTB`**.

  `ButtonID` is the ID number of the stored button being disabled.

  `CellTL` is the ID number of the area's top left touch-screen cell.

  `CellBR` is the ID number of the area's bottom right touch-screen cell.

- **`int GEEnableBtnAreaWF(int Port, char CellTL, char CellBR)`**

  This non-blocking version of the **`GEEnableBtnArea`** function immediately returns 0 while the Graphics Engine's response is pending, and so is suitable for use in costatements' **`waitfor`** statements. See the following function description for all other information.

- **`int GEEnableBtnArea(int Port, char CellTL, char CellBR)`**

  If successful, instructs the Graphics Engine to re-enable all DisableBtnArea command-disabled user-defined buttons that are at least partially in the desgnated touch-screen area and returns 1; otherwise returns a negative error code from the list at the beginning of this chapter. Note that the buttons' screen display is unchanged.

  `Port` is one of **`GE_PORTZ0`**, **`GE_PORTZ1`**, **`GE_PORTA`**, or **`GE_PORTB`**.

  `CellTL` is the ID number of the area's top left touch-screen cell.

  `CellBR` is the ID number of the area's bottom right touch-screen cell.

CHAPTER 8: OTHER LIBRARIES

The libraries described in this chapter are specific to one or more types of controllers.

5KEY.LIB

These LCD and keypad functions support the PK2100 and PK2200 series controllers. This is the ***old five-key system***. It uses the real-time kernel (RTK). The standard LCD is 2 × 20. To run the five-key system with a 2 × 16 LCD, write **`#define LCD16x2`** at the start of the program.

- **`void _5keysettime( char *time )`**

  Sets real-time clock time, based on string **`*time`**. The string format is "hh:mm:ss".

- **`void _5keysetdate( char *date )`**

  Sets real-time clock date, based on string pointed to by **`*date`**. The string format is "mm-dd-yy".

- **`void _5keygettime( char *time )`**

  Gets real-time clock time and stores it in **`*time`**. The string format is "hh:mm:ss".

- **`void _5keygetdate( char *date )`**

  Gets real-time clock date and stores it in **`*date`**. The string format is "mm-dd-yy".

- **`void lcd–server( char mode, long position, char *lcd_msg )`**

  Clears number of lines, specified by **`mode`**, and displays message **`*lcd_msg`** at **`position`**. See **`CPLC.LIB`** for description of position fields.

- **`int _5key_float( char *label, float *data, float max, float min, char *help[], char size, char modify, char delay )`**

  This is the five-key system handler for a float parameter. It modifies or monitors the following parameters.

`label`	the address of the item label (string)
`value`	pointer to a **`float`** variable
`max, min`	the data limits
`help[]`	an array of help strings
`size`	size of the help array (two times the actual number of help lines); use **`sizeof(help)`**
`modify`	if 1, **`value`** is updated; if 0, **`value`** is only monitored
`delay`	number of 25 ms RTK ticks after which the software will release the current five-key task, freeing other lower priority tasks.

The function returns an integer representing one of the following keys: MENU, ITEM, ADD or DELETE. It returns –1 when no key has been pressed.

- **`int _5key_integer( char *label, int *value, int max, int min, char *help[], char size, char modify, char delay )`**

This is the five-key system handler for an integer parameter. It modifies or monitors the following parameters.

`label`	the item label (string)
`value`	pointer to an integer variable
`min, max`	the data limits
`help[]`	an array of help strings
`size`	size of the help array (two times the actual number of help lines); use `sizeof(help)`
`modify`	if 1, `value` is updated; if 0, `value` is only monitored
`delay`	number of 25 ms RTK ticks after which the software will release the current five-key task, freeing other lower priority tasks.

The function returns an integer representing one of the following keys: MENU, ITEM, ADD or DELETE. It returns –1 when no key has been pressed.

- **`int _5key_boolean( char *label, char *value, char *help[], char size, char modify, char delay )`**

This is the five-key system handler for a Boolean parameter. It modifies or monitors the following parameters.

`label`	the item label (string)
`value`	pointer to a “Boolean” variable
`help[]`	an array of help strings
`size`	size of the help array (two times the actual number of help lines); use `sizeof(help)`
`modify`	if 1, `value` is updated; if 0, `value` is only monitored
`delay`	number of 25-millisecond RTK ticks after which the software will release the current five-key task, freeing other lower priority tasks.

The function returns an integer representing one of the following keys: MENU, ITEM, ADD or DELETE. It returns –1 when no key has been pressed.

- **`int _5key_time( char *label, char *string,`**
  `        char *help[], char size,`
  `        char set_clock, char modify,`
  `        char delay )`

This is the five-key system handler for a time parameter. It modifies or monitors the following parameters.

`label`	the item label (string)
`string`	the time string
`help[]`	an array of help strings
`size`	size of the help array (two times the actual number of help lines); use **`sizeof(help)`**
`set_clock`	if non-zero, set the real-time clock
`modify`	if 1, **`value`** is updated; if 0, **`value`** is only monitored
`delay`	number of 25 ms RTK ticks after which the software will release the current five-key task, freeing other lower priority tasks.

The function returns an integer representing one of the following keys: MENU, ITEM, ADD or DELETE. It returns –1 when no key has been pressed.

- **`int _5key_date( char *label, char *string,`**
  `        char *help[], char size,`
  `        char set_clock, char modify,`
  `        char delay )`

This is the five-key system handler for a date parameter. It modifies or monitors the following parameters.

`label`	the item label (string)
`string`	the date string
`help[]`	an array of help strings
`size`	size of the help array (two times the actual number of help lines); use **`sizeof(help)`**
`set_clock`	if non-zero, set the real-time clock
`modify`	if 1, **`value`** is updated; if 0, **`value`** is only monitored
`delay`	number of 25 ms RTK ticks after which the software will release the current five-key task, freeing other lower priority tasks.

The function returns an integer representing one of the following keys: MENU, ITEM, ADD or DELETE. It returns –1 when no key has been pressed.

- **`void _5key_setmenu( char *menu, char *item,`**
  `char  mode, void *ptr, float max,`
  `float min, char *help[], char size,`
  `char modify, char delay, char display )`

  Adds a five-key item to the five-key linked list. Items with the same menu label are grouped together. The following parameters are used.

  `menu` the menu label (string)

  `item` the item label (string)

  `mode` type of data being created according to the list below

Data Mode	Type	Macro
0	**`float`**	**`_5key_Fdata`**
1	**`int`**	**`_5key_Idata`**
2	**`char`** (Boolean)	**`_5key_Bdata`**
3	time strings	**`_5key_Tdata`**
4	date strings	**`_5key_Ddata`**

  `ptr` pointer to the data

  `max, min` the data limits

  `help[]` array of help strings

  `size` size of the help array (two times the actual number of help lines); use **`sizeof(help)`**

  `modify` determines the handling of the data. To modify or to monitor.

  `delay` number of RTK ticks (25 ms) after which the software will release the current five-key task, allowing other lower priority tasks to execute

  `display` 1 if item is to be added to the list of periodically displayed items; 0 if item is not to be added to the list.

- **`int _5key_init_item( _5KEYITEM *thisitem,`**
  `char *d-menu, char *d_item,`
  `char data_mode, void *data_ptr,`
  `float max_data, float min_data,`
  `char *my_help[], char help_line,`
  `char data_modify, char delay )`

  Is called by **`_5key_setmenu`** to create a five-key item. The following parameters are used.

  `thisitem` points to a five-key item structure for the five-key link list

  `d_menu` points to a menu label

  `d_item` points to an item label

  `data_mode` type of data being created according to the list below

Data Mode	Type	Macro
0	**`float`**	**`_5key_Fdata`**
1	**`int`**	**`_5key_Idata`**
2	**`char`** (Boolean)	**`_5key_Bdata`**
3	time strings	**`_5key_Tdata`**
4	date strings	**`_5key_Ddata`**

  `max_data` is the upper limit and **`min_data`** is the lower limit for the data

  `my_help[]` is a list of help strings

  `help_line` is twice the actual number of help strings

  `data_modify` is 1 if data are to be modified through the five-key system; else 0, if data are just monitored

  `delay` is the five-key task suspend period

  `idisp` is 1 if data are to be displayed periodically when there are no keypad and LCD activities; else 0.

- **`int _5key_server( _5KEYITEM *t_item )`**

  Services a five-key item for display to the LCD and actions. The function returns any of the five-key menu keys pressed.

- **`void _5key_menu( void )`**

  Services the linked list created with **`_5key_setmenu()`**. This function must be called inside an RTK task.

- **`void _5key-setalarm ( int(*func1)(), int(*func2)(), int(*func3)(), int(*func4)() )`**

  Sets up the service functions for the software alarms.

  > **`func1()`**, the service function for _ALARM1
  > **`func2()`**, the service function for _ALARM2
  > **`func3()`**, the service function for _ALARM3, and
  > **`func4()`**, the service function for _ALARM4.

  All the functions default to **`NO_FUNCTION`**. Service functions can be changed or turned off at run-time as long as there is no conflict with the execution of a service function.

- **`void _5key_setfunc ( int(*func1)(), int(*func2)(), int(*func3)(), int(*func4)() )`**

  Sets up the service functions for the function keys.

  > **`func1()`**, the service function for F1
  > **`func2()`**, the service function for F2
  > **`func3()`**, the service function for F3, and
  > **`func4()`**, the service function for F4.

  All the functions default to **`NO_FUNCTION`**. Service functions can be changed or turned off at run-time as long as there is no conflict with the execution of a service function.

- **`void _5key_setmsg( byte message_no, char *the_message )`**

  Sets one of ten message strings for periodic display.

`message_no`	the message number, 0–9
`the_message`	the message string.

  All the messages default to NULL.

5KEYEXTD.LIB

These keypad functions support the PK2100 and PK2200 series controllers. They use the real-time kernel (RTK).

- **`int _5key_12out( void )`**

  This is the five-key server for the ten “virtual” digital outputs and two “virtual” relay outputs. The digital output and the relay output states can be modified through the five-key system. If an output state changes, this function will refresh the display to reflect the change.

  The function returns an integer representing one of the following keys: MENU, ITEM, ADD or DELETE. It returns –1 when no key has been pressed.

- **`int _5key_dacout( void )`**

  This is the five-key server for the “virtual” DAC channel. If the output value changes, this function will refresh the screen to reflect it.

  The function returns an integer representing one of the following keys: MENU, ITEM, ADD or DELETE. It returns –1 when no key has been pressed.

- **`int _5key_uinput( void )`**

  This is the five-key server for the six “virtual” universal inputs. If an input state changes, this function will refresh the display to reflect it.

  The function returns an integer representing one of the following keys: MENU, ITEM, ADD or DELETE. It returns –1 when no key has been pressed.

- **`int _5key_diginput( void )`**

  This is the five-key server for the seven “virtual” digital inputs. If an input state changes, this function will refresh the screen to reflect it.

  The function returns an integer representing one of the following keys: MENU, ITEM, ADD or DELETE. It returns –1 when no key has been pressed.

- **`int _5key-bank1dig( void )`**

  This is the five-key server for the “virtual” digital inputs 1–8. If an input state changes, this function will refresh the screen to reflect it.

  The function returns an integer representing one of the following keys: MENU, ITEM, ADD or DELETE. It returns –1 when no key has been pressed.

- **`int _5key_bank2dig( void )`**

  This is the five-key server for the “virtual” digital inputs 9–16. If an input state changes, this function will refresh the screen to reflect it.

  The function returns an integer representing one of the following keys: MENU, ITEM, ADD or DELETE. It returns –1 when no key has been pressed.

- **`int _5key_14out( void )`**

  This is the five-key server for the 14 “virtual” digital outputs. The digital output states can be modified through the five-key system. If an output state changes, this function will refresh the display to reflect it.

  The function returns an integer representing one of the following keys: MENU, ITEM, ADD or DELETE. It returns –1 when no key has been pressed.

CPLC.LIB

These functions support the PK2100 and PK2200 series controllers.

- **`void uplc_init( void )`**

  Initializes drivers and variables for the following.

  interrupt routine for background timer 1
  LCD, when selected
  keypad, if selected (keypad is scanned at 25 ms)
  virtual drivers, virtual timers and virtual watchdogs, when selected.

  The timer 1 interrupt routine also services the watchdog timer.

- **`void lc_kxinit( void )`**

  Initializes keypad driver and associated variables as well as virtual watchdog variables.

- **`void up_beepvol( int vol )`**

  Sets beeper volume: **`vol`** = 1 for low volume; 2 for high volume.

- **`void lc_loadtab( int *tab, int tab_size )`**

  Loads **`tab`** tables to match LCD screen.

- **`void lc_settab( char flag )`**

  Sets the tab variable **`lc_usetab`**.

- **`int lc_kxget( char mode )`**

  Fetches key value from FIFO keypad buffer. If **`mode`** = 0, value is removed from buffer; else value remains in buffer.

  The function returns the key value, or –1 if no key was pressed.

- **`void lc_setbeep( int delay )`**

  Sets beeper duration for **`delay`** counts of 1280 Hz cycles.

- **`void up_beep( unsigned int k )`**

  Sets beeper on for **`k`** milliseconds.

- **`unsigned int up_lastkey( void )`**

  Returns time since last key was pressed, in units of 1/40 second. The function returns elapsed time.

- **`void lc_init_keypad( void )`**

  Initializes **`timer1`**, keypad driver and variables, and the real-time kernel.

- **`void GLOBAL_INIT( void )`**

  Refer to **`VDRIVER.LIB`** for a description of this function.

- **`int up_synctimer( void )`**

  Synchronizes the virtual **`SEC_TIMER`** with the real-time clock (RTC). The function returns 0 if RTC is read properly, and –1 otherwise.

DRIVERS.LIB

These are miscellaneous hardware drivers.

- **`int plcport( int bit )`**

  Checks the specified bit of the PLCBus port. The function returns 1 if the specified bit is set, or 0 if not.

- **`void set16adr( int address )`**

  Sets the current address for the PLCBus. All read and write operations will access this address until a new address is set. **`address`** is a 16-bit physical address (for 4-bit bus). The high-order nibble contains the expansion register value, while the remaining nibbles form a 12-bit address (the first and third nibbles must be swapped).

- **`void set12adr( int address )`**

  Sets the current address for the PLCBus. All read and write operations will access this address until a new address is set. **`address`** is a 12-bit physical address (for 4-bit bus) with the first and third nibbles swapped (most significant nibble are in the low four bits).

- **`void set4adr( int address )`**

  Sets the current address for the PLCBus. All read and write operations will access this address until a new address is set. **`address`** contains the last 4 bits of the physical address (for 4-bit bus) in bits 8–11. A 12-bit address may be passed to this function, but only the last 4 bits will be set. This function should only be called if the first 8 bits of the address are the same as the address in the previous call to **`set12adr`**.

- **`char read4data( int address )`**

  Sets the last 4 bits of the current PLCBus address using **`address`** (bits 8–11), then reads 4 bits of data off of the bus with a **`BUSRD0`** cycle. Returns PLCBus data in the lower 4 bits (upper bits are undefined).

- **`char read12data( int address )`**

  Sets the current PLCBus address using the 12-bit **`address.`** Then reads 4 bits of data off of the bus with a **`BUSRD0`** cycle. The function returns PLCBus data in the lower 4 bits (upper bits are undefined).

- **`void write4data( int address, char data )`**

  Sets the last 4 bits of the current PLCBus address using **`address`** (bits 8–11). Then writes the low 4 bits of **`data`** to the bus.

- **`void write12data( int address, char data )`**

  Sets the current PLCBus address using the 12-bit **`address.`** Then writes the low 4 bits of **`data`** to the bus.

- **`void hv_wr( char v )`**

  Writes 8 bits to the high-voltage driver. Each bit affects one high-voltage output. A 1 enables the corresponding output; 0 disables the output.

