Dynamic C®

For Rabbit Semiconductor Microprocessors
Integrated C Development System

User's Manual

040313 • 019-0125-B

This manual (or an even more up-to-date revision) is available for free download at the Z-World website: www.zworld.com

Table of Contents

1. Installing Dynamic C

Insert the installation disk or CD in the appropriate disk drive on your PC. The installation should begin automatically. If it doesn't, issue the Windows "Run..." command and type the following command.

```
‹disk›:\SETUP
```

The installation program will begin and guide you through the installation process.

1.1 Requirements

Your IBM-compatible PC should have at least one free COM port and be running one of the following.

- Windows 95
- Windows 98
- Windows 2000
- Windows Me
- Windows NT

1.2 Assumptions

It is assumed that the reader has a working knowledge of:

- the basics of operating a software program and editing files under Windows on a PC.
- programming in a high-level language.
- assembly language and architecture for controllers.

For a full treatment of C, refer to one or both of the following texts:

- *The C Programming Language* by Kernighan and Ritchie (published by Prentice-Hall).
- *C: A Reference Manual* by Harbison and Steel (published by Prentice-Hall).

2. Introduction to Dynamic C

Dynamic C is an integrated development system for writing embedded software. It is designed for use with Z-World controllers and other controllers based on the Rabbit microprocessor. The Rabbit 2000 and the Rabbit 3000 are high-performance 8-bit microprocessors that can handle C language applications of approximately 50,000 C+ statements or 1 MB.

2.1 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, full-featured, text editor. Dynamic CPrograms can be executed and debugged interactively at the source-code or machine-code level. Pull-down menus and keyboard shortcuts for most commands make Dynamic C easy to use.

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 together.

Debugging under Dynamic C includes the ability to use `printf` commands, watch expressions, breakpoints and other advanced debugging features. Watch expressions can be used to compute C expressions involving the target's program variables or functions. Watch expressions can be evaluated while stopped at a breakpoint or while the target is running its program.

Dynamic C provides extensions to the C language (such as *shared* and *protected* variables, costatements and cofunctions) that support real-world embedded system development. Dynamic C supports cooperative and preemptive multi-tasking.

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.

2.1.1 Speed

Dynamic C compiles directly to memory. Functions and libraries are compiled and linked and downloaded on-the-fly. On a fast PC, Dynamic C might load 30,000 bytes of code in 5 seconds at a baud rate of 115,200 bps.

2.2 Dynamic C Enhancements and Differences

Dynamic C differs from a traditional C programming system running on a PC or under UNIX. The reason? To be better help customers write the most reliable embedded control software possible. It is not possible to use standard C in an embedded environment without making adaptations. Standard C makes many assumptions that do not apply to embedded systems. For example, standard C implicitly assumes that an operating system is present and that a program starts with a clean slate, whereas embedded systems may have battery-backed memory and may retain data through power cycles. Z-World has extended the C language in a number of areas.

2.2.1 Dynamic C Enhancements

Many enhancements have been added to Dynamic C. Some of these are listed below.

- Function chaining, a concept unique to Dynamic C, allows special segments of code to be embedded 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.

- Costatements allow concurrent parallel processes to be simulated in a single program.

- Cofunctions allow cooperative processes to be simulated in a single program.

- Slice statements allow preemptive processes in a single program.

- Dynamic C supports embedded assembly code and stand-alone assembly code.

- Dynamic C has shared and protected keywords that help protect data shared between different contexts or stored in battery-backed memory.

- Dynamic C has a set of features that allow the programmer to make fullest use of extended memory. Dynamic C supports the 1 MB address space of the microprocessor. The address space is segmented by a memory management unit (MMU). 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 KB physical address space). The keyword `xmem` selects extended memory, which means anywhere in the 1024 KB or 1 MB code space. `root` and `xmem` are semantically meaningful in function prototypes and more efficient code is generated when they are used. Their use must match between the prototype and the function definition. The directive `#memmap` allows further control. See “Memory Management” on page 107, for further details on memory.

2.2.2 Dynamic C Differences

The main differences in Dynamic C are summarized here and discussed in detail in chapters “Language” on page 15 and “Keywords” on page 143.

