for Zilog Z180 microprocessors

Version 6.x

Integrated C Development System

Application Frameworks

019-0081 • 020330-B

Dynamic C 32 v. 6.x Application Frameworks

Part Number 019-0081 • 020330 - B • Printed in U.S.A.

Copyright

© 2002 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.
- 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: zworld@zworld.com

Table of Contents

About This Manual vii

Chapter 1: Introduction 11

Chapter 2: Real-Time Programming 13

Interrupt Latency 14
Causes 14
Nested Interrupts 15
Multitasking 16
Cooperative Multitasking 16
Preemptive Multitasking 16
Tips for Multitask Programming 18
Assign Priority Levels 18
Identify Shared Variables 18

Chapter 3: Costatements 21

Solving Problems with Costatements 22
Solving Problems Without Costatements 23
Solving the Problem With Costatements 23
Summary 24
Costatement Functions 25
Syntax 25
State 25
The CoData Structure 25
The Implementation of Costatements 27
The firsttime Flag and firsttime Functions 28
Associated Keywords and Functions 29
Delay Functions 29
Other Functions 30
Costatement Topics 30
Timing Issues 30
Dependency Relations Among Costatements 32
Nesting Costatements 33
Error Exits from Costatements 33
Detecting Errors In waitfor Statements 35
waitfor and Problems with Evaluating C Expressions 35

Chapter 4: The Virtual Driver 37

Periodic Timer Interrupts 38
Calling Sequence 39
The RTK Driver 39
The Fastcall Driver 39
Virtual Watchdog Timers 40
Global Initialization 40

Chapter 5: Real-Time Kernels 41

The Simplified Real-Time Kernel 42
The Real-Time Kernel 43
Kernel Functions 46
Restrictions on the Use of Suspend 47
RTK Internals 48

Chapter 6: Failure and Recovery 51

Software Failures 52
Hardware Failures 52
Hardware Watchdog Timer 52
Reset and Super-Reset 53
Recovery of Protected Variables 54

Chapter 7: The Five-Key System 55

Operator Features 56
Programmer Features 57
Special Key Combinations 59
Five-Key Functions 59
Reading the Keypad without FK.LIB 61

Chapter 8: RS-232 Communication 63

Serial Communication 64
Send and Receive Buffers 64
Echo Option 64
CTS/RTS Control 65
XMODEM File Transfer 65
Modem Communication 65

Function Libraries and Sample Programs 66
Software Support 66
RS-232 Software 67
XMODEM Commands 70
Miscellaneous Functions 71
RS-485 Drivers 73
Libraries 73
Sample Programs 74

Chapter 9: Master-Slave Networking 75
Communication Protocol 76
Hardware Connection 76
Software Support 77
Miscellaneous Functions 78
Libraries and Sample Programs 81
Libraries 81
Sample Programs 81

Appendix A: Execution Speed 83

Appendix B: Old 5-Key System 87
Code-Driven Approach 88
Linked-List Approach 89
Updating and Monitoring Parameters 90
Monitoring Function Keys 91
Monitoring Help Keys 91
Periodic Display 91
Software Alarms 92
5-Key Support Functions 92

Index 95

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

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

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

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

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

Assumptions

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

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

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

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

- Knowledge of basic Z180 assembly language and architecture.

For documentation from Zilog, refer to the following texts.

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

Acronyms

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

Table 1. Acronyms

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

Icons

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

Table 2. Icons

Icon	Meaning	Icon	Meaning
	Refer to or see		Note
	Please contact	Tip	Tip
	Caution		High Voltage
FD	Factory Default		

Conventions

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

Table 3. Typographical Conventions

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

CHAPTER 1: INTRODUCTION

Dynamic C combines a C compiler, an editor, and a source-level debugger. The Dynamic C development system runs on IBM or compatible PCs. The target controller connects to a serial port on the PC, and typically communicates at 19,200 to 57,600 baud.

Dynamic C allows the user to develop software on the target controller interactively. Dynamic C includes many useful features, such as the following.

- Direct compilation to the target controller's memory. Compilation, linking and downloading occur simultaneously.
- Compilation to File. EPROM files, or downloadable files that work with the Z-World Download Manager, are easy to create with Dynamic C.
- Embedded assembly language. It is possible to write entire functions in assembly language, or include portions of assembly language in a C program. Assembly language is sometimes important for performance in time-critical programs.
- Z-World has made innovative extensions to the C language. The costatement makes cooperative multitasking very convenient. The **`shared`** keyword makes access to a variable atomic, so that the integrity of the variable can be maintained among different tasks. The **`protected`** keyword causes the compiler to keep a backup copy of the variable during modification so that it can be reconstructed after a catastrophe such as a power failure. Interrupt service routines can be written in C. These, and other, extensions to C simplify the task of building robust systems.
- Z-World software libraries support real-time programming, networking, and serial communication.

CHAPTER 2: REAL-TIME PROGRAMMING

Programmers should be aware of certain issues regarding real-time programming. Real-time systems thrive on interrupts and interrupt processing.

Chapter 2 discusses the following topics.

- Interrupt Latency
- Multitasking

Interrupts can occur at any time, in particular, in the middle of a write to multibyte variables (anything except **char**). The variable value can be indeterminate. Declare variables accessible in interrupt service routines as **shared** to prevent errors.

There is no operating system to handle interrupts in a real-time program that runs stand-alone on a controller. Furthermore, real-time applications require far more than a generic response to interrupts.

Dynamic C makes writing interrupt routines just as easy as writing any C functions, but this does not mean that real-time software is easy to develop. A major issue for the real-time programmer is *interrupt latency.*

Interrupt Latency

Interrupt latency is usually defined as the worst-case delay between an interrupt request and the start of the interrupt service. Typical values for this delay are about 100 microseconds.

Low interrupt latency is necessary when speed is important, or when it is important that an interrupt not be missed, such as a periodic timer interrupt.

Causes

A delay between an interrupt request and the start of the interrupt service can occur for several reasons.

1. The current instruction must finish executing. This is never more than 14 clock periods on the Z180.
2. The return address must be pushed on the stack. This is about 10 more clock periods.
3. Atomic store and read operations associated with **shared** and **protected** variables result in interrupts being disabled for about 10 microseconds on a 9-MHz processor. (Atomic structure assignments, of course, can cause longer delays.)

The above-mentioned delays are relatively minor since they are simply the overhead from processing interrupts. More serious delays occur when interrupts are disabled when an interrupt is requested. Interrupts can be disabled, for example, during critical sections of code. Interrupt processing itself disables interrupts until the interrupt service routine reenables them or exits.

If software turns off interrupts for 25 microseconds and an interrupt request happens just as the interrupts are disabled, the interrupt service will be delayed for an additional 25 microseconds.

A running program will fail if the interrupt latency is too great. For example, if the serial port is receiving characters every 173 microseconds (57,600 baud), then an interrupt latency of 2×173 µs = 347 microseconds will cause a lost character because the serial port can only hold two characters before it loses one.

Interrupt routines are usually most efficient when written in assembly language, but interrupt routines can be written in Dynamic C at the expense of a minimum overhead of about 25 microseconds on a 9-MHz processor, which is the time necessary to save the registers. If the service routine leaves interrupts off for its duration, then it is necessary to add another 30 microseconds to the latency calculation, to allow for the return from interrupt, plus the amount of time it takes to execute the routine. The advantage of assembly-language routines is that they need only to save the registers that they use, while the C interrupt routine assumes that everything possible must be saved. Assembly language also has more direct access to the machine registers.

Nested Interrupts

With sufficient care, it is possible to reenable interrupts inside an interrupt routine. This lessens the interrupt latency because interrupts do not stay disabled for the duration of the interrupt handler.

In general, the cause of a type *X* interrupt should be removed before interrupts are reenabled for the type *X* interrupt. When this is done, one interrupt routine can interrupt another interrupt routine. For example, a serial port interrupt routine on the Z180 can reenable interrupts in general while disabling the serial port interrupt specifically. The serial port interrupt can then be reenabled upon returning from the interrupt routine. However, such a practice can create difficulties, particularly if a nested interrupt lasts for a long time. There is no way to get back to the original interrupt service until the nested interrupt is complete, at which point some data may be lost.

A logical solution in this situation is to establish a software flag that prevents a lengthy task from interrupting a service routine. Were this idea taken to extremes, there would eventually be an operating system where every interrupt is logged by the operating system and then dispatched in a priority order. Z-World provides two real-time kernels to help the user solve these problems.

Multitasking

A *task* is an ordered list of operations to perform. In a multitasking environment, more than one task (each representing a sequence of operations) can *appear* to execute in parallel. In reality, a single processor can only execute one instruction at a time, so the parallel tasks execute *almost* in parallel.

If an application has multiple tasks to perform, multitasking software can usually take advantage of natural delays in each task to increase the overall performance of the system. Each task can do some of its work while the other tasks are waiting for an event, or for something to do.

Although multitasking may actually decrease processor throughput slightly, it is an important concept. A controller is often connected to more than one external device. A multitasking approach makes it possible to write a program controlling multiple devices without having to think about all the devices at the same time. In other words, multitasking is an easier way to think about the system.

There are two types of multitasking available for developing applications in Dynamic C: *preemptive* and *cooperative*.

Cooperative Multitasking

In a cooperative multitasking environment, each task voluntarily gives up control so other tasks can execute. A kernel is not required in this case because the tasks cooperate among themselves. If periodic task scheduling or time delays are not required, a cooperative multitasking environment does not require a timer interrupt.

The following advantages are offered by cooperative multitasking.

- Much easier communication between tasks.
- Greater predictability of mutual task interaction.
- Much simplified programming.

Dynamic C has a language extension called the ***costatement*** to support cooperative multitasking. In essence, a costatement is a cooperative task.

Refer to Chapter 3, Costatements, and to the Dynamic C ***Technical Reference*** manual for more information.

Preemptive Multitasking

Preemption means tasks are interrupted and control is taken away involuntarily (by an interrupt). We say a task is *preempted* by another task, perhaps of a higher priority. A task has no control of when preemption may take place, when preemption is enabled, but a task can turn off preemption altogether (by disabling interrupts).

A preemptive multitasking environment needs a kernel to stop and start tasks. This software that monitors, regulates, and dispatches tasks, usually uses a timer interrupt to indicate when it is time to preempt the currently active task.

Because a task does not know when preemption may take place, programmers must be careful when variables are shared among preemptive tasks. Cooperation, and thus, communication, among preemptive tasks is the major problem.

The real-time kernel (RTK) supplied with Dynamic C supports prioritized preemption: only tasks of a higher priority can interrupt tasks of lower priority. As many priority levels as needed can be created when using the RTK. The RTK also has a **`suspend`** function for a high-priority task to suspend itself (for a specified length of time or until awakened by other tasks) so that lower priority tasks can execute. The **`suspend`** function helps a task to be cooperative.

Z-World also ships a *simplified* real-time kernel (SRTK). Like the full RTK, the simplified RTK is prioritized and preemptive. However, there are only three levels of priority in the SRTK. The top-priority task executes at 25-millisecond intervals, the low-priority task executes at 100-millisecond intervals, and background processing (even lower priority) can execute when no other tasks are executing.

There is also a “fast call” task available with either kernel, or when no kernel is running, to execute as often as 1280 times per second. The “fast call” task preempts all other tasks.

The fixed properties of the SRTK make it compact and easy to use. The SRTK can combine with cooperative multitasking (costatements) to provide a robust multitasking environment.

A timer interrupt handler may have to be written if a higher frequency is desired. Assuming the interrupt handler has about 100 instructions, the maximum frequency of timer interrupts could be about 6–10 kHz on a 9-MHz board.

Note that the SRTK depends on a fixed frequency of Timer 1 (a Z180 on-chip timer). Consequently, changing the frequency of Timer 1 would change the timing characteristics of the SRTK.

Refer to Chapter 5, Real-Time Kernels, for further details.

Tips for Multitask Programming

Assign Priority Levels

Determine which operations ***must*** be performed within a fixed amount of time. For example, if a controller grips a falling object after the object passes a sensor, the grip action must be initiated and performed in time so that the object is caught. If the same system has a liquid-crystal display (LCD) to display the time, the gripper routine should get higher priority than the LCD routine.

Generally, the tighter the deadline, the higher the priority. Lower priority tasks are the ones that can be interrupted (preempted). Their completion can be delayed by a preempting task.

Tight deadlines do *not* imply high frequency. The falling object may arrive once in an hour, or at random. The LCD should be updated every second, but the gripper routine still must have a higher priority.

Identify Shared Variables

In a real-time system with concurrent tasks that share data, subtle problems can occur with variables that are stored and fetched in a nonatomic manner. When several instructions are necessary to fetch or store the variable, an interrupt can occur between the instructions. For example, a floating-point variable occupies four bytes. When a value is stored in this variable, it typically takes two or more instructions to perform the store to memory. If an interrupt occurs between these store instructions, and a new task of higher priority that uses the same variable takes over, then the new task will see a corrupted value, partly old and partly new. Such nonatomic writes can be made atomic by disabling the interrupts to prevent this situation.

```
DI();
fnum = xnum + 5.34;
EI() ;
```

DI disables the interrupts, assuring that no other task will begin executing until the computation is complete. **EI** then enables the interrupts.

The keyword **shared** declares a variable to be *atomic*. Dynamic C automatically handles interrupts on stores and fetches of **shared** variables.

```
shared float f1, f2;        // atomic variables
```

Interrupts, if enabled, will be disabled before a multi-instruction load or store. Following the load or store, interrupts are then restored to their previous state. Byte fetches and stores are naturally atomic, and nothing needs to be done. Loads and stores of 16-bit data may or may not need their interrupts disabled, depending on the type of processor instruction used.

If a structure is declared **`shared`**, then any fetch of an element will be atomic. An assignment that causes the entire shared structure to be moved in memory is also atomic.

CHAPTER 3: COSTATEMENTS

Costatements allow *cooperative multitasking* within an application. Costatements are blocks of code that can suspend their own execution at various times for various reasons, allowing other costatements or other program code to execute. Costatements operate concurrently.

There are several advantages such as the following to using costatements.:

- Costatements are a feature built into the language.
- Costatements are cooperative instead of preemptive.
- Costatements can operate without multiple stacks.