- **`void hv_enb( void )`**

  Enables high-voltage driver.

- **`void hv_dis( void )`**

  Disables high-voltage driver.

- **`void lcd_init( char mode )`**

  Initializes the LCD; **`mode`** should normally be set to 0x18.

- **`void lputc( char cc )`**

  Sends a character to the LCD and updates the cursor (without wrap-around); **`cc`** is the character to send: if the high bit is set, it will be treated as a control character. Possible control characters are as follows.

\n	Newline (position cursor to line 1, column 0)
\xFF	Clear screen
\xF0	Clear line 0
\xF1	Clear line 1
\xF2	Cursor OFF (cursor invisible, blink off)
\xF3	Cursor ON (solid cursor block)
\xF4	Cursor BLINK (blinks continuously)
\xF5	Shift display left
\xF6	Shift display right
\x80–\xA7	Position cursor at line 0
\xC0–\xE7	Position cursor at line 1

- **`void lcd_clr_line( char code )`**

  Clears a line on the LCD; **`code`** should be 0x80 to clear line 0 and 0xC0 to clear line 1.

- **`void lcd_wait( void )`**

  Waits until the LCD is ready to accept data.

- **`int lprintf( char *fmt, ... )`**

  Operates the same as **`printf`**, but outputs to LCD.

- **`char *lputs( char *p )`**

  Sends the null-terminated string **`*p`** to the LCD and updates the cursor (without wraparound). All characters (except null) are sent directly to the LCD; control characters are not recognized. The function returns a pointer to the string.

- **`void* intoff( void *ptr )`**

  Saves the current interrupt state in **`*ptr`** and then disables interrupts. The function returns the pointer **`ptr`**.

- **`void* inton( void *ptr )`**

  Enables interrupts if they were previously on, according to **`*ptr`**. **`ptr`** must have been set previously by a call to **`intoff`**. The function returns the pointer **`ptr`**.

- **`void doint( void )`**

  Enables interrupts for a short time and then disables them (if they were previously off). This allows interrupts to be processed in code where they are otherwise disabled.

- **`int tm_rd( struct tm *t )`**

  Reads the current system time into the structure **`t`**. This routine works with either the Toshiba or Epsom clocks. The function returns 0 if successful, and –1 if the clock is failing or is not installed.

  The following structure is used to hold the time and date:

  ```
  struct tm {
       char tm_sec;       // 0-59
       char tm_min;       // 0-59
       char tm_hour;      // 0-23
       char tm_mday;      // 1-31
       char tm_mon;       // 1-12
       char tm_year;      // 00-150 (1900-2050)
       char tm_wday;      // 0-6 where 0 means Sunday
    };
  ```

- **`int tm_wr( struct tm *t )`**

  Sets the system time according to the structure **`t`** (see the description in **`tm_rd`** above). This routine works with either the Toshiba or Epson clocks. The function returns 0 if successful, and –1 if the clock is failing or is not installed.

- **`void mktm( struct tm *timeptr, long time )`**

  Fills the structure pointed to by **`timeptr`** according to time, specified in seconds since January 1, 1980.

- **`long mktime( struct tm *timeptr )`**

  Converts the contents of **`timeptr`** into a long integer. The function returns time in seconds since January 1, 1980.

- **`long clock( void )`**

  Reads the system clock and converts time to a long integer. The function returns system time in seconds since January 1, 1980.

- **`long phy_adr( char *adr )`**

  Converts a logical (16-bit) address to a physical (20-bit) address. **`adr`** points to the address. The function returns 20-bit address as a long integer.

- **`void dmacopy( long dest, long src,`**
  `unsigned int count )`

  Uses DMA to copy **`count`** bytes from one physical address (**`src`**) to another (**`dest`**).

- **`void outportn( int port, char *buf, char count )`**

  Writes **`count`** bytes to the specified output port. **`buf`** points to the sequence of bytes to write.

- **`void init_timer0( unsigned int count )`**

  Initializes timer 0. **`count`** is the value placed in the reload register. Some common count values and the frequencies they generate are provided below for a 9.216 MHz clock.

9126	50 Hz	7680	60 Hz	7200	64 Hz
4608	100 Hz	2304	200 Hz	1152	400 Hz
900	512 Hz	600	768 Hz	500	928 Hz
450	1024 Hz				

- **`void timer0_isr( void )`**

  `timer0` interrupt service routine, runs the real-time kernel.

- **`void init_timer1( unsigned int count )`**

  Initializes **`timer1`**. **`count`** is the value placed in the reload register. Some common count values and the frequencies they generate are provided below for a 9.216 MHz clock.

9126	50 Hz	7680	60 Hz	7200	64 Hz
4608	100 Hz	2304	200 Hz	1152	400 Hz
900	512 Hz	600	768 Hz	500	928 Hz
450	1024 Hz				

- **`void tdelay( int msec )`**

  Waits for **`msec`** milliseconds, assuming that **`timer1`** is running at 750 Hz. The actual delay is related to the frequency of **`timer1`** by the formula delay = 3 × (**`msec`**/4)/**`freq1`**.

- **`void int_timer1( void )`**

  `timer1` interrupt service routine. Drives the beeper and keypad. Also runs the real-time kernel if **`RUNKERNEL`** is defined.

- **`void save_shadow( void )`**

  Saves PLCBus shadow registers on the stack.

- **`void restore_shadow( void )`**

  Restores PLCBus shadow registers from the stack and resets the current bus address.

- **`void write24data( long address, char data )`**

  Sets the current PLCBus address using the 24-bit **`address`**, then writes 8 bits of **`data`** to the bus.

- **`void write8data( long address, char data )`**

  Sets the last 8 bits of the current PLCBus address using **`address`** (bits 16–23), then writes 8 bits of **`data`** to the bus.

- **`int read24data0( long address )`**

  Sets the current PLCBus address using the 24-bit **`address`**, then reads 8 bits of data off of the bus with a **`BUSRD0`** cycle. The function returns PLCBus data in the lower 8 bits (upper bits are 0).

- **`int read8data0( long address )`**

  Sets the last 8 bits of the current PLCBus address using **`address`** (bits 16–23), then reads 8 bits of data from the bus with a **`BUSRD0`** cycle. The function returns PLCBus data in the lower 8 bits, with the upper bits 0.

- **`int read24data1( long address )`**

  Sets the current PLCBus address using the 24-bit **`address`**, then reads 8 bits of data from the bus with a **`BUSRD1`** cycle. The function returns PLCBus data in the lower 8 bits (upper bits are 0).

- **`int read8data1( long address )`**

  Sets the last 8 bits of the current PLCBus address using **`address`** (bits 16–23), then reads 8 bits of data from the bus with a **`BUSRD1`** cycle. The function returns PLCBus data in the lower 8 bits (upper bits are 0).

- **`void set24adr( long address )`**

  Sets the current address for the PLCBus. All read and write operations will access this address until a new address is set. **`address`** is a 24-bit physical address (for the 8-bit bus), with the first and third bytes swapped (low byte most significant).

- **`void set8adr( long address )`**

  Sets the current address for the PLCBus. All read and write operations will access this address until a new address is set. **`address`** contains the last 8 bits of the physical address (for the 8-bit bus) in bits 16–23. A 24-bit address may be passed to this function, but only the last 8 bits will be set. This function should only be called if the first 16 bits of the address are the same as the address in the previous call to **`set24adr`**.

- **`void plcbus_isr( void )`**

  This function is used to service all PLCBus /AT line interrupts. The /AT line is connected to INT1 of the Z180. Each interrupt service routine (ISR) is responsible for assuring its device releases the /AT signal once the ISR has been performed.

- **`void relocate_int1( void )`**

  Reprograms the **`INT1`** vector.

- **`int DelayTicks( CoData *pfb, unsigned int ticks )`**

  Provides tick time mechanism for costatements. Ticks occur 1280 times per second. (The period is 781.25 µs.) The function returns 1 if the specified tick delay has lapsed. Otherwise, it returns 0.

- **`int DelayMs( CoData *pfb, long delayms )`**

  Provides millisecond time mechanism for costatements. The function returns 1 if the specified millisecond delay has lapsed. Otherwise, it returns 0.

- **`int DelaySec( FuncBlk *pfb, long delaysec )`**

  Provides second time mechanism for costatements. Returns 1 if the specified second delay has lapsed; otherwise, it returns 0.

- **`int eei_rd( int address )`**

  Reads two consecutive byte areas of the EEPROM for integer data. The low byte is from **`address`** and the high byte is from **`address+1`**. The function returns the integer at **`address`** from EEPROM.

- **`int eei_wr( int address, unsigned int value )`**

  Writes an integer value to the EEPROM at **`address`**. The lower byte is at **`address`** and the high byte is at **`address+1`**. The function returns 0 if the write was successful.

- **`void DMA0( unsigned int cnt )`**

  Loads **`cnt`** to DMA0 counter to count high-speed pulses in hardware. Maximum count is 64,000. **`_DMAFLAG0`** is set to 0. If the DMA has counted out, the interrupt service routine for DMA0 will generate an interrupt in which **`_DMAFLAG0`** is set to 1. Events are edge sensed. C1A and C1B must both be low for /DREQ0 to generate an interrupt.

- **`void DMA1( unsigned int cnt )`**

  Loads **`cnt`** to the DMA1 counter to count high-speed pulses in hardware. Maximum count is 64,000. **`_DMAFLAG1`** is set to 0. If the DMA has counted out, the interrupt service routine for DMA1 will generate an interrupt in which **`_DMAFLAG1`** is set to 1. Events are edge sensed. C2A and C2B must both be low for /DREQ1 to generate an interrupt. C2B uses one of the RS-485 receivers for differential input. For example, tie C2B– to 5 volts; when the signal at C2B+ is lower than 5 V, a negative edge is generated for the DMA counter.

- **`unsigned int DMASnapShot( char channel,`**
  `        unsigned int *count )`

  Takes a "snap shot" of a DMA **`channel`** (0 or 1) for the number of pulses counted. The function returns 0 if the pulse train is too fast to have a snapshot taken; or 1 if a snapshot is obtained and valid data is in **`*count`**.

- **`void powerdown( void )`**

  Turns the power off. Power can only be turned back on by external means. This only works for boards with a switching power supply (except for the PK2200).

- **`void powerup( void )`**

  Reverses the effect of powerdown so power stays on after external power is disabled. See **`powerdown`**.

- **`void nmiint( void )`**

  Default power-fail interrupt handler. The function does nothing and ***never returns***.

- **`void setperiodic( int period )`**

  Sets a timer to periodically power up the BL1100. After this call, the board may be put to sleep and will automatically awaken at the specified interval. Execution will begin in the main function when power is restored. **`period`** may be 4 (to wake once per second), 8 (to wake once per minute), or 12 (to wake once per hour). Works only for boards that have a switching power supply, except the PK2200.

- **`void sleep( void )`**

  Puts the controller to sleep. Works for all boards that use a switching power supply, except the PK2200.

  The function does not return.

- **`void init_timer( void )`**

  Initializes the system clock.

- **`void off_485( void )`**

  Turns off the RS-485 driver for Z180 port 1. Different controllers have different methods of driving RS-485.

- **`void on_485( void )`**

  Turns on the RS-485 driver for Z180 port 1. Different controllers have different methods of driving RS-485.

DMA.LIB

These functions support DMA use on all Z-World controllers.

- **`void DMA0Count( unsigned int count )`**

  Loads **`count`** to DMA0 counter to count high-speed pulses in hardware. The maximum count is 64,000. The function sets the flag **`_DMAFLAG0`** to 0. DMA0 causes an interrupt when **`count`** negative edges have been detected. The interrupt service routine for DMA0 will set **`_DMAFLAG0`** to 1. A user program can monitor **`_DMAFLAG0`** to determine whether **`count`** has finished.

- **`void DMA1Count( unsigned int count )`**

  Loads **`count`** to DMA1 counter to count high-speed pulses in hardware. The maximum count is 64,000. The function sets the flag **`_DMAFLAG1`** to 0. DMA1 will cause an interrupt when **`count`** negative edges have been detected. The interrupt service routine for DMA1 will set **`_DMAFLAG1`** to 1. A user program can monitor **`_DMAFLAG1`** to determine whether the count has finished.

- **`unsigned int DMASnapShot( byte channel,`**
  `unsigned int *count )`

  Reads the number of pulses that a DMA channel (**`channel`** = 0 or 1) has counted. A DMA counter is initialized with either **`DMA0Count`** or **`DMA1Count`**. The function returns 0 if a DMA channel is counting too fast to allow a stable reading of the **`count`** value. If the function reads a stable `count` value, it returns 1 and sets the parameter **`*count`**. Note that a DMA interrupt will still occur when the DMA channel finishes counting, even if the **`count`** cannot be read.

- **`void DMA0_Off( void )`**
  `void DMA1_Off( void )`

  Turns the named DMA channel off.

- **`unsigned int DMA0_SerialInit( byte channel,`**
  `byte mode, byte baud )`

  Initializes serial port **`channel`** (must be 0 or 1) of the Z180 for DMA0 to serial transfers.

  The term **`mode`** is defined as follows.

bit0 = 0 for 1 stop bit	1 for 2 stop bits
bit1 = 0 for no parity	1 for parity
bit2 = 0 for 7 data bits	1 for 8 data bits
bit3 = 0 for even parity	1 for odd parity.

  The term **`baud`** is the baud rate in multiples of 1200 bps (e.g., 8 for 9600 bps).

- **`unsigned int DMA0_Rx( byte port,`**
  `unsigned long address, unsigned int count,`
  `int interrupts, int increments )`

  Initiates a transfer using DMA0 to receive **`count`** bytes from a serial port (**`port`** = 0 or 1) to absolute memory locations starting at **`address`**. The logical memory address for ordinary arrays may be converted to a physical address with **`phy_adr(array)`**. Simply pass the array name directly for **`xdata`** arrays.

  DMA0 will generate an interrupt at the end of the transfer if **`interrupts`** is 1. The user program must provide the interrupt service routine. DMA0 does not generate an interrupt if **`interrupts`** is 0. The term **`increments`** must be 0 to increment the memory address, and 1 to decrement the memory address.

  The function returns 1 if successful, 0 if DMA0 is busy, –1 if the serial port is busy, and –2 if **`channel`** is not 0 or 1.

- **`unsigned int DMA0_Tx( byte port,`**
  `unsigned long address, unsigned int count,`
  `int interrupts, int increments )`

  Initiates a transfer using DMA0 to transmit **`count`** bytes to a serial port (**`port`** = 0 or 1) from absolute memory locations starting at **`address`**. The logical memory address for ordinary arrays may be converted to a physical address with **`phy_adr(array)`**. Simply pass the array name directly for **`xdata`** arrays.

DMA0 will generate an interrupt at the end of the transfer if **`interrupts`** is 1. The user program must provide the interrupt service routine. DMA0 generates no interrupt if **`interrupts`** is 0. The term **`increments`** must be 0 to increment the memory address, and 1 to decrement the memory address.

The function returns 1 if successful, 0 if DMA0 is busy, –1 if the serial port is busy, and –2 if **`channel`** is not 0 or 1.

- **`unsigned int DMA0_MM( unsigned long dst, unsigned long src, unsigned int count, int mode, int interrupts )`**

  Initiates a transfer using DMA0 to copy **`count`** bytes from absolute memory locations starting at **`src`** to absolute memory locations starting at **`dst`**. The logical memory address for ordinary arrays may be converted to a physical address with **`phy_adr(array)`**. Simply pass the array name directly for **`xdata`** arrays.

  DMA0 will generate an interrupt at the end of the transfer if **`interrrupts`** is 1. The user program must provide the interrupt service routine. DMA0 generates no interrupt if **`interrupts`** is 0. The term **`mode`** must be 0 for cycle-stealing transfers, and 1 for burst transfers.

  The function returns 1 if successful, and 0 if DMA0 is busy.

