



Dynamic C[®] 32

for Zilog Z180 microprocessors

Version 6.x

Integrated C Development System

Technical Reference

019-0083 • 020330-B

Dynamic C 32 v. 6.x Technical Reference

Part Number 019-0083 • 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.
 - Modbus® is a registered trademark of Modicon, Inc.
 - Hayes Smart Modem® is a registered trademark of Hayes Microcomputer Products, Inc.
-

Notice to Users

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

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

Company Address



Z-World, Inc.

2900 Spafford Street
Davis, California 95616-6800 USA

Telephone: (530) 757-3737

Facsimile: (530) 753-5141

Web Site: <http://www.zworld.com>

E-Mail: zworld@zworld.com

TABLE OF CONTENTS

About This Manual	ix
Chapter 1: Installing Dynamic C	13
Installation Requirements	14
Installation Procedure	14
Chapter 2: Introduction to Dynamic C	15
Why C?	16
The Nature of Dynamic C	16
Speed	17
Dynamic C is Different	17
How Dynamic C Differs	18
Initialized Variables	18
Function Chaining	19
Global Initialization	20
Costatements	21
Interrupt Service Routines	21
Embedded Assembly Code	21
Shared and Protected Variables	22
Extended Memory	22
External Functions and Data	22
Function-Calling Methods	23
Subfunctions	24
Enumerated Types	24
Default Storage Class	24
Dynamic C and Z-World Controllers	24
Physical Memory	25
Watchdog Timer	25
Real-Time Operations	25
Restart Conditions	25

Chapter 3: Using Dynamic C	27
Installation	28
Writing Programs	28
Compiling Programs	28
Compiler Options	30
Debugging Programs	32
Polling	33
Disassembler	34
Single Stepping	34
Break Points	35
Watch Expressions	35
Returning to Edit Mode	36
Creating Stand-alone Programs	36
Controller with Program in EPROM	37
Controller with Program in Flash Memory	37
Controller with Program in RAM	37
Help	37
Function Lookup	37
Function Assistance	38
Chapter 4: Dynamic C Environment	39
Editing	40
Menus	41
File Menu	42
Edit Menu	46
Compile Menu	50
Run Menu	52
Inspect Menu	54
Options Menu	58
Window Menu	68
Help Menu	71
Chapter 5: The Language	75
Overview	76
Program Files	76
Support Files	77
Statements	78
Declarations	78
Functions	78
Prototypes	79
Type Definitions	80
Modules	82

Macros	84
Program Flow	86
Loops	86
Continue and Break	87
Branching	89
Data	91
Primitive Data Types	91
Aggregate Data Types	92
Storage Classes	94
Pointers	94
Argument Passing	95
Memory Management	96
Memory Partitions	97
C Language Elements	99
Keywords	99
Names	112
Numbers	112
Strings and Character Data	113
Operators	115
Directives	124
Punctuation	127
Extended Memory Data	127

Chapter 6: Using Assembly Language 129

Register Summary	131
General Concepts	131
Comments	132
Labels	132
Defining Constants	132
Expressions	133
Special Symbols	133
C Variables	134
Standalone Assembly Code	135
Embedded Assembly Code	135
No IX, Function in Root Memory	136
Using IX, Function in Root Memory	138
No IX, Function in Extended Memory	139
C Functions Calling Assembly Code	140
Assembly Code Calling C Functions	142
Indirect Function Calls in Assembly	143
Interrupt Routines in Assembly	143
Common Problems	145

Chapter 7: Costatements	147
Overview	148
Syntax	150
Name	150
State	151
Waitfor	151
Yield	153
Abort	154
The CoData Structure	155
The Firsttime Flag and Firsttime Functions	157
Advanced CoData Usage	158
Chapter 8: Interrupts	161
Interrupt Vectors	164
Chapter 9: Remote Download	167
The Download Manager	169
Enter Password	169
Set Password	169
Report DLM Parameters	169
Download Program	170
Execute Downloaded Program	170
Hangup Remote Modem	170
The DLM Code	170
The Downloaded Program (DLP)	171
How to Use the DLM	172
The DLP File Format	173
Chapter 10: Local Upload	175
The Program Loader Utility	176
On-line Help	177
Set Communication Parameters	177
Reset the Target Controller	178
Select the Program File	179
Common Problems	180

Appendix A: Run-Time Error Processing	181
Long Jumps	184
Watchdog Timer	184
Protected Variables	185
Appendix B: Efficiency	187
Nodebug Keyword	188
Static Variables	188
Execution Speed	189
Subfunctions	189
Function Entry and Exit	190
Appendix C: Software Libraries	191
Headers	193
Function Headers	194
Modules	194
Appendix D: Extended Memory	197
Physical Memory	198
Memory Management	198
Memory Partitions	200
Control over Memory Mapping	202
Extended Memory Functions	202
Suggestions	203
Extended Memory Data	204
Appendix E: Compiler Directives	207
Default Compiler Directives	208
Appendix F: File Formats	209
Layout of ROM Files	210
Layout of Downloadable Files	210
Layout of Download to RAM Files	211
Hex File Information	212
Jumping to Another Program	213
Burning ROM	213
Copyright Notice	213

Appendix G: Reset Functions **215**
Reset Differentiation 216
Reset Generation 218

Appendix H: Existing Function Chains **219**

Appendix I: New Features **221**
Dynamic C 32 IDE 222
 Compiler Options, Output Generation Group 222
 Compiler Options, File Type for ‘Compile to File’ Group 222
 Target Communication 222
New Libraries 222
Program Loader Utility 222

Index **223**

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

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

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

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 Z80 assembly language and architecture.



For documentation from Zilog, refer to the following texts.

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

Acronyms

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







Table 1. Acronyms

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

Icons

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

Table 2. Icons

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

Conventions

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

Table 3. Typographical Conventions

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



CHAPTER 1: *INSTALLING DYNAMIC C*

Installation Requirements

Your PC must meet the following requirements in order to successfully install and use Dynamic C 32.

- CPU is '386SX or higher (Pentium or higher is recommended)
- OS is Windows 95, 98, 2000, Me or NT
- CD-ROM (or compatible) drive
- hard drive with at least 32 megabytes of free space
- at least 16 megabytes of RAM
- at least one free COM port

Installation Procedure

Dynamic C 32 software comes on one CD-ROM. Insert the CD-ROM into the appropriate drive on the PC. After a few moments the installation program should start automatically. If not, then issue the Windows "Run..." command and type the following command.

```
<disk>\SETUP
```

where <disk> is the name of the CD-ROM drive. If the CD-ROM drive is "D:" then type

```
D:\SETUP
```

The installation program will begin to run and guide you through the installation process. Note that you will be asked to read and understand the Z-World Software End User License Agreement. If you decline to accept the terms of the agreement then the installation program will terminate without installing Dynamic C 32.

When installation is complete a new Windows program group that includes Dynamic C 32, the Program Loader Utility, and an on-line help file will have been created. In addition, desktop shortcut icons for these items have also been created.

The Dynamic C 32 application may now be run from Windows using any of the standard methods (E.G: double-clicking on the icon) to launch it.



Please contact Z-World's Technical Support at (530)757-3737 if there are any problems.



CHAPTER 2:
INTRODUCTION TO DYNAMIC C

Dynamic C is an integrated development system that runs on an IBM-compatible PC and is designed for use with Z-World controllers and control products.

Z-World's Zilog Z180 microprocessor based controllers include a variety of analog inputs and outputs, digital inputs and outputs, high-current outputs, serial communication channels, clocks and timers. Z-World controllers are programmed using an enhanced form of the well-known C programming language... Dynamic C.

Why C?

Programmable controllers provide the most flexible way to develop a control system. And C is the preferred language for embedded systems programming. It is widely known and produces efficient and compact code. Because C is a high-level language, code can be developed much faster than with assembly language alone. And C allows programming at the machine level when necessary.

The Nature of Dynamic C

Dynamic C integrates the following development functions

Editing, Compiling, Linking, Loading, Debugging

into one program. In fact, compiling, linking and loading are one function. Dynamic C has an easy-to-use built-in text editor. Programs can be executed and debugged interactively at the source-code level. Ultimately, EPROM files or down-loadable files can be created for programs that will run stand-alone in the controller. Pull-down menus and keyboard shortcuts for most commands make Dynamic C efficient.

Because all the development functions are integrated, it is possible to switch from one function to another with a simple keystroke.

Dynamic C also supports assembly language programming. It is not necessary to leave C or the development system to write assembly language code. C and assembly language may be mixed, line by line, in a program.

For debugging, Dynamic C provides a standard I/O window, an assembly window, a "watch" window, a register window and a stack window. The standard I/O window allows the program in a controller to print messages on the development screen. The assembly window displays an assembly view of compiled code. The watch window allows the programmer to type and evaluate expressions, monitor or set variables, and call functions. Dynamic C's debugger allows breakpoints to be set and cleared on-the-fly, to single-step with and without descent into functions, and to view execution at the assembly level as well as at the source-code level.

Dynamic C provides extensions to the C language (such as *shared* and *protected* variables) that support real-world system development. Interrupt service routines may be written in C. Dynamic C supports real-time multi-tasking with its real-time kernel and its *costatement* extension.

Dynamic C comes with many function libraries, all in source code. These libraries support real-time programming, machine level I/O, and provide standard string and math functions.



Please refer to the Dynamic C *Application Frameworks* and *Function Reference* manuals.

Speed

Dynamic C compiles directly to Z180 memory. Functions and libraries are compiled and linked and downloaded on-the-fly. On a fast PC, Dynamic C can compile more than 250 lines of source code per second, generating about 2500 bytes of machine code per second. Thus, a large program—say 8,000 lines of code—might generate 80 KBytes of machine code and take about 30 seconds to compile and download.

The application code might only be 400 lines, yet it can make calls to several thousand lines of library code, all of which are compiled when the program is compiled.

Dynamic C is Different

Dynamic C differs from a traditional C programming system running on a PC or under UNIX. The motivation for being different is to be better: to help customers write the most reliable *embedded* control software possible. Some of the devices and constructs that C programmers employ on other systems just don't work very well for embedded systems. At the very least, they must be used with caution. In some instances, Z-World has extended the C language where the value of the extension is compelling.

In an embedded system, there is no operating system or supervisor that can halt a program if it goes wrong or perform services for the program. An embedded program has to do it all, and handle its own errors and keep on running. An embedded program also has to initialize itself.

In an embedded system, a program runs from EPROM (or flash) and uses a separate RAM for data storage. Many Z-World controllers have battery-backed RAM providing nonvolatile storage.

Often, an embedded program comprises a number of concurrently executing tasks, rather than a single task.

How Dynamic C Differs

The differences in Dynamic C are summarized here and are discussed after the summary.

- Variables that are initialized when declared are considered *named constants* and are placed in ROM. It is an error to try to change such “variables.”
- The default storage class is **static**, not **auto**.
- There is no **#include** directive, nor are there any include (header) files. Library functions and data are bound to a program by other means. There is a **#use** directive.
- Dynamic C does not support enumerated types.
- The **extern** and **register** keywords have an altered meanings.
- Function chaining, a concept unique to Dynamic C, allows special segments of code to be included within one or more functions. When a named function chain executes, all the segments belonging to that chain execute. Function chains allow software to perform initialization, data recovery, or other kinds of tasks on request.
- “Cstatements” allow concurrent parallel processes to be simulated in a single program.
- Dynamic C allows the programmer to write interrupt service routines in C.
- Dynamic C supports embedded assembly code.
- Dynamic C has *shared* and *protected* keywords that help protect data from unexpected loss.
- Dynamic C has a set of features that allow the programmer to make fullest use of extended memory.
- Dynamic C provides two forms of argument passing (using the IX index register vs. using the stack pointer SP).
- Dynamic C provides a **subfunc** construct to optimize frequently used code.

Initialized Variables

Static variables initialized when they are declared are considered *named constants*. The compiler places them in the same area of memory as program code, typically in EPROM or flash memory. Uninitialized variables are placed in RAM, and must be initialized by the application program.

```
int i = 100;    // initialized in declaration here,  
               // becomes a named constant
```

```

int k;           // variable placed in RAM, then
k = 100;        // initialized by your program.

```

When a program is being compiled directly to a controller that has EPROM, the compiler places constants and program code in RAM since it cannot modify the controller's EPROM. Under these circumstances, constants can be modified, intentionally or not, but it is an error to do so.



The default storage class for local variables is *static*, not *auto*, so be doubly careful when initializing variables in functions.

Function Chaining

Function chaining, a concept unique to Dynamic C, allows special segments of code to be distributed in one or more functions. When a named function chain executes, all the segments belonging to that chain execute. Function chains allow the software to perform initialization, data recovery, or other kinds of tasks on request.

Dynamic C provides two directives, **#makechain** and **#funcchain**, and one keyword, **segchain**.

- **#makechain** *chain_name*
Creates a function chain. When a program executes the named function chain, all of the functions or chain segments belonging to that chain execute. (No particular order of execution can be guaranteed.)
- **#funcchain** *chain_name name*
Adds a function, or another function chain, to a function chain.
- **segchain** *chain_name { statements }*
Defines a program segment (enclosed in curly braces) and attaches it to the named function chain.

Function chain segments defined with **segchain** must appear in a function directly after data declarations and before executable statements, as shown below.

```

my_function() {
    data declarations
    segchain chain_x{
        some statements which execute under chain_x
    }
    segchain chain_y{
        some statements which execute under chain_y
    }
    function body which executes when my_function
    is called
}

```

A program will call a function chain as it would an ordinary void function that has no parameters. For example, if the function chain is named `recover`, this is how to call it.

```
#makechain recover
...
recover();
```

Several function chains can be found in Dynamic C libraries. These are listed in *Appendix X*.

Global Initialization

Embedded systems typically have no operating system to perform services such as initialization of data—something programmers who are accustomed to an operating system might take for granted.

Various hardware devices in a system need to be initialized not only by setting variables and control registers, but often by complex initialization procedures. For this purpose, Dynamic C provides a specific function chain: `_GLOBAL_INIT`.

Any global initialization may be performed by adding segments to the `_GLOBAL_INIT` function chain, as shown in this example.

```
int my_func( long j ){
    int    b = 100;
    int    c, z;
    char*  k;

    segchain _GLOBAL_INIT{
        c = 40; k = "Press any key...";
        for( z = 0, z < c; z++ ){
            ...
        }
    }

    your function code
    ...
}
```

Then, have the program call `_GLOBAL_INIT` during program startup, or when the hardware resets. This function chain executes all the global initialization statements indicated by all `_GLOBAL_INIT` segments in the program (and in Dynamic C libraries as well).

Z-World supports two levels of initialization. A *major* initialization, or *super* initialization, takes place only when there is a need to erase all past history, such as when installing a new program EPROM, or when a system loses its memory. A *minor*, or *normal*, initialization taking place every time the system resets or powers up. In a minor initialization, exactly which data are (re)initialized depends on the nature of the system.



For further detail, refer to Appendix G, Reset Functions, and to the Dynamic C *Application Frameworks* manual.

Costatements

Dynamic C provides a capability whereby the program can execute a set of tasks (almost) simultaneously. A data structure, some additions to the C language, and some functions comprise what Z-World calls *costatements*. A costatement is a construct—a block of code—that can suspend its own execution, thereby allowing other code to execute. A *set* of costatements, presumably in an endless loop, executes concurrently, seemingly in parallel. All of the tasks in the set are in states of partial completion.

Costatements may execute repeatedly, or execute once, when triggered, and then stop.



For further detail, refer to Chapter 7, Costatements, in this manual, and to the Dynamic C *Application Frameworks* manual.

Interrupt Service Routines

Interrupt service routines can be written in Dynamic C using the C language. The keyword `interrupt` designates a function as an interrupt service routine.

```
interrupt my_handler() {  
    ...  
}
```

Embedded Assembly Code

There are times when assembly language is necessary or desirable. For time-critical or machine-dependent code, it is natural to choose assembly language.

Dynamic C allows Z180 assembly code to be embedded in a C program. Assembly code may be written within a C function or complete assembly code functions may be written. C-language statements may also be embedded in assembly code.



For further detail, refer to Chapter 6, Using Assembly Language.

Shared and Protected Variables

An important feature of Dynamic C is the ability to declare variables as *protected*. Such a variable is protected against loss in case of a power failure or other system reset because the compiler generates code that creates a backup copy of a protected variable before the variable is modified. If the system resets while the protected variable is being modified, the variable's value can be restored when the system restarts.

A system that shares data among different tasks or among interrupt routines can find its shared data corrupted if an interrupt occurs in the middle of a write to a multi-byte variable (such as type `int` or `float`). The variable might be only partially written at its next use.

Declaring a multi-byte variable *shared* means that changes to the variable are *atomic*, that is, any change to the variable is a complete change. (Interrupts are disabled while the variable is being changed.)

Extended Memory

Dynamic C supports the 1-MByte address space of the Z180 microprocessor. The address space of the Z180 is segmented by a memory management unit. Dynamic C allows programs containing up to 512 KBytes in ROM (code and constants) and 512 KBytes of RAM (data). Normally, Dynamic C takes care of memory management, but there are instances where the programmer will want to take control of it.

Dynamic C has keywords and directives to help put code and data in the proper place. The keyword `root` selects root memory (addresses within the 64-KByte physical address space of the Z180). The keyword `xmem` selects extended memory, which means anywhere in the 512-KByte code space. The directive `#memmap` allows further control. Special statements `xdata` and `xstring` declare blocks of data in extended memory. Certain functions, such as `xgetfloat` and `xstrlen` help to access data in extended memory.



Refer to Appendix D, Memory Management, and to the `XMEM.LIB` library in the Dynamic C *Function Reference* manual for more details.

External Functions and Data

The keyword `static` cannot apply to functions.

The keyword `extern` is used in module headers (those enclosed in `BeginHeader` and `EndHeader` comments. A variable or function may already be declared `extern` in your a controller's BIOS symbol table. Otherwise, declare a variable `extern` if it is to be defined later in the program or in another file.

Two files in the Dynamic C directory—`LIB.DIR` and `DEFAULT.H`—contain lists of libraries that are needed for particular controllers. These files are used automatically, but they can be modified if necessary.

Dynamic C has no `#include` directive, but does have a `#use` directive. Whereas the `#include` directive causes program text to be inserted in place of the directive, Z-World's `#use` directive does not cause text insertion, but identifies a library from which functions and data may be taken. The file `DEFAULT.H` contains various sets of `#use` directives, one set for each controller Z-World offers.

Dynamic C compiles, links, and downloads directly to a Z-World controller (or to a file). Dynamic C functions are not compiled separately and then linked. There are no pre-compiled software libraries. Dynamic C uses source-code libraries from which necessary functions are extracted during compilation. Since there are no `#include` directives in Dynamic C, source libraries make global variables and function prototypes available with special headers such as the following.

```
/**/ BeginHeader my_proc, my_func, my_var */
void my_proc( int j );
float my_func( float arg );
extern int my_var;
/**/ EndHeader */
```

These headers are found throughout library source code. Such headers must be created to make functions known to the Dynamic C compiler if other libraries are created.

Function-Calling Methods

Dynamic C provides a choice of two function-calling mechanisms. Two keywords (and two directives), listed in Table 2-1, provide this choice.

Table 2-1. Function-Calling Methods

Key, Directive	Description
<code>useix,</code> <code>#useix</code>	Use the IX register as a 'frame reference pointer' for stack-based variables and arguments. See Chapter 6, Using Assembly Language.
<code>nouseix,</code> <code>#nouseix</code>	Use the stack pointer (SP) as a 'frame reference pointer.' This is the normal case.

It is generally more efficient to use the IX register. Do not use the IX register as a frame reference pointer for functions that can be suspended under the real-time kernel.

Subfunctions

Subfunctions allow often-used code sequences to be turned into an in-line “subroutine” within a C function. The subfunction `nextbyte` in the following example,

```
static char nextbyte();
subfunc nextbyte: *ptr++;
...
...nextbyte(); ...
...nextbyte(); ...
...
```

can save ten or more bytes of code memory each time it is called.

Enumerated Types

Dynamic C does not have enumerated types.

Default Storage Class

Unlike traditional C compilers, the default storage class for local variables is `static`, not `auto`. The default setting may be changed with the directive `#class`.



Attempts to write recursive or re-entrant functions will fail if this default storage class is `static`. Recursive or re-entrant functions require `auto` variables.

Dynamic C and Z-World Controllers

Z-World controllers are based on the Z180 microprocessor, which has an instruction set nearly identical to that of a Zilog Z80. The Z180 is a well-established and popular microprocessor. It is a descendent of the original Z80 microprocessor, but the Z180 also has the following on-chip “peripheral” devices.

- Dual 16-bit programmable timers
- Dual asynchronous serial communication ports
- A clocked serial communication port
- Dual DMA channels for high-speed data transfer between memory and I/O devices.

The Z180 has a relatively efficient instruction set. At 9.216 MHz, many instructions take about 1 microsecond. Floating-point arithmetic is accomplished in software. Floating-point add, subtract and multiply take about 100 microseconds with a 9.216 MHz clock. Division is somewhat slower.

Physical Memory


Depending on the product and its jumper wiring, Z-World controllers can address up to 512 KBytes of ROM, and 512 KBytes of RAM. It is often not necessary to have memory chips this large on miniature controllers. Typical SRAM chips have 32 or 128 KBytes.

Watchdog Timer

Programs sometimes fail or get stuck. Z-World controllers provide a “watchdog” timer that will initiate a hardware reset unless the software signals the timer periodically. A failed program will generally fail to “hit” the watchdog timer. The watchdog timer can help the controller recover from system hang-ups, endless loops and hardware upsets resulting from electrical transients. The watchdog timer provides a natural way to recover from most fatal software errors.

Real-Time Operations

Dynamic C includes two real-time function libraries, and extensions to the C language to support real-time operations.

 Refer to Chapter 7, Costatements, and to the Dynamic C *Application Frameworks* manual for more information about the real-time kernels.

Restart Conditions

Z-World embedded applications need to differentiate the causes of reset and restart. Possible hardware resets are listed in Table 2-2.

Table 2-2. Hardware Resets

Regular Reset	The system /RESET line is pulled low and released.
Power Fail Reset	Power drops below a threshold, the supervisor chip pulls /RESET low and causes a reset.
Watchdog Reset	Software failed to “hit” the watchdog timer. It pulls /RESET low and causes a reset.

In addition to these hardware resets, an application may cause a *super reset*. Z-World’s super reset is a mechanism to initialize certain persistent data in battery-backed RAM. A normal reset does not initialize these data, but *retains* their values. A super reset always occurs when a program is first loaded. Subsequent resets are normal resets, unless the software performs a super reset intentionally.

Dynamic C includes the functions listed in Table 2-3 to differentiate the various resets.

Table 2-3. Reset Functions

<code>_sysIsSuperReset</code>	This function detects whether a super reset was requested. It also manages protected variables and calls the <code>sysSupRstChain</code> function chain.
<code>_sysIsPwrFail</code>	This function determines whether the system had a power failure just before restarting.
<code>_sysIsWDTO</code>	This function determines whether the system was reset by a watchdog timeout.

If these reset functions are to be used, call them before doing anything else in the `main` function.

Dynamic C can generate two types of system reset. The function `sysForceReset` causes a watchdog reset. The function `sysForceSupRst` causes a super reset.



See Appendix G, Reset Functions.



CHAPTER 3: **USING DYNAMIC C**

The Dynamic C 32 compiler can generate up to 512 KBytes of code and 512 KBytes of data, and fully supports extended memory.

To run Dynamic C 32 under Windows, double-click the Dynamic C icon in the Dynamic C program group or use one of the other standard Windows methods to launch Dynamic C 32.

Installation

Dynamic C 32 must be installed on a hard disk and requires about 32 MBytes of disk space. The PC must be running Windows 95, 98, 2000, Me or NT, on a machine having a 386SX processor or better. At least 16 Mbytes of RAM are required to run Dynamic C and there must be one free serial port to communicate with the target controller.



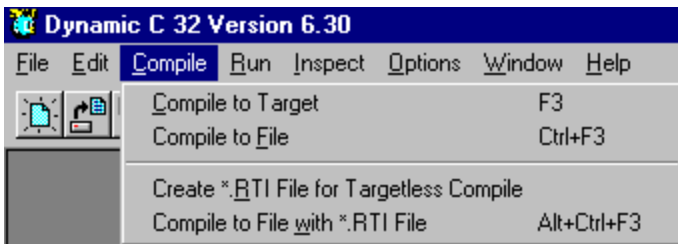
Refer to the installation instructions in Chapter 1, Installing Dynamic C.

Writing Programs

A Dynamic C text window is used to enter the program text line-by-line. Fragments of program text may be cut and pasted from one application to another (for instance, from Microsoft WORD to Dynamic C) or from one Dynamic C text window to another. Dynamic C allows text to be selected and scrolled, and program files can be created and saved using the same techniques as in other Windows programs.

Compiling Programs

Dynamic C provides several ways to compile programs, as shown in the **Compile** menu.



Compile to Target

F3

Dynamic C compiles, links and downloads machine code directly to a target controller. If the controller has flash memory, Dynamic C places code in flash memory. If the controller has EPROM, Dynamic C places code in RAM. Dynamic C communicates with the controller through a PC serial port. If the compilation is successful, Dynamic C enters **run** (AKA **debug**) mode and maintains communication with the target controller.

Compile to File

Ctrl+F3

Dynamic C compiles the program to a file whose nature and format can be selected in the compiler options dialog. No file is generated if compilation errors occur. Note that a controller has to be connected to the PC; **Compile to File** takes target information from the controller.

Create *.RTI File for Targetless Compile

When this menu option is selected, Dynamic C creates a *Remote Target Information* (.RTI) file by saving target information taken from the required attached controller. A dialog box prompts for the location and file name to save the .RTI file.

Compile to File with *.RTI File

Alt+Ctrl+F3

Dynamic C allows a program to be compiled to a binary file or a downloadable file without having a target controller present. Before compiling programs this way, first create a *Remote Target Information* (.RTI) file for the specific controller the programs will run on.

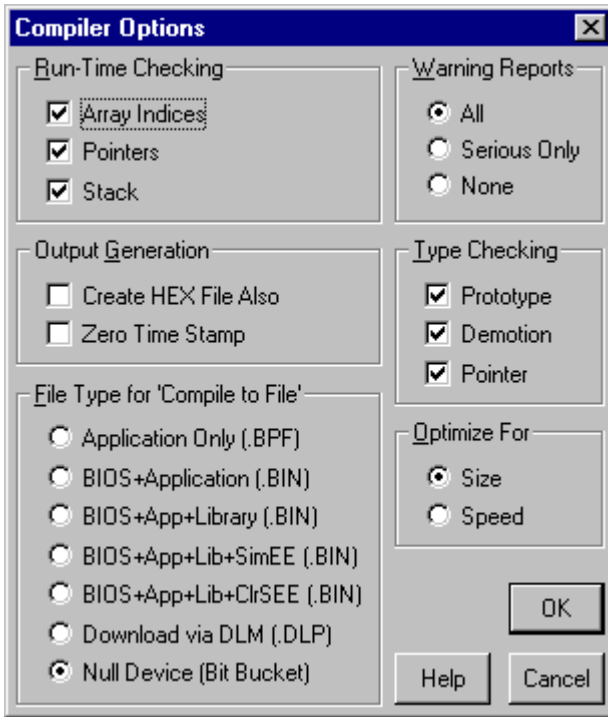


It is essential that the .RTI file is created with a target controller identical (board type, BIOS, memory size, jumper settings, etc.) to the controller on which programs compiled with the .RTI file will run.

Once a suitable .RTI file has been created, the **Compile to File with *.RTI File** command generates output files the same way **Compile to File** does. A dialog box prompts for which .RTI file to open. All compiler options apply.

Compiler Options

The **Compiler** selection on the **Options** menu provides many options.



The **File Type for 'Compile to File'** options specifically affect the **Compile to File** and the **Compile to File with *.RTI File** commands.

Application Only (.BPF)

When this option is selected, the **Compile to File** commands generate a .BPF file containing the compiled application code, but without the BIOS code. This option is included for backward compatibility only, for controllers with standard a EPROM which require the Program Loader Utility to load the application code into SRAM.

The "BIOS+... (.BIN)" Options

These options select what type of a .BIN (EPROM) file is generated. If the **Create HEX File Also** compiler option is selected, the **Compile to File** commands also generate an Intel hex format file (.HEX) in addition to the .BIN file. In most cases, either the .BIN or the .HEX file format will work with external device (standard or Flash EPROM) programmers.

BIOS+Application (.BIN)

When this option is selected, the **Compile to File** commands generate a .BIN (standard or Flash EPROM) file containing the BIOS code of the target controller (where the BIOS library functions are located) and the compiled application code. It is important to note that the BIOS's symbol library is excluded from the .BIN file.

This option is useful for programming either standard or Flash EPROM types. When this type of .BIN file is loaded via the Program Loader Utility (PLU) into a Flash equipped target controller, communication with Dynamic C or the PLU is still possible. This presumes that the .BIN file code does not overwrite the target controller's BIOS symbol library or simulated EEPROM area, located at the end of the Flash.

However, if this type of .BIN file is externally programmed into a blank EPROM which is then installed into the target controller, communication with Dynamic C or the PLU is no longer possible. This is because the BIOS's symbol library is excluded from the .BIN file, which can be useful in situations where extra code security is desired. Also excluded from the .BIN file is the simulated EEPROM area, located at the end of a Flash.

BIOS+App+Library (.BIN)

When this option is selected, the **Compile to File** commands generate a .BIN (Flash EPROM) file containing the BIOS code of the target controller (where the BIOS library functions are located), the compiled application code and the BIOS's symbol library (located near the end of the Flash). It is important to note that the Flash's simulated EEPROM area is excluded from the .BIN file.

This option is useful only for externally programmed Flash types, it **should not** be used to generate .BIN files intended for loading to the target controller via the Program Loader Utility (PLU). After installing the externally programmed Flash into the target controller, communication with Dynamic C or the PLU is still possible. Note that the Flash's simulated EEPROM area may or may not have been modified by the external device programmer.

BIOS+App+Lib+SimEE (.BIN)

When this option is selected, the **Compile to File** commands generate a .BIN (Flash EPROM) file containing the BIOS code of the target controller (where the BIOS library functions are located), the compiled application code, the BIOS's symbol library (located near the end of a Flash EPROM) and the Flash's simulated EEPROM area (located at the end of the Flash). It is important to note that the Flash's simulated EEPROM area is copied from the attached target controller or .RTI file (if available, otherwise it is cleared) into the .BIN file.

This option is useful only for externally programmed Flash EPROM types, it **should not** be used to generate .BIN files intended for loading to the target controller via the Program Loader Utility (PLU). After installing the externally programmed Flash EPROM into the target controller communication with Dynamic C or the PLU is still possible.

BIOS+App+Lib+ClrSEE (.BIN)

When this option is selected, the **Compile to File** commands generate a .BIN (Flash EPROM) file containing the BIOS code of the target controller (where the BIOS library functions are located), the compiled application code, the BIOS's symbol library (located near the end of a Flash EPROM) and the Flash's simulated EEPROM area (located at the end of the Flash). It is important to note that the Flash's simulated EEPROM area is cleared (zeroed) in the .BIN file.

This option is useful only for externally programmed Flash EPROM types, it **should not** be used to generate .BIN files intended for loading to the target controller via the Program Loader Utility (PLU). After installing the externally programmed Flash EPROM into the target controller communication with Dynamic C or the PLU is still possible.

Download via DLM (.DLP)

When this option is selected, the **Compile to File** commands generate a downloadable program file (.DLP) to be used by the Z-World download manager (DLM). When choosing this option, be prepared to enter certain parameters generated by the DLM in a dialog box that appears after **Compile to File** is clicked.



Refer to Chapter 9, Remote Download, for details.

The DLM must be resident in any controller that will receive the downloadable file.

Null Device (Bit Bucket)

When this option is selected, the **Compile to File** commands generate no output. This option allows very fast compilation and is useful just to (1) perform syntax checking, (2) perform type checking or (3) get the sizes of each code and data segment. The memory mapping scheme is identical to compiling with code with BIOS.

Debugging Programs

Once a program has been compiled successfully with a target controller connected, Dynamic C enters *run mode* (also called *debug mode*). Modern symbolic debuggers, such as Dynamic C's debugger, make debugging relatively easy. There are two general methods; expect to use a combination of the two.

1. Make the program report its behavior by including debugging code—such as calls to `printf`—in the program. This is useful, but it is often not sufficient, especially if the `printf` contents scroll off the screen too fast. Dynamic C, however, offers an option to save all the content printed to the **STDIO** window into a file for later examination. This allows the programmer to save a huge file of debug information and then use another program on the PC to analyze the contents.
2. Probe and test the program as it runs. Unfortunately for debugging, programs run faster than humans do. Addressing this difference, Dynamic C lets the program run at a speed amenable to testing. Slow it down here, make it run fast there, and stop whenever needed to examine its state.

Dynamic C provides a variety of windows, listed in Table 3-1, to monitor a program's state.

Table 3-1. Dynamic C Monitoring Windows

Watch window	Evaluates variables, expressions, and functions
STDIO window	Calls to <code>printf</code> display in the STDIO window
Assembly window	Examines, or step, the compiled code
Register window	Shows Z180 register values, past and present
Stack window	Shows the (top 8 bytes of the) processor stack, past and present

The assembly, register, stack, **STDIO**, and watch windows are all scrolling windows. The windows can be scrolled to view the history of contents of registers, stack and watch expressions of the last few steps. This feature is very useful to show how variables, registers or the stack change during execution of the program.

An important aspect of the Dynamic C debugger is that it is symbolic. This means that the executing program is linked to source code. The part of the program that is executing is highlighted in the source-code window. When expressions, variables, and functions are evaluated, they are evaluated in C, using the names in the application, and normal integer, floating, and character representations of constants apply. The execution of the program can also be viewed at the machine level.

Polling

Under normal debugging conditions, Dynamic C monitors the activity on the target controller. The controller is interrupted every 100 milliseconds. This is called polling. If your the application has very tight timing requirements, these interrupts could cause the application to fail. Dynamic C allows polling to be enabled or disabled at the programmer's option.

There are three commands on the **RUN** menu.

Run (with polling),

Run w/ No Polling, and

Toggle Polling (allows user to control polling).

Single Stepping

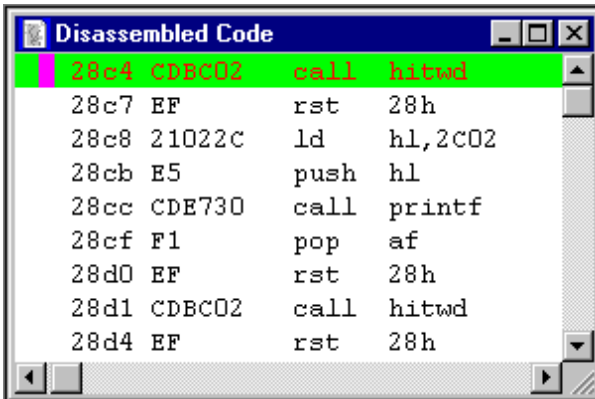
Often there is a need simply to observe the program execute, statement by statement. There are two commands on the **RUN** menu) for *single stepping*.

Trace into (with descent into function calls), and

Step over (prevents descent into function calls).

An execution cursor highlights the current source statement (or assembly instruction, if the assembly window is being used).

When one of the two single-stepping commands is clicked, the current statement executes, debugging windows are updated, and the execution cursor advances to the next statement in the execution sequence. To examine code in greater detail, the assembly window may be activated to show the compiled code in assembly language format.



The screenshot shows a window titled "Disassembled Code" with a list of assembly instructions. The first instruction, at address 28c4, is highlighted in green: "28c4 CDBC02 call hitwd". The other instructions are: "28c7 EF rst 28h", "28c8 21022C ld hl,2C02", "28cb E5 push hl", "28cc CDE730 call printf", "28cf F1 pop af", "28d0 EF rst 28h", "28d1 CDBC02 call hitwd", and "28d4 EF rst 28h".

Address	Hex	Op	Operand
28c4	CDBC02	call	hitwd
28c7	EF	rst	28h
28c8	21022C	ld	hl,2C02
28cb	E5	push	hl
28cc	CDE730	call	printf
28cf	F1	pop	af
28d0	EF	rst	28h
28d1	CDBC02	call	hitwd
28d4	EF	rst	28h

This option may not be needed if the source code is already in assembly language. Single stepping through assembly code is instruction by instruction. The machine state (registers and stack) can also be displayed independently at any time.

Disassembler

Besides displaying the assembly code at the execution point, Dynamic C also allows code to be disassembled independently of the execution point. The **Disassemble at Cursor** option of the **INSPECT** menu (**CTRL-F10**) disassembles the machine code that represents the code at the screen cursor.

This feature allows the compiled code of statements to be previewed long before the code is executed. Another command on the **INSPECT** menu, **Disassemble at Address (ALT-F10)**, allows the machine code to be disassembled at any address (except at addresses in the BIOS area). This feature is especially handy to preview code generated for library functions.

Combined with the scrolling register, watch and stack windows, the disassembler features let the programmer trace the history of the code and know exactly which machine instruction caused what changes *after the fact*.

Break Points

At times, there may be a need to run a program at full speed and then stop at **break points**. These break points can be placed (and removed) at run time anywhere in the source code. The line of code is highlighted where a break point has been inserted in the source code.

There are **hard** break points and **soft** breakpoints. Interrupts are disabled at hard break points. Interrupts are restored to their former state when execution resumes after a hard break point. Soft break points do not affect the interrupt state. The interrupt flag may be toggled independently using the **Toggle interrupt** command on the **RUN** menu, or by using **EI** and **DI** in the watch window (see below). The message bar at the bottom of the Dynamic C window reports the current interrupt state. The **iff** in the watch window may also be used to determine the interrupt state.

Watch Expressions

Watch expressions allow the programmer to obtain the value of a variable, to evaluate an arbitrary expression, or to invoke a function out of sequence. To do this, select **Add/Delete Watch Expressions** from the **INSPECT** menu (or press **CTRL-W**). This invokes the watch expression dialog box, where an expression for evaluation is entered. If the cursor is placed over a variable name, or some text in the source file is highlighted, this text will appear in the dialog box when the dialog box is opened. The result of a watch expression will appear in the watch window after the dialog box closes.

A watch expression may be any valid C expression, including assignments, function calls, and preprocessor macros (do not type a semicolon after the expression). For example, the expression

```
MyVar = MyFunc(8)
```

would call the function **MyFunc** with the value **8** and assign the return value to the variable **MyVar** (assuming **MyVar** and **MyFunc** have been defined somewhere in the compiled program). A simpler watch expression might include only the name of a variable and return its value.

If a watch expression contains operations on **long** or **float** data types, the programmer must include a **dummy call** to the appropriate operator. For example, if division of long integers is desired in a watch expression, a dummy call in the program is required as follows.

```
long k, l; // variables for dummy '/' operation
k / l; // compile '/' for Watch Expressions
```

There are two basic ways to work with the watch dialog.

- 1 *Immediate Evaluation.* Enter an expression in the dialog box edit line and click **Evaluate**. The expression is evaluated only once, with the results displayed immediately in the watch window.
- 2 *Repeated Evaluation.* Enter an expression in the watch line and click **Add to top**. The expression will be added to the top of the watch list. (Watch list entries are deleted using the **Delete from top** button.) All the entries in the watch list are evaluated every time the program stops at a break point, after single-stepping, and after the stop command **<CTRL-Z>**. A watch window update can be forced using the **Update Watch Window** command **<CTRL-U>**.

The keyboard shortcut **<CTRL-W>** allows a variable to be evaluated very quickly. Just position the text cursor in the variable, type **<CTRL-W>** and hit **<ENTER>**.

The ability to evaluate expressions and function calls periodically and at will is a very powerful facility. Besides providing the ability to monitor the program state, this allows the program to be changed.

The watch dialog can be used to set the value of variables. Functions called via the watch dialog can be very effective (and possibly dangerous). For example, the PLCBus may be reset this way, or events can be simulated by changing the values at hardware inputs and outputs. A sophisticated programmer might even write functions meant only to be executed in the watch dialog for debugging purposes.

Returning to Edit Mode

After debugging, it is possible to continue editing the source code. Click on the **Edit mode** in the **EDIT** menu. The keyboard shortcut is **<F4>**.

Creating Stand-alone Programs

As mentioned previously under Compiling, EPROM files can be created using the **Compile to File** command. Generally, a program in a Z-World controller will run by itself, once the controller is disconnected from the PC running Dynamic C and is reset (for example, by turning power off and then on). Check to make sure that the controller is in run mode. The controller manual provides detailed instructions.

Controller with Program in EPROM

Once an EPROM has been burned, place it in the controller's EPROM socket. The program runs when the controller restarts.

Controller with Program in Flash Memory

Dynamic C places the program code in nonvolatile flash memory when compilation is to a flash-equipped controller. The program runs when the controller restarts in run mode.

Controller with Program in RAM

Dynamic C places the program in RAM when compilation is to a controller with EPROM. As long as the controller's RAM is powered, the program runs when the controller restarts in run mode.



