



Dynamic C[®] 32

for Zilog Z180 microprocessors

Version 6.x

Integrated C Development System

Function Reference

010124 - C

Dynamic C 32 v. 6.x Function Reference

Part Number 019-0082 • 010124 - C • Printed in U.S.A.

Copyright

© 2001 Z-World, Inc. All rights reserved.

Z-World, Inc. reserves the right to make changes and improvements to its products without providing notice.

Trademarks

- Dynamic C® is a registered trademark of Z-World, Inc.
 - PLCBus™ is a trademark of Z-World, Inc.
 - Windows® is a registered trademark of Microsoft Corporation.
 - Modbus® is a registered trademark of Modicon, Inc.
 - Hayes Smart Modem® is a registered trademark of Hayes Microcomputer Products, Inc.
-

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: zwworld@zwworld.com

TABLE OF CONTENTS

About This Manual	vii
Chapter 1: General Support Libraries	11
Global Initialization	12
BIOS Functions	12
COSTATE.LIB	16
CTYPE.LIB	17
MATH.LIB	18
STDIO.LIB	22
STRING.LIB	27
SYS.LIB	30
UTIL.LIB	33
XMEM.LIB	33
Chapter 2: Multitasking Libraries	37
RTK.LIB	38
SRTK.LIB	40
VDRIVER.LIB	40
Chapter 3: AASC Libraries	43
AASC.LIB	44
XModem Functions in AASC.LIB	50
Chapter 4: Other Communication Drivers	55
MODEM232.LIB	56
NETWORK.LIB	56
PRPORT.LIB	58
SCC232.LIB	62
SERIAL.LIB	68
S0232.LIB	70
S1232.LIB	74
Z0232.LIB	74
Z1232.LIB	78

Chapter 5: Modbus Slave Libraries	79
Getting Started	80
Standard Modbus Slave Procedure	80
Advanced Modbus Slave Procedure	83
Modbus Registers	84
Modbus Slave Command Handlers	85
Modbus Slave Serial Interface	87
High-Resolution Timer	89
Modbus Slave Supported Commands	89
Modbus Slave Unsupported Commands	90
 Chapter 6: Modbus Master Libraries	 91
Getting Started	92
Standard Modbus Master Procedure	92
Advanced Modbus Master Procedure	94
Modbus Master Timeouts	95
Modbus Registers	95
Modbus Master Command Functions	96
Modbus Master Serial Interface	104
Modbus Master Supported Commands	105
Modbus Master Unsupported Commands	105
Modbus Master Command Function Return Values	106
 Chapter 7: Other Libraries	 111
5KEY.LIB	112
5KEYEXTD.LIB	117
CPLC.LIB	119
DRIVERS.LIB	120
DMA.LIB	127
FK.LIB	131
IOEXPAND.LIB	133
KDM.LIB	137
LCD2L.LIB	145
MISC.LIB	147
PBUS_LG.LIB	147
PBUS_TG.LIB	151

Appendix A: Dynamic C Libraries	155
Appendix B: Using AASC Libraries	161
AASC Library Description	162
AASC Library Operation	163
Read	163
Write	164
Peek	164
Status and Errors	164
Library Use	164
Sample Program	164
XModem Transfer	167
Library Use	167
Sample Program	168
Appendix C: Library Lists for Z-World Products	171
BL1000	172
BL1100	172
BL1200	172
BL1300	172
BL1400	172
BL1500	172
BL1600	172
BL1700	172
LP3100	173
PK2100	173
PK2200	173
PK2300	173
PK2400	173
PK2500	173
PK2600	173
CM7100	173
CM7200	173
OP7100	174
Index	175

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 Z80 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	Electrically 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
	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.
<i>Italics</i>	Indicates that something should be typed instead of the italicized words (e.g., in place of <i>filename</i> , 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`s 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 int inport(unsigned int 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 int port,
 unsigned int 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. It returns a negative value if unable to write the EEPROM.

- **void di()**

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

- **void DI()**

Disables interrupts. Dynamic C expands this call inline.

- **void ei()**

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

- **void EI()**

Enables interrupts. Dynamic C expands this call inline.

- **int iff()**

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

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

Reads the value of the specified **bit** at memory address. The **bit** may be from 0 to 31. Use **BIT** (upper case) for inline 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 int BIT(void *address, unsigned int 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 inline. 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 int bit)**

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

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

- **void SET(void *address, unsigned int 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 inline. This is equivalent to the following expression:

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

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

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

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

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

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

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

- **unsigned int IBIT(unsigned int port,
 unsigned int 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 int port, unsigned int 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 int port, unsigned int 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()**

“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()**

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 non-zero 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()**

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()**

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()**

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()**

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()**

Detects keystrokes in the Dynamic C **STDIO** window. The function returns non-zero 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 int sysclock()**

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)

The function returns clock speed / 1200.

- **int powerlo()**

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 upper-case equivalent.

- **int tolower(int c)**

Converts character **c** to its lower-case equivalent.

- **int islower(int c)**

Checks whether **c** is a lower-case character. The function returns non-zero if so, and zero otherwise.

- **int isupper(int c)**

Checks whether **c** is an upper-case character. The function returns non-zero if so, and zero otherwise.

- **int isdigit(int c)**

Checks whether **c** is an ASCII digit (0–9). The function returns non-zero if so, and zero otherwise.

- **int isxdigit(int c)**

Checks whether **c** is a hexadecimal digit (0–9, a–f, A–F). The function returns non-zero if so, and zero otherwise.

- **int ispunct(int c)**

Checks whether **c** is a punctuation mark. The function returns non-zero if so, and zero otherwise.

- **int isspace(int c)**

Checks whether **c** is a blank, tab, new line, or form feed. The function returns non-zero if so, and zero otherwise.

- **int isprint(int c)**

Checks whether **c** is printable. The function returns non-zero if so, and zero otherwise.

- **int isalpha(int c)**

Checks whether **c** is an ASCII letter. The function returns non-zero if so, and zero otherwise.

- **int isalnum(int c)**

Checks whether **c** is alphanumeric (A to Z, a to z and 0 to 9). The function returns non-zero if so, and zero otherwise.

- **int isgraph(int c)**

Checks whether **c** is a visible printing character. The function returns non-zero if so, and zero otherwise.

- **int iscntrl(int c)**

Checks whether **c** is a control character (less than 20_H). The function returns non-zero if so, and 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 10³⁸ to -10³⁸) 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 $-\text{INF}$ and $+\text{INF}$.

- **float acsc(float x)**

Computes the arccosecant of **x**. The value of **x** must be between $-\text{INF}$ and $+\text{INF}$.

- **float asec(float x)**

Computes the arccosecant of **x**. The value of **x** must be between $-\text{INF}$ and $+\text{INF}$.

- **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 $-\text{INF}$ and $+\text{INF}$.

- **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:

<i>angle</i>	$x \neq 0, y \neq 0$
$\text{PI}/2$	$x = 0, y > 0$
$-\text{PI}/2$	$x = 0, y < 0$
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 $|\mathbf{x}| > 89.8$ (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 $x \geq 0$; 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 $= 22.7 - 2 \times 10.3 = 2.1$.

- **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 $x * (\text{radix} ** n)$, 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 $-8 \times \text{PI} \leq x \leq +8 \times \text{PI}$. 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° ($\text{PI}/2$) 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.

- **void 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 and places the formatted string in ***buffer**. 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,

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

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

The functions **printf()** and **sprintf()** are not reentrant. The function **doprnt** implements **printf** and **sprintf**, and uses the character output function **putc** specified by the programmer. 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 ]]letter
```

where

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

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.

letter selects the interpretation of the data according to the following list.

- d** decimal conversion (expects type **int**)
- o** octal conversion
- x** hex conversion
- u** unsigned decimal conversion (expects type **unsigned int**)
- c** single **char** representation (expects type **char**)

s string (with null termination)

e mantissa/exponent form of floating point (expects type **float**)

f normal floating point (expects type **float**)

g use e or f conversion, whichever is shorter. (expects **float**)

l decimal conversion (expects type **long**)

- **void printf(char *fmt, ...)**

This standard function accepts a variable number of value arguments, composes a formatted string from the values, and writes the formatted string to the **STDIO** window. 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. Passes a function **put** that outputs 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 **_gtoa** 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 **_gtoa** 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 unsigned integer. 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 "1000000000", and returns -10. If **f** is 45.678, **ftoa** will generate the character string "45678" and return the integer exponent -3, indicating 45678×10^{-3} .

- **void plhex(char left, int n1, 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 **n1** is the expected length of the output. Asterisks will be written if **num** requires more width than **n1**. 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 n1, 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**. The programmer has quite a bit of control over format.

The **flag**, if **'-'**, indicates that the output field is left-justified. If it is **'0'**, 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'**. These formats correspond to programming conventions established many years ago. E format displays a mantissa with “e” and an exponent. F format is standard decimal format. G format allows the compiler to decide whether to use “e” or “f” format.

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 non-zero (true), the function prints **digits** significant digits. Otherwise, the function prints six significant digits. The function prints all asterisks 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 int value, char *buf)**

Converts unsigned integer **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)**

Converts a character string to a floating-point value. The initial “white space” is ignored. This is ANSI compatible. The function returns the converted value.

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

Converts a character string to an integer value. The initial “white space” is ignored. This is ANSI compatible. The function returns the converted value.

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

Converts a character string to a long integer value. The initial “white space” is ignored. This is ANSI compatible. The function returns the converted value.

- **void *memset(void* dst, byte ch, unsigned int 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 int 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 int 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 ai < bi
> 0 if ai > bi

```

- **int strncmp(char *a, char *b, unsigned int 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 int strspn(char *src, char *set)**

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

- **unsigned int 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 int 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 int 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. The function skips the initial “white space.” The function sets the tail pointer ****tail** to the character position at which the conversion failed or finished. The next conversion may resume at the location specified by ****tail**. If no conversion takes place, ****tail** returns 0L. The initial “white space” is ignored. This function is ANSI compatible. The function returns the converted value.