- **`unsigned int DMA0_MIO( unsigned int ioaddr, unsigned long memaddr, unsigned int count, int interrupts, int increments )`**

  Initiates a transfer using DMA0 to write **`count`** bytes from absolute memory locations starting at **`memaddr`** to the I/O port designated by **`ioaddr`**. The external device must generate negative-going /DREQ0 pulses for each byte transferred.

  The logical memory address for ordinary arrays may be converted to a physical address with **`phy_adr(array)`**. Simply pass the array name directly for **`xdata`** arrays.

  DMA0 will generate an interrupt at the end of the transfer if **`interrupts`** is 1. The user program must provide the interrupt service routine. DMA0 generates no interrupt if **`interrupts`** is 0. The term **`increments`** must be 0 to increment the memory address, and 1 to decrement the memory address.

  The function returns 1 if successful, and 0 if DMA0 is busy.

- **`unsigned int DMA0_IOM( unsigned long memaddr, unsigned int ioaddr, unsigned int count, int interrupts, int increments )`**

  Initiates a transfer using DMA0 to read **`count`** bytes from the I/O port designated by **`ioaddr`** to the absolute memory locations starting at **`memaddr`**. The external device must generate negative-going /DREQ0 pulses for each byte transferred. The logical memory address for ordinary arrays may be converted to a physical address with **`phy_adr(array)`**. Pass the array name directly for **`xdata`** arrays.

  DMA0 will generate an interrupt at the end of the transfer if **`interrupts`** is 1. The user program must provide the interrupt service routine. DMA0 generates no interrupt if **`interrupts`** is 0. The term **`increments`** must be 0 to increment the memory address, and 1 to decrement the memory address.

  The function returns 1 if successful, and 0 if DMA0 is busy.

- **`unsigned int DMA1_MIO( unsigned int ioaddr, unsigned long memaddr, unsigned int count, int interrupts, int increments )`**

  Initiates a transfer using DMA1 to write **`count`** bytes from absolute memory locations starting at **`memaddr`** to the I/O port designated by **`ioaddr`**. The external device must generate negative-going /DREQ1 pulses for each byte transferred. The logical memory address for ordinary arrays may be converted to a physical address with **`phy_adr(array)`**. Pass the array name directly for **`xdata`** arrays.

  DMA1 will generate an interrupt at the end of the transfer if **`interrupts`** is 1. The user program must provide the interrupt service routine. DMA1 generates no interrupt if **`interrupts`** is 0. The term **`increments`** must be 0 to increment the memory address, and 1 to decrement the memory address.

  The function returns 1 if successful, and 0 if DMA1 is busy.

- **`unsigned int DMA1_IOM( unsigned long memaddr, unsigned int ioaddr, unsigned int count, int interrupts, int increments )`**

  Initiates a transfer using DMA1 to read **`count`** bytes from the I/O port designated by **`ioaddr`** to the absolute memory locations starting at **`memaddr`**. The external device must generate negative-going /DREQ1 pulses for each byte transferred. The logical memory address for ordinary arrays may be converted to a physical address with **`phy_adr(array)`**. Pass the array name directly for **`xdata`** arrays.

DMA1 will generate an interrupt at the end of the transfer if **`interrupts`** is 1. The user program must provide the interrupt service routine. DMA1 generates no interrupt if **`interrupts`** is 0. The term **`increments`** must be 0 to increment the memory address, and 1 to decrement the memory address.

The function returns 1 if successful, and 0 if DMA1 is busy.

FK.LIB

These are LCD and keypad support functions for use without the real-time kernel (RTK).

- **`int fk_helpmsg( char **hptr )`**

  Displays a series of help messages when the HELP key is pressed. The current display is saved and each message string is displayed for 1.8 seconds, then the previous display is restored. The input should be an array of strings declared like this.

  ```
  char *hptr[]={"Str 1","Str 2",...,"StrN",""};
  ```

  The last string must be ***null***. The function returns non-zero if help is off, and zero if help is on.

- **`void fk_monitorkeypad( void )`**

  Monitors the keypad for keys pressed. This function should be called from an SRTK or RTK high-priority task. It sets global variable **`fk_tkey`** to values from 1 to 12 depending on the key pressed. The value is 0 if no key was pressed. The function also monitors for the 2-key reset combination. If a reset combination is detected, the function will not return but will force a watchdog timeout. There is no buffer. Key presses not processed within 100 ms will be lost.

- **`int fk_item_alpha( char *s1, char *var,`**
  `int wdsize )`

  Modifies a string using the five-key system. The term **`*s1`** is a string containing a prompt. The term **`*var`** is the string to be displayed and/or modified. The function returns 0 if not done, and 1 if done, and returns 1 or 0 in global variable **`fk_newmenu`**.

- **`int fk_item_int( char *string, int *num,`**
  `int lower, int upper )`

  Displays/modifies an integer number using the five-key system. The term **`*string`** is a **`printf`** format having the form **`%`*n*`u`** where *n* is 1 digit, for example, **`%5d`**. The term **`*num`** is the integer to be displayed and/or modified. The arguments **`upper`** and **`lower`** are the upper and lower limits for the number. The function returns 0 if not done, and 1 if done, and returns 1 or 0 in global variable **`fk_newmenu`**.

- **`int fk_item_unsigned int( char *string, unsigned int *num, unsigned int lower, unsigned int upper )`**

  This function is the same as **`fk_item_int`**, but applies to unsigned integers. (Remember that **`unsigned int`** is a convention in this manual only and is not a C keyword.)

- **`int fk_item_float( char*s1, float *num, float lower, float upper )`**

  Displays/modifies a floating-point number using the five-key system. The term **`*s1`** is a **`printf`** format for displaying the number. The format code should be in the form of **`%n.mf`**. The displayed line appears as follows.

  ```
  vvvvvv wwww.yyyy
  ```

  where **`vvvvv`** is a prompt string, **`wwww`** is *n* chars long, and **`yyyy`** is *m* chars long. The value *n* must be at least 1. The sum $n + m$ cannot exceed 9. The default is $n = 5$ and $m = 2$. The term **`*num`** is the floating-point number to be displayed and/or modified. The arguments **`upper`** and **`lower`** are the upper and lower limits for the number. This function will work for numbers in the ranges [1E6,–1E–4], [1E–4,1E6] with the appropriate format specification.

  The function returns 0 if not done, and 1 if done, and returns 1 or 0 in global variable **`fk_newmenu`**.

- **`int fk_item_enum( char *prompt, int *choice, char *s1,...*sn, "")`**

  Allows the user to choose from a list of null-terminated strings (maximum 20). The string **`*prompt`** must contain a string field code (**`%s`** or **`%ns`**) used to print the strings. The last of the strings (after **`*s1`**, ... **`*sn`**) must be ***null***. The term **`*choice`** returns the choice made by the user, from 0 to (n - 1). The function returns 0 if not done, and 1 if done, and returns 1 or 0 in global variable **`fk_newmenu`**.

- **`int fk_item_setdate( struct tm *time )`**

  A five-key function to modify the day, month and year fields of a **`tm`** structure. The term **`*time`** is the structure to be modified. The function returns 0 if not done, and 1 if done, and returns 1 or 0 in global variable **`fk_newmenu`**.

- **`int fk_item_settime( struct tm *time )`**

  A five-key function to modify the hour, minute and second fields of a **`tm`** structure. The term **`*time`** is the structure to be modified. The function returns 0 if not done, and 1 if done, and returns 1 or 0 in global variable **`fk_newmenu`**.

IOEXPAND.LIB

These are support functions for the BL1100 expansion boards. They are divided into two classes.

1. Functions that are hard-coded for default base addresses, 0xFxxx.
2. Functions that allow users to specify a board by its node number.

The former class is faster, but is limited to systems with one expansion board; the latter class, therefore, should be limited to multiple expansion board applications.

There is a structure of default addresses to improve lookup speeds for Class 2 functions. There is a structure that holds the default addresses. Instead of specifying a node number (0–3), specify –1. This will load the correct default addresses. The second set of functions allows for stacking of up to four expansion boards on top of the BL1100.

The board addresses are set through jumper J10.

Refer to the ***BL1100 User's Manual*** for the proper board addresses.

- **`int exp_init( int ppia, int ppib, int ppicu, int ppicl )`**

  Initializes the PIO ports of a BL1100 expansion card with the default address of 0xFxxx. The U5 PPI uses mode 0 or the basic I/O mode. **`ppia`**, **`ppib`**, **`ppicu`**,and **`ppicl`** are output values for the PPI output register. Configures Port A as input if **`ppia`** = –1, and Port B as input if **`ppib`** = -1. Configures port C upper nibble as inputs if **`ppicu`** = –1; and port C lower nibble as inputs if **`ppicl`** = –1. All PPI output ports are reset to low when the mode is changed. It is important to output a correct value to the output port right after the mode is changed.

- **`int mux_ch( int chan )`**

  Sets the DG509A multiplexer (U17) of the BL1100 expansion card with the default address of 0xFxxx. **`chan`** is 0 to 3 for (AN0–, AN0+) to (AN3–, AN3+), respectively, to multiplex on (MUX-DA, MUX-DB).

- **`int ad20_mux( int chan )`**

  Sets the multiplexer for the 20-bit AD7703 of the BL1100 expansion card with the default address of 0xFxxx. Channels 0 to 3 select unipolar operation (0 to 2.5 volts) for (AN0–, AN0+) to (AN3–, AN3+), respectively, while channels 4 to 7 select bipolar operation (-2.5 to 2.5 volts) for (AN0–, AN0+) to (AN3–, AN3+), respectively.

- **`int ad20_rdy( void )`**

  Tests AD7703 DRDY status from RDTTL bit 1. The function returns 0 if the AD20 is ready, or 1 if AD20 is busy.

- **`int ad20_cal( int mode )`**

  Calibrates the AD7703 on the BL1100 expansion card with the default address of 0xFxxx. Mode 0 calibration does not use the multiplexer. Mode 1 calibration uses the multiplexer to get zero and full scale on Ain. Mux ch0 is the A/D signal to be measured. Mux ch1 is Ain for the Mode 1 first step to calibrate the system offset. Mux ch2 is Ain for Mode 1 second step to calibrate the system gain. Mode 2 calibration uses the current channel to get Ain as zero to calibrate the system offset.

  The following shows the state of SC1 and SC2 during calibration:

Mode	SC1	SC2	Cal type	Zero	FS Steps
0	0	0	self-cal	AGND	REF+1
1	1	1	system offset	Ain	1st of 2
1	0	1	system gain	Ain	2nd of 2
2	1	0	system offset	Ain	REF+1

  The function returns 0, if calibration was completed, or –1, if there is an error during calibration.

- **`long ad20_rd( void )`**

  Reads 20-bit data from the AD7703 serial data port. The 125-millisecond step response time of AD7703 dictates that a time delay should be guaranteed after a multiplexer switching. A/D data will be valid when DRDY is low for data output at a rate up to 4 kHz. The polarity and channel to read should be set previously with **`ad20_mux`**. Ain ranges from 0 to 2.5 volts for the unipolar mode (PA0 = 0). LSB = 2.5 volts/ 1048576 = 2.384 microvolts. Ain ranges from –2.5 to +2.5 volts for the bipolar mode (PA0 = 1). LSB = 5 volts/1048576 = 4.768 microvolts.

  The function returns 20-bit A/D data. For the unipolar mode, 0x00000 = AGND, 0x7FFFF = 1.25 V, and 0xFFFFF = 2.5 V. For the bipolar mode, 0x00000 = –2.5 V, 0x7FFFF = AGND, and 0xFFFFF = 2.5 V.

- **`int exp_init_n( int node, int ppia, int ppib,`**
  `        int ppicu, int ppicl, int def )`

  Initializes the PIO port of a BL1100 expansion card corresponding to the specified node. Node is 0 to 3 for node addresses 0xCxxx to 0xFxxx, respectively. If node equals -1, the function uses the default address saved in **`def_na`**. If **`def`** equals 1, the node is saved as the default node in **`def_na`**. If **`def`** equals 0, the node is not saved.

The function returns 0 if initialization is okay, or –1 if an unknown mode is requested.

Consult the ***BL1100 User's Manual*** for the address configuration.

- **`int get_na( int node, struct node_addr *na )`**

  Gets the node address from the specified node (0–3). The function returns 0 if **`node`** is proper; or –1 if **`node`** is out of range. Node address data are returned in **`struct node_addr *na`**.

- **`int set_def_na( int node )`**

  Sets node address to default node address. The function returns data from **`get_na`**.

- **`int get_def_na( struct node_addr *na )`**

  Gets the default node address. The function returns the node number.

- **`int mux_ch_n( int node, int chan, int def )`**

  Sets DF509A multiplexers on specified BL1100 expansion card node. **`node`** is 0 to 3 for address 0xCxxx to 0xFxxx, respectively. **`chan`** is 0 to 3 for (AN0–, AN0+) to (AN3–, AN3+), respectively. If **`node`** equals –1, the function uses the default address saved in **`def_na`**. If **`def`** equals 1, the node is saved as default node in **`def_na`**. If **`def`** equals 0, the node is not saved. The function returns 0 if the **`mux`** setup is okay, or –1 if **`node`** is out of range.

- **`int ad20_mux_n( int node, int chan, int def )`**

  Sets the DG509A multiplexer for the 20-bit AD7703 of a BL1100 expansion card. **`node`** 0–3 specifies the node address 0xCxxx to 0xFxxx, respectively. **`chan`** 0–3 selects unipolar operation (0 to 2.5 volts) for (AN0–, AN0+) to (AN3–, AN3+), respectively. **`chan`** 4–7 selects bipolar operation (–2.5 to +2.5 volts) for (AN0–, AN0+) to (AN3–, AN3+), respectively. If **`node`** equals –1, the function uses the default address saved in **`def_na`**. If **`def`** equals 1, the node is saved as the default node in **`def_na`**. If **`def`** equals 0, the node is not saved. The function returns 0 if successful, or –1 for invalid **`node`**.

- **`int ad20_rdy_n( int node )`**

  Tests AD7703 DRDY status from RDTTL bit 1 of a specified BL1100 expansion card **`node`**. **`node`** 0–3 specifies the node addresses 0xCxxx to 0xFxxx, respectively. If **`node`** equals –1, the function uses the default node saved in **`def_na`**. The function returns 0 if the AD20 is ready, or –1 if the AD20 is busy or **`node`** is out of range.

- **`int ad20_cal_n( int node, int mode, int def )`**

  Calibrates the AD7703 on a specified BL1100 expansion card. **`node`** 0–3 specifies the node address 0xCxxx to 0xFxxx, respectively. If **`node`** equals –1, the function uses the node saved in **`def_na`**. If **`def`** equals 1, the node is saved as default node in **`def_na`**. If **`def`** equals 0, the node is not saved. Mode 0 calibration does not use the multiplexer. Mode 1 calibration uses the multiplexer to get zero and full scale on Ain. Mux ch0 is the A/D signal to be measured. Mux ch1 is Ain for the Mode 1 first step to calibrate the system offset. Mux ch2 is Ain for Mode 1 second step to calibrate the system gain. Mode 2 calibration uses the current channel to get Ain as zero to calibrate the system offset. The following shows the state of SC1 and SC2 during calibration.

Mode	SC1	SC2	Cal type	Zero	FS Steps
0	0	0	self-cal	AGND	REF+1
1	1	1	system offset	Ain	1st of 2
1	0	1	system gain	Ain	2nd of 2
2	1	0	system offset	Ain	REF+1

  The function returns 0 if calibration was completed, or –1 if there was an error during calibration.