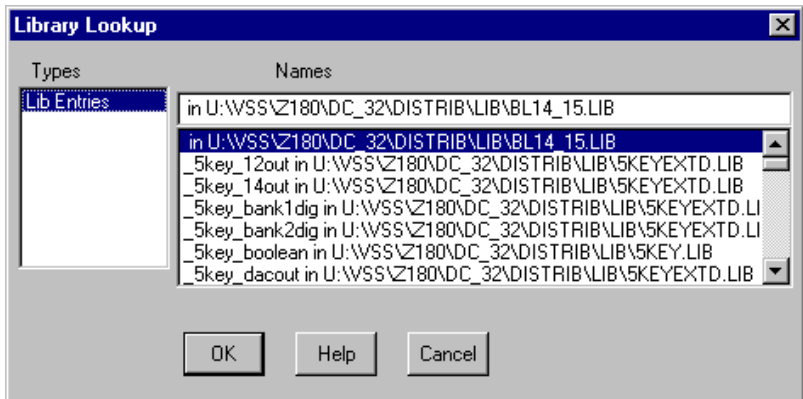
Controllers that have no backup battery will lose the contents of their RAM if they are disconnected from a power source.

Help

Dynamic C provides three forms of on-line help. The first is a standard Windows help system, containing descriptions of the available menus, keystrokes, dialog box options and other information about Dynamic C.

Function Lookup

The second form of on-line help provides information about the use of Dynamic C library functions. Most library functions have descriptive headers that are displayed when help regarding the function is requested. If a function name in a program is selected or clicked on, the help command (**CTRL-H** for short) will display the function header. If the function name is unknown to Dynamic C, a library lookup dialog will appear. Click **Lib Entries** to browse all the library functions known to Dynamic C.



Browsing has two benefits.

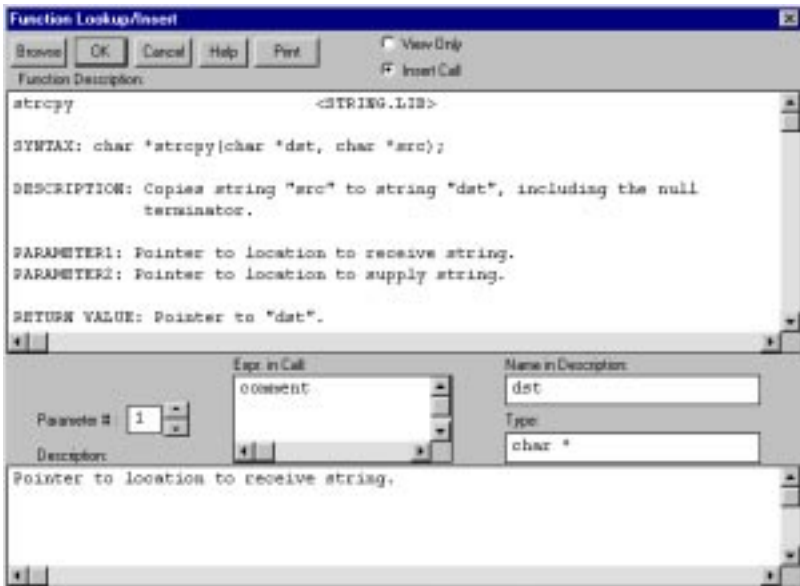
1. Review available functions.
2. Quick access to the function header, including its prototype. This provides a quick reminder how to call the function.

An additional benefit is that a function prototype can be copied from the help window and used to form a function call. This saves typing and time. (Remember that any text in a library help window can be copied and then pasted elsewhere.)

Function Assistance

The third form of help is a variant of the function “lookup.” Click on the **Insert Call** button in the lookup dialog box for the function assistant to place the function call in the program. The function assistant places a prototypical call in the program when the OK button is clicked.

However, every individual function parameter can be specified in the dialog. The function assistant reminds the programmer what types the parameters have and the order of the parameters. The function assistant in the following example shows that parameter 1 is named **dst**, a commonly used abbreviation for “destination,” and that it is a **char*** (pointer to **char**). The word “comment” in the example is the expression that replaces parameter 1 in the function call.





CHAPTER 4: ***DYNAMIC C ENVIRONMENT***

Dynamic C can be used to edit source files, compile programs, and run programs or choose options for these activities. There are two modes: *edit mode* and *run mode*. The run mode can be also called the *debug mode*. Compilation is, in effect, the transition between the edit mode and the run mode. Developers work with Dynamic C by editing text, issuing menu commands (or keyboard shortcuts for these commands), and viewing various debugging windows.

Programs can compile

- directly to a target controller,
- to a file for burning an EPROM
- to a file meant for downloading to a controller in which the Z-World Download Manager resides, or
- to a file meant for downloading to controller RAM.

In order to compile or run a program, a controller must be connected to the PC or a .RTI (Remote Target Information) file for compilation must exist.

Dynamic C includes editing options, compiler options, and memory options. Most of the options are in the **Options** menu.

Details about how to work with Windows have been omitted intentionally.



Refer to the *Microsoft Windows Users Guide* for details regarding the use of Windows. Dynamic C follows Windows software standards very closely.

Editing

Once a file has been created or has been opened for editing, the file is displayed in a text window. It is possible to open or create more than one file and one file can have several windows. Dynamic C supports normal Windows text editing operations.

Use the mouse (or other pointing device) to position the text cursor, to select text, or to extend a text selection. Scroll bars may be used to position text in a window. Dynamic C will, however, work perfectly well without a mouse, although it may be a bit tedious.

It is also possible to scroll up or down through the text using the arrow keys or the PageUp and PageDown keys or the Home and End keys. The left and right arrow keys allow scrolling left and right.

Arrows Use the up, down, left and right arrow keys to move the cursor in the corresponding direction.

The CTRL key works in conjunction with the arrow keys this way.

CTRL-Left	Move to previous word
CTRL-Right	Move to next word
CTRL-Up	Scroll up one line (text moves down)
CTRL-Down	Scroll down one line

Home Moves the cursor *backward* in the text.

Home	Move to beginning of line
CTRL-Home	Move to beginning of file
SHIFT-Home	Select to beginning of line
SHIFT-CTRL-Home	Select to beginning of file

End Moves the cursor *forward* in the text.

End	Move to end of line
CTRL-End	Move to end of file
SHIFT-End	Select to end of line
SHIFT-CTRL-End	Select to end of file

Sections of the program text can be “cut and pasted” (add and delete) or new text may be typed in directly. New text is inserted at the present cursor position or replaces the current text selection.

The **Replace** command in the **Edit** menu is used to perform search and replace operations either forwards or backwards.

Menus

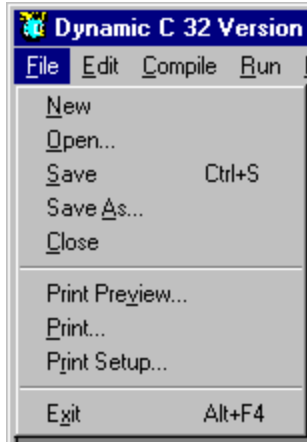
Dynamic C has eight command menus,

File Edit Compile Run Inspect Options Window Help

as well as the standard Windows system menus. An available command can be executed from a menu by clicking the menu and then clicking the command, or by (1) pressing the **ALT** key to activate the menu bar, (2) using the left and right arrow keys to select a menu, (3) and using the up or down arrow keys to select a command, and (4) pressing **ENTER**. It is usually more convenient to type keyboard shortcuts (such as <**CTRL-H**> for **Help**) once they are known. Pressing the **ESC** key will make any visible menu disappear. A menu can be activated by holding the **ALT** key down while pressing the underlined letter of the menu name (use the space bar and minus key to access the system menus). For example, type <**ALT-F**> to activate the **File** menu.

File Menu

Click the menu title or press **ALT-F** to select the **File** menu. The **File** menu commands and their functions are described below.

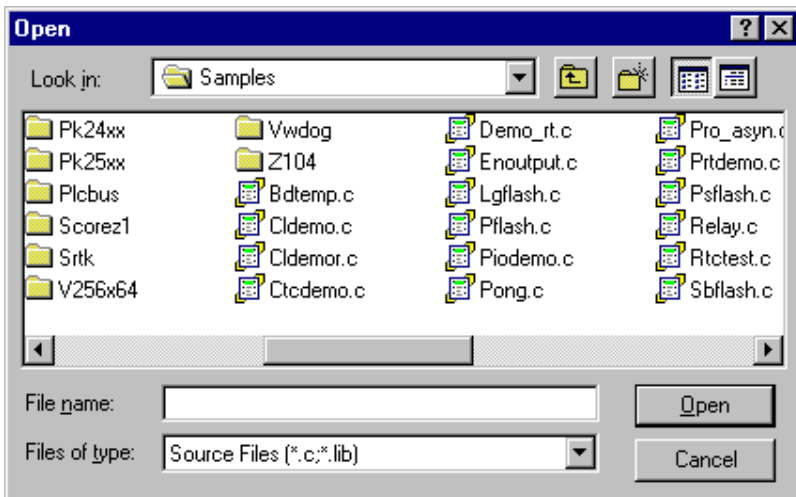


New

Creates a new, blank, untitled program in a new window.

Open

Presents a dialog in which to specify the name of a file to open. Unless there is a problem, Dynamic C will present the contents of the file in a text window. The program can then be edited or compiled.



To select a file, type in the desired file name, or select one from the list. The file's directory may also be specified.



Refer to the *Microsoft Windows User Guide* for more information.

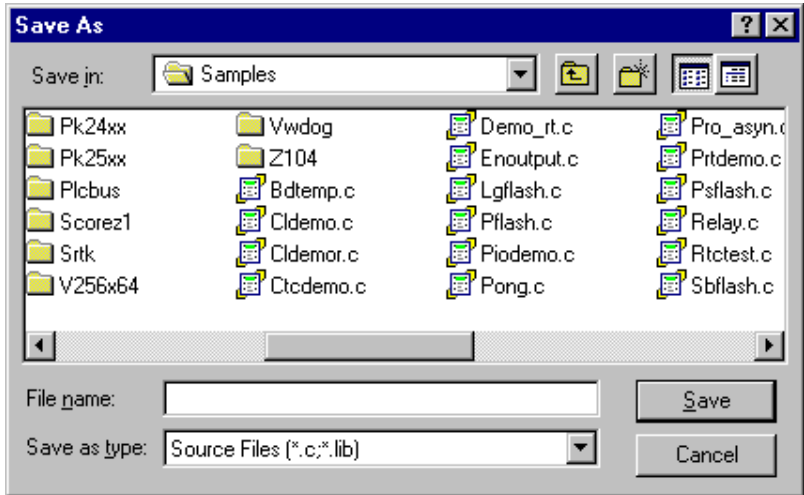
Save

The **Save** command updates an open file to reflect the latest changes. If the file has not been saved before (the file is new and untitled), the **Save As** dialog will appear.

Use the **Save** command often while editing to protect against loss during power failures or system crashes. The keyboard shortcut is **CTRL-S**.

Save As

Allows a new name to be entered for a file and then saves the file under the new name.



Close

Closes the active window. The active window may also be closed by pressing **CTRL-F4** or by double-clicking on its system menu. If there are unsaved changes a dialog prompting to save or discard the changes will be presented.

The file is saved when **Yes** is clicked or “y” is typed. If the file is untitled, there will be a prompt for a file name in the **Save As** dialog. Any changes to the document will be discarded if **No** is clicked or “n” is typed. **Cancel** results in a return to Dynamic C, with no action taken.

Print Preview...

Shows approximately what printed text will look like. Dynamic C switches to print preview mode when this command is selected, and allows the programmer to navigate through images of the printed pages.

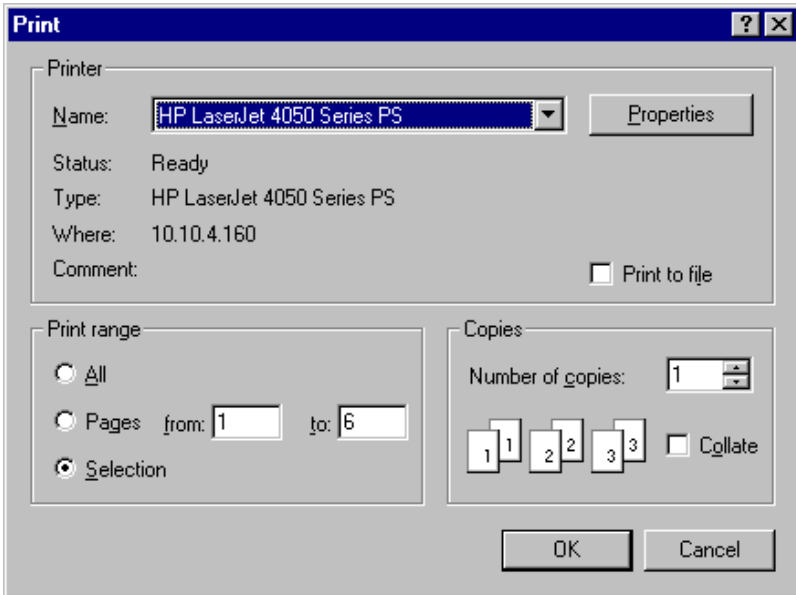
The following tool bar appears at the top of the print preview window.



From left to right, the icons on the print preview toolbar perform the following functions: select the previous or next page; select one or two pages displayed at a time; show the displayed page number; print (same as the **File>Print...** menu command); Close (exit print preview mode).

Print...

Text can be printed from any Dynamic C window. There is no restriction to printing only source code. For example, the contents of the assembly window or the watch window can be printed. Dynamic C displays the following type of dialog when the **Print** command is selected.



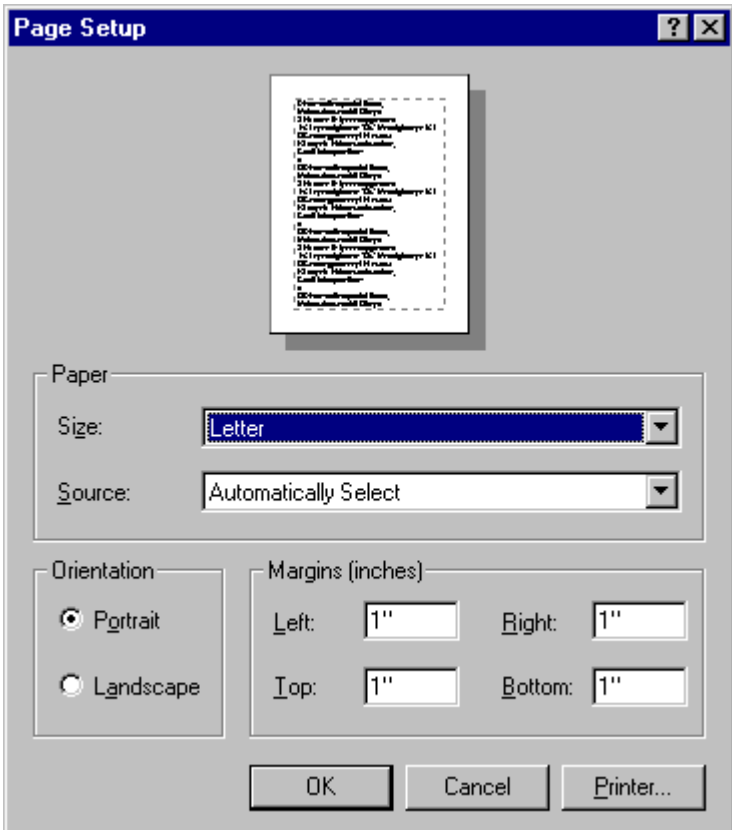
If the **Print to File** option is selected, Dynamic C creates a file (it will ask for a path list) in a format suitable to send to the specified printer (I.E.: if the selected printer is a PostScript printer, the file will contain PostScript).

To choose a printer, click the printer name in the drop-down list box and then click on one of the names in the displayed list. Click the **Properties** button to adjust or inspect options available on the selected printer.

The **Print range** can be **All**, specific **Pages**, or even just a **Selection** of text if a block of text is currently selected in the active window. As many copies of the text as needed may be printed. If more than one copy is requested, the pages may be collated or uncollated.

Print Setup...

Allows the printer's page set up to be specified in the following type of dialog box.



Depending on the printer selected, it may be possible to specify paper size and paper orientation (portrait, or tall, vs. landscape, or wide). Most printers have these options. A specific printer may or may not have more than one paper source. The page's margins are also set up here.

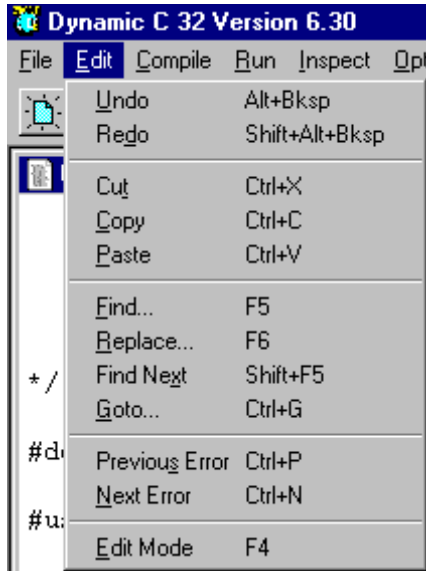
The **Printer** button allows you to specify the printer to be used and to set its **Properties**, as has been previously covered in the **File>Print...** description.

Exit

To exit Dynamic C. When this is done, Windows will either return to the Windows Program Manager or to another application. The keyboard shortcut is **<ALT-F4>**.

Edit Menu

Click the menu title or press **<ALT-E>** to select the **EDIT** menu. The **Edit** menu commands and their functions are described below.



Undo

Undoes recent changes in the active edit window. This command may be repeated several times to undo multiple changes. The amount of editing that may be undone will vary with the type of operations performed, but should suffice for a few large cut and paste operations or many lines of typing. Dynamic C discards all undo information for an edit window when the file is saved. The keyboard shortcut is **<ALT-BACKSPACE>**.

Redo

Redoes modifications recently undone. This command only works immediately after one or more **Undo** operations. The keyboard shortcut is **<ALT-SHIFT-BACKSPACE>**.

Cut

Removes selected text from the active window. A copy of the text is saved on the "clipboard." The contents of the clipboard may be pasted virtually anywhere, repeatedly, in the same or other source files, or even in word-processing or graphics program documents. The keyboard shortcut is **<CTRL-X>**.

Copy

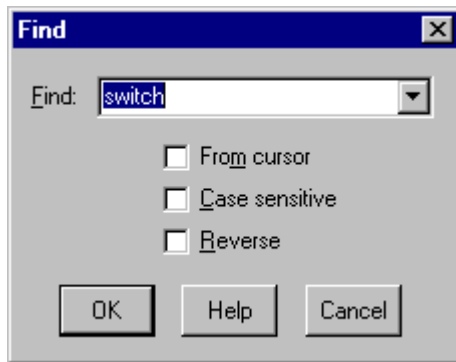
Makes a copy of selected text in the active window. The copy of the text is saved on the “clipboard.” The contents of the clipboard may be pasted virtually anywhere. The keyboard shortcut is **⟨CTRL-C⟩**.

Paste

Pastes text on the clipboard as a result of a copy or cut (in Dynamic C or some other Windows application). The paste command places the text at the current insertion point. Note that nothing can be pasted in a debugging window. It is possible to paste the same text repeatedly until something else is copied or cut. The keyboard shortcut is **⟨CTRL-V⟩**.

Find...

Finds specified text. The following dialog box appears in response to the **Find** command.



Type the text to be found in the **Find** drop-down text box, or click on the down-arrow icon to the right of the box and a drop-down list of previous **Find** text is displayed. If you click on a previous **Find** text it will become selected and ready for editing in the **Find** box. In this example, the **Find** command (and the **Find Next** command, too) will find occurrences of the word “switch.”

Use the **From cursor** checkbox to choose whether to search the entire file or to begin at the cursor location. If **Case sensitive** is selected, the search will only find occurrences that match exactly. Otherwise, the search will find matches having either uppercase or lowercase letters. For example, “switch,” “Switch” and “SWITCH” would all match. If **Reverse** is selected, the search will occur in reverse, that is, the search will proceed toward the beginning of the file, rather than toward the end of the file.

The keyboard shortcut for **Find** is **⟨F5⟩**.

Replace...

Replaces specified text. The following dialog box appears in response to the **Replace** command.



Type the text to be found in the **Find** drop-down text box, or click on the down-arrow icon to the right of the box and a drop-down list of previous **Find** text is displayed. If you click on a previous **Find** text it will become selected and ready for editing in the **Find** box.

Then, type the text to substitute in the **Change to** drop-down text box, or select and/or edit a previous **Change to** text from the drop-down list. In this example, the **Replace** command will find an occurrence of the word “reg7” and replace it with “reg9.”

Use the **From cursor** checkbox to choose whether to search the entire file or to begin at the cursor location. If **Case sensitive** is selected, the search will only find occurrences that match exactly. Otherwise, the search will find matches having either uppercase or lowercase letters. For example, “reg7,” “Reg7” and “REG7” would all match.

The **Selection only** checkbox allows the substitution to be performed only within the currently selected text. This box is disabled if no text is selected. When used in conjunction with the **Change All** button it limits text replacements to within a selected block of text.

If **Reverse** is selected, the search will occur in reverse, that is, the search will proceed toward the beginning of the file, rather than toward the end of the file. If **No prompt** is selected then the text replacement(s) will be made without prompting. Otherwise, a prompt dialog asks whether or not to make each change. This is an important safeguard, particularly if the **Change All** button is clicked.

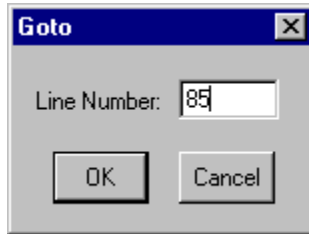
The keyboard shortcut for **Replace** is **⌘F6**.

Find Next

Once search text has been specified with the **Find** or **Replace** commands, the **Find Next** command (**⌘SHIFT-F5** for short) will find the next occurrence of the same text, searching forward or in reverse, case sensitive or not, as specified with the previous **Find** or **Replace** command. If the previous command was **Replace**, the operation will be a text replacement.

Goto...

Positions the insertion point at the start of the specified line. The following dialog is displayed when the **Goto** command is issued.



Type the line number (or approximate line number) to which to jump. That line, and lines in the vicinity, will be displayed in the source window.

Previous Error

Locates the previous compilation error or warning in the source code. Any errors or warnings will be displayed in a list in the message window after a program is compiled. Dynamic C selects the previous error or warning in the list and positions the offending line of code in the text window when the **Previous Error** command (**⌘CTRL-P** for short) is made. Use the keyboard shortcuts to locate errors or warnings quickly.

Next Error

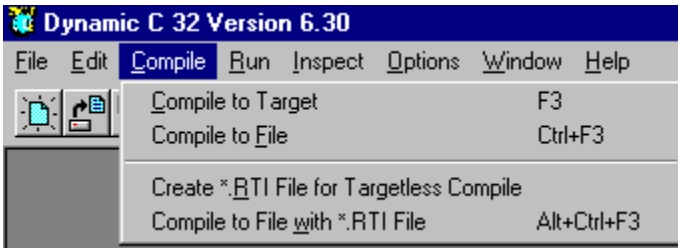
Locates the next compilation error or warning in the source code. Any errors or warnings will be displayed in a list in the message window after a program is compiled. Dynamic C selects the next error or warning in the list and positions the offending line of code in the source window when the **Next Error** command (**⌘CTRL-N** for short) is made. Use the keyboard shortcuts to locate errors or warnings quickly.

Edit Mode

Switches Dynamic C back to edit mode from run mode (also called debug mode). After a program has been compiled or executed, Dynamic C will not allow any modification to the program unless the **Edit Mode** is selected. The keyboard shortcut is **⌘F4**.

Compile Menu

Click the menu title or press <ALT-C> to select the **Compile** menu.



There are three ways to compile.

1. Directly to a target controller, connected via COM port to Dynamic C.
2. To a file, with a controller connected via COM port to Dynamic C.
3. To a file, with no controller connected. This requires a Remote Target Information (*.RTI) file for the intended controller.

Use **Compile to File** to generate a .BPF, .BIN, .DLP or NULL program file, as set by the compiler options selected in the **OPTIONS** menu. Table 4-1 summarizes the file types.

Table 4-1. Dynamic C 'Compile to File' Types

Application Only (.BPF)	The Compile to File or Compile to File with *.RTI File command generates a .BPF file. This option is included for backward compatibility only and is used for downloading programs via the Program Loader Utility to RAM.
BIOS+Application BIOS+App+Library BIOS+App+Lib+SimEE BIOS+App+Lib+ClrSEE (.BIN)	The Compile to File or Compile to File with *.RTI File command generates a .BIN (EPROM) file. If the Create HEX File Also compiler option is selected, the command also generates an Intel hex format (.HEX) file.
Download via DLM (.DLP)	The Compile to File or Compile to File with *.RTI File command generates a downloadable program file (.DLP) to be used by the Z-World Down Load Manager (DLM).
Null Device (Bit Bucket)	The Compile to File or Compile to File with *.RTI File command generates no output. The fast compilation is useful to (1) perform syntax checking, (2) perform type checking, or (3) get the sizes of each code and data segment.

The **Memory Options** command (in the **Options** menu) affects the placement and allocation of code and data in the target controller's memory. The **Serial Options** command (in the **Options** menu) specifies the speed and mode when the generated code is uploaded from the PC to the target.



For more details, refer to the **OPTIONS** menu discussion later in this chapter.

The **Compile** menu commands and their functions are described here.

Compile to Target

Compiles program and loads it in target controller's memory. Dynamic C automatically determines whether the target has on-target RAM, flash EPROM or development-board RAM, and compiles with the appropriate memory map. The controller's reference manual describes which platform is available for the target being used. Any compilation errors are listed in the message window that is activated automatically. Otherwise, the program is ready to run and Dynamic C is in run (or debug) mode. The program will start running without a pause if **#nodebug** precedes the main function. (Dynamic C will also lose contact with the target.) The keyboard shortcut is **⌘F3**.

Compile to File

Compiles program to a file. A target controller must be connected because Dynamic C takes configuration information from the target. Any compilation errors are listed in the message window that is activated. Otherwise, Dynamic C generates a file according to the compiler options that have been selected. The keyboard shortcut is **⌘CTRL-F3**.

Create *.RTI File for Targetless Compile

It is possible to compile without a target controller present if a Remote Target Information (***.RTI**) file for the intended controller is created. The **Compile to File with *.RTI File** command may be used once that has been done.

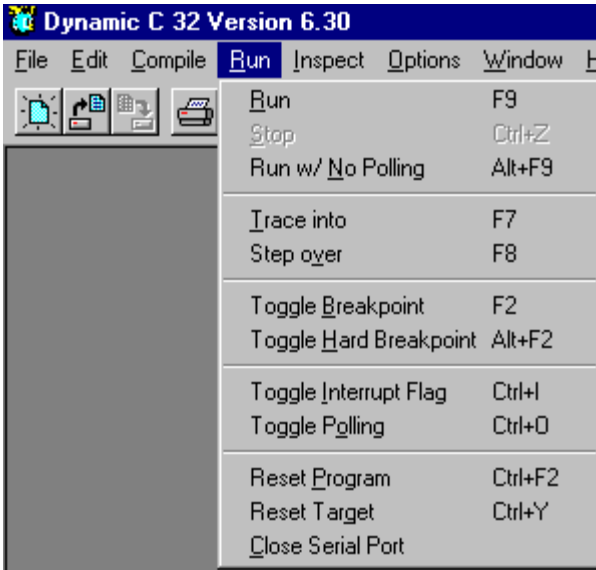
A target controller is still needed to create the ***.RTI** file. The intended target must be identical to the controller used to create the ***.RTI** file.

Compile to File with *.RTI File

Compiles program to a file using ***.RTI** file created. Any compilation errors are listed in the message window that is activated. Otherwise, Dynamic C generates a file according to the compiler options that have been selected. The keyboard shortcut is **⌘ALT-CTRL-F3**.

Run Menu

Click the menu title or press **ALT-R** to select the **Run** menu. The **Run** menu commands and their functions are described here.



Run

Starts program execution from the current breakpoint. Registers are restored, including interrupt status, before execution begins. The keyboard shortcut is **ALT-F9**.

Stop

The **Stop** command places a hard break point at the point of current program execution. Usually, the compiler cannot stop within ROM code or in **nodebug** code. On the other hand, the target can be stopped at the **rst 028h** instruction if **rst 028h** assembly code is inserted as in-line assembly code in **nodebug** code. However, the debugger will never be able to find and place the execution cursor in **nodebug** code. The keyboard shortcut for this command is **CTRL-Z**.

Run w/ No Polling

This command is identical to the **Run** command, with an important exception. When running in polling mode (**F9**), the development PC polls or interrupts the target system every 100 milliseconds to obtain or send information about target break points, watch lines, keyboard-entered target input, and target output from **printf** statements.

Polling creates interrupt overhead in the target, which can be undesirable in programs with tight loops. The **Run w/ No Polling** command allows the program to run without polling and its overhead. (Any `printf` calls in the program will cause execution to pause until polling is resumed. Running without polling also prevents debugging until polling is resumed.) The keyboard shortcut for this command is **⟨ALT-F9⟩**.

Trace Into

Executes one C statement (or one assembly language instruction if the assembly window is displayed) with descent into functions. Execution will not descend into functions stored in ROM because Dynamic C cannot insert the required break points in the machine code. If `nodebug` is in effect, execution continues until code compiled without the `nodebug` keyword is encountered. The keyboard shortcut is **⟨F7⟩**.

Step over

Executes one C statement (or one assembly language instruction if the assembly window is displayed) without descending into functions. The keyboard shortcut is **⟨F8⟩**.

Toggle Breakpoint

Toggles a regular (“soft”) break point at the location of the execution cursor. Soft break points do not affect the interrupt state at the time the break point is encountered, whereas hard break points do. The keyboard shortcut is **⟨F2⟩**.

Toggle Hard Breakpoint

Toggles a hard break point at the location of the execution cursor. A hard break point differs from a soft breakpoint in that interrupts are disabled when the hard break point is reached. The keyboard shortcut is **⟨ALT-F2⟩**.

Toggle Interrupt Flag

Toggles interrupt state. The keyboard shortcut is **⟨CTRL-I⟩**.

Toggle Polling

Toggles polling mode. When running in polling mode (**F9**), the development PC polls or interrupts the target system every 100 milliseconds to obtain or send information regarding target break points, watch lines, keyboard-entered target input, and target output from `printf` statements. Polling creates interrupt overhead in the target, which can be undesirable in programs with tight loops.

This command is useful to switch modes while a program is running. The keyboard shortcut is **⟨CTRL-O⟩**.

Reset Program

Resets program to its initial state. The execution cursor is positioned at the start of the main function, prior to any global initialization and variable initialization. (Memory locations not covered by normal program initialization may not be reset.) The keyboard shortcut for this command is **⟨CTRL-F2⟩**.



The initial state includes only the execution point (program counter), memory map registers, and the stack pointer. The **Reset Program** command will not reload the program if the previous execution overwrites the code segment.

Reset Target

Tells the target system to perform a software reset including system initialization. Resetting a target *always* brings Dynamic C back to edit mode. The keyboard shortcut is **⟨CTRL-Y⟩**.

Close Serial Port

Closes the serial port currently in use by Dynamic C.

Inspect Menu

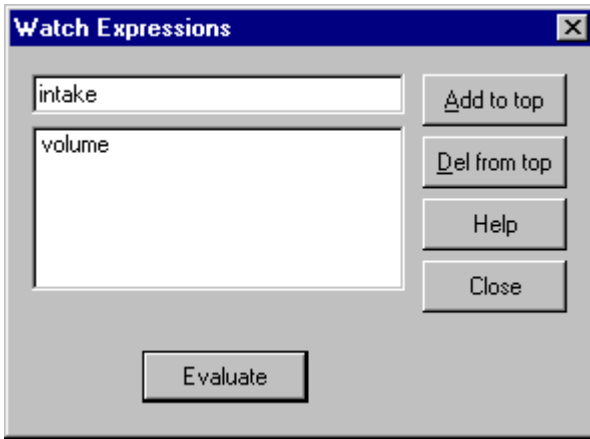
Click the menu title or press **⟨ALT-I⟩** to select the **INSPECT** menu.



The **Inspect** menu provides commands to manipulate watch expressions, view disassembled code, and produce hexadecimal memory dumps. The **Inspect** menu commands and their functions are described here.

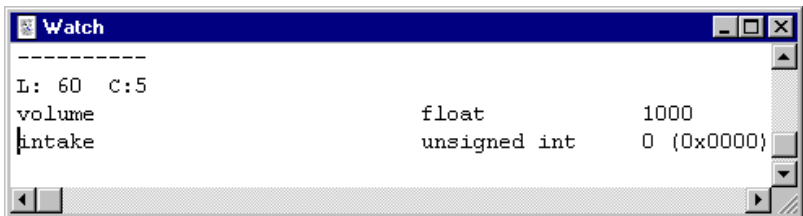
Add/Del Watch Expression

This command provokes Dynamic C to display the following dialog.



This dialog works in conjunction with the watch window. The text box at the top is the current expression. An expression may have been typed here or it was selected in the source code. This expression may be evaluated immediately by clicking the **Evaluate** button or it can be added to the expression list by clicking the **Add to top** button. Expressions in this list are evaluated, and the results are displayed in the watch window, every time the watch window is updated. Items are deleted from the expression list by clicking the **Del from top** button.

An example of the results displayed in the watch window appears below.



The keyboard shortcut is **CTRL-W**.



Refer also to *Watch Expressions* in the *Debugging* section in Chapter 3, Using Dynamic C.

Clear Watch Window

Removes entries from the watch dialog and removes report text from the watch window. There is no keyboard shortcut.

Update Watch Window

Forces expressions in the watch expression list to be evaluated and displayed in the watch window. Normally the watch window is updated every time the execution cursor is changed, that is when a single step, a break point, or a stop occurs in the program. The keyboard shortcut is **<CTRL-U>**.

Disassemble at Cursor

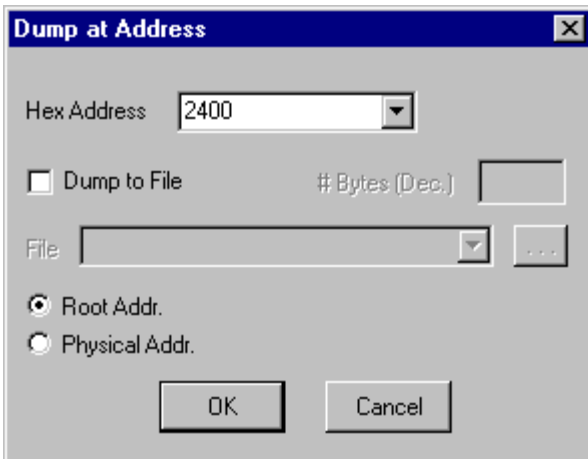
Loads, disassembles and displays the code at the current editor cursor. This command only works in user application code (not the libraries) that is not declared **nodebug**. This command does not stop the execution on the target either. The keyboard shortcut is **<CTRL-F10>**.

Disassemble at Address

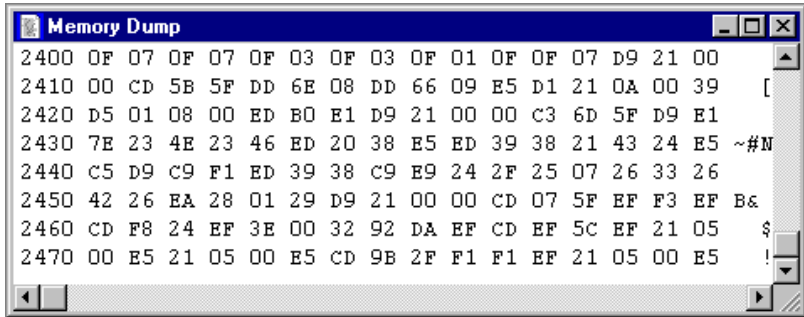
Loads, disassembles and displays the code at the specified address. This command produces a dialog box that asks for the address at which disassembling should begin. Addresses may be entered in two formats: a 4-digit hexadecimal number that specifies any location in the root space (the valid range is **2000h** to **FFFFh**), or a 2-digit **CBR** page number followed by a colon followed by a 4-digit logical address (the page number ranges from **00h** to **FFh**, while the valid range for the logical address is from **E000h** to **FFFFh**). Note that the disassembler rejects any attempt to disassemble code between address **00000h** and **02000h** in the physical memory, regardless of how the address is expressed in logical address. The keyboard shortcut is **<ALT-F10>**.

Dump at Address

Allows blocks of raw values in any memory location (except the BIOS, at **00000h** through **01FFFh**) to be looked at.



Values can either be displayed on the screen or written to a file if “Dump to File” is checked. A typical screen display appears below.



The dump window can be scrolled. Scrolling causes the contents of other memory addresses to appear in the window. The window always displays 128 bytes and their ASCII equivalent. Values in the dump window are updated only when Dynamic C stops, or comes to a break point.

If “Dump to File” is checked, specify the number of bytes and the pathname of the file. The file output closely resembles the memory dump in the window above.

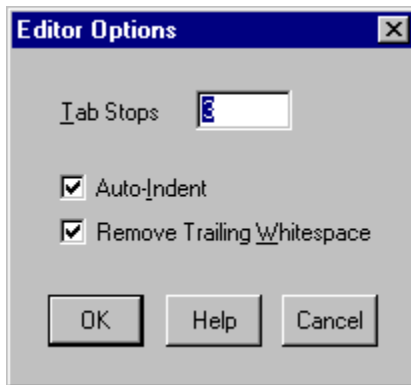
Options Menu

Click the menu title or press <ALT-O> to select the **OPTIONS** menu. The **Options** menu commands and their functions are described here.



Editor

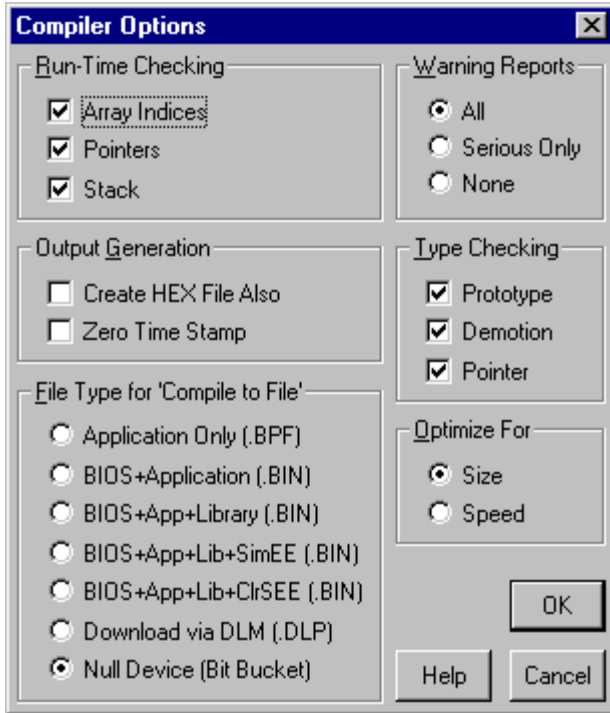
The **Editor** command gets Dynamic C to display the following dialog.



Use this dialog box to change the behavior of the Dynamic C editor. By default, **Tab Stops** are set every three characters, but may be set to any value greater than zero. **Auto-Indent** causes the editor to indent new lines to match the indentation of previous lines. **Remove Trailing Whitespace** causes the editor to remove extra space or tab characters from the end of a line.

Compiler

The **Compiler** command gets Dynamic C to display the following dialog, which allows compiler operations to be changed.



The **Run-Time Checking** options group control the generation of code for checking the application's run-time operation. When selected, the code generated by each of these options will cause a run-time error if a problem is detected. These options increase the amount of code and cause slower execution, but they can be valuable debugging tools. These options are described in Table 4-2.

Table 4-2. Run-Time Checking Options

Array Indices	Checks array bounds. This feature adds code for every array reference.
Pointers	Check for invalid pointer assignments. A pointer assignment is invalid if the code attempts to write to a location marked not writeable. Locations marked not writeable include the entire root code segment. This feature adds code for every pointer reference.
Stack	Check for stack corruption. Stack errors are reported on return from the function.

The **Output Generation** options group control miscellaneous aspects of the compiler's code and file output generation.

Create HEX File Also

When this option is selected, the **Compile** menu's **Compile to File** and **Compile to File with *.RTI File** commands also create an Intel hex format file (.HEX) in addition to the .BPF or .BIN file.

Zero Time Stamp

When this option is selected, the compile time-stamp and performance information which Dynamic C 32 normally embeds in the application code is forced to zero. Thus a given version of Dynamic C 32 can always compile a fixed set of application and library code to exactly the same output file, which is useful for code certification. It is important to note that when this option is selected, an application must not depend on the uniqueness of its **compile time-stamp** to determine when it has been updated with a newer version.

The **File Type for 'Compile to File'** option group specifies the file type when **Compile to File** or **Compile to File with *.RTI File** commands are issued. The file types appear in Table 4-3.