- If a variable is explicitly initialized in a declaration (e.g., `int x = 0;`), it is stored in flash memory (EEPROM) and cannot be changed by an assignment statement. Such a declaration will generate a warning that may be suppressed using the `const` keyword:

  ```
  const int x = 0
  ```

  To initialize static variables in Static RAM (SRAM) use `#GLOBAL_INIT` sections. Note that other C compilers will automatically initialize all static variables to zero that are not explicitly initialized before entering the main function. Dynamic C programs do not do this because in an embedded system you may wish to preserve the data in battery-backed RAM on reset

- The numerous include files found in typical C programs are not used because Dynamic C has a library system that automatically provides function prototypes and similar header information to the compiler before the user’s program is compiled. This is done via the `#use` directive. This is an important topic for users who are writing their own libraries. Those users should refer to the Modules section of the language chapter. It is important to note that the `#use` directive is a replacement for the `#include` directive, and the `#include` directive is not supported.

- When declaring pointers to functions, arguments should not be used in the declaration. Arguments may be used when calling functions indirectly via pointer, but the compiler will not check the argument list in the call for correctness.

- Bit fields are not supported.

- Separate compilation of different parts of the program is not supported or needed.

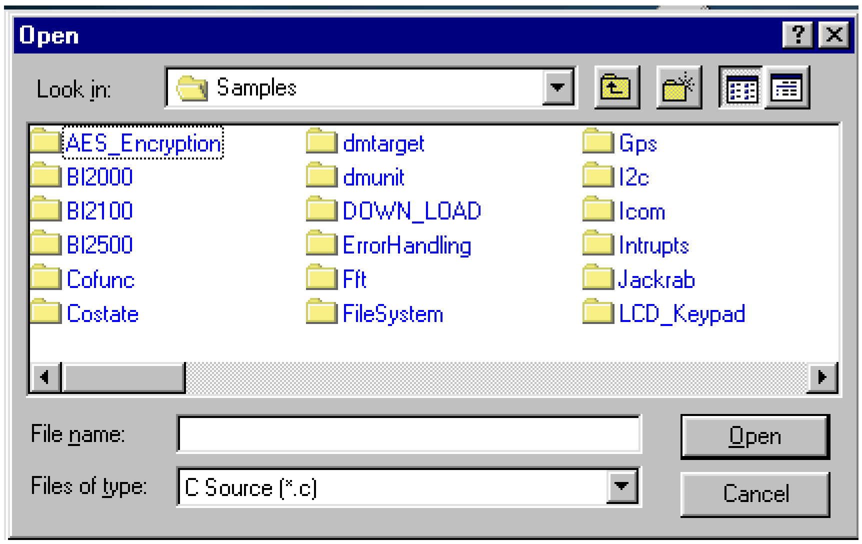
2.3 Dynamic C Differences Between Rabbit and Z180

A major difference in the way Dynamic C interacts with a Rabbit-based board compared to a Z180 or 386EX board is that Dynamic C expects no BIOS kernel to be present on the target when it starts up. Dynamic C stores the BIOS kernel as a C source file. Dynamic C compiles and loads it to the Rabbit target when it starts. This is accomplished using the Rabbit CPU's bootstrap mode and a special programming cable provided in all Rabbit product development kits. This method has numerous advantages.

- A socketed flash is no longer needed. BIOS updates can be made without a flash-EPROM burner since Dynamic C can communicate with a target that has a blank flash EPROM. Blank flash EPROM can be surface-mounted onto boards, reducing manufacturing costs for both Z-World and other board developers. BIOS updates can then be made available on the Web.
- Advanced users can see and modify the BIOS kernel directly.
- Board developers can design Dynamic C compatible boards around the Rabbit CPU by simply following a few simple design guidelines and using a "skeleton" BIOS provided by Z-World.
- A major feature is the ability to program and debug over the Internet or local Ethernet. This requires the use of a RabbitLink board, available alone or as an option with Rabbit-based development kits.

3. Quick Tutorial

Sample programs are provided in the Dynamic C `Samples` folder similar to the one shown below.



The subfolders contain sample programs that illustrate the use of the various Dynamic C libraries. E.g., the subfolders “Cofunc” and “Costate” have sample programs illustrating the use of `COFUNC.LIB` and `COSTATE.LIB`, libraries that support cooperative multitasking using Dynamic C language extensions. The sample program `Pong.c` demonstrates output to the STDIO window.

Read the comment block at the top of the sample program for a general description of its purpose. Further details are provided in this comment block when needed. Comments are also in the source code. The sample program documentation is provided by the software engineers and is a rich source of information.