- **`long ad20_rd_n( int node, int def )`**

  Reads 20-bit data from the AD7703 serial data port. The 125-millisecond step response time of AD7703 dictates that a time delay should be guaranteed after a multiplexer switching. A/D data will be valid when DRDY is low for an output data rate up to 4 kHz. The polarity and channel to read should be set previously with **`ad20_mux`**. Ain ranges from 0 to 2.5 volts for the unipolar mode (PA0 = 0). LSB = 2.5 volts/ 1048576 = 2.384 microvolts. Ain ranges from –2.5 to +2.5 volts for the bipolar mode (PA0 = 1). LSB = 5 volts/1048576 = 4.768 microvolts. **`node`** 0–3 specifies the node address 0xCxxx to 0xFxxx, respectively. If **`node`** equals –1, the function uses the node saved in **`def_na`**. If **`def`** equals 1, the node is saved as the default node in **`def_na`**. If **`def`** equals 0, the node is not saved.

  The function returns 20–bit A/D data. For the unipolar mode, 0x00000 = AGND, 0x7FFFF = 1.25 V, and 0xFFFFF = 2.5 V. For the bipolar mode, 0x00000 = –2.5 V, 0x7FFFF = AGND, and 0xFFFFF = 2.5 V. Returns –1 for an invalid node.

KDM.LIB

These KDM (keyboard/display module) functions provide software drivers for KDM keypads, the text LCD, the graphic LCD, the beeper, and the timer that drives the keypad. The beeper also drives the real-time kernel (RTK) when **`RUNKERNEL`** is defined.

- **`int lk_kxinit( void )`**

  Initializes variables, buffers and hardware driver associated with servicing the KDM keypad.

- **`int lk_loadtab( int *tab, int tab_size )`**

  Loads keypad numerical table values. Used to rearrange the keypad keys. **`tab`** points to an integer array containing the new keypad arrangement. **`tab_size`** is the table size to change. For example, **`new-table[] = {4,3,2,1,....}`** will rearrange the ordering of the first four keys.

- **`int lk_settab( char flag )`**

  Sets the keypad translate table for keypad sizes greater than 24.

- **`int lk_keyw( char flag )`**

  Writes to specified bits in the key register.

- **`int lk_kxget( char mode )`**

  Gets character from the KDM keypad. If **`mode`** = 0, removes the character from the keypad buffer and returns it. If **`mode !`** = 0, returns the character (if available), but does not remove it from the keypad buffer. The function returns the keypad character pressed, or –1 if the keypad buffer is empty.

- **`int lk_setbeep( int count )`**

  Sets up the variable that is used for the KDM beeper.

- **`int lk_led( int mode )`**

  Turns LEDs on the KDM on/off without conflicting with the keypad driver. **`mode`** = 0 turns off the LEDs. **`mode`** =1 turns on the yellow LED. **`mode`** = 2 turns on the green LED. **`mode`** = 3 turns on both LEDs. The function returns the mode that was passed.

- **`int lk_tdelay( int delay )`**

  Convenient delay mechanism that is tied to **`timer1`** periodic interrupt.

- **`int lk_int_timer1( void )`**

  Service routine for **`timer1`** interrupt. Drives the beeper and the keypad. Also drives the real-time kernel if **`RUNKERNEL`** is defined.

- **`int lg_init_keypad( void )`**

  Initializes **`timer1`**, KDM keypad driver and the graphic LCD.

- **`int lk_init_keypad( void )`**

  Initializes **`timer1`**, keypad driver and the LCD.

- **`void lk_wr( int x )`**

  Writes low byte of **`x`** to LCD register in the high byte of **`x.`**

- **`int lk_rd( int addr )`**

  Reads data from the LCD read register **`addr`**. The function returns the data from LCD read register **`addr`**.

- **`int lk_init( void )`**

  Initializes LCD on the KDM. Initializes software variables associated with use of the LCD.

- **`int lk_cmd( int cmd )`**

  Sends command in the lower byte of **`cmd`** to the LCD register specified by the upper byte of **`cmd`**.

- **`int lk_wait( void )`**

  Waits for appropriate LCD unit to clear its busy flag. The function returns 0 or 1 depending on the LCD controller.

- **`int lk_char( char x )`**

  Sends one character to data register of the appropriate LCD.

- **`int lk_ctrl( char x )`**

  Sends one character to control register of the appropriate LCD.

- **`int lk_putc( char x )`**

  Low-level driver (**`printf`** analog) for the LCD. Sends a character to the LCD and updates software variables for storing the LCD screen status.

- **`int lk_nl( void )`**

  Generates a new line on the LCD screen.

- **`int lk_pos( int line, int col )`**

  Positions LCD cursor to **`line`** and **`col`** location.

- **`int lk_printf( char *fmt, ... )`**

  This is the **`printf`** analog for the LCD. The following escape sequences are available.

  esc p *n mm* positions cursor to line *n* and column *mm.* Example:

  ```
  lk_printf("\x1bp234");
  ```

  means line 2, column 34. Lines are numbered 0, 1, 2, 3. Columns 0,1,..39.
  esc 1 Turns cursor on
  esc 0 Turn cursor off
  esc c Erases from cursor position to end of line

esc bEnables blinking cursor mode
esc nDisables blinking cursor mode
esc eErases display and homes cursor

- **`void lk_cgram( char *p )`**

  Special character generator for the LCD. **`*p`** (first byte) is the number of bytes to store (up to 64 for 8 characters). The lower five bits of each byte make one row of the character from left to right and from top to bottom. The eighth row of each is in the cursor position.

- **`int lk_stdcg( void )`**

  Loads a table of special characters, **`lk_stdchars`**, to the LCD.

- **`int lk_run_menu( char *call_menu,`**
  `struct lk_menu *menu, int index )`

  Menuing scheme for the KDM unit. The following mtype codes in the menu structures are available.

Code	Description
0	end of menu
1	view floating
2	view floating and adjust in limits
3	view floating and enter new value on enter
4	like 2 but call specified function passing pointer after each step
5	like 3 but call specified function passing pointer to new value
8	view logical
9	view logical and adjust true/false
10	like 9 but call specified function passing pointer to variable
16	view date/time
17	view/modify date/time
18	view/modify date/time and call routine
20	view time (16-bit)
21	view/modify time (16-bit)
22	view/modify time (16-bit) and then call routine
32	call a new menu (**`msg`** is the top line name for new menu, **`valptr`** is the pointer to the new menu structure, the index is always passed as 0)
40	call a function (**`msg`** is displayed, **`ptr`** and **`limit`** are ignored)

The string **`call_menu`** is initially printed when the menu is entered. The pointer **`menu`** points to the **`lk_menu`** structure. The index is the starting point in the menu, often zero. The **`run_menu`** function returns the last value of the index.

- **`void lk_setdate( char *msg, struct tm *dat )`**

  Sets date data and prints to the LCD. Also prints **`msg`** to the LCD. Used by **`lk_run_menu`**.

- **`int lk_chkdat( struct tm *dat )`**

  Checks validity of date data. May change day of the month. The function returns 0 if date data is okay, or 1 for invalid date data.

- **`void lk_showdate( char *msg, struct tm *tmm )`**

  Displays date data and **`msg`** to the LCD.

- **`unsigned int lk_settime( char *msg,`**
  `        unsigned int time )`

  Sets time and prints to the LCD. Also prints **`msg`** to LCD.

- **`int lk_showtime( char *msg, unsigned int time )`**

  Displays **`msg`** and time data on the LCD.

- **`int st_hour( unsigned int j )`**

  Hour parser used by **`lk_run_menu`**. The function returns `j`/1800.

- **`int st_min( unsigned int j )`**

  Minutes parser used by **`lk_run_menu`**. The function returns (`j` mod 1800)/30.

- **`int st_sec( unsigned int j )`**

  Seconds parser used by **`lk_run_menu`**. The function returns 2 × (`j` mod 30).

- **`unsigned int mk_st( int hour, int min, int sec )`**

  Time data builder used by **`lk_run_menu`**. The function returns **`hour`** × 1800 + **`min`** × 30 + **`sec`** × 2.

- **`unsigned int ad_st( unsigned int t1,`**
  `        unsigned int t2 )`

  Time data adder used by **`lk_run_menu`**. The function returns adjusted time data of the two times added together.

- **`int lk_secho( void )`**

  Pulls character from key buffer and generates a short beep.

- **`int lk_lecho( void )`**

  Pulls character from the keypad buffer and generates a long beep.

- **`void lk_viewl( char *fmt, char var )`**

  Views a logical variable.

- **`float lk_getknum( void )`**

  Gets a floating-point number from the keypad. The function returns the floating-point number entered through the keypad.

- **`void lg_init( void )`**

  Initializes the graphic LCD and its associated software variables.

- **`void lg_char( char x )`**

  Writes a character to the graphic LCD.

- **`void lg_putc( char x )`**

  Low-level driver (**`printf`** analog) for the graphic LCD. Puts **`char`** on the graphic LCD and updates software variables that store the graphic LCD screen status.

- **`void lg_nl( void )`**

  Generates a new line on the graphic LCD screen.

- **`void lg_pos( int line, int col )`**

  Positions cursor on the graphic LCD screen.

- **`void lg_printf( char *fmt, ... )`**

  This is the **`printf`** analog for the graphic LCD. The following escape sequences are available.

  esc p *n mm* positions cursor to line *n* and column *mm*. Example:

  ```
  lg_printf("\x1bp234");
  ```

  means line 2, column 34. Lines are numbered 0, 1, 2, 3.
  Columns 0,1,..39.
  esc 1Turns cursor on
  esc 0Turn cursor off
  esc cErases from cursor position to end of line
  esc bEnables blinking cursor mode
  esc nDisables blinking cursor mode
  esc eErases display and homes cursor

- **`void Set_Display_Mode( int mode )`**

  Sets the display mode of the graphic LCD. **`mode`** is **`DISPLAY_TEXT`** (4) or **`DISPLAY_GRAPHICS`** (8).

- **`void Clear_GrTxt_Screen( void )`**

  Clears the graphic LCD text screen.

- **`void Stall( int tix )`**

  Software delay loop. Counts down **`tix`** × 10.

- **`void sta01( void )`**

  Writes 4 to the LCD write register and waits for a 3 on the LCD read register.

- **`void sta03( void )`**

  Writes 4 to the LCD write register and waits for a 0x08 on the LCD read register.

- **`void lg_wr( int x )`**

  Writes data to graphic LCD register. The register value is in the high byte and data value is in the low byte of **`x`**. Uses **`sta01`** to wait for clear to write.

- **`void lg_wr03( int x )`**

  Writes data to graphic LCD register. The register value is in the high byte and data value is in the low byte of **`x`**. Uses **`sta03`** to wait for clear to write.

- **`void lg_rd( void )`**

  Waits for clear and reads the graphic LCD read register.

- **`void grp_home_area( char gal, char gah,`**
  `char ghl, char ghh )`

  Sets the graphic area by defining the home (**`ghl,ghh`**) and the area (**`gal,gah`**).

- **`void text_home_area( char tal, char tah,`**
  `char thl, char thh )`

  Sets the text area by defining the home (**`thl,thh`**) and the area (**`tal,tah`**).

- **`void Graph_Init( void )`**

  Initializes the graphic LCD text and graphics areas.

- **`void Set_Pointer( int address, int ptr )`**

  Sets the appropriate pointer by using the “pointer set” command. **`address`** is the address to set the pointer to. **`ptr`** is the pointer to set: 1 = cursor, 2 = offset, 4 = address.

See page 25 of the ***Toshiba ST-LCD*** manual.

- **`int Text_Addr( int col, int row )`**

  Computes location of text based on the **`row`** and **`col`** data.

  The function returns

  `GRTXT_BASE_ADDRESS + row × LK-COLS + col .`

- **`void Set_Auto_Mode( int mode )`**

  Sets the graphic LCD into auto mode.

- **`void Set_Overlap_Mode( int mode )`**

  Sets the graphic LCD to overlap mode.

- **`void Define_Cursor( int lines )`**

  Defines the cursor for the graphic LCD.

- **`void Set_Pixel( int col, int row, int wr_mode )`**

  Sets an LCD pixel to coordinates (**`col`**, **`row`**). **`wr_mode`** = 0 to clear, **`wr_mode`** = 1 to set, and **`wr_mode`** = 2 to XOR. (0,0) is the lower left corner. **`col`** ranges from 0 to 239; **`row`** ranges from 0 to 63.

- **`void Clear_Gr_Screen( void )`**

  Erases the graphic palette by writing 0s to all addresses in the graphic LCD RAM.

- **`void Map_Bit_Pattern( int *config,`**
  `        char *bitarray, int wr_mode )`

  Maps a bit pattern to the graphic LCD area. **`config`** points to an array of 4-integer data defining the upper left corner (x,y) to start the pattern and the width and height of the figure in dots. **`bitarray`** points to a character data array that has '**`1`**' or '**`*`**' in each location to set a dot in. Data appear in sequential order, starting at the top left corner, progressing left to right and top to bottom. **`wr_mode`** = 0 to clear; **`wr_mode`** = 1 to set and **`wr_mode`** = 2 to XOR.

- **`void Draw_Line( int stx, int sty, int enx,`**
  `        int eny, int wr_mode )`

  Draws a line from starting point (**`stx,sty`**) to end point (**`enx,eny`**). **`wr_mode`** = 0 to clear, **`wr_mode`** = 1 to set and **`wr_mode`** = 2 to XOR.

- **`void Draw_Poly( int numpoints, int *point,`**
  `        int wr_mode )`

  Draws a polygon by connecting successive points. **`numpoints`** is the number of (**`x,y`**) coordinate pairs. **`point`** points to an integer array of (**`x,y`**) coordinate pairs.

- **`void Draw_Axis( int ox, int oy, int ex, int ey, int ticks_x, int ticks_y, int wr_mode )`**

  Draws an axis with (`ox,oy`) as the axis origin. (`ex,ey`) are the highest coordinates of the axis. `ticks_x` is the number of x-axis ticks. `ticks_y` is the number of y-axis ticks.

- **`void Sin_Wave( int ox, int oy, int ex, int ey, int cycles, int wr_mode )`**

  Draws a sine wave with (`ox,oy`) as the sine-wave origin. (`ex,ey`) are the highest possible coordinates of the sine wave. `cycles` is the number of cycles to display.

LCD2L.LIB

These functions support a 2 × 20 LCD on controllers with an LCD port.

- **`void lc_wr( char data )`**

  Low-level routine for writing `char data` to a control register of the LCD. The control register accessed is embedded in `char data`.

- **`int lc_rd( void )`**

  Low-level routine to read the LCD register `LCDWR`. The function returns the busy flag in bit 7 and the address counter of the LCD in the lower seven bits.

- **`void lc_init( void )`**

  Initializes the PK2100 or PK2200 LCD by executing the recommended LCD power-up protocol. Sets LCD for auto increment; display and cursor on; and clears the display memory.

- **`int lc_cmd( int cmd )`**

  Waits for LCD busy flag to clear, then sends `cmd` to the LCD command register. The function returns 0 if successful in writing to the LCD, or –1 if there is a timeout because the LCD is busy.

- **`int lc_wait( void )`**

  Waits for the LCD busy flag to clear. The function returns 0 when the LCD busy flag has cleared, or –1 if it times out after ten tries.

- **`void lc_char( char x )`**

  Writes `char x` to the LCD data register.

- **`void lc_ctrl( char x )`**

  Writes `char x` to the control register of the LCD. Unlike `lc_wr`, this function waits for the busy flag of the LCD to clear before writing data to an LCD control register.

- **`int lc_putc( char x )`**

  Decodes **`char x`** for special command sequence for writing to the LCD command or data registers. This function serves as the driver for **`lc_printf`**.

- **`void lc_nl( void )`**

  Moves the LCD cursor to the first column of the next line. If the current line is the last LCD line, then the cursor position is only moved to column 0 of the current line.

- **`void lc_pos( int line, int col )`**

  Positions PK2100 LCD cursor at the specified **`line`** (0–3) and **`col`** (0–19).

- **`void lc_printf( char *fmt, ... )`**

  This is the **`printf`** analog for the PK2100 LCD.