Table 4-3. Dynamic C 'Compile to File' Types

Application Only (.BPF)	The Compile to File or Compile to File with *.RTI File command generates a .BPF file. This option is included for backward compatibility only and is used for downloading programs via the Program Loader Utility to RAM.
BIOS+Application BIOS+App+Library BIOS+App+Lib+SimEE BIOS+App+Lib+ClrSEE (.BIN)	The Compile to File or Compile to File with *.RTI File command generates a .BIN (EPROM) file. If the Create HEX File Also compiler option is selected, the command also generates an Intel hex format (.HEX) file.
Download via DLM (.DLP)	The Compile to File or Compile to File with *.RTI File command generates a downloadable program file (.DLP) to be used by the Z-World Down Load Manager (DLM).
Null Device (Bit Bucket)	The Compile to File or Compile to File with *.RTI File command generates no output. The fast compilation is useful to (1) perform syntax checking, (2) perform type checking, or (3) get the sizes of each code and data segment.

The **Warning Reports** option group tells the compiler whether to report all warnings, no warnings or serious warnings only. It is advisable to let the compiler report all warnings because each is a potential run-time bug.



Demotions (such as converting a **long** to an **int**) are considered non-serious with regard to warning reports.

The **Type Checking** options group tells Dynamic C to perform the appropriate type checking as described in Table 4-4.

Table 4-4. Type Checking Options

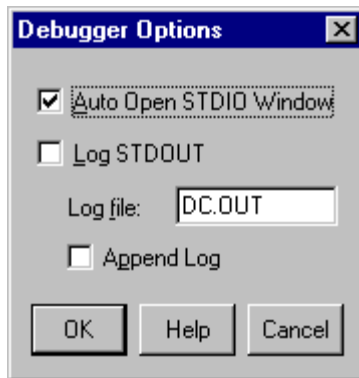
Prototypes	Performs strict type checking of arguments of function calls against the function prototype. The number of arguments passed must match the number of parameters in the prototype. In addition, the types of the arguments must match those defined in the prototype. Z-World recommends prototype checking because it identifies likely run-time problems. To fully use this feature, all functions should have prototypes (including functions implemented in assembly).
Demotion	Detects demotion. A demotion automatically converts the value of a larger or more complex type to the value of a smaller or less complex type. The increasing order of complexity of Dynamic C's scalar types is: <code>char</code> <code>unsigned int</code> <code>int</code> <code>unsigned long</code> <code>long</code> <code>float</code> A demotion deserves a warning because information may be lost in the conversion. For example, when a long variable whose value is 0x10000 is converted to an int value, the resulting value is 0. The high-order 16 bits are lost. An explicit type casting can eliminate demotion warnings. All demotion warning reports are considered non-serious.
Pointer	Generates warnings if pointers to different types are intermixed without type casting. While type casting has no effect in straightforward pointer assignments of different types, type casting does affect pointer arithmetic and pointer dereferences. All pointer warning reports are considered non-serious.

The **Optimize For** option group optimizes the program for *size* or for *speed*. When the compiler knows more than one sequence of instructions that perform the same action, it selects either the smallest or the fastest sequence, depending on the programmer's choice for optimization.

The difference made by this option is less obvious in the user application (in which most code is not marked **nodebug**). The speed gain by optimizing for speed is most obvious for functions that are marked **nodebug** and have no **auto** local (stack-based) variables.

Debugger

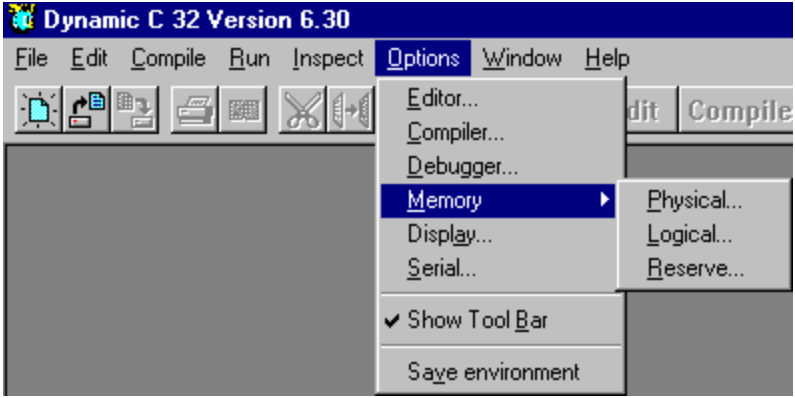
The **Debugger** command gets Dynamic C to display the following dialog.



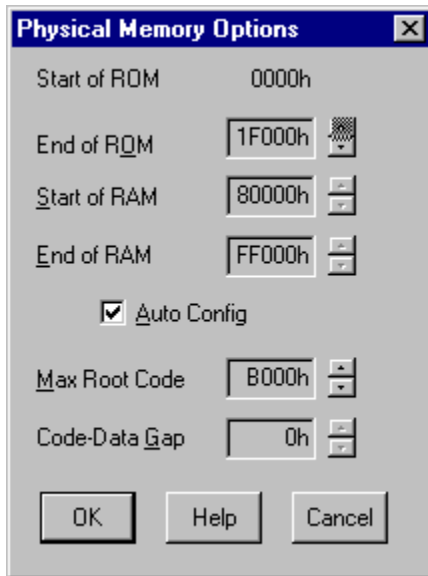
The options on this dialog box may be helpful when debugging programs. In particular, they allow **printf** statements and other **STDIO** output to be logged to a file. Check the box labeled **Log STDOUT** to send a copy of all standard output to the **Log File**. The name of the log file can be specified, and the **Append Log** checkbox selects whether to append or overwrite if the log file already exists. Normally, Dynamic C automatically opens the **STDIO** window when a program first attempts to print to it. This can be changed with the checkbox labeled **Auto Open STDIO Window**.

Memory

The **Memory** command gets Dynamic C to display a submenu. Click one of the three submenu items to specify memory settings.



The **Physical...** memory options submenu produces the following dialog.

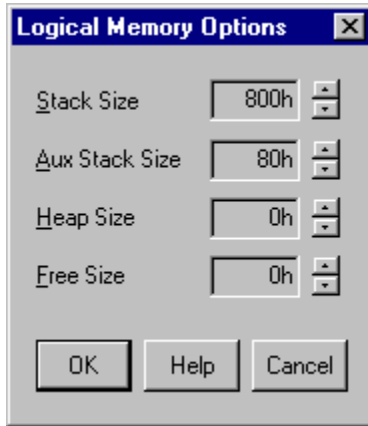


The size and boundaries of RAM and ROM can be specified according to the information in Table 4-5.

Table 4-5. Physical Memory Options

Start of ROM	ROM (EPROM or Flash) always starts at address 0000H.
End of ROM	This option is only used to build an application EPROM or to compile to Flash. Since the ROM always starts at physical address 0000H, this option also specifies the size of the EPROM to be built. For example, if 1000H (64K) is specified as the end of ROM, an EPROM that has 64K bytes is needed for the application.
Start of RAM	RAM starts at 4000H (256K), 8000H (512K), or A000H (640K). Normally, this option is set by the Auto Config feature. However, if code that is meant for download to RAM is compiled with multiple programs resident, this option can be changed so different programs occupy different segments of RAM.
End of RAM	The physical address where RAM ends depends on the RAM chip. The difference between end of RAM and start of RAM is equal to the size of the RAM chip.
Max Root Code	This is the anticipated maximum size of root code. This parameter is meaningful only when building an application EPROM or compiling to Flash. The size of root code in the actual program can be less than or equal to this amount. The maximum root code size cannot exceed 44K (B000H). To get optimum memory allocation, compile the program to the Null Device and then use the information in the information window to decide this parameter.
Code-Data Gap	This option allows the compiler to load programs in RAM using a 32K RAM on the target. It is meaningful only when compiling directly to target RAM. Set this option to 8000H only if the target RAM size is 32K, otherwise set it to 0000H.
Auto Config	This box, when checked, makes Dynamic C determine the start and end of RAM automatically. The code-data gap will also be adjusted automatically. Auto Config should always be checked for most programmers. This allows the physical memory options to be set automatically when Dynamic C connects to the target controller. Programmers who wish to create a program that resides in a different part of memory can turn this option off.

The **Logical...** memory options submenu provides the following dialog.



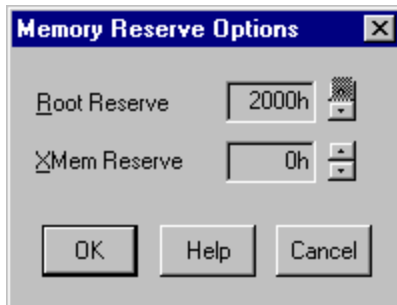
The **Stack Size** option specifies the number of bytes (in hex) allocated for the run-time stack. If the function calls nest deeply or if there are large amounts of **auto** local function data, use this option to increase the memory allocated.

The **Aux Stack Size** option specifies the number of bytes (in hex) allocated to an alternate stack used mainly for stack verification bookkeeping.

The **Heap Size** option specifies the number of bytes (in hex) in the heap (used for dynamic memory allocation functions such as **malloc**). Heap space must be allocated before using dynamic memory allocation.

The **Free Size** option specifies the number of bytes (in hex) that are not allocated for other purposes such as the heap. This space is completely under the program's control and is accessed entirely by pointers. Use the **Information Window** (under the **WINDOW** menu) to find out where this memory is allocated.

The **Reserve...** memory options submenu provides the following dialog.

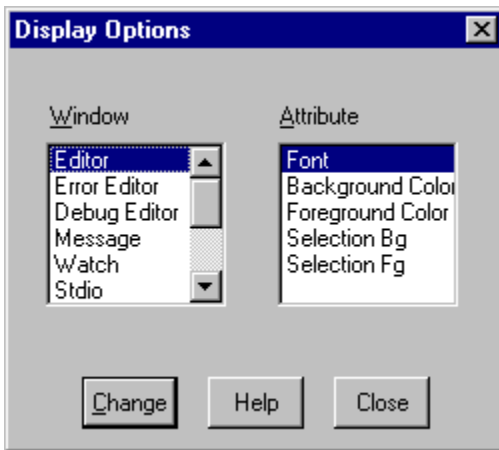


Root Reserve and **XMem Reserve** specify how the compiler allocates memory. When compiling code whose destination is not specified (that is, *anymem* code), Dynamic C first compiles all code to root until the amount of root memory left is less than the size of the **Root Reserve**. Then the compiler places all *anymem* code in extended memory until the amount of extended memory left is less than the **XMem Reserve**. The compiler then returns to the root until memory is exhausted. Functions specifically placed in root memory or in XMEM are always compiled in the area specified. The reserves guarantee a minimum of space in both the root and extended memory for functions that must go in one of those areas.

Leave enough space in the **Root Reserve** for all library functions invoked in the program.

Display

The **Display** command gets Dynamic C to display the following dialog.

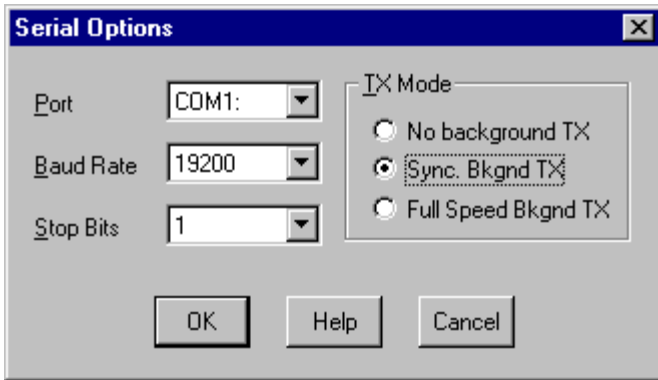


Use the **Display** options dialog box to change the appearance of Dynamic C windows. First choose the window from the window list. Then select an attribute from the attribute list and click the change button. Another dialog box will appear to make the changes. Note that Dynamic C allows only fixed-pitch fonts and solid colors (if a dithered color is selected, Dynamic C will use the closest solid color).

The Editor window attributes affect all text windows, except two special cases. After an attempt is made to compile a program, Dynamic C will either display a list of errors in the message window (compilation failed), or Dynamic C will switch to run mode (compilation succeeded). In the case of a failed compile, the editor will take on the **Error Editor** attributes. In the case of a successful compile, the editor will take on the **Debug Editor** attributes.

Serial

The **Serial** command gets Dynamic C to display the following dialog.



Use this dialog to tell Dynamic C how to communicate with the target controller. The COM port, baud rate, and number of stop bits may be selected. The transmission mode radio buttons also affect communication by controlling the overlap of compilation and downloading.

In the **No Background TX** mode, Dynamic C will not overlap compilation and downloading. This is the most reliable mode, but also the slowest—the total compile time is the sum of the processing time and the communication time.

In the **Full Speed Bkgnd TX** mode, Dynamic C will almost entirely overlap compilation and downloading. This mode is the fastest, but may result in communication failure.

The **Sync. Bkgnd TX** mode provides partial overlap of compilation and downloading. This is the default mode used by Dynamic C.

Show Tool Bar













The **Show Tool Bar** command toggles the tool bar on or off:



Dynamic C remembers the toolbar setting on exit.

Table 4-6 explains what the toolbar buttons mean.

Table 4-6. Dynamic C Toolbar

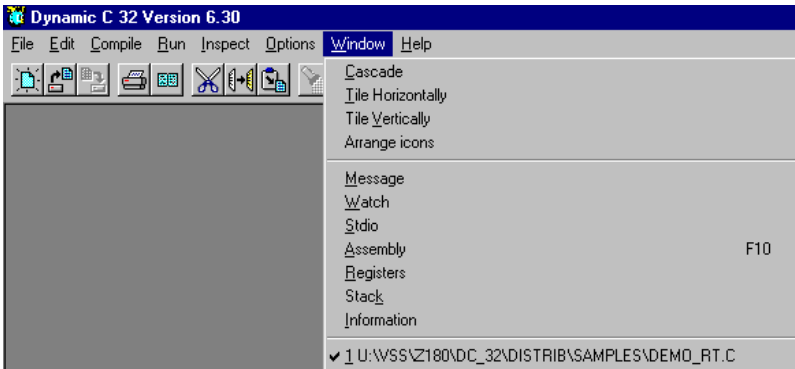
 New file	 Find
 Open file	 Replace
 Save file	 Repeat "Find" or "Replace"
 Print	Edit Switch to Edit mode
 Print preview	Compile Compile to RAM
 Cut (delete)	Assemb Toggle assembly window
 Copy	Regs Toggle register window
 Paste	Stack Toggle stack window
	 Show "Help" contents

Save Environment

The **Save Environment** command gets Dynamic C 32 to update its Windows Registry entries and the **DCW.CFG** configuration file immediately with the current options settings. Dynamic C always updates these items on exit. Saving them while working provides an extra measure of security.

Window Menu

Click the menu title or press **ALT-W** to select the **Window** menu.



The first group of items is a set of standard Windows commands that allow application windows to be arranged in an orderly way.

The second group of items presents the various Dynamic C debugging windows. Click on one of these to activate or deactivate the particular window. It is possible to scroll these windows to view larger portions of data, or copy information from these windows and paste the information as text anywhere. The contents of these windows can be printed.

The third group is a list of current windows, including source code windows. Click on one of these items to bring that window to the front.

The individual Window menu commands are described here.

Cascade

The **Cascade** command gets Dynamic C to display windows “on top of each other,” but with a small offset. The window being worked in is displayed in front of the rest.

Tile Horizontally

The **Tile Horizontally** command gets Dynamic C to display windows in *horizontal* (landscape) orientation, although the windows are stacked vertically.

Tile Vertically

The **Tile Vertically** command gets Dynamic C to display windows in *vertical* (portrait) orientation.

Arrange icons

When one or more Dynamic C windows have been minimized, they are displayed as icons. The **Arrange icons** command arranges them neatly.

Message

Click the **Message** command to activate the **Message window**. A compilation with errors also activates the message window because the message window displays compilation errors.

Watch

The **Watch** command activates the **Watch window**. The **Add/Del Items** command on the **Inspect** menu will do this too. The **Watch window** displays the results whenever Dynamic C evaluates watch expressions.

Stdio

Click the **Stdio** command to activate the **STDIO window**. The **STDIO** window displays output from calls to `printf`. If the program calls `printf`, Dynamic C will activate the **STDIO** window automatically, unless another request was made by the programmer (see *Debugger Options*).

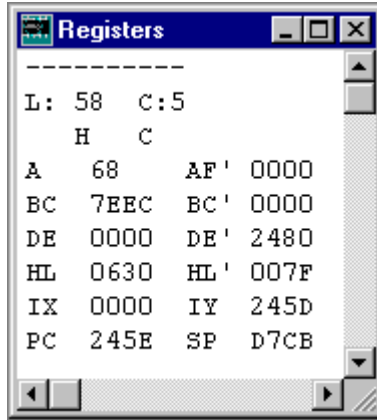
Assembly

Click the **Assembly** command to activate the **Assembly window**. The assembly window displays machine code generated by the compiler in assembly language format.

The **Disassemble at Cursor** or **Disassemble at Address** commands on the **Inspect** menu also activate the assembly window.

Registers

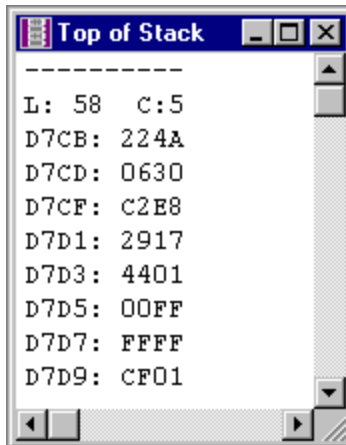
Click the **Registers** command to activate the **Register window**.



The register window displays the Z180 register set. Letter codes indicate the bits of the status (or flags, F) register. The window also shows the source-code line and column at which the register “snapshot” was taken. It is possible to scroll back to see the succession of register snapshots.

Stack

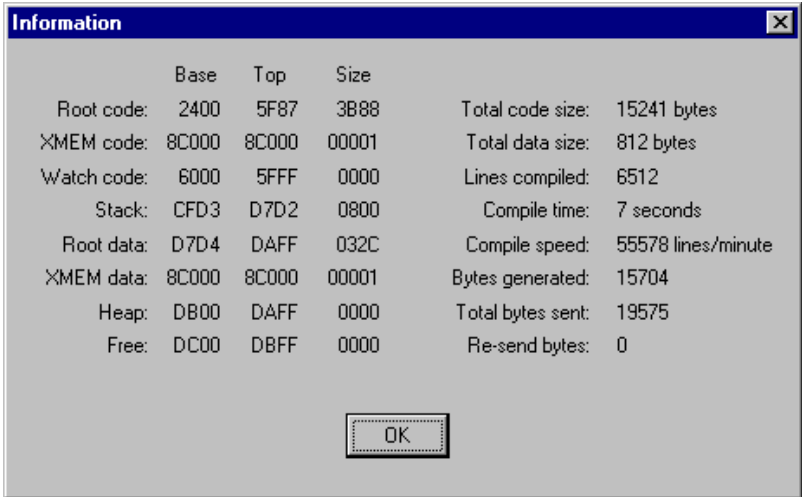
Click the **Stack** command to activate the **Stack window**.



The stack window displays the top 8 bytes of the run-time stack. It also shows the line and column at which the stack “snapshot” was taken. It is possible to scroll back to see the succession of stack snapshots.

Information

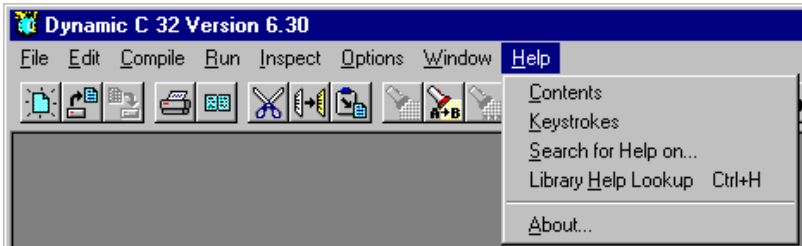
Click the **Information** command to activate the **Information window**.



The information window displays how the memory is partitioned and how well the compilation went. In this example, no space has been allocated to the heap or free space. The base and top of these memory partitions can be changed with commands from the **Options** menu.

Help Menu

Click the menu title or press **ALT-H** to select the **Help** menu. The **Help** menu commands and their functions are described here.



Contents

This standard item displays the contents page of the on-line help system.

Keystrokes

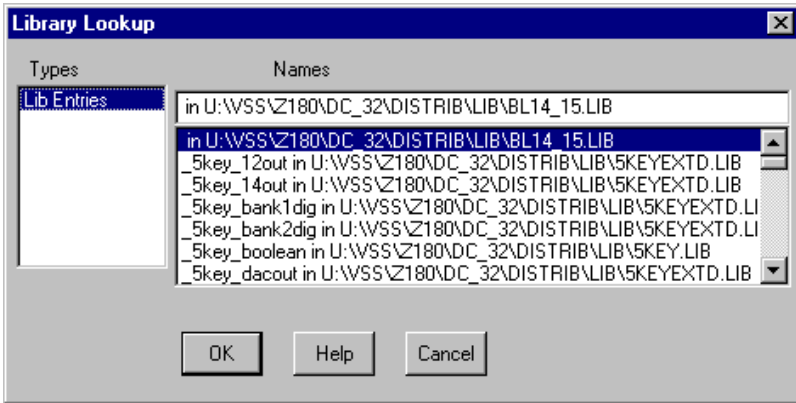
Select this item for information on available keystrokes and their functions. Many commands on the Dynamic C menus are also available directly through the keyboard. In addition, some operations can only be performed through the keyboard (certain cursor movement and editing operations).

Search for Help on...

Select this standard item to search for help on a particular topic. Type in a keyword and press **ENTER** to see a list of related topics. Then select a topic from the list and press **ENTER** again to view the topic.

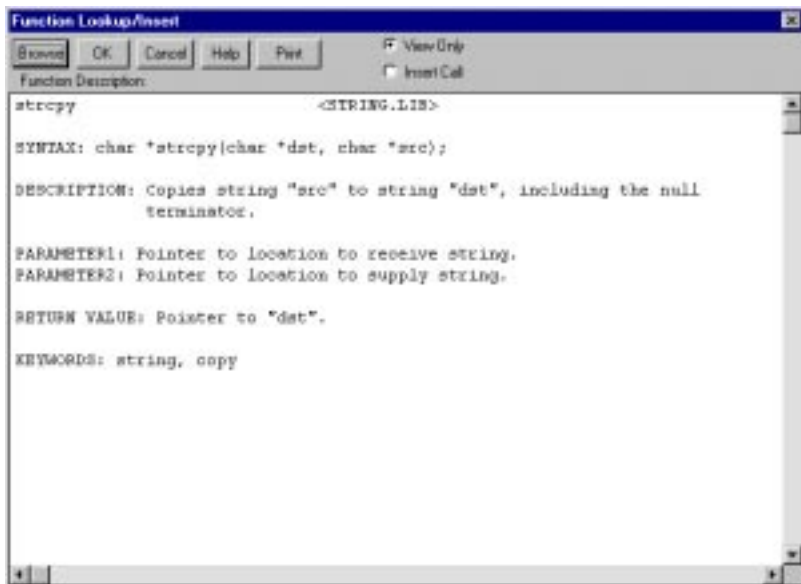
Library Help Lookup

Obtains help information for library functions. When a function name is clicked or selected in source code and then the help command is issued, Dynamic C displays help information for that function. The keyboard shortcut is **<CTRL-H>**. If Dynamic C cannot find a unique description for the function, it will display the following dialog box.

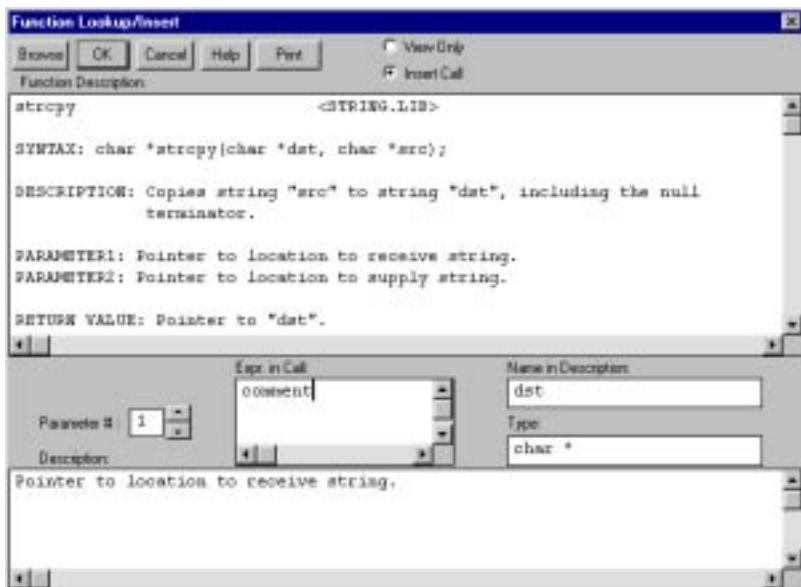


Click **Lib Entries** to display a list of the library functions currently available to the program in the libraries named in the **LIB.DIR** file. Then select a function name from the list to receive information about that function.

Dynamic C displays a dialog like the following one when a function is selected for display of help information.



Although this may be sufficient for most purposes, the **Insert Call** button can be clicked to turn the dialog into a “function assistant.”



The function assistant will place a call to the function displayed at the insertion point in the source code. The function call will be prototypical if OK is clicked; the call needs to be edited for it to make sense in the context of the code.

Each parameter can be specified, one-by-one, to the function assistant. The function assistant will return the name and data type of the parameter. When parameter expressions are specified in this dialog, the function assistant will use those expressions when placing the function call.

If the text cursor is placed on a valid C function call (and one that is known to the function assistant), the function assistant will analyze the function call, and will copy the actual parameters to the function lookup dialog. Compare the function parameters in the **Expr. in Call** box in the dialog with the expected function call arguments.

Consider, for example, the following code.

```
...  
    x = strcpy( comment, "Lower tray needs paper." );  
...
```

If the text cursor is placed on **strcpy** and the **Library Help Lookup** command is issued, the function assistant will show **comment** as parameter 1 and **Lower tray needs paper.** as parameter 2. The arguments can then be compared with the expected parameters, and the arguments in the dialog can then be modified.

The function help dialog will probably be needed only when the programmer is unfamiliar with or unsure of a function.

About...

The Windows standard **About...** item displays the Dynamic C 32 version number and the Z-World, Inc. copyright notice.



CHAPTER 5: ***THE LANGUAGE***

This chapter is not intended to be a C-language tutorial. The reader is expected to know how to program, and to know the basic principles of the C language. The objective of this chapter is to

1. Present the C-language features, and
2. Review the differences between C and Dynamic C.

Most punctuation in the examples is literal: it is generally required where examples indicate.

The C language is “case-sensitive,” that is, upper case (capital) letters are distinct from lower case letters. The term `putchar` is not the same as `PutChar`. All keywords in C are lower case.

This manual shows syntax by example rather than by any formalism.



For a more formal treatment of the C language, refer to the many good textbooks available.

Overview

Program Files

Programs are built by creating text files containing program code (that is, source files). Then there are libraries—files of useful functions. There are many library files already in the Dynamic C **LIB** subdirectory. The default library file extension is **.LIB**.

A controller program requires at least one application file containing the main program and perhaps other functions and global data. The default application file extension is **.C**. (There are many sample programs in the **SAMPLES** subdirectory.)

Dynamic C links the application program to functions and data in the other files selected for use with the application. The compiler will extract the functions and data when needed.

Code in the BIOS of the target controller (or the **RTI** file) is also linked (and is very important) to the program.

Thus, the overall structure of an application consists of a main program (called **main**), zero or more functions, and zero or more global data, all of which are distributed throughout one or more text files. The order in which these are defined is not very important. The minimum program is one file, containing only

```
main() { }
```

Libraries are “linked” with the application through the `#use` directive. The `#use` directive identifies a file from which functions and data may be extracted. Files identified by `#use` directives are nestable, as shown in Figure 5-1.

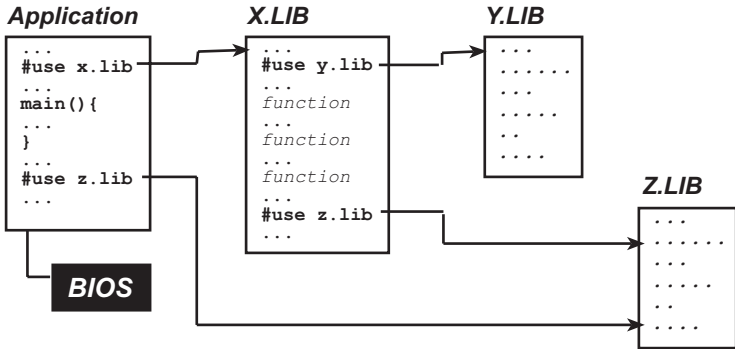


Figure 5-1. Nesting Files in Dynamic C

The Modules section later in this chapter explains how Dynamic C knows which functions and global variables in a library to use.

Support Files

Dynamic C has several support files without which it is not possible to build an application. These files are listed in Table 5-1.

Table 5-1. Dynamic C Support Files

File	Meaning
DCW.INI	Most Windows applications have <code>.INI</code> files. This <code>.INI</code> file is the one for Dynamic C. It contains the display options and other environmental parameters.
DCW.CFG	Contains configuration data for the target controller.
DC.HH	Contains prototypes, basic type definitions, <code>#defines</code> , and default modes for Dynamic C. This file can be modified by the programmer.
LIB.DIR	Contains pathnames for all libraries that are to be known to Dynamic C. The programmer can add to, or remove libraries from this list. The factory default is for this file to contain all libraries on the Dynamic C distribution disk.
DEFAULT.H	Contains a set of <code>#use</code> directives for each control product that Z-World ships. This file can be modified.

Statements

Except for comments, everything in a C program is a *statement*. Virtually *all* statements end with a semicolon. A C program is treated as a stream of characters where line boundaries are (generally) not meaningful. Any C statement may be written on as many lines as needed. Comments (the `/* . . . */` kind) may occur almost anywhere, even in the middle of a statement, as long as they begin with `/*` and end with `*/`.

A statement can be many things. A declaration of variables is a statement. An assignment is a statement. A `while` or `for` loop is a statement. A *compound* statement is a group of statements enclosed in curly brackets { and }.

Declarations

A variable must be *declared* before it can be used. That means the variable must have a name and a type, and perhaps its storage class could be specified. If an array is declared, its size must be given.

```
// static integer variable & static integer array
static int thing, array[12];

// auto float array with 2 dimensions
auto float matrix[3][3];

// initialized pointer to char array
char *message = "Press any key...";
```

If an aggregate type (`struct` or `union`) is being declared, its internal structure has to be described, as shown below.

```
struct { // description of struct
    char flags;
    struct { // a nested structure here
        int x;
        int y;
    } loc;
} cursor;

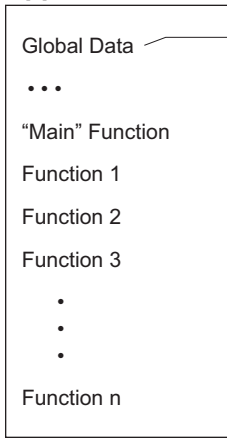
...

int a;
a = cursor.loc.x; // use of struct element here
```

Functions

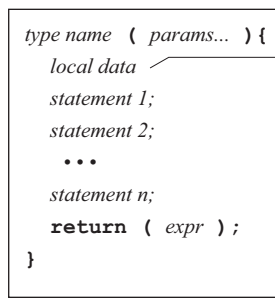
The basic unit of a C-language application program is a function. Most functions accept parameters—or arguments—and return results, but there are exceptions. All C functions have a return type that specifies what kind of result, if any, it returns. A function with a `void` return type returns no result. If a function is declared without specifying a return type, the compiler assumes that it is to return an `int` (integer) value. See Figure 5-2.

Application



Items accessible by all functions

Function



Items accessible by this function only

Figure 5-2. Functions in C Programming

Functions may *call* other functions. (A function may even call itself. Programmers call such a function a recursive function.) The **main** function is called automatically after the program compiles or when the controller powers up. The beginning of the **main** function is the entry point to the entire program.

Prototypes

A function may be declared with a *prototype*. This is so that

1. Functions that have not been compiled may be called.
2. Recursive functions may be written.
3. The compiler may perform type-checking on the parameters to make sure that calls to the function receive arguments of the expected type. A function prototype describes how to call the function and is nearly identical to the function's initial code.

```
// this is a function prototype  
long tick_count ( char clock_id );  
  
// this is the function's definition  
long tick_count ( char clock_id ) {  
    ...  
}
```

It is not necessary to provide parameter names in a prototype, but the parameter type is required, and all parameters must be included. (If the function accepts a variable number of arguments, as `printf` does, use an ellipsis.)

```
// this prototype is as good as the one above
long tick-count ( char );
// this is a prototype that uses ellipsis
int startup ( device id, ... ){
    ...
}
```

Type Definitions

Both types and variables may be defined. One virtue of high-level languages such as C and Pascal is that abstract data types can be defined. Once defined, the data types can be used as easily as simple data types like `int`, `char` and `float`. Consider this example.

```
typedef int MILES; // a basic type named MILES
typedef struct{ // a structure type...
    float re; // ...
    float im; // ...
} COMPLEX; // ...named COMPLEX

MILES distance; // declare variable of type MILES
COMPLEX z, *zp; // declare complex variable and ptr
```

Use `typedef` to create a meaningful name for a class of data. Consider this example.

```
typedef uint node;
void NodeInit( node ); // informative type name
void NodeInit( uint ); // not very informative
```


This example shows many of the basic C constructs.

```
/* Put descriptive information in your program
code using this form of comment, which can be
inserted anywhere and can span lines. The double
slash comment (shown below) may be placed at end-
of-line.*/

// Make driver functions available.
#use drivers.lib
// Define a macro (E.G.: a symbolic constant).
#define SIZE 12

int g, h;          // Define global integers.

// Declare prototypes for functions defined below.
float sumSquare(int, int);
void init();

// Program starts here.
main() {
    float x;          // x is local to main.
    init(10);         // Call a void function.
    x = sumSquare(g, h); // x gets sumSquare
    value.
    printf("x = %f",x); // printf is a stan-
    dard
}                    // function.

// void functions do things but return no value.
void init(int a){    // Integer argument.
    g = a;          // g gets a (IE: 10).
    h = SIZE;      // h gets the symbolic
}                    // constant defined above.

// Other functions do things and return a value.
float sumSquare(int a, int b){ // Integer args.
    float temp;    // Local (auto) variable.
    temp = a*a + b*b; // Arithmetic.
    return( temp ); // Return value.
}

/* and here is the end of the program */
```

This program calculates the sum of squares of two numbers, **g** and **h**, which are initialized to 10 and 12, respectively. The **main** function calls the **init** function to give values to the global variables **g** and **h**. Then it uses the **sumSquare** function to perform the calculation and assign the result of the calculation to the variable **x**. It prints the result using the library function **printf**, which includes a formatting string as the first argument.

Notice that all functions have { and } enclosing their contents, and all variables are declared before use. The functions `init` and `sumSquare` were defined before use, but there are alternatives to this. The Prototypes section earlier in this chapter explained this.

Modules

Modules provide Dynamic C with the ability to know which functions and global variables in a library to use.

A library file contains a group of *modules*. A module has three parts: the key, the header, and a body of code (functions and data).

A module in a library has a structure like this one.

```
/** BeginHeader func1, var2, .... */
    prototype for func1
    declaration for var2
/** EndHeader */
    definition of func1 var2 and possibly other
    functions and data
```

The Key

The line (a specially-formatted comment)

```
/** BeginHeader name1, name2, .... */
```

begins the header of a module and contains the module *key*. The *key* is a list of names (of functions and data). The key tells the compiler what functions and data in the module are available for reference. It is important to format this comment properly. Otherwise, Dynamic C cannot identify the module correctly.

If there are many names after `BeginHeader`, the list of names can continue on subsequent lines. All names must be separated by commas.

The Header

Every line between the comments containing `BeginHeader` and `EndHeader` belongs to the *header* of the module. When an application `#uses` a library, Dynamic C compiles every header, and just the headers, in the library. The purpose of a header is to make certain names defined in a module known to the application. With proper function prototypes and variable declarations, a module header ensures proper type checking throughout the application program.

The Body

Every line of code after the `EndHeader` comment belongs to the *body* of the module until (1) end-of-file or (2) the `BeginHeader` comment of another module. Dynamic C compiles the *entire* body of a module if *any* of the names in the key are referenced (used) anywhere in the application.

To minimize waste, it is recommended that a module header contain only prototypes and **extern** declarations. (Prototypes and **extern** declarations do not generate any code by themselves.) Define code and data only in the body of a module. That way, the compiler will generate code or allocate data *only* if the module is used by the application program. Programmers who create their own libraries must write modules following the guideline in this section. Remember that the library must be included in `LIB.DIR` and a `#use` directive for the library must be placed somewhere in the code.

Example

```
/**/ BeginHeader ticks */
extern ulong ticks;
/**/ EndHeader */

ulong ticks;

/**/ BeginHeader Get_Ticks */
ulong Get_Ticks();
/**/ EndHeader */

ulong Get_Ticks(){
    ...
}

/**/ BeginHeader Inc_Ticks */
void Inc_Ticks( int i );
/**/ EndHeader */

#asm
Inc_Ticks::
    or    a
    di
    ...
    ei
    ret
#endasm
```

There are three modules defined in this code. The first one is responsible for the variable `ticks`, the second and third modules define functions `Get_Ticks` and `Inc_Ticks` that access the variable.

Note that although `Inc_Ticks` is actually an assembly language routine, it has a function prototype in the module header, allowing the compiler to check calls to `Inc_Ticks`.

If the application program calls `Inc_Ticks` or `Get_Ticks` (or both), the module bodies corresponding to the called routines will be compiled. The compilation of these routines further triggers compilation of the module body corresponding to `ticks` because the functions use the variable `ticks`.

Macros

Macros can be defined in Dynamic C. A macro is a name replacement feature. Dynamic C has a text preprocessor that *expands* macros before the program text is compiled. The programmer assigns a name to a fragment of text. Subsequently, Dynamic C replaces the name with the text fragment wherever the macro name appears in the program (this is a macro *call*). In this example,

```
#define OFFSET 12
#define SCALE 72
int i, x;
i = x * SCALE + OFFSET;
```

the variable `i` gets the value `x * 72 + 12`. Macros can have parameters. For example,

```
#define word( a, b ) (a<<8 | b)
char c;
int i, j;
i = word( j, c );           // same as i = (j<<8|c)
```

The compiler removes surrounding white space (comments, tabs and spaces) and collapses each sequence of white space in the macro definition into one space. It places a `\` before any `"` or `\` to preserve their original meaning within the definition.

Dynamic C implements the `#` and `##` macro operators. The `#` operator forces the compiler to interpret the parameter immediately following as a string literal. For example, if a macro is defined

```
#define report(value,fmt)\
    printf( #value "=" #fmt "\n", value )
```

then the macro in

```
report( string, %s );
```

will expand to

```
printf( "string" "=" "%s" "\n", string );
```

and because C always concatenates adjacent strings, the final result is

```
printf( "string=%s\n", string );
```

The `##` operator concatenates the preceding character sequence with the following character sequence, deleting any white space in between. Given

```
#define set(x,y,z)    x ## z ## _ ## y()
```

the macro in

```
set( AASC, FN, 6 );
```

will expand to

```
AASC6_FN();
```

For parameters immediately adjacent to the `##` operator, the corresponding argument is not expanded before substitution, but appears as it does in the macro call.

Generally speaking, Dynamic C expands macro calls recursively until they can expand no more. Another way of stating this is that macro definitions can be nested.

The exceptions to this rule are

1. Arguments to the `#` and `##` operators are not expanded.
2. To prevent infinite recursion, a macro does not expand within its own expansion.

The following complex example illustrates this.

```
#define A                B
#define B                C
#define uint            unsigned int
#define M(x)            M ## x
#define MM(x,y,z)      x = y ## z
#define string          something
#define write( value, fmt )\
    printf( #value "=" #fmt "\n", value )
```

The code

```
uint z;
M (M) (A,A,B);
write(string, %s);
```

will expand first to

```
unsigned int z;           // simple expansion
MM (A,A,B);              // M(M) does not expand recursively
printf( "string" "=" "%s" "\n", string );
                        // #value → "string" #fmt → "%s"
```

then to

```
unsigned int z;
A = AB;                  // from A = A ## B
printf( "string" "=" "%s" "\n", something );
                        // string → something
```

then to

```
unsigned int z;
B = AB;                  // A → B
printf( "string=%s\n", something ); // concatenation
```

and finally to

```
unsigned int z;
C = AB;                  // B → C
printf("string = %s\n", something);
```

Restrictions: The number of arguments in a macro call must match the number of parameters in the macro definition. An empty parameter list is allowed, but the macro call must have an empty argument list. Macros are restricted to 32 parameters and 126 nested calls. A macro or parameter name must conform to the same requirements as any other C name. The C language does not perform macro replacement inside string literals or character constants, comments, or within a **#define** directive.

A macro definition remains in effect unless removed by an **#undef** directive. If an attempt is made to redefine a macro without using **#undef**, a warning will appear and the original definition will remain in effect.

Program Flow

Three terms describe the flow of execution of a C program: sequencing, branching and looping. **Sequencing** is simply the execution of one statement after another. **Looping** is the repetition of a group of statements. **Branching** is the choice of groups of statements.

Program flow is altered by “calling” a function, that is transferring control to the function. Control is passed back to the calling function when the called function returns.

In the following descriptions, the *recommended* form allows the programmer to enclose any number of statements in the body of a control structure simply by adding or deleting lines of code. Strictly speaking, the C language does not require this regularity.