Be careful with the double pointer.

- **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 it also finds a terminating delimiter, it changes it to a null character so that the token is terminated. This function modifies the source string. The function returns a null pointer if it does not find a token.

- **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 int 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.

The function returns zero when it is executed. After **longjmp** is executed, the program counter, stack pointer, etc., are restored to the

```
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
```

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 int 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 pre-empted 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 int bfree(void *lo, void *hi)**

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

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

Returns block (***f**) of dynamically allocated RAM to the free list. The function returns non-zero 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 int count,
 unsigned int 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 int n, unsigned int 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( *p, *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 non-zero otherwise.

- **char *realloc(void *ptr, unsigned int 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 int 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 int 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()**

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()**

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()**

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()**

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(int vector, int(*function)())**

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



reload_vec writes to the flash memory when executed on a controller with a flash EPROM. Be careful not to have this function call write repeatedly to the same flash EPROM address since the flash EPROM has a maximum of about 10,000 writes.

UTIL.LIB

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

- **int IsZ80180()**

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)**

Converts logical address **address** to a physical address according to the memory mapping registers. Uses BBR, CBR and CBAR to determine the physical address of any given logical address. The function returns 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 int 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 int 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 int xstrlen(unsigned long address)**
Returns the length of the string at the extended memory **address**. The address is a physical **address** (20 bits).
- **unsigned int 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, and the virtual driver.

RTK.LIB

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

- **int request(unsigned int 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 run_timer()**
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 rkernl()**

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 int 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.

- **int init_kernel()**

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. **#define NTASKS** to be the number of tasks. Then set up a periodic interrupt with a service routine that calls **run_timer**. An option is to define **TASKSIZE_STORE** to be the size of the task storage area (this defaults to 50 if **TASKSIZE_STORE** is not defined).

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()**

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()**

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

- **void init_srtkernel()**

Initializes the simplified real-time kernel. Once this is called, periodic interrupts will automatically invoke the SRTK high- 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 “fastcall” very high priority task.

The virtual driver also provides delay routines for use by **waitfor** statements **DelayMs**, **DelaySec**, and **DelayTick**.

- **void VdInit()**

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.

For fastcall service, the virtual driver clocks **vd_quickloop** every **n** ticks (1/1280 seconds) where $1 \leq n \leq 255$. **vd_quickloop** must be defined and the definition will override the dummy version in the library. (**#define VD_FASTCALL 1** must be defined 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 VdGetFreeWd(byte 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** \leq 2, **VdWdogHit** must be called every 25 milliseconds. If **count** = 255, hit the watchdog at least every 6.375 seconds.

- **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.

- **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.

- **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. If *n* = 0, it turns off **fastcall**. Use **#define VD_FASTCALL 1**, call **VdInit**, then call this function. (**VdInit** initializes fastcall as *off*.) The function returns 1 for success, 0 for a bad *n* value.

- **void VdAdjClk()**

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

- **vd_fastcall()**

Is called by the virtual driver to run an ultra-fast thread every *n* ticks, where *n* is the argument to **vd_initquickloop()** and should be between 0 and 255. Use **#define VD_FASTCALL 1** to activate this thread. *n* = 0 shuts off **fastcall**.



*CHAPTER 3: **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.LIB

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.

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 be used by any applicable hardware drivers.

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

Opens a channel of device **Type**, and initializes the device with parameter **Param**.

PARAMETERS: **Type** is the type of communication device to open.

DEV_Z0 for the Z0 port,
DEV_Z1 for the Z1 port,
DEV_SCC for the Serial Communication Controller port,
DEV_ZNET for the network device, and
DEV_UART for the XP8700.

CRTS specifies whether CTS/RTS handshaking should be used: 1 means CTS/RTS handshaking is used, 0 means CTS/RTS handshaking is *not* used.

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

Number of Data Bits	Number of Stop Bits	Number of Parity Bits
ASCII_PARAM_7D	ASCII_PARAM_1STOP	ASCII_PARAM_NOPARITY
ASCII_PARAM_8D	ASCII_PARAM_2STOP	ASCII_PARAM_OPARITY
		ASCII_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 online help for additional macros.

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

ASCII_PARAM_7D | ASCII_PARAM_1STOP | ASCII_NOPARITY

Two commonly used combination macros have also been defined.

ASCII_PARAM_1200—Basic quantum for baud rate. Multiply by the factor baud rate ÷ 1200 (for example, 8 for 9600 bps).

ASCII_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*ASCII_PARAM_1200 | ASCII_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 **void *brqfnc** is null.

RETURN VALUE: 16-bit quantity of type **CHANNEL** for all further channel operations. **aascOpen** returns null if no channels can be assigned if break processing is not used.

- **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 going to share a channel. See **aascOpen** for parameter details.

- **void aascClose(CHANNEL Channel)**

Closes the channel numbered **Channel**. First, **aascClose** calls the device-dependent routine to close the device. Then the storage associated with this channel is reattached to the free list.

PARAMETER: **Channel** is the logical channel.

- **void aascSetReadBuf(CHANNEL channel, char *Buffer, unsigned size)**

Designates a memory area pointed to by **Buffer** of **size** to be the receive buffer for **channel**.

PARAMETERS: **channel** to be read from must be opened by an **aascOpen** call.

Buffer is the address of the receive buffer.

size is the size of the receive buffer.

- **void aascSetWriteBuf(CHANNEL Channel,
char *Buffer, unsigned size)**

Designates a memory area pointed to by **Buffer** of **size** to be the transmit buffer for **Channel**.

PARAMETERS: **Channel** to write to must be opened by an **aascOpen** call.

Buffer is the address of the transmit buffer.

size is the size of the transmit buffer.

- **void aascRxSwitch(CHANNEL Channel, char OnOff)**

Activates or deactivates the channel receiver.

PARAMETERS: **Channel** is the logical channel.

OnOff 0 is off, otherwise the channel transmitter is on.

- **void aascTxSwitch(CHANNEL Channel, char OnOff)**

Switches the channel transmitter on or off.

PARAMETERS: **Channel** is the logical channel.

OnOff 0 is off, otherwise the channel transmitter is on.

- **unsigned aascReadChar(CHANNEL Channel,
char *Dest)**

Reads a character from channel **Channel** to the memory pointed to by **Dest**. 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).

PARAMETERS: **Channel** is the logical channel.

Dest is the address (buffer) to read character into.

RETURN VALUE: The actual number of bytes read from the channel.

- **unsigned aascReadBlk(CHANNEL Channel,
void *Dest, unsigned Length, char Flags)**

Reads a block of **Length** bytes from logical channel **Channel** to the memory pointed to by **Dest**. If **Flags** is non zero, either the entire **Length** or no bytes will be read. The receiver will be enabled automatically if flow control is enabled and the receive buffer has more than 16 bytes left (after the read).

PARAMETERS: **Channel** is the logical channel.

Dest is the address (buffer) to read into.

Length is the number of bytes to read.

If **Flag** is non-zero either all **Length** bytes will be read or no bytes will be read.

RETURN VALUE: The actual number of bytes read from the channel.

- **unsigned aascWriteChar(CHANNEL Channel,
char Src)**

Writes a character **Src** to logical channel **Channel**. The transmitter is enabled automatically after the character is transferred.

RETURN VALUE: The actual number of bytes written to the channel.

- **unsigned aascWriteBlk(CHANNEL Channel,
void *Src, unsigned Length, char Flags)**

Writes a block of **Length** bytes to logical channel **Channel** from the memory pointed to by **Src**. If **Flags** is non zero, either the entire **Length** or no bytes will be written. The transmitter is turned on automatically after the bytes are written to the buffer.

PARAMETERS: **Channel** is the logical channel.

Src is the address (buffer) to write from.

Length is the number of bytes to write.

If **Flag** is non-zero either all **Length** bytes will be written or no bytes will be written.

RETURN VALUE: The actual number of bytes written to the channel.