Using costatements effectively requires knowledge of their syntax, their supporting data structures, and the mechanisms by which they may be put to use.

Chapter 3 discusses the following topics.

- Solving problems with costatements
- Costatement functions
- Costatement topics: timing, dependency, nesting, and errors

Solving Problems with Costatements

Costatements are Z-World's extension to C to facilitate cooperative multitasking. The following example shows how to use costatements to simplify the solution to a real-time problem.

Example

The following sequence of events is common in real-time programming.

Start:

1. Wait for a push-button to be pressed.
2. Turn on the first device.
3. Wait 60 seconds.
4. Turn on the second device.
5. Wait 60 seconds.
6. Turn off both devices.

Go back to the start.

The most rudimentary way to perform this function is to idle ("busy wait") in a tight loop at each of the steps where waiting is specified. But most of the computer time will used waiting for the task, leaving no execution time for other tasks. Figure 3-1 shows the execution of this task schematically.

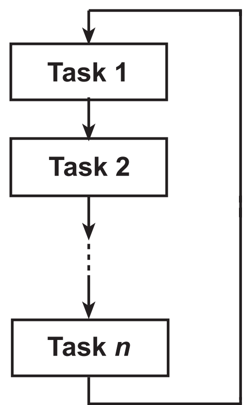

Figure 3-1. Execution of Sequence of Tasks in a Program

If there are other tasks to be run, this control problem can be solved better by creating a larger loop that processes a number of tasks. Now, each task can relinquish control when it is waiting, thereby allowing other tasks to proceed. Each task then does its work in the idle time of the other tasks.

This situation is like that of a worker who goes from station to station performing jobs according to an instruction book at each station. By traveling from station to station, the worker can execute long sequences of jobs without having to waste time at any station waiting for the next event.

This is similar to a cook who has 5 to 10 ovens and many cook pots to prepare many different meals at the same time.

Solving Problems Without Costatements

One way to implement multitasking is with state machines. A state machine is a programming construct that goes through a sequence of states, the states being indexed by the value of one or more variables. Here is what a state machine solution might look.

```
state = 1;                 // initialization:
for(;;){
     (other tasks or state machines)
// The variable time is incremented by system every sec.
   if( state==1 ){
      if( buttonpushed() ){
         state=2;   turnondevice1();
         timer1 = time;
   }else if( state==2 ){
      if( (time-timer1) >= 60L){
         state=3;   turnondevice2();
         timer2=time;
      }
   }else if( state==3 ){
      if( (time-timer2) >= 60L){
         state=1;   turnoffdevice1();
         turnoffdevice2();
      }
   }
   (other tasks or state machines)
}
```

Solving the Problem With Costatements

The Dynamic C costatement provides an easier way to implement the task.

```
for(;;){
   costate{ ... }                  // task 1
   costate{                        // task 2
      waitfor( buttonpushed() );
      turnondevice1();
      waitfor( DelaySec(60L) );
      turnondevice2();
      waitfor( DelaySec(60L) );
      turnoffdevice1();
      turnoffdevice2();
   }
   ...
   costate{ ... }                  // task n
}
```

This solution is elegant and simple. Note that the costatement (the one that is written out) looks much like the original description of the problem. All the branching, nesting and variables within the task are hidden in the implementation of the costatement and its **`waitfor`** statements.

Summary

The Dynamic C costatement provides a formalized way of generating code that advances until it encounters a point in the logic where it is necessary to create a delay or wait for an event. Waiting is accomplished by jumping out of the costatement to allow the larger loop to continue execution. Whenever a costatement is defined, the compiler creates an associated **`CoData`** data structure at the same time. The address of the current starting point in the costatement and the time at which the currently running delay (if any) began are saved in the data structure.

A costatement can be thought of as a local computer with its own instruction counter. One or more statements are executed in the costatement on each pass of the execution thread.

Costatements are cooperative concurrent tasks because they can suspend their own operation. There are several ways they do this.

- They can **`waitfor`** events, conditions, or the passage of time.
- They can **`yield`** temporarily to other costatements.
- They can **`abort`** their own operation.

Costatements can also resume their own execution from the point at which they suspended their operation. In general, each costatement in a set of costatements is in a state of partial completion. Some are suspended, some are executing. With the passage of time, all costatements should suspend and then resume.

All costatements in the program, except those that use pointers as their names are initialized whenever the function chain **`_GLOBAL_INIT`** is called.

The functions **`VdInit`** and **`uplc_init`** also call **`_GLOBAL_INIT`**.

See Chapter 4, *Virtual Driver,* and the *CPLC.LIB* library in the ***Dynamic C Function Reference*** manual for more information.

Costatement Functions

Syntax

The general format of a costatement appears below.

```
costate [ name [state] ] {
  [ statement | yield; | abort; | waitfor( ex-
  pression ); ] . . .
}
```

A costatement can have as many statements, including **abort** statements, **yield** statements, and **waitfor**s as needed.

Costatements may be named or unnamed. The name of a named costatement can be one of the following.

- A valid C name not previously used. This results in the creation of a structure of type **CoData** of the same name.
- The name of a local or global **CoData** structure that has already been defined.
- A pointer to an existing structure of type **CoData**.

If *name* is missing, then the compiler creates an "unnamed" structure of type **CoData** for the costatement.

State

The term *state* can be one of the following.

- **always_on**. The costatement is always active. (Unnamed costatements are always on.)
- **init_on**. The costatement is initially on and will automatically execute the first time it is encountered in the execution thread. The costatement becomes inactive after it completes (or aborts).

If *state* is absent, the costatement is *initially off*. The software must trigger the costatement for the costatement to execute. Then it will execute once and become inactive again. Unnamed costatements are **always_on**.

The CoData Structure

Each costatement is associated with a structure of type **CoData**. For this discussion, assume that each costatement corresponds to a static **CoData** structure.

This is the **CoData** structure.

```
typedef struct {
  char CSState;
  unsigned int lastlocADDR;
  char lastlocCBR;
  char ChkSum;
  char firsttime;
  union{
    unsigned long ul;
    struct {
      unsigned int u1;
      unsigned int u2;
    } us;
  } content;
  char ChkSum2;
} CoData;
```

CSState

The **CSState** field contains two flags, **STOPPED** and **INIT**. The functions **CoBegin**, **CoReset**, **CoPause** and **CoResume** set these two flags. The functions **isCoDone** and **isCoRunning** report these flags. Table 3-1 summarizes the meanings of the **STOPPED** and **INIT** flags.

Table 3-1. Explanation of `STOPPED` ***and*** `INIT` ***Flags in*** `CSState`

STOPPED	INIT	Meaning
Yes	Yes	The costatement either is "done," or has been initialized to run from the beginning, but set to inactive. This condition can be set by **CoReset**.
Yes	No	The costatement is paused, waiting to resume execution from wherever it was paused. This condition can be set by **CoPause**.
No	Yes	The costatement has been initialized to run from the beginning, and will run when the program execution reaches it. This condition can be set by **CoBegin**.
No	No	The costatement is active and running and will resume execution where it left off when the program execution reaches it. This is the normal condition of a running costatement. **CoResume** will return the flags to this state.

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

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

The **CSState** field applies only if the costatement has a name. The **CSState** flag has no meaning for unnamed costatements.

Last Location

The two fields **lastlocADDR** and **lastlocCBR** represent the 24-bit address of the location at which to resume execution of the costatement. If **lastlocADDR** is zero (as it is when initialized), the costatement executes from the beginning, subject to the **CSState** flag. If **lastlocADDR** is nonzero, the costatement resumes at the 24-bit address represented by **lastlocADDR** and **lastlocCBR**.

These fields are zeroed when (1) the **CoData** structure is initialized by a call to **_GLOBAL_INIT**, **CoBegin** or **CoReset**, (2) the costatement is executed to completion, or (3) the costatement is aborted.

Check Sum

The **ChkSum** field is a one-byte check sum of the address. (It is the exclusive-or result of the bytes in **lastlocADDR** and **lastlocCBR**.) If **ChkSum** is not consistent with the address, the program will generate a run-time error and reset. The check sum is maintained automatically. It is initialized by **_GLOBAL_INIT**, **CoBegin** and **CoReset**.

First Time

The **firsttime** field is a flag that is used by **waitfor** statements. It is set to 1 before the **waitfor** expression is evaluated the first time. This aids in calculating elapsed time for the functions **DelayMS**, **DelaySec**, and **DelayTicks**.

Content

The **content** field (a union) is used by the costatement delay routines to store a delay count.

Check Sum 2

The **ChkSum2** field is currently unused.

The Implementation of Costatements

yield

At a **yield** statement, the processor (1) stores the address of the following statement in **lastlocADDR** and **lastlocCBR** as the resume address, and (2) exits the costatement. Consequently, when the costatement is executed again, it continues from the statement that follows the **yield** statement.

abort

At an **abort** statement, the processor (1) resets **lastlocADDR** and **lastlocCBR** to zeros to indicate the costatement is reset, and (2) exits the costatement block.

waitfor

The **waitfor** statement can be viewed as the following equivalent code.

```
x->firsttime = 1;
while( !expression ) yield;
```

where **x->firsttime** will become 0 (reset by Dynamic C) after the expression is evaluated. In this code, ***x** is the **CoData** structure corresponding to the costatement. The field **firsttime** indicates whether it is the first time the expression is evaluated.

The `firsttime` *Flag and* `firsttime` *Functions*

A **firsttime** function is a delay function that can be called from a **waitfor** statement. For example, the first time the **DelayMs** function is called, it must set up the countdown variables for the specified amount of delay (stored in the field **content** of a **CoData** structure. All subsequent calls to **DelayMs** merely check whether the delay has expired. The initialization flag must be associated with the **CoData** structure because several costatements may call **DelayMs**.

A **firsttime** function is declared with the keyword **firsttime**. A proper **firsttime** function definition would look like this.

```
firsttime int MyDelay( CoData *ptr, delay
  params... ){
    some code
}
```

The first argument of a **firsttime** function must *always* be a pointer to a **CoData** structure. A **firsttime** function will use this pointer to check whether the costatement's **firsttime** field is 1. If so, the function will set up variables required to count the delay. The **firsttime** function should also set the **firsttime** flag to 0 so subsequent visit to the **waitfor** do not reset the delay counter.

Calling a First-Time Function

From within a costatement, use a **firsttime** function as an argument to a **waitfor** statement.

```
costate{
...
  waitfor( MyDelay(1000) );
...
}
```

Note that the call to **MyDelay** has only one parameter. The **CoData** pointer, required in the function definition, is not to be included in the call. The compiler automatically passes the address of the **CoData** structure as the first argument if a **firsttime** function is called from within a costatement.

Associated Keywords and Functions

waitfor

A costatement can wait for an event, a condition, or the passage of a certain amount of time. For this purpose, there is the **waitfor** statement, permitted only inside a costatement.

```
waitfor ( expression );
```

The **waitfor** suspends progress of the costatement, pending some condition indicated by *expression*.

When a program reaches **waitfor**, if *expression* evaluates false (zero), the reentry point for the costatement is set at the **waitfor** statement and the program jumps out of the costatement. Then, the program evaluates the **waitfor** expression each time it reenters the costatement. If the expression is false, the program jumps out again. If the expression is true (nonzero), the program continues with the statement following the **waitfor**.

Delay Functions

Three special functions (others can be added) allow delays to be used in the expression evaluated by a **waitfor**.

```
int DelaySec( unsigned long seconds );
int DelayMs( unsigned long milliseconds );
int DelayTicks( unsigned int ticks);
```

Thus, expressions such as the following can be used.

```
// wait for 30 minutes
  waitfor( DelaySec(30L*60L) );
// wait for device or 40 milliseconds
  waitfor( DelayMs(40L) || device_ready() );
```

These delay functions depend on the virtual driver. Initialize the virtual driver with a call to **VdInit** before they can be used.

See Chapter 4, The Virtual Driver.

yield

A costatement can **yield** to other costatements. The **yield** statement is permitted only inside a costatement.

```
yield;
```

The **yield** makes an unconditional exit from a costatement.

abort

A costatement can terminate itself using the **abort** statement, permitted only inside a costatement.

```
abort;
```

The **abort** statement, in effect, causes the execution to jump to the very end of the costatement, where it exits. The costatement will then terminate. If the costatement is always on, the next time the program reaches it, it will restart from the top. If the costatement is not always on, it becomes inactive and will not execute again until turned on by some other software. (Unnamed costatements are always on.)

Other Functions

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

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

- **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 clear.

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

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

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

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

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

  The function **isCoRunning** returns true (1) if the **STOPPED** flag is not set. Otherwise it returns 0.

Costatement Topics

Timing Issues

Costatements in most instances are grouped as periodically executed tasks. A costatement can be part of a real-time task, which executes every *n* milliseconds, as shown in Figure 3-2.

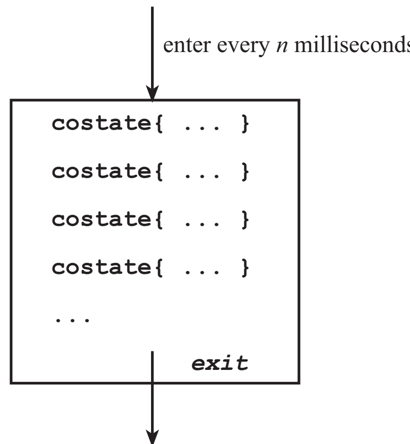

Figure 3-2. Costatement as Part of Real-Time Task

If all goes well, the first costatement will be executed at the periodic rate. The second costatement will, however, be delayed by the first costatement. The third will be delayed by the second, and so on. The frequency of the routine and the time it takes to execute comprise its ***granularity***.

If the routine executes every 25 milliseconds and the entire group of costatements executes in 5 to 10 milliseconds, then the granularity is 30 to 35 milliseconds. Thus the delay between the occurrence of a **`waitfor`** event and the statement following the **`waitfor`** can be as much as the granularity, 30 to 35 milliseconds. The routine may also be interrupted by higher priority tasks or interrupt routines, increasing the variation in delay.

The consequences of such variations in the time between steps depends on the program's objective. Suppose that the typical delay between an event and the controller's response to the event is 25 milliseconds, but under unusual circumstances the delay may reach 50 milliseconds. An occasional slow response may have no consequences whatsoever. If a delay is added between the steps of a process where the time scale is measured in seconds, then the result may be a very slight reduction in throughput.