Loops

A **while** loop tests a condition at the *start* of the loop. As long as the expression is true (that is, nonzero), the loop body (statement or compound statement) governed by the while expression will execute. If the expression is initially false (zero), the program will skip the loop body altogether.

Recommended form	C syntax
<pre>while(expression){ some statements }</pre>	<pre>while(expression) statement</pre>

A **do** loop tests a condition at the *end* of the loop. As long as the expression is true (that is, nonzero) the loop body (statement or compound statement) governed by the while expression will repeat. A **do** loop executes at least once before its test. Unlike other controls, the **do** loop requires a semicolon at the end.

Recommended form	C syntax
<pre>do{ some statements }while(expression);</pre>	<pre>do statement while(expression);</pre>

The **for** loop is more complex: it sets an initial condition (exp_1), evaluates a terminating condition (exp_2), and provides a stepping expression (exp_3) that is evaluated at the end of each iteration. Each of the three expressions is optional.

Recommended form	C syntax
<pre>for(exp₁ ; exp₂ ; exp₃){ some statements }</pre>	<pre>for(exp₁ ; exp₂ ; exp₃) statement</pre>

If the end condition is initially false, a **for** loop body will not execute at all. A typical use of the **for** loop is to count n times.

```
sum = 0;
for( i = 0; i < n; i++ ){
    sum = sum + array[i];
}
```

This loops sets i to 0 initially, continues as long as i is less than n (stops when i equals n), and increments i at each pass. Another use for the **for** loop is the infinite loop, which is useful in control systems.

```
for(;;){
    some statements
}
```

Here, there is no initial condition, no end condition, and no stepping expression. The loop body (some statements) continues to execute endlessly.

Continue and Break

Two other constructs are available to help in the construction of loops: the **continue** statement and the **break** statement.

The **continue** statement causes the program control to skip unconditionally to the next pass of the loop.

Example	Equivalent Code
<pre> get_char(); while(! EOF){ some statements if(bad) continue; more statements } </pre>	<pre> get_char(); while(! EOF){ some statements if(bad) goto xxx; more statements xxx: } </pre>

The **break** statement causes the program control to jump unconditionally out of a loop.

Example	Equivalent Code
<pre> for(i=0;i<n;i++){ some statements if(cond_RED) break; more statements } </pre>	<pre> for(i=0;i<n;i++){ some statements if(cond_RED) goto yyy; more statements } yyy: more code </pre>

The **break** keyword also applies to the **switch/case** statement described in the next section. The **break** statement jumps out of the innermost control structure (loop or switch statement) only.

There will be times when **break** is insufficient. The program will need to either jump out more than one level of nesting or there will be a choice of destinations when jumping out. Use a **goto** statement in such cases. For example,

```

while( some statements )
    for( i=0;i<n;i++ ){
        some statements
        if( cond_RED ) goto yyy;
        some statements
        if( code_BLUE ) goto zzz;
        more statements
    }
}
yyy:
    handle cond_RED
zzz:
    handle code_BLUE

```


Branching

The **goto** statement mentioned previously is the simplest form of branching statement. Coupled with a statement label, it simply transfers program control to the labeled statement.

```
    some statements
abc:
    other statements
goto abc;
    ...
    more statements
goto def;
    ...
def:
    more statements
```

Notice the colon (:) at the end of the labels.

The next simplest form of branching is the **if** statement. The simple form of the **if** statement tests a condition and executes a statement or compound statement if the condition expression is true (nonzero). The program will ignore the **if** body when the condition is false (zero).

Recommended form	C syntax
<pre>if(<i>expression</i>){ <i>some statements</i> }</pre>	<pre>if(<i>expression</i>) <i>statement</i></pre>

A more complex form of the **if** statement tests the condition and executes certain statements if the expression is true, and executes another group of statements when the expression is false.

Recommended form	C syntax
<pre>if(<i>expression</i>){ <i>some statements if true</i> }else{ <i>some statements if false</i> }</pre>	<pre>if(<i>expr</i>) <i>stmt_T</i> else <i>stmt_F</i></pre>

The fullest form of the **if** statements produces a “chain” of tests.

Recommended form	C syntax
<pre>if (<i>expr</i>₁) { <i>some statements</i> } else if (<i>expr</i>₂) { <i>some statements</i> } else if (<i>expr</i>₃) { <i>some statements</i> ... } else { <i>some statements</i> }</pre>	<pre>if (<i>expr</i>₁) <i>stmt</i>₁ else if (<i>expr</i>₂) <i>stmt</i>₂ else if (<i>expr</i>₂) <i>stmt</i>₂ else if (<i>expr</i>₃) <i>stmt</i>₃ ... else <i>stmt</i>_n</pre>

The program evaluates the first expression (*expr*₁). If that proves false, it tries the second expression (*expr*₂), and continues testing until it finds a true expression, an **else** clause, or the end of the **if** statement. An **else** clause is optional. Without an **else** clause, an **if** statement that finds no true condition will execute none of the controlled statements.

The **switch** statement, the most complex branching statement, allows the programmer to phrase a “multiple choice” branch differently.

Recommended form
<pre>switch (<i>expression</i>) { case <i>const</i>₁ : <i>statements</i>₁ break ; case <i>const</i>₂ : <i>statements</i>₂ break ; case <i>const</i>₃ : <i>statements</i>₃ break ; ... default : <i>statements</i>_{DEFAULT} }</pre>

The **switch** expression is evaluated. It must have an integer value. If one of the *const*_N expressions matches the **switch** expression, the sequence of statements identified by the *const*_N expression is executed. If there is no match, the sequence of statements identified by the **default** label is executed. (The **default** part is optional.)

Unless the **break** keyword is included at the end of the case's statements, the program will "fall through" and execute the statements for any number of other cases. The **break** keyword causes the program to exit the **switch / case** statement.

Notice the colons (:) at the end of the **cases** and after **default**.

Data

Data (variables and constants) have type, size, structure, and storage class.

Primitive Data Types

Basic, or primitive, data types are provided in Table 5-2.

Table 5-2. Dynamic C Basic Data Types

Type	Description
char	8-bit unsigned integer. 8-bit characters fit precisely into a char , hence the name. Range: 0 to 255 (0xFF)
int	16-bit signed integer. Range: -32,768 to +32,767
unsigned int	16-bit unsigned integer. In this manual, the term uint is shorthand for unsigned int . Range: 0 to 65,535
long	32-bit signed integer. Range: -2,147,483,648 to +2,147,483,647
unsigned long	32-bit unsigned integer. In this manual, the term ulong is shorthand for unsigned long . Range: 0 to $2^{32} - 1$
float	32-bit IEEE floating-point value. The sign bit is 1 for negative values. The exponent has 8 bits, giving exponents from -127 to +128. The mantissa has 24 bits. Only the 23 least significant bits are stored; the high bit is implicitly 1. (Z180 controllers do not have floating point hardware.) Range: -6.085×10^{38} to $+6.085 \times 10^{38}$

C supports string constants but not a string data type. (A string in C is really an array of characters.)

The structures of the primitive data types are shown in relative size in Figure 5-3.

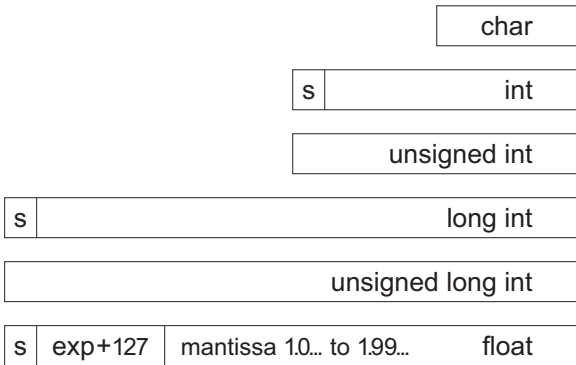


Figure 5-3. Structures of Dynamic C Primitive Data Types

Aggregate Data Types

Simple data types can be grouped into more complex *aggregate* forms.

Array

A data type, whether it is simple or complex, can be replicated in an *array*. The declaration

```
int item[10];           // an array of 10 integers
```

represents a contiguous group of 10 integers. Array elements are referenced by their subscript.

```
j = item[n];           // the nth element of item
```

Array subscripts count up from 0. Thus, `item[7]` above is the *eighth* item in the array. Notice the `[` and `]` enclosing both array dimensions and array subscripts. Arrays can be “nested.” The following doubly dimensioned array, or “array of arrays”.

```
int matrix[7][3];
```

is referenced in a similar way.

```
scale = matrix[i][j];
```

The first dimension of an array does not have to be specified as long as an initialization list is specified.

```
int x[][2] = { {1, 2}, {3, 4}, {5, 6} };  
char string[] = "abcdefg";
```

Structure

Variables may be grouped together in *structures* (**struct** in C) or in arrays. Structures may be nested.

```
struct {
    char flags;
    struct {
        int x;
        int y;
    } loc;
} cursor;
```

Structures can be nested. Structure *members*—the variables within a structure—are referenced using the dot operator.

```
j = cursor.loc.x
```

The size of a structure is the sum of the sizes of its components.

Union

A *union* overlays simple or complex data. That is, all the union members have the same address. The size of the union is the size of the largest member.

```
union {
    int ival;
    long jval;
    float xval;
} u;
```

Unions can be nested. Union members—the variables within a union—are referenced, like structure elements, using the dot operator.

```
j = u.ival
```

Composites

Composites of structures, arrays, unions, and primitive data may be formed. This example shows an array of structures that have arrays as structure elements.

```
typedef struct {
    int *x;
    int c[32];           // array in structure
} node;
node list[12];         // array of structures
```

Refer to an element of array **c** (above) as shown here.

```
z = list[n].c[m];
...
list[0].c[22] = 0xFF37;
```

Storage Classes

Variable storage can be **static**, **auto**, or **register**. These terms apply to local variables, that is, variables defined within a function. If a variable does not belong to a function, it is called a *global variable*—meaning available anywhere—but there is no keyword in C to represent this fact. Global variables (not declared within a function) always have **static** storage.

The term **static** means the data occupies a permanent fixed location for the life of the *program*. The term **auto** refers to variables that are placed on the system stack for the life of a *function call*.

The term **register** describes variables that are allocated as if they were static variables, but their values are saved on function entry and restored when the function returns. Thus, **register** variables can be used with reentrant functions as can **auto** variables, yet they have the speed of static variables.

Variables and structures may be created dynamically from free memory space (the “heap”). The standard C functions **malloc** and **free** allocate and release blocks of storage. Such dynamic variables are neither local nor global. The program accesses dynamic variables through pointers.

Pointers

A pointer is a variable that holds the 16-bit logical address of another variable, a structure, or a function. Variables can be declared pointers with the indirection operator (*****).

```
int *index;
```

In this example, the variable **index** is a pointer to an integer. The statement

```
j = *index;
```

references the value of the integer by the use of the asterisk. Pointers may point to other pointers.

```
int **thing;    // ptr to a ptr to an integer
j = **thing;    // j gets the value ref'd by thing
```

Conversely, a pointer can be set to the address of a variable using the **&** (address) operator.

```
int *p, thing;
p = &thing;
```

Then ***p** and **thing** have identical values. (But note that **p** and **thing** do not, since **p** is a pointer and **thing** is an **int**.)

It is possible to do pointer arithmetic, but this is slightly different from ordinary integer arithmetic. Here are some examples.

```
typedef ... xyz;      // arbitrary type & size
xyz f[10], *p, *q;    // an array and some ptrs
...
p = &f;               // p → array element 0
q = p+5;              // q → array element 5
q++;                  // q → array element 6
p = p + q;            // illegal!
```



Beware of using uninitialized pointers. Uninitialized pointers can reference ANY location in memory. Storing data using an uninitialized pointer can overwrite code or cause a fault.

A common mistake is to declare and use a pointer to **char**, thinking there is a string. But an uninitialized pointer is all there is.

```
char* string;
...
strcpy( string, "hello" ); // invalid!
printf( string );         // Invalid!
```

Pointer checking is a run-time option of Dynamic C. Use the compiler options command in the **OPTIONS** menu.

Argument Passing

In C, function arguments are generally passed *by value*. That is, arguments passed to a C function are generally copies—on the program stack—of the variables or expressions specified by the caller. Changes made to these copies do not affect the original values in the calling program.

In Dynamic C and most other C compilers, however, arrays are always passed by address. This policy includes strings (which are character arrays).

Dynamic C passes **structs** by value—on the stack. Passing a large **struct** takes a long time and can easily cause a program to run out of memory. Pass pointers to large **structs** if such problems occur.

For a function to modify the original value of a parameter, pass the address of, or a pointer to, the parameter and then design the function to accept the address of the item.

Memory Management

Z180 instructions can specify 16-bit addresses, giving a *logical* address space of 64 KBytes (65,536 bytes). Dynamic C supports a 1-megabyte *physical* address space (20-bit addresses).

An on-chip memory management unit (MMU) translates 16-bit Z180 addresses to 20-bit memory addresses. Three MMU registers (CBAR, CBR, and BBR) divide the logical space into three sections and map each section onto physical memory, as shown in Figure 5-4.

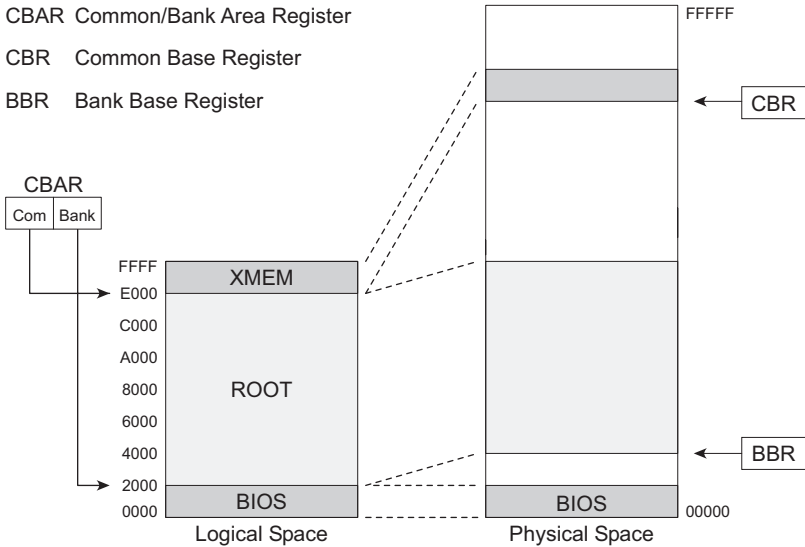


Figure 5-4. Z180 On-Chip Memory Management Unit (MMU) Registers

The logical address space is partitioned on 4-KByte boundaries. The upper half of CBAR identifies the boundary between the **ROOT** memory and **XMEM**. The lower half of CBAR identifies the boundary between the **BIOS** and the **ROOT**. The start of the **BIOS** is always address 0. The two base registers CBR and BBR map **XMEM** and **ROOT**, respectively, onto physical memory.

Given a 16-bit address, the Z180 uses CBAR to determine whether the address is in **XMEM**, **BIOS**, or **ROOT**. If the address is in **XMEM**, the Z180 uses the CBR as the base to calculate the physical address. If the address is in **ROOT**, the Z180 uses the BBR. If the address is in the **BIOS**, the Z180 uses a base of 0.

A physical address is, essentially,
 (base << 12) + logical address.

Figure 5-5 shows the address locations.

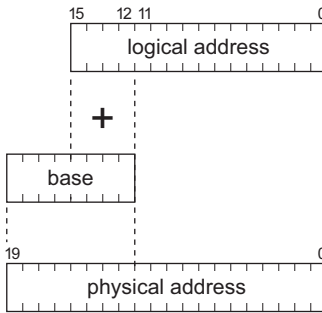


Figure 5-5. Z180 Physical Addresses

Memory Partitions

Table 5-3 explains the memory partitions in Dynamic C.:

Table 5-3. Dynamic C Memory Partitions

Name	Size	Description
BIOS	8 kbytes	Basic Input/Output System. The BIOS is always present and is always mapped to address 0 of ROM or flash. The BIOS contains the power-up code, the communication kernel, and important system features.
ROOT	48 kbytes	The area between the BIOS and XMEM (the bank area). The root—"normal" memory—resides in a fixed portion of physical memory. Root code grows upward in logical space from address 2000 (hex) and root data (static variables, stack and heap) grow down from E000. (Initialized static variables are placed with code, whether in ROM, flash, or RAM.)
XMEM	8 kbytes	XMEM is essentially an 8-kbyte "window" into extended physical memory. XMEM can map to any part of physical memory (ROM, flash, or RAM) simply by changing the CBR.

The **XMEM** area has many mappings to physical memory. The mappings can change by changing the CBR as the program executes. Extended memory functions are mapped into **XMEM** as needed by changing the CBR. The mapping is automatic in C functions. However, code written in assembly language that calls functions in extended memory may need to do the mapping more specifically.

Functions may be classified as to where Dynamic C may load them. The keywords in Table 5-4 apply to function definitions.

Table 5-4. Memory Keyword Definitions

Keyword	Description
root	The function must be placed in root memory. It can call functions residing in extended memory.
xmem	The function must be placed in extended memory. Calls to extended memory functions are not as efficient as calls to functions in root memory. Long or infrequently used functions are appropriate for placement in extended memory.
anymem	This keyword lets the compiler decide where to place the function. A function's placement depends on the amount of reserve memory available. Refer to the Memory Options command in the OPTIONS menu.

Depending on which compiler options are selected, code segments will be placed in RAM, ROM, or flash.

Figure 5-6 shows the memory layout with code in RAM.

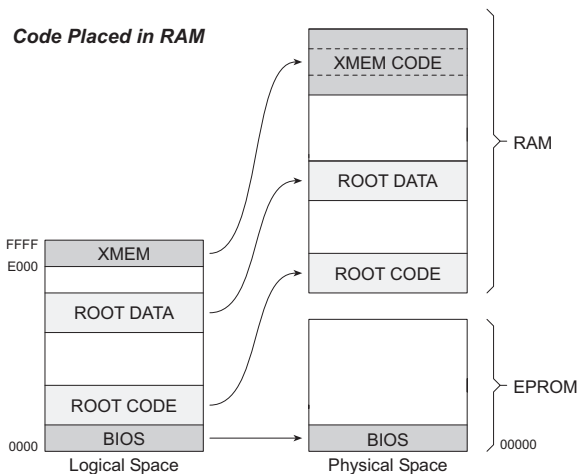


Figure 5-6. Memory Layout with Code in RAM

Figure 5-7 shows the memory layout with code in ROM or flash.

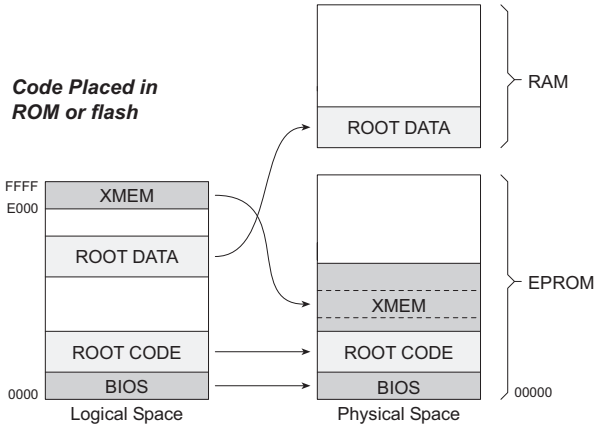


Figure 5-7. Memory Layout with Code in ROM or Flash

C Language Elements

A Dynamic C program is a set of files, each of which is a stream of characters that compose statements in the C language. The language has grammar and syntax, or rules for making statements. Syntactic elements—often called tokens—form the basic elements of the C language. Some of these elements are listed in Table 5-5.

Table 5-5. C Language Elements

keywords	Words used as instructions to Dynamic C
Names	Words used to name data
Numbers	Literal numeric values
Strings	Literal character values enclosed in quotes
operators	Symbols used to perform arithmetic
punctuation	Symbols used to mark beginnings and endings
directives	Words that start with # and control compilation

Keywords

A keyword is a reserved word in C that represents a basic C construct. The word **while** represents the beginning of a “while” loop. It cannot be used for any other purpose. There are many keywords, and they are summarized in the following pages.

abort—Jumps out of a costatement.

```
for(;;){
    costate {
        ...
        if( condition ) abort;
    }
    ...
}
```



See Chapter 7, Costatements.

anymem—Allows the compiler to determine in which part of memory a function will be placed.

```
anymem int func(){
    ...
}
#memmap anymem
#asm anymem
...
#endasm
```

auto—A local function variable is located on the system stack and exists as long as the function call does.

```
int func(){
    auto float x;
    ...
}
```

break—Jumps out of a loop, if, or case statement.

```
while( expression ){
    ...
    if( condition ) break;
}
switch( expression ){
    ...
    case 3:
        ...
        break;
    ...
}
```

case—Identifies the next “case” in a switch statement.

```
switch( expression ){
    case const :
        ...
    case const:
        ...
    case const:
        ...
    ...
}
```

char—Declares a variable, or array, as a type character. This type is also commonly used to declare 8-bit integers and “Boolean” data.

```
char c, x, *string = "hello";
int i;
...
c = (char)i;
```

continue—Skip to the next iteration of a loop.

```
while( expression ){
    if( nothing to do ) continue;
    ...
}
```

costate—Indicates the beginning of a costatement.

```
costate [ name [ state ] ] {
    ...
}
```

Name can be absent. If name is present, state can be **always_on** or **init_on**. If state is absent, the costatement is initially off.



See Chapter 7, Costatements, and keywords **abort**, **yield**, and **waitfor**.

debug—Indicates a function is to be compiled in debug mode.

```
debug int func(){
    ...
}

#asm debug
...
#endasm
```



See also **nodebug** and directives **#debug** and **#nodebug**.

default—Identifies the default “case” in a switch statement. The default case, which is optional, executes only when the switch expression does not match any other case.

```
switch( expression ){
    case const :
        ...
    case const:
        ...
    default:
        ...
}
```

do—Indicates the beginning of a do loop. A do loops tests at the end and executes at least once.

```
do
    ...
while( expression );
```

The statement must have a semicolon at the end.

else—Indicates a false branch of an if statement.

```
if( expression )
    statement // executes when true
else
    statement // executes when false
```

extern—Indicates that a variable is defined in the BIOS, later in a library file, or in another library file. Its main use is in module headers.

```
/**/ BeginHeader ..., var */
extern int var;
/**/ EndHeader */
int var;
...
```

firsttime—Declares a function to be a **waitfor** delay function.



For details, see Chapter 7, Costatements.

float—Declares a variable, function, or array, as 32-bit IEEE floating point.

```
int func(){
    float x, y, *p;
    float PI = 3.14159265;
    ...
}

float func( float par ){
    ...
}
```

for—Indicates the beginning of a **for** loop. A **for** loop has an initializing expression, a limiting expression, and a stepping expression. Each expression can be empty.

```
for(;;)                // an endless loop
    ...
}
for( i = 0; i < n; i++ ) // counting loop
    ...
}
```

goto—Causes a program to go to a labeled section of code.

```
...
if( condition ) goto RED;
...
RED:
    statements
```

Use **goto** to jump forward or backward in a program. Never use **goto** to jump *into* a loop body or a switch case. The results are unpredictable. However, it is possible to jump *out of* a loop body or switch case.

if—Indicates the beginning of an **if** statement.

```
if( tank_full ) shut_off_water();
if( expression ){
    statements
}else if( expression ){
    statements
}else if( expression ){
    statements
}else if( expression ){
    statements
    ...
}else{
    statements
}
```

If one of the expressions is true (they are evaluated in order), the statements controlled by that expression are executed.

An **if** statement can have zero or more **else if** parts. The **else** part is optional and executes when none of the **if** expressions is true (nonzero).

int—Declares a variable, function, or array to be an integer. If nothing else is specified, **int** implies a 16-bit *signed* integer.

```
int i, j, *k;           // 16-bit signed
unsigned int x;        // 16-bit unsigned
long int z;           // 32-bit signed
unsigned long int w;   // 32-bit unsigned

int funct ( int arg ){
...
}
```

interrupt—Indicates that a function is an interrupt service routine.

```
interrupt isr (){
...
}
```

An interrupt service routine returns no value and takes no arguments.



See also **ret**, **reti**, and **retn**.

long—Declares a variable, function, or array to be 32-bit integer. If nothing else is specified, **long** implies a *signed* integer.

```
long i, j, *k;           // 32-bit signed
unsigned long int w;     // 32-bit unsigned

long funct ( long arg ){
...
}
```

main—Identifies the **main** function. All programs start at the beginning of the **main** function. (This is actually not a keyword, but is a function name.)

nodebug—Indicates a function is not compiled in debug mode.

```
nodebug int func(){
...
}

#asm nodebug
...
#endasm
```



See also **debug** and directives **#debug** and **#nodebug**.

norst—Indicates that a function does not use the **RST** instruction for break points.

```
norst void func(){
...
}
```


nouseix—Indicates a function does not use the IX register as a stack frame reference pointer.

```
nouseix void func() {  
    ...  
}
```



See also **useix** and directives **#useix** and **#nouseix**.

NULL—The null pointer. (This is actually a macro, not a keyword.) Same as **(void *)0**.

pop—A keyword used in conjunction with certain directives (**#memmap** and **#class**). These directives can push and pop their states.

```
#memmap root  
    root functions  
  
#memmap push xmem  
    xmem functions here  
  
#memmap pop  
    now back to root functions
```

protected—Declares a variable to be “protected” against system failure. This means that a copy of the variable is made before it is modified. If a transient effect such as power failure occurs while the variable is being changed, the system will restore the variable from the copy.

```
main() {  
    protected int state1, state2, state3;  
    ...  
}
```

push—A keyword used in conjunction with certain directives (**#memmap** and **#class**). These directives can push and pop compilation modes.

```
#class static      // static local vars are default  
#class push auto  // auto local vars are default  
#class pop        // now back to static
```

register—Declares the storage class of a variable. The variable has the speed of a static variable, yet can be used in reentrant functions.

```
int func() {  
    register float x, y;  
    register int i;  
    ...  
}
```

ret—Indicates that an interrupt service routine (written in C) uses the **ret** instruction.

```
interrupt ret isr () {  
    ...  
}
```



See also **interrupt**.

reti—Indicates that an interrupt service routine (written in C) uses the **reti** instruction.

```
interrupt reti isr () {  
    ...  
}
```



See also **interrupt**.

retn—Indicates that an interrupt service routine (written in C) uses the **retn** instruction.

```
interrupt retn isr () {  
    ...  
}
```



See also **interrupt**.

return—Explicit return from a function. For functions that return values, this will return the function result.

```
void func () {  
    ...  
    if( expression ) return;  
    ...  
}  
  
float func (int x){  
    ...float temp;  
    ...  
    return( temp * 10 + 1 );  
}
```

root—Indicates a function is to be placed in root memory.

```
root int func(){  
    ...  
}  
  
#memmap root  
#asm root  
    ...  
#endasm
```

segchain—Identifies a function chain segment (within a function).

```
int func ( int arg ){
    ...int vec[10];
    ...
    segchain _GLOBAL_INIT{
        for( i = 0; i<10; i++ ){ vec[i] = 0; }
    }
    ...
}
```

This example adds a segment to the function chain `_GLOBAL_INIT`. When this function chain executes, this and perhaps other segments elsewhere execute. The effect in this example is to (re)initialize `vec`.

shared—Indicates that changes to a multi-byte variable (such as a `float`) are atomic. Interrupts are disabled while the variable is being changed. Local variables cannot be shared.

```
shared float x, y, z;
shared int j;
...
main(){
    ...
}
```

If `i` is a shared variable, expressions of the form `i++` (or `i = i+1`) constitute *two* atomic references to variable `i`, a read and a write. Be careful because `i++` is not an atomic operation.

short—Declares that a variable or array is short integer (16 bits). If nothing else is specified, short implies a 16-bit *signed* integer.

```
short i, j, *k;                // 16-bit, signed
unsigned short int w;          // 16-bit, unsigned
short funct ( short arg ){
    ...
}
```

size—Declares a function to be optimized for size (as opposed to speed).

```
size int func (){
    ...
}
```

sizeof—A built-in function that returns the size—in bytes—of a variable, array, structure, union, or of a data type.

```
j = 2 * sizeof(float);
int list[] = { 10, 99, 33, 2, -7, 63, 217 };
...
x = sizeof(list);
```

speed—Declares a function to be optimized for speed (as opposed to size).

```
speed int func () {  
    ...  
}
```

static—Declares a local variable to have a permanent fixed location in memory, as opposed to **auto**, where the variable exists on the system stack. Global variables are by definition **static**. Local variables are **static** by default, unlike standard C.

```
int func () {  
    ...int i;           // static by default  
    static float x;    // explicitly static  
    ...  
}
```

struct—Indicates the beginning of a structure definition.

```
struct {  
    ...int x;  
    int y;  
} abc;           // defines a struct object  
  
typedef struct {  
    ...int x;  
    int y;  
} xyz;          // defines a struct type...  
xyz thing;     // ...and a thing of type xyz
```

Structure definitions can be nested.

subfunc—Begins the definition of a subfunction. A subfunction encapsulates a useful code sequence and reduces the amount of storage required by the parent function.

```
func () {  
    int aname ();  
    subfunc aname: { k = inport (x); k + 4; }  
    ...  
    ... aname (); ...  
    ...  
    ... aname (); ...  
    ...  
}
```



See Appendix B, Efficiency, for details.

switch—Indicates the start of a switch statement.

```
switch( expression ){
    case const :
        ...
        break;
    case const :
        ...
        break;
    case const :
        ...
        break
    ...
    default :
        ...
}
```

The switch statement may contain any number of cases. It compares a case-constant expression with the switch expression. If there is a match, the statements for that case execute. The default case, if it is present, executes if none of the case-constant expressions match the switch expression.

If the statements for a case do not include a **break**, **return**, **continue**, or some means of exiting the **switch** statement, the cases following the selected case will execute, too, regardless of whether their constants match the switch expression.

typedef—Identifies a type definition statement. Abstract types can be defined in C.

```
typedef struct {
    int x;
    int y;
} xyz;           // defines a struct type...
xyz thing;      // ...and a thing of type xyz
typedef uint node; // meaningful type name
node master, slavel, slave2;
```

union—Identifies the beginning of a “union.” Items in a union have the same address. The size of a union is that of its largest member.

```
union {
    int x;
    float y;
} abc;           // overlays a float and an int
```

unsigned—Declares a variable or array to be unsigned. If nothing else is specified in a declaration, **unsigned** means 16-bit unsigned integer.

```
unsigned i, j, *k;           // 16-bit, unsigned
unsigned int x;             // 16-bit, unsigned
unsigned long w;           // 32-bit, unsigned

unsigned funct ( unsigned arg ){
    ...
}
```

Values in a 16-bit unsigned integer range from 0 to 65,535 instead of -32768 to +32767. Values in an unsigned long integer range from 0 to $2^{32}-1$.

useix—Indicates that a function uses the IX register as a stack frame pointer.

```
useix void func(){
    ...
}
```



See also **nouseix** and directives **#useix** and **#nouseix**.

waitfor—Used in a costatement, this keyword identifies a point of suspension pending the a condition, completion of an event, or a delay.

```
for(;;){
    costate {
        ...waitfor( input(1) == HIGH );
        ...
    }
    ...
}
```



See Chapter 7, Costatements.

while—Identifies the beginning of a while loop. A while loop tests at the beginning and may execute zero or more times.

```
while( expression ){
    ...
}
```

xdata—This keyword declares a block of data in extended memory.

There are two forms.

```
xdata name { value_1 , ... value_n };
```

```
xdata name [ n ];
```

The name of the block represents the 20-bit physical address of the block.

The value list of the first form may include constant expressions of type **int**, **float**, **uint**, **long**, **ulong**, **char**, and (quoted) strings.

xmem—Indicates that a function is to be placed in extended memory.

```
xmem int func() {  
    ...  
}
```

```
#memmap xmem
```

xmemok—Indicates that assembly-language code embedded in a C function can be compiled to extended memory.

```
#asm xmemok
```

```
...
```

```
#endasm
```

This keyword does not apply to C functions or to **#memmap**.

xstring—This keyword declares a table of strings in extended memory.

The table entries are 20-bit physical addresses (as unsigned long integers). The name of the table represents the 20-bit address of the table (as an unsigned long integer).

```
xstring name { string_1, ... string_n };
```

yield—Used in a costatement, this keyword causes the costatement to pause temporarily, allowing other costatements to execute. The yield statement does not alter program logic, but merely postpones it.

```
for(;;) {  
    costate {  
        ...  
        ...long computation  
        yield;  
        ...  
    }  
    ...  
}
```



See Chapter 7, Costatements.

Names

Names identify variables, certain constants, arrays, structures, unions, functions, and abstract data types. Names must begin with a letter or an underscore (`_`), and thereafter must be letters, digits, or an underscore. Names may not contain any other symbols, especially operators. Names are distinct up to 32 characters, but may be longer. Names may not be the same as any keyword. Names are case-sensitive.

Examples

```
my_function           // ok
_block                // ok
test32                // ok
jumper-              // not ok, uses a minus sign
3270type              // not ok, begins with digit
// The following two names are not distinct!
Clean_up_the_data_in_the_arrays_now
Clean_up_the_data_in_the_arrays_later
```

References to structure and union elements require “compound” names. The simple names in a compound name are joined with the dot operator (period).

```
cursor.loc.x = 10; // set structure element to 10
```

Use the `#define` directive to create names for constants. These can be viewed as symbolic constants. See *Macros*, previously discussed.

```
#define READ  010
#define WRITE 020
#define ABS   0
#define REL   1
#define READ_ABS  READ + ABS
#define READ_REL  READ + REL
```

The term `READ_ABS` is the same as `010 + 0` or `10`, and `READ_REL` is the same as `010 + 1` or `11`. Note that Dynamic C does not allow anything to be assigned to a constant expression.

```
READ_ABS = 27; // produces compiler error
              // because 010 + 1 is 10
```

Numbers

Numbers are constant values and are formed from digits, possibly a decimal point, and possibly the letters **U**, **L**, **X**, or **A–F**, or their lower case equivalents. A decimal point or the presence of the letter **E** or **F** indicates that a number is real (has a floating-point representation).

Integers have several forms of representation. The normal decimal form is most common.

10 -327 1000 0

An integer is long (32-bit) if its magnitude exceeds the 16-bit range (-32768 to +32767) or if it has the letter **L** appended.

0L -32L 45000 32767L

An integer is unsigned if it has the letter **U** appended. It is long if it also has **L** appended or if its magnitude exceeds the 16-bit range:

0U 4294967294U 32767U 1700UL

An integer is hexadecimal if preceded by **0x**.

0x7E 0xE000 0xFFFFFFFF

It may contain digits and the letters **a–f** or **A–F**.

An integer is octal if begins with zero and contains only the digits **0–7**.

0177 020000 00000630

A real number can be expressed in a variety of ways.

4.5	means 4.5
4f	means 4.0
0.3125	means 0.3125
456e-31	means 456×10^{-31}
0.3141592e1	means 3.141592

Strings and Character Data

A *string* is a group of characters enclosed in double quotes (“”).

“Press any key when ready...”

Strings in C have a terminating null byte appended by the compiler. Although the C language does not have a string data type, it does have character arrays that serve the purpose. Dynamic C does not have string operators, such as concatenate, but library functions are available.



See **STRING.LIB** in the *Dynamic C Function Reference* manual.

Strings are multi-byte objects, and as such they are always referenced by their starting address, and usually by a `char*` variable. The following example illustrates typical usage. Note that passing a pointer to a string is the same as passing the string.

```
char* select = "Select option\n";
char start[32];
strcpy(start, "Press any key when ready...\n");
printf( select );    // pass pointer to string
...
printf( start );    // pass string
```

Character constants have a slightly different meaning. They are not strings. A character constant is enclosed in single quotes (' ') and is a representation of an 8-bit integer value.

```
'a'    '\n'    '\x1B'
```

Any character can be represented by an alternate form, whether in a character constant or in a string. Thus, nonprinting characters and characters that cannot be typed may be used.

A character can be written using its numeric value preceded by a backslash.

```
\x41          // the hex value 41
\101          // the octal value 101
\B10000001   // the binary value 10000001
```

There are also several “special” forms preceded by a backslash.

```
\a bell           \b backspace
\f formfeed       \n newline
\r carriage return \t tab
\v vertical tab   \0 null char
\\ backslash      \c the actual character c
\' single quote   \" double quote
```

Examples

```
"He said \"Hello.\"" // embedded double quotes
char j = 'Z';        // character constant
char* MSG = "Put your disk in the A drive.\n";
// embedded newline at end
printf( MSG );      // print MSG
char* default = "";
// empty string: a single null byte
```

Operators

An operator is a symbol such as +, −, or & that expresses some kind of operation on data. Most operators are *binary*—they have two operands.

```
a + 10 // two operands with binary operator "add"
```

Some operators are *unary*—they have a single operand,

```
-amount // single operand with unary "minus"
```

although, like the minus sign, some unary operators can *also* be used for binary operations.

There are many kinds of operators with operator *precedence*. Precedence governs which operations are performed before other operations, when there is a choice.

For example, given the expression

```
a = b + c * 10;
```

will the + or the * be performed first? Since * has higher precedence than +, it will be performed first. The expression is equivalent to

```
a = b + (c * 10);
```

Parentheses can be used to force any order of evaluation. The expression

```
a = (b + c) * 10;
```

uses parentheses to circumvent the normal order of evaluation.

Associativity governs the execution order of operators of equal precedence. Again, parentheses can circumvent the normal associativity of operators.

For example,

```
a = b + c + d; // (b+c) performed first
a = b + (c + d); // now c+d is performed first
int *a(); // function returning ptr to int
int (*a)(); // ptr to function returning int
```

Unary operators and assignment operators associate from **right to left**.

Most other operators associate from left to right.

Certain operators, namely *, &, (), [], -> and . (dot), can be used on the left side of an assignment to construct what is called an *lvalue*. For example,

```
float x;
*((char*)&x) = 'L'; // x's LS byte gets value
*((char*)&x)+3 = 'H'; // x's MS byte gets value
```

When the data types for an operation are mixed, the resulting type is the more precise.

```
float x, y, z;
int i, j, k;
char c;

z = i / x;           // same as (float)i / x
j = k + c;          // same as k + (int)c
```

By placing a type name in parentheses in front of a variable, the program will perform type casting or type conversion. In the example above, the term `(float)i` means the “the value of `i` converted to floating point.”

The operators are summarized in the following pages.

- () Grouping. Expressions enclosed in parentheses are performed first. Parentheses also enclose function arguments. In the expression

```
a = (b + c) * 10;
```

the term `b + c` is evaluated first.

- [] Array subscripts or dimension.

```
int a[12];           // array dimension is 12
j = a[i];            // references the ith element
```

All array subscripts count from 0.

- .
- The dot operator joins structure (or union) names and subnames in a reference to a structure (or union) element.

```
struct {
    int x;
    int y;
} coord;

m = coord.x;
```

- > Right arrow. Used with pointers to structures and unions, instead of the dot operator.

```
typedef struct{
    int x;
    int y;
} coord;

coord *p;           // ptr to structure
...
m = p->x;           // ref to structure element
```

- ! Logical NOT. This is a unary operator. Observe that C does not provide a Boolean data type. In C, logical FALSE is equivalent to 0. Logical TRUE is equivalent to nonzero. The NOT operator result is 1 if the operand is 0. The result is 0 otherwise.

```
test = get_input(...);
if( !test ){
    ...
}
```

- ~ Bitwise complement. This is a unary operator. Bits in a **char**, **int**, or **long** value are inverted:

```
int switches;
switches = 0xFFF0;
j = ~switches;           // j becomes 0x000F
```

- ++ Pre- or post-increment. This is a unary operator designed primarily for convenience. If the ++ precedes an operand, the operand is incremented before use. If the ++ operator follows an operand, the operand is incremented after use.

```
int i, a[12];
i = 0;

q = a[i++];           // q gets a[0], then i becomes 1
r = a[i++];           // r gets a[1], then i becomes 2
s = ++i;              // i becomes 3, then s = i
i++;                  // i becomes 4
```

If the ++ operator is used with a pointer, the value of the pointer increments by the size of the object (in bytes) to which it points. With operands other than pointers, the value increments by 1.

- Pre- or post-decrement. This is a unary operator designed for convenience. If the -- precedes an operand, the operand is decremented before use. If the -- operator follows an operand, the operand is decremented after use.

```
int j, a[12];
j = 12;

q = a[-j];           // j becomes 11, then q gets a[11]
r = a[-j];           // j becomes 10, then r gets a[10]
s = j--;             // s = 10, then j becomes 9
j--;                 // j becomes 8
```

If the -- operator is used with a pointer, the value of the pointer decrements by the size of the object (in bytes) to which it points. With operands other than pointers, the value decrements by 1.