- **unsigned aascPeek(CHANNEL Channel,
void *pMatchee, unsigned size)**

Tries to match as much data of up to size **size** as possible pointed to by **pMatchee** (not null-character terminated).

PARAMETERS: **Channel** is the logical channel.

pMatchee is the address of string to match.

size is the number of bytes to attempt to match.

RETURN VALUE: The number of bytes actually matched.

- **unsigned aascScanTerm(CHANNEL Channel,
char Term)**

Scans the receive buffer of logical channel **Channel** for the terminating character **Term**. Note that this function does not read any bytes from the receive buffer. The receiver will be enabled automatically if flow control is enabled and the receive buffer has more than 16 bytes remaining.

RETURN VALUE: The packet size terminated by **Term**.

- **void aascPipe(CHANNEL Channel, CHANNEL Out,
CHANNEL In)**

Makes a pipe by diverting the output of **Channel** to the input of **Out**, and diverting the input of **Channel** from **In**.

PARAMETERS: **Channel**, **Out**, and **In** are logical channels.

- **long aascGetError(CHANNEL Channel)**

Gets the current error condition.

PARAMETER: **Channel** is the logical channel.

RETURN VALUE: Depends on the device. For specific return values, see the description of the device driver's **<device_name>GetErr()** function (for example, **z0GetErr()**).

- **void aascClearError(CHANNEL Channel)**

Clears the error condition.

PARAMETER: **Channel** is the logical channel.

- **unsigned aascReadBufLeft(CHANNEL Channel)**

Computes the number of bytes left to be read from the receive buffer of logical channel **Channel**.

RETURN VALUE: The number of bytes left to be read.

- **unsigned aascWriteBufLeft(CHANNEL Channel)**

Computes the number of bytes left to be transmitted from the transmit buffer of logical channel **Channel**.

RETURN VALUE: The number of bytes left to be transmitted.

- **unsigned aascReadBufFree(CHANNEL Channel)**

Computes the number of bytes free in the receive buffer of logical channel **Channel**.

RETURN VALUE: The number of free bytes.

- **unsigned aascWriteBufFree(CHANNEL Channel)**

Computes the number of free bytes in the transmit buffer of logical channel **Channel**.

RETURN VALUE: The number of free bytes.

- **void aascFlush(CHANNEL Channel)**

Flushes the buffers associated with logical channel **Channel**, and loses all information that may be left in the buffers. If the channel is capable of CTS/RTS flow control, the programmer should determine whether to explicitly reenale the receive channel by calling **aascRxSwitch**.

aascRxSwitch will disable RTS explicitly to allow the other side to transmit.

- **void aascFlushRdBuf(CHANNEL Channel)**

Flushes the read buffer associated with logical channel **Channel**, and loses all information that may be left in the buffer. If the channel is capable of CTS/RTS flow control, the programmer should determine whether to explicitly reenale the receive channel by calling **aascRxSwitch**. **aascRxSwitch** will explicitly disable RTS to allow the other side to transmit.

- **void aascFlushWrBuf(CHANNEL Channel)**

Flushes the write buffer associated with logical channel **Channel**. All information is erased from the buffer.

- **void aascPrintf(CHANNEL Chan, char *fmt, ...)**

Prints a formatted string to channel **Chan**.

PARAMETERS: **Chan** is the channel to send to.

fmt is the format of the string to be printed.

Arguments (if any) should follow **fmt**.

- **void aascVPrintf(CHANNEL Chan, char *fmt, void *firstArg)**

Prints a formatted string to channel **Chan**.

PARAMETERS: **Chan** is the channel to send to.

fmt is a format string.

firstArg is a pointer to the first argument.

XModem Functions in AASC.LIB

The XModem protocol performs packet-based file transfers with CRC error detection.

The packet structure for XModem transfer appears below.

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

- **void aascXMRdInitPhy(unsigned Where, unsigned Length, unsigned long XmemSrcAddr)**

Initializes location and size of physical memory for **aascReadXModem()** PC-to-target data transfer. Specifies the location on the target and the maximum number of bytes to be transferred from the PC.

PARAMETERS: **Where** is the root memory location on the target where the data being transferred are placed.

Length is the maximum number of bytes to transfer.

XmemSrcAddr is the final memory destination.

- **void aascXMRdInitLog(unsigned Where,
 unsigned Length)**

Initializes location and size of logical memory for **aascReadXModem()** PC-to-target data transfer. Specifies the location on the target and the maximum number of bytes to be transferred from the PC.

This default function tells the default callback read function **aascRdCBackLocLg** to advance the buffer pointer by the packet size after receiving each packet.

PARAMETERS: **Where** is the root memory location on the target where the data being transferred are placed.

Length is the maximum number of bytes to transfer.

- **unsigned aascReadXModem(CHANNEL Channel,
 char *(*read_callback_loc)(),
 void (*read_callback_mod)(),
 char Initialize)**

Performs XModem PC-to-target download. Call this function once with **Initialize** set to 1. Then set **Initialize** to 0, and call this function repeatedly until its return value is non zero.

Call **aascXMRdInitPhy()** for physical memory transfers or **aascXMRdInitLog()** for logical memory transfers before using **aascReadXModem()**.

PARAMETERS: **Channel** is the channel being read from.

read_callback_loc is a pointer to a callback function that will be called by this function BEFORE each XModem packet is received. This function determines where the packet is placed in memory using the callback function **aascRdCBackLocLg()**.

read_callback_mod is a pointer to a callback function that will be called by this function AFTER each XModem packet is received. This function performs further processing on the data. A default function **aascRdCBackLocPh()** that does no processing is provided.

Initialize is the initialization flag. Set **Initialize** to 1 to initialize XModem on the first call. Set **Initialize** to 0 for all subsequent calls.

RETURN VALUE:

XX_SUCCESS	XX_TIMEOUT
XX_COMMERR	XX_CANCEL
XX_SEQ	XX_CHKSUM
XX_NOSTART	XX_NOBEGPAK
XX_SYNC	

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

Dummy function called by **aascReadXModem()** after a packet is received. Can be replaced by user-defined function if modifications are required on a packet.

PARAMETERS: **PackSize** is the packet size being used by XModem (128 or 1024 bytes).

PackSize is the number of the current packet.

RETURN VALUE: Root memory logical address where packet from PC will be placed before transfer to physical memory.

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

Default callback function for addressing blocks for PC-to-target transfers. This is called by **aascReadXModem()** before receiving a packet from the PC. This function advances the pointer to the target memory by **PackSize** after each packet is sent.

PARAMETERS: **PackSize** is the packet size being used by XModem.

0 - use 128 byte XModem packets

1 - use 1024 byte XModem packets

PackNum is the number of the current packet.

RETURN VALUE: The logical memory address where the packet from the PC will be placed.

- **void aascXMWrInitPhy(unsigned Where, unsigned Length, unsigned XmemSrcAddr)**

Initializes location and size of physical memory to be transferred to the PC.

PARAMETERS: **Where** is the address on the target where the data being transferred are placed.

Length is the maximum number of bytes to receive.

XmemSrcAddr is the physical memory source of the data to transfer.

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

Initializes location and size of logical memory to be transferred to the PC.

PARAMETERS: **Where** is the address on the target where the data being transferred are placed.

Length is the maximum number of bytes to receive.

- **int aascWriteXModem(CHANNEL Channel,
char Pak1K, char Initialize,
unsigned(*write_callback)())**

Performs XModem target-to-PC upload. Call this function once with **Initialize** set to 1. Then set **Initialize** to 0, and call this function repeatedly until its return value is non-zero.

Call **aascXMWrInitPhy()** for physical memory transfers or **aascXMWrInitLog()** for logical memory transfers before the first call to **aascReadXModem()**.

PARAMETERS: **Channel** is the logical channel being written to.

Pak1K is the XModem packet size.

0 - use 128 byte XModem packets

1 - use 1024 byte XModem packets

Initialize is the initialization flag. Set **Initialize** to one to initialize XModem on the first call. Set **Initialize** to zero for all subsequent calls.

write_callback is a pointer to a callback function that will be called by this function BEFORE each XModem packet is sent so that further processing can be performed on the data. The default functions **aascWrCallBackLg()** and **aascWrCallBackPh()** are provided for logical and physical memory transfers. See on-line help on these functions for further details.

RETURN VALUE:

XX_SUCCESS
XX_TIMEOUT
XX_COMMERR
XX_CANCEL
XX_NOSTART
XX_SYNC

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

Default callback function for addressing data for target-to-PC transfers. **aascWrCallBackPh** is called by **aascWriteXModem()** before sending a packet to the PC. **aascWrCallBackPh** advances the pointer to the target's memory by **PackSize** after each packet is sent. **aascWrCallBackPh** determines the address based on **PackSize** and **Packnum**.

PARAMETERS: **PackSize** is the packet size being used by XModem.

- 0 - use 128 byte XModem packets
- 1 - use 1024 byte XModem packets

PackNum is the number of the current packet.