Consider the delay between sensing a defective product on a moving belt and activating the reject solenoid that pushes the object into the reject bin. If such a critical delay cannot exceed 40 milliseconds, then a system will sometimes fail if its worst-case delay is 50 milliseconds.

`waitfor` Accuracy Limitations

If an idle loop is used to implement a delay, the processor continues to execute statements almost immediately (within nanoseconds) after the delay has expired. In other words, idle loops give precise delays. Such precision cannot be achieved with **`waitfor`** delays.

A particular application may not need very precise delay timing. Suppose the application requires a 60-second delay with only 100 milliseconds of delay accuracy; that is, an actual delay of 60.1 seconds is considered acceptable. Then, if the processor guarantees to check the delay every 50 milliseconds, the delay would be at most 60.05 seconds, and the accuracy requirement is satisfied.

Dependency Relations Among Costatements

A program can contain a large number of costatements. In general, there will be groups of costatements that share the same loop or the same context. Costatements in a second context that preempt costatements in the first context are restricted in how they can share data, so special care must be paid to mutual communication among preempting tasks.

Costatements may be initially OFF, initially ON, or ALWAYS ON. Those that are not always on can be turned on remotely, ***even across preempting tasks***.

The following code will start a remote costatement and wait for its completion:

```
costate...{
   ...
   CoBegin( name );
   waitfor( isCoDone(name) );
   ...
}
```

Setting the start field does not actually cause the costatement to execute. This simply means that the costatement is active and is ready to execute. The calling program's execution thread must pass through the costatement.

However, this code works when *only one* task can request the costatement at a time. If several tasks request the execution of a costatement at the same time, the costatement will certainly execute—once, maybe twice. In general, each of the multiple requesters is not guaranteed a separate execution of the costatement.

Request the execution of a costatement in the following cases.

- The requester and requestee are in different tasks, where one preempts the other, so direct calls are out of the question.
- Multiple costatements must start simultaneously, something that cannot be done with a direct call.
- One task needs to start another, but no further synchronization is needed between the tasks.

Nesting Costatements

If a costatement needs to run another costatement as a subtask, where the original task cannot proceed until the subtask has completed, a direct call using a **waitfor** is probably the best way to accomplish the desired end. This is done by placing the subordinate costatement in a C function that returns 0 on each pass of the execution thread except the last pass, when completion occurs, at which time it returns 1.

```
// Upper level call within a costatement
  costate...{
    waitfor( function(args) );
  }
// function outline
  int function( args ){
    ...
    costate...{
      flag=0;
      body of costatement
      flag=1;
    }
    return flag;
  }
```

In this case, the subroutine is called repeatedly each time the execution thread passes through the **waitfor**. The subroutine returns zero each time, until the execution of the costatement is completed, when it returns a one, satisfying the **waitfor** in the upper-level costatement. This approach has the additional advantage of providing local variables and arguments that are passed. A disadvantage is that the arguments must be passed repeatedly.

Error Exits from Costatements

In special circumstances, such as the failure of a communication link, it may be necessary to make an error exit from a costatement or group of costatements.

A costatement can be aborted, either (statically) by writing **abort** in the costatement directly, or (dynamically) by calling the function **CoReset** to terminate a remotely executing costatement.

```
costate xxx always_on {
  if( bad condition ) abort;
}
```

OR

```
if( bad condition ) CoReset(&xxx);
```

Use of `setjmp` and `longjmp` for Error Exit

Programmers use the **setjmp** and **longjmp** functions to perform an error exit from nested functions. If the nested functions contain costatements, it is usually necessary to reinitialize the costatements as a part of the error exit. When costatements are nested, the nesting is usually rebuilt on each pass of the execution thread.

For example, the following nested costatements might be involved in outputting data to a user terminal.

```
costate X {                    // outermost level
  waitfor(printmenu());
}
printmenu(){
  costate Y {                  // intermediate level
    waitfor(cursorpos());
  }
}
cursorpos(){
  costate Z {                  // innermost level
    waitfor(sendstring());
  }
}
```

In this example, the outermost costatement **X** waits for a lower level routine to print a menu. The **printmenu** routine in turn waits for a cursor positioning routine, which in turn sends a string to the terminal, and waits for the completion of **sendstring**. Since costatements are used at each level, the function nest is rebuilt and torn down on each pass of the execution thread. However, the reentry position at each level for each costatement is held in static storage. If the **sendstring** routine runs into trouble, a **longjmp** may provide an appropriate escape. The **setjmp** and **longjmp** would be structured as follows in this case.

```
// in the global declarations
  jmp_buf savreg;         // storage for longjmp
  ...
// in your highest level function...
  if( setjmp(savreg) ){

  code to recover from error

  CoReset(&Z);            // terminate sendstring
  CoReset(&Y);            // terminate cursorpos
  CoReset(&X);            // terminate printmenu

}
  ...
```

```
// the call to the nested costatements
   costate X {
      waitfor(printmenu(args));
   }
   ...
// somewhere in your code will be an error exit
   if( error ) longjmp(savreg,1);
   ...
```

Detecting Errors In `waitfor` *Statements*

Sometimes there may be a time limit on how long to wait for a certain condition.

```
costate xxx always_on {
   ...
   waitfor(DelayMs(200L) || x);
   if( !x ){          // x not set and we can't wait
      errorflag = 1; // any longer!
      abort;
   }
}
```

It is important that **x** not change between the test in the **waitfor** and the test following the **waitfor**. There would be no risk of change if **x** reflects a condition that is updated, for example, at the top of a costatement loop, and remains stable through out the loop. If **x** is subject to change, then a different scheme, such as the following, could be used.

```
costate xxx always_on {
   ...
   waitfor((flag = DelayMs(200L)) || x);
   if( flag ){        // x not set and we can't wait
      errorflag = 1; // any longer!
      abort;
   }
}
```

`waitfor` *and Problems with Evaluating C Expressions*

A potential problem arises with the logical operators **||** (OR) and **&&** (AND), which are subject to short-circuit evaluation rules. Remaining terms in an OR expression, such as **a || b || c**, are not evaluated if any of the preceding terms are *true*. Remaining terms in an AND expression, such as **a && b && c,** are not evaluated if any of the preceding terms are *false*.

Furthermore, the **DelayMs** routine does not start its time delay until it is actually called. An expression such as

```
waitfor( test && DelayMs(50L) );
```

has the (perhaps unexpected) result of waiting until **test** is true and then waiting 50 milliseconds more. Because of short-circuit evaluation rules, **DelayMs** will not be evaluated if **test** is false.

If the order of the terms is reversed,

```
waitfor( DelayMs(50L) && test );
```

it waits for **test**, but not less than 50 milliseconds. It is possible to avoid short-circuit evaluation problems by using the bitwise AND and OR operators (**|** and **&**).

CHAPTER 4: THE VIRTUAL DRIVER

The virtual driver is a set of functions available for Z-World controllers that provides several services. Chapter 4 discusses the following topics.

- Periodic timer interrupts
- Global initialization

Periodic Timer Interrupts

To invoke the virtual driver, a program must include **VDRIVER.LIB** in its list of libraries. A call to **VdInit** must be included at the beginning of the program. The virtual driver is called 1280 times per second by a clock interrupt. If no real-time kernel, fastcall, or virtual watchdog is in use, the virtual driver just updates the second, millisecond, and tick timers used by the **DelaySec**, **DelayMs** and **DelayTicks** functions.

Figure 4-1 summarizes the operation of the virtual driver at each clock interrupt discussed in this chapter.

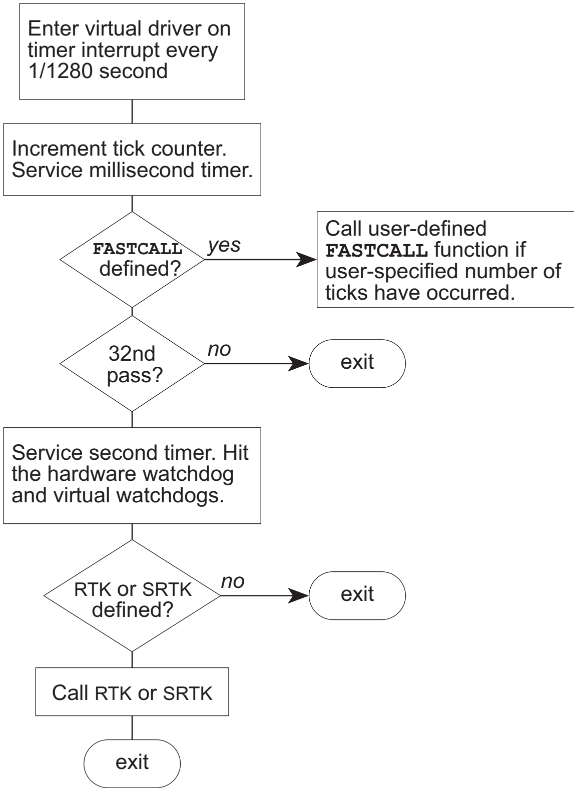

Figure 4-1. Operation of Virtual Driver at Various Clock Interrupts

Calling Sequence

To enable or disable the various services of the virtual driver, include **#define** parameters before calling the virtual driver initialization function (**VdInit**). The examples below illustrate how to do this.

```
// # RTK tasks. For RTK only, no default
#define NTASKS nn

// max # of virtual watchdog timers, default 10
#define N_WATCHDOG nn

// load and run RTK or SRTK,  default off
#define RUNKERNEL state      // 1 true, 0 off

// load fastcall, but don't run it
#define VD_FASTCALL state    // 1 load, 0 don't load

// initialize the virtual driver. default off
VdInit();

// Init fastcall to be called every nn (0-255) clock
// interrupts. It's off (nn=0) by default. Must be
// called after VdInit().
vd_initquickloop( nn );
```

The RTK Driver

If **#define RUNKERNEL 1** is defined in a program that uses the virtual driver, the function will call the real-time kernel (RTK) or simplified real-time kernel (SRTK) every 25 milliseconds. To use a real-time kernel, the program must use **RTK.LIB** or **SRTK.LIB**.

The Fastcall Driver

If the virtual driver is active, and

```
#define VD_FASTCALL 1
```

has been declared, a **vd_quick_loop** function will be called every *nn* clock interrupts. The programmer is expected to provide the **vd_quick_loop** function. (If the function is not provided, the virtual driver will call a dummy routine). The term *nn* is a number from 0 to 255, defined by **vd_initquickloop(***nn***)**, as shown in the calling sequence. If *nn* = 0, then **FASTCALL** is turned off. If *nn* = 1, **vd_quick_loop** is called 1280 times per second. If *nn* = 255, **vd_quick_loop** is called 5 times per second. Call **vd_initquickloop** after **VdInit**, otherwise *nn* will be reset to zero.

FASTCALL may be used with both the RTK and the SRTK, and without either. The **vd_quick_loop** function will preempt the highest priority tasks in either kernel. **FASTCALL** provides a method for running very fast tasks that must have response times on the order of a few milliseconds, or that need to be invoked as continuations of interrupt routines. The function **DelayTicks** has a **waitfor** granularity of less than a millisecond.

Virtual Watchdog Timers

The virtual driver has a scheme for establishing any number of independent watchdog timers in software. Each virtual watchdog is a byte that is decremented every 25 milliseconds. If any virtual watchdog reaches zero, the virtual driver deliberately times out the hardware watchdog, causing a hardware reset. The maximum number of virtual watchdogs is set by the parameter **N_WATCHDOGS**, which defaults to 10. Create virtual watchdogs with the function

```
wd = VdGetFreeWd( count );
```

This function enables a new virtual watchdog, starts it counting at **count**, and returns an integer ID. Make **count** greater than 1, because otherwise the watchdog will reach zero and time out on the first decrement. Once a virtual watchdog is created, it must be "hit" periodically so it does not time out and reset the system. "Hits" are needed by all the virtual watchdog timers. To hit the virtual watchdog use the function

```
VdWdogHit( int wd );
```

Hitting the watchdog resets its count to the **count** value with which it was created. To release the virtual watchdog **wd** use the function

```
VdReleaseWd( int wd );
```

Global Initialization

When the virtual driver is initialized with **VdInit**, the driver calls **_GLOBAL_INIT**, which executes all the **_GLOBAL_INIT** function chain segments in the program and libraries. This results in the initialization of the entire program. The **_GLOBAL_INIT** segment must appear after the variable declarations and before the executable statements.

The initialization statements placed in a **_GLOBAL_INIT** segment can be complex C, assembly code with loops, branches, and function calls, or they can be simple assignments—whatever is required by the code. The **_GLOBAL_INIT** segment can take action to precondition all or part of the hardware, as well as to preset variable data.

Dynamic C treats a variable initialized with the syntax

```
type name = value;       e.g.,  int i = 0;
```

as a constant that cannot be changed when compiled to ROM. The **_GLOBAL_INIT** segment provides a way to perform this type of initialization.

```
int i;
#GLOBAL_INIT{ i = 0; }
```

The function chain **_GLOBAL_INIT** also initializes **CoData** structures for costatements, and must be called before a program can use costatements.

CHAPTER 5: REAL-TIME KERNELS

Two function libraries support preemptive multitasking. These are the real-time kernel (**RTK.LIB**) and the simplified real-time kernel (**SRTK.LIB**). Chapter 5 discusses these topics.

- The simplified real-time kernel
- The real-time kernel
- Kernel functions

The real-time kernel (RTK) and the simplified real-time kernel (SRTK) are included with Dynamic C's function libraries. The RTK and SRTK allow a program to be divided into tasks by priority. These tasks can be treated as separate programs running independently of one another. The execution of the tasks is ***interleaved*** in time. There are two main advantages to this.

1 More urgent tasks (higher priority) are performed in preference to less urgent tasks.

2 It is easier to write and organize a program when separate tasks or sequences of events can be handled as if they were isolated from one another.

The RTK allows many levels of priorities, but the SRTK has only the three levels listed in Table 5-1.

Table 5-1. Simplified Real-Time Kernel Priorities

Name	Frequency	Priority
srtk_hightask	Every 25 ms	High
srtk_lowtask	Every 100 ms	Low
-none-	—	Background, lowest

The background task is code that executes when no other task is executing.