- + Unary plus, or binary addition. (Standard C does not have unary plus.) Unary plus does not really do anything.

```
a = b + 10.5;           // binary addition
z = +y;                 // just for emphasis!
```

- Unary minus, or binary subtraction.

```
a = b - 10.5;          // binary subtraction
z = -y;                 // z gets the negative of y
```

- * Indirection, or multiplication. As a unary operator, it indicates indirection. When used in a declaration, the * indicates that the following item is a pointer. When used as an indirection operator in an expression, the * provides the value at the address specified by a pointer.

```
int *p;                 // p is a pointer to integer
int j = 45;

p = &j;                 // p now points to j.
k = *p;                 // k gets the value to which p
                       // points, namely 45.
*p = 25;                // The integer to which p
                       // points gets 25. Same as j = 25,
                       // since p points to j.
```

Beware of using uninitialized pointers. Also, the indirection operator can be used in complex ways.

```
int *list[10]           // array of 10 ptrs to int
int (*list)[10]         // ptr to array of 10 ints

float** y;              // ptr to a ptr to a float
z = **y;                // z gets the value of y

typedef char **stp;
stp my_stuff;           // my_stuff is typed char**
```

As a binary operator, the * indicates multiplication.

```
a = b * c;              // a gets the product of b and c
```

- (*type*) “Cast” operator. The cast operator converts one data type to another. Floating-point values are truncated when converted to integer. The bit patterns of character and integer data are not changed with the cast operator, although high-order bits will be lost if the receiving value is not large enough to hold the converted value.

```
unsigned i; float x = 10.5; char c;

i = (unsigned)x;        // i gets 10;
c = *(char*)&x;         // c gets the low byte of x

typedef ... typeA;
typedef ... typeB;
```

```

typeA item1;
typeB item2;
...
item2 = (typeB)item1; // forces item1 to be
                    // treated as a typeB

```

- & Address operator, or bitwise AND. As a unary operator, this provides the address of a variable:

```

int x;
z = &x; // z gets the address of x

```

As a binary operator, this performs the bitwise AND of two integer (`char`, `int`, or `long`) values.

```

int i = 0xFFF0;
int j = 0x0FFF;
z = i & j; // z gets 0x0FF0

```

- sizeof**—The `sizeof` operator is a unary operator that returns the size (in bytes) of a variable, structure, array, or union. It operates at compile time as if it were a built-in function, taking an object or a type as a parameter.

```

typedef struct{
    int x;
    char y;
    float z;
} record;
record array[100];
int a, b, c, d;
char cc[] = "Fourscore and seven";
char *list[] = { "ABC", "DEFG", "HI" };
// number of bytes in array
#define array_size sizeof(record)*100
a = sizeof(record); // 7
b = array_size; // 700
c = sizeof(cc); // 20
d = sizeof(list); // 6

```

Why is `sizeof(list)` equal to 6? List is an array of 3 pointers (to `char`) and pointers have two bytes.

Why is `sizeof(cc)` equal to 20 and not 19? C strings have a terminating null byte appended by the compiler.

- / Divide. This is a binary operator. Integer division truncates; floating-point division does not.

```

int i = 18, j = 7, k; float x;
k = i / j; // result is 2;
x = (float)i / j; // result is 2.591...

```

% Modulus. This is a binary operator. The result is the remainder of the left-hand operand divided by the right-hand operand.

```
int i = 13;
j = i % 10;           // j gets i mod 10 or 3
int k = -11;
j = k % 7;           // j gets k mod 7 or -4
```

<< Shift left. This is a binary operator. The result is the value of the left operand shifted by the number of bits specified by the right operand:

```
int i = 0xF00F;
j = i << 4;           // j gets 0x00F0
```

The most significant bits of the operand are lost; the vacated bits become zero.

>> Shift right. This is a binary operator. The result is the value of the left operand shifted by the number of bits specified by the right operand:

```
unsigned int i = 0xF00F;
int j = 0xF00F;
k = i >> 4;           // k gets 0x0F00
k = j >> 4;           // k gets 0xFF00
```

The least significant bits of the left operand are lost; the most significant bits are either zeroed if the operation is unsigned, or copies of the sign bit if the operation is signed.

< Less than. This binary (relational) operator yields a “Boolean” value. The result is 1 if the left operand $<$ the right operand, and 0 otherwise.

```
if( i < j ){
    body                // executes if i < j
}
OK = a < b;           // true when a < b
```

<= Less than or equal. This binary (relational) operator yields a “Boolean” value. The result is 1 if the left operand \leq the right operand, and 0 otherwise.

```
if( i <= j ){
    body                // executes if i <= j
}
OK = a <= b;         // true when a <= b
```


- > Greater than. This binary (relational) operator yields a “Boolean” value. The result is 1 if the left operand > the right operand, and 0 otherwise.

```
if( i > j ){
    body // executes if i > j
}
OK = a > b; // true when a > b
```

- >= Greater than or equal. This binary (relational) operator yields a “Boolean” value. The result is 1 if the left operand ≥ the right operand, and 0 otherwise.

```
if( i >= j ){
    body // executes if i >= j
}
OK = a >= b; // true when a >= b
```

- == Equal. This binary (relational) operator yields a “Boolean” value. The result is 1 if the left operand equals the right operand, and 0 otherwise.

```
if( i == j ){
    body // executes if i = j
}
OK = a == b; // true when a = b
```



Note that the == operator is not the same as the assignment operator (=). A common mistake is to write

```
if( i = j ){
    body
}
```

Here, *i* gets the value of *j*, and the *if* condition is true when *i* is non-zero, *not* when *i* equals *j*.

- != Not equal. This binary (relational) operator yields a “Boolean” value. The result is 1 if the left operand ≠ the right operand, and 0 otherwise.

```
if( i != j ){
    body // executes if i != j
}
OK = a != b; // true when a != b
```

- ^ Bitwise exclusive OR. A binary operator, this performs the bitwise XOR of two integer (8-bit, 16-bit or 32-bit) values.

```
int i = 0xFFFF0;
int j = 0x0FFF;
z = i ^ j; // z gets 0xF00F
```

| Bitwise inclusive OR. A binary operator, this performs the bitwise OR of two integer (8-bit, 16-bit or 32-bit) values.

```
int i = 0xFF00;
int j = 0x0FF0;
z = i | j;           // z gets 0xFFFF0
```

&& Logical AND. This is a binary operator that performs the “Boolean” AND of two values. If either operand is 0, the result is 0 (FALSE). Otherwise, the result is 1 (TRUE).

|| Logical OR. This is a binary operator that performs the “Boolean” OR of two values. If either operand is nonzero, the result is 1 (TRUE). Otherwise, the result is 0 (FALSE).

? : Conditional operators. This is a three-part operation unique to the C language. It has three operands and the two operator symbols ? and :. If the first operand evaluates true (nonzero), then the result of the operation is the second operand. Otherwise, the result is the third operand.

```
int i, j, k;
...
i = j < k ? j : k;
```

The ? : operator is for convenience. The above statement is equivalent to the following.

```
if( j < k )
    i = j;
else
    i = k;
```

If the second and third operands are of different type, the result of this operation is returned at the higher precision.

= Assignment. This binary operator causes the value of the right operand to be assigned to the left operand. Assignments can be “cascaded” as shown in this example.

```
a = 10 * b + c;           // a gets the result of
                          // the calculation
a = b = 0;                // b gets 0 and a gets 0
```

+= Addition assignment.

```
a += 5;                  // Add 5 to a. Same as a = a + 5
```

-= Subtraction assignment.

```
a -= 5;                  // Subtract 5 from a.
                          // Same as a = a - 5
```

***=** Multiplication assignment.

```
a *= 5;           // Multiply a by 5.
                  // Same as a = a * 5
```

/= Division assignment.

```
a /= 5;           // Divide a by 5.
                  // Same as a = a / 5
```

%= Modulo assignment.

```
a %= 5;           // a mod 5. Same as a = a % 5
```

<<= Left shift assignment.

```
a <<= 5;          // Shift a left 5 bits.
                  // Same as a = a << 5
```

>>= Right shift assignment.

```
a >>= 5;          // Shift a right 5 bits.
                  // Same as a = a >> 5
```

&= Bitwise AND assignment.

```
a &= b;           // AND a with b.
                  // Same as a = a & b
```

^= Bitwise XOR assignment.

```
a ^= b;           // XOR a with b.
                  // Same as a = a ^ b
```

|= Bitwise OR assignment.

```
a |= b;           // OR a with b.
                  // Same as a = a | b
```

, Comma operator. This operator, unique to the C language, is a convenience. It takes two operands: the left operand—typically an expression—is evaluated, producing some effect, and then discarded. The right-hand expression is then evaluated and becomes the result of the operation.

This example shows somewhat complex initialization and stepping in a **for** statement.

```
for ( i=0, j=strlen(s)-1; i<j; i++, j- ) {
    ...
}
```

Because of the comma operator, the initialization has two parts: (1) set **i** to 0 and (2) get the length of string **s**. The stepping expression also has two parts: increment **i** and decrement **j**.

The comma operator exists to allow multiple expressions in loop- or **if** conditions.

Table 5-6 shows the operator precedence, from highest to lowest. All operators grouped together have equal precedence.

Table 5-6. Operator Precedence

Operators	Associativity
() [] -> . (dot)	left to right
! ~ ++ -- - (type) * & sizeof	right to left
* / %	left to right
+ -	left to right
<< >>	left to right
< <= > >=	left to right
== !=	left to right
&	left to right
^	left to right
	left to right
&&	left to right
	left to right
? :	right to left
= += -= etc...	right to left
, (comma)	left to right

Directives

Directives are special keywords prefixed with the symbol #. They tell the compiler how to proceed. Only one directive per line is allowed, but a directive may span more than one line if a backslash (\) is placed at the end of the line(s).

- **#asm** [*options...*]
#endasm

Begins and ends blocks of assembly code. The following options are available.

nodebug disable debug code during assembly

debug enable debug code during assembly

xmemok OK to compile to extended memory when assembly code is embedded in a C function

- **#class** [**push**] [*options...*]
#class pop

Controls the default storage class for local variables. The following options are available.

auto local variables are placed on the stack

static local variables have permanent, fixed storage

These options are nestable to 16 levels using the push and pop options.

- **#debug**
#nodebug

Enables or disables **debug** code compilation.

- **#define** *name text*
#define *name(params...) text*

Defines a macro with or without parameters. A macro without parameters may be considered a symbolic constant. (But in actuality it is not.)

- **#fatal** *"..."*
#error *"..."*
#warns *"..."*
#warnt *"..."*

Instructs the compiler to act as if a fatal error (**#fatal**), an error (**#error**), a serious warning (**#warns**) or a trivial warning (**#warnt**) was issued. The string in quotes following the directive is the message to be printed.

- **#funcchain** *chain-name name*

Adds a function, or another function chain, to a function chain.

- **#if** *constant_expression*
#elif *constant_expression*
#else
#endif

These directives control conditional compilation. Combined, they can form a multiple-choice **if**. When the condition of one of the choices is met, the Dynamic C code selected by the choice is compiled, whatever it may be. Code belonging to the other choices is ignored entirely.

```
main(){
    #if BOARD_TYPE == 1
        #define product "Ferrari"
    #elif BOARD_TYPE == 2
        #define product "Maserati"
```

```

#elif BOARD_TYPE == 3
    #define product "Lamborghini"
#else
    #define product "Chevy"
#endif
    ...
}

```

The **#elif** and **#else** directives are optional. Any code between an **#else** and an **#endif** is compiled when all of the expressions are false.

- **#ifdef** *name*
#ifndef *name*

Similar to the **#if** above, these directives enable and disable code compilation, respectively, based on whether the name has been defined with a **#define** directive.

- **#interleave**
#nointerleave

Controls whether Dynamic C will intersperse library functions with the program's functions during compilation. **#nointerleave** forces the user-written functions to be compiled first.

- **#KILL** *name*

To redefine a symbol found in the BIOS of a controller, first "kill" the prior name.

- **#makechain** *chain-name*

Creates a function chain. When a program executes the function chain named in this directive, all of the functions or segments belonging to that chain execute.

- **#memmap** [**push**] [*options...*]
#memmap pop

Controls the default memory area for functions. The following options are available.

anymem the compiler decides where to place functions
root functions in root memory
xmem functions in extended memory

These options are nestable to 16 levels using the push and pop options.

- **#undef** *name*

Removes (undefines) a defined macro.

- **#use** *libraryname*

Activates a library (named in **LIB.DIR**) so modules in the library can be linked with the application program. This directive immediately reads in all the headers in the library unless they have already been read.

- **#useix**
#nouseix

Controls whether functions use the IX register as a stack frame reference pointer or the SP (stack pointer) register.

Punctuation

Punctuation marks serve as boundaries in C programs. Table 5-7 lists the punctuation marks.

Table 5-7. Punctuation Marks

Symbols	Description
:	Terminates a statement label.
;	Terminates a simple statement (or a do loop). Required by C!
,	Separates items in a list, such as an argument list, declaration list, initialization list or expression list.
()	Encloses argument or parameter lists. Function calls always require parentheses. Macros with parameters also require parentheses.
{ }	Begins and ends a compound statement, a function body, a structure or union body, or encloses a function chain segment.

Extended Memory Data

Most of the details of calling extended memory functions are handled by the compiler. The situation is more complicated for extended data. To access extended memory data, use function calls to exchange data between extended memory and root memory. These functions are provided in the Dynamic C libraries.

An extended memory address is represented by an unsigned long integer which contains the 20-bit physical address. Pointers, on the other hand, are 16-bit machine addresses. They are not interchangeable. However, there are library functions to convert between these address formats.



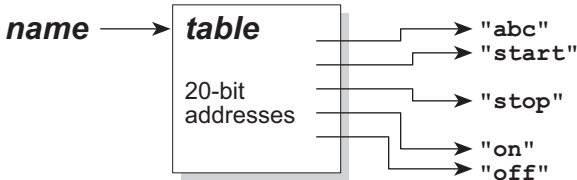
See **XMEM.LIB**.

Dynamic C includes two nonstandard keywords to support extended memory data: **xstring** and **xdata**.

The declaration

```
xstring name { string 1, ... string n };
```

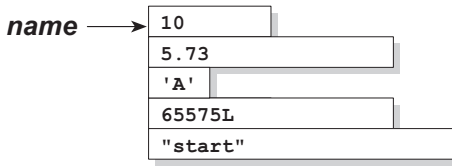
defines *name* as the extended memory address of a table of extended memory addresses (as **unsigned long** ints) and corresponding strings.



The **xdata** statement has two forms. The declaration

```
xdata name { value 1, ... value n };
```

defines a block of initialized extended memory data. The values must be constant expressions of type **char**, **int**, **unsigned int**, **long**, **unsigned long**, **float**, or **string**.

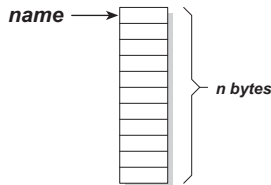


The other form

```
xdata name [ n ];
```

defines a block of *n* bytes in extended memory.

In either case, the term *name* is represented by an unsigned long integer containing the 20-bit physical address of the block.



See **XDATA.C** in the **SAMPLES** subdirectory for more details.



CHAPTER 6:
USING ASSEMBLY LANGUAGE

Dynamic C permits programing in assembly language. Assembly-language statements may either be embedded in a C function or entire functions may be written in assembly language. C statements may also be embedded in assembly code and refer to C-language variables in the assembly code.

A program may be debugged at the assembly language level by opening the assembly window. Single-stepping and break points are supported in the assembly window.

When the assembly window is open, single-stepping occurs instruction by instruction rather than statement by statement.

Use the **#asm** and **#endasm** directives to place assembly code in programs. For example, the following function will add two 64-bit numbers together.

```
useix int eightadd( char *ch1, char *ch2 ){
    #asm
        ld    l, (ix+ch2)           ; get dest ptr to hl
        ld    h, (ix+ch2+1)
        ld    e, (ix+ch1)           ; get src ptr to de
        ld    d, (ix+ch1+1)
        ld    b, 8                   ; number of bytes
        XOR   a                      ; clear carry
    loop:
        ld    a, (de)               ; ch1 source byte
        adc   a, (hl)               ; add ch2 byte
        ld    (hl), a               ; result to ch2 addr
        inc   hl
        inc   de
        djnz loop                   ; do 8 bytes
    #endasm
    return;
}
```

The same program could be written in C, but it would be many times slower because C does not provide an add-with-carry operation (**adc**).

A C statement may be placed within assembly code by placing a C in column 1.

The keyword **nodebug** can be placed on the same line as **#asm**. The main reason for the **nodebug** option is to prevent Dynamic C from running out of debugger table memory, which is limited to about 5,000 break points for the entire program (not counting libraries). If **nodebug** is specified for an entire function, then all the blocks of assembly code within the function are assembled in **nodebug** mode. There is no need to place the **nodebug** directive on each block.

Register Summary

Figure 6-1 shows the Z180's basic register set.

General Registers		Alternate Registers		Special Registers	
A	F	A'	F'	I	R
B	C	B'	C'	IX (index)	
D	E	D'	E'	IY (index)	
H	L	H'	L'	SP (stack pointer)	
				PC (program counter)	

Figure 6-1. Z180 Basic Register Set

Register A is the accumulator. Registers B–L are general-purpose registers and can be coupled in pairs BC, DE, HL for 16-bit values. Registers B, C, D, and E may also be coupled (and called BCDE) for 32-bit values. Register F (flags) holds status bits.

Flags

S	Z		H		P/V	N	C
7	6	5	4	3	2	1	0

S: sign bit Z: zero bit
H: half-carry P/V: parity or overflow
N: negative op C: carry

The alternate set of registers (A'–L') is often used to save and restore register values.



Refer to the Zilog *Z180 MPU User's Manual* for instructions to swap register sets.

The PC is the program counter; SP is the stack pointer. The IX and IY registers are index registers. The I register is the interrupt vector register. (The R register may be ignored.)

Dynamic C uses the HL register pair (1) to pass the first 16-bit argument, and (2) to return a 16-bit function result. Dynamic C uses the BCDE register group (1) to pass the first 32-bit argument and (2) to return a 32-bit function result.

The Z180 has many other special-purpose registers.

General Concepts

Place a body of assembly code between the **#asm** directives.

```
#asm [ options ]
#endasm
```

The **#asm** directive accepts options.



See *Directives* in Chapter 5, *The Language*, for details.

Comments

Comments in embedded assembly code starts with a semicolon (;). The assembler ignores all text from the semicolon to the end of line.

Labels

A label is a name followed by one or two colons (:). A label followed by a single colon is *local*, whereas one followed by two colons is *global*. A local label is not visible to the code out of the current embedded assembly segment (that is, code before the `#asm` or after the `#endasm` directive). Unless it is followed immediately by the keyword `equ`, the label identifies the current code segment address. If the label is followed by `equ`, the label “equates” to the value of the expression after the keyword `equ`.

Because C preprocessor macros are expanded in embedded assembly code, Z-World recommends that preprocessor macros be used instead of `equ` whenever possible.

Defining Constants

Constants may be created and defined in assembly code. The keyword `db` (“define byte”) places bytes at the current code segment address. The keyword `db` should be followed immediately by numerical values and strings separated by commas as shown here.

Example

Each of the following defines a string “ABC” in code space.

```
db 'A', 'B', 'C'  
db "ABC"  
db 0x41, 0x42, 0x43
```

The numerical values and characters in strings are used to initialize sequential byte locations.

The keyword `dw` defines 16-bit *words*, least significant byte first. The keyword `dw` should be followed immediately by numerical values, as shown in this example.

Example

The following defines three constants. The first two constants are literals, and the third constant is the address of variable `xyz`.

```
dw 0x0123, 0xFFFF, xyz
```

The numerical values initialize sequential word locations, starting at the current code segment address.

Expressions

The assembler parses most C-language constant expressions. A C-language constant expression is one whose value is known at compile time. All operators except the following are supported.

?:	(conditional)	[]	(array index,
.	(dot),	->	(points-to)
*	(dereference)	sizeof()	

For example, consider the following code.

```
#define FLAG1 1
#define FLAG2 4
#asm
...
and    ~(FLAG1 | FLAG2)
...
ld     de, FLAG1+0x80
...
#endasm
```

The preprocessor expands macros before the assembler parses any text.

Special Symbols

Table 6-1 lists special symbols that can be used in an assembly language expression.

Table 6-1. Special Assembly-Language Symbols

@PC	The symbol @PC evaluates to the current address in the PC (program counter) register.
@SP	The symbol @SP indicates the amount of stack space (in bytes) used for stack-based variables. This does not include arguments.
@RETVAl	The symbol @RETVAl evaluates to the offset, from the <i>frame reference point</i> to the stack space reserved for struct function returns.



See *Embedded Assembly Code* in Chapter 6, Using Assembly Language, for details on **@SP**.

See *C Functions Calling Assembly Code* in Chapter 6, Using Assembly Language, for details on **@RETVAl**.

The `@PC` symbol is useful when referring to an offset from the current `PC` address, as in the `riasmseq` assembly code multi-line macro definition.

```
// disable/restore interrupts assembly sequences
#define diasmseq    ld a,i $ push af $ di
#define riasmseq    pop af $ jmp novf,@PC+3 $ ei
```

Notice the `$` symbol in between the several assembly language statements on the multi-line assembly macro definition lines. Each `$` symbol denotes a new logical line of assembly code.

When the `diasmseq` and `riasmseq` macros are used as matched pairs in assembly code they save the current interrupt enable state, disable interrupts and then later restore the previous interrupt enable state. It is important, when making any stack references, to remember to take into account that these macros use the `AF` register pair pushed on the stack to preserve and then restore the interrupt enable state. The `jmp novf,@PC+3` instruction permits Dynamic C to make an absolute (NB: not PC-relative or position independent) jump in a macro definition where the jump address would not be unique (I.E.: if a label was included in the macro definition) at compile time.

C Variables

C variable names may be used in assembly language. What a variable name represents (the value associated with the name) depends on the variable. For a global, static local or register local variable, the name represents the *address* of the variable in root memory. For an `auto` variable or formal argument, the variable name represents its own *offset* from the frame reference point.



See *Embedded Assembly Code* in Chapter 6, *Using Assembly Language*, for details.

The name of a structure element represents the offset of the element from the beginning of the structure. In the following structure, for example,

```
struct s {
    int x;
    int y;
    int z;
};
```

the embedded assembly expression `s+x` evaluates to 0, `s+y` evaluates to 2, and `s+z` evaluates to 4, regardless of where structure `s` may be.

In nested structures, offsets can be composite, as shown here.

```
struct s {
    int x;           // s+x = 0
    struct a{       // s+a = 2
        int b;      // a+b = 0   s+a+b = 2
        int c;      // a+c = 2   s+a+c = 4
    }
};
```

Standalone Assembly Code

A standalone assembly function is one that is defined outside the context of a C-language function. It can have no **auto** variables and no formal parameters. Dynamic C always places a standalone assembly function in root memory.

When a program calls a function from C, it puts the first argument into a *primary register*. If the first argument has one or two bytes (**int**, **uint**, **char**, **pointer**), the primary register is HL (with register H containing the most significant byte). If the first argument has four bytes (**long**, **ulong**, **float**), the primary register is BCDE (with register B containing the most significant byte). Assembly-language code can use the first argument very efficiently. *Only* the first argument is put into the primary register, while *all* arguments—including the first—are pushed on the stack.

C function values return in the primary register, if they have four or fewer bytes, either in HL or BCDE.

Assembly language allows assumptions to be made about arguments passed on the stack, and “auto” variables can be defined by reserving locations on the stack for them. However, the offsets of such implicit arguments and variables must be kept track of. If a function expects arguments or needs to use stack-based variables, Z-World recommends using the embedded assembly techniques described in the next section.

Embedded Assembly Code

When embedded in a C function, assembly code can access arguments and local variables (either **auto** or **static**) by name. Furthermore, the assembly code does not need to manipulate the stack because the functions “prolog” and “epilog” already do so.

The concept and structure of a *stack frame* must be understood before correct embedded assembly code can be written. A stack frame is a run-time structure on the stack that provides the storage for all **auto** variables, function arguments and the return address.

Figure 6-2 shows the general appearance of a stack frame.

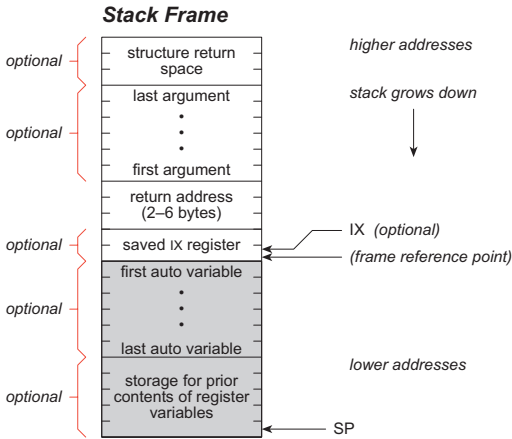


Figure 6-2. General Appearance of assembly Code Stack Frame

The return address is always necessary. The presence of auto variables and register variables depends on the definition of the function. The presence of arguments and structure return space depends on the function call. (The stack pointer may actually point lower than the indicated mark temporarily because of temporary information pushed on the stack.)

The shaded area in the stack frame is the stack storage allocated for **auto** and **register** variables. The assembler symbol `@SP` represents the size of this area. The meaning of this symbol will become apparent later.

The following sections describe how to access local variables in various types of functions.

No IX, Function in Root Memory

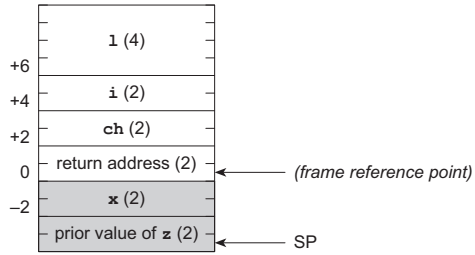
Assume this simple function has been called.

```

int gi;                // this is a global variable
root nouseix
void func( char ch, int i, long l ){
    auto    int x;
    static int y;
    register int z;
#asm
    some assembly code referencing gi, ch, i, l, x,
    y, and z
#endasm
}

```


Figure 6-3 shows how the stack frame will look.



**Figure 6-3. Assembly Language Stack Frame
No IX, Function in Root Memory**

The symbols for **gi**, **ch**, **i**, **l**, **x**, **y** and **z** will have the following values when used in the assembly code.

l	offset = +6	gi	a 16-bit address (in root memory)
i	offset = +4	x	offset = -2
ch	offset = +2	y, z	16-bit addresses (in root memory)

There is a common method to access the stack-based variables **l**, **i**, **ch** and **x**. Consider, for example, the case of loading variable **x** into HL.

The following code (using the symbol **@SP**) is one way to do it:

```
ld hl,@SP+x ; hl ← the offset from SP to the variable
add hl,sp   ; hl ← the address of the variable
ld a,(hl)  ; a ← the LSB of x
inc hl     ; hl now points to the MSB of x
ld h,(hl)  ; h ← the MSB of x
ld l,a     ; l ← the LSB of x
;; at this point, hl has the value of x
```

For static variables (**gi**, **y**, and **z**), the access is much simpler because the symbol evaluates to the address directly. The following code shows, for example, how to load variable **y** into HL.

```
ld hl,(y)   ; load hl with contents of y
```

Using IX, Function in Root Memory

Access to stack-based local variables is fairly inefficient. The efficiency improves if there is a register for a frame pointer. Dynamic C can use the register IX as a frame pointer. The function in the previous section would then become the following.

```
int gi; // this is a global variable
root useix
void func( char ch, int i, long l ){
    auto int x;
    static int y;
    register int z;
#asm
    some assembly code referencing gi, ch, i, l, x,
    y, and z
#endasm
}
```

The keyword **useix** is the only change from the previous sample function. Figure 6-4 shows the stack frame for this function.

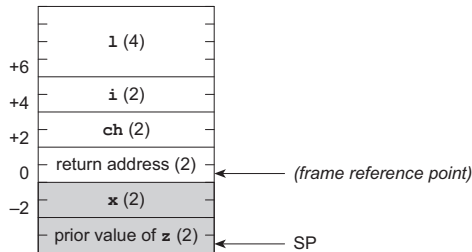


Figure 6-4. Assembly Language Stack Frame Using IX, Function in Root Memory

The arguments will have slightly different offsets because of the additional two bytes for the saved IX register value.

l	offset = +8
i	offset = +6
ch	offset = +4

Now, access to stack variables becomes easier. Consider, for example, how to load **ch** into register A.

```
ld a, (ix+ch) ; a ← ch
```

The IX+offset load instruction takes 14 cycles and three bytes. If the program needs to load a four-byte variable such as `l`, the IX+offset instructions are as follows.

```
ld e,(ix+1) ; load LSB of l
ld d,(ix+1+1) ;
ld c,(ix+1+2) ;
ld b,(ix+1+3) ; load MSB of l
```

This takes a total of 56 cycles and 12 bytes. Even if IX is the frame reference pointer, the `@SP` symbol may still be used.

```
ld hl,@SP+1 ; hl ← the offset from SP to the variable
add hl,sp ; hl ← the address of the variable
ld e,(hl) ; e ← the LSB of l
inc hl ;
ld d,(hl) ;
inc hl ;
ld c,(hl) ;
inc hl ;
ld b,(hl) ; b ← the MSB of l
```

This takes 52 cycles and 11 bytes. The two approaches are competitive. Nonetheless, the use of IX+offset is always beneficial when used to access single- or double-byte variables.

The offset from IX is a signed 8-bit integer. To use IX+offset, the variable must be within +127 or -128 bytes of the frame reference point. The `@SP` method is the only method for variables out of this range, even if IX is used as a frame reference pointer.

No IX, Function in Extended Memory

Functions that are (possibly) compiled to extended memory are not much different from functions compiled to root memory. Examine this extended memory function.

```
int gi; // this is a global variable
xmem
void func( char ch, int i, long l ){
    auto int x;
    static int y;
    register int z;
    #asm xmemok
        some assembly code referencing gi, ch, i, l, x,
        y, and z
    #endasm
}
```

If the `xmem` keyword is present, Dynamic C compiles the function to extended memory. Otherwise, Dynamic C can determine where to compile the function. On the other hand, the `xmemok` keyword must be present since this function is compiled to extended memory.

This is because functions compiled to extended memory have a 6-byte return address instead of a 2-byte return address. In this example, the IX register is not used. Figure 6-5 shows the stack frame of the function.

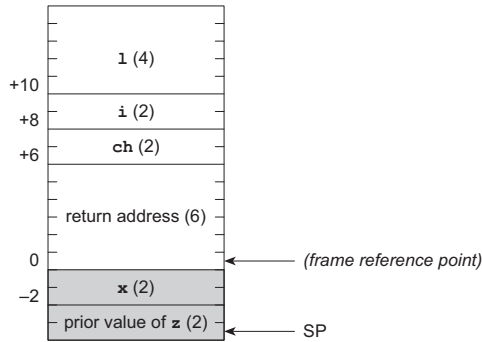


Figure 6-5. Assembly Language Stack Frame No IX, Function in ExtendedMemory

Because of the additional 4 bytes for the return address, the arguments will have slightly different offsets.

l	offset = +10
i	offset = +8
ch	offset = +6

Because the compiler maintains the offsets automatically, there is no need to worry about the change of offsets. The `@SP` approach discussed previously as a means of accessing stack-based variables works whether a function is compiled to extended memory or not, as long as the C-language names of local variables and arguments are used.

A function compiled to extended memory can use IX as a frame reference pointer as well. This adds an additional two bytes to argument offsets because of the saved IX value. Again, the IX+offset approach discussed previously can be used because the compiler maintains the offsets automatically.

C Functions Calling Assembly Code

Dynamic C does not assume that registers are preserved in function calls. In other words, the function being called need not save and restore registers. The exception is the memory management unit register CBR (common base register). If a function is in root memory and the caller is in extended memory, the compiler assumes that the CBR is preserved by the called function.

If a C-callable assembly function is expected to return a result (of primitive type), the function must pass the result in the “primary register.” If the result is an **int**, **uint**, **char** or a pointer, return the result in HL (register H contains the most significant byte). If the result is a **long**, **ulong** or **float**, return the result in BCDE (register B contains the most significant byte). A C function containing embedded assembly code may, of course, use a C **return** statement to return a value. A standalone assembly routine, however, must load the primary register with the return value before the **ret** instruction.

In contrast, if a function returns a structure (of any size), the calling function reserves space on the stack for the return value before pushing the last argument (if any). A C function containing embedded assembly code may use a C **return** statement to return a value. A standalone assembly routine, however, must store the return value in the structure return space on the stack before returning.

An in-line assembly code may access the stack area reserved for structure return values by the symbol **@RETVAl**, which is an offset from the frame reference point. The following code shows how to clear field **f1** of a structure (as a returned value) of type struct **s**.

```

typedef struct ss {
    int  f0;                // first field
    char f1;                // second field
} xyz;

xyz my_struct;
...
my_struct = func();
...
xyz func() {
    #asm
    ...
    xor a                    ; clear register A.
    ld hl,@SP+@RETVAl+ss+f1 ; hl ← the offset from
                           ; SP to the f1 field of
                           ; the returned structure.
    add hl,sp                ; hl now points to f1.
    ld (hl),a                ; load a (now 0) to f1.
    ...
    #endasm
}

```

It is crucial that **@SP** be added to **@RETVAl** because **@RETVAl** is an offset from the frame reference point, not from the current **SP**.

Assembly Code Calling C Functions

A program may call a C function from assembly code. To make this happen, set up part of the stack frame prior to the call and “unwind” the stack after the call. The procedure to set up the stack frame is described here.

1. Save all registers that the calling function wants to preserve. A called C function may change the value of any register. (Pushing registers values on the stack is a good way to save their values.)
2. If the function return is a **struct**, reserve space on the stack for the returned structure. Most functions do not return structures.
3. Compute and push the last argument, if any.
4. Compute and push the second to last argument, if any.
5. Continue to push arguments, if there are more.
6. Compute and push the first argument, if any. Also load the first argument into the primary register (HL for **int**, **uint**, **char** and pointers or BCDE for **long**, **ulong**, and **float**) if it is of a primitive type.
7. Issue the call instruction.

The caller must unwind the stack after the function returns.

1. Recover the stack storage allocated to arguments. With no more than 6 bytes of arguments, the program may pop data (2 bytes at time) from the stack. Otherwise, it is more efficient to compute a new **SP** instead. The following code demonstrates how to unwind arguments totaling 36 bytes of stack storage.

```
    ; note that HL is changed by this code!  
    ; Use ex de,hl to save HL if HL has the return value  
  
    ;;ex de,hl    ; save HL (if required)  
    ld hl,36     ; want to pop 36 bytes  
    add hl,sp    ; compute new SP value  
    ld sp,hl    ; put value back to SP  
    ;;ex de,hl    ; restore HL (if required)
```

2. If the function returns **struct**, unload the returned structure.
3. Restore registers previously saved. Pop them off if they were stored on the stack.
4. If the function return was not a **struct**, obtain the returned value from HL or BCDE.

Indirect Function Calls in Assembly

Indirect function calls are calls made to a function through a pointer to the function. The Z180 instruction set does not have an opcode for indirect function calls. However, they can still be done. The following code illustrates how.

```
    ; assume HL has the address of the called function
    ld de,retAddr ; explicitly load the return address
    push de      ; save the return address
    ...
    jp (hl) ; indirect jump to address specified by HL
    ...
retAddr:
    ; execution continues here when the function returns
```

If HL is supposed to contain an argument, use register IY or IX (if IX is not used as a frame reference pointer) instead of HL.

Interrupt Routines in Assembly

Dynamic C allows interrupt service routines to be written in C (declared with the keyword `interrupt`). However, the efficiency of one interrupt routine affects the latency of other interrupt routines. Assembly routines can be more efficient than the equivalent C functions, and therefore more suitable for interrupt service routines.

Either standalone assembly code or embedded assembly code may be used for interrupt routines. The benefit of embedding assembly code in a C-language interrupt routine is that there is no need to worry about saving and restoring registers or reenabling interrupts. The drawback is that the C interrupt function does save all registers, which takes some amount of time. A standalone assembly routine needs to save and restore only the registers it uses.

In general, an interrupt routine performs the following actions:

1. Turn off interrupts upon entry. (The Z180 does this automatically.)
2. Save all registers (that will be used) on the stack. Interrupt routines written in C save all registers on the stack automatically. Standalone assembly routines must push the registers explicitly.
3. Determine the cause of the interrupt. Some devices, such as the ASCII serial ports on the Z180, map multiple causes to the same interrupt vector. An interrupt handler must determine what actually caused the interrupt.

4. Remove the cause of the interrupt. For example, an ASCII serial port may cause an interrupt because it has received a byte. The interrupt routine would read the byte from the receive buffer.

If an interrupt has more than one possible cause, check for all the causes and remove all the causes at the same time.

5. When finished, restore registers saved on the stack. Naturally, this code must match the code that saved the registers. Interrupt routines written in C perform this automatically. Standalone assembly routines must pop the registers explicitly.
6. Reenable interrupts. Interrupts are disabled for the entire duration of the interrupt routine (unless they are enabled explicitly). The interrupt handler must reenable the interrupt so that other interrupts can get the attention of the CPU. Interrupt routines written in C reenable interrupts automatically when the function returns. Standalone assembly interrupt routines, however, must reenable the interrupt (using the instruction **ei**) explicitly.

The interrupts should be reenabled immediately before the return instructions **ret** or **reti**. If the interrupts are enabled earlier, the system can stack up the interrupts. This may or may not be acceptable because there is the potential to overflow the stack.

7. Return. The three types of interrupt returns are: **ret**, **reti** and **retn**.



Refer to Chapter 8, Interrupts, and to the Zilog *Z180 MPU User's Manual* to learn about their differences.

Common Problems

Unbalanced stack. Ensure the stack is “balanced” when a routine returns. In other words, the **SP** must be same on exit as it was on entry. From the caller’s point of view, the **SP** register must be identical before and after the call instruction.

Using the @SP approach after pushing temporary information on the stack. The @SP approach for in-line assembly code assumes that **SP** points to the low boundary of the stack frame. This might not be the case if the routine pushes temporary information onto the stack. The space taken by temporary information on the stack must be compensated for.

The following code illustrates the concept.

```
;SP still points to the low boundary of the call frame
push hl          ; save HL
;SP now two bytes below the stack frame!
...
ld hl,@SP+x+2   ; Add 2 to compensate for altered SP
add hl,sp        ; compute as normal
ld a,(hl)        ; get the content
...
pop hl           ; restore HL
;SP again points to the low boundary of the call frame
```

CBR not preserved. Dynamic C assumes that root functions preserve the CBR (common base register, for memory management). While most functions have nothing to do with the CBR, some functions in extended memory do manipulate the CBR. Make sure the CBR is preserved in a function in root memory.

Registers not preserved. In Dynamic C, the caller is responsible for saving and restoring all registers. An assembly routine that calls a C function must assume that all registers will be changed.

Unpreserved registers in interrupt routines cause unpredictable and unrepeatable problems. In contrast to normal functions, interrupt functions are responsible for saving and restoring all registers themselves.



CHAPTER 7: **COSTATEMENTS**

Dynamic C supports multi-threaded real-time programming. Either the real-time kernel (**RTK.LIB**) or the simplified real-time kernel (**SRTK.LIB**) may be used. **Costatements** are another option. Costatements offer *cooperative multi-tasking* within an application.

There are several advantages to costatements.

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

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

Overview

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. For example, the code shown in Figure 7-1 will operate as shown in the diagram.

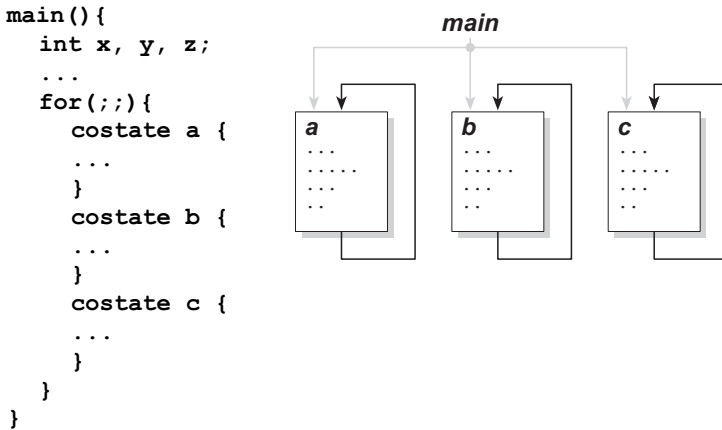


Figure 7-1. Overview of Costatements

Blocks **a**, **b**, and **c** (each of them costatements) will operate independently, concurrently, and with their own timing. The keyword **costate** identifies a costatement.

Using costatements presupposes that there will be more than one costatement. It is only when there is more than one task that costatements can be considered *cooperative*, because it is only when there is more than one task that any task can execute in the idle time of another task.

Nevertheless, some single tasks are easier to write using costatements. Costatements can be used, for example, to create delays.

A typical set of costatements will execute in an endless loop. However, this is not a requirement.

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

1. They can wait for an event, a condition, or the passage of a certain amount of time. The **waitfor** statement is used. Special functions are available to cover the passage of time: **DelaySec**, **DelayMS**, **DelayTicks**, **IntervalSec**, and **IntervalMS**.
2. They can use a **yield** statement to yield temporarily to other costatements.
3. They can use an **abort** statement to cancel their own operation.

Since costatements can suspend their own execution, they 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, other costatements suspend and others resume. Placing the costatements in a loop is the simplest way to give each costatement a chance to progress in its turn.

Costatements can be active (ON) or inactive (OFF). A costatement may be declared to be “always on,” “initially on,” or “initially off. A costatement that is *initially on* will execute once and then become inactive. A costatement that is *initially off* will not execute until it is started by some other part of the program. Then it will execute once and become inactive again.