RETURN VALUE: The address of the next location to transfer data from; 0 if the requested packet number exceeds the file size.

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

Default callback function for addressing data for target-to-PC transfers. **aascWrCallBackLg** is called by **aascWriteXModem()** before sending a packet to the PC. **aascWrCallBackLg** advances the pointer to the target's memory by **PackSize** after each packet is sent. **aascWrCallBackLg** determines the address based on **PackSize** and **Packnum**.

PARAMETERS: **PackSize** is the packet size being used by XModem.

- 0 - use 128 byte XModem packets
- 1 - use 1024 byte XModem packets

PackNum is the number of the current packet.

RETURN VALUE: The address of the next location to transfer data from; 0 if the requested packet number exceeds the file size.



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

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

- **int Dget_modem_command()**

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" <i>just a new line</i>



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()**

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()**

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 non-zero, 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 non-zero, 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_scdbuf(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_scbdbuf(void)**

Resets the receive buffer.

- **void Dreset_scbtbuf(void)**

Resets the transmit buffer and stops transmission.

- **void Drestart_scamodem(void)**

Restarts a modem during startup or because of abnormal operation in SCC port A.

- **void Drestart_schmodem()**

Restarts a modem during startup 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_scalch(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_scalch(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 non-zero, 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 Dschsend_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 non-zero, 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 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 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 startup 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 non-zero, 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 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 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()**
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 startup 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 non-zero, 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 exactly analogous to the functions in **Z0232.LIB**. Just replace the “0” in that library’s function name with “1” to get the corresponding function names in this library.



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 primarily 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 even 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 use 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 μ s 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** actually 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 primarily 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 **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 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: $4X[Reg] = (4X[Reg] \& And) | (Or \& \sim And)$. 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 Unavail-  
    able (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_NOGRESF 0x0B // Gateway Target Device  
    Failed to Respond (Modbus Plus)
```

MS_NOGRESF 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: **OTHER LIBRARIES***

The libraries described in Chapter 7 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)**

Serves 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()**

Serves 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()**

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()**

This is the five-key server for the “virtual” DAC channel. If the output value changes, this function will refresh the screen 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_uinput()**

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 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_diginput()**

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 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_bank1dig()**

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 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_bank2dig()**

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 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_14out()**

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 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.

CPLC.LIB

These functions support the PK2100 and PK2200 series controllers.

- **void uplc_init()**

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()**

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()**

Returns time since last key was pressed, in units of 1/40 second. The function returns elapsed time.

- **void lc_init_keypad()**

Initializes **timer1**, keypad driver and variables, and the real-time kernel.

- **void GLOBAL_INIT()**

Refer to **VDRIVER.LIB** for a description of this function.

- **int up_synctimer()**

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. The function 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()**

Enables high-voltage driver.

- **void hv_dis()**

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.

<code>\n</code>	Newline (position cursor to line 1, column 0)
<code>\xFF</code>	Clear screen
<code>\xF0</code>	Clear line 0
<code>\xF1</code>	Clear line 1
<code>\xF2</code>	Cursor OFF (cursor invisible, blink off)
<code>\xF3</code>	Cursor ON (solid cursor block)
<code>\xF4</code>	Cursor BLINK (blinks continuously)
<code>\xF5</code>	Shift display left
<code>\xF6</code>	Shift display right
<code>\x80–\xA7</code>	Position cursor at line 0
<code>\xC0–\xE7</code>	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()**

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()**

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()**

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()**

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 $\text{delay} = 3 \times (\text{msec}/4)/\text{freq1}$.

- **void int_timer1()**

timer1 interrupt service routine. Drives the beeper and keypad. Also runs the real-time kernel if **RUNKERNEL** is defined.

- **void save_shadow()**

Saves PLCBus shadow registers on the stack.

- **void restore_shadow()**

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()**

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()**

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 μ s.) The function returns 1 if the specified tick delay has lapsed. Otherwise, it returns 0.

- **int DelayMs(CoData *pfb, long delays)**

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. The function 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()**
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()**
Reverses the effect of powerdown so power stays on after external power is disabled. See **powerdown**.
- **void nmiint()**
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()**
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()**
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 DMA1_Off()

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_addr(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 **interrupts** 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)**. 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 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)**. Simply 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_addr(array)**. Simply 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()**

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 should be processed within 100 ms or they will be lost.

- `int fk_item_alpha(char *s1, char *var,
 int wdsiz)`

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 `%nu` 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 www.yy
```

where `vvvvv` is a prompt string, `www` is `n` chars long, and `yy` 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()**

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()**

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()**

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()**
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()**
Initializes **timer1**, KDM keypad driver and the graphic LCD.
- **int lk_init_keypad()**
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()**
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()**
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()**
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 I Turns cursor on
 esc O Turn cursor off
 esc c Erases from cursor position to end of line
 esc b Enables blinking cursor mode
 esc n Disables blinking cursor mode
 esc e Erases 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()**
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 $(j \bmod 1800)/30$.
- **int st_sec(unsigned int j)**
Seconds parser used by **lk_run_menu**. The function returns $2 \times (j \bmod 30)$.
- **unsigned int mk_st(int hour, int min, int sec)**
Time data builder used by **lk_run_menu**. The function returns $\text{hour} \times 1800 + \text{min} \times 30 + \text{sec} \times 2$.
- **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()**
Pulls character from key buffer and generates a short beep.
- **int lk_lecho()**
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()**
Gets a floating-point number from the keypad. The function returns the floating-point number entered through the keypad.
- **void lg_init()**
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()**

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 l Turns cursor on

esc O Turn cursor off

esc c Erases from cursor position to end of line

esc b Enables blinking cursor mode

esc n Disables blinking cursor mode

esc e Erases 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()**

Clears the graphic LCD text screen.

- **void Stall(int tix)**

Software delay loop. Counts down **tix** × 10.

- **void sta01()**

Writes 4 to the LCD write register and waits for a 3 on the LCD read register.

- **void sta03()**

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()**

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()**

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()**

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 are LCD support functions. They support the 2×20 LCD on all Z-World products that have 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()**
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()**
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()**
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()**
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()**

Loads eight special characters of arrows and lines to the LCD special character location.

- **void lcd_init_printf()**

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()**

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 ()**

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 ()**

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 ()**

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 **pppx000y000z**.

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 **pppx001y000z**.

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()**

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()**

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()**

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), $V_{OUT} = (\text{val}/4096) \times 10.22$ volts with Z-World default settings. In current-output mode (J1 pins 1-2 jumpered), $I_{OUT} = (\text{val}/4096) \times 22$ 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), $V_{OUT} = (\text{val}/4096) \times 10.22$ V with Z-World default settings. In current-output mode (J1 pins 1-2 jumpered), $I_{OUT} = (\text{val}/4096) \times 22$ mA with Z-World default settings. The board address must have been set previously with a call to **PBus12_Addr**.

- **void DAC_On()**

Controls the high-side switch activation line. Only used with switch option U10-LT1188.

- **void DAC_Off()**

Controls the high-side switch activation line. Only used with switch option U10-LT1188.

- **void Reset_PBus()**

Resets the PLCBus.

- **int Poll_PBus_Node(int addr)**

Polls a PLCBus device by performing a **BUSRDO** 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()**

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()**

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()**

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()**

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 **pppx000y000z**.

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 **pppx001y000z**.

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()**

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()**

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()**

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), $V_{OUT} = (\text{val}/4096) \times 10.22$ volts with Z-World default settings. In current-output mode (J1 pins 1-2 jumpered), $I_{OUT} = (\text{val}/4096) \times 22$ 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), $V_{OUT} = (\text{val}/4096) \times 10.22$ V with Z-World default settings. In current-output mode (J1 pins 1–2 jumpered), $I_{OUT} = (\text{val}/4096) \times 22$ mA with Z-World default settings. The board address must have been set previously with a call to **PBus12_Addr**.

- **void DAC_On()**

Controls the high-side switch activation line. Only used with switch option U10-LT1188.

- **void DAC_Off()**

Controls the high-side switch activation line. Only used with switch option U10-LT1188.

- **void Reset_PBus()**

Resets the PLCBus.

- **int Poll_PBus_Node(int addr)**

Polls a PLCBus device by performing a **BUSRDO** 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()**

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

Dynamic C's function libraries provide a way to bring in only those portions of system code that a particular program uses. The file **LIB.DIR** contains a list of all libraries known to Dynamic C. This list may be modified by the user. In particular, any library created by a user must be added to this list.

Libraries are "linked" with a user's application through the **#use** directive. Files identified by **#use** directives are nestable, as shown in Figure A-1.