3.1 Run DEMO1.C

This sample program will be used to illustrate some of the functions of Dynamic C. Open the file `Samples/DEMO1.C` using the File menu or the keyboard shortcut <Ctrl+O>. The program will appear in a window, as shown in Figure 1 below (minus some comments). Use the mouse to place the cursor on the function name `printf` in the program and press <Ctrl+H>. This brings up a documentation box for the function `printf`. You can do this with all functions in the Dynamic C libraries, including libraries you write yourself.

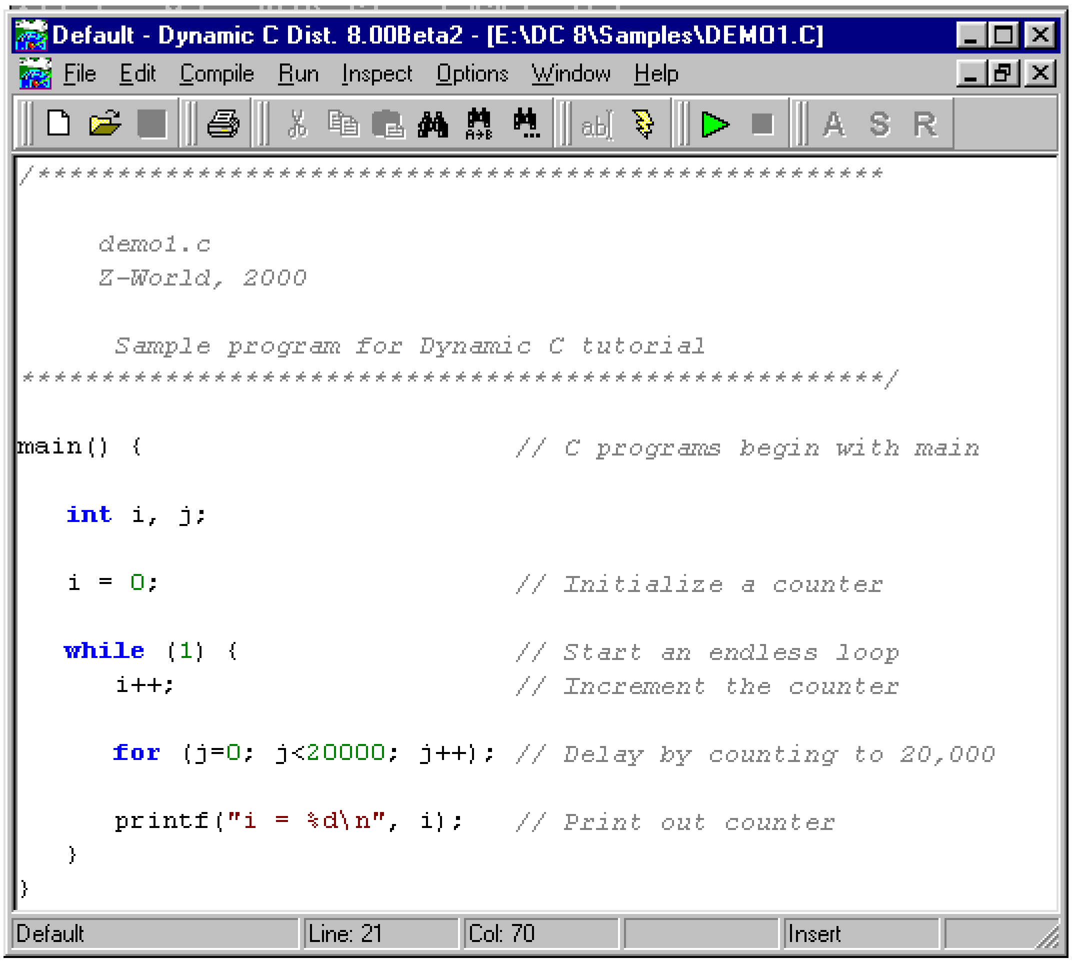
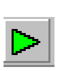


Figure 3-1   Sample Program DEMO1.C



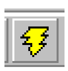
To run `DEMO1.C` compile it using the Compile menu, and then run it by selecting Run in the Run menu. (The keyboard shortcut <F9> will compile and run the program. You may also use the green triangle toolbar button as a substitute for <F9>.)

The value of the counter should be printed repeatedly to the Stdio window if everything went well. If this doesn't work, review the following points:

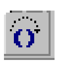
- The target should be ready, indicated by the message "BIOS successfully compiled..." If you did not receive this message or you get a communication error, recompile the BIOS by typing <Ctrl+Y> or select Reset Target / Compile BIOS from the Compile menu.