The virtual driver's **FASTCALL** service can be used with either kernel to implement a very high priority task.

Each RTK or SRTK task can contain multiple execution threads by using costatements.

The Simplified Real-Time Kernel

Unlike the RTK, the SRTK requires no function pointers to tasks. The routines **srtk_hightask** and **srtk_lowtask** are called every 25 milliseconds and every 100 milliseconds, respectively, by the virtual driver if (1) **SRTK.LIB** is used, (2) **RUNKERNEL** is defined, (3) the virtual driver is initialized, and (4) **init_srtkernel** has been called.

The following example shows a complete simple program using the SRTK.

```
#use vdriver.lib      // or include VDRIVER.LIB and
#use srtk.lib         // SRTK.LIB in LIB.DIR

#define RUNKERNEL 1   // use the kernel

int HCOUNT, LCOUNT;
```

```
main(){
  HCOUNT = LCOUNT = 0;
  VdInit();                    // Need virtual driver
  init_srtkernel();            // Initialize the SRTK
  while(1){ ... }  // stay alive while SRTK works
}
// This high priority task executes every 25 ms
  srtk_hightask(){ HCOUNT++;    }
// This low priority task executes every 100 ms
  srtk_lowtask(){
    LCOUNT++;
    costate{             // print every 1/2 second
      waitfor(DelayMs(500));
      printf("%d %d\n", HCOUNT, LCOUNT);
    }
    costate{   // When HCOUNT gets too big, reset
      waitfor(  HCOUNT >= 32000 );
      HCOUNT = 0;
      LCOUNT = 0;
    }
  }
```

In this example, costatements create two execution threads within the low-priority task. The SRTK was designed with costatements in mind, although some preemption is available. Background tasks can be placed in (or called from) the `while` loop in `main`. Background tasks will be preempted by any other task.

In above example, the background task would consume at least 90% percent of the CPU—even if it does nothing but idle—because the two higher priority tasks do so little computation.

The Real-Time Kernel

A clock is required for the RTK to operate. The Programmable Reload Timer (PRT) on the Z180, or any other device that can create periodic interrupts or "ticks," can be used for this purpose. Generally, the periodic interrupts run at a rate of 20 to 500 ticks per second. Higher rates require a faster microprocessor.

To use the real-time kernel, (1) define an array of task pointers, (2) specify the number of tasks, and (3) `#define RUNKERNEL`.

If the virtual driver is used, there is no need to initialize the kernel and timer interrupts. Timer interrupts will occur 40 times per second. Without the virtual driver, the kernel and timer interrupts must be initialized explicitly, and the `DelaySec`, `DelayMs` and `DelayTicks` functions cannot be used.

The following example shows how a program might look with and without the virtual driver.

```
#define NTASKS 4
#define RUNKERNEL 1
#use RTK.LIB

// the four task prototypes
  int heater(), pump(), sensor(), backgnd();

// array of 4 task pointers
  int (*Ftask[4])() = { heater,   // task 0
                        pump,     // task 1
                        sensor,   // task 2
                        backgnd };// task 3

/****** WITH VIRTUAL DRIVER ***********/
main(){
  VdInit();             // initialize VD and RTK

  run_every(0,  5); // run task0 every 5 ticks
  run_every(1, 15); // run task1 every 15 ticks
  run_every(2,100); // run task2 every 100 ticks

  backgnd();          // the lowest priority task
}

/****** WITHOUT VIRTUAL DRIVER *********/
main(){
  DI();                 // disable interrupts
  init_kernel();        // initialize kernel

  run_every(0,  5);
  run_every(1, 15);
  run_every(2,100);

  init_timer0(9216); // set timer interrupts 50Hz
  EI();              // enable interrupts
  backgnd();
}
```

The program can request that a task be invoked at a particular time (48-bit time in ticks) or after a certain delay (32-bit count in ticks) or that the task be invoked every *nn* ticks (*nn* being a 16-bit number). A task that is running can also request that its own execution be suspended for up to *m* ticks (16-bit number). Any task or interrupt routine can also request that another task be run at the first opportunity.

The RTK multitasking capability may be used as a tool to separate control loops into independent functions or modules. In this case, each task is specified to run every *nn* ticks by calling the **run_every** function once as part of the program initialization.

For example, the following scheme could be used for a task to turn on a valve each time the liquid level in a tank is low.

```
main(){
   ...
   run_every(4,100);      // task4 every 100 ticks
   ...
}
task4(){
   if( level() <= LOW  ) openvalve();
   if( level() >= HIGH ) closevalve();
   return;
}
```

If ticks occur 50 times per second, then the state of the valve will be sampled every two seconds. A very small amount of execution time will be expended on this task.

If the nature of the task is such that there are several sequences of code separated by dead time awaiting an external event or waiting for a fixed time to elapse, then the **suspend** function could be invoked. In the above example, suppose that, in order to open the valve, a motor must start, run for approximately three seconds until a contact is closed, indicating that the valve is fully open, and then the motor must be turned off. Assume that the task of closing the valve is similar, and that ticks occur 50 times per second. The following code could be used.

```
task4(){
   if( level()<=LOW && valveclosed() ){
      motor( OPEN );
      while( valveclosed() ) suspend( 5 );
      motor( OFF );
   }else if( level()>=HIGH && !valveclosed() ){
      motor( CLOSE );
      while( !valveclosed() ) suspend( 5 );
      suspend( 10 );
      motor( OFF );
   }
}
```

The first call to **suspend** stops the program from executing for the number of ticks in the argument, five ticks, or 0.1 second. The first **while** loop causes the program to **suspend** repeatedly for 0.1 second until the valve is no longer closed. (The suspend time allows the time *precision* for detecting the event to be specified.) The second **while** loop causes the program to **suspend** repeatedly for 0.1 second until the valve closes.

The additional **suspend(10)** after the motor closes ensures that the motor continues to run for ten ticks (0.2 seconds) to close the valve tightly.

The above routine could also be made into an endless loop. In that case it would be invoked one time by a call such as

```
request(task4);
```

at initialization. It would time its own progress by means of calls to **suspend** inserted in the code.

Kernel Functions

The following routines control the activity of the RTK.

- **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), which is the number of ticks since **init_kernel** was 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 gettimer(void* time)**

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

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

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

- **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 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_cancel(int tasknum)**

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

- **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.

Restrictions on the Use of Suspend

When a task is suspended, the RTK saves the top of the stack, as much as is associated with the task, in a special static array. This array has a default size of 50 bytes per task. This array larger can be made larger, if the task has many **auto** variables, by changing **TASKSTORE_SIZE**, which is defined in the library.

When a suspended task resumes execution, the top of the stack is restored to its previous state. However, the value of the stack pointer can differ, because lower priority tasks may have started or stopped during the **suspend**. They may have had their state saved on, or restored from, the stack. These problems can result from the actual absolute address of the stack frame changing.

- Pointers to **auto** variables or arguments within the task or its subroutines can point to the wrong location after a **suspend**. Avoid this problem by using **auto** variables or arguments only in routines that do not suspend, directly, or indirectly through a subroutine. Be very careful about pointers to **auto** variables, or use only **static** variables.
- The code generated by the compiler uses the IX register as a stack frame pointer under the **useix** option. This creates problems when using **suspend**.

  After a **suspend**, the IX pointer is adjusted for any change in the stack's absolute address. However the IX register will be incorrect on return to the function that called the suspended function. This is because the IX register is saved on the stack on function entry and restored on function exit. If the function suspends, then the restored IX register will usually be incorrect. This can cause the following two types of problems.

  1. Invalid pointers to **auto** variables or function arguments.
  2. Fatal errors from the stack corruption check.

If stack corruption checking is enabled, false stack corrupt messages can occur because these routines use the IX register to access the stack information on a return statement. Stack corruption checking can be disabled globally using the **COMPILER OPTIONS** command on the **OPTIONS** menu. Stack checking can also be disabled by means of the **nodebug** keyword in the function declaration.

Problems with **suspend** are avoided by not using pointers to **auto** variables or arguments belonging to the task and by following one of these rules.

1. Call **suspend** only from a root task (not a subfunction).
2. Call **suspend** only from the main program and from first-level subfunctions, and do not use any **auto** variables in the root task. Disable stack corruption checking for the main routine. (**auto** variables may be used in subroutines as long as the number of bytes used for these variables is less than **TASKSTORE_SIZE**.)

RTK Internals

The following pseudocode shows the major RTK routines.

```
BEGIN run_timer
  Step timer48        (48-bit timer)
  BEGIN task loop
    IF task has non-zero suspend count
      Decrement count and set task request when
      count becomes zero.
    ENDIF
    CASE operation mode
      "run_at":      Set task request flag.
      "run_after":   Set task request flag.
      "run_every":   Set task request flag.
    ENDCASE
  END task loop
  call rkernel() to invoke next task
END run_timer

BEGIN rkernel
  IF blocked, return, with block entry, to kernel
  BEGIN task loop, high priority first
    IF task is running, return from kernel
    IF task is requested
      Set task running and resume execution of task.
      Return or suspend.
      Clear request flag.
    ENDIF
  END task loop
  Execution never gets here. Background task will
  run instead.
END rkernel
```

The sample program **SAMPLES\DEMO_RT.C** in Dynamic C demonstrates the use of the real-time kernel (without the use of the virtual driver). The demo program has several tasks, which are used as follows.

Task 1 simulates a tank of liquid holding 1000 liters of liquid. When the level in the tank is low, the *level* flag is turned on. The level flag is equivalent to a mechanical level sensor on a physical tank. When the *intake* flag is on, 25 liters of liquid enter the tank each tick. Ten liters leave the tank every tick. If the *heater* flag is on, the temperature of the tank increases 1°C every tick. The liquid entering the tank has a temperature of 20°C. Task 1 runs every five ticks and requests that Task 2 be run immediately after it completes.

Task 2 is the control loop. Whenever the level in the tank is low, the intake flag is set, opening the intake valve. When the temperature is low, the heater flag is set, turning on the heater. The setpoint is the desired temperature of the tank, held in the variable **settemp,** which is 56°C by default.

Task 4 simulates the control of a train that goes from station to station, stopping for a certain time at each station. The **suspend** function of the RTK**,** which allows a task to suspend itself for a certain number of ticks, controls the speed of the train and the time at the stations. All the graphical displays and messages generated by this task are actually output by the background task.

Task 5 is invoked every five ticks and calls the library routine **runwatch**. This processes any watch lines entered while the program is running. Since Tasks 0–4 have a higher priority, they will not be delayed by this use of watch lines to snoop on the variables of the running program.

Backgnd could be called Task 6, but is named **backgnd** to emphasize that it runs when no other task is active. All display updates are performed by this task. This ensures that vital higher priority tasks are not delayed by the speed of **printf**. Also, one **printf** call cannot interrupt another **printf** since **printf** is not reentrant.

CHAPTER 6: FAILURE AND RECOVERY

Recovery from failures is a complex subject that is beyond the scope of this manual. Chapter 6 discusses the following topics briefly.

- Software failures
- Hardware failures
- The watchdog timer
- Reset and super-reset
- Protected variables

Even if a program is written and compiled correctly, the program may still crash because of conditions that are beyond the control of the programmer and the compiler. For example, blackouts, brownouts and power spikes can put the controller in an unpredictable state. Therefore, crash detection and recovery is an important issue for all embedded applications.

Z-World is aware of this issue, and has derived methods and support routines to facilitate the development of robust embedded systems. As a rule, the logic associated with the detection and handling of errors can be expected to be a substantial portion of a program's overall code. Z-World provides various ways to detect failures.

Software Failures

Certain software-related failures can be actively verified. For example, if the program is compiled with the debugging options activated, the compiler inserts code to do the following.

1. Check for stack integrity.
2. Check for pointer store validity.
3. Check for array bound overflowing.

If such exceptions occur, the debugging code generated by compiler calls the routine **`exception`**, which in turn calls the function to which **`ERROR_EXIT`** points. To handle such exceptions correctly, make sure that **`ERROR_EXIT`** contains the address of a correct exception handler.

Hardware Failures

Hardware-related failures are difficult to verify with software because the software runs on the failing hardware! The only verifiable hardware failure is power failure. Special hardware on the controller detects low voltage before the power goes out completely. This causes a nonmaskable interrupt (NMI), and the processor executes the NMI handler. There is usually not much time between the NMI and complete power failure.

Refer to a specific controller manual for details on how best to write a power failure routine for that controller.

Hardware Watchdog Timer

A hardware watchdog timer is a device that will reset the processor on the controller unless the timer receives a signal from the software within a specified time, usually about 1.6 seconds. The software must "hit" the watchdog timer. A correctly functioning program, which elects to use a watchdog timer, will periodically hit the watchdog at critical places to keep the processor from resetting. The assumption is that a failure must have occurred if the watchdog is not hit, and the system will reset to allow the software to attempt a recovery.

The virtual watchdog timers described in Chapter 4, The Virtual Driver, in this manual work by entering an endless loop so the hardware watchdog times out. Multiple virtual watchdog timers allow a programmer to create more robust error handlers by monitoring several different tasks separately for missed execution.

Reset and Super-Reset

A reset because of a power failure, a software failure or a watchdog time-out causes the program counter to be reset to 0000_H. The Dynamic C BIOS restarts the program automatically in these cases.

Z-World's super reset is a mechanism to initialize certain data in battery-backed RAM which should be retained after a normal reset. A super reset always occurs after a program is first loaded. It is therefore necessary to distinguish whether the program is starting from scratch during a genuine start-up, or is recovering from a failure. If it is a genuine start-up, the program must initialize its variables. If recovering from a failure, the program should recover critical data from before the failure.

Certain controllers have the helpful ability to determine whether the crash occurred because of a power fail or a watchdog time-out. Even so, a more general approach is necessary to detect these and other kinds of failures.

For this purpose, then, the Dynamic C compiler places a time stamp in each program. The time stamp is passed as an argument to **main**. It is highly unlikely that two programs would ever have the same time stamp. Thus, when a program is starting up the first time, assume that the previous program, whatever it might have been, had a different time stamp. And, if the previous time stamp is the same as the current time stamp, this is a simple reset, not a super reset.