For each costatement, there is a structure of type **CoData** that supports its operation. For example, the **CoData** structure maintains a position pointer that tells the costatement where to resume execution when it has been suspended.

Costatements may be named or unnamed. An unnamed costatement is “always on.” 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**.

A **CoData** structure may be declared independently of a costatement. Thus, many costatements can use a single **CoData** structure (one at a time). A single costatement may point to different **CoData** structures at different times.

All costatements in a 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`. Refer to the *Virtual Driver* in the *Dynamic C Function Reference* manual for more information.

Four functions, `CoBegin`, `CoResume`, `CoPause`, and `CoReset` are available to operate costatements remotely. Two functions, `isCoDone` and `isCoRunning`, return the state of a costatement.

A `firsttime` keyword is available to help create `waitfor` functions.

Syntax

The general format of a costatement appears below.

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

A costatement can have as many statements, including `abort` statements, `yield` statements, and `waitfors` as needed.

Name

The term *name*, which is optional, can be any 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. For the costatement to execute, it must be triggered by the software. Then the costatement will execute once and become inactive again.

Waitfor

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

```
waitfor ( expression );
```

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

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

The diagram on the left side of Figure 7-2 shows the execution thread the first time through a costatement when a **waitfor** evaluates false. The diagram on the right shows the execution thread through a costatement when a **waitfor** continues to evaluate false.

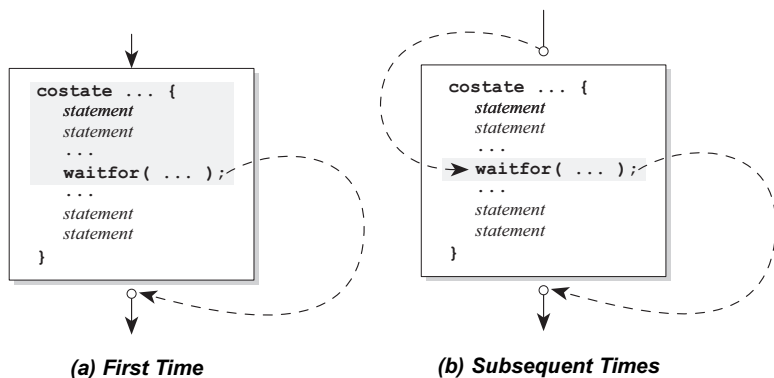


Figure 7-2. Execution of **waitfor** Statement

When the **waitfor** is encountered in a costatement for the first time, a *first time* flag associated with that the costatement is set. This flag is used by routines perform timing delays.

Figure 7-3 diagram shows the execution thread through a costatement when a **waitfor** finally evaluates true.

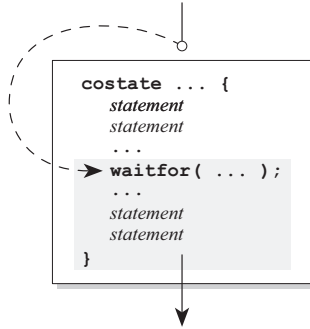


Figure 7-3. Execution of True waitfor Statement

Delay Functions

Three special functions (others may be created) allow the use of delays in the expression evaluated by a **waitfor**.

```
int DelaySec( ulong seconds );
int DelayMs(  ulong milliseconds );
int DelayTicks( uint ticks );

int IntervalSec( ulong seconds );
int IntervalMs(  ulong milliseconds );
```

Thus, an expression such as the following may be used.

```
// wait for 30 minutes
waitfor( DelaySec(30L*60L) );

// wait for device or 40 milliseconds
waitfor( DelayMs(40L) || device-ready() );
```



The virtual driver must be initialized with a call to **VdInit** before these delay functions can be used.



Refer to the *Dynamic C Function Reference* manual and the *Dynamic C Application Frameworks* manual for more details about the virtual driver and the delay functions.

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, as shown in Figure 7-4.

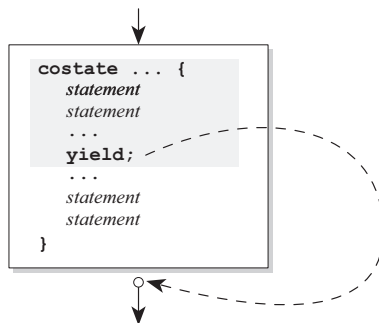


Figure 7-4. Unconditional Yield Exit from Costatement

The next time the program executes the costatement, it will resume at the statement following the **yield**, as shown in Figure 7-5. Compare this action with the description of the **abort** statement in the next section.

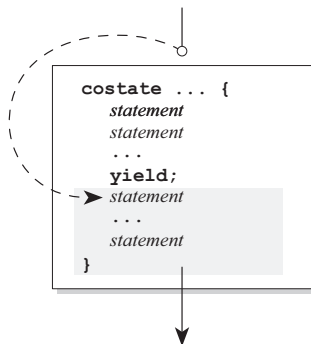


Figure 7-5. Resumption of Program after Yield

Example

Here is a loop containing two costatements.

```
while(1) {
    costate{
        for(i = 0; i < 30000; i++){
            some program code
            yield;
        }
    }
    costate{
        waitfor(DelayMs(500));
        printf("i = %d\n",i);
    }
}
```

Exactly one iteration of the `for` loop gets executed on each pass through the endless `while` loop. The second costatement checks whether 500 milliseconds have passed since the program first entered it. It will print the value of `i` if 500 milliseconds have passed.

The result is a loop that does two things concurrently. The code will output the value of `i` every half second, and the `for` loop increments `i` (and might do other things). The process will go on forever since both costatements are in an endless loop.

Abort

A costatement can terminate itself. For this purpose, there is the `abort` statement, which is permitted only inside a costatement.

```
abort;
```

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

Figure 7-6 illustrates the execution of the `abort` statement.

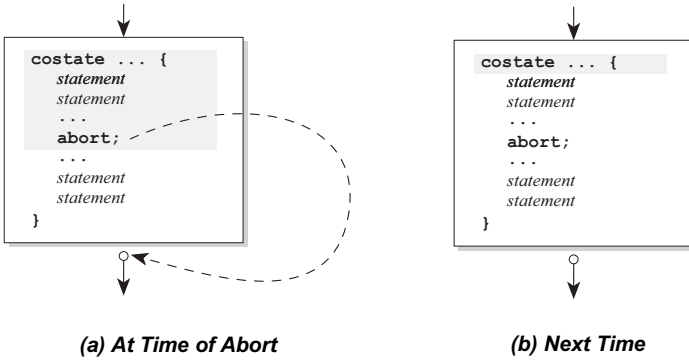


Figure 7-6. Execution of `Abort` Statement

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.



Use the functions provided to operate costatements. Do not use the fields of a `CoData` structure directly.

The structure `CoData` follows.

```
typedef struct {
    char CSState;
    uint lastlocADDR;
    char lastlocCBR;
    char ChkSum;
    char firsttime;
    union{
        ulong u1;
        struct {
            uint u1;
            uint u2;
        } us;
    } content;
    char ChkSum2;
} CoData;
```

Costatement State

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, as indicated in Table 7-1.

Table 7-1. Meanings of STOPPED and INIT Flags

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 your 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** flags. If **lastlocADDR** is non-zero, 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 checksum of the address. (It is the exclusive-or result of the bytes in **lastlocADDR** and **lastlocCBBR**.) If **ChkSum** is not consistent with the address, the program will generate a run-time error and reset. The checksum 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 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 the following.

```
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 visits to **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** above 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.

Advanced CoData Usage

Up to this point, the discussion has assumed that **CoData** structures are static and that there is one for each costatement.

A costatement is like a script. It specifies the sequence of operations to perform. The **CoData** data structure, on the other hand, is like an actor. It is responsible for “acting out” the script. With a static **CoData** structure for each costatement, there one “actor” for each “script.”

However, there are instances where multiple “actors” are needed for the same “script”. For example, if a factory has n identical machines, and there is a costatement to control the machines, a program with static **CoData** will look like the following program.

```
...
for(;;){
    costate{
        control sequence for machine 1
    }
    costate{
        control sequence for machine 2
    }
    ...
    costate{
        control sequence for machine n
    }
}
...
```

Although it is extremely simple, the above code is wasteful. A second approach is given below.

```
CoData Machine[ n ]; // an array of codata blocks
CoData ThisMachine; // one of the machines
int i;
...
for( i=0; i<n; i++){ // for all machines,
    CoBegin( &Machine[i] ); // enable machine
}
...
for(;;){ // endless loop
    for( i=0; i<n; i++){
        ThisMachine = Machine[i]; // get machine info
        costate ThisMachine always-on{
            Control sequence. Applies to any machine
        }
        Machine[i]=ThisMachine; // store it back
    }
}
...
```

This program is more space efficient than the one before it. It uses the same costatement for all the machines. However, the **CoData** structure must be copied from, and back to, the array because the **Machine** array is the actual storage for the states of each individual machine.

The following example offers another way to implement the same program.

```
CoData Machine[n]; // an array of codata blocks
CoData *pMachine; // ptr to a machine
uint i;
...
for( i=0; i<n; i++){ // for all machines,
    CoBegin(&Machine[i]); // enable machine
}
...
for(;;){
    for( i=0; i < n; i++){
        pMachine=&Machine[i];
        costate pMachine always-on {
            control sequence for all machines
        }
    }
}
}
```

In this approach, **pMachine** is a *pointer* to a **CoData** structure. Using pointers, there is no need to copy **CoData** structures before and after the costatement.



For further information, refer to the *Dynamic C Application Frameworks* manual.



It is never acceptable to have more than one costatement sharing a **CoData** (unless there is a guarantee they will not use the **CoData** at the same time, as in the second example above). The fields in **CoData** can control only one costatement at a time.



CHAPTER 8: **INTERRUPTS**

Dynamic C provides facilities for writing interrupt service routines (ISRs) in C and for setting up ISRs at compile time. Interrupt service routines may be written in assembly language.



See Chapter 6, Using Assembly Language.

A function that services interrupts must save and restore registers (including the memory management unit's CBR register). The keyword **interrupt** applies to a C function that services interrupts. All C-language ISRs save and restore registers.

Three additional keywords—**ret**, **reti**, or **retn**—can be used to select the return-from-interrupt instruction that will be performed. The following example shows an interrupt service routine in skeletal form.

```
interrupt reti iservice() {  
    EI();           // reenable interrupts (optional)  
    body of code...  
    return;       // optional at end of code  
}
```

When the above **return** is executed, the final two machine-level instructions after the registers have been restored are as follows.

```
ei    ; enable interrupts  
reti ; return from interrupt
```

If the **ret** keyword were to be used, then the final two instructions would be as follows.

```
ei    ; enable interrupts  
ret   ; return from interrupt
```

If the **retn** keyword were to be used, the final instruction would be as follows.

```
retn ; return from interrupt
```

No **ei** is necessary for **retn** since this instruction restores the previous state of the interrupts. If none of the keywords for the type of return is given, the default **ret** is assumed.

Dynamic C uses the **reti** instruction to return from an interrupt created by a Z180 peripheral. The **reti** instruction creates a particular type of bus cycle that the Z180 peripheral recognizes as acknowledging the completion of the interrupt service routine. The **ret** type of return can be used for interrupts created by devices not in the Z180 scheme, although it would not hurt to use **reti**. The only consideration would be the possibility of affecting devices in the Z180 family that might be part of the system, that is, accidentally sending the interrupt acknowledge signal to them before servicing the device's interrupt.

The **retn** instruction is used to return from a nonmaskable interrupt and it restores the interrupt state to the state prior to the nonmaskable interrupt.



More information on the Z180 interrupts can be found in the Zilog manuals.

If an interrupt routine is short, or cannot be interrupted, then interrupts can be left disabled throughout its execution. However, to keep interrupt latency (the amount of time that another interrupt request must wait before service) at a minimum, avoid disabling interrupts for long periods.

In addition, communication with the Dynamic C host system will be disrupted if interrupts are off for long periods, although the communication link can tolerate interrupts being off for approximately 0.5 seconds.

Two functions enable and disable interrupts.

```
void EI();           // enable interrupts
void DI();           // disable interrupts
```

The following function returns 1 if interrupts are enabled and 0 otherwise.

```
int iff();
```

The following functions read and set the 8-bit Z180 I register.

```
uint readireg();
void setireg( int value );
```

Normally, the I register points to a 256-byte vector table defined by the debugger startup code. If the location of the table changes, copy the interrupt vectors used by the debugger to the new area before modifying the I register.

Interrupt Vectors

There are two types of Z180 interrupt vectors. The first type, which handles modes 0, 1 and nonmaskable interrupts, requires that a jump instruction be inserted at the vector location because control is actually transferred to that location. This type includes the following vectors.

```
08h:   jp restart_service           ; mode 0 int
38h:   jp interrupt0_service        ; mode 1 int
66h:   jp nmi_service               ; nonmaskable
```

Use the following preprocessor directives to set the vectors at 38_H and 66_H.

```
#JUMP_VEC RST38_VEC function_name
#JUMP_VEC NMI_VEC function_name
```

The term **RST38_VEC** refers to the interrupt at 38_H and **NMI_VEC** refers to the interrupt at 66_H. Note that jump instructions are not usually stored at these locations because these locations are usually in the library EPROM area and cannot be changed. Instead, these locations jump to a relay vector in RAM which is actually modified.

The second type handles the Mode 2 interrupt used by Z180 peripheral devices, Z180 internal I/O devices and Dynamic C. This involves a 256-byte table, identified by the I register, that can contain addresses of up to 128 interrupt service routines.

Use the following preprocessor directive to set interrupt vectors in the page specified by the I register.

```
#INT_VEC ( const_expression ) function-name
```

The constant expression is the offset, in bytes, of the interrupt vector, which is always an even number from 0 to 126. The function name is the name of the interrupt service routine.



The vector table can be set with assignment expressions during Dynamic C development, but these assignments will not work when the code is in flash or burned into ROM. Always use the preprocessor directive, which is executed at compile time.

#INT_VEC expressions are processed as they are encountered during compilation. If a program specifies more than one location for a vector, the last one will be used. This can happen accidentally if, for example, an ISR is written for a device and then a library function that includes its own ISR for the same device is invoked. The library ISR will be used and the written ISR will be ignored, a situation that can be confusing.

Example

The following program illustrates the use of interrupt service routines written in Dynamic C.

```
int PRT1_init( int tc );    // initialize PRT1
#define TDE1 1             // PRT ch1 down-count enable
#define TIE1 5            // PRT ch1 interrupt enable
shared long counter;      // shared between different
                           // interrupt levels
#define ticks 2304        // (9.216MHz / 20) * .005 sec

main(){
    counter = 0L;          // initialize counter
    PRT1_init( ticks );   // 5 ms interrupts
    for(;;){
        if( counter >= 5000 ) break;
        outport( ENB485, !(counter & 64) );
    }
    IRES( TCR, TDE1 );    // disable count down
    IRES( TCR, TIE1 );    // disable interrupts
    printf( "Counter has reached 5000.\n" );
}

// this interrupt routine increments the "counter"
#INT_VEC PRT1_VEC ccc
interrupt reti ccc(){
    inport( TCR );
    inport( TMDR1L );     // clear TIF
    EI();
    counter = counter + 1;
}

int PRT1_init( int tc ){
    IRES( TCR, TDE1 );    // disable count down
    IRES( TCR, TIE1 );    // disable interrupts
    outport( TMDR1L, tc );
    outport( TMDR1H, tc >> 8 ); // set data reg
    outport( RLDR1H, tc >> 8 ); // set reload counter
    outport( RLDR1L, tc ); // set reload counter
    ISET( TCR, TDE1 );    // enable count down
    ISET( TCR, TIE1 );    // enable interrupts
    EI();
}
}
```

The interrupt routine `ccc` increments a counter every 5 milliseconds. The program prints a message and stops as soon as the counter reaches 5000.



CHAPTER 9: **REMOTE DOWNLOAD**

Z-World provides field programmability for its controllers. A downloadable program file can be created by selecting the appropriate compiler option. The Z-World Download Manager (DLM), resident in a controller, will receive the program, place it in memory, and start it running. Remote downloading requires a communications program such as ProComm that has an **XMODEM** transfer protocol available.

The downloaded program (DLP) and the DLM exist simultaneously as separate programs on the remote controller. They occupy different portions of memory. It is necessary for Dynamic C to know certain memory-mapping parameters about the DLM before it compiles the downloadable file.

Figure 9-1 shows how the DLM and DLP are arranged in memory. The download program can occupy both root and extended memory.

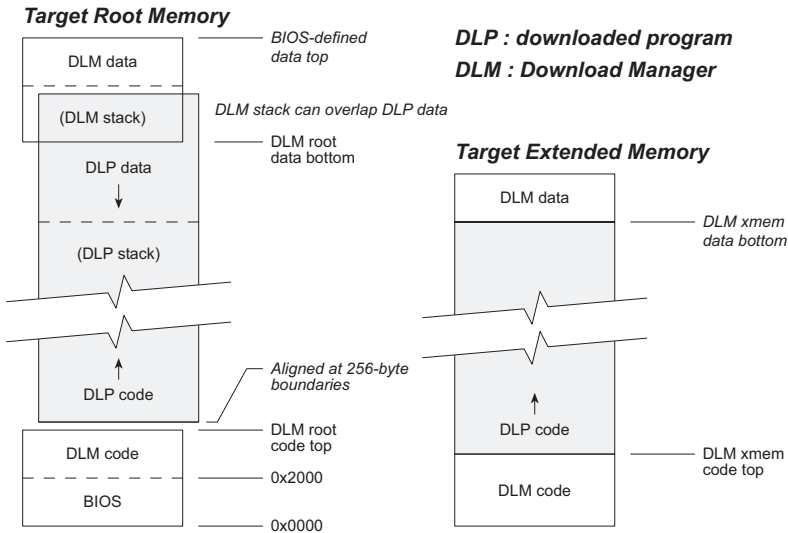


Figure 9-1. Arrangement of DLM and DLP in Memory

The DLM may run in RAM, EPROM, or flash. The file may be downloaded to flash or RAM.

The Z-World DLM uses memory-mapping information contained in the DLP to place the DLP machine code in the correct locations in target memory. The DLM does not download uninitialized data for the DLP. Initialized data reside in code space.

The Download Manager

The DLM is found in source code, for example, **DLM_Z0.C** in the **SAMPLES\AASC** subdirectory. The DLM may be modified in any way.

Once the DLM is installed on the target (compile it to flash or burn an EPROM), connect the target to the PC's serial port. A modem connection is acceptable. Start the communication program. Then, issue a break request (**ALT-B** in ProComm). The break request will cause the DLM to restart.

The Download Manager displays the following menu continuously:

Download Manager Menu

- 1) Enter Password**
- 2) Set Password**
- 3) Report DLM Parameters**
- 4) Download Program**
- 5) Execute Downloaded Program**
- 6) Hang-up Remote Modem**

Enter Choice #:

Enter Password

Choose **Enter Password** before enabling choices 2, 3 or 4.

Set Password

Choose **Set Password** (press **2**) to change the password. The DLM will prompt for a new password twice for verification. The DLM must allow password changes. See below.

Report DLM Parameters

This menu choice causes the DLM to report some memory-mapping parameters such as the following.

DLM Root Code Top	00:7579
DLM Root Data Bottom	E7:A7E5
DLM Xmem Code Top	0000D000
DLM Xmem Data Bottom	0008C000

Dynamic C requires these parameters to compile a downloadable program correctly. Dynamic C prompts for these values when a program is compiling to a DLP file. Actual values will differ.

Download Program

This menu choice initiates an **XMODEM** download on the target side. The upload must then be initiated on the PC side, using the communication program's **XMODEM** communication facilities. The file must be a downloadable program file created with Dynamic C. The DLM verifies the correctness of all data transmitted.

Execute Downloaded Program

This menu choice causes the DLM to shut down the interrupts it uses (but not the serial interrupt used for serial communication because that interrupt vector is shared by the DLP) and jump to the startup code of the DLP, from which it will not return. The DLM stores the CRC check sum for each segment and the number, size, and locations of downloaded segments. The DLM verifies the check sum for each segment before the DLP is invoked. The DLP will not run unless all CRCs (generally 3 or 4) are correct.

When the DLM is invoked again with another break request, it will start up at 0x2200, regardless of what else is running.

Hangup Remote Modem

This menu choice causes the DLM to issue standard modem hang-up and reset commands, and then jump to the DLP.

The DLM Code

Password security and timeout periods for the DLM can be controlled during compilation by changing the following macro definitions at the beginning of the DLM source code.

DLM_PASSWORD_STR defines the default password. If set to the null string (""), then just press **ENTER** after choosing **Enter Password** to gain entry.

DLM_PASSWORD_LVL Setting the password level to 0 enables the **Set Password** command to change the password at runtime.

DLM_MIN_PW_LEN Sets the minimum length of a valid password

DLM_MAX_PW_LEN Sets the maximum length of a valid password. The password must be between the minimum and the maximum values. The program will prompt for a new password twice for verification.

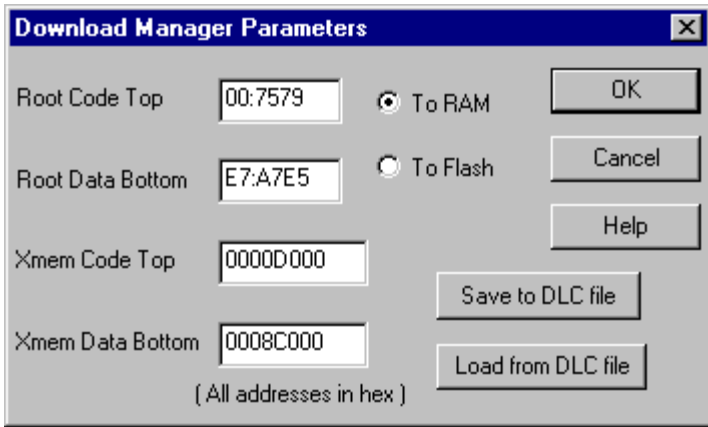
PSW_TIMEOUT Sets the number of milliseconds the DLM should wait for a password before jumping back to the DLP (if one is present).

DLM_TIMEOUT Sets the general timeout period for serial communications. If a serial communication function times out, the DLM returns control to the DLP (if one is present).

HANGUP_TIMEOUT Specifies the timeout period the DLM will allow for successful hangup of the modem when transferring control to the DLP. If the hangup operation is unsuccessful, the DLM still transfers machine control to the DLP.

The Downloaded Program (DLP)

To create a downloadable program, select **Download via DLM (.DLP)** in the **Compiler Options** dialog under the **OPTIONS** menu. Then issue the **Compile to File** command. Dynamic C will present the **Download Manager Parameters** dialog shown below.



Fill in the fields in this dialog to match those reported by the DLM when the DLM parameters (menu choice 3, described previously) were requested. The fields will be all zeros the first time the DLP is compiled. Thereafter, the field values will appear with the last values used.

Buttons in the dialog box allow the download parameters to be saved or retrieved to/from a download configuration (DLC) file. This is convenient to create DLPs for more than one type of controller. Be careful not to compile a DLP for a configuration different from the one on which it will actually run.

How to Use the DLM

Here is a step-by-step example of how to use the DLM to download and run a program. This example also demonstrates the use of an **.RTI** file for targetless compilation.

1. With a target controller connected to a PC, start Dynamic C, and open the file **DLM_Z0.C** in the **SAMPLES\AASC** subdirectory. Set **DLM_PASSWORD_STR** to the desired password, or to "" for no password.
2. Issue the **Create *.RTI File Targetless Compile** command to create an **.RTI** file for later use.
3. Compile **DLM_Z0.C** to the target.
4. Reset the target. Start the communication program. Enter the password when the **Download Manager Menu** appears.
5. Press 3 to display the DLM memory map. Jot down the numbers.
6. With Dynamic C open, open the program that is to be downloaded.
7. Select the **.DLP for download** compiler option.
8. Issue the **Compile to File with *.RTI File** command. Dynamic C will prompt for the name of the **.RTI** file to use. After that, the **Download Manager Parameters** dialog box will appear. Enter *exactly* the numbers from the DLM display in the corresponding fields on the dialog box. Click the OK button.

Assuming successful compilation, a download file will be created having the same name as the source file with a **.DLP** extension.

9. Press 4 in the DLM menu to initiate the download. Then initiate an **XMODEM** upload in the communication program. (Use the **Page Up** and **X** keys in ProComm.)
10. When the transfer is complete, the DLP is ready to run. Press 5 in the DLM menu to run it.
11. To terminate the DLP and return to the DLM, issue a break request in the communication program (**ALT-B** in ProComm). The Download Manager menu will reappear.

The DLP File Format

The DLP file created by Dynamic C has the following format.

1. A 128-byte header

Offset	Contents	Type
0	DLP Root Code Bottom	ulong
4	DLP Root Code Top	ulong
8	DLP Xmem Code Bottom	ulong
12	DLP Xmem Code Top	ulong
16	DLP Root Data Bottom	ulong
20	DLP Root Data Top	ulong
24	DLP Xmem Data Bottom	ulong
28	DLP Xmem Data Top	ulong
32	NUMSEG	uint
34–125	<i>Reserved</i>	—
126	CRC for this header	uint

2. Following the header, there are NUMSEG segments consisting of the following entries.

Offset	Contents	Type
0	Physical address for the segment body	ulong
4	SEGLEN (length of segment body)	ulong
8	The segment body	—
(SEGLEN+8)	CRC for segment and first 8 bytes	uint



CHAPTER 10: **LOCAL UPLOAD**

Z-World provides field programmability for its controllers. An uploadable to Flash EPROM (.BIN) or to SRAM (.BPF) program file can be created by selecting the appropriate compiler option. The Z-World Program Loader Utility (PLU), running on a PC connected via a COM port to the target controller, will transfer the program file into the target controller's memory, and optionally start it running immediately after upload. Local uploading requires a direct serial connection between the host PC and the target controller.

The uploadable program (.BIN or .BPF) should be of the appropriate type for the target controller. A **BIOS+Application (.BIN)** type program file is suitable for a Flash EPROM equipped target, while an **Application Only (.BPF)** type program file should only be uploaded to SRAM on a standard (non-Flash) EPROM equipped target controller. The PLU determines the target controller EPROM type and defaults to selection of only the appropriate program file type.

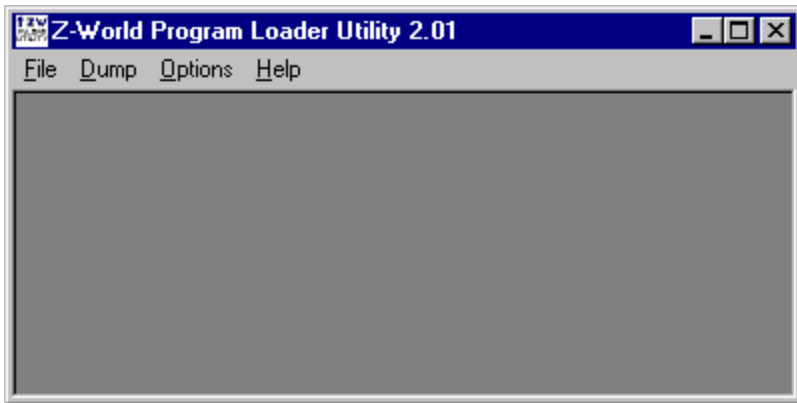


See *Appendix F, File Formats* for more information on these file types.

It is possible to override the default program type, however this is done at the sole discretion of the user. The reader is cautioned against such usage unless they have great confidence in their ability as well as complete and detailed knowledge of the target controller's workings. In addition, such nonstandard uses of the PLU are beyond the scope of this manual.

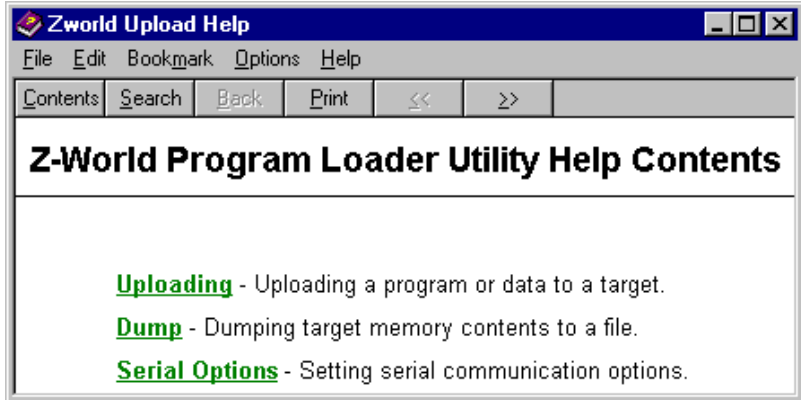
The Program Loader Utility

The Program Loader Utility (PLU) is found in the main Dynamic C 32 installation directory as **PRGLOADR.EXE**. In addition, the Dynamic C 32 installer places a shortcut icon on the Windows desktop as well as a Start bar Program Group entry. Start the PLU by any of the standard Windows methods (E.G.: double click on its icon).



On-line Help

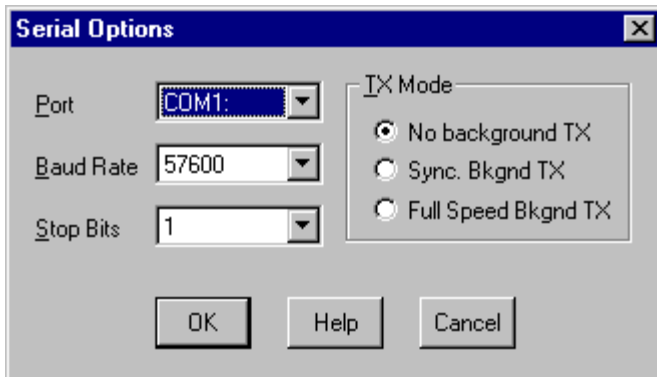
Select the **Help>Contents** menu item to display the following dialog box.



Click on an **Uploading**, **Dump**, or **Serial Options** link for information about that **Help** topic.

Set Communication Parameters

Select the **Options>Serial** menu item to display the following dialog box.



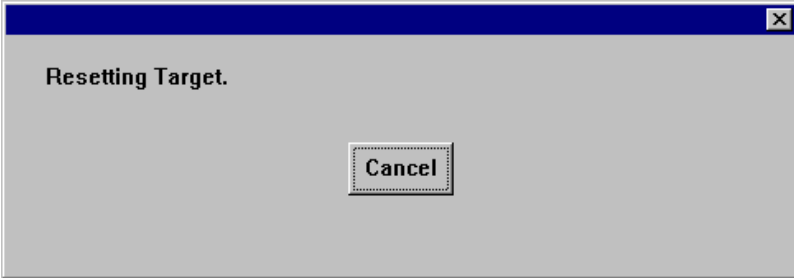
Select the appropriate COM port from the **Port** list box. Also set the serial **Baud Rate** and **Stop Bits** parameters, if necessary.

The **Tx Mode** group box provides three choices, but because the program files are pre-compiled by Dynamic C the choices are all equivalent. The PLU does not compile programs, and so can not overlap compilation and target communication.

Reset the Target Controller

The target controller must be reset into **program mode** in order to communicate with the PLU. Often, this entails setting a jumper or pressing keys, and either cycling power or pressing a reset button. Consult the specific controller's hardware manual for details.

When the target controller is in **program mode**, select the **Options>Reset Target** menu item to perform a **software reset** of the target controller. The following message is briefly displayed.



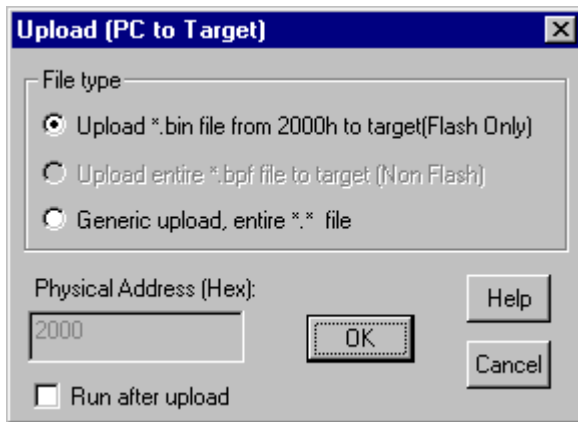
If the target controller is communicating with the PLU then the **Resetting Target** message will disappear. If you see the following message box then the target controller is not communicating with the PLU.



The problem may be that the communication parameters need to be changed (either in the PLU or the controller), or that the target controller is not running in program mode. Click the OK button, check the controller's power supply and jumper settings, check the PLU's communication parameters, and try again.

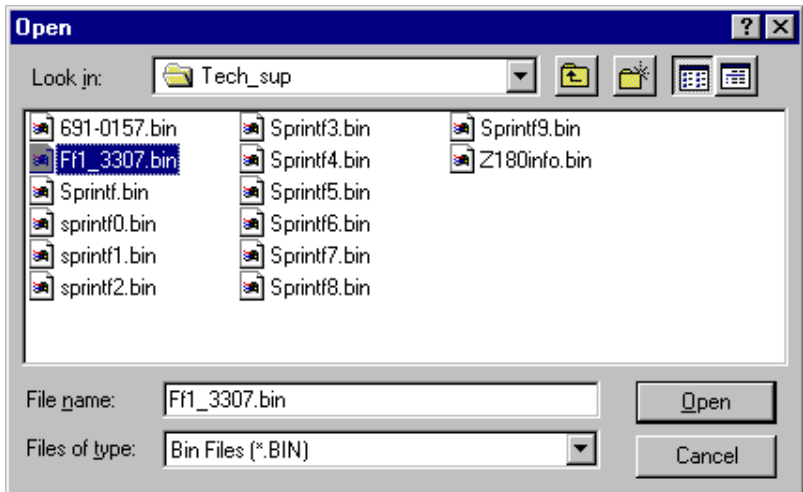
Select the Program File

Click on the **File>Upload** menu item to display the following dialog box.



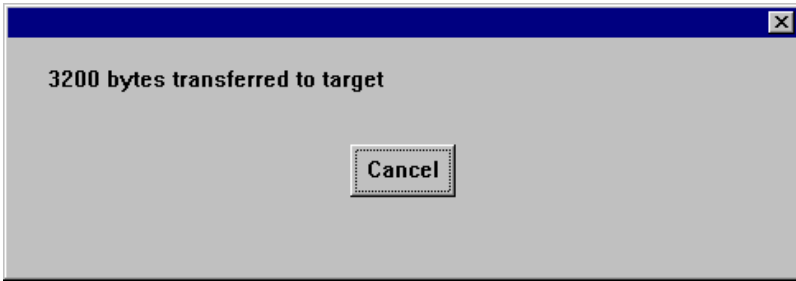
The PLU defaults to the appropriate program file **Upload . . .** type for the attached target controller. Subject to the **cautions and disclaimers** expressed previously, click on **Generic upload, entire *.* file** to override the program file type. The **Physical Address (Hex):** edit box is enabled, allowing unrestricted upload directly to any physical address.

Select the **Run after upload** checkbox to optionally start the uploaded program immediately after uploading it. Click the **OK** button to proceed to the file selection dialog.



Browse to find and select the uploadable program file, which must be located on a local drive or network drive accessible to the PC that is running the PLU.

Click the **Open** button to start uploading the selected program file. A message box similar to this one displays the upload progress.



When the progress message box disappears, the PLU has finished uploading the program file to the target controller. If the **Run after upload** checkbox was enabled before the upload, then the controller should begin to run the newly uploaded application.

The controller should be returned to **run mode** in order to assure that the program will resume execution after the next power interruption. As mentioned before, this often entails setting a jumper or pressing keys, and either cycling power or pressing a reset button. Consult the specific controller's hardware manual for details.

Common Problems

Reuse of programming port. An application program intended for upload via the PLU which reuses a serial port that is used by the PLU for communication with the target controller raises special concerns. In particular, if the PLU communicates with the target controller via Z0 or a PLCBus UART board, the application must use the run-time **reload_vec** function to set the Z0 or /INT1 (respectively) interrupt service vector, instead of the **#INT_VEC** compile-time directive.

Manual software reset. Although the PLU attempts a **software reset** of the target controller at start-up, it does not automatically attempt a software reset before every upload. When using the PLU to update the program on a series of target controllers it is recommended to select the **Options>Reset Target** menu item before each upload. This will help to ensure that the PLU is communicating properly with each target controller in turn.



APPENDIX A:
RUN-TIME ERROR PROCESSING

Compiled code generated by Dynamic C calls an error-handling routine for abnormal situations. The error handler supplied with Dynamic C prints any error messages to the **STDIO** window. When software runs stand-alone (disconnected from Dynamic C), such an error message will *hang* while waiting for a response from the PC being used to development the program or program the controller. Be sure to provide for an error handler unless there is a certainty that there will never be any run-time errors.

Your program calls the error handler *indirectly* through the global function pointer **ERROR_EXIT**. The following example shows the use of the standard error handler.

```
main() {
    ... // ERROR_EXIT is a pointer
    (*ERROR_EXIT) (50,0); // to the standard handler
} // or to your own
```

In this example, the standard Dynamic C error handler would send the message “**Run Time Error 50**” to the **STDIO** window. The first argument is the error number. The second argument specifies the address at which the error occurred.

The following example illustrates the use of a custom error-handling function that can take the place of the standard error handler:

```
void my_handler( uint code, uint address ) {
    error processing code...
}

main() {
    ...
    ERROR_EXIT = my_handler; // substitute my handler
    some statements...
    (*ERROR_EXIT) ( code, addr ); // call my own handler
    some statements...
}
```

A built-in Dynamic C symbol—**ROM**—is set to 1 if the compilation is to an EPROM file. Use this variable to conditionally install a custom error handler such as the one below.

```
#if ROM
    ERROR_EXIT = user_error_handler;
#endif
```

Table A-1 lists the ranges of Dynamic C error codes.

Table A-1. Ranges of Dynamic C Error Codes

Code	Meaning
0 – 99	User, nonfatal. For example, 49 = overflow from <code>pow10</code> .
100 – 127	System, nonfatal
128 – 227	User, fatal, no return possible
228 – 255	System, fatal, no return possible

Table A-2 lists the fatal errors generated by Dynamic C.

Table A-2. Dynamic C Fatal Errors

Code	Meaning
228	Pointer store out of bounds
229	Array index out of bounds
230	Stack corrupted
231	Stack overflow
232	Aux stack overflow
233	<i>Not used</i>
234	Domain error (e.g., <code>acos(2)</code>)
235	Range error (e.g., <code>tan(pi/2)</code>)
236	Floating point overflow
237	Long divide by zero
238	Long modulus, modulus zero
239	Subtraction overflow
240	Integer divide by zero
241	Unexpected interrupt
242	Execute outside program bounds (RST 38)

The standard error routine reports only fatal errors.

Long Jumps

Error recovery is performed using Dynamic C's `setjmp` and `longjmp` functions. If an error is detected anywhere in a program, a “long jump” can be made to a safe location so that the necessary recovery tasks can be performed. Typically a jump is made from a deeply nested function back to the main program.

The `setjmp` function marks a place in the code and saves the stack pointer and important registers. The `longjmp` function causes a return to the place marked by the `setjmp` call. The processor stack is immediately “unwound” and a known state is restored. This example shows how to do this.

```
// probably in main()
jmp_buf savreg;      // you must make a save buffer
...
if( setjmp(savreg) ){
    code to recover from the error
}
...
// then, somewhere, deeper in your code...
if( big error ) longjmp(savreg,1);
```

When `longjmp` is executed, the execution resumes immediately after the call to `setjmp`, and the value returned by the call to `setjmp` is the same as the second argument passed to `longjmp`. This value can be the error code as long as it is nonzero. (The return value of `setjmp` is 0 when it is called directly.)

Call `longjmp` in the same function as the call to `setjmp` or in a function called directly or indirectly from that function. (The `main` function is always a safe place to put `setjmp`.)

A “long jump” restores the SP, IX, and PC registers and also restores the auxiliary stack pointer.

Watchdog Timer

Most Z-World controllers have a watchdog timer. The watchdog timer is used to ensure that software does not get stuck. Even error-free software is susceptible to transient problems such as power surges, power outages, and dropped bits.

A watchdog timer will reset the system after a certain period (typically about 1.6 seconds) if the software does not reset the watchdog timer within that period. This safety feature helps to ensure that the program continues to function.

The function call

```
hitwd() ;
```

resets the watchdog timer. A program must call **hitwd** at least at the frequency of the watchdog timer (about once per second) no matter what else it is doing.



Although the watchdog timer can be disabled on some, but not all, Z-World controllers, Z-World does not recommend disabling the watchdog timer.

Protected Variables

A program may need to recover protected variables at when it restarts. However, if the program has never run before, it must initialize the protected variables.

The function **_prot_recover** recovers protected variables; the function **_prot_init** initializes them. The function **_sysIsSuperReset** calls the appropriate protected variable function.



See Appendix G, Reset Functions, for more information.



APPENDIX B: EFFICIENCY

There are a number of methods that can be used to reduce the size of a program, or to increase its speed.

Nodebug Keyword

Dynamic C places an **RST 28H** instruction in debug code at the beginning of each C statement to provide locations for break points. These “jumps” to the debugger consume one byte and about 25 clocks of execution time for each statement. A function will not have **RST 28H** instructions if the **nodebug** keyword is used in the function declaration.