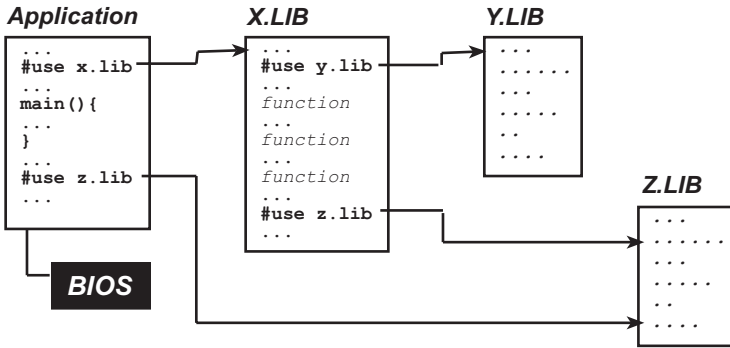


Figure A-1. Linking Nestable Files in Dynamic C

The file **DEFAULT.H** contains several lists of libraries to **#use**, one list for each product that Z-World ships. Dynamic C usually knows which controller is being used, so it selects the libraries appropriate to that controller. These lists are the *defaults*. A programmer may find it convenient or necessary to add or remove libraries from one or more of the lists.

The default libraries for a Z-World controller contain many function names, global variable names, and in particular, many macro names. It is likely that a programmer may try to use one of the Z-World names for a newly written program. Unpredictable problems can arise. Z-World recommends that **DEFAULT.H** be edited to comment out libraries that are not needed.

Table A-1 lists the libraries included with Dynamic C. Other libraries, **LSTAR.LIB**, **MICROG.LIB**, **LGIANT.LIB**, **RG.LIB**, **SCOREZ1.LIB**, **LPLC.LIB**, and **PS.LIB** exist only for backward compatibility.

Table A-1. Dynamic C Libraries

5KEY.LIB	Basic support for the “five-key system” for the PK2100 series and PK2200 series controllers.
5KEYEXTD.LIB	Extensions to the “five-key system.”
96IO.LIB	Driver functions for the BL100’s DGL96 daughter board.
AASC.LIB	Abstract Asynchronous Serial Communication functions.
AASCDIO.LIB	STDIO -specific routines supporting the AASC library.
AASCSCC.LIB	SCC-specific routines to support the AASC library. The SCC is the Zilog 85C30 Serial Communication Controller.
AASCUART.LIB	AASC functions for the XP8700 expansion board.
AASCURT2.LIB	AASC functions for the XP8700 expansion board used on controllers (BL1700) with 16-bit PLCBus addressing.
AASCZ0.LIB	Z0-specific routines to support the AASC library. Z0 is the Z180 ASCII Serial Port 0.
AASCZ1.LIB	Z1-specific routines to support the AASC library. Z1 is the Z180 ASCII Serial Port 1.
AASCZN.LIB	ZNet-specific routines to support the AASC library.
BIOS.LIB	Contains prototypes of functions and declarations of variables defined in, and used by, the BIOS.
BL11XX.LIB	Functions for the BL1100.
BL13XX.LIB	Functions for the BL1300.
BL14_15.LIB	Functions for the BL1400 series and BL1500 series controllers.
BL16XX.LIB	Functions for the BL1600.
CIRCBUF.LIB	Abstract data type functions for circular buffers (used by the AASC driver).
CM71_72.LIB	Functions for the CM7100 series and CM7200 series core modules.
COM232.LIB	Z104 communication functions.
CPLC.LIB	Functions for the PK2100, PK2200, and BL1600.

continued...

Table A-1. Dynamic C Libraries (continued)

DC.HH	This file contains definitions basic to, and required by, Dynamic C. This file is required.
DEFAULT.H	Contains lists of #use directives for various Z-World controllers. Dynamic C automatically selects the list appropriate for controller being programmed.
DMA.LIB	Support functions for the Z180 on-chip DMA (direct memory access) channels.
DRIVERS.LIB	Driver functions for some hardware devices.
EZIO.LIB	Driver functions for a board-independent unified I/O space.
EZIOBL17.LIB	Functions for the BL1700.
EZIOCMMN.LIB	Common definitions for all EZIO... libraries.
EZIODPWM.LIB	DMA pulse-width modulation functions used with the BL1700, PK2300, and PK2500.
EZIOLP31.LIB	Functions for the LP3100.
EZIOMGPL.LIB	Functions for the BL1400 and BL1500 PLCBus.
EZIOOP71.LIB	Functions for the OP7100.
EZIOBDV.LIB	PLCBus device drivers supporting the EZIO library.
EZIOPK23.LIB	Functions for the PK2300.
EZIOPK24.LIB	Functions for the PK2400.
EZIOPK25.LIB	Functions for the PK2500
EZIOPLC.LIB	PLCBus functions for boards that have native PLCBus ports (BL1200 series, BL1600 series, PK2100 series, and PK2200 series.
EZIOPLC2.LIB	PLBUS functions for boards (BL1700) that use 16-bit PLCBus addressing.
FK.LIB	New “five-key system” support for the PK2100 series and PK2200 series controllers. They are to be used with cooperative multitasking (i.e., costatements).
GLCD.LIB	Graphic LCD primitives for OP7100, PK2100, PK2200, and PK2400.
IOE.LIB	BL1100 expansion board libraries for single expansion board with default address.

continued...

Table A-1. Dynamic C Libraries (continued)

IOEXPAND.LIB	BL1100 expansion board addresses for expansion boards with non-default addresses.
KDM.LIB	Driver functions for Z-World keyboard/display modules.
LCD2L.LIB	Two-line LCD support for the PK2100 series and PK2200 series controllers.
LQVGA.LIB	Support library for landscape LCD (OP7100).
MATH.LIB	Useful mathematical and trigonometric functions.
MISC.LIB	Miscellaneous functions for KDM support.
MODEM232.LIB	Modem functions for the PK2100 series and PK2200 series controllers. Used with Z0232.LIB , S0232.LIB , XP87XX.LIB , NETWORK.LIB and SCC232.LIB .
NETWORK.LIB	Opto22 9-bit binary protocol to support master-slave networking. Uses ASCI port 1 of the Z180.
PBUS_LG.LIB	Functions that operate the BL1100 PLCBus.
PBUS_TG.LIB	Functions that operate the BL1000 PLCBus.
PK21XX.LIB	Functions for the PK2100.
PK22XX.LIB	Functions for the PK2200.
PLC_EXP.LIB	PLCBus functions for boards that have native PLCBus ports (BL1200 series, BL1600 series, PK2100 series, and PK2200 series).
PQVGA.LIB	Support library for portrait LCD (OP7100).
PRPORT.LIB	Functions that implement a parallel port communication protocol between a controller and a printer or printer port.
PWM.LIB	Pulse-width modulation functions for BL1700, PK2300, and PK2500.
RTK.LIB	Real-time kernel (RTK).
S0232.LIB	Serial communication driver for SIO port 0 on the BL1100 series controller.
S1232.LIB	Serial communication driver for SIO port 1 on the BL1100 series controller.

continued...

Table A-1. Dynamic C Libraries (concluded)

SCC232.LIB	Serial communication driver for the ports on the SCC chip, Zilog's 85C30 Serial Communication Controller.
SERIAL.LIB	Serial drivers.
SRTK.LIB	Simplified real-time kernel for all controllers.
STDIO.LIB	Functions relating to the STDIO window in Dynamic C.
STEP.LIB	Functions for XP8800 expansion board.
STEP2.LIB	Functions for second XP8800 expansion board.
STRING.LIB	This file contains functions for manipulating strings.
SYS.LIB	General system functions.
UART2.LIB	Functions for second XP8700 expansion board.
UART232.LIB	Functions for XP8700 expansion board.
UIBOARD.LIB	Functions for XP8200 expansion board used with BL1200 or PK2100.
VDRIVER.LIB	Virtual driver timer functions (for all controllers).
XMEM.LIB	Functions for moving information to and from extended memory, as well as other functions (such as address computation) related to extended memory.
Z0232.LIB	Serial communication driver for Z0. Z0 is the Z180 ASCI Serial Port 0.
Z104.LIB	Functions for Z104 and ZISA boards.
Z1232.LIB	Serial communication driver for Z1. Z1 is the Z180 ASCI Serial Port 1.
ZNPAKFMT.LIB	Lower level functions supporting the ZNet.



APPENDIX B: 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 B-1 lists the support libraries used with **AASC.LIB**.

Table B-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 <i>master controller</i> 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 B-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 B-1.

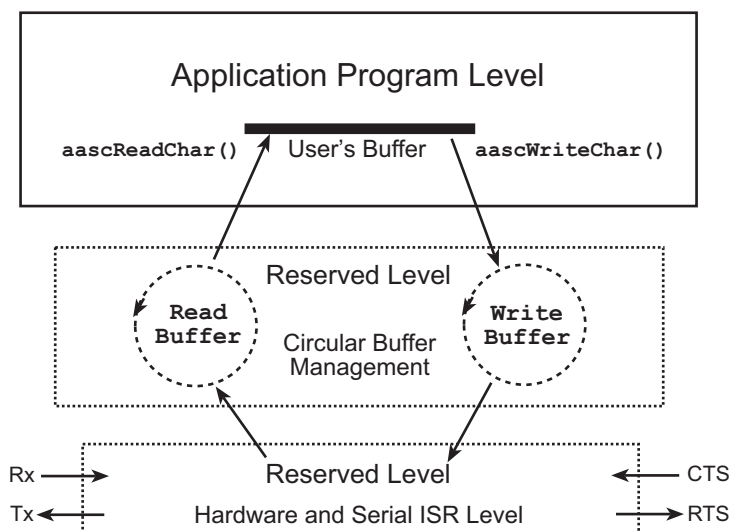


Figure B-1. Hierarchy of AASC Functions

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. **_GLOBAL_INIT()** 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.)

```

#use aasc.lib
#use aascscclib
#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.