A program example performing a conditional super-reset is shown below.

```
// CCVer is the version of the compiler
// Sec is the number of seconds from
// the beginning of 1980

main( int CCVer, unsigned long Sec ){
  static unsigned long CurSec;
  if( Sec != CurSec ){     // super-reset
       CurSec = Sec;
       perform super initialization
  }else{                   // reset due to a crash
       perform recovery
  }
  ...
}
```

This code only works if there is battery-backed RAM that stores **CurSec**. It could be modified to use flash EPROM or EEPROM.

Recovery of Protected Variables

Z-World provides a language construct and support routines for low-level recovery of important variables. This is particularly important if the application uses part of the battery-backed RAM to store log files because the log files (and the associated data structures) must persist over crashes.

A variable may be declared to be **protected**. When a variable (or structure) is protected, the compiler generates the program so that, when it stores a value in the variable, it will (1) make a backup copy of the variable, (2) set a flag indicating the backup copy is valid, (3) change the variable, and (4) reset the flag. If the controller fails during the write to the variable, the Z-World recovery function **_prot_recover** will check the flag and reestablish the correct version.

The following code shows how the previous example would be modified to include recovery for protected variables.

```
protected long CriticalVar;

// CCVer is the version of the compiler
// Sec is the number of seconds from
// the beginning of 1980

main( int CCVer, unsigned long Sec ){
  static unsigned long CurSec;
  if( Sec != CurSec ){       // super-reset
    _prot_init();
    CurSec = Sec;
    perform super initialization
  }else{                     // reset due to a crash
    _prot_recover();
    perform recovery
  }
  ...
}
```

The function **_prot_init** initializes Dynamic C's internal protection data structure. Call **_prot_init** before setting **CurSec**. The function **_prot_recover** recovers the damaged variable. (Note that only one variable could possibly be damaged if a system failure occurred.)

CHAPTER 7: THE FIVE-KEY SYSTEM

The five-key system is a library of functions (**`FK.LIB`**) supporting human use of the LCD and the keypad on Z-World's PK2100 and PK2200 series controllers. Chapter 7 discusses the following topics.

- Features for the operator
- Features for the programmer
- The five-key functions

The five-key system uses the LCD and the keypad on Z-World's PK2100 and PK2200 series controllers. Figure 7-1 shows the standard assignment of the keys on the keypad..

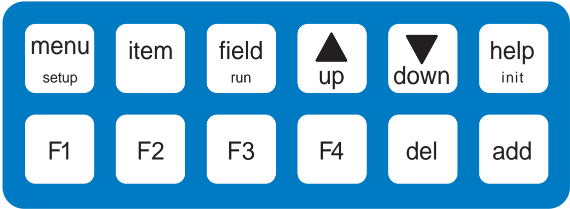

Figure 7-1. PK2100/PK2200 Keypad

The LCD has two lines of 20 characters in the standard version. Other versions are available. An underline cursor may be positioned under any character.

The five-key system uses the costatement programming paradigm and runs under the simplified real-time kernel. There also is an old five-key system library (**`5KEY.LIB`**) that requires the real-time kernel. The **`5KEY.LIB`** library is described in Appendix B.

The five-key system provides a simple scheme for changing operating parameters. It is possible to cycle through a number of menus by pressing the MENU key. For each menu, a number of items may be selected by pressing the ITEM key. Each item may have one or more fields, each of which is selected by pressing the FIELD key.

When the desired parameter is selected, adjust the parameter by pressing the UP or DOWN keys. These keys cycle forward or backward through a list of acceptable values. It is not possible to select a wrong value.

In addition to the five keys, the HELP key provides help in context for each specific menu item. The keys **F1**, **F2**, **F3**, **F4**, **del**, and **add** can each be programmed for a specific task.

Operator Features

The five-key system allows an operator to do the following tasks.

- Update parameters using the five keys—MENU, ITEM, FIELD, UP, and DOWN. Parameters can be of several types: string, float, integer, unsigned, time string, and date string. An enumerated parameter acts as a menu allowing the user to choose from a list of choices identified by user-defined strings. The five-key system continues to operate and respond to key presses while it displays messages or parameter values.
- Ask for help—using the HELP key—on any five-key menu or parameter.

Programmer Features

A programmer may extend the five-key system in these ways.

- Write code to use the **F1**, **F2**, **F3**, **F4**, **add** and **del** keys.
- Provide help messages via the HELP key.

The five-key system is programmed in the following way.

1. Initialize the virtual driver and simplified real-time kernel (SRTK).
2. Call **fk_monitorkeypad** from the high-priority SRTK task.
3. Follow the model shown in the sample code below to define menu titles, prompts and menu items in the SRTK low-priority task or a function called by it.

Sample Code

The following code shows how a program using **FK.LIB** might appear. Refer to the Dynamic C sample program **SAMPLES\FKEY\FKSAMP.C** for a complete sample program.

```
// SAMPLE PROGRAM
#use srtk.lib
#use fk.lib
#use vdriver.lib
#nointerleave
#define RUNKERNEL 1
  float f;
  int i;
  unsigned int u;
  char strng[11];
  main(){
    strcpy(strng,"abcdefghi");
    f = 25.34;
    i = 15;
    u = 5;
    VdInit();          // SRTK needs virtual driver
    init_srtkernel(); // initialize the SRTK
    up_beepvol(1);    // needed for PK2100 beeper
  // loop while SRTK works
    while(1){}        // background processing
  }
// HIGH PRIORITY TASK
  srtk_hightask(){
    fk_monitorkeypad();
    ...
  }
// continued next page
```

```
// LOW PRIORITY TASK
  srtk_lowtask(){
    fivekey_lo_task();
    ...
  }
// FIVEKEY TASK
  int fivekey_lo_task(){
    costate{
      fk_helpmsg();      // put up help messages
      PutMenus();
      user_keys();       // process user keys
    }
  }
  int PutMenus(){
    extern char fk_newmenu;
    int choice;

    costate{
      for(;;){       // loop through all the menus
        lcd_erase();
        lcd_printf(01, "time/date menu");
        fk_newmenu=0;
        for(;;){                // begin menu 1
          waitfor(fk_item-setdate(Time));
          if (fk_newmenu) break;
          waitfor(fk_item-settime(Time));
          if (fk_newmenu) break;
        }                       // end menu 1
        lcd_erase();
        lcd_printf(0x00000001,"I/O Menu1");
        fk_newmenu=0;
        for(;;){                // begin menu 2
          waitfor(fk_item_enum("enum sel %8s",
                    &choice,"first","second",
                    "third","last",0));
          if(fk_newmenu) break;
          waitfor(fk_item-int("integer = %4d",
                    &i,10,1000));
          if(fk_newmenu) break;
          waitfor(fk_item_unsigned int("unsigned
  int = %2u",
                    &u,0,10));
          if(fk_newmenu) break;
          waitfor(fk_item_alpha("Alpha = ",
                    strng,strlen(strng)));
          if(fk_newmenu) break;
          waitfor(fk_item_float("Float =%8.4f",
                    &f,-900.,9999.));
          if(fk_newmenu) break;
        }                       // end menu 2
      }                         // end menu loop
    }                           // end costatement
    return 1;
  }
  int user_keys(){
    ...                  // handle bottom row keys
  }
```

Special Key Combinations

The board resets when the menu key and certain other keys are held down simultaneously for more than 0.5 seconds. The keys that work in these combinations all have sublabels to describe their actions. These resets are described in Table 7-1.

Table 7-1. PK2100/PK2200 Keypad Resets

Key Strokes	Result
menu + field (setup + run)	Restart the program currently loaded.
menu + up (setup + pgm 19.2)	Put the board in program mode with PC serial communications set to 19,200 baud.
menu + down (setup + pgm 28.8/38.4)	Put the board in program mode with PC serial communications set to 28,800 baud for the PK2200, and to 38,400 baud for the PK2100.

The function **`fk_monitorkeypad`** described below detects these two key combinations.

Five-Key Functions

- **`void fk_monitorkeypad()`**

  Monitors the keypad for keys pressed. This function should be called as 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 is 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 time-out. There is no buffer. Key presses should be processed within 100 milliseconds or they will be lost.

- **`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 nonzero if help is off, and zero if help is on.

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

  Modifies a string using the five-key system. The term **`*s1`** is a string containing a prompt. The term **`*var`** is the string to be displayed and/or modified. **`wdsize`** is the maximum length of the word string to be edited.

- **`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.

- **`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.

- **`int fk_item_int( char *string, int *num, int lower, int upper )`**

  Displays/modifies an integer number using the five-key system. The term **`*string`** is a **`printf`** format having the form **`%`***n***`u`**, where *n* is one 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.

- **`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.

- **`int fk_item_float( char*s1, float *num, float lower, float upper )`**

  Displays/modifies a floating-point number using the five-key system. The term **`*s1`** is a **`printf`** format for displaying the number. The format code should be in the form of **`%n.mf`**. The displayed line appears as follows.

  ```
  vvvvvv wwww.yyyy
  ```

  where **`vvvvv`** is a prompt string , **`wwww`** is *n* chars long. and **`yyyy`** is *m* chars long. The value *n* must be at least 1. The sum $n + m$ cannot exceed 9. The default is $n = 6$ and $m = 3$. 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.

- **`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 `%`*`n`***`s`**) 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, and is from 0 to (n - 1).

Reading the Keypad without FK.LIB

The following program illustrates a way to read keystrokes without using **`FK.LIB`** or **`SRTK.LIB`**. The virtual driver is still required. The function **`lc_kget`** is found in **`LCD2L.LIB`**. It detects and responds to 2-key reset combinations.

```
#use vdriver.lib

int key;

main(){
   VdInit();
   for(;;){
      if((key = lc_kget(0)) != 0){
         printf("key = %d\n", key);
      }
   }
}
```

CHAPTER 8: RS-232 COMMUNICATION

Z-World supports RS-232 communication (with function libraries) for the various serial ports found in Z-World controllers and their accessories. Chapter 8 discusses the following topics.

- Serial communication
- Function libraries and sample programs

Serial Communication

Z-World supports RS-232 communication (with function libraries) for the following ports.

- Z180 Ports 0 and 1.
- SIO Ports 0 and 1 on the BL1100.
- SCC (Z80C30) Ports A and B on the BL1300.
- The XP8700 expansion card.

The functional support for serial communications consists of the following.

- Initialization of the serial ports.
- Reading data from the receive circular buffer.
- Writing data to the transmit circular buffer.

The RS-232 packages have the following features.

- Circular send and receive buffers serviced with serial interrupts.
- Echo option.
- CTS/RTS control
- XMODEM protocol for downloading and uploading data
- Modem option

Send and Receive Buffers

Serial communication is made easier with a background interrupt routine updating the send and receive buffers for serial communication. Every time a new character is received, it is put into the receive circular buffer. The receive buffer can be read one character at a time, or as a stream of characters terminated with a special character.

Data are sent by writing to the circular transmit buffer. If the serial port is not already transmitting, Z-World write functions will automatically initiate transmission. Once the last character of the buffer has been sent, the transmit interrupt is turned off. Data may be written one character at a time or as a stream of characters.

Echo Option

If the *echo* option is selected when a serial port is initialized, any character received is automatically echoed (transmitted back). This feature is ideal for systems with a dumb terminal and for checking transmitted characters. With or without echo, the serial drivers automatically parse out the BACKSPACE character (ASCII 0x08). A separate function call after initialization of the serial port can put the serial drivers in BINARY mode, that is, all data are placed in the serial receive buffer.

CTS/RTS Control

If the CTS/RTS option is selected, the support software will pull the RTS line high when the receive buffer has reached 80% of its capacity. Thus, the transmitting device (if its CTS is enabled) will stop transmitting. The software pulls the RTS line again when the received buffer has gone below 20% of its capacity.

If the device with which the controller is communicating does not support CTS and RTS, the CTS and RTS lines on the controller side can be tied together to make communication possible.

XMODEM File Transfer

Z-World supports the XMODEM protocol for downloading data from, and uploading data to, a controller. Currently, the library supports downloading an array of data whose size is a multiple of 128 bytes.

Uploaded data are written to specified area in RAM. The targeted area for writing should not conflict with the current resident program or data.

The serial driver is put into BINARY mode during XMODEM transfer. Character echo is suspended.

Modem Communication

Modems allows RS-232 communication across long distances using telephone lines. If the modem option is selected, character streams that are read from the receive buffer are automatically parsed for modem commands. If a modem command is found, appropriate actions are taken. Normally the communication package would be in COMMAND mode while waiting for valid modem command or messages. Once a link is established, communication goes into DATA mode (that is, regular RS-232 communication). In DATA mode, the modem is still monitored for a **NO_CARRIER** message.

The software assumes that modem commands (for which it scans) are terminated with CR (carriage return, or ASCII 0x0D). Therefore, the modem option is easiest to use when the protocol also has CR as the terminating character. Otherwise, the software has to check for both terminating characters. The message-terminating character cannot be any of the ASCII characters used in the modem commands, nor can it be a line-feed character (0x0A).

The software supports communication with a Hayes Smart Modem or other compatible modem. The CTS, RTS and DTR lines of the modem are not used. If the modem used is not truly Hayes Smart Modem compatible, tie the CTS, RTS and DTR lines on the modem side together. The CTS and RTS lines on the controller side also have to be tied together.

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 on both sides.

Figure 8-1 shows the correct modem to controller serial connections. The ISA COM ports 1 and 2 can be connected directly to a modem since they are already data terminal equipment (DTE).

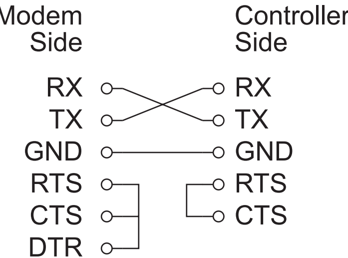

Figure 8-1. Modem to Controller Serial Connections

Function Libraries and Sample Programs

Software Support

The functions in this section are described for Port 0 of the Z180. Similar functions exist for the Z180 Port 1; SIO Ports 0 and 1; SCC Ports A and B, the XP8700 expansion card, and ISA COM Ports 1 and 2.

See the ***XP8700 and XP8800 User's Manual*** for further details on the XP8700 expansion card.