```
nodebug int myfunc( int x, int z ){  
    ...  
}
```

Once a function is **nodebug**, it is no longer possible to single-step into the function or set a break point in the function, except when the assembly window is active. (It is possible to single-step through any assembly code.) The **nodebug** keyword also reduces entry and exit bookkeeping for the function and turns off all checking for array bounds, stack corruption and pointer stores.

If the **nodebug** option is used for the **main** function, the program will begin to execute as soon as it finishes compiling (as long as the program is not compiling to a file).

Use the **nodebug** keyword with the **#asm** directive.



See *Chapter 5: Using Assembly Language*, for more information.

Use the directive **#nodebug** anywhere within the program to enable **nodebug** for all statements following the directive. The **#debug** directive has the opposite effect.

Static Variables

Static variables are much more efficient on the Z180 than **auto** variables. In Dynamic C, the default local storage class is **static**, while most C compilers use **auto**. Use **auto** variables in reentrant or recursive functions.

Here are some rules concerning declarations that will help to conserve code and save time.

1. Use global variables for global communication or constants.

2. Avoid **auto** variables within functions whenever possible. To save code space and execution time while preserving reentrancy, use the **register** storage class for one- or two-byte items.
3. The **shared** and the **protected** keywords in data declarations cause slower fetches and stores, except for one-byte items and some two-byte items.
4. When there are more than 128 bytes of auto variables declared in a function, keep in mind that the first 128 bytes are more easily accessed than later declarations, owing to the limited 8-bit range of Z180 IX register addressing.

Execution Speed

Compiler Options can be used to set a switch to optimize for speed or for size. The default is size. If speed is selected, then the program size might increase somewhat. Using **static** variables with **nodebug** functions will increase program speed greatly. Stack checking must be disabled for good speed.

Subfunctions

Subfunctions, extensions in Dynamic C, allow often-used code sequences to be turned into a “subroutine” within the scope of a C function.

```
func () {  
    int aname();  
    subfunc aname: { k = inport (x); k + 4; }  
    ...  
    ... aname(); ...  
    ...  
    ... aname(); ...  
    ...  
}
```

The subfunction is prototyped as if it were a regular function. It must be **static** and may not have any arguments. Variables used within the subfunction must be available within the scope of the parent C function. The actual code after the **subfunc** keyword can appear anywhere in the enclosing function. The return value, if any, is indicated by placing an expression followed by a semicolon at the end of the subfunction body. This causes the expression value to be loaded into the primary register (**HL** or **BCDE**).

All subfunction calls take three bytes, low overhead compared to some simple expressions. For example, the expression `*ptr++` can generate 14 bytes or more. Substitute the following code.

```
static char nextbyte();
subfunc nextbyte: *ptr++;
nextbyte();
...
nextbyte();
...
```

This can save ten or more bytes each time `nextbyte` occurs.

Subfunctions can also make a program easier to read and understand if descriptive names are used for obscure expressions. The advantage of the subfunction over a regular function is that it has access to all the variables within the program and the calling overhead is low.

Observe that the equivalent C function

```
nextbyte( char *ptr ) { return *ptr++; }
```

can be used for the same purpose. However, the calling overhead is much greater, a minimum of eight bytes, and at least eleven bytes if `ptr` is an `auto` variable.

Subfunction calls cannot be nested.

Function Entry and Exit

The following events occur when a program enters a function.:

1. Save `IX` on the stack and make `IX` the stack frame reference pointer (if in `useix` mode).
2. Create stack space for `auto` variables or to save `register` variables.
3. Set up stack corruption checks if stack checking is on.
4. Notify Dynamic C of the entry to the function so that single-stepping modes can be resolved (if in debug mode).

Items three and four consume significant execution time and are eliminated when stack checking is disabled or if the debug mode is off.

Disable stack checking if speed is needed during debugging. In general, avoid using `auto` variables, except when a function must be reentrant. Using the `IX` register as a frame reference pointer (`useix` and `#useix` options) results in faster and more compact access to arguments and `auto` variables, especially for `char` variables. The `useix` option is especially valuable when embedding assembly language inside a C program. In this case it is easiest to access the variables using the `ix` register. Use `nouseix` only for functions that can suspend under the real-time kernel.



APPENDIX C: SOFTWARE LIBRARIES

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

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

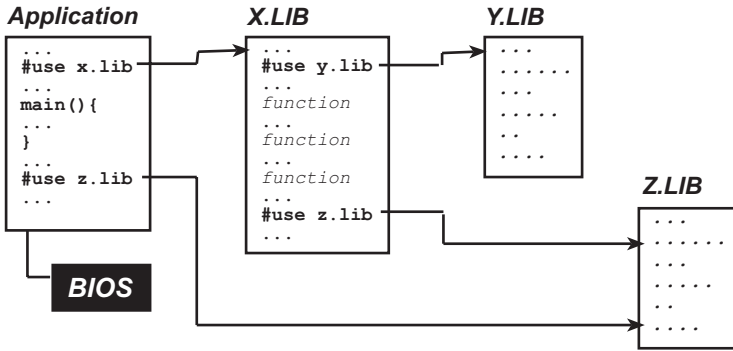


Figure C-1. Linking Nestable Files in Dynamic C

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

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

Headers

Table C-1 describes the three kinds of headers in Dynamic C libraries.

Table C-1. Dynamic C Library Headers

Library Headers	Describe libraries. Library headers should tell a programmer how to use the library.
Function Headers	Describe functions. Function headers form the basis for function lookup help.
Module Headers	Makes functions and global variables in the library known to Dynamic C

Users who develop their own libraries are encouraged to include descriptive headers for the library and all of its functions. In particular, accurate and correctly formatted headers must be defined for function help to work with functions.

Library Headers

A library has a single header at the beginning that describes the nature of the library. The header is a specially formatted comment, such as the following example.

```
/* START LIBRARY DESCRIPTION *****  
DRIVERS.LIB  
  
    Copyright (c) 1994, Z-World.  
DESCRIPTION: Miscellaneous hardware drivers li-  
brary. Many of these routines disable inter-  
rupts for short periods. Define NODISINT to  
prevent this.  
SUPPORT LIBRARIES:  
END DESCRIPTION *****/
```

Function Headers

Each function in a Z-World library has a descriptive header preceding the function to describe the function. The header is a specially formatted comment, such as the following example.

```
/* START FUNCTION DESCRIPTION *****
plcport                                <DRIVERS.LIB>
SYNTAX: int plcport(int bit);
KEY WORDS:
DESCRIPTION: Checks the specified bit of the PLC
             bus port.
RETURN VALUE: 1, if specified bit is set, else
             zero.
END DESCRIPTION *****/
```

Function headers are extracted by Dynamic C to provide on-line help messages.

Modules

A library file contains a group of *modules*. A module has three parts: the key, the header, and a body of code (functions and data).

A module in a library has a structure like this one.

```
/** BeginHeader func1, var2, .... */
    prototype for func1
    declaration for var2
/** EndHeader */
    definition of func1, var2 and possibly other
    functions and data
```

The Key

The line (a specially-formatted comment)

```
/** BeginHeader name1, name2, .... */
```

begins the header of a module and contains the *key* of a module. The *key* is a list of names (of functions and data). The key tells the compiler what functions and data in the module are available for reference. It is important to format this comment properly. Otherwise, Dynamic C cannot identify the module correctly.

If there are many names after **BeginHeader**, the list of names can continue on subsequent lines. All names must be separated by commas.

The Header

Every line between the comments containing **BeginHeader** and **EndHeader** belongs to the *header* of the module. When an application **#uses** a library, Dynamic C compiles every header, and just the headers, in the library. The purpose of a header is to make certain names defined in a module known to the application. With proper function prototypes and variable declarations, a module header ensures proper type checking throughout the entire application program.

The Body

Every line of code after the **EndHeader** comment belongs to the *body* of the module until (1) end-of-file or (2) the **BeginHeader** comment of another module. Dynamic C compiles the *entire* body of a module if *any* of the names in the key are referenced (used) anywhere in the application.

An Example Module

Notice in the following (extremely contrived) example that the header contains only function and variable declarations, not definitions. Any function or variable which is actually defined in a header will be compiled into every application that **#uses** the library containing the header, whether or not the specific function or variable is ever referenced.

```
/** BeginHeader foo, bar, foobar */
struct foo { int i; int j }; // declaration!
extern float bar; // declaration!
int foobar(int b, struct foo *pf); // prototype!
/** EndHeader */

float bar; // variable definition here!

// function definition follows!
nodebug int foobar(int b, struct foo *pf) {
    bar = b + (float) (pf->i) / (float) (pf->j);
    return((bar - b) >= 0.5);
}
```

To minimize waste, Z-World recommends that a module header contain only macros, prototypes, **extern** variable declarations or other declarations that do not directly generate code or data. Define code and data only in the body of a module. That way, the compiler will generate code or allocate data *only* if the module body is used by the application program.

Programmers who create their own libraries must write modules following the guideline in this section. Remember that a library must be included in **LIB.DIR** and a **#use** directive for the library has to be placed somewhere in the code.



*APPENDIX D: **EXTENDED MEMORY***

Physical Memory

Depending on PAL coding and board jumper settings, Z-World controllers can address up to 512K of ROM or 256K of flash memory, and 512K of RAM. The maximum memory available is 1 megabyte total.

Usually, memory chips installed on Z-World controllers have a capacity less than 512K. A typical SRAM chip has 32K or 128K.



If a memory chip has less than 512K, addresses outside the memory range map to addresses within the range. For example, for a 32K chip, addresses evaluate modulo 32K. If memory is addressed beyond the range of the chip, data may seem to be replicated in memory. Or worse, data may be overwritten.

Memory Management

Z180 instructions can specify 16-bit addresses, giving a *logical* address space of 64K (65,536 bytes). Dynamic C supports a 1-megabyte *physical* address space (20-bit addresses).

An on-chip memory management unit (MMU) translates 16-bit Z180 addresses to 20-bit memory addresses. Three MMU registers (CBAR, CBR, and BBR) divide the logical space into three sections and map each section onto physical memory, as shown in Figure D-1.

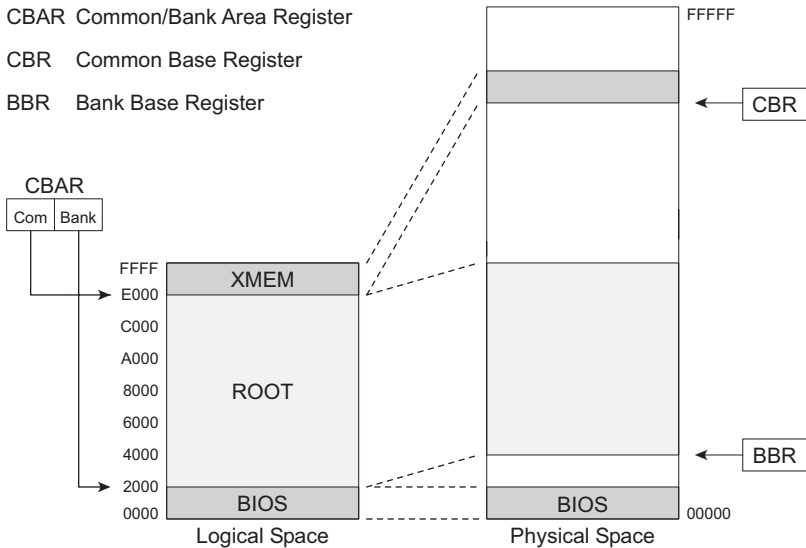


Figure D-1. Z180 On-Chip Memory Management Unit (MMU) Registers

The logical address space is partitioned on 4K boundaries. The upper half of the CBAR identifies the boundary between the **ROOT** memory and **XMEM**. The lower half of CBAR identifies the boundary between the **BIOS** and the **ROOT**. The start of the **BIOS** is always address 0. The two base registers CBR and BBR map **XMEM** and **ROOT**, respectively, onto physical memory.

Given a 16-bit address, the Z180 uses CBAR to determine whether the address is in **XMEM**, **BIOS**, or **ROOT**. If the address is in **XMEM**, the Z180 uses the CBR as the base to calculate the physical address. If the address is in **ROOT**, the Z180 uses the BBR. If the address is in the **BIOS**, the Z180 uses a base of 0.

A physical address is, essentially,

$$(\text{base} \ll 12) + \text{logical address}.$$

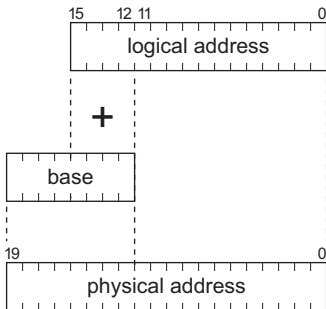


Figure D-2. Z180 Physical Addresses

Figure D-2 shows the address locations.

Memory Partitions

Table D-1. Dynamic C Memory Partitions

Name	Size	Description
BIOS	8K	Basic Input/Output System. The BIOS is always present and is always mapped to address 0 of ROM or flash. The BIOS contains the power-up code, the communication kernel, and important system features.
ROOT	48K	The area between the BIOS and XMEM (the bank area). The root—"normal" memory—resides in a fixed portion of physical memory. Root <i>code</i> grows upward in logical space from address 2000 (hex) and root <i>data</i> (static variables, stack and heap) grow down from E000. (Initialized static variables are placed with code, whether in ROM, flash, or RAM.)
XMEM	8K	XMEM is essentially an 8K "window" into extended physical memory. XMEM can map to any part of physical memory (ROM, flash, or RAM) simply by changing the CBR.

Table D-1 explains the memory partitions in Dynamic C.

The **XMEM** area has many mappings to physical memory. The mappings can change by changing the CBR as the program executes. Extended memory functions are mapped into **XMEM** as needed by changing the CBR. The mapping is automatic in C functions. However, code written in assembly language that calls functions in extended memory may need to do the mapping more specifically.

Functions may be classified as to where Dynamic C may load them. The keywords in Table D-2 apply to function definitions.

Table D-2. Memory Keyword Definitions

Keyword	Description
root	The function must be placed in root memory. It can call functions residing in extended memory.
xmem	The function must be placed in extended memory. Calls to extended memory functions are not as efficient as calls to functions in root memory. Long or infrequently used functions are appropriate for placement in extended memory.
anymem	This keyword lets the compiler decide where to place the function. A function's placement depends on the amount of reserve memory available. Refer to the Memory Options command in the OPTIONS menu.

Depending on which compiler options are selected, code segments will be placed in RAM, ROM, or flash.

Figure D-3 shows the memory layout with code in RAM.

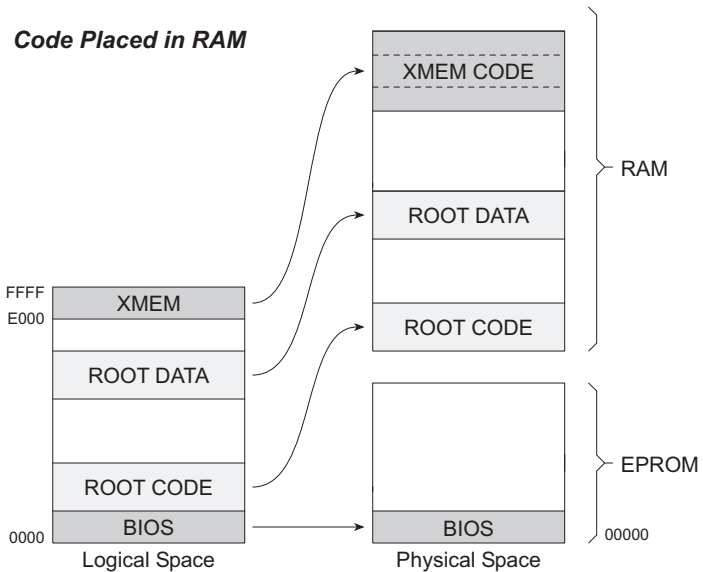


Figure D-3. Memory Layout with Code in RAM

Figure D-4 shows the memory layout with code in ROM or flash.

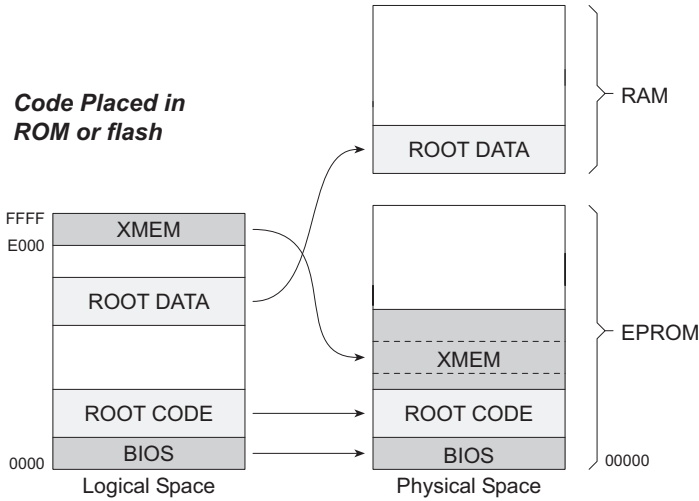


Figure D-4. Memory Layout with Code in ROM or Flash

Memory management in Dynamic C is automatic. The Dynamic C compiler emits code that will set the mapping registers.

Control over Memory Mapping

The programmer controls how Dynamic C allocates and maps memory.



Refer to the discussion of the **OPTIONS** menu in Chapter 4, The Dynamic C Environment.

Extended Memory Functions

Physical memory is divided into 4K “pages.” Two consecutive pages are visible in the extended memory window (**XMEM**) at any one time. Additional code is required to handle calls to functions or jumps to locations not currently mapped in the extended memory window.

A program can use many pages of extended memory. Under normal execution, code in extended memory maps to the logical address region E000_H to F000_H, the lower half of **XMEM**. As execution approaches F000, the pages are shifted so that the code in the region F000 to FFFF (the upper half) is moved down to the E000 to F000 region. The program automatically calls a function in root memory to accomplish this task. The function modifies the CBR to “slide” the code down one page and then jumps to the new location. This transfer of control is made at the end of the first statement that crosses F000. (Hence, no single C expression can be more than 4K long.)

However, **switch** or **while** statements that cause program jumps can be as long as desired. If a jump crosses page boundaries, the program uses a bouncer to execute the jump.

While any C function can call any other C function, no matter where in memory it is located, calling a function located in extended memory is less efficient than calling a function in root memory. That is because the program must use a bouncer to modify the CBR before and after the call.

A bouncer is a 4-byte code in root memory that points at the extended memory function and manipulates the stack and the CBR. Because of bouncers, calling extended memory functions is no different from calling root memory functions, assembly language or otherwise. *All* function “entry points” are in root memory.

Suggestions

Pure Assembly Routines

Pure assembly functions (not inline assembly code) *must* reside in root memory.

C Functions

C functions can be placed in root memory or extended memory. While access to variables in C statements is not affected by the placement of the function, there is bouncer overhead to call C functions in extended memory. Dynamic C will automatically place C functions in extended memory as root memory fills. Short, frequently used functions may be declared with the keyword **root** to force Dynamic C to load them in root memory.

Inline Assembly in C Functions

Inline assembly code may be written in any C function, regardless of whether it is compiled to extended memory or root memory.

However, because the bouncer of an extended memory function introduces four more bytes between the last pushed argument and the return address, the actual offset of arguments from the stack pointer *depends* on whether the code is compiled to extended memory or not. Therefore, it is important to use the symbolic names of stack-based variables instead of numeric offsets to access the variables. For example, if **j** is a stack variable, **@SP+j** is the actual offset of the variable from the stack pointer. Alternatively, if **IX** is the frame reference pointer, **“ix+j”** specifies the address of the stack-based variable.

Dynamic C issues a warning when it finds assembly code embedded in an extended memory function to discourage inline assembly segments that do not use symbolic offsets for stack-based variables. The warning can be disabled by appending the keyword `xmemok` after the `#asm` directive. Use symbolic names, not numeric offsets.



All static variables, even those local to extended memory functions, are placed in root memory. Keep this in mind if functions have many variables or large arrays. Root memory can fill up quickly.

Extended Memory Data

Most of the details of calling extended memory functions are handled by the compiler. The situation is more complicated for extended data. To access extended memory data, use function calls to exchange data between extended memory and root memory. These functions are provided in the Dynamic C libraries.



See `XMEM.LIB`.

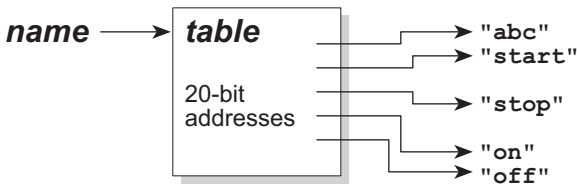
Extended memory addresses are 20-bit physical addresses (the lower 20 bits of a long integer). Pointers, on the other hand, are 16-bit machine addresses. They are not interchangeable. However, there are library functions to convert address formats.

Dynamic C includes two nonstandard keywords to support extended memory data: `xstring` and `xdata`.

The declaration

```
xstring name { string 1, ... string n };
```

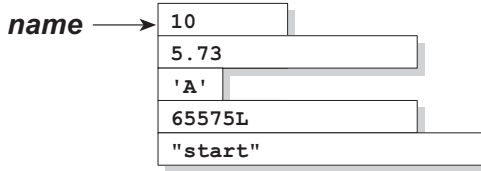
defines a table of 20-bit physical string addresses (as `unsigned long ints`), and corresponding strings. The term *name* represents the 20 bit physical address of the table in an `unsigned long int`.



The **xdata** statement has two forms. The declaration

```
xdata name { value 1, ... value n };
```

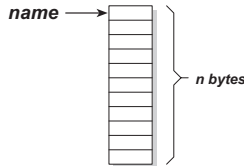
defines a block of initialized extended memory data. The values must be constant expressions of type **char**, **int**, **unsigned int**, **long**, **unsigned long**, **float**, or **string**.



The other form

```
xdata name [ n ];
```

defines a block of *n* bytes in extended memory.



In either case, the term *name* represents the 20-bit (physical) address of the block.

Use the following functions to move blocks of data between logical memory and physical memory. Pass addresses of extended memory data as long integers containing the 20-bit physical address in the least significant bits. Names declared with **xdata** and **xstring** are 20-bit extended memory addresses.

- **xmem2root(long src, void *dst, uint n)**
Copies *n* bytes from extended memory (**src**) to root memory starting at **dst**.
- **root2xmem (void *src, long dst, uint n)**
Copies *n* bytes from root memory (**src**) to extended memory starting at **dst**.
- **uint xstrlen (long address)**
Returns the length of the string at the address found at **address**. Keep in mind that an **xstring** declares an array of 20-bit addresses of strings.
- **long xgetlong (long address)**
Returns the long integer at the extended memory address.

The following example illustrates the use of extended memory.

```
xstring greetings {"hello there",
                  "good-bye",
                  "nice to see you",
                  "how have you been"};
xdata table      { 1.23, 1.45, 1.67, 1.85,
                  1.93, 2.04, 5.03, 6.78 };
xdata store[10000];
main(){
    float y;
    long j, k;
    int a;
    char my_chars[30];
// get one floating number at j
    j = ...
    xmem2root ( table + j*4, // x address
              &y,           // destination
              4 );         // # bytes
// two bytes from store
    j = ...
    xmem2root( store + j, &a, 2 );
    root2xmem( &a, store + j, 2 ); // other direction
// copy string to root
    j = 2;           // if we want "nice to see you"
    k = xgetlong( greetings+j*4 );// addr of XMEM string
    xmem2root( k,
              my_chars,           // destination
              xstrlen(k)+1 );// 1 is for null byte
}
```



Declarations involving **xdata** and **xstring** must be made outside the context of a function.



Refer also to **XDATA.C** in the Dynamic C **SAMPLES** subdirectory for another example.



*APPENDIX E: **COMPILER DIRECTIVES***

Compiler directives are commands that instruct the compiler how to proceed. They take the form of preprocessor commands, an example of which appears here.

```
#nouseix
```



These directives are detailed in Chapter 5, The Language. (The `#nodebug` directive automatically disables index checking, pointer checking and stack verification.)

Default Compiler Directives

Default compilation options are specified in the library header file `DC.HH`. The file `DC.HH` is compiled before any other library or user code. The following major defaults are set in `DC.HH`.

1. The default storage class for variables is `static`.

```
#class static
```

This default may be changed, but Z-World libraries will not work then. However, `static` is far more efficient and `auto` is often not required in embedded programming. Reentrant functions require `auto` variables.

2. The default memory allocation is `anymem`.

```
#memmap anymem
```

This allows Dynamic C to choose between root memory and extended memory.

3. The `nodebug` option is enabled when compiling code to ROM.

```
#if ROM == 1  
#nodebug  
#endif
```

The `#nodebug` global directive has extensive implications for generated code. Stack, index and pointer checking are disabled. All debugging features are removed from the code (especially `RST 28s`, which are used for break points). This generates smaller code that runs efficiently.

4. The default for the use of the IX register is `#nouseix`.



*APPENDIX F: **FILE FORMATS***

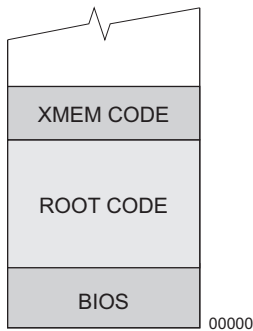
Use the **Compile to File** or **Compile to File with *.RTI File** command to generate an output file. Select the appropriate output file format in the **Compiler Options** dialog.

Layout of ROM Files

When a program (say, *filename.C*) is compiled for ROM, the compiler generates a file named *filename.BIN*. Dynamic C can also create an extended Intel HEX file (*filename.HEX*).

Select the **BIOS+Application (.BIN)** compiler option. Check the **Create HEX File Also** box to create an Intex hex file.

The resulting file contains the three code segments back-to-back. (Initialized data are constants and considered code. Uninitialized data are not included in the ROM file.)



The BIOS included in the ROM file is either (1) a copy of the first 2000_H of ROM of a Z-World controller connected to a development system or (2) a copy of the BIOS in an **.RTI** file, depending on the compile command selected.

Layout of Downloadable Files

Select the **Download via DLM (.DLP)** compiler option for downloadable files.

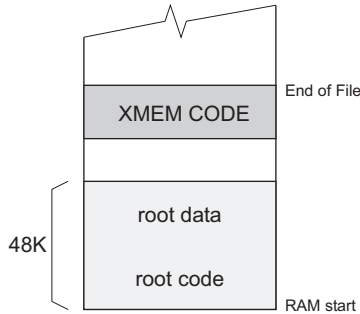


Details of the file format are found in Chapter 9, Remote Download.

Layout of Download to RAM Files

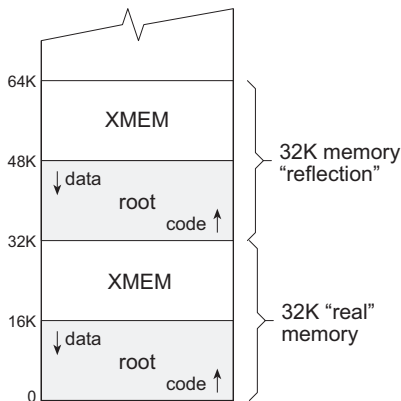
Select the **Application Only (.BPF)** compiler option.

The following diagram shows the locations of different segments in a RAM file.



Uninitialized data do not occupy any space in the RAM file. The root is always 48 kbytes long. The load address for the code in RAM is embedded in the file output. The default load address is 40000_H (256 kbytes) or 80000_H (512 kbytes), but can be changed. A RAM file starts with root code and not the BIOS, as the BIOS is expected to be in the ROM of the receiving controller.

The memory map for a 32-kbyte SRAM is as above, but the architecture takes advantage of memory reflection. The default code data gap is 8000_H (32 kbytes). This results in the actual layout of the code in the RAM as shown below.



The root data appear to start at 48 kbytes (and at 80 kbytes, 112 kbytes, ...) since it really starts at 16 kbytes because of the duplication of the memory image. Extended memory also appears to start at 48 kbytes. In this scheme, the sum of root code and data is limited to 16 kbytes, and the total extended memory code is limited to 16 kbytes.



Remember that all static variables, even those local to extended memory functions, are placed in root memory.

Hex File Information

A HEX file includes an identification flag and other pertinent information. This information starts at address 2300_H in a ROM file and at 0300_H in a RAM file, as indicated in Table F-1.

Table F-1. HEX File Information

0x?300	0xAA; identification byte
0x?301	0x55; identification byte
0x?302	0x81; identification byte
0x?303	0x42; identification byte
0x?304	0x01 for ROM file, 0x00 for RAM file
0x?305-6	16-bit cyclic redundancy check
0x?307	BBR register value
0x?308-9	16-bit address of first free byte above root code

Startup code for the program appears at 2200_H in a ROM file and at 0200_H in a RAM file. The startup code performs the following functions.

1. Load the stack pointer (SP).
2. Set the I register (interrupt base).
3. Enable interrupts (**ei**).
4. Call library function named **.startup**.
5. Call **bfree** to initialize heap management.
6. Set **ERROR_EXIT** to **exit()** as a default.
7. Set aux stack (checking and debugging) and stack limit.
8. Reset **RST 28** vector to debugger.
9. Push time in seconds since 1-JAN-1980 on stack as an **unsigned long**.
10. Push program revision, as an **int**, on stack.
11. Call **main()**.

Jumping to Another Program

Conceivably, several programs may be downloaded, all in different sections of physical memory. There is a function resident in ROM that makes it possible to jump from one program to another.

```
void newbbr( uint offset, uint CBAR_BBR )
```

This function *does not* return, but jumps to and starts up the program specified by its arguments.

For example, say a program has been downloaded at address $9C000_H$. Then, the BBR is $9C_H - 2_H$ (for the size of the BIOS) or $9A_H$. The CBAR is always $E2_H$ (for RAM). The offset is always 2200_H (for RAM). The following call to **newbbr** would be made.

```
newbbr( 0x2200, 0xE29A )
```

Using the **newbbr** function requires a fair amount of mastery with Dynamic C and the target controller.

Burning ROM

Z-World controllers support several types of EPROM, including the following.

27C256	32 kbytes	28 pins
27C512	64 kbytes	28 pins
27C010	128 kbytes	32 pins

Copyright Notice

The Dynamic C library is copyrighted. Place the following copyright notice on any ROM created with Dynamic C.

© 1990-2002 Z-World, Inc.

In addition to this notice, a copyright notice may be added to protect other user-written code.

Purchasers of the copyrighted Dynamic C software and a copyrighted Z-World ROM are granted permission to copy the ROM, as described above, provided that

1. The resulting ROMs are used only with Z-World controllers.
2. The above copyright notice is placed on the ROM.



*APPENDIX G: **RESET FUNCTIONS***

Z-World's embedded applications need to differentiate the cause of resets and restarts. Table G-1 lists some possible hardware resets.

Table G-1. Possible Hardware Resets

Regular Reset	The system /RESET line is pulled low and released.
Power Failure Reset	Power drops below a threshold, and the supervisor chip pulls /RESET low and causes a reset.
Watchdog Reset	The watchdog timer was not reset. It pulls /RESET low and causes a reset.

In addition to these hardware resets, an application may cause a *super reset*. A super reset is necessary because important system data should persist over the occurrence of regular resets and power failures.

Z-World's super reset is a mechanism to initialize certain persistent data in battery-backed RAM. A normal reset does not initialize these data, but *retains* their values. A super reset always occurs when a program is first loaded. Subsequent resets are normal resets, unless the software performs a super reset intentionally.

Reset Differentiation

Dynamic C includes a set of functions to differentiate the various resets. These functions are grouped into two main categories.

1. The function names begin with an underbar (`_`), have important side effects, and may only be called *once and only once* at the beginning of the `main` program.
2. The function names do not begin with an underbar, have no side effects, and may be called anywhere in a program.

- **`int _sysIsSuperReset()`**

This function detects whether a super reset was requested. The function returns 1 if a super reset was requested and 0 if not.

If a super reset was requested, this function calls `_prot_init` to initialize the protected variable feature. In addition, it calls the function chain `sysSupRstChain`. Additional code may be added to this function chain.

If a super reset was not requested, this function also calls `_prot_recover` to recover partially written protected variables.

- **int _sysIsPwrFail()**

This function determines whether the system had a power failure just before restarting. The function return 1 if a power failure occurred and 0 otherwise. A custom power-failure handler cannot be used with this function.

- **int _sysIsWDTO()**

This function determines whether the system was reset by a watchdog timeout. The function returns 1 if a watchdog timeout occurred and 0 otherwise.

The following is the recommended reset detection sequence. It should be done before anything else in the **main** function.

```
main() {
    ...
    declarations
    ...
    if( _sysIsSuperReset()    ){
        statements
    }else if( _sysIsPwrFail() ){
        statements
    }else if( _sysIsWDTO()   ){
        statements
    }else{
        statements
    }
    ...
    rest of main
}
```

Functions of the second category have names similar to those in the first category, but they do not have initial underbars.