- **`void lc_cgram( char *p )`**

  Character matrix = 5 rows × 8 cols. **`p`** points to a data array with the following format: first character is the number of bytes to store (8 bytes per character) with a maximum of 64, the lower five bits of each byte form one row of the character from left to right, and the eighth row per special character is in the cursor position.

- **`void lc_stdcg( void )`**

  Loads eight special characters of arrows and lines to the LCD special character location.

- **`void lcd_init_printf( void )`**

  Initializes the LCD with **`lcd_init`**. Also initializes related variables to allow for saving duplicate image of the LCD screen.

- **`void lcd_putc( char x )`**

  Decodes **`char x`** for special command sequence for writing to the LCD command or data registers. Serves as the driver for **`lcd_printf`**. Like **`lc_putc`** except that shadow variables for the LCD are also updated.

- **`void lcd_erase( void )`**

  Erases entire LCD and homes cursor. LCD shadow variables are updated.

- **`void lcd_erase_line( int line )`**

  Erases a specified line on the LCD and updates shadow variables.

- **`void lcd_printf( long cursor, char *fmt, ... )`**

  This is the **`printf`** analog for the LCD screen. Displays a string at a specified starting position and leaves the cursor at a specified end position. **`cursor`** bytes are Y1,X1,Y2,X2, where the most significant byte, Y1, is the start line number (0, 1, 2 or 3); X1 the is start column number (0, 1, 2...), and Y2 and X2 are the final line and column coordinates. The upper four bits of Y2 are used to specify the final state of the cursor (1 = on, 0 = off). Only cursor positioning takes place if **`*fmt`** is a null string.

  When **`lcd_printf`** runs, a semaphore is invoked to ensure that only one execution thread is running through it, so it can be called from various tasks without interference. Execution is suspended for 10 ticks when the semaphore is busy.

  A duplicate copy of the display contents and the cursor location is updated in memory when **`lcd_printf`** prints to the LCD display. The **`lcd_savscrn`** copies this image to a user-specified area. **`lcd_resscrn`** copies the user-saved area back to the screen and the image area. Using these routines, a task can interrupt the current thread and save the current display, use the display in a new thread, and then restore the original display.

- **`void lcd_savscrn( void *s )`**

  Saves LCD screen image to vector identified by **`s`**.

- **`void lcd_resscrn( void *s )`**

  Restores image stored in vector identified by **`s`** to the LCD.

MISC.LIB

- **`void setbeep( int delay )`**

  Sets up a timed beep. **`delay`** specifies the length of the beep in number of **`timer1`** ticks. **`timer1`** interrupt performs the beep in the background, so this function returns immediately.

PBUS_LG.LIB

This library contains the PLCBus support functions for the BL1100 controller and the PLCBus interface library for the BL1100 and the BL1300 controllers. The library contains the functions necessary to access PLCBus devices through PIO Port A on the BL1100. The library also provides low-level PLCBus functions as well as high-level functions for the relay and DAC expansion boards.

The bus must interface to the PIO port as follows.

PIO pin 0: STB	PIO pin 4: D2
PIO pin 1: A3	PIO pin 5: D3
PIO pin 2: A2	PIO pin 6: D0
PIO pin 3: A1	PIO pin 7: D1

- **`void PBus12_Addr( int addr )`**

  Sets the current address for the PLCBus . All read and write operations will access this address until a new address is set. **`addr`** is the 12-bit physical address with the first and third nibbles swapped (most significant nibble in the lower four bits).

- **`void PBus4_Write( char data )`**

  Writes 4-bit data on PLCBus . The address must be set by a call to **`PBus12_Addr`** before calling this function. **`data`** should contain the value to write in the lower four bits.

- **`int PBus4_Read0( void )`**

  Reads 4 bits of data from the PLCBus using a **`BUSRD0`** cycle. The address must be set by a call to **`PBus12_Addr`** before calling this function. The function returns PLCBus data in the lower four bits (the upper bits are undefined).

- **`int PBus4_Read1( void )`**

  Reads 4 bits of data from the PLCBus using a **`BUSRD1`** cycle. The address must be set by a call to **`PBus12_Addr`** before calling this function. The function returns PLCBus data in the lower four bits (the upper bits are undefined).

- **`int PBus4_ReadSp( void )`**

  Reads 4 bits of data from the PLCBus using a **`BUSSPARE`** cycle. The address must be set by a call to **`PBus12_Addr`** before calling this function. The function returns PLCBus data in the lower four bits (the upper bits are undefined).

- **`int Relay_Board_Addr( int board )`**

  Converts a logical relay board address to a physical PLCBus address. **`board`** must be a number between 0 and 63, and represents the relay board to access. This number has the binary form `pppzyx` where **`ppp`** is determined by the board PAL number and **`x`**, **`y`**, and **`z`** are determined by jumper J1 on the board. **`ppp`** values of 000, 001, 010, etc., correspond to PAL numbers of FPO4500, FPO4510, FPO4520, etc.; **`x`**, **`y`**, and **`z`** correspond to jumper J1 pins 1-2, 3-4, and 5-6, respectively (0 = closed, 1 = open). The resulting address is in the form **`pppx`**000**`y`**000**`z`**.

The function returns the PLCBus address of the board specified, with the first and third nibbles swapped; this address may be passed directly to **`PBus12_Addr`**.

- **`void Set_PBus_Relay( int board, int relay, int state )`**

  Sets a relay on an expansion bus relay board. **`board`** must be a number between 0 and 63, and represents the relay board to access. This number has the binary form **`pppzyx`** where **`ppp`** is determined by the board PAL number and **`x`**, **`y`**, and **`z`** are determined by jumper J1 on the board. **`ppp`** values of 000, 001, 010, etc., correspond to PAL numbers of FPO4500, FPO4510, FPO4520, etc.; **`x`**, **`y`**, and **`z`** correspond to jumper J1 pins 1-2, 3-4, and 5-6, respectively (0 = closed, 1 = open). **`relay`** is the relay number on the board (0–5 for XP8300 board; 0–7 for XP8400 board). **`state`** must be 1 to turn the relay on and 0 to turn the relay off.

- **`int DAC_Board_Addr( int bd )`**

  Converts a logical DAC board address to a physical PLCBus address. **`bd`** must be a number between 0 and 63, and represents the DAC board to access. This number has the binary form **`pppzyx`** where **`ppp`** is determined by the board PAL number and **`x`**, **`y`**, and **`z`** are determined by jumper J3 on the board. `ppp` values of 000, 001, 010, etc., correspond to PAL numbers of FPO4800, FPO4810, FPO4820, etc.; **`x`**, **`y`**, and **`z`** correspond to jumper J3 pins 1-2, 3-4, and 5-6, respectively (0 = closed, 1 = open). The resulting address is in the form **`pppx`**001`y`000`z`.

  The function returns the PLCBus address of the board specified, with the first and third nibbles swapped; this address may be passed directly to **`PBus12_Addr`**.

- **`void Write_DAC1( int val )`**

  Loads Register A of DAC #1 with the given 12-bit value. The board address must have been set previously with a call to **`PBus12_Addr`**. The value in **`val`** will not actually be output until **`Latch_DAC1`** is called.

- **`void Write_DAC2( int val )`**

  Loads Register A of DAC #2 with the given 12-bit value. The board address must have been set previously with a call to **`PBus12_Addr`**. The value in **`val`** will not actually be output until **`Latch_DAC2`** is called.

- **`void Latch_DAC1( void )`**

  Moves the value from Register A of DAC 1 to the Register B. The value in Register B represents the actual DAC output. The board address must have been set previously with a call to **`PBus12_Addr`**, and the value should have been loaded into Register A with a call to **`Write_DAC1`**.

- **`void Latch_DAC2( void )`**

  Moves the value from Register A of DAC #2 to Register B. The value in Register B represents the actual DAC output. The board address must have been set previously with a call to **`PBus12_Addr`**, and the value should have been loaded into Register A with a call to **`Write_DAC2`**.

- **`void Init_DAC( void )`**

  Initializes DAC board and sets all output values to 0. Call this function before writing data to the DAC. The board address must have been set previously with a call to **`PBus12_Addr`**.

- **`void Set_DAC1( int val )`**

  Sets DAC #1 to the value specified in the lower 12 bits of **`val`**. In voltage-output mode (J1 pins 2-3 jumpered), V_{OUT} = (**`val`**/4096) × 10.22 volts with Z-World default settings. In current-output mode (J1 pins 1-2 jumpered), I_{OUT} = (**`val`**/4096) × 22 milliamps with Z-World default settings. The board address must have been set previously with a call to **`PBus12_Addr`**.

- **`void Set_DAC2( int val )`**

  Sets DAC #2 to the value specified in the lower 12 bits of **`val`**. In voltage-output mode (J1 pins 2-3 jumpered), V_{OUT} = (**`val`**/4096) × 10.22 V with Z-World default settings. In current-output mode (J1 pins 1–2 jumpered), I_{OUT} = (**`val`**/4096) × 22 mA with Z-World default settings. The board address must have been set previously with a call to **`PBus12_Addr`**.

- **`void DAC_On( void )`**

  Controls the high-side switch activation line. Only used with switch option U10-LT1188.

- **`void DAC_Off( void )`**

  Controls the high-side switch activation line. Only used with switch option U10-LT1188.

- **`void Reset_PBus( void )`**

  Resets the PLCBus.

- **`int Poll_PBus_Node( int addr )`**

  Polls a PLCBus device by performing a **`BUSRD0`** cycle and checking the low bit of the returned value. **`addr`** is the 12-bit physical address of the device, with the first and third nibbles swapped.

  The function returns 1 if **`node`** answers poll, 0 if not.

- **`void Reset_PBus_Wait( void )`**

  Provides the minimum delay necessary for PLCBus expansion boards after a bus reset, assuming a 9 MHz CPU. This delay will be insufficient for a faster CPU and must be increased.

PBUS_TG.LIB

These functions support the BL1000 controller. The PLCBus interface library is provided for the BL1000. This library contains functions necessary to access PLCBus devices through PIO Port B on the BL1000. The library provides low-level PLCBus functions as well as high-level functions for the relay and DAC expansion boards.

The bus must interface to the PIO port as follows.

PIO pin 0: D1	IO pin 4: A1
PIO pin 1: D0	PIO pin 5: A2
PIO pin 2: D3	PIO pin 6: A3
PIO pin 3: D2	PIO pin 7: STB

- **`void PBus12_Addr( int addr )`**

  Sets the current address for the PLCBus. All read and write operations will access this address until a new address is set. **`addr`** is the 12-bit physical address with the first and third nibbles swapped (most significant nibble in the low four bits).

  The function returns None.

- **`void PBus4_Write( char data )`**

  Writes 4-bit data on the PLCBus. The address must be set by a call to **`PBus12_Addr`** before calling this function. **`data`** should contain the value to write in the lower four bits.

- **`int PBus4_Read0( void )`**

  Reads 4 bits of data from the PLCBus using a `BUSRD0` cycle. The address must be set by a call to **`PBus12_Addr`** before calling this function. The function returns the PLCBus data in the lower four bits (the upper bits are undefined).

- **`int PBus4_Read1( void )`**

  Reads 4 bits of data from the PLCBus using a **`BUSRD1`** cycle. The address must be set by a call to **`PBus12_Addr`** before calling this function. The function returns the PLCBus data in the lower four bits (the upper bits are undefined).

- **`int PBus4_ReadSp( void )`**

  Reads 4 bits of data from the PLCBus using a **`BUSSPARE`** cycle. The address must be set by a call to **`PBus12_Addr`** before calling this function. The function returns the PLCBus data in the lower four bits (the upper bits are undefined).

- **`int Relay_Board_Addr( int board )`**

  Converts a logical relay board address to a physical PLCBus address. **`board`** must be a number between 0 and 63, and represents the relay board to access. This number has the binary form **`pppzyx`** where **`ppp`** is determined by the board PAL number and **`x`**, **`y`**, and **`z`** are determined by jumper J1 on the board. **`ppp`** values of 000, 001, 010, etc., correspond to PAL numbers of FPO4500, FPO4510, FPO4520, etc.; **`x`**, **`y`**, and **`z`** correspond to jumper J1 pins 1-2, 3-4, and 5-6, respectively (0 = closed, 1 = open). The resulting address is in the form **`pppx`**000**`y`**000**`z`**.

  The function returns the PLCBus address of the board specified, with the first and third nibbles swapped; this address may be passed directly to **`PBus12_Addr`**.

- **`void Set_PBus_Relay( int board,int relay,`**
  `int state )`

  Sets a relay on an expansion bus relay board. **`board`** must be a number between 0 and 63, and represents the relay board to access. This number has the binary form **`pppzyx`**, where **`ppp`** is determined by the board PAL number and **`x`**, **`y`**, and **`z`** are determined by jumper J1 on the board. **`ppp`** values of 000, 001, 010, etc., correspond to PAL numbers of FPO4500, FPO4510, FPO4520, etc.; **`x`**, **`y`**, and **`z`** correspond to jumper J1 pins 1-2, 3-4, and 5-6, respectively (0 = closed, 1 = open). **`relay`** is the relay number on the board (0–5 for XP8300 board; 0–7 for XP8400 board). **`state`** must be 1 to turn the relay on and 0 to turn the relay off.

- **`int DAC_Board_Addr( int bd )`**

  Converts a logical DAC board address to a physical PLCBus address. **`bd`** must be a number between 0 and 63, and represents the DAC board to access. This number has the binary form **`pppzyx`** where **`ppp`** is determined by the board PAL number and **`x`**, **`y`**, and **`z`** are determined by jumper J3 on the board. **`ppp`** values of 000, 001, 010, etc., correspond to PAL numbers of FPO4800, FPO4810, FPO4820, etc.; **`x`**, **`y`**, and **`z`** correspond to jumper J3 pins 1-2, 3-4, and 5-6, respectively (0 = closed, 1 = open). The resulting address is in the form **`pppx`**001**`y`**000**`z`**.

  The function returns the PLCBus address of the board specified, with the first and third nibbles swapped; this address may be passed directly to **`PBus12_Addr`**.

- **`void Write_DAC1( int val )`**

  Loads Register A of DAC #1 with the given 12-bit value. The board address must have been set previously with a call to **`PBus12_Addr`**. The value in **`val`** will not actually be output until **`Latch_DAC1`** is called.

- **`void Write_DAC2( int val )`**

  Loads Register A of DAC #2 with the given 12-bit value. The board address must have been set previously with a call to **`PBus12_Addr`**. The value in **`val`** will not actually be output until **`Latch_DAC2`** is called.

- **`void Latch_DAC1( void )`**

  Moves the value in Register A of DAC #1 to Register B. The value in Register B represents the actual DAC output. The board address must have been set previously with a call to **`PBus12_Addr`**, and the value should have been loaded into Register A with a call to **`Write_DAC1`**.

- **`void Latch_DAC2( void )`**

  Moves the value in Register A of DAC #2 to Register B. The value in Register B represents the actual DAC output. The board address must have been set previously with a call to **`PBus12_Addr`**, and the value should have been loaded into Register A with a call to **`Write_DAC2`**.

- **`void Init_DAC( void )`**

  Initializes DAC board and sets all output values to 0. Call this function before writing data to the DAC. The board address must have been set previously with a call to **`PBus12_Addr`**.

- **`void Set_DAC1( int val )`**

  Sets DAC #1 to the value specified in the lower 12 bits of **`val`**. In voltage-output mode (J1 pins 2-3 jumpered), V_{OUT} = (**`val`**/4096) × 10.22 volts with Z-World default settings. In current-output mode (J1 pins 1-2 jumpered), I_{OUT} = (**`val`**/4096) × 22 milliamps with Z-World default settings. The board address must have been set previously with a call to **`PBus12_Addr`**.