- A message reports “No Rabbit Processor Detected” in cases where the wall transformer is not connected or not plugged in.
- The programming cable must be connected to the controller. (The colored wire on the programming cable is closest to pin 1 on the programming header on the controller). The other end of the programming cable must be connected to the PC serial port. The COM port specified in the Communications dialog box must be the same as the one the programming cable is connected to. (The Communications dialog box is accessed via the Communications tab of the Options | Project Options menu.)
- To check if you have the correct serial port, press <Ctrl+Y>. If the “BIOS successfully compiled …” message does not display, choose a different serial port in the Communications dialog box until you find the serial port you are plugged into. Don’t change anything in this menu except the COM number. The baud rate should be 115,200 bps and the stop bits should be 1.

3.1.1 Single Stepping



To experiment with single stepping, we will first compile `DEMO1.C` to the target without running it. This can be done by clicking the compile button on the task bar. This is the same as pressing <F5>. Both of this actions will compile according to the setting of “Default Compile Mode.” (See “Default Compile Mode” in Chapter 14, for how to set this parameter.) Alternatively you may select Compile | Compile to Target from the main menu.



After the program compiles a highlighted character (green) will appear at the first executable statement of the program. Press the <F8> key to single step (or use the toolbar button). Each time the <F8> key is pressed, the cursor will advance one statement. When you get to the statement: `for(j=0, j< ...`, it becomes impractical to single step further because you would have to press <F8> thousands of times. We will use this statement to illustrate watch expressions.

3.1.2 Watch Expression

Watch expressions may only be added, deleted or updated while in run mode. To add a watch expression click on the toolbar button pictured here, or press <Ctrl+W> or choose Add Watch from the Inspect menu. The “Add Watch Expression” popup box will appear. Type the lower case letter “j” and click on either Add or OK. The former keeps the popup box open, the latter closes it. Either way the “Watches” window appears. This is where information on watch expressions will be displayed. Now continue single stepping. Each time you do, the watch expression (`j`) will be evaluated and printed in the “Watches” window. Note how the value of “j” advances when the statement `j++` is executed.

3.1.3 Breakpoint

Move the cursor to the start of the statement:

```
for (j=0; j<20000; j++);
```

To set a breakpoint on this statement, press <F2> or select Toggle Breakpoint from the Run menu. A red highlight appears on the first character of the statement. To get the program running at full speed, press <F9>. The program will advance until it hits the breakpoint. The breakpoint will start flashing both red and green colors.

To remove the breakpoint, press <F2> or select Toggle Breakpoint on the Run menu. To continue program execution, press <F9>. Now the counter should be printing out regularly in the Stdio window.

You can set breakpoints while the program is running by positioning the cursor to a statement and using the <F2> key. If the execution thread hits the breakpoint, a breakpoint will take place. You can toggle the breakpoint with the <F2> key and continue execution with the <F9> key.

3.1.4 Editing the Program

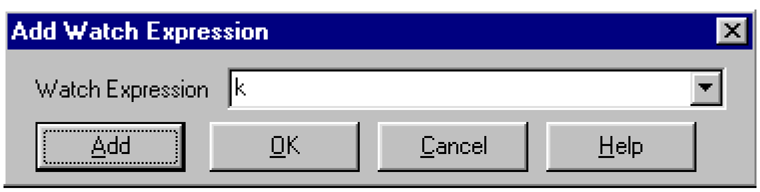
Press <F4>to put Dynamic C into edit mode. Use the Save as choice on the File menu to save the file with a new name so as not to change the original demo program. Save the file as `MYTEST.C`. Now change the number 20000 in the `for` statement to 10000. Then use the <F9> key to recompile and run the program. The counter displays twice as quickly as before because you reduced the value in the delay loop.

3.2 Run DEMO2.C

Go back to edit mode and open the program `DEMO2.C`. This program is the same as the first program, except that a variable `k` has been added along with a statement to increment `k` by the value of `i` each time around the endless loop. Compile and run `DEMO2.C`.

3.2.1 Watching Variables Dynamically

Press <Ctrl+W> to open the “Add Watch Expression” popup box.



Type “k” in the text entry box, then click OK (or Add) to add the expression `k` to the top of the list of watch expressions. Now press <Ctrl+U>, the keyboard shortcut for updating the watch window. Each time you press <Ctrl+U>, you will see the current value of `k`.

Add another expression to the watch window:

```
k*5
```

Then press <Ctrl+U> several times to observe the watch expressions `k` and `k*5`.