```

#include <vdriver.lib>
#include <aasc.lib>
#include <aascscclib>
#define BUFSIZE 1024      // Size of circular buffer.
#define BAUDMULT 8        // multiples of 1200 bps
                          // (8 x 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.
            aascXMrInitLog( (unsigned) aascBuffer, 128);
            aascWriteXModem( aascChannel, 0, 1,
                aascWrCallBackLg );
            while( !aascWriteXModem( aascChannel, 0, 0,
                aascWrCallBackLg ) ) hitwd();
            printf("\n\nXmodem transfer finished...\n\r");
            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) );
}

```




*APPENDIX C: **LIBRARY***
LISTS FOR Z-WORLD PRODUCTS

BL1000

AASC.LIB, BIOS.LIB, DMA.LIB, DRIVERS.LIB, EZIO.LIB
TGiant.LIB

BL1100

AASC.LIB, BIOS.LIB, DMA.LIB, DRIVERS.LIB, EZIO.LIB
96IO.LIB
BL11XX.LIB

BL1200

AASC.LIB, BIOS.LIB, DMA.LIB, DRIVERS.LIB, EZIO.LIB

BL1300

AASC.LIB, BIOS.LIB, DMA.LIB, DRIVERS.LIB, EZIO.LIB
BL13XX.LIB
DRIVERS.LIB
PRPORT.LIB
SERIAL.LIB

BL1400

AASC.LIB, BIOS.LIB, DMA.LIB, DRIVERS.LIB, EZIO.LIB
BL14_15.LIB

BL1500

AASC.LIB, BIOS.LIB, DMA.LIB, DRIVERS.LIB, EZIO.LIB
BL14_15.LIB

BL1600

AASC.LIB, BIOS.LIB, DMA.LIB, DRIVERS.LIB, EZIO.LIB
BL16XX.LIB

BL1700

AASC.LIB, BIOS.LIB, DMA.LIB, DRIVERS.LIB, EZIO.LIB
EZIOBL17.LIB

LP3100

EZIOLP31.LIB

PK2100

5KEY.LIB

5KEYEXTD.LIB

AASC.LIB, BIOS.LIB, DMA.LIB, DRIVERS.LIB, EZIO.LIB

PK21XX.LIB

PK2200

AASC.LIB, BIOS.LIB, DMA.LIB, DRIVERS.LIB, EZIO.LIB

PK2300

AASC.LIB, BIOS.LIB, DMA.LIB, DRIVERS.LIB, EZIO.LIB

PK2400

AASC.LIB, BIOS.LIB, DMA.LIB, DRIVERS.LIB, EZIO.LIB

PK2500

AASC.LIB, BIOS.LIB, DMA.LIB, DRIVERS.LIB, EZIO.LIB

PK2600

AASC.LIB, BIOS.LIB, DMA.LIB, DRIVERS.LIB, EZIO.LIB

EZIOBL17.LIB

EZIOOP71.LIB, KP_OP71.LIB, LQVGA.LIB, PQVGA.LIB

OP71HW.LIB, OP71L.LIB, OP71P.LIB

CM7100

AASC.LIB, BIOS.LIB, DMA.LIB, DRIVERS.LIB, EZIO.LIB

CM71_72.LIB

CM7200

AASC.LIB, BIOS.LIB, DMA.LIB, DRIVERS.LIB, EZIO.LIB

OP7100

AASC.LIB, BIOS.LIB, DMA.LIB, DRIVERS.LIB, EZIO.LIB
EZIOOP71.LIB, KP_OP71.LIB, LQVGA.LIB, PQVGA.LIB
OP71HW.LIB, OP71L.LIB, OP71P.LIB

Symbols

_5key_12out	117
_5key_14out	118
_5key_bank1dig	118
_5key_bank2dig	118
_5key_boolean	113
_5key_dacout	117
_5key_date	114
_5key_diginput	118
_5key_float	112
_5key_init_item	115
_5key_integer	113
_5key_menu	116
_5key_server	116
_5key_setalarm	116
_5key_setfunc	117
_5key_setmenu	115, 116
_5key_setmsg	117
_5key_time	114
_5key_uinput	118
_5keygetdate	112
_5keygettime	112
_5keysetdate	112
_5keysettime	112
_ALARM1 ... _ALARM4	116
_GLOBAL_INIT	12, 80
_pow10	21
_prot_init	33
_prot_recover	33
5KEY.LIB	112, 157
5KEYEXTD.LIB	117, 157
96IO.LIB	157
9th-bit binary communication	57

A

a24_32	35
a32_24	35
AASC libraries	44, 162
buffer sizes	162
callback functions	167
description	162
operations	163
peek	164
read	163
write	164
sample programs	164
status and errors	164
use	164
XModem transfer	167
AASC.LIB	44, 157, 162
aascClearError	48
aascClose	45
AASCDIO.LIB	157, 162
aascFlush	49
aascFlushRdBuf	49
aascFlushWrBuf	49
aascGetError	48
aascOpen	44, 45
aascPeek	48, 164
aascPipe	48
aascPrintf	49
aascRdCBackLocLg	51, 52
aascReadBlk	47
aascReadBufFree	49
aascReadBufLeft	48
aascReadChar	46
aascReadXModem	51
aascRxSwitch	46
aascScanTerm	48, 164
AASCSCC.LIB	157, 162

aascSetReadBuf	45
aascSetWriteBuf	46
aascTxSwitch	46
AASCUART.LIB	157, 162
AASCURT2.LIB	157, 162
aascVPrintf	50
aascWrCallBackLg	54
aascWrCallBackPh	54
aascWriteBlk	47
aascWriteBufFree	49
aascWriteBufLeft	49
aascWriteChar	47
aascWriteXModem	53
aascXMRdInitLog	51
aascXMRdInitPhy	50
aascXMWrInitLog	53
aascXMWrInitPhy	52
AASCZ0.LIB	157, 162
AASCZ1.LIB	157, 162
AASCZN.LIB	157, 162
abs	18
acos	18
acot	19
acsc	19
ad_st	141
ad20_cal	134
ad20_cal_n	136
ad20_mux	134
ad20_mux_n	136
ad20_rd	135
ad20_rd_n	137
ad20_rdy	134
ad20_rdy_n	136
ADD key	113, 114, 117, 118
adelay_50ms	57
alarm functions	
five-key system	116
asec	19
asin	19
atan	19
atan2	19
atof	27
atoi	27
atol	27

B

baud rate	57, 62, 63, 71, 75
Begin Reply Time Out	95
bfree	31
BIOS.LIB	12, 157
bit	13
BL1100	133, 147
setperiodic	127
BL11XX.LIB	157
BL13XX.LIB	157
BL14_15.LIB	157
BL16XX.LIB	157
Boolean parameters	
five-key system	113
buffer	
receive	
56, 57, 62, 63, 65, 67, 72, 76	
initialization ..	62, 63, 71, 75
reading	65, 67, 72, 76
transmit	63, 72, 76
initialization ..	62, 63, 71, 75
writing	
64, 65, 66, 67, 71, 72, 75, 76	

C

calloc	31
ceil	19
changing parameters with the five-	
key system	
112, 113, 114, 117, 118	
check sum	
computing	22
check_opto_command	56
checking for modem commands	
65, 67, 73, 77	
checksum	61
CIRCBUF.LIB	157, 162
Clear_Gr_Screen	144
Clear_GrTxt_Screen	142
clink_init	60
clock	123
CM71_72.LIB	157
CoBegin	16

CoData	16, 17	disabling interrupts	13, 56
COM232.LIB	157	DMA channels	56
comp48	38	Z180 serial channels 0 and 1 ..	56
CoPause	17	disabling the RS-485 driver	58
CoReset	16	Dkill_s0	73
CoResume	17	Dkill_z0	76
cos	19	DMA channels	
cosh	19	disabling interrupts	56
CPLC.LIB	119, 157	DMA.LIB	127, 158
CRC	56, 57	DMA0	126
computing	22	DMA0_IOM	130
CSIO	56	DMA0_MIO	130
CTS	62, 63, 71, 72, 75	DMA0_MM	129
CTS/RTS		DMA0_Off	128
XP8700	62, 63, 75	DMA0_Rx	128
cyclic redundancy check	56, 57	DMA0_SerialInit	128
computing	22	DMA0_Tx	129
D		DMA0Count	127
DAC output	117	DMA1	126
DAC_Board_Addr	149, 152	DMA1_IOM	131
DAC_Off	150, 153	DMA1_MIO	130
DAC_On	150, 153	DMA1_Off	128
data types		DMA1Count	128
five-key system ...	112, 113, 114	dmacopy	123
date and time	122	DMASnapShot	126, 128
date parameters		doint	122
five-key system	114	doprnt	24
DC.HH	158	doreti	59
deciphering modem commands ..	56	downloading data	65, 73, 77
DEFAULT.H	158	Draw_Axis	144
Define_Cursor	143	Draw_Line	144
deg	19	Draw_Poly	144
DelayMs	92, 125	Dread_s0	72
DelaySec	126	Dread_s0lch	72
DelayTicks	125	Dread_sca	65
DELETE key ..	113, 114, 117, 118	Dread_scalch	65