- **`void Set_DAC2( int val )`**

  Sets DAC #2 to the value specified in the lower 12 bits of **`val`**. In voltage-output mode (J1 pins 2–3 jumpered), V_{OUT} = (**`val`**/4096) × 10.22 V with Z-World default settings. In current-output mode (J1 pins 1–2 jumpered), I_{OUT} = (**`val`**/4096) × 22 mA with Z-World default settings. The board address must have been set previously with a call to **`PBus12_Addr`**.

- **`void DAC_On( void )`**

  Controls the high-side switch activation line. Only used with switch option U10-LT1188.

- **`void DAC_Off( void )`**

  Controls the high-side switch activation line. Only used with switch option U10-LT1188.

- **`void Reset_PBus( void )`**

  Resets the PLCBus.

- **`int Poll_PBus_Node( int addr )`**

  Polls a PLCBus device by performing a **`BUSRD0`** cycle and checking the low bit of the returned value. **`addr`** is the 12-bit physical address of the device, with the first and third nibbles swapped. The function returns 1 if **`node`** answers poll, 0 if not.

- **`void Reset_PBus_Wait( void )`**

  Provides the minimum delay necessary for PLCBus expansion boards after a bus reset, assuming a 9 MHz CPU. This delay will be insufficient for a faster CPU and must be increased.

Appendix A: Dynamic C Libraries

The libraries provided with Dynamic C 32 are listed here, each with a brief description of its purpose. For detailed information on the requirements for library code, of particular interest to those who write their own custom libraries, see the "Software Libraries" appendix in the **Dynamic C 32** ***Version 6.x*** **Technical Reference** manual.

LIB

These are the mainstream libraries, found in the **`LIB`** subfolder of the main Dynamic C 32 installation folder.

5KEY.LIB

For PK2100 series and PK2200 series controllers. Basic support for the original "five-key system."

5KEYEXTD.LIB

For PK2100 series and PK2200 series controllers. Extensions to the original "five-key system."

96IO.LIB

For BL1000 series controllers. Driver functions for the DGL96 daughter board.

AASC.LIB

Abstract Application-Level Serial Communication (AASC) high level functions, this library is automatically **`#use`**d by the low level **`AASC*.LIB`** libraries. These high level functions are intended to be called from the application.

AASCDIO.LIB

STDIO window functions supporting the **`AASC.LIB`** library, primarily useful while debugging an application. Although the application should **`#use`** this library if it is required, these low level functions are not intended to be called from the application.

AASCDUM.LIB

Dummy device functions supporting the **`AASC.LIB`** library. Although the application should **`#use`** this library if it is required, these low level functions are not intended to be called from the application.

AASCSCC.LIB

For BL1300 series, BL1700 series and PK2600 (BL1700 side only) series controllers. Zilog 85C30 Serial Communication Controller (SCC) functions supporting the **`AASC.LIB`** library. Although the application should **`#use`** this library if it is required, these low level functions are not intended to be called from the application.

AASCSIOA.LIB

For BL1100 series controllers. Zilog 84C90 KIO Serial/Parallel/Counter/Timer SIOA functions supporting the **`AASC.LIB`** library. Although the application should **`#use`** this library if it is required, these low level functions are not intended to be called from the application.

AASCUART.LIB

For BL1200 series, BL1600 series, PK2100 series and PK2200 series controllers. XP8700 UART PLCBus expansion board functions supporting the **`AASC.LIB`** library. Although the application should **`#use`** this library if it is required, these low level functions are not intended to be called from the application.

AASCURT2.LIB

For BL1700 series and PK2600 (BL1700 side only) series controllers. XP8700 UART PLCBus expansion board functions supporting the **`AASC.LIB`** library. Although the application should **`#use`** this library if it is required, these low level functions are not intended to be called from the application.

AASCZ0.LIB

Z180 built-in Z0 (ASCI0) functions supporting the **`AASC.LIB`** library. Although the application should **`#use`** this library if it is required, these low level functions are not intended to be called from the application.

AASCZ1.LIB

Z180 built-in Z1 (ASCI1) functions supporting the **`AASC.LIB`** library. Although the application should **`#use`** this library if it is required, these low level functions are not intended to be called from the application.

AASCZN.LIB

Z180 built-in Z1 (ASCI1) ZNet functions supporting the **`AASC.LIB`** library. Although the application should **`#use`** this library if it is required, these low level functions are not intended to be called from the application.

BIOS.LIB

Prototypes of functions and declarations of variables that are defined in the BIOS, and which are available for use by the application program.

BL11XX.LIB

For BL1100 series controllers. ADC, DAC, I/O and timer functions.

BL13XX.LIB

For BL1300 series controllers. Serial Communication Controller (SCC) functions.

BL14_15.LIB

For BL1400 series and BL1500 series controllers. Timer, RTC, RTC RAM, PIO, simulated PLCBus, character LCD and keypad functions.

BL16XX.LIB

For BL1600 series controllers. Digital I/O and virtual I/O functions.

CIRCBUF.LIB

Circular buffer functions supporting the **`AASC.LIB`** library.

CM71_72.LIB

For CM7100 series and CM7200 series core modules. Keypad and beeper functions.

COM232.LIB

For Z104/ZISA series controllers. COM1 and COM2 communication functions.

COSTATE.LIB

Declarations, definitions and functions supporting costatements, a form of cooperative multitasking unique to Dynamic C.

CPLC.LIB

For BL1600 series, PK2100 series, PK2200 series and Z104/ZISA series controllers. Timer, character LCD, keypad, virtual driver and beeper functions.

CTYPE.LIB

Character case conversion and type classification functions.

DC.HH

This file contains definitions basic to, and required by, Dynamic C 32. Even though it is not strictly a library, it is included in the **`lib.dir`** file's library list.

DEFAULT.H

Contains lists of **`#use`** directives for various Z-World Z180-based controllers. Based on the controller's **`BOARD_TYPE`**, Dynamic C 32 automatically selects the list appropriate for controller being programmed. Even though it is not strictly a library, it is included in the **`lib.dir`** file's library list.

DMA.LIB

Functions supporting the Z180 built-in direct memory access (DMA) channels.

DRIVERS.LIB

PLCBus, high voltage output, character LCD, interrupt, time, RTC, logical to physical address conversion, DMA memory copy, port output, PRT, interrupt vector, EEPROM, DMA counter, flash EPROM write and RS-485 driver enable/disable functions.

EPSONRTC.LIB

Epson 72421 RTC existence check function.

FK.LIB

For PK2100 series and PK2200 series controllers. New "five-key system" support to be used with costatements (Dynamic C's cooperative multitasking).

GATE_P.LIB

For PK2100 series and PK2200 series controllers. Functions for gate programming of ladder logic using the "five-key system."

GESUPRT.LIB

Support functions for interfacing any controller to Z-World's Graphics Engine sample program running on an OP7100 series or PK2600 (OP7100 side only) series controller. The Graphics Engine provides a flexible and on-the-fly customizable operator interface via its controller's RS-232 communication port.

GLCD.LIB

For OP7100 series (when **`#use`**d with the obsolete **`LQVGA.LIB`** or **`PQVGA.LIB`** libraries), PK2240, PK2400 and PK2600 (OP7100 side only, when **`#use`**d with the obsolete **`LQVGA.LIB`** or **`PQVGA.LIB`** libraries) series controllers. High level (abstracted) graphic LCD primitive functions.

IOEXPAND.LIB

For BL1100 series controllers. Expansion board functions supporting both default and non-default board addresses.

KDI.LIB

For BL1100 series, BL1200 series, BL1400 series, BL1500 series, BL1600 series, BL1700 series, CM7100 series, CM7200 series, PK2100 series, PK2200 series, PK2400 series and PK2600 (BL1700 side only) series controllers. High level (abstracted) keypad/display interface functions.

KDM.LIB

For BL1200 series, BL1600 series, BL1700 series, PK2100 series, PK2200 series and PK2600 (BL1700 side only) series controllers. Driver functions for Z-World OP6000 series, OP6100, OP6200 and OP6300 keypad/display modules.

KP.LIB

For BL1500 series, LP3100 series, OP7100 series, PK2200 series, PK2400 series and PK2600 (OP7100 side only) series controllers. High level (abstracted) keypad interface functions.

KP_KDI.LIB

For BL1500 series, PK2200 series and PK2400 series controllers. Low level (hardware specific) keypad interface functions supporting the **`KP.LIB`** library.

KP_LP31.LIB

For LP3100 series controllers. Low level (hardware specific) keypad interface functions supporting the **`KP.LIB`** library.

KP_OP71.LIB

For OP7100 series and PK2600 (OP7100 side only) series controllers. Low level (hardware specific) touch-screen keypad interface functions supporting the **`KP.LIB`** library.

LCD2L.LIB

For BL1200 series, BL1600 series, BL1700 series, CM7100 series, CM7200 series, PK2100 series, PK2200 series, PK2600 (BL1700 side only) series and Z104/ZISA series controllers. Two line character LCD support functions.

LP.LIB

For BL1100 series, BL1200 series, BL1400 series, BL1500 series, BL1600 series, BL1700 series, CM7100 series, CM7200 series, PK2100 series, PK2200 series, PK2400 series and PK2600 (BL1700 side only) series controllers. Switch library uses controller's **`BOARD_TYPE`** definition to select the appropriate **`LP_*.LIB`** function library. Also contains function help headers.

LP_16.LIB

For BL1100 series, BL1200 series, BL1600 series and PK2100 series controllers. Nonnative (16-bit addressable port) character LCD interface functions, **`#use`**d by the **`LP.LIB`** library.

LP_8.LIB

For BL1700 series, CM7100 series, CM7200 series, PK2200 series and PK2600 (BL1700 side only) series controllers. Native (8-bit addressable port) character LCD interface functions, **`#use`**d by the **`LP.LIB`** library.

LP_BL145.LIB

For BL1400 series and BL1500 series controllers. Shared I/O character LCD port interface functions, **`#use`**d by the **`LP.LIB`** library.

MATH.LIB

Mathematical and trigonometric functions.

MISC.LIB

For BL1000 series and BL1100 series controllers. Miscellaneous driver functions for Z-World OP6000 series, OP6100, OP6200 and OP6300 keypad/display modules.

MM.LIB

High level (abstracted) MODBus network master device functions.

MMZ.LIB

Low level (hardware specific) MODBus network master device Z0 (ASCI0) and Z1 (ASCI1) serial I/O functions supporting the **`MM.LIB`** library.

MODEM232.LIB

Modem functions supporting the **`COM232.LIB`**, **`NETWORK.LIB`**, **`S0232.LIB`**, **`S1232.LIB`**, **`SCC232.LIB`**, **`UART232.LIB`**, **`UART2.LIB`**, **`UART3.LIB`**, **`Z0232.LIB`** and **`Z1232.LIB`** libraries.

MS.LIB

High level (abstracted) MODBus network slave device functions.

MSZ.LIB

Low level (hardware specific) MODBus network slave device Z0 (ASCI0) and Z1 (ASCI1) serial I/O functions supporting the **`MS.LIB`** library.

NETWORK.LIB

Opto22 9th bit binary protocol master-slave networking via the Z180's built-in Z1 (ASCI1) serial port.

OP71HW.LIB

For OP7100 series and PK2600 (OP7100 side only) series controllers. Generic (i.e. both landscape and portrait mode) graphic and hardware (backlight, contrast, I/O) functions. The application must **`#use`** the appropriate one of the **`OP71L.LIB`** or **`OP71P.LIB`** libraries as well as any font libraries that are required.

OP71L.LIB

For OP7100 series and PK2600 (OP7100 side only) series controllers. Landscape mode LCD buffer transfer functions supporting the **`OP71HW.LIB`** library.

OP71P.LIB

For OP7100 series and PK2600 (OP7100 side only) series controllers. Portrait mode LCD buffer transfer function supporting the **`OP71HW.LIB`** library.

PBUS_LG.LIB

For BL1100 series controllers. Simulated PLCBus functions.

PBUS_TG.LIB

For BL1000 series controllers. Simulated PLCBus functions.

PK21XX.LIB

For PK2100 series controllers. Digital I/O, virtual I/O, DAC, ADC, universal I/O, high gain input, beeper and keypad functions.

PK22XX.LIB

For PK2200 series controllers. Digital I/O, virtual I/O, beeper and keypad functions.

PLC_EXP.LIB

For BL1200 series, BL1600 series, PK2100 series and PK2200 series controllers. Native (8-bit addressable port) PLCBus functions.

PRPORT.LIB

For BL1000 series, BL1100 series and BL1300 series controllers. Functions that implement a parallel port communication protocol between the Z-World controller and a PC compatible printer or printer port.

PWM.LIB

Pulse width modulation functions. Uses Z180's built-in timer 0.

RTK.LIB

Real time kernel (RTK) preemptive multitasking functions.

S0232.LIB

For BL1100 series controllers. Serial communication driver for the KIO's SIO port 0 (SIOA).

S1232.LIB

For BL1100 series controllers. Serial communication driver for the KIO's SIO port 1 (SIOB).

SCC232.LIB

For BL1300 series controllers. Serial communication driver for the Zilog 85C30 Serial Communication Controller (SCC).

SERIAL.LIB

Z180 built-in Z0 (ASCI0), Z1 (ASCI1) as well as Z80-SIO (Serial I/O) SIOA and SIOB serial communication functions.

SF1000_Z.LIB

For BL1100 series, BL1400 series, BL1500 series, BL1600 series, BL1700 series, LP3100 series, PK2200 series and PK2600 (BL1700 side only) series controllers. Driver functions for Z-World's SF1000 series of serial flash EPROM memory boards.

SRTK.LIB

Simplified real time kernel (SRTK) preemptive multitasking functions.

STDIO.LIB

"String" formatting functions, character and "string" I/O functions for Dynamic C 32's STDIO window.

STEP.LIB

For BL1200 series, BL1600 series, PK2100 series and PK2200 series controllers. Native (8-bit addressable port) PLCBus XP8800 expansion board stepper motor control functions.

STEP2.LIB

For BL1700 series and PK2600 (BL1700 side only) series controllers. Nonnative (16-bit addressable port) PLCBus XP8800 expansion board stepper motor control functions.

STRING.LIB

"String" manipulation functions.

SYS.LIB

General system functions.

TGIANT.LIB

For BL1000 series controllers only. ADC, power fail functions.

THERMADC.LIB

For BL1200 series, BL1600 series, PK2100 series, PK2200 series, PK2300 series, PK2400 series and Z104/ZISA series controllers. Thermistor via ADC temperature measurement functions.

TIO.LIB

Device independent terminal I/O functions via the **`AASC.LIB`** library.

TL.LIB

For LP3100 series controllers. High level (abstracted) character (text) LCD functions.

TL_LP31.LIB

For LP3100 series controllers. Low level (hardware specific) character (text) LCD functions.

TOSHRTC.LIB

Toshiba 8250 RTC existence check function.

UART2.LIB

For BL1200 series, BL1600 series, BL1700 series, PK2100 series, PK2200 series and PK2600 (BL1700 side only) series controllers. Functions supporting an XP8700 UART PLCBus expansion board addressed at 0x040010.

UART232.LIB

For BL1200 series, BL1600 series, BL1700 series, PK2100 series, PK2200 series and PK2600 (BL1700 side only) series controllers. Functions supporting an XP8700 UART PLCBus expansion board addressed at 0x040018.

UART3.LIB

For BL1200 series, BL1600 series, BL1700 series, PK2100 series, PK2200 series and PK2600 (BL1700 side only) series controllers. Functions supporting an XP8700 UART PLCBus expansion board addressed at 0x040008.

UIBOARD.LIB

For BL1200 series, BL1600 series, PK2100 series, PK2200 series and Z104/ZISA series controllers. XP8200 universal digital I/O, virtual I/O, analog comparator inputs and high current digital outputs PLCBus expansion board functions.

UTIL.LIB

Low level support (utility) functions.