The evaluation of an expression that will result in a run-time exception will ignore the exception, and the result of the expression will be undefined.

3.3 Run DEMO3.C

The example below, sample program `DEMO3.C`, uses costatements. A costatement is a way to perform a sequence of operations that involve pauses or waits for some external event to take place.

3.3.1 Cooperative Multitasking

Cooperative multitasking is a way to perform several different tasks at virtually the same time. An example would be to step a machine through a sequence of tasks and at the same time carry on a dialog with the operator via a keyboard interface. Each separate task voluntarily surrenders its compute time when it does not need to perform any more immediate activity. In preemptive multitasking control is forcibly removed from the task via an interrupt.

Dynamic C has language extensions to support both types of multitasking. For cooperative multitasking the language extensions are *costatements* and *cofunctions*. Preemptive multitasking is accomplished with *slicing* or by using the µC/OS-II real-time kernel that comes with Dynamic C Premier.

3.3.1.1 Advantages of Cooperative Multitasking

Unlike preemptive multitasking, in cooperative multitasking variables can be shared between different tasks without taking elaborate precautions. Cooperative multitasking also takes advantage of the natural delays that occur in most tasks to more efficiently use the available processor time.

The `DEMO3.C` sample program has two independent tasks. The first task prints out a message to STDIO once per second. The second task watches to see if the keyboard has been pressed and prints the entered key.

```
main() {
    int secs;                                 // seconds counter
    secs = 0;                                 // initialize counter
(1) while (1) {                               // endless loop

// First task will print the seconds elapsed.
(2)   costate {
        secs++;                               // increment counter
(3)     waitfor( DelayMs(1000) );             // wait one second
        printf("%d seconds\n", secs);         // print elapsed seconds
(4)   }
// Second task will check if any keys have been pressed.
      costate {
(5)     if ( !kbhit() )  abort;               // key been pressed?
        printf("  key pressed = %c\n", getchar() );
      }

(6) }          // end of while loop
}              // end of main
```

The numbers in the left margin are reference indicators and not part of the code. Load and run the program. The elapsed time is printed to the STDIO window once per second. Push several keys and note how they are reported.

The elapsed time message is printed by the costatement starting at the line marked (2). Costatements need to be executed regularly, often at least every 25 ms. To accomplish this, the costatements are enclosed in a `while` loop. The `while` loop starts at (1) and ends at (6). The statement at (3) waits for a time delay, in this case 1000 ms (one second). The costatement executes each pass through the `while` loop. When a `waitfor` condition is encountered the first time, the current value of `MS_TIMER` is saved and then on each subsequent pass the saved value is compared to the current value. If a `waitfor` condition is not encountered, then a jump is made to the end of the costatement (4), and on the next pass of the loop, when the execution thread reaches the beginning of the costatement, execution passes directly to the `waitfor` statement. Once 1000 ms has passed, the statement after the `waitfor` is executed. A costatement can wait for a long period of time, but not use a lot of execution time. Each costatement is a little program with its own statement pointer that advances in response to conditions. On each pass through the `while` loop as few as one statement in the costatement executes, starting at the current position of the costatement's statement pointer. Consult Chapter 5 "Multitasking with Dynamic C" for more details.

The second costatement in the program checks to see if a key has been pressed and, if one has, prints out that key. The `abort` statement is illustrated at (5). If the `abort` statement is executed, the internal statement pointer is set back to the first statement in the costatement, and a jump is made to the closing brace of the costatement.

Observe the value of secs while the program is runningTo illustrate the use of snooping, use the watch window to observe `secs` while the program is running. Add the variable `secs` to the list of watch expressions, then press <Ctrl+U> repeatedly to observe as `secs` increases.

3.4 Summary of Features

This chapter provided a quick look at the interface of Dynamic C and some of the powerful options available for embedded systems programming. The following several paragraphs are a summary of what we've discussed.

Development Functions

When you load a program it appears in an editor window. You compile by clicking Compile on the task bar or from the Compile menu. The program is compiled into machine language and downloaded to the target over the serial port. The execution proceeds to the first statement of main, where it pauses, waiting to run. Press <F9> or select Run on the Run menu. If want to compile and run the program with one keystroke, use <F9>, the run command; if the program is not already compiled, the run command compiles it.

Single Stepping

This is done with the F8 key. The F7 key can also be used for single stepping. If the F7 key is used, then descent into subroutines will take place. With <F8> the subroutine is executed at full speed when the statement that calls it is