```
int sysIsSuperReset()
int sysIsPwrFail()
int sysIsWDTO()
```

These functions reflect the cause of the last reset. They can be called anywhere in the program as often as needed. Functions of the first category can only be called at the beginning of **main**.

Reset Generation

Software can generate two types of system reset.

Call **sysForceReset** to turn off interrupts and wait until the watchdog resets. This reset will be registered as a watchdog reset when the application restarts.

Call **sysForceSupRst** to request a super reset. This function turns off interrupts and waits until the watchdog resets. This reset will be registered as a super reset when the application restarts.

The controller must have a hardware watchdog enabled for either of these functions to work.



*APPENDIX H: **EXISTING FUNCTION CHAINS***

The function chains in Table H-1 exist in the libraries specified. Segments may be added to these chains. Don't redefine a chain if its library is #used, noting that **SYS.LIB** is #used by default.

Table H-1. Dynamic C Function Chains

Library	Function Chain	Description
AASC.LIB	_aascInitDF	Registers AASC hardware dependent function pointers.
SRTK.LIB	_srtk_hightask	A chain of code executed by SRTK every 25 ms in the high-priority task.
SRTK.LIB	_srtk_lowtask	A chain of code executed by SRTK every 100 ms in the low-priority task.
SYS.LIB	_sys_25ms	After sysInitTimer1 is called, a chain of code executed every 25 ms in the PRT1 ISR (with interrupts disabled).
SYS.LIB	_sys_25msPostEI	After sysInitTimer1 is called, a chain of code executed every 25 ms in the PRT1 ISR (with interrupts enabled).
SYS.LIB	_sys_390	On PK2100 series controllers after sysInitTimer1 is called, a chain of code executed every 0.390 ms in the PRT1 ISR (with interrupts disabled).
SYS.LIB	_sys_781	After sysInitTimer1 is called, a chain of code executed every 0.781 ms in the PRT1 ISR (with interrupts disabled).
SYS.LIB	_sys_781PostEI	After sysInitTimer1 is called, a chain of code executed every 0.781 ms in the PRT1 ISR (with interrupts enabled).
SYS.LIB	sysSupRstChain	A chain of tasks to perform when super resetting.
VDRIVER.LIB	_GLOBAL_INIT	Performs general global initialization tasks. Users are encouraged to add segments to this chain.



*APPENDIX I: **NEW FEATURES***

The reader is encouraged to read the *Release_Notes.txt* file in Dynamic C's main installation folder for more comprehensive information on changes.

Dynamic C 32 IDE

Compiler Options, Output Generation Group

The **Zero Time Stamp** checkbox has been added to help facilitate code certification. When enabled, forces to zero the compile time-stamp and compiler performance information which is embedded into the compiled code. Identical compiler output will always generated, given a fixed set of Dynamic C 32 compiler version and options, application code and library code.

Compiler Options, File Type for 'Compile to File' Group

The compile to .BIN (EPROM) file options have been expanded to include the ability to create .BIN files which can communicate with Dynamic C or the Program Loader Utility after being externally programmed into a Flash EPROM chip. The simulated EEPROM area options include replacing it (copying from the attached controller or new-style .RTI file), clearing (zeroing) it or excluding it (not specifically programming the Flash chip's simulated EEPROM area at all).

In support of the new .BIN file options, the Remote Target Information (.RTI) file has been extended to include a Flash equipped source controller's simulated EEPROM area. While Dynamic C 32 version 6.30 can use a .RTI file created by a previous version of Dynamic C, a .BIN file created by compiling to file with an older .RTI file effectively only gives the choice between clearing or excluding the simulated EEPROM area. Previous versions of Dynamic C can not use the extended .RTI file format.

Target Communication

High resolution inter-character gaps have been reintroduced, allowing more reliable target communication during debugging, especially when combining high bps serial rates with slow controllers.

New Libraries

GESUPRT.LIB provides support for interfacing an application to the Graphics Engine. **SF1000_Z.LIB** provides support for applications requiring the extra nonvolatile storage that is available from Z-World's SF1000 series of Serial Flash memory cards.

Program Loader Utility

A new chapter has been added to explain basic operation of the Program Loader Utility which is included with Dynamic C. See *Chapter 10: Local Upload*.

Symbols

! logical NOT operator	117	%= operator	123
!= operator	121	& (address operator)	94, 119
# operator	84, 85	&& operator	122
## operator	84, 85	&= operator	123
#asm ... 21, 83, 124, 130, 131, 188, 204		() parentheses	
#class	24, 105, 125, 208	as operators	116
#debug ... 101, 105, 107, 109, 125, 188		(type) operator	118
#define ... 81, 84, 85, 86, 112, 125, 126		* (indirection operator)	94, 118
#elif	125, 126	*= operator	123
#else	125, 126	+ operator	118
#endasm 21, 83, 124, 130, 131, 132		++ increment operator	117
#endif	125, 126	+= operator	122
#error	125	-> right arrow operator	116
#fatal	125	. dot	
#funcchain	19, 125	as operator	116
#if	125	.BPF files	60
#ifdef	126	.DLP for Download	172
#ifndef	126	.RTI	222
#include		/ operator	119
absence of	18, 23, 77	/= operator	123
#INT_VEC	164, 180	; semicolon operator	78, 132
#interleave	126	< operator	120
#JUMP_VEC	164	<< operator	120
#KILL	126	<<= operator	123
#makechain	19, 126	<= operator	120
#mmap	22, 105, 126, 208	= operator	122
#nodebug . 51, 101, 105, 107, 109, 111, 125, 188, 208		= operator	121
#nointerleave	126	> operator	121
#nouseix	23, 105, 127, 208	>= operator	121
#undef	86, 126	>> operator	120
#use ... 18, 23, 77, 81, 83, 127, 195		>>= operator	123
#useix	23, 105, 127, 190	? : operator	122
#warns	125	@PC	134
#warnt	125	@RETVAl	141
% operator	120	@SP 136, 137, 139, 140, 141, 145, 203	
		[] array indices	116
		\ backslash	132
		for character literals	84
		^ operator	121
		^= operator	123

<code>_aascInitDF</code>	220	addresses in assembly language	134, 137
<code>_GLOBAL_INIT</code> ...	107, 150, 156, 157, 220	aggregate data types	92
function chain	20	ALT key	41
initializing CoData	150	ALT-Backspace	46
<code>_prot_init</code>	185, 216	ALT-C	50
<code>_prot_recover</code>	185, 216	ALT-CTRL-F3	50, 51
<code>_srtek_hightask</code>	220	ALT-F	41, 42
<code>_srtek_lowtask</code>	220	ALT-F10	35, 56
<code>_sys_25ms</code>	220	ALT-F2	52, 53
<code>_sys_25msPostEI</code>	220	ALT-F4	42, 45
<code>_sys_390</code>	220	ALT-F9	34, 52
<code>_sys_781</code>	220	ALT-H	71
<code>_sys_781PostEI</code>	220	ALT-I	54
<code>_sysIsPwrFail</code>	26, 217	ALT-O	58
<code>_sysIsSuperReset</code>	26, 185, 216, 217	ALT-R	52
<code>_sysIsWDTO</code>	26, 217	ALT-SHIFT-backspace	46
{ } curly braces	78	ALT-W	68
operator	122	always_on	149, 151, 154, 156
= operator	123	analog input	16
operator	122	analog output	16
~ bitwise complement operator	117	AND	
27C010	213	assign operator (&=)	123
27C256	213	logical operator (&&)	122
27C512	213	anymem	66, 100, 126, 208
A		Append Log	62
abort	100, 149, 150, 154, 156	application files	76
About Dynamic C	74	Application Only (.BPF)...	30, 211
abstract data types	80	argument passing .	18, 23, 95, 135, 136, 141, 142, 143
active window	43	modifying value	95
adc (add-with-carry)	130	arguments	23
Add to Top button	55	arrange icons	
add-assign operator (+)=)	122	command	68
add-with-carry (adc)	130	arranged icons	69
Add/Del Items <CTRL-W>	36, 55, 69	arrays	92, 93, 95
Add/Del Watch Expression		bounds checking	188
<CTRL-W>	35	characters	113, 114
Add/Del Watch Expression <CTRL-W>	54, 55	indices	116
adding watch window items	54, 55	subscripts	92
address operator (&)	94, 119	arrow keys	40, 41
address space	22	for cursor	
		positioning	41
		for editing text	41

ASCII serial port 143
 assembly code multi-line macro
 134
 assembly language . 16, 21, 34, 53,
 83, 124, 130, 131, 132, 133,
 140, 141, 142, 143, 144, 145,
 188, 203
 #asm directive 21
 #endasm directive 21
 embedding C statements 130
 Assembly window 70
 assembly window .. 16, 33, 34, 68,
 69, 70, 130
 assign operator (=) 122
 assignment operators 122, 123
 associativity 115, 124
 auto 18, 19, 23, 24, 100, 125, 134,
 135, 136, 138, 139, 188, 189,
 190, 208
 Auto Open STDIO Window 62
 auxiliary stack 184, 212
 size 65

B

backslash
 character literals 114
 continuation in directives 124
 backup battery 37
 basic unit of a C program 78
 battery backup 37
 baud rate 67
 BBR 96, 97, 98, 198, 199, 200,
 201, 202, 213
 BCDE 131, 135, 141, 142
 BCDE (primary register) 189
 BeginHeader 23, 82, 83, 102, 194,
 195
 beginning of file 41
 beginning of line 41
 BIN files . 31, 32, 50, 60, 176, 210
 binary operators 115
 BIOS 22, 30, 31, 32, 35, 76, 96, 97,
 98, 102, 126, 199, 200, 201,
 210, 211, 213
 library functions 31
 symbol library 31, 32
 BIOS+App+Lib+ClrSEE (.BIN) 32
 BIOS+App+Lib+SimEE (.BIN) 31
 BIOS+App+Library (.BIN) 31
 BIOS+Application (.BIN) 31, 210
 bitwise
 AND operator (&) 119
 complement operator (~) 117
 exclusive OR operator (^) ... 121
 inclusive OR operator (|) 122
 body
 module 82, 83, 195
 bounce 203
 BPF files 30, 50, 176
 branching 89, 90
 break 87, 88, 90, 100, 109
 example 88
 break points 16, 35, 53, 56, 104,
 130, 188, 208
 hard 35, 52, 53
 interrupt status 35, 52, 53
 soft 35, 52, 53, 183
 breaking out of a loop 88
 breaking out of a switch statement
 88
 buttons, toolbar 67

C

C files 76
 C functions calling assembly code
 140
 C language . 16, 17, 18, 19, 20, 21,
 22, 23, 24, 25, 33, 35, 76, 80,
 91, 95, 99, 113, 133, 135
 C statements embedded in assem-
 bly code 130
 C strings 91
 C variables in assembly language
 134
 cascaded windows 68, 69
 case 90, 100, 101, 102, 109
 case-sensitive searching 47, 48, 49
 cast operator (type) 118

CBAR .. 96, 97, 98, 198, 199, 200, 201, 202, 213
 CBR (common-base register) .. 56, 96, 97, 98, 140, 145, 162, 198, 199, 200, 201, 202, 203
 char 80, 101, 111, 128, 190, 205
 characters 114
 arrays 91, 113, 114
 constants 114
 embedded quotes 114
 nonprinting values 114
 special values 114
 check sum 170
 checking
 array bounds 188
 indices 208
 pointers 94, 188, 208
 stack 188, 189, 190, 208
 syntax 32
 type 32, 79
 ChkSum 157
 ChkSum2 157
 Clear Watch Window 54, 55
 clipboard 46, 47
 clocked serial communication .. 24
 clocks 16
 Close <CTRL-F4> 43
 closing a file 42, 43
 CoBegin 150, 156, 157
 CoData ... 149, 150, 155, 156, 157, 158, 160
 description 156
 general usage 158
 initialization 150
 structure
 user defined 159
 code size 189
 Code with BIOS 30, 32
 coercion 118
 COM port 67, 169
 PC 29, 169
 comma operator 123
 comment 81
 multi-line 81
 single line 81
 comments 78
 common base register (CBR) 140, 145
 communication
 RS-232 16, 24
 RS-485 16, 24
 serial 16, 24, 67, 163
 clocked 24
 with Dynamic C 29
 compilation 28, 32, 40, 50, 51, 60, 69, 71, 126
 direct 23
 direct to controller 16
 errors 49
 speed 16, 17
 targetless . 29, 30, 31, 32, 50, 51
 COMPILE menu 41, 50, 51
 compile time-stamp 60
 Compile to File <CTRL-F3> ... 32, 50, 51, 60, 171, 210
 Compile to File with *.RTI File
 <ALT-CTRL-F3> 29, 30, 31, 32, 50, 51, 60, 172, 210
 Compile to Target <F3> 29, 50, 51
 compiler directives 124, 208
 default 208
 compiler options 30, 40, 50, 58, 59, 60, 61, 62, 94, 171, 189, 201, 210, 211, 213, 222
 Application Only (.BPF) 30, 211
 BIOS+App+Lib+ClrSEE (.BIN) 32
 BIOS+App+Lib+SimEE (.BIN) 31
 BIOS+App+Library (.BIN) .. 31
 BIOS+Application (.BIN) ... 31, 210
 Create HEX File Also 60
 Download via DLM (.DLP) 32, 210
 Null Device (Bit Bucket) 32
 Program Loader Utility 30
 simulated EEPROM 31, 32
 Zero Time Stamp 60, 222

compiling 16, 28, 60
 to file 29, 30, 31, 32, 40, 50, 51, 210
 to flash 51
 to RAM 50, 51
 to ROM 50, 51
 to target 29, 32, 40, 50, 51
 compound
 names 112
 statements 78
 concurrent processes ... 18, 21, 148
 conditional operation (? :) 122
 constants
 character 114
 named 112
 content 157
 contents
 HELP 71
 continue 87, 88, 101, 109
 example 88
 cooperative multitasking .. 21, 148, 149, 150, 151, 152, 153, 154, 155, 156, 157, 158, 159, 160
 CoPause 150, 156
 copying text <CTRL-C> 46, 47
 copyright 213
 CoReset 150, 156, 157
 CoResume 150, 156
 costate 101, 150, 154, 158
 costatements 18, 21, 100, 101, 110, 111, 148, 149, 150, 151, 152, 153, 154, 155, 156, 157, 158, 159, 160
 abort 154
 aborted 154
 always on 149, 151, 154
 firsttime flag and functions 157, 158
 initially off 149, 151
 initially on 149, 151
 multiple threads .. 158, 159, 160
 named 149, 150
 shared code 159
 shared data 160
 state 149, 150, 151, 154, 156, 157
 suspended .. 148, 149, 151, 152, 153, 154
 syntax 150
 unnamed 149, 150, 154
 waitfor 151, 152
 yield 153
 CRC 170
 Create *.RTI File for Targetless
 Compile ... 29, 30, 31, 32, 50, 51, 172
 Create HEX File Also 60, 210
 creating
 new file 42
 standalone programs 36, 37
 CSState 156
 CTRL key 41
 CTRL-C 47
 CTRL-F10 34, 56
 CTRL-F2 54
 CTRL-F3 50, 51
 CTRL-F4 43
 CTRL-G 49
 CTRL-H 37, 38, 72, 73, 74
 CTRL-I 35, 52, 53
 CTRL-N 49
 CTRL-O 34, 52, 53
 CTRL-P 49
 CTRL-U 36, 54, 56
 CTRL-V 47
 CTRL-W 35, 36, 54, 55
 CTRL-X 46
 CTRL-Y 52, 54
 CTRL-Z 36, 52
 curly braces { } 78
 cursor
 execution 34, 53, 54
 positioning 49
 positioning with arrow keys . 41
 text 40, 74
 custom error handler 182
 cutting text <CTRL-X> 46

D

- data in extended memory 127, 128, 204, 205, 206
- data types 92
 - aggregate 92
 - primitive 91, 92
- db 132
- DC.HH 208
- DCW.CFG 68
- debug 101, 124
 - editor 66
 - mode 49, 52, 190
 - same as run mode 32
- debug mode 29
- debugger 16, 32, 212
 - options 58, 62
- debugging .. 16, 32, 33, 36, 52, 53, 55, 56, 101, 104, 124, 125, 163, 188, 208, 212
 - Append Log 62
 - assembly-level view 16
 - Auto Open STDIO Window . 62
 - Log File 62
 - Log STDOUT 62
 - windows 33, 34
- declarations 78, 82, 194
- decrement operator (--) 117
- default 90, 102, 109
 - directives 208
 - storage class 18, 19, 24
- DEFAULT.H 23
- Del from Top button 36, 55
- Delay Functions 152
- DelayMS 149, 157
- DelaySec 149, 157
- DelayTicks 149, 157
- deleting watch window items .. 54, 55
- demotion 60
- descriptive function headers 37, 38
- DI 35, 163
- digital input/output 16
- direct
 - compilation 16, 23
 - memory access (DMA) 24
- directives 22, 124, 164, 208
 - #asm 21, 83, 124, 130, 131, 188, 204
 - #class 24, 125, 208
 - #debug 101, 105, 107, 109, 111, 125, 188
 - #define 81, 84, 85, 86, 125, 126
 - #elif 125, 126
 - #else 125, 126
 - #endasm . 21, 83, 124, 130, 131, 132
 - #endif 125, 126
 - #error 125
 - #fatal 125
 - #funcchain 19, 125
 - #if 125
 - #ifdef 126
 - #ifndef 126
 - #interleave 126
 - #KILL 126
 - #makechain 19, 126
 - #mmap 126, 208
 - #nodebug 51, 101, 105, 107, 109, 111, 125, 188, 208
 - #nointerleave 126
 - #nouseix 23, 105, 127, 208
 - #undef 86, 126
 - #use 18, 23, 77, 81, 83, 127, 195
 - #useix 23, 105, 127, 190
 - #warns 125
 - #warnt 125
 - default 208
- Disassemble at Address <ALT-F10> 56, 70
- Disassemble at Cursor <CTRL-F10> 34, 56, 70
- disassembled code 54
- disassembler 34
- display
 - options 58, 66
- divide-assign operator (/=) 123
- division operator (/) 119
- DLC (download configuration) file 171

- DLM (Download Manager)... 168, 169, 170, 171, 172, 173, 210
 - DLM_MAX_PW_LEN 170
 - DLM_MIN_PW_LEN 170
 - DLM_PASSWORD_LVL 170
 - DLM_PASSWORD_STR 170, 172
 - DLM_TIMEOUT 170
 - DLM_Z0.C 169, 172
 - DLP files .. 50, 168, 170, 171, 172, 173, 210
 - DMA channels 24
 - do loop 87, 102
 - dot operator 93, 112, 116
 - download
 - configuration (DLC) file 171
 - local 176
 - remote .. 33, 168, 169, 170, 171, 172, 173, 210
 - Download Manager (DLM) 40, 51, 168, 169, 170, 171, 172, 173, 210
 - Download Program 170
 - Download to RAM 211
 - Download via DLM (.DLP) 32, 171, 210
 - downloadable
 - files 51
 - program 29, 30, 31, 32, 170, 171, 172, 173
 - downloading 29, 30, 31, 32
 - dummy call 36
 - Dump at Address 56
 - Dump to File 57
 - dump window 57
 - dw 132
 - dynamic
 - memory allocation 65
 - storage allocation 94
 - variables 94
 - Dynamic C . 14, 16, 17, 23, 28, 51, 126, 213
 - Application Frameworks 160
 - communication 29, 163
 - debugger 32, 33
 - differences 17, 18, 19, 20, 21, 22, 23, 24, 25, 76
 - exit 45, 68
 - Help Contents 37
 - installation 14, 28
 - installation procedure 14
 - installation requirements 14
 - license agreement 14
 - program group 14, 28
 - support files 77
 - usage 28
- ## E
- EDIT menu . 36, 41, 46, 47, 48, 49
 - edit mode 36, 40, 46, 49, 54
 - editing 16, 40
 - options 40
 - editor 16
 - options 58
 - EI 35, 162, 163
 - ei 144, 162, 212
 - else 102
 - embedded assembly code .. 16, 21, 135, 141, 142, 143, 144, 145
 - embedded quotes 114
 - End key 40, 41
 - end of file 41
 - end of line 41
 - EndHeader ... 23, 82, 83, 102, 194, 195
 - Enter Password 169, 170
 - enumerated types
 - absence of 24
 - EPROM 16, 18, 19, 22, 25, 29, 30, 31, 32, 37, 40, 50, 64, 98, 99, 112, 164, 168, 182, 198, 201, 202, 208, 210, 212, 213
 - file generation 60
 - flash 29, 37, 40, 51, 98, 99, 112, 168, 198, 201, 202
 - equ 132
 - equal operator (==) 121
 - ERROR_EXIT 103, 182, 212

errors

- codes 183
- editor 66
- fatal 183, 184, 185
- handler
 - custom 182
 - standard 182
- locating 49
- logging 184, 185
- recovery 184, 185
- run-time 182

ESC key 41

- to close menu 41

Evaluate button 36, 55

examples

- break 88
- continue 88
- for loop 87
- goto 88
- modules 83
- multithreaded costatements 159
- of array 92
- union 28, 93

Execute Downloaded Program 170

execution 52, 55, 56

- cursor 34, 53, 54
- speed 189

Exit <ALT-F4> 45, 212

Expr. in Call 74

extended memory .. 18, 22, 63, 66, 96, 97, 98, 100, 102, 111, 124, 126, 127, 139, 140, 145, 169, 198, 199, 200, 201, 202, 203, 208

- data 127, 128, 204, 205, 206
- functions 204
- strings 128, 204, 205, 206

extensions

- real time 25

extern 18, 22, 83, 102, 195

F

F (status register) 70

F10 68

F2 52, 53

F3 50, 51

F4 36, 49

F5 47

F6 48

F7 34, 52, 53

F8 34, 52, 53

F9 34, 52

fast 102

fatal errors 183, 184, 185

FILE menu 41, 42, 43, 44, 45

Find

- Case sensitive 47
- From cursor 47
- Reverse 47

Find <F5> 46, 47

Find next <SHIFT-F5> 46, 49

firsttime 102, 150, 157

- flaf 157
- flag 157, 158
- functions 157, 158

flash 98, 198, 201

flash EPROM... 29, 37, 40, 51, 98, 99, 112, 168, 198, 201, 202

flash memory 18, 19

float 36, 80, 102, 111, 128, 205

- values 113

floating-point speed 24

for 78, 103

- character literals 114
- loop 87
- example 87

frame

- reference point 141
- reference pointer .. 23, 105, 139, 140, 143, 190, 203

free 94

free memory 65

Free Size 65

free size 65

free space 71, 94
 Full Speed Bkgnd TX 67
 function calls.. 34, 74, 79, 86, 100,
 127, 135, 136, 140, 141, 142,
 145, 190
 indirect 143
 function chains 18, 19, 20, 107,
 126, 150, 156, 157, 216, 220
 _aascInitDF 220
 _GLOBAL_INIT 220
 _srtk_hightask 220
 _srtk_lowtask 220
 _sys_25ms 220
 _sys_25msPostEI 220
 _sys_390 220
 _sys_781 220
 _sys_781PostEI 220
 sysSupRstChain 26, 220
 function headers 193, 194
 descriptive 38
 function headers, descriptive 37
 function help 194
 function libraries 17, 23, 76, 77, 82,
 102, 127, 220
 function lookup <CTRL-H> 37, 38,
 72, 73, 74
 function returns 141, 142, 143, 190
 functions 78
 entry and exit 190
 prototypes 23, 38, 79, 80, 82, 83

G

GESUPRTLIB 222
 global initialization 20, 21, 156,
 157
 global variables 94, 188
 goto 88, 89, 103
 example 88
 Goto <CTRL-G> 46, 49
 Graphics Engine 222
 greater than operator (>) 121
 greater than or equal operator (>=)
 121

H

Hangup Remote Modem 170
 HANGUP_TIMEOUT 171
 hard break points 35, 52, 53
 hardware reset 25, 184, 185
 header
 BeginHeader 23
 EndHeader 23
 function 37, 38, 193, 194
 library 193
 module 82, 83, 102, 195
 Heap Size 65
 heap storage 71, 94, 212
 Help
 contents 37
 online 37
 topical 37
 help
 online 71, 72
 HELP menu 37, 38, 41, 71, 72, 73,
 74
 HEX files 30, 60, 210, 212
 information 212
 hexadecimal integer values 113
 high-current output 16
 hitwd 185
 HL ... 131, 135, 137, 141, 142, 143
 HL (primary register) 189
 Home key 40, 41
 horizontal tiling 68, 69

I

I register 163, 164, 212
 IBM PC 16, 52, 67
 icons
 arranged 68, 69
 IEEE floating point 102
 if 102, 103
 multichoice 90
 simple 89
 with else 89
 iff 35, 163

immediate evaluation
 watch line 36
 increment operator (++) 117
 index checking 208
 index registers 18, 23, 131, 143
 indirect function calls 103, 143
 indirection operator (*) 94, 118
 Information Window 65
 Information window 71
 information window 65, 68, 69, 71
 init_on 149, 151
 initialization
 global 20, 21
 initialized data 210
 initialized static variables
 placed in ROM 19
 input
 analog 16
 digital 16
 insertion point 47, 49
 Inspect 69, 70
 INSPECT menu 34, 41, 54, 55, 56,
 69
 installation
 Dynamic C 14, 28
 requirements 28
 installation procedure 14
 installation requirements 14
 int 80, 81, 104, 107, 110, 111, 128,
 205
 as default function type 78
 integer values 113
 Intel
 extended HEX format 212
 HEX files 60
 HEX format 30, 60, 210, 212
 interrupt service routines ... 17, 21,
 104, 106, 143, 144, 145, 162,
 163, 164, 165
 example 162, 165
 interrupt status
 and break points 35, 52, 53
 interrupt vectors
 setting 164
 interrupts .. 21, 104, 106, 107, 143,
 144, 145, 162, 163, 164, 165,
 170, 218
 base 212
 disabling 163
 flag 35, 53
 latency 143, 163
 Mode 0 164
 Mode 1 164
 Mode 2 164
 service routines 21
 IntervalMS 149
 IntervalSec 149
 isCoDone 150, 156
 isCoRunning 150, 156
 ISR (interrupt service routines)
 162, 164
 IX (index register) 18, 23, 105, 110,
 127, 131, 138, 139, 140, 143,
 184, 189, 190, 203, 208
 IY (index register) 131, 143

J

jump vectors
 setting 164

K

kernel
 real-time 25, 190
 key module 82, 194
 keyboard shortcuts 72
 keystrokes 72
 <ALT E> select EDIT menu . 46
 <ALT R> select RUN menu . 52
 <ALT-Backspace> undoing
 changes 46
 <ALT-C> select COMPILE
 menu 50
 <ALT-CTRL-F3> Compile to
 File with *.RTI File 51
 <ALT-F> select FILE menu . 41,
 42
 <ALT-F10> Disassemble at
 Address 34, 56

<ALT-F2> Toggle hard break point 52, 53
 <ALT-F4> Exit 45
 <ALT-F4> Quitting Dynamic C 42
 <ALT-F9> Run w/ No Polling 52
 <ALT-H> select HELP menu 71
 <ALT-I> select INSPECT menu 54
 <ALT-O> select OPTIONS menu 58
 <ALT-SHIFT-backspace> redoing changes 46
 <ALT-W> select WINDOW menu 68
 <CTRL-C> copying text. 46, 47
 <CTRL-F> Compile to File .. 50
 <CTRL-F10> Disassemble at Cursor 34, 56
 <CTRL-F2> Reset Program 52, 54
 <CTRL-F3> Compile to File with *.RTI File 50, 51
 <CTRL-F4> Close 43
 <CTRL-G> Goto 46, 49
 <CTRL-H> Library Help lookup 37, 38, 41, 72, 73, 74
 <CTRL-I> Toggle interrupt . 35, 52, 53
 <CTRL-N> next error 46, 49
 <CTRL-O> Toggle polling .. 52, 53
 <CTRL-P> previous error 46, 49
 <CTRL-U> Update Watch window 36, 54, 56
 <CTRL-V> pasting text .. 46, 47
 <CTRL-W> Add/Del Items . 35, 36, 54, 55
 <CTRL-X> cutting text 46
 <CTRL-Y> Reset target . 52, 54
 <CTRL-Z> Stop 36, 52
 <F10> Assembly window 68
 <F2> Toggle break point 52, 53
 <F3> Compile to Target.. 50, 51

<F4> switching to edit mode 46
 <F5> finding text 46, 47
 <F6> replacing text 46
 <F7> Trace into 52, 53
 <F8> Step over 52, 53
 <F9> Run 52
 <SHIFT-F5> Find next ... 46, 49
 keywords.. 19, 22, 76, 98, 99, 100, 101, 102, 103, 104, 105, 106, 107, 108, 109, 110, 111, 139, 150, 162, 188, 201

L

language elements 99, 112, 113
 operators 115
 lastlocADDR 156, 157
 lastlocCBR 156, 157
 latency interrupts 143
 less than operator (<) 120
 less than or equal operator (<=) 120
 Lib Entries 37, 72
 LIB files 76
 LIB.DIR 23, 72, 83, 127, 195
 libraries .. 17, 23, 76, 77, 102, 127, 220
 function 23
 function prototypes 23
 lookup dialog 37
 modules 82, 194
 real-time programming ... 17, 25
 library functions 37, 66, 72, 126
 library headers 193
 Library Help lookup <CTRL-H> 72, 73, 74, 194
 license agreement 14
 line continuation in directives . 124
 linking 16
 local download 176
 PRGLOADR.EXE 176
 locating errors 49
 Log File 62
 Log STDOUT 62
 logical AND (&&) 122

- logical memory 63, 65, 96, 97, 98, 198, 199, 200, 201, 202, 203
 - logical operators 117, 122
 - logical OR (|) 122
 - long 36, 104, 111, 128, 205
 - long integer values 113
 - longjmp 184
 - lookup function <CTRL-H> 72, 73, 74
 - loops 86, 87, 102, 103
 - breaking out of 88
 - skipping to next pass 88
- M**
- macro
 - multi-line assembly code 134
 - macros ... 84, 85, 86, 125, 132, 133
 - restrictions 86
 - with parameters 84
 - main function . 51, 76, 78, 79, 104, 188, 212, 216, 217
 - malloc 65, 94
 - memory
 - allocation 66
 - dump 54
 - extended . 18, 22, 63, 66, 96, 97, 98, 100, 102, 111, 124, 126, 127, 139, 140, 145, 169, 198, 199, 200, 201, 202, 203, 204, 205, 206, 208
 - data 127, 128, 205
 - strings 128, 204, 205
 - flash 18, 19
 - logical .. 63, 65, 96, 97, 98, 198, 199, 200, 201, 202, 203
 - management 96, 97, 98, 100, 102, 106, 145, 198, 202, 211
 - physical .. 25, 63, 64, 96, 97, 98, 111, 127, 128, 198, 199, 200, 201, 202, 203, 204, 205, 206
 - random access 18, 19, 22, 25, 29, 37, 40, 51, 64, 98, 164, 168, 198, 201, 211, 212, 216
 - read-only 18, 19, 22, 25, 29, 30, 31, 32, 37, 40, 50, 51, 64, 98, 99, 112, 164, 168, 182, 198, 201, 202, 208, 210, 212, 213
 - reflection 211
 - reserve 63, 65
 - root . 22, 63, 66, 96, 97, 98, 100, 102, 106, 126, 134, 135, 136, 138, 139, 140, 145, 169, 198, 199, 200, 201, 203, 204, 205, 206, 208, 211, 212
 - memory management unit (MMU) 22, 96, 97, 98, 140, 198, 199, 200, 201, 202, 211
 - Memory options 40, 51, 63
 - Logical 65
 - Physical 63
 - Reserve 65
 - Root Reserve 66
 - XMem Reserve 66
 - memory options 58
 - menu commands 41, 42
 - menus
 - COMPILE 41, 50, 51
 - EDIT 36, 41, 46, 47, 48, 49
 - FILE 41, 42, 43, 44, 45
 - HELP 37, 38, 41, 71, 72, 73, 74
 - INSPECT 34, 41, 54, 55, 56, 69
 - OPTIONS 30, 40, 41, 50, 51, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67, 71, 94, 202
 - RUN 34, 35, 41, 52, 53, 54
 - system 41
 - WINDOW 41, 65, 68, 69, 70, 71
 - Message window 69
 - message window 49, 51, 68, 69
 - Microsoft Windows Users Guide 40, 42
 - minimized windows 69
 - minus operator (-) 118
 - MMU (memory management unit) 22, 96, 97, 98, 140, 198, 199, 200, 201, 202, 211
 - mod-assign operator (%=) 123

Mode 0 interrupts	164	Next error <CTRL-N>	46, 49
Mode 2 interrupts	164	No Background TX	67
modes		nodebug . 52, 53, 56, 62, 104, 124,	
debug	29, 49, 52, 190	130, 188, 189, 208	
debug (same as run mode)	32	nonmaskable interrupts ...	163, 164
edit	36, 40, 46, 49, 54	norst	104
preview	43	NOT	
run	29, 32, 49, 52	logical operator (!)	117
modules	77, 82, 83, 102	not equal operator (!=)	121
body	82, 83, 195	nouseix	23, 105, 136, 190
example	83	NULL	105
header	82, 83, 102, 195	NULL device	50
key	82, 194	Null Device (Bit Bucket)	32
library	82, 194	numbers	112
modulus operator (%)	120	O	
mouse	40	Object File Option	60
moving		octal integer values	113
to beginning of file	41	offsets in assembly language .	134,
to beginning of line	41	137, 139, 140	
to end of file	41	online help	37, 71, 72, 194
to end of line	41	Open	42
multi-line assembly code macro		opening an existing file	42
134		operators	115, 118
multiplication operator (*)	118	, comma	123
multiply-assign operator (*=) ..	123	! logical NOT	117
multitasking ... 148, 149, 150, 151,		!= not equal	121
152, 153, 154, 155, 156, 157,		# (macros)	84, 85
158, 159, 160		## (macros)	84, 85
cooperative	21	% modulus	120
multithreaded costatements ...	158,	%= assign	123
159, 160		& address	119
multithreaded systems	148	& bitwise AND	119
N		&& logical AND	122
named		&= assign	123
constants	18, 112	() parentheses	116
costatements	150	(type) cast	118
names	112	* indirection	118
#define	112	* multiplication	118
New	42	*= assign	123
new libraries		+ plus	118
GESUPRT.LIB	222	+ unary plus	118
SF1000_Z.LIB	222	++ increment	117
newbbr	213	+= assign	122

-> right arrow	116	editor	58
. dot	116	memory	58, 63
/ division	119	serial	58, 67
/= assign	123	OPTIONS menu 30, 40, 41, 50, 51,	
< less than	120	58, 59, 60, 61, 62, 63, 64, 65,	
<< shift left	120	66, 67, 71, 94, 202	
<<= assign	123	OR assign operator (!=)	123
<= less than or equal	120	OR logical operator (!)	122
= assign	122	output	
== equal	121	analog	16
> greater than	121	digital	16
>= greater than or equal	121	high-current	16
>> shift right	120	Output Generation	60
>>= assign	123	Create HEX File Also	60
? : conditional	122	Zero Time Stamp	60
[] array indices	116		
^ bitwise exclusive OR	121	P	
^= assign	123	PageDown key	40
logical OR	122	PageUp key	40
bitwise inclusive OR	122	parallel processes	18, 21
= assign	123	passing arguments 18, 23, 95, 135,	
~ bitwise complement	117	136, 141, 142, 143	
assignment	122, 123	passwords in DLM	170, 172
associativity	115, 124	Paste	47
binary	115	pasting text <CTRL-V>	46, 47
comma	123	PC	16, 52, 67, 134
conditional	122	COM port	29, 169
decrement	117	serial port	29
in assembly language	133	PC (program counter)	184
logical	117, 122	physical address	97, 199
minus	118	physical memory 25, 63, 64, 96, 97,	
precedence	115, 124	98, 111, 127, 128, 198, 199,	
relational	120, 121	200, 201, 202, 203, 204, 205,	
sizeof	119	206	
unary	115	PLU	180
unary minus	118	cautions and disclaimers	176,
optimization	188	179	
Optimize For (size or speed)	62	common problems	180
Options		Dump	177
Display	66	Generic upload	179
options		Help	177
compiler	58, 59, 60, 61, 62	Manual software reset	180
debugger	58, 62	Open	180
display	58, 66	program mode	178

Reset Target 178, 180
 Resetting Target 178
 Reuse of programming port 180
 Run after upload 179, 180
 run mode 180
 Serial Options 177
 software reset 178, 180
 Upload 179
 upload to physical address .. 179
 Uploading 177
 use of #INT_VEC 180
 use of reload_vec 180
 PLU (Program Loader Utility) 176,
 222
 PRGLOADR.EXE 176
 plus operator (+) 118
 pointer checking 94, 188, 208
 pointers 94, 95, 114, 116, 118, 119,
 127, 160
 uninitialized 94
 polling 33, 52, 53
 pop 105, 125, 126
 ports
 serial 24, 67
 positioning text 49
 post-decrement operator (--) ... 117
 post-increment operator (++) .. 117
 power fail 25, 26
 power failure 43, 105, 184, 216,
 217
 pre-decrement operator (--) 117
 pre-increment operator (++) ... 117
 precedence 115, 124
 preprocessor 208
 preserving registers . 142, 143, 145
 preview mode 43
 Previous error <CTRL-P> .. 46, 49
 PRGLOADR.EXE 176
 primary register 131, 135, 141, 142,
 189
 primitive data types 91, 92
 Print 44, 45
 Properties 44, 45
 Range
 All 44
 Pages 44
 Selection 44
 Print Preview 43, 44
 Print Setup 45
 printf ... 33, 52, 53, 62, 69, 81, 114
 ProComm 168, 169, 172
 program
 example 81
 program counter (PC) 184
 program flow 86, 87, 88, 89, 90
 program group 14
 Dynamic C 14, 28
 Program Loader Utility (PLU) 176,
 222
 PRGLOADR.EXE 176
 programmable ROM 18, 19, 22, 25,
 29, 30, 31, 32, 37, 40, 50, 51,
 64, 98, 99, 112, 164, 168, 182,
 198, 201, 202, 208, 210, 212,
 213
 programming
 real-time 17, 21
 promotion 116
 protected variables 17, 18, 22, 105,
 185, 189, 216
 prototypes 195
 function ... 23, 38, 79, 80, 82, 83
 in headers 82, 194
 PSW_TIMEOUT 170
 punctuation 127
 push 105, 125, 126
Q
 quitting Dynamic C <ALT-F4> 42,
 45
R
 RAM .. 64, 98, 164, 198, 201, 211,
 212, 213
 static . 18, 19, 22, 25, 29, 37, 40,
 51, 64, 98, 164, 168, 198, 201,
 211, 212, 216

read-only memory .. 18, 19, 22, 25, 29, 30, 31, 32, 37, 40, 50, 51, 64, 98, 99, 112, 164, 168, 182, 198, 201, 202, 208, 210, 212, 213
 readireg 163
 real-time
 extensions 25
 kernel (RTK) 17, 25, 190
 library 25
 operations 25
 programming 17, 21
 systems 148
 redoing changes
 <ALT-SHIFT-backspace> 46
 reentrant functions 188, 190
 Register window 70
 registers 18, 105, 136, 138, 139, 189, 190
 set 131
 snapshots 70
 variables 94
 window 16, 33, 68, 69, 70
 regular reset 25
 relational operators 120, 121
 Release_Notes.txt 222
 reload_vec 180
 remote download 33, 168, 169, 170, 171, 172, 173, 210
 Remote Target Information 222
 remote target information (RTI) file
 29, 30, 31, 32, 40, 50, 51, 76, 172, 210
 Replace 48
 Change All 48
 From cursor 48
 No prompt 48
 Reverse 48
 Selection only 48
 Replace <F6> 41, 48
 replacing text 46, 48, 49
 Report DLM Parameters 169
 reserve memory 63, 65
 reset. 25, 26, 36, 37, 216, 217, 218
 hardware 25
 power fail 25
 regular 25
 software 54
 super 216
 watchdog 25
 Reset Functions 26
 _sysIsPwrFail 26
 _sysIsSuperReset 26
 _sysIsWDTO 26
 reset generation 218
 Reset program <CTRL-F2> 52, 54
 Reset target <CTRL-Y> 52, 54
 resetting program 54
 restarting 25, 26
 program 54
 target controller 54
 ret 105, 106, 141, 144, 162, 163
 reti 105, 106, 144, 162, 163
 retn 105, 106, 144, 162, 163
 return 106, 109, 141, 162
 return address 136, 140
 returning to edit mode 36
 reverse searching 47, 48, 49
 ROM 52, 53, 64, 98, 164, 182, 198, 201, 208, 210, 212, 213
 programmable 18, 19, 22, 25, 29, 30, 31, 32, 37, 40, 50, 51, 64, 98, 99, 112, 164, 168, 182, 198, 201, 202, 208, 210, 212, 213
 root 66, 96, 106, 126, 199, 200, 201, 203, 204
 code 211
 memory .. 22, 63, 66, 96, 97, 98, 100, 102, 106, 126, 134, 135, 136, 138, 139, 140, 145, 169, 198, 199, 200, 201, 203, 204, 205, 206, 208, 211, 212
 reserve 66
 root2xmem 205
 RS-232 communication 16, 24
 RS-485 communication 16, 24

rst 028h	52	Search for Help on.....	72
RST 28	208, 212	searching for text	47, 48, 49
RST 28H	188	searching in reverse	47, 48, 49
RTI (remote target information) file		segchain	19, 107
29, 30, 31, 32, 40, 50, 51, 76,		selecting	
172		COMPILE menu <ALT-C> ..	50
RTI files	210	EDIT menu <ALT-E>	46
RTK (real-time kernel) 17, 25, 190		FILE menu <ALT-F>	41, 42
RTK.LIB	148	HELP menu <ALT-H>	71
Run <F9>	34, 52	INSPECT menu <ALT-I>	54
RUN menu ..	34, 35, 41, 52, 53, 54	OPTIONS menu <ALT-O> ...	58
run mode	29, 32, 49, 52	RUN menu <ALT-R>	52
Run w/ No Polling <ALT-F9> .	34,	WINDOW menu <ALT-W> .	68
52		selecting text	
run-time		to beginning of file	41
checking	59	to end of file	41
errors		to end of line	41
and standalone programs	182	to start of line	41
stack size	65	serial communication ...	16, 24, 67,
running		163	
a program	52	clocked	24
in polling mode	52	serial options	51, 58, 67
standalone	16	serial port	29, 67
with no polling	52	ASCII	143
		PC	169
S		Set Password	169, 170
sample programs 76, 128, 169, 172,		setireg	163
206		setjmp	184
basic C constructs	81	setting	
SAMPLES subdirectory ..	76, 128,	interrupt vectors	164
206		jump vectors	164
SAMPLES\AASC subdirectory		SF1000 series	222
169, 172		SF1000_Z.LIB	222
Save	43	shared variables ...	17, 18, 22, 107,
save and restore registers	184	189	
Save as	43	shift left operator (<<)	120
Save Environment	68	shift right operator (>>)	120
saving a file	42, 43	SHIFT-F5	49
saving and restoring registers .	162	shift-left-assign operator (<<=)	123
save as a new file	42	shift-right-assign operator (>>=)	123
scroll bars	40	123	
scrolling	40, 41, 70	short	107
search		shortcuts	
HELP	72	keyboard	72

Show Tool Bar	67	Stack window	70
single stepping	34, 56, 130, 190	standalone	
in assembly language	188	assembly code	135
with descent <F7>	53	programs	16
without descent <F8>	53	standard error handler	182
size	107	startup code	212
sizeof.....	107, 119	statements	78
skipping to next loop pass	88	static	
soft break points	35, 52, 53, 183	RAM 18, 19, 22, 25, 29, 37, 40,	
software		51, 64, 98, 164, 168, 198, 201,	
errors	25	211, 212, 216	
libraries 23, 76, 77, 82, 102, 127,		variables	18, 19, 22, 24, 108,
220		125, 134, 135, 136, 138, 139,	
reset	54	188, 189, 208, 212	
software reset		status register (F)	70
Reset Target	178	STDIO window 16, 33, 62, 68, 69,	
source window	69	182, 183	
SP (stack pointer) 18, 23, 127, 131,		Step over <F8>	34, 52, 53
136, 141, 142, 145, 184, 203,		Stop <CTRL-Z>	36, 52
212		stop bits	67
special characters	114	stopping a running program	52
special symbols		storage class	78, 100, 105, 108,
in assembly language	133	125, 136, 189	
speed	108	auto	94
SRTK.LIB	148	default	18, 19, 24, 208
stack . 95, 100, 105, 135, 136, 137,		register	94
138, 140, 141, 142, 143, 144,		static	94
145, 184, 190		strcpy	74, 114
checking	188, 189, 190, 208	string	205
frame .. 135, 136, 138, 140, 141,		STRINGLIB	112
142, 143, 145		strings	111, 113, 114, 125, 128
frame reference point	141	extended memory	205
frame reference pointer 23, 105,		functions	113
139, 140, 143, 190, 203		in C	91
limit	212	in extended memory .. 128, 204,	
pointer (SP) .. 18, 127, 131, 136,		205, 206	
141, 142, 145, 184		terminating null byte	113
size		struct 78, 93, 95, 108, 116, 134,	
auxiliary	65	136, 141, 142	
run-time	65	structures .. 93, 116, 134, 136, 141,	
snapshots	71	142	
verification	65	return space	136, 141, 142
window	16, 33, 69, 70		
stack pointer (SP).....	203, 212		

subdirectories
 SAMPLES 76, 128, 206
 SAMPLES\AASC 169, 172
 subfunc 18, 24, 108, 189, 190
 subfunctions 18, 24, 108, 189, 190
 subscripts
 array 92
 subtract assign operator (=) ... 122
 super reset 21, 25, 26, 216, 218
 support files 77
 suspend 23
 suspended costatements . 148, 149,
 151, 152, 153, 154
 switch 90, 100, 101, 102, 109, 203
 breaking out of 88
 switching to edit mode .. 36, 46, 49
 symbolic constant 125
 Sync. Bkgnd TX 67
 syntax
 checking 32
 costatements 150
 SYS.LIB 220
 sysForceReset 26, 218
 sysForceSupRst 26, 218
 sysIsPwrFail 217
 sysIsSuperReset 217
 sysIsWDTO 217
 sysSupRstChain 26, 216, 220

T

table of operator precedence ... 124
 Target Communication 222
 targetless compilation .. 29, 30, 31,
 32, 50, 51
 text cursor 40, 74
 Tile Horizontally 69
 tiling windows 68, 69
 timer 16
 programmable 24
 watchdog 25, 184, 185, 217, 218
 Toggle break point <F2> 52, 53
 Toggle hard break point <ALT-F2>
 52, 53

Toggle interrupt <CTRL-I> 35, 52,
 53
 Toggle polling <CTRL-O> 34, 52,
 53
 toolbar 67
 print preview 44
 topical help 37
 Trace into <F7> 34, 52, 53
 type
 casting 116, 118
 checking 32, 61, 79
 conversion 116, 118
 definitions 80
 typedef 80, 109
 types
 function 78

U

unary
 minus operator (-) 118
 operators 115
 plus operator (+) 118
 unbalanced stack 145
 undoing changes <ALT-Back-
 space> 46
 uninitialized
 data 211
 pointers 94
 union 78, 93, 109, 116
 unpreserved registers 142, 143, 145
 unsigned 110
 unsigned int 128, 205
 unsigned integer values 113
 unsigned long 128, 204, 205
 untitled files 43
 Update Watch window <CTRL-U>
 36, 54, 56
 uplc_init
 initialize CoData structures. 150
 useix 23, 110, 138, 190
 using
 assembly language 21
 Dynamic C 28

V

variables
 global 94
VdInit 152
 initialize CoData structures. 150
vertical tiling 68
void 78

W

waitfor 102, 110, 149, 150, 151,
 152, 154, 157, 158
warning reports 61
warnings 204
watch
 dialog 36, 55
 expressions 35, 55, 56, 69
 line
 immediate evaluation 36
 list 56
 for repeated evaluation 36
 window .. 16, 33, 35, 36, 54, 55,
 68, 69
 adding items 54, 55
 clearing 55
 deleting items 54, 55
 updating 56
watch expression
 Add to top 55
 Del from top 55
 Evaluate 55
WATCH menu 36
Watch window 69
watchdog 26
 reset 25
 timeout 25, 26
watchdog timer 25, 184, 185, 217,
 218
while ... 78, 86, 100, 101, 110, 203
WINDOW menu 41, 65, 68, 69, 70,
 71
windows 14, 28, 40, 68
 assembly 16, 33, 34, 68, 69, 70,
 130

 cascaded 68, 69
 debugging 33, 34
 information 65, 68, 69, 71
 message 68, 69
 minimized 69
 program group 14
 register 16, 33, 68, 69, 70
 Run... 14
 stack 16, 33, 34, 68, 69, 70
 STDIO . 16, 33, 62, 68, 69, 182,
 183
 tiled horizontally 68, 69
 tiled vertically 68
 watch 16, 33, 35, 36, 54, 55, 56,
 68, 69

X

xdata 22, 111, 128, 204, 205
XDATA.C 128, 206
xgetfloat 22
xgetong 205
xmem 66, 96, 97, 98, 111, 126,
 139, 199, 200, 201, 202, 203
XMEM reserve 66
XMEM.LIB 22, 127
xmem2root 205
xmemok 111, 124, 139, 204
XMODEM 168, 170
XOR assign operator (^=) 123
xstring 22, 111, 128, 204, 205
xstrlen 22, 205

Y

yield 111, 149, 150, 153, 154

Z

Z180... 16, 17, 21, 24, 70, 96, 131,
 164, 188, 189, 198, 199, 200,
 201
Z80 24, 163, 164
Zero Time Stamp 60, 222
Zilog 16, 24, 163



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