The function names for these ports incorporate the following keys.

Port	Key	Port	Key
Z180 Port 0	`z0`	SIO Port 0	`s0`
Z180 Port 1	`z1`	SIO Port 1	`s1`
SCC Port A	`sca`	ISA COM 1	`com1`
SCC Port B	`scb`	ISA COM 2	`com2`
XP8700	`uart`		

For example, the initialization routine for Z180 Port 0 is called **`Dinit_z0`** and is described here. The same function for SIO Port 0 is named **`Dinit_s0`**. The equivalent function for the XP8700 expansion card is **`Dinit_uart`**. The equivalent function call for SCC Port B is **`Dinit_scb`**. The equivalent function call for ISA Port 2 is **`Dinit_com2`**. Each function described (for **`z0`**) in this section has eight other functions that use the other keys in their names.

Interrupt Handling

Normally the serial interrupt service routine is declared with this compiler directive.

```
#INT_VEC SER0_VEC routine_name
```

However, if the same serial port is used for Dynamic C programming and for communication, the program has to be downloaded first through Dynamic C before the address of the serial interrupt service routine is loaded into the interrupt vector table. Put simply, the service routine must be loaded at run time. The function

```
void reload_vec( int vector, int(*function)() );
```

will load the address of the service routine function into the specified location in the interrupt vector table. ***Do not use the* `#INT_VEC` *directive in this case.*** It is not possible to do any further debugging through Dynamic C once the service routine has taken over.

For communication with a serial device other than the Dynamic C programming port on the PC, the program has to make sure that the hardware is properly configured before sending any serial messages. For example, when using the Z180's Port 0 for serial communication with a modem, the keypad (PK2100 or PK2200 only) may be used to trigger the initialization of the serial port. Without such a trigger, the modem might not communicate with the support software properly because the initialization routine also send commands to the modem to initialize it.

When executable programs are generated for the EPROM or for download to RAM, there will be no need for communication with Dynamic C. The compile-time directive (`#INT_VEC`) can then be used freely.

RS-232 Software

- ```
  int Dinit_z0( void* rbuf,  void* tbuf,
                int   rsize, int   tsize,
                byte  mode,  byte  baud,
                byte  modem, byte  echo )
  ```

  Initializes Z180 Port 0 for communication.

  `rbuf` pointer to the receive buffer.

  `tbuf` pointer to the transmit buffer.

  `rsize` size of the receive buffer.

  `tsize` size of the transmit buffer.

  `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=CTS/RTS off	1=CTS, RTS enabled
  ```

baud the baud rate in multiples of 1200 (for example, specify 8 for 9600 baud).

modem If 1, modem is supported. If 0, no modem.

echo If 1, every character is echoed. If 0, no echo.

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).

- **`void z0binaryset()`**

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

- **`void z0binaryreset()`**

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 Dread_z01ch( char* ch )`**

Reads a character from the receive buffer into character **`ch`**. The function returns

0 buffer is empty.

1 byte has been successfully extracted from buffer.

- **`int Dwrite_z01ch( char ch )`**

Places a character in the transmit buffer. If not already transmitting, the function initiates transmission. It returns

0 transmit buffer did not have space for **`ch`**.

1 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 buffer is terminated with a null character according to the C convention.

The function returns

0 no message found with the specified terminating character.

1 message has been successfully extracted from buffer.

- **`int Dwrite_z0( char* buffer, int count )`**

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

  0 transmit buffer did not have space for **`count`** bytes.
  1 Write is successful.

- **`void Dz0send_prompt()`**

  Places CR, LF and > in the transmit buffer.

- **`void Dreset_z0rbuf()`**

  Resets the receive buffer.

- **`void Dreset_z0tbuf()`**

  Resets the transmit buffer and stops transmission.

- **`void Dkill_z0()`**

  Disables Z180 Port 0.

Sample Program

This example shows RS-232 communication between a controller and a PC functioning as a “dumb terminal.”

```
#define CR 0x0d        // carriage return
  char baud = 8;       // 9600 baud, divided by 1200
  char mode = 4;       // 1 stop bit
                       // no parity
                       // 8 data bits
  char ismodem = 0;    // no modem is connected
  char isecho  = 1;    // data received are echoed
  char tbuf[384];      // circ. transmit buffer
  char rbuf[200];      // circ. receive buffer

main(){
  char buf[20];        // messages
  Dinit_z0(rbuf,tbuf,200,384,mode,baud,ismodem,isecho);
  Dz0send_prompt();    // send a prompt
  while(1){
  // wait for a message terminated with CR
    while( Dread_z0(buf,CR)==0 );
  // here you could do whatever you want with the message
  // in this example, the message is sent back
    Dz0send_prompt();
    Dwrite_z0(buf,strlen(buf)); // send message back
  // then, some more ...
    Dz0send_prompt();
    Dwrite_z0(“it works!”,9);   // write another
    Dz0send_prompt();
  }
}
```

For modem communication, set **ismodem** to 1. The baud rate is either 2400 or 1200. The above example above will work just as well when the PC has dialed up the controller connected to a modem. Z-World recommends using a Hayes-compatible Smart Modem. Otherwise, the modem connected to the controller would have to have its RTS (4) , CTS (5) and DTR (20) lines tied together. Also, a NULL modem is required for the serial connection between the modem and the controller's serial port.

The equivalent program for the serial communication controller (SCC) Port A would be as follows.

```
main(){
   char buf[20];          // messages
   Dinit_sca(rbuf,tbuf,200,384,mode,baud,ismodem,isecho);
   Dscasend_prompt();     // send a prompt
   while(1){
   // wait for a message terminated with CR
      while( Dread-sca(buf,CR)==0 );
   // here you could do whatever you want with the message
   // in this example, the message is sent back
      Dscasend_prompt();
      Dwrite_sca(buf,strlen(buf)); // send message
   back
   // then, some more ...
      Dscasend_prompt();
      Dwrite_sca("it works!",9);   // write another
      Dscasend_prompt();
   }
}
```

The serial communication software implements delays with the **suspend** function of the real-time kernel (RTK) when the RTK is used. Otherwise, the delays are implemented with a software countdown loop.

XMODEM Commands

- **int Dxmodem_z0down(char* buffer, int count)**

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

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

- **int Dxmodem_z0up (unsigned long address, int* pages, int dest, int(*parser)())**

  Receives (uploads) a file using XMODEM protocol. The parameters are described below.

address the physical address in RAM where the received data is 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, the logical address of the buffer must first be converted to a physical address using the library function **phy-adr**.

pages the number of 4-kbyte blocks of data that have been transferred.

dest If an RS-485 master-slave network is set up, specify **dest** = 0 when the upload is intended for the master. If **dest** is nonzero, the upload is intended for the designated slave.

parser the function that handles parsing of the uploaded data.

The function returns

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

Miscellaneous Functions

- **int Dget_modem_command()**

Deciphers a Hayes-compatible modem command. The function returns –1 if no modem command is matched.

The modem commands are summarized below.

```
0   "\nOK"                // okay respond
1   "\nCONNECT"           // connect at 300 baud
2   "\nRING"              // ring detected
3   "\nNO CARRIER"        // no carrier
4   "\nERROR"             // command error
5   "\nCONNECT 1200"      // connect at 1200 baud
6   "\nNO DIALTONE"       // no dial tone
7   "\nBUSY"              // line busy
8   "\nNO ANSWER"         // no answer
9   "\nCONNECT 2400"      // connect at 2400 baud
10   "\n"                 // just a line feed
```

A Hayes Smart Modem (or compatible) is recommended. A NULL modem is needed between the PK2100 and the modem.

Some modems may require that the RTS(4), CTS(5) and DTR(20) lines on the modem side be tied together.

- **`void Drestart_z0modem()`**

  Restarts the modem (during start of program or abnormal operation).

- **`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.

  The function returns

  0 Valid modem command.
  –1 Invalid modem command.

- **`void Dz0_circ_int()`**

  This is an interrupt service routine for Z180 Port 0.

- **`void Ddelay_1sec()`**

  Creates a 1-second delay (approximately). If **`RUNKERNEL`** is defined, **`suspend(50)`** is used to generate the delay.

- **`void Ddelay_100ms()`**

  Creates a 100-millisecond delay (approximately).

- **`void reload_vec( int vector, int(*function)() )`**

  Load the address of an interrupt service routine (ISR) into the vector table. This function is useful during program development when either of the Z180 Port 0 (Z0) or the PLCBus XP8700 UART #3 is used as the Dynamic C programming port. It is also useful as a safeguard against the BIOS rewriting the Z0 or /INT1 (PLCBus UARTs) vector when the controller is powered up or reset into programming mode, as occurs when using the Program Loader Utility (PLU) to update the controller's application either via port Z0 or PLCBus UART #3.

  If the application uses the Z0 or PLCBus UART serial ports it should call **`reload_vec`** once each for the used Z0 or PLCBus ISRs early in its normal initialization to ensure that the respective vectors point to its ISRs. For other serial ports use the compile-time interrupt directive **`#INT_VEC`** to load the ISR address into the interrupt vector table when generating the executable code for EPROM or for download to RAM.

  `vector` is the offset for the specific interrupt.

  `function` is a pointer to the interrupt service routine.

Since Dynamic C 32 v. 6.30, **`reload_vec`** updates the vector only when different from the current vector. Avoid repeatedly updating a vector with different ISR addresses on a Flash EPROM equipped controller since the Flash EPROM has a maximum life of about 10,000 writes.

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

  Computes the CRC (cyclic redundancy check, or check sum) for data in **`buffer`**. Calls to **`getcrc`** can be “concatenated” to compute the CRC for a large buffer.

  `buffer` is a pointer to characters for which to compute the CRC.

  `count` is the number of characters in **`buffer`**, limited to 255, for this function.

  `accum` is the accumulated CRC value from previous computation.

  The function returns the integer CRC value.

- **`void resetZ180int()`**

  This is a generic reset function that resets, or disables, interrupts for the DMA channels, the Z180 serial channels 0 and 1, the programmable reload timers, and CSIO, INT1 and INT2.

RS-485 Drivers

If a serial port supports full-duplex RS-485 communication, the same drivers used for RS-232 communication may be used. Just turn on RS-485 transmission according to the particular port. For example, to use Z180 Port 1 as an RS-485 port, do the following.

```
outport(ENB485,1);
Dinit_z1( ... );
```

Libraries

Table 8-1 lists the Dynamic C libraries that support serial communication.

Table 8-1. Dynamic C Libraries Supporting Serial Communication

`Z0232.LIB`	Z180 Port 0
`Z1232.LIB`	Z180 Port 1
`S0232.LIB`	SIO Port 0
`S1232.LIB`	SIO Port 1
`SCC232.LIB`	SCC Port A and B
`XP87XX.LIB`	XP8700 PLCBus expansion board
`COM232.LIB`	ISA COM port
`MODEM232.LIB`	Accessory library used by all the preceding libraries

Sample Programs

Table 8-2 lists sample programs that illustrate serial communication.

Table 8-2. Dynamic C Serial Communication Sample Programs

RS232.C	Simple RS-232 communication using the Z180 Port 0.
XP87XX.C	Simple RS-232 communication with the XP8700 PLCBus expansion board.
Z1232.C	Simple RS-232 communication using the Z180 Port 1.
SCC232.C	Simple RS-232 communication using SCC Ports A and B.
DOWNLOAD.C	RS-232 communication for a monitor program using the Z180 Port 0 of a PK2100.

The programs in Table 8-3 use the serial port as a diagnostic port. Except for **Z1REM.C** and **SCCREM.C**, these programs are also the master programs that can talk to a slave running **SREMOTE.C** or **CSREMOTE.C** via the RS-485 half-duplex linkage.

Table 8-3. Dynamic C Sample Programs

Z0REM.C	RS-232 communication with Z180 Port 0.
Z1REM.C	RS-232 communication with Z180 Port 1.
S0REM.C	RS-232 communication with SIO Port 0.
SCCREM.C	RS-232 communication with SCC Ports A and B.
UARTREM.C	RS-232 communication with the XP8700 PLCBus expansion board.
CZ0REM.C	RS-232 communication with the Z180 Port 0 for the PK2100 and the PK2200.
CUARTREM.C	RS-232 communication with the XP8700 PLCBus expansion board.
COM232.C	Simple RS-232 communication for ISA COM Ports 1 and 2.
COM1REM.C	RS-232 communication with the ISA COM Port 1.
COM2REM.C	RS-232 communication with the ISA COM Port 2.

CHAPTER 9: MASTER-SLAVE NETWORKING

Z-World supports master-slave networks using 2-wire and 4-wire RS-485 networks. The master and slave controllers communicate using a protocol based on the ninth bit. Chapter 9 discusses these topics.

- Communication protocol
- Hardware connections
- Software support
- Libraries and sample programs

Z-world has library functions for master-slave two-wire half-duplex RS-485 9-bit binary communication. This protocol is supported only on Z180 Port 1, which is configured for RS-485 communication on most Z-World controllers. Boards that provide access to Z180 Port 1 can be the master or the slave. There should only be one master, which will then have a board address of 0. The slaves must have their own distinct identification numbers from 1–255.

The functional support for the master-slave serial communication consists of the following steps.

1. Initialization of Z180 Port 1 for RS-485 communication.
2. Master sends inquiry and waits for response from a slave.
3. Slaves monitor for their address during the ninth bit of a transmission. The targeted slave replies to the master.

Communication Protocol

The binary command message protocol adopted is similar to that used for the Opto-22 binary protocol. A master message is composed as shown here.

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

The slave's response is composed as shown here.

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

The term **len** is the length of the message that follows.

During a transfer from the master, the address byte is transferred in the ninth bit of the address mode, and only the slave that matches this address will listen to the rest of the message, which is sent in regular 8-bit data mode.

Hardware Connection

Figure 9-1 shows the connections for a two-wire RS-485 network. Any Z-World's controller can be a master or a slave. There should only be one master, but there can be up to 255 slaves.

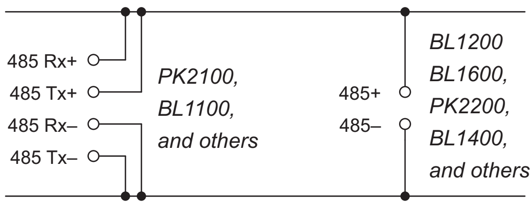

Figure 9-1. Two-Wire RS-485 Network Connections with Z180 Port 1

Some Z-World controllers, such as the PK2200, have half-duplex RS-485 ports. Other controllers, such as the PK2100, have full-duplex RS-485 ports. A full-duplex RS-485 port can be made into a half-duplex RS-485 port by connecting RX+ to TX+ and RX– to TX–.

Software Support

- **`void op_init_z1( char baud, char* rbuf, byte address )`**

  Initializes Z180 Port 1 for RS-485 ninth-bit protocol binary communication. The data format defaults to 8 bits, no parity, 1 stop bit.

  `baud` selects the baud rate in multiples of 1200 (specify 16 for 19,200 baud).

  `rbuf` the receive buffer.

  `address` the network address of the board: 0 for the master board, 1–255 for slaves.

- **`int check_opto_command()`**

  Checks for a valid and completed command or reply in the receive buffer. The function returns with

  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( byte dest, char* message, byte len, int delays )`**

  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]
  ```

  The parameters are identified below.