Dget_modem_command	56	Dread_scb	67
di	13	Dread_scb1ch	67
digital input	118	Dread_z0	76
Dinit_s0	71	Dread_z0lch	76
Dinit_sca	62	Dreset_s0rbuf	72
Dinit_scb	63	Dreset_s0tbuf	72
Dinit_z0	75	Dreset_scarbbuf	62
		Dreset_scatbuf	63

Dreset_scrbuf	63
Dreset_scrtbuf	63
Dreset_z0rbuf	76
Dreset_z0tbuf	76
Drestart_s0modem	73
Drestart_scamodem	64
Drestart_scbmodem	64
Drestart_z0modem	77
driver	
virtual	117, 118
DRIVERS.LIB	120, 158
Ds0_circ_int	73
Ds0modem_chk	73
Ds0send_prompt	71
Dscamodem_chk	65
Dscasend_prompt	64
Dscbmodem_chk	67
Dscbsend_prompt	66
Dwrite_s0	72
Dwrite_s0lch	72
Dwrite_sca	64
Dwrite_sca1ch	65
Dwrite_scb	67
Dwrite_scb1ch	67
Dwrite_z0	76
Dwrite_z0lch	76
Dxmodem_s0down	73
Dxmodem_s0up	73
Dxmodem_scadown	65
Dxmodem_scaup	66
Dxmodem_scbdown	68
Dxmodem_scbup	68
Dxmodem_z0down	77
Dxmodem_z0up	78
Dynamic C	
list of function libraries	
157, 158, 159, 160	
Dz0_circ_int	77
Dz0modem_chk	77
Dz0send_prompt	75

E

echo option	62, 63, 71, 75
ee_rd	13
ee_wr	13
eei_rd	126
eei_wr	126
ei	13
enable interrupts	13
End Reply Time Out	95
exit	16
exp	20
exp_init	134
exp_init_n	135
extended five-key service functions	
117, 118	
EZIO.LIB	158
EZIOBL17.LIB	158
EZIOCMMN.LIB	158
EZIODPWM.LIB	158
EZIOLP31.LIB	158
EZIOMGPL.LIB	158
EZIOOP71.LIB	158
EZIOPBDV.LIB	158
EZIOPK23.LIB	158
EZIOPK24.LIB	158
EZIOPK25.LIB	158
EZIOPLC.LIB	158
EZIOPLC2.LIB	158

F

F1, F2, F3, F4	117
fabs	20
fastblock	61
fastcall	41
five-key programming	
112, 113, 114, 115, 116, 117, 118	

five-key system	112, 113, 114, 115, 116, 117, 118
ADD key	113, 114, 117, 118
alarm functions	116
Boolean parameters	113
changing parameters	
112, 113, 114, 117, 118	
data types	112, 113, 114
date parameters	114
DELETE key	
113, 114, 117, 118	
extended service functions	
117, 118	
float parameters	112
function keys	117
integer parameters	113
ITEM key ...	113, 114, 117, 118
MENU key .	113, 114, 117, 118
monitoring parameters	
112, 113, 114, 117, 118	
parameter list	116
adding items	115
parameters	
Boolean	113
date	114
float	112
integer	113
time	114
service functions	116, 117
string messages	117
time parameters	114
FK.LIB	131, 158
fk_helpmsg	131
fk_item_alpha	132
fk_item_enum	133
fk_item_int	132
fk_item_setdate	133
fk_item_settime	133
fk_item_uint	132
fk_monitorkeypad	131
FKSAMP.C	131
float parameters	
five-key system	112
floating-point functions	18
floor	20
fmod	20
formatting convention	26
free	31
frexp	20
ftoa	25
function chain	12
function keys	
five-key system	117
function libraries	
list	157, 158, 159, 160
G	
get_def_na	135
get_na	135
getchar	22
getcrc	22
gets	22
gettimer	38
getvect	32
GLCD.LIB	158
GLOBAL_INIT	120
Graph_Init	143
grp_home_area	143
gtoa	24
gtoan	24
H	
Hayes compatible modem	56
Hayes Smart Modem	56
high-current output	117, 118
High-Resolution Timer	89
hitwd	15
hltoa	26
hrtInit	89
hrtRead	89
htoa	26
hv_dis	121
hv_enb	121
hv_wr	121

I

IBIT	14
iff	13, 32
Init_DAC	150, 153
init_kernel	39
init_srtkernel	40
init_timer	127
init_timer0	123
init_timer1	124
initialization	
receive buffer	62, 63, 71, 75
transmit buffer...	62, 63, 71, 75
Z180 Port 1	57
initiating	
serial transmission	
64, 65, 67, 72, 76	
input	
digital	118
RS-232	65, 67, 72, 76
universal	118
int_timer1	124
INT1	56
INT2	56
integer parameters	
five-key system	113
interrupt service routines	73, 77
interrupts	
disabling	56
intoff	122
inton	122
intrmode	15
intrmode_1	15
intrmode_2	15
IOE.LIB	158
IOEXPAND.LIB	133, 159
IRES	15
isalnum	18
isalpha	18
isctrl	18
isCoDone	17
isCoRunning	17
isdigit	17
ISSET	14

isgraph	18
islower	17
isprint	18
ispunct	17
isspace	18
isupper	17
isxdigit	17
IsZ80180	33
ITEM key	113, 114, 117, 118
itoa	26

K

kbhit	16
KDM.LIB	137, 159
kernel	
real-time	116

L

labs	20
Latch_DAC1	149, 153
Latch_DAC2	149, 153
lc_cgram	146
lc_char	145
lc_cmd	145
lc_ctrl	145
lc_init	145
lc_init_keypad	119
lc_kxget	119
lc_kxinit	119
lc_loadtab	119
lc_nl	145
lc_pos	146
lc_printf	146
lc_putc	145
lc_rd	145
lc_setbeep	119
lc_settab	119
lc_stdcg	146
lc_wait	145
lc_wr	145
lcd_clr_line	121
lcd_erase	146
lcd_erase_line	146

lcd_init	121	lk_settab	137
lcd_init_printf	146	lk_settime	141
lcd_printf	146	lk_showdate	140
lcd_putc	146	lk_showtime	141
lcd_resscrn	147	lk_stdcg	139
lcd_savscrn	147	lk_tdelay	138
lcd_server	112	lk_viewl	141
lcd_wait	121	lk_wait	138
LCD2L.LIB	145, 159	lk_wr	138
ldexp	20	log	20
lg_char	141	log10	20
lg_init	141	longjmp	31
lg_init_keypad	138	lprintf	122
lg_nl	142	lputc	121
lg_pos	142	lputs	122
lg_printf	142	LQVGA.LIB	159
lg_putc	142	ltoa	24
lg_rd	143	ltoan	24
lg_wr	143		
lg_wr03	143	M	
lk_cgram	139	malloc	31
lk_char	139	Map_Bit_Pattern	144
lk_chkdat	140	master message format	56
lk_cmd	138	master-slave serial communication	
lk_ctrl	139	56	
lk_getknum	141	math functions	18
lk_init	138	MATH.LIB	16, 17, 18, 159
lk_init_keypad	138	memchr	29
lk_int_timer1	138	memcmp	30
lk_keyw	137	memcpy	29
lk_kxget	138	memory	
lk_kxinit	137	extended	
lk_lecho	141	and uploaded data	
lk_led	138	66, 68, 73, 78	
lk_loadtab	137	memset	27
lk_nl	139	MENU key	113, 114, 117, 118
lk_pos	139	MISC.LIB	159
lk_printf	139	mk_st	141
lk_putc	139	mktime	123
lk_rd	138	mktm	123
lk_run_menu	140	MM.LIB	92, 94
lk_secho	141	mmaInit	94
lk_setbeep	138	mmaZ0	93
lk_setdate	140		

mmaZ1	93	MODEM232.LIB	56, 159
mmCRC	94	modf	20
mmExec	94	Modicon, Inc.	79, 91
mmFetchCommCnt	103	website	90, 110
mmForceCoils	94	monitoring parameters with the	
mmInput	94	five-key system	
mmLRC	94	112, 113, 114, 117, 118	
mmRdExcStat	104	MS.LIB	80, 83
mmRecv	94, 104	MS_TIMER	88
mmrInit	94, 95	msaInit	87
mmrZ0	93	msaZ0	81
mmrZ1	93	msaZ1	81
mmSend	94, 104	msDone	85
MMZ.LIB	92, 94	msError	88
Modbus		msIn	83, 85
ASCII protocol	79, 91	msInput	86
RTU protocol	79, 91	msOut	83
Modbus Master		msOutRd	86
Advanced Procedure	94	msOutWr	86
Command Functions	94, 96	msRead	83, 86, 87
Command Return Values	106	msRecv	88
Getting Started	92	msrInit	87
Standard Procedure	92	msRun	82