V256X64.LIB

For BL1100 series, BL1200 series, BL1400 series, BL1500 series, BL1600 series, BL1700 series, CM7100 series, CM7200 series, PK2100 series, PK2200 series, PK2400 series and PK2600 (BL1700 side only) series controllers. Low level (hardware specific) Noretaki graphic VFD primitive functions supporting the **`GLCD.LIB`** library. Root data and **`xdata`** versions of 6 pixels wide by 8 pixels high fonts, ASCII characters 0x20 through 0x7E inclusive.

VDRIVER.LIB

Virtual driver, fastcall, timer and delay functions supporting the **`RTK.LIB`** and **`SRTK.LIB`** libraries.

VWDOG.LIB

Virtual watchdog functions supporting the **`VDRIVER.LIB`** library.

WINTEK.LIB

For PK2240 controllers. Low level (hardware specific) graphic LCD primitive functions supporting the **`GLCD.LIB`** library. Root data and **`xdata`** versions of 6 pixels wide by 8 pixels high fonts, ASCII characters 0x20 through 0xFF inclusive.

XMEM.LIB

Logical (root)/physical memory information transfer, address translation and conversion functions related to extended memory.

Z0232.LIB

Z180 built-in Z0 (ASCI0) serial communication driver functions.

Z104.LIB

For Z104/ZISA series controllers. VGA initialization, control, graphics and text functions. Parallel printer, virtual I/O and PC104 memory and I/O port access functions. Power fail and beeper functions.

Z1232.LIB

Z180 built-in Z1 (ASCI1) serial communication driver functions.

ZNPAKFMT.LIB

Low level ZNet functions supporting the **`AASCZN.LIB`** library.

LIB\DEMO

These libraries, found in the **`LIB\DEMO`** subfolder of the main Dynamic C 32 installation folder, are required to compile certain factory installed demonstration programs.

QVGADEMO.LIB

Bit mapped fonts, graphics and functions used in the **`SAMPLES\OP71XX`** subfolder's **`op71_demo.c`** and **`pk26_gedemo.c`** sample programs. These are the factory installed demonstration programs for OP7100 series and PK2600 (OP7100 side only) series controllers, respectively.

ZWLOGOS.LIB

Several sizes of Z-World's bit mapped logo.

LIB\EASYSTRT

These libraries were, or are based upon libraries which were, originally delivered with Z-World's EasyStart controller kits. Found in the **`LIB\EASYSTRT`** subfolder of the main Dynamic C 32 installation folder, they contain some unique functionality and are slated for reorganization and integration into the mainstream libraries.

EZIO.LIB

Board-independent unified I/O space driver functions.

EZIOBL17.LIB

For BL1700 series and PK2600 (BL1700 side only) series controllers. Low level (hardware specific) functions supporting the **`EZIO.LIB`** library.

EZIOCMMN.LIB

Common definitions, high level (abstracted) ADC and board initialization functions.

EZIODPWM.LIB

For BL1700 series, PK2300 series, PK2500 series and PK2600 (BL1700 side only) series controllers. DMA-driven pulse width modulation (PWM) functions.

EZIOLGPL.LIB

For BL1100 series controllers. Simulated PLCBus functions.

EZIOLP31.LIB

For LP3100 series controllers. Low level (hardware specific) functions supporting the **`EZIO.LIB`** library.

EZIOMGPL.LIB

For BL1400 series and BL1500 series controllers. Simulated PLCBus functions.

EZIOOP71.LIB

For OP7100 series and PK2600 (OP7100 side only) series controllers. Low level (hardware specific) functions supporting the **`EZIO.LIB`** library.

EZIOPBDV.LIB

Low level (hardware specific) PLCBus expansion board device drivers supporting the **`EZIO.LIB`** library.

EZIOPK23.LIB

For PK2300 series controllers. Low level (hardware specific) functions supporting the **`EZIO.LIB`** library.

EZIOPK24.LIB

For PK2400 series controllers. Low level (hardware specific) functions supporting the **`EZIO.LIB`** library.

EZIOPK25.LIB

For PK2500 series controllers. Low level (hardware specific) functions supporting the **`EZIO.LIB`** library.

EZIOPLC.LIB

For BL1200 series, BL1600 series, PK2100 series and PK2200 series controllers. Native (8-bit addressable port) PLCBus functions.

EZIOPLC2.LIB

For BL1700 series and PK2600 (BL1700 side only) series controllers. Nonnative (16-bit addressable port) PLCBus functions.

EZIOPPLC.LIB

For BL1000 series, BL1100 series, BL1400 series and BL1500 series controllers. Simulated PLCBus functions.

EZIOTGPL.LIB

For BL1000 series controllers. Simulated PLCBus functions.

ZIO.LIB

Function help descriptions for ZIO routines.

ZIO1.LIB

Common call interface to ZIO local I/O.

ZIO1DB.LIB

Common call interface to ZIO local debounced I/O.

ZIO1L.LIB

Common call interface to ZIO local debounced I/O and long range networked remote I/O.

ZIO1S.LIB

Common call interface to ZIO local debounced I/O and short range networked remote I/O.

ZIO2.LIB

Common call interface to ZIO local I/O and PLCBus devices.

ZIO2DB.LIB

Common call interface to ZIO local debounced I/O and PLCBus devices.

ZIO3L.LIB

Common call interface to ZIO long range networked remote I/O.

ZIO3S.LIB

Common call interface to ZIO short range networked remote I/O.

ZIONET.LIB

High level (abstracted) ZIO Z180 built-in Z1 (ASCI1) network functions.

LIB\FONT

These are bit mapped font libraries, found in the **LIB\FONT** subfolder of the main Dynamic C 32 installation folder. They may be **#use**d by applications for controllers with a graphic LCD or by applications which use an OP7100 series or PK2600 series (OP7100 side only) controller, running the Graphics Engine sample program, as an operator interface. Note that fonts displayed in applications which **`#use OP71HW.LIB`** and either **`OP71L.LIB`** or **`OP71P.LIB`** (including the Graphics Engine) should always be landscape mode.

12X16L.LIB

Landscape mode **`xdata`** font. 12 pixels wide by 16 pixels high, ASCII characters 0x20 through 0x7F inclusive.

16X20L.LIB

Landscape mode **`xdata`** font. 16 pixels wide by 20 pixels high, ASCII characters 0x20 through 0x7F inclusive.

8X10L.LIB

Landscape mode **`xdata`** font. 8 pixels wide by 10 pixels high, ASCII characters 0x20 through 0x7F inclusive.

ENGFNT2L.LIB

Landscape mode **`xdata`** fonts. 16 pixels wide by 32 pixels high, ASCII characters 0x20 through 0x7F inclusive. 17 pixels wide by 35 pixels high, ASCII characters 0x20 through 0x7F inclusive.

ENGFONT.LIB

Portrait mode fonts. Root data and **`xdata`** versions of 6 pixels wide by 8 pixels high, ASCII characters 0x20 through 0xFF inclusive. Root data and **`xdata`** versions of 17 pixels wide by 35 pixels high, ASCII characters 0x20 through 0x7F inclusive.

ENGFONT2.LIB

Portrait mode **`xdata`** fonts. 16 pixels wide by 32 pixels high, ASCII characters 0x20 through 0x7F inclusive. 17 pixels wide by 35 pixels high, ASCII characters 0x20 through 0x7F inclusive.

ENGFONTL.LIB

Landscape mode fonts. Root data and **`xdata`** versions of 6 pixels wide by 8 pixels high, ASCII characters 0x20 through 0xFF inclusive. Root data and **`xdata`** versions of 17 pixels wide by 35 pixels high, ASCII characters 0x20 through 0x7F inclusive.

LIB\OBSOLETE

These are deprecated libraries, deemed obsolete by Z-World, found in the **`LIB\OBSOLETE`** subfolder of the main Dynamic C 32 installation folder. Most of these obsolete libraries' functionality has been completely replaced by, or moved into, the mainstream libraries. Where an obsolete library still has content, those functions are retained to maintain backwards compatibility with legacy applications. The use, in new applications, of any functions found in these deprecated libraries is strongly discouraged.

IOE.LIB

For BL1100 series controllers. Empty library, replaced by the **`IOEXPAND.LIB`** library.

LGIANT.LIB

For BL1100 series controllers. Empty library, replaced by the **`BL11XX.LIB`** library.

LITTLEG.LIB

For BL1600 series controllers. Empty library, replaced by the **`BL16XX.LIB`** library.

LQVGA.LIB

For OP7100 series and PK2600 (OP7100 side only) series controllers. Low level (hardware specific) landscape mode graphic LCD primitive functions supporting the **`GLCD.LIB`** library.

LSTAR.LIB

For PK2200 series controllers. Empty library, replaced by the **`PK22XX.LIB`** library.

MICROG.LIB

For BL1400 series and BL1500 series controllers. Empty library, replaced by the **`BL14_15.LIB`** library.

PQVGA.LIB

For OP7100 series and PK2600 (OP7100 side only) series controllers. Low level (hardware specific) portrait mode graphic LCD primitive functions supporting the **`GLCD.LIB`** library.

PS.LIB

For BL1300 series controllers. Empty library, replaced by the **`BL13XX.LIB`** library.

RG.LIB

For PK2100 series controllers. Empty library, replaced by the **`PK21XX.LIB`** library.

SCOREZ1.LIB

For CM7100 series and CM7200 core modules. Empty library, replaced by the **`CM71_72.LIB`** library.

APPENDIX B: LIBRARY LISTS FOR Z-WORLD PRODUCTS

The libraries included with Dynamic C 32 are listed here according to the Z180-based controller(s) they are intended for use with.

ALL

AASC.LIB, **AASCDIO.LIB**, **AASCDUM.LIB**, **AASCZ0.LIB**, **AASCZ1.LIB**, **AASCZN.LIB**, **BIOS.LIB**, **CIRCBUF.LIB**, **COSTATE.LIB**, **CTYPE.LIB**, **DC.HH**, **DEFAULT.H**, **DMA.LIB**, **DRIVERS.LIB**, **EPSONRTC.LIB**, **EZIO.LIB**, **EZIOPBDV.LIB**, **GESUPRT.LIB**, **MATH.LIB**, **MM.LIB**, **MMZ.LIB**, **MS.LIB**, **MSZ.LIB**, **MODEM232.LIB**, **NETWORK.LIB**, **PWM.LIB**, **RTK.LIB**, **SERIAL.LIB**, **SRTK.LIB**, **STDIO.LIB**, **STRING.LIB**, **SYS.LIB**, **TIO.LIB**, **TOSHRTC.LIB**, **UTIL.LIB**, **VDRIVER.LIB**, **VWDOG.LIB**, **XMEM.LIB**, **Z0232.LIB**, **Z1232.LIB**, **ZIO.LIB**, **ZIO1.LIB**, **ZIO1DB.LIB**, **ZIO1L.LIB**, **ZIO1S.LIB**, **ZIO2.LIB**, **ZIO2DB.LIB**, **ZIO3L.LIB**, **ZIO3S.LIB**, **ZIONET.LIB**, **ZNPAKFMT.LIB**

BL1000

EZIOPPLC.LIB, **EZIOTGPL.LIB**, **MISC.LIB**, **PBUS_TG.LIB**, **PRPORT.LIB**, **TGIANT.LIB**

BL1100

96IO.LIB, **AASCSIOA.LIB**, **BL11XX.LIB**, **EZIOLGPL.LIB**, **EZIOPPLC.LIB**, **IOEXPAND.LIB**, **KDI.LIB**, **LP.LIB**, **LP_8.LIB**, **MISC.LIB**, **PBUS_LG.LIB**, **PRPORT.LIB**, **S0232.LIB**, **S1232.LIB**, **SF1000_Z.LIB**

BL1200

AASCUART.LIB, **EZIOCMMN.LIB**, **EZIOPLC.LIB**, **KDI.LIB**, **KDM.LIB**, **LCD2L.LIB, LP.LIB**, **LP_8.LIB**, **PLC_EXP.LIB**, **STEP.LIB**, **THERMADC.LIB**, **UART2.LIB**, **UART232.LIB**, **UART3.LIB**, **UIBOARD.LIB**

BL1300

AASCSCC.LIB, **BL13XX.LIB**, **PRPORT.LIB**, **SCC232.LIB**

BL1400

BL14_15.LIB, **EZIOMGPL.LIB**, **EZIOPPLC.LIB**, **KDI.LIB**, **LP.LIB**, **LP_BL145.LIB**, **SF1000_Z.LIB**

BL1500

BL14_15.LIB, **EZIOMGPL.LIB**, **EZIOPPLC.LIB**, **KDI.LIB**, **KP.LIB**, **KP_KDI.LIB**, **LP.LIB**, **LP_BL145.LIB**, **SF1000_Z.LIB**

BL1600

BL16XX.LIB, **CPLC.LIB**, **EZIOCMMN.LIB**, **EZIOPLC.LIB**, **KDI.LIB**, **KDM.LIB**, **LCD2L.LIB**, **LP.LIB**, **LP_8.LIB**, **PLC_EXP.LIB**, **SF1000_Z.LIB**, **STEP.LIB**, **THERMADC.LIB**, **UART2.LIB**, **UART232.LIB**, **UART3.LIB**, **UIBOARD.LIB**

BL1700

AASCSCC.LIB, **AASCURT2.LIB**, **EZIOBL17.LIB**, **EZIOCMMN.LIB**, **EZIODPWM.LIB**, **EZIOPLC2.LIB**, **KDI.LIB**, **KDM.LIB**, **LCD2L.LIB**, **LP.LIB**, **LP_16.LIB**, **SF1000_Z.LIB**, **STEP2.LIB**, **UART2.LIB**, **UART232.LIB**, **UART3.LIB**

CM7100

CM71_72.LIB, **KDI.LIB**, **LCD2L.LIB**, **LP.LIB**, **LP_16.LIB**

CM7200

CM71_72.LIB, **KDI.LIB**, **LCD2L.LIB**, **LP.LIB**, **LP_16.LIB**

LP3100

EZIOCMMN.LIB, **EZIOLP31.LIB**, **KP.LIB**, **KP_LP31.LIB**, **SF1000_Z.LIB**, **TL.LIB**, **TL_LP31.LIB**

OP7100

12X16L.LIB, **16X20L.LIB**, **8X10L.LIB**, **ENGFNT2L.LIB**, **ENGFONT.LIB**, **ENGFONT2.LIB**, **ENGFONTL.LIB**, **EZIOOP71.LIB**, **KP.LIB**, **KP_OP71.LIB**, **OP71HW.LIB**, **OP71L.LIB**, **OP71P.LIB**, **QVGADEMO.LIB**, **ZWLOGOS.LIB**

PK2100

5KEY.LIB, **5KEYEXTD.LIB**, **AASCUART.LIB**, **CPLC.LIB**, **EZIOCMMN.LIB**, **EZIOPLC.LIB**, **FK.LIB**, **GATE_P.LIB**, **KDI.LIB**, **KDM.LIB**, **LCD2L.LIB**, **LP.LIB**, **LP_8.LIB**, **PK21XX.LIB**, **PLC_EXP.LIB**, **STEP.LIB**, **THERMADC.LIB**, **UART2.LIB**, **UART232.LIB**, **UART3.LIB**, **UIBOARD.LIB**

PK2200

12X16L.LIB, **16X20L.LIB**, **5KEY.LIB**, **5KEYEXTD.LIB**, **8X10L.LIB**, **AASCUART.LIB**, **CPLC.LIB**, **ENGFNT2L.LIB**, **ENGFONT.LIB**, **ENGFONT2.LIB**, **ENGFONTL.LIB**, **EZIOCMMN.LIB**, **EZIOPLC.LIB**, **FK.LIB**, **GATE_P.LIB**, **GLCD.LIB**, **KDI.LIB**, **KDM.LIB**, **KP.LIB**, **KP_KDI.LIB**, **LCD2L.LIB**, **LP.LIB**, **LP_16.LIB**, **PK22XX.LIB**, **PLC_EXP.LIB**, **SF1000_Z.LIB**, **STEP.LIB**, **THERMADC.LIB**, **UART2.LIB**, **UART232.LIB**, **UART3.LIB**, **UIBOARD.LIB**, **V256X64.LIB**, **WINTEK.LIB**, **ZWLOGOS.LIB**

PK2300

EZIOCMMN.LIB, **EZIODPWM.LIB**, **EZIOPK23.LIB**, **THERMADC.LIB**

PK2400

12X16L.LIB, **16X20L.LIB**, **8X10L.LIB**, **ENGFNT2L.LIB**, **ENGFONT.LIB**, **ENGFONT2.LIB**, **ENGFONTL.LIB**, **EZIOCMMN.LIB**, **EZIOPK24.LIB**, **GLCD.LIB**, **KDI.LIB**, **LP.LIB**, **LP_BL145.LIB**, **KP.LIB**, **KP_KDI.LIB**, **THERMADC.LIB**, **WINTEK.LIB**, **ZWLOGOS.LIB**

PK2500

EZIOCMMN.LIB, **EZIODPWM.LIB**, **EZIOPK25.LIB**

PK2600

12X16L.LIB, **16X20L.LIB**, **8X10L.LIB**, **AASCSCC.LIB**, **AASCURT2.LIB**, **ENGFNT2L.LIB**, **ENGFONT.LIB**, **ENGFONT2.LIB**, **ENGFONTL.LIB**, **EZIOBL17.LIB**, **EZIOCMMN.LIB**, **EZIOOP71.LIB**, **EZIOPLC2.LIB**, **KDI.LIB**, **KDM.LIB**, **KP.LIB**, **KP_OP71.LIB**, **LCD2L.LIB**, **LP.LIB**, **LP_16.LIB**, **OP71HW.LIB**, **OP71L.LIB**, **OP71P.LIB**, **QVGADEMO.LIB**, **SF1000_Z.LIB**, **STEP2.LIB**, **UART2.LIB**, **UART232.LIB**, **UART3.LIB**, **ZWLOGOS.LIB**

Z104/ZISA

COM232.LIB, **CPLC.LIB**, **LCD2L.LIB**, **THERMADC.LIB**, **UIBOARD.LIB**, **Z104.LIB**

APPENDIX C: USING AASC LIBRARIES

The Abstract Application-Level Serial Communication (AASC) library and its low-level support functions facilitate serial communication between controllers and between a controller and another device such as a PC.