  `dest` the slave destination (1–255).

  `message` the message.

  `len` the length of the message. The maximum message length is 251 bytes.

  `delays` the number of delays to wait for the slave reply. Each delay is ~50 milliseconds. However, if the RTK is in use, the delay is made by a **`suspend(2)`**.

  The function returns

  –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, byte 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]
  ```

  reply the slave's reply string.

  count the length of the reply. The longest reply is 252 bytes because two CRC bytes are appended at the end.

  delays the number of delays before the message is transmitted back. Each delay is ~50 milliseconds. However, the delay is made by a `suspend(2)` if the RTK is being used.

Miscellaneous Functions

- **`void misticware( char* tbuf, char count )`**

  This is the gateway for RS-485 ninth-bit protocol for binary communication. The receive and the transmit buffers must already be set up. Interrupt-driven transmission must be intitialized.

  tbuf the transmit buffer. Data in the buffer should already be in the correct format.

  `count` the number of bytes to be transmitted.

- **`void optodelay()`**

  This function produces a delay of ~50 milliseconds. The delay is implemented with a `suspend(2)` if the RTK is being used. Otherwise, it is a software countdown delay.

- **`int rbuf_there()`**

  Monitors the receive buffer for a completed command or reply. The function returns

  1 if a completed command or reply is available.
  0 if a completed command or reply is not available.

- **`void op_send_z1( char* tbuf, byte count )`**

  This function is called by `misticware` to initiate transmission of data.

- **`void op_rec_z1()`**

  This function is called by `misticware` to reset and to ready the receiver for data reception.

- **`void op_kill_z1()`**

  Disables Z180 Port 1. The RS-485 driver is also disabled.

- **void z1_op_int()**

  This is an interrupt service routine for the Z180 Port 1 used in master-slave networking.

Sample Program

```
// MASTER.C (running in slave controller)
// The master sends a string of messages to the slave.
// Slave replies. The master prints the reply at STDIO.
   char rbuf[255];        // receive buffer
   char reply[40];        // reply buffer
   char msg[40];          // message buffer

main(){
   int i,j,ercode;
   VdInit();

// initialize 19200 baud, receive and master
   op_init_z1(16,rbuf,0);
   j = 1;

   while(1){
   // wait 10000 counts before polling
      for(i = 0; i< 10000; i++) runwatch();
      sprintf(msg,"Message # %d", j++);
   // send message to slave 1 and wait for reply
      ercode = TalkToSlave(1,msg,strlen(msg),3,reply);
      if( ercode == 1){       // if reply is valid
         printf("%s\n", reply);
      }else{         // if bad reply or link failure
         printf("Link Failure or Bad Link\n");
      }
   }
}

int TalkToSlave( int slave_no, char *query,
                 int len, int ndelay, char *reply ){
   int i,ercode;
   // sends message and waits for reply
   ercode = sendOp22( slave_no, query, len, ndelay);

// if reply is valid, copy to reply buffer
   if( ercode == 1){
      for( i = 0; i < rbuf[0] - 2; i++){
         reply[i] = rbuf[1+i];
      }
      reply[i] = '\0';
   }
   return ercode;
}

// continued....
```

```
// SLAVE.C (running in slave controller)
// The slave waits for a message from the master.
// Slave prints the master's message to STDIO and
   also
// replies to the master.
   char rbuf[255];          // receive buffer
   char query[40];          // query buffer
   char reply[40];          // reply buffer
main(){
   int j;
   VdInit();
// initialize for 19200 baud, receive buffer,slave1
   op_init_z1( 16, rbuf, 1);
   j = 1;
   while(1){
      runwatch();
      if( masterquery(query) ){
      // check for master query
         sprintf(reply,"%s, Slave Reply %d", query, j++);
      // sends back reply to the master
         replyOpto22( reply, strlen(reply), 0);
      }
   }
}
// check for master query
// copy message from the master to query if it's valid
int masterquery( char *query ){
   int i;
   if( check_opto_command() != 1) return 0;
   for( i = 0; i< rbuf[1] - 2; i++){
      query[i] = rbuf[2 + i];
   }
   query[i] = '\0';         // put an end-of-string
   return 1;
}
```

Note that **sendOp22** just sets up the message for transfer using the ninth-bit protocol. It ***does not poll*** for the completion of the transfer. All transfers of messages are done in the background interrupt routine.

Libraries and Sample Programs

Libraries

Table 9-1 lists the Dynamic C libraries that support master-slave communication.

Table 9-1. Dynamic C Libraries That Support Master-Slave Communication

NETWORK.LIB	Half-duplex RS-485 communication drivers.
MODEM232.LIB	Accessory library used by preceding libraries.

Sample Programs

Table 9-2 lists the sample programs that illustrate master-slave communication.

Table 9-2. Dynamic C Sample Programs for Master-Slave Communication

RS485.C	Simple slave RS-485 program. Talks back to master board running **RS232.C**.
SREMOTE.C	Diagnostic port via the RS-485 linkage. The master has to be running one of the following programs: **Z0REM.C**, **UARTREM.C**, **CZ0REM.C**, **CUARTREM.C**, **S0REM.C** , **COM1REM.C** or **COM2REM.C**.
CSREMOTE.C	Same as **SREMOTE.C** but will only run on the PK2100 or the PK2200.

APPENDIX A: EXECUTION SPEED

Table A-1 lists the execution times of various arithmetic operations. Z-World controllers have clocks frequencies ranging from 6.144 to 18.432 MHz.

Table A-1. Execution Times of Arithmetic Operations (μs)

Operation	Clock Frequency (MHz)			
	6.144	9.216	12.288	18.432
16-bit integer add	2.7	1.8	1.4	0.9
16-bit integer multiply	24	16	12	8
16-bit integer divide	142.5	95	72	47.5
Long (32-bit) integer add	21	14	10.5	7
Long integer multiply	123	82	60.5	41
Long integer divide	609	406	304.5	203
Floating-point (32-bit) add or subtract	117	78	19.5	39
Floating-point multiply	171	114	85.5	57
Floating-point divide	429	286	214.5	143
Sine or cosine	4650	3100	2325	1550
Square root	1275	850	637.5	425

Table A-2 lists the execution times of various logical and counting operations for a 9.216-MHz clock.

Table A-2. Execution Times of Logical Operations

Operation	Execution Time
`if(k)`	2.6 μs
`for(k=0;k<100;k++){}` *for* loop overhead	12.8 μs
`sub(n);` subroutine call overhead with 1-integer argument	5.8 μs
`switch(n){...}` first case each additional case	22 μs 7 μs
`costate{}` costatement entry + exit overhead	32 μs
`costate{waitfor(0)}` costatement waiting	19 μs

Table A-3 lists the execution times of other operations. These times are approximate times, assuming a typical environment. Hand coding in assembly language or other special care can result in substantially increased performance.

Table A-3. Execution Times of Other Operations

Operation	Execution Time
Response to an interrupt or interrupt latency	100 μs
Sustained data throughput, interrupt driven	20K bytes per second
Burst data I/O using DMA	500K bytes per second

APPENDIX B: OLD 5-KEY SYSTEM

The old five-key system is a set of functions (**5KEY.LIB** and **5KEYEXTD.LIB**) that has the all of the functionality of the newer **FK.LIB**, and a few additional features. But the old five-key system is more complex and somewhat harder to use, and the old five-key system runs only with the full real-time kernel (RTK), not the simplified real-time kernel (SRTK).

With the old 5-key system, the ADD and DELETE keys can be used for an "extended" five-key system where menu items can be added or deleted while a program is running. The old 5-key system also supports up to four alarm functions.

The old 5-key menu server operates as a task running under the real-time kernel. The menu server can either be "code-driven" or "linked-list driven." The code-driven technique is more flexible, but the linked-list method is easier to use.

Code-Driven Approach

In the code-driven approach, the programmer creates menus for updating parameters. A real-time-kernel task "calls" all the menus repeatedly. Each menu handles its own list of items.

The following examples shows a conceptual code-driven menu system.

```
main(){
   initialization
   background();
}

indirect background(){
   request(TASK1);
   while(1);
}

indirect task1(){         // the code-driven menu
   server
   while(1){
      menu1();
      optional menu 1 housekeeping
      menu2();
      ...
      menuN();
   }
}

// continued
```

```
int menu1(){
  int k;
  while(1){
    k = _5key_parameter1();
    optional housekeeping for parameter 1
    if( k == MENU ) return;   // menu key pressed?

    k = _5key_parameter2();
    optional housekeeping for parameter 2
    if( k == MENU ) return;   // menu key pressed?

    k = _5key_parameter3()
    optional housekeeping for parameter 3
    if( k == MENU ) return;   // menu key pressed?
  }
}
int menu2(){
  code here
}
int menuN(){
  code here
}
```

Linked-List Approach

The following code shows the linked-list approach. The library function **_5key_menu** is called from a real-time-kernel task. This function uses the linked lists that were created during initialization with calls to **_5key_setmenu**. The linked-list method handles parameters easily, but there is no way to do "housekeeping" for specific parameters or menus.

```
  main( ){
// initialization routines...
    _5key_setmenu( menu1, parameter1 );
    _5key_setmenu( menu1, parameter2 );
    _5key_setmenu( menu1, parameter3 );
    _5key_setmenu( menu2, parameter4 );
    _5key_setmenu( menu2, parameter5 );
    _5key_setmenu( menu2, parameter6 );
    _5key_setmenu( menu3, parameter7 );
    _5key_setmenu( menu3, parameter8 );
    _5key_setmenu( menu3, parameter9 );
    background();
  }
  indirect background(){
    request(TASK1);
    while(1);
  }
  indirect task1(){
    _5key_menu(); // endless 5key service
  }
```

Refer to the ***Dynamic C Function Reference*** manual for more information on function use..

Updating and Monitoring Parameters

The old five-key system supports **float** values, **integer** values, **Boolean** values, and **time** and **date** character strings. Parameters can be *modified* or *monitored* under the five-key system. When data are monitored, they are presumed to be modified somewhere else in the program. Their values are displayed when changed.

Each data type is serviced with a different function call, but the functions are similar to one another. In the following example, let the **float** parameter **Tb** (boiler temperature) be used for setting the temperature of a steam boiler.

These declarations apply.

```
#define MODIFY          1
#define NOT_MODIFY      0
#define DISPLAY         1
#define NO_DISPLAY      0
#define RTCLK        0x10
#define NO_RTCLK     0x00
#define NO_HELP      (char**)0
#define NO_FUNCTION  (int(*)())0
float Tb;
char *Tb_help[] = { "Tb is the temperature",
                      "of boiler 3. Use UP, ",
                      "DOWN and FIELD keys",
                      "to modify"};
```

Code-Driven Approach

The function call to change the variable **Tb** would be

```
_5key_float( "Tb", &Tb, 300.00, 100.00,
               Tb_help, sizeof(Tb_help),
               MODIFY, 10 );
```

or if the variable is simply monitored,

```
_5key_float( "Tb", &Tb, 300.00, 100.00,
               Tb_help, sizeof(Tb_help),
               NOT_MODIFY, 10 );
```

The following declarations are needed for a string **time** that displays and allows changes to the real-time clock.

```
char  time[9];
char *time–help[] = { "time display and set",
                        "the real-time clock " };
```

The function call would be the following.

```
_5key_time( "time", time,
              time_help, sizeof(time_help),
              1, MODIFY, 10 );
```

Linked-List Approach

The two variables previously mentioned, **Tb** and **time**, would be added to the linked list with the following calls.

```
_5key_setmenu( "Menu1", "Tb ", _5key_Fdata, &Tb,
                     300.00, 100.00,
                     Tb_help, sizeof(Tb_help),
                     RTCLK|MODIFY, 10, NO_DISPLAY );
_5key_setmenu( "Menu1", "time", _5key_Tdata, time,
                     0.0, 0.0,
                     time_help, sizeof(time_help),
                     RTCLK|MODIFY, 10, NO_DISPLAY );
```

Monitoring Function Keys

The five-key system monitors the function keys F1, F2, F3 and F4. Programmers may write functions that correspond to each function key. Load function key handlers using the procedure **_5key_setfunc**. For example, with the call

```
_5key_setfunc( test1, test2, NO_FUNCTION, test1 );
```

function **test1** is executed whenever F1 or F4 is pressed. Function **test2** is executed whenever F2 is pressed. Pressing F3 will not cause anything to happen. Function-key service routines can control the display and the keypad.

Monitoring Help Keys

An item-specific help message can be displayed when the HELP key is pressed. If no help message is needed, a call like the following will disable help messages.

```
_5key_float( "Tb ", &Tb, 300.00, 100.00,
                 NO_HELP,0, MODIFY, 10 );
```

Periodic Display

When there is no keypad and no display activity for 1000 ticks (25 seconds), the five-key system will display the **time**, **date**, up to 10 messages and up to 10 linked-list parameters. The time and date are always enabled. String messages can be set up with the following call.

```
_5key_setmsg( message_no, "the message" );
```

The **message_no** can be from 0 to 9. The message corresponding to the **message_no** can be changed at any time. Passing **NULL** instead of a message will turn off that particular message.

Linked-list parameters can also be displayed. They are set through the **_5key_setmenu** calls. The previously mentioned boiler temperature, **Tb**, can be displayed periodically with this call.