Supported Commands	105	msrZ0	81
Timeouts	95	msrZ1	81
Unsupported Commands	105	msSend	88
Modbus Registers	84, 95	msStart	85
Modbus Slave		msTimer	88
Advanced Procedure	83	msWrite	87
Command Handlers	83, 85	MSZ.LIB	80, 89
Getting Started	80	mux_ch	134
High-Resolution Timer	89	mux_ch_n	136
Serial Interface	87		
Standard Procedure	80	N	
Supported Commands	89	NETWORK.LIB	56, 159
Unsupported Commands	90	nmiint	127
modem		NO_FUNCTION	116, 117
commands		number of bits	62, 63, 71, 75
deciphering	56		
communication		O	
56, 62, 63, 71, 75		off_485	127
checking for commands		on_485	127
65, 67, 73, 77		op_init_z1	57
restarting	73, 77		

op_kill_zl	58	plink_getc0	59
opto 22 binary protocol	56, 57, 58	plink_init0	59
outchrs	27	plink_intr0	60
output	13	plink_rdy0	59
outputn	123	plinki0	60
output		plint	25
DAC	117	Poll_PBus_Node	150, 154
high-current	117, 118	poly	21
relay	117, 118	pow	21
RS-232		pow10	21
64, 65, 66, 67, 71, 72, 75, 76		powerdown	126
RS-485	57	powerlo	16
outstr	27	powerup	127
P		PQVGA.LIB	159
pack	31	printf	23, 24
parameters		field codes	23
five-key system		programming	
Boolean	113	five-key	
date	114	112, 113, 114, 115, 116, 117, 118	
float	112	PRPORT.LIB	58, 159
integer	113	prsend0	58
time	114	prsend0_init	58
parity	62, 63, 71, 75	prsend1	58
PBUS_LG.LIB	147, 159	prsend1_init	58
PBUS_TG.LIB	151, 159	PRT	56
PBus12_Addr	147, 151	putc	23
PBus4_Read0	148, 151	putchar	22
PBus4_Read1	148, 151	puts	22
PBus4_ReadSp	148, 151	PWM.LIB	159
PBus4_Write	148, 151	Q	
pflt	26	qsort	32
phex	25	R	
phy_adr	66, 68, 73, 78, 123	rad	19
pint	25	read12data	120
pioint	61	read24data0	124
piolatch	59	read24data1	125
PK21XX.LIB	159	read4data	120
PK22XX.LIB	159	read8data0	124
PLC_EXPLIB	159	read8data1	125
plcbus_isr	125	readireg	32
plcport	120	real-time kernel	116
plhex	25		

realloc	32	s0modemset	71
receive buffer		s0modemstat	71
56, 57, 62, 63, 65, 67, 72, 76		S1232.LIB	74, 159
initialization	62, 63, 71, 75	sample programs	
reading	65, 67, 72, 76	echo transmission	166
relay output	117, 118	five-key system	131
Relay_Board_Addr	148, 152	XModem data transfer	168
reload_vec	33	save_shadow	124
relocate_int1	125	scabinaryreset	64
replyOpto22	57	scabinaryset	64
request	38	scamodemset	64
RES	14	scamodemstat	64
Reset_PBus	150, 154	scbbinaryreset	66
Reset_PBus_Wait	150, 154	scbbinaryset	66
resetZ180int	56	scbmodemset	66
restarting modem communication		scbmodemstat	66
73, 77		SCC port A	62, 63
restore_shadow	124	SCC port B	63
rkernl	39	SCC232.LIB	62, 160
root2xmem	34	sccint	64
RS-232 serial communication		sendfast	61
64, 65, 66, 67, 71, 72, 75, 76		sendOp22	56
CTS/RTS control	62, 63, 75	ser_init_s0	70
serial input	65, 67, 72, 76	ser_init_s1	70
serial output		ser_init_z0	69
64, 65, 66, 67, 71, 72, 75, 76		ser_init_z1	68
RS-485 serial communication		ser_kill_s0	70
57, 58		ser_kill_s1	70
disabling serial driver	58	ser_kill_z0	70
serial output	57	ser_kill_z1	69
RTK.LIB	38, 159	ser_rec_s0	70
RTS	62, 63, 71, 72, 75	ser_rec_s1	70
run_after	38	ser_rec_z0	70
run_at	38	ser_rec_z1	69
run_cancel	38	ser_send_s0	70
run_every	38, 39	ser_send_s1	70
run_timer	38	ser_send_z0	69
runwatch	16	ser_send_z1	69
S		serial communication	57
S0232.LIB	70, 159	master-slave	56
s0binaryreset	70	RS-232	
s0binaryset	70	64, 65, 66, 67, 71, 72, 75, 76	
		RS-485	57, 58
		master-slave	56

serial transmission		sta03	142
initiating	64, 65, 67, 72, 76	Stall	142
terminating	63, 72, 76	STDIO.LIB	160
SERIAL.LIB	68, 160	STEP.LIB	160
service functions		STEP2.LIB	160
five-key system	117, 118	stop bits	62, 63, 71, 75
SET	14	strcat	28
Set_Auto_Mode	143	strchr	28
Set_DAC1	150, 153	strcmp	28
Set_DAC2	150, 153	strcpy	27
set_def_na	135	strcspn	28
Set_Display_Mode	142	string messages in the five-key	
Set_Overlap_Mode	143	system	117
Set_PBus_Relay	148, 152	STRING.LIB	27, 160
Set_Pixel	144	strlen	29
Set_Pointer	143	strncat	28
set12adr	120	strncmp	28
set16adr	120	strncpy	27
set24adr	125	strpbrk	28
set4adr	120	strrchr	28
set8adr	125	strspn	28
setbeep	147	strstr	30
setireg	32	strtod	29
setjmp	30	strtok	29
setperiodic	127	strtol	29
BL1100	127	struct tm	122
setvect	32	suspend	39
setwaits	59	swap	31
sin	21	SYS.LIB	30, 33, 160
Sin_Wave	144	sysclock	16
sinh	21		
SIO port 0	71, 72, 73	T	
sizeof	112, 113, 114, 115	tan	21
slave response format	57	tanh	21
sleep	127	tdelay	124
sprintf	22	Text_Addr	143
sqrt	21	text_home_area	143
SRTK.LIB	40, 160	time and date	122
srth_hightask	40	time parameters	
srth_lowtask	40	five-key system	114
st_hour	141	Timeouts	95
st_min	141	timer0_isr	123
st_sec	141	timers	
sta01	142	PRT	56

tm 122
tm_rd 122
tm_wr 122
tolower 17
toupper 17
transmission
 initiating 64, 65, 67, 72, 76
transmit buffer 63, 72, 76
 initialization 62, 63, 71, 75
 writing
 64, 65, 66, 67, 71, 72, 75, 76
trigonometric functions 18

U

UART2.LIB 160
UART232.LIB 160
UIBOARD.LIB 160
universal input 118
up_beep 119
up_beepvol 119
up_lastkey 119
up_synctimer 120
uplc_init 80, 88, 92, 119
uploading data 66, 68, 73, 78
utoa 26

V

vd_initquickloop 41
VdAdjClk 41
VdGetFreeWd 41
VdInit 40, 80, 88, 92
VdReleaseWd 41
VDRIVER.LIB 40, 160
VdWdogHit 41
virtual driver 15, 117, 118
virtual I/O driver 117

W

waitfor 94
wderror 15
Write_DAC1 149, 152

Write_DAC2 149, 153
write12data 121
write24data 124
write4data 121
write8data 124

X

x_makadr 34
xdata 66, 68, 73, 78
xgetchar 33
xgetfloat 34
xgetint 34
xgetlong 34
xmadr 33
XMEM.LIB 33, 160
xmem2root 34
XModem data transfer
 packet structure 50
 protocol 65, 66, 68, 73, 77, 78
 sample program 168
xputchar 34
xputfloat 34
xputint 34
xputlong 34
xstrlen 34

Z

Z0232.LIB 74, 94, 160
z0binaryreset 74
z0binaryset 74
z0modemset 75
z0modemstat 74
Z104.LIB 160
Z1232.LIB 78, 94, 160
Z180 13
 port 0 75, 76, 77
 port 1 58, 68
 initialization 57
 serial channels 0 and 1
 disabling interrupts 56
ZNPAKFMT.LIB 160



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