AASC Library Description

AASC libraries allow the programmer to create buffered character streams that perform input/output to/from ports in the communication devices. One principal library, **`AASC.LIB`**, contains all the functions required for these tasks. Table C-1 lists the support libraries used with **`AASC.LIB`**.

Table C-1. Drivers Used in AASC.LIB

Driver Library	Description
`AASCDIO.LIB`	Contains specific standard input/output (**`STDIO`**) routines to support the AASC libraries.
`AASCSCC.LIB`	Operates channels on the Zilog 85C30 Serial Communication Controller used in BL1100 and BL1700 controllers.
`AASCUART.LIB`	Operates RS-232 port on the XP8700 PLCBus expansion board supported by most Z-World controllers.
`AASCURT2.LIB`	Operates RS-232 port on the XP8700 PLCBus expansion board on controllers (e.g., BL1700) with 16-bit PLCBus addressing.
`AASCZ0.LIB`	Handles communication on the Z0 port of the Zilog Z180 microprocessor used by Z-World controllers. This port is usually connected to an RS-232 driver.
`AASCZ1.LIB`	Handles communication on the Z1 port of the Zilog Z180 microprocessor used by Z-World controllers. This port is usually connected to an RS-485 driver.
`AASCZN.LIB`	Operates ZNet-specific routines on the RS-485 network. All participating controllers must use the same driver. One controller is designated the *master controller* by defining the macro **`ZNMASTER`** to be non-zero before invoking **`#use AASCZN.LIB`**. This library uses the Z1 port of the Zilog Z180 microprocessor.

The AASC libraries are as device-independent as possible. Programs include only the **`AASC.LIB`** code and the code required for the communication devices used by the application (for example, **`AASCSCC.LIB`**). The application handles different communication devices simply by creating separate device channels.

Two hidden circular buffers for each AASC channel store incoming and outgoing information. This allows the application to process incoming and outgoing information in chunks not larger than the circular buffers. The buffer size is specified in the application.

AASC support libraries implement custom device drivers and interrupt service routines (ISRs) for each communication device. The application only needs to initialize a channel and a local buffer, then make function calls to check the status of the buffers, and read or write to/from the buffers.

AASC Library Operation

AASC libraries read (receive), write (transmit), peek (search), provide status, and handle errors. Figure C-1 shows the hierarchy of these AASC functions. Note that the management of the circular buffer and the hardware/serial ISR levels is hidden from the programmer. These two reserved levels are contained in the support libraries listed in Table C-1.

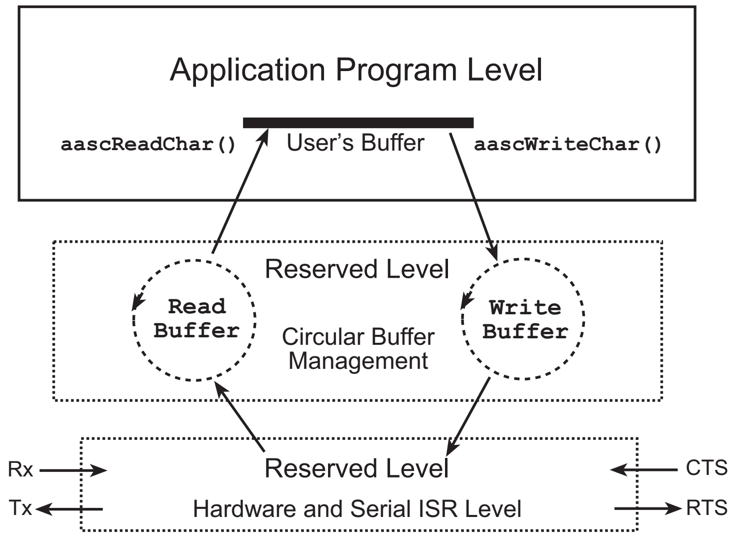

Figure C-1. Hierarchy of AASC Functions

Read

Information is received either by block or by byte. Only one method is needed, but the other can always be implemented. It is more efficient to have both methods available. The block read function supports fixed-size and variable-size reads. The application may read exactly *n* bytes, it may read nothing at all, or it may read up to *n* bytes. In any case, the function returns the number of bytes actually read.

Read operations may preempt write operations and vice versa, but a read operation cannot preempt another read operation and a write operation cannot preempt another write operation.

Write

The transmit (write) routines are mirror images of the read functions. There is one function for byte writes and one for block writes. The block write function can write part of a block, or it may write all or none of the block. This is important for multi-threaded programs because writing all or none prevents interleaving messages originating from different cooperative threads.

Peek

A special function supported by the AASC libraries allows the application to "peek" into the buffer without retrieving a byte. The peek function **`aascPeek`** searches for a substring, for example, to identify the type of incoming packet, without actually changing the contents of the buffer. Another "peek" type function, **`aascScanTerm`**, can also search for a particular character such as the terminating character of a packet.

Status and Errors

AASC libraries provide full status reports about the application. The libraries can report the number of bytes used and the number of bytes still free in the read or write buffers. Such information is useful for the application to schedule message checking or dynamic transmission.

AASC libraries also report both hardware errors (for example, framing error, parity error) and software errors (for example, buffer overrun). Error conditions are not cleared automatically.

Library Use

Follow these six steps when using AASC libraries.

1. Identify the communication device (e.g., Z0, SCC Channel A, UART).
2. Allocate and initialize the channel with **`aascOpen()`**.
3. Set up read (receive) and write (transmit) circular buffers (e.g., use **`aascSetReadBuf()`**).
4. Carry out reads and writes (e.g., use **`aascWriteChar()`**).
5. Check status and handle errors (e.g., use **`aascGetError()`**).
6. When finished, close the channel with **`aascClose()`**.

Sample Program

The following sample program provides an example of the use of the AASC framework in asynchronous serial communication with a terminal. The program demonstrates how to use port SCC Channel A as an AASC device. Other sample programs may be found in the Dynamic C **`SAMPLES\AASC`** subdirectory.

This program simply echoes text typed at an ascii terminal back to the terminal. Connect a controller with a serial communication controller IC through SCC Channel A to a PC or dumb terminal. If using a PC, Windows **`terminal.exe`** can be used in **ANSI Terminal Emulation** with **Local Echo** disabled and **Flow Control** set to **None**. If RTS/CTS handshaking is enabled by setting the macro **SHAKE** to non-zero, enable **Flow Control**" within **`terminal.exe`** to **Hardware**. This sample program defaults to settings of **No Parity**, **One Stop Bit**, and **Eight Data Bits**. Set your PC accordingly.

The following steps describe how this "echoing" process works.

1. The program accesses **`AASC.LIB`** and the appropriate AASC library **`AASCSCC.LIB`** with **`#use`**.
2. Definitions are created for circular read and write buffers, and for the user buffer **`workBuffer`**. A user buffer pointer, **`pworkBuffer`**, is also created for this example.
3. The **`_GLOBAL_INIT()`** function chain is called to initialize the AASC framework.
4. The function **`aascOpen()`** is used to create a channel to the **`DEV_SCC`** device at 8N1.
5. The program checks to make sure that a controller with an SCC IC is being used.
6. The transmitter and receiver for the channel **`chan`** are switched on by **`aascTxSwitch()`** and by **`aascRxSwitch()`**.
7. The program sets up the circular buffers with **`aascReadBuf`** and **`aascSetWrite Buf`**.
8. If a character is read, the program enters another loop that sends the characters in **`workBuffer`** back to the remote terminal. The function will not return until all the characters are read from **`workBuffer`** and sent back to the terminal. (For example, if two characters are in **`workBuffer`**, the function will return only when both characters are sent.)

SCCECHO.C

```
#use aasc.lib
#use aascscc.lib
#define BUFSIZE 684       // Size of circular buffer.
#define BAUDMULT 8        // multiples of 1200 bps
                          // (8 × 1200 bps = 9600 bps).
#define SHAKE 0           // Set to 1 for RTS/CTS handshaking.
char readBuffer[BUFSIZE],writeBuffer[BUFSIZE];
char workBuffer[BUFSIZE],*pworkBuffer;
struct _Channel *aascChannel;
main(){
   _GLOBAL_INIT();        // This must be the first action
                          // performed in main().

   // Open channel A of the SCC at 8N1

   aascChannel = aascOpen( DEV_SCC, SHAKE,
      SCC_A | SCC_1STOP | SCC_NOPARITY | SCC_8DATA |
      SCC_1200*BAUDMULT, NULL);
   if(aascChannel==NULL) {
      printf("SCC channel A not available.");
      return;
   }

   // Set up the circular buffers.

   aascSetReadBuf( aascChannel, readBuffer,
      sizeof( readBuffer));
   aascSetWriteBuf( aascChannel, writeBuffer,
      sizeof( writeBuffer));

   // Process the data transfer.

   while(1) {
      hitwd();

      // Perform data transfer.

      if( aascReadChar( aascChannel, workBuffer) ) {
         while( !aascWriteChar( aascChannel,
            workBuffer[0]) ) {
            hitwd();
         }
      }
   }
}
```

XModem Transfer

The AASC libraries have extensive support for the **`XModem-CRC`** transfer protocol. The AASC libraries allow the application to define callback functions to read or write each block of an XModem packet. This means there is no need to have the entire transfer block ready before transmission, or to allocate space for the entire incoming block. Default callback functions are provided for normal read-to-memory or write-from-memory operations.

Library Use

1. Initialize the virtual driver.
2. Initialize the AASC framework with an appropriate device such as SCC Channel A.
3. Initialize an XModem data buffer and the number of bytes to transfer with **`aascXMWrInitPhy()`** or **`aascXMRdInitPhy()`** for physical memory, or **`aascXMWrInitLog()`** or **`aascXMRdInitLog()`** for logical memory.
4. Initialize XModem transfer with **`aascWriteXModem()`** or **`aascReadXModem()`**.
5. Perform the XModem transfer with **`aascWriteModem()`** or **`aascReadXModem()`**.

Sample Program

The following sample program provides an example of the use of an AASC framework in XModem data transfer. The program sends one block of 128 characters to a remote device using **`XModem-CRC`**. Configure the remote device for 9600 bps at 8N1 without RTS/CTS flow control.

The virtual driver must be used since XModem incorporates costatements to enable multitasking.

Note that any channel may be used by changing SCC Channel A to the desired port. For example, to use port Z1 on the Z180, change **`AASCSCC.LIB`** to **`AASCZ1.LIB`**, and change the parameters in **`aascOpen()`** to reflect those for Z1.

The following steps describe the XModem transmission example.

1. The program accesses the appropriate libraries with **`#use`**.
2. Definitions are created for the circular read and write buffers, and for the XModem buffer.
3. **`aascInit()`** is called to initialize the AASC framework.
4. A data string is created for transfer.
5. **`VdInit()`** is called to initialize the virtual driver.
6. **`aascOpen()`** is used to create a channel to the **`SCC_A`** device at 8N1 and 9600 bps.
7. The program checks for the presence of the SCC chip on the controller.
8. The circular buffers are then initialized by **`aascSetReadBuf()`** and by **`aascSetWriteBuf()`**, and are made accessible to the AASC framework.
9. XModem transmission is then performed by repeatedly calling **`aascWriteXModem()`** with the initialization parameter set to 0.
10. XModem transmission finishes when **`aascWriteXModem()`** returns a 1.

XM_SEND.C

```
#use vdriver.lib
#use aasc.lib
#use aascscc.lib
#define BUFSIZE 1024      // Size of circular buffer.
#define BAUDMULT 8        // multiples of 1200 bps
                          // (8 × 1200 bps = 9600 bps).
struct _Channel *aascChannel;
char circBufIn[BUFSIZE], circBufOut[BUFSIZE];
char aascBuffer[BUFSIZE];
int aascInit(void);
void main(void){
   // Initialize the AASC framework.
   if( !aascInit() ) exit(-1);
   // Create some data to transfer.
   strcpy( aascBuffer, "This is some xmodem data transfer...");
   // Process the data transfer.
   while(1) {
      hitwd();
      printf("Press any key to initiate Xmodem
         Controller-to-Device transfer.\r");
      hitwd();
      if( kbhit() ) {
         getchar();
         printf("\n\nXmodem transfer initiated...\n");
         hitwd();
         // Set up XModem transfer to logical memory.
         aascXMWrInitLog( (unsigned) aascBuffer, 128);
         aascWriteXModem( aascChannel, 0, 1,
            aascWrCallBackLg );
         while( !aascWriteXModem( aascChannel, 0, 0,
            aascWrCallBackLg ) ) hitwd();
         printf("\n\nXmodem transfer finished...\n\n");
         hitwd();
      }
   }
}
int aascInit(void){
   // Initialize the virtual driver
   VdInit();
   // Open channel A of the SCC at 8N1
   aascChannel = aascOpen( DEV_SCC, 0,
      SCC_A | SCC_1STOP | SCC_NOPARITY | SCC_8DATA |
         SCC_1200*BAUDMULT, NULL);
   if(aascChannel==NULL) {
      printf("SCC channel A not available.");
      return;
   }
   // Set up the circular buffers.
   aascSetReadBuf( aascChannel, circBufIn,
      sizeof(circBufIn) );
   aascSetWriteBuf( aascChannel, circBufOut,
      sizeof(circBufOut) );
}
```

Z-World, Inc.
2900 Spafford Street
Davis, California 95616-6800 USA

Telephone:	(530) 757-3737
Facsimile:	(530) 753-5141
Web Site:	http://www.zworld.com
E-Mail:	zworld@zworld.com

Printed in U.S.A.