```
_5key_setmenu( "Menu1", "Tb ", _5key_Fdata, &Tb,
                  300.00, 100.00,
                  Tb_help, sizeof(Tb_help),
                  MODIFY, 10, DISPLAY );
```

If the menu setup is code-driven, it is still possible to use the five-key linked list, but **_5key_menu** cannot be called.

Software Alarms

Four variables, **_ALARM1**, **_ALARM2**, **_ALARM3**, and **_ALARM4**, are monitored by the five-key system. When any of these software alarms becomes nonzero, the five-key system resets it to zero, and calls a specified function. It handles the functions keys the same way, and except for alarms, the triggers are generated in software. The alarm service is particularly useful when an alarm condition has to transfer control of the display and keypad to a predefined function. Alarm handlers might be loaded as follows.

```
_5key_setalarm( alarm1,alarm2,alarm3,NO_FUNCTION );
```

Here, function **alarm1** is executed whenever Alarm 1 signals. Alarms 2 and 3 each have handlers. However, Alarm 4 has no function. Nothing will happen when Alarm 4 signals.

5-Key Support Functions

The functions listed here may be used to develop a menu system specific

These functions are described in the ***Dynamic C Function Reference*** manual.

to an application.

Some of the functions (for example, **_5key_float**) return an integer representing one of the keys MENU, ITEM, UP, DOWN, ADD or DELETE. These integers are also defined as symbolic constants in **5KEY.LIB**. These functions return such a value to indicate that the particular key has been pressed. They return –1 when no key has been pressed or the value is being monitored.

When data are monitored, they are presumed to be changed somewhere else in the program. The PK2100 display reflects the change when monitored data changes.

The following list groups the five-key functions.

Initialization Functions

```
_5key_setmenu
_5key_setalarm
_5key_setfunc
_5key_setmsg
```

Five-Key Service Functions

```
_5key_float
_5key_integer
_5key_boolean
_5key_time
_5key_date
_5key_menu
-5key_boolean()
```

Extended Five-Key Service Functions

```
_5key_12out
_5key_uinput
_5key_diginput
_5key_dacout()
```

Miscellaneous Functions

```
lcd_server
_5keysettime
_5keysetdate
_5keygettime
_5keygetdate
_5key_init_menu
```

Symbols

#define 39, 43
#INT_VEC 67, 72
_5key_float 91, 92
_5key_menu 89, 92
_5key_setalarm 92
_5key_setfunc 91
_5key_setmenu 89, 92
_5key_setmsg 91
_ALARM1 ... _ALARM4 92
_GLOBAL_INIT 24, 27, 40
_prot_init 54
_prot_recover 54
5KEY.LIB 88, 92
5KEYEXTD.LIB 88

A

abort 24, 25, 27, 30, 33, 35
ADD key 56, 57
alarm functions 88
always_on 25, 27, 35
assembly language 15

B

background task
  SRTK 42
battery-backed RAM 53, 54
baud rate 12, 67, 77
BINARY mode 65
BIOS 53
buffer
  receive 68, 69, 77, 78
    initialization 67
    reading 68
  transmit 69, 78
    initialization 67
    writing 68, 69

C

CCVer 53, 54
check sum 76, 77, 78
  computing 73
check_opto_command 77
checking for modem commands 72
ChkSum 27
ChkSum2 27
CoBegin 26, 27, 30
CoData 25, 28, 30
  description 25, 26, 27
  initialization 24
COM ports
  ISA 66
COMMAND mode 65
command protocol
  master-slave 76
communication
  RS-232 .... 64, 67, 68, 69
  RS-485 .... 76, 77, 78, 81
  serial 64, 67, 76, 77, 78, 81
    master-slave 77, 78
comp48 46
compile-time interrupt directive 72
connections
  RS-485 two-wire network 76
content 27, 28
controller execution speed
  arithmetic 84
  logical decision making 84
cooperative multitasking 16, 17
CoPause 30
CoReset 26, 27, 30, 33, 34
CoResume 30
costate 25, 30, 32, 33, 35

costatements 16, 21, 22, 23, 26, 42
abort ... 30
always_on ... 25
delay functions ... 29
dependencies ... 32
error exit ... 33
firsttime flag and functions ... 28
granularity ... 31
init_on ... 25
named ... 25
nested ... 33
problems with C expression evaluation ... 35
syntax ... 25
timing considerations ... 31
unnamed ... 25
waitfor ... 29, 31, 35, 36
yield ... 29
CRC (cyclic redundancy check) 76, 77, 78
computing ... 73
CSIO ... 73
CSState ... 26, 27
CTS ... 67, 68
CTS/RTS ... 65
cyclic redundancy check 76, 77, 78
computing ... 73

D

DATA mode ... 65
Ddelay_100ms ... 72
deciphering modem commands 71
DEL key ... 57
delay
modem communications ... 72
delay functions ... 29
delay_1sec ... 72
DelayMs ... 27, 28, 36, 38, 43
DelaySec ... 27, 38, 43
DelayTicks ... 27, 38, 39, 43
DELETE key ... 56
DEMO_RT.C ... 49
Dget_modem_command ... 71
Dinit_com2 ... 66
Dinit_sca ... 70
Dinit_scb ... 66
Dinit_uart ... 66
Dinit_z0 ... 66
disabling interrupts ... 73
DMA channels ... 73
Z180 Serial Channels 0 and 1 73
disabling the RS-485 driver ... 78
Dkill_z0 ... 69
DMA channels
disabling interrupts ... 73
DOWN key ... 56
downloading
data ... 70
programs ... 72
Dread_sca ... 70
Dread_z0 ... 68
Dread_z01ch ... 68
Dreset_z0rbuf ... 69
Dreset_z0tbuf ... 69
Drestart_z0modem ... 72
Dscasend_prompt ... 70
Dwrite_sca ... 70
Dwrite_z0 ... 69
Dwrite_z01ch ... 68
Dxmodem_z0down ... 70
Dxmodem_z0up ... 70
Dynamic C
Function Reference ... 89
programming port ... 72
Dz0_circ_int ... 72
Dz0modem_chk ... 72
Dz0send_prompt ... 69

E

echo option ... 67
EEPROM ... 53, 72
error exits ... 33
use of setjmp and longjmp ... 34
ERROR_EXIT ... 52
exception ... 52

execution speed
- other operations 85

expression evaluation
- and costatements 35

F

F1, F2, F3, F4 .... 56, 57
failure detection and recovery .. 54
- hardware failures 52
- hardware watchdog 52
- power failure 52
- protected variables 54
- reset 53
- software failures 52
- super reset 53

fastcall 17, 38, 39, 42
FIELD key 56
firsttime 27, 28
- flag 28
- functions 28
  - calling 28
  - definition of 28

Five-Key system 55, 56, 63, 75
- DOWN key 56
- extending 57
- FIELD key 56
- HELP key 56, 57
- ITEM key 56
- linked-list driven 91
- MENU key 56
- operation 56
- UP key 56

FK.LIB 88
fk_helpmsg 59
fk_item_alpha 60
fk_item_enum 61
fk_item_int 60
fk_item_setdate 60
fk_item_settime 60
fk_item_uint 60
fk_monitorkeypad 57, 59
FKSAMP.C 59
flash EPROM 53

G

getcrc 73
gettimer 46
global initialization 40
- initializing CoData 24

granularity 39
- costatement 31

H

hardware watchdog 52
Hayes Smart Modem 65, 70, 71
HELP key 56, 57, 91

I

identifying shared variables 18
init_kernel 46
init_on 25
init_srtkernel 42
initialization
- receive buffer 67
- transmit buffer 67
- Z180 Port 1 77

initiation
- serial transmission 68, 69, 78

input
- RS-232 68

INT1 .... 72, 73
INT2 73
interrupt-driven transmission 78
interrupts 14, 18
- #INT_VEC 67
- disabling 73
- latency 15
  - causes 14
- nested 15
- routines 15
- SER0_VEC 67
- serial 67
- service functions 72
- service routines 72, 79
- vector table 72

ISA
- COM ports 66

isCoDone 30
isCoRunning 30
ITEM key 56
IX register 47

K

kernel
  real-time . 17, 39, 42, 43, 48, 70, 72, 78, 88
keyboard
  see Five-Key system 55, 56, 63, 75

L

lastlocADDR 27
lastlocCBR 27
latency 14
LCD
  see Five-Key system 55, 56, 63, 75
longjmp 34

M

master message format 76, 77
master-slave
  command protocol 76
  networking 79, 81
  serial communication 77, 78
  software support 77
memory
  extended
    uploaded data 70
  random access 53, 54
MENU key 56
misticware 78
modem commands 71
  deciphering 71
modem communication 67
  checking for commands 72
  delay 72
  restarting 72
multitasking 16, 44
  cooperative 16, 17
  preemptive 16, 17, 18
  priority levels 18
  shared variables 17, 18

N

N_WATCHDOG 39, 40
nested costatements 33
  example 34
network connections
  two-wire RS-485 76
ninth-bit address protocol 76
ninth-bit binary communication 77, 78
NMI (nonmaskable interrupt) ... 52
NO_CARRIER 65
nodebug 48
nonmaskable interrupt (NMI) ... 52
NTASKS 39, 44
number of bits 67

O

old Five-Key system 92
  alarm functions 92
  changing parameters 90
  code-driven 88, 90
  data types 90
  function keys 91
  HELP key 91
  linked-list-driven 89
  monitoring data 91
  monitoring parameters 90
  string messages 91
  time
    and date 91
op_init_z1 77
op_kill_z1 78
op_rec_z1 78
op_send_z1 78
opto-22 binary protocol 76, 77, 78, 79

optodelay ... 78
output
  RS-232 ... 68, 69
  RS-485 ... 76, 77

P

parity ... 67
phy_adr ... 71
PLCBus ... 72
PLU
  Program Loader Utility ... 72
preemption ... 16, 17, 18
  with fastcall ... 39
preemptive multitasking 16, 17, 18
printf ... 49
problems with C expression evaluation ... 35
Program Loader Utility
  PLU ... 72
programmable reload timer (PRT) 43, 73
programming ... 56
protected variables ... 14, 15
  crash recovery ... 54
protocol
  command
    master-slave ... 76

R

RAM
  battery-backed ... 53, 54
rbuf_there ... 78
read-only memory ... 72
real-time kernel (RTK) 17, 38, 39, 42, 43, 48, 70, 72, 78, 88
  and the old Five-Key system 88
  and virtual driver ... 43
  array of RTK task pointers ... 43
  sample programs ... 49
real-time programming ... 13
receive buffer ... 64, 68, 69, 77, 78
  initialization ... 67
  reading ... 68
reload_vec ... 72
replyOpto22 ... 78
request ... 45, 46, 88, 89
reset ... 53
resetZ180int ... 73
restarting modem communication 72
restrictions on use of suspend ... 47
rkernel ... 48
ROM
  programmable ... 72
RS-232 communication 64, 67, 68, 69
  features ... 64
    circular buffers ... 64
    CTS/RTS Control ... 65
    modem communication ... 65
    XMODEM file transfer ... 65
  interrupt handling ... 67
  serial inputs ... 68
  serial outputs ... 68, 69
  software support ... 66, 67
RS-485 communication 76, 77, 78, 81
  disabling driver ... 78
  drivers ... 73
  serial outputs ... 76, 77
  two-wire network connections 76
RTK (real-time kernel) 17, 39, 42, 43, 48, 70, 88
RTK.LIB ... 44
RTS ... 67, 68
run_after ... 46
run_at ... 46
run_cancel ... 46
run_every ... 44, 45, 46, 47
run_timer ... 48
RUNKERNEL .. 39, 42, 43, 44, 72
runwatch ... 49

S

sample programs
  communication 69
  Five-Key system 59
  master-slaves 79, 80, 81
  real-time kernel (RTK) 49
SCC 64, 66, 70
sendOp22 77
SER0_VEC 67
serial communication ... 64, 67, 76, 77, 78, 81
  master-slave 77, 78
serial transmission
  initiating 68, 69
  terminating 69
setjmp 34
shared variables 14, 15, 18
  identifying 18
  multitasking 17, 18
simplified real-time kernel (SRTK) 17, 39, 42, 43, 57, 88
  sample program 42
SIO 64, 66
slave response format 76, 78
Smart Modem
  Hayes 65, 70
software support
  RS-232 67
SRTK (simplified real-time kernel) 17
SRTK.LIB 42
srtk_hightask 42
srtk_lowtask 42
stack corruption checking ... 47, 48
state machine 23
  example 22
stop bits 67
string messages
  in the Five-Key system 91
super reset 53
suspend 17, 45, 46, 47, 48, 49, 70, 72, 78
syntax
  costatement 25

T

TASKSTORE_SIZE 47, 48
time and date
  in the old Five-Key system ... 91
timer
  watchdog 40, 52
timers
  PRT 73
transmission
  initiating 68, 69, 78
  interrupt-driven 78
transmit buffer 69, 78
  initialization 67
  writing 68, 69
two-wire connections
  RS-485 network 76

U

UART 72
UP key 56
uplc_init
  initialize CoData structures ... 24
uploading data 70
useix 47

V

VD_FASTCALL 39
vd_initquickloop 39
vd_quick_loop 39
VdGetFreeWd 40
VdInit 38, 39, 40
  and waitfor delay functions .. 29
  initialize CoData structures ... 24
VdReleaseWd 40
VDRIVER.LIB 38
VdWdogHit 40
virtual driver 37, 43, 53
  and real-time kernel 39
  fastcall 39
  global initialization 40
virtual watchdog 38, 40, 53

W

waitfor .. 25, 27, 28, 29, 31, 32, 33, 35, 36, 39
watchdog timer 40, 52

X

xdata 71
XMODEM
  commands 70
  protocol 64, 65, 70
XP8700 .... 66, 72

Y

yield 24, 25, 27, 29

Z

Z0 72
z0binaryreset 68
z0binaryset 68
z1_op_int 79
Z180 .... 64, 66
  Port 0 .... 67, 68, 69, 72
  Port 1 .... 78, 79
    initialization 77
  Serial Channels 0 and 1
    disabling interrupts 73

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

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

Printed in U.S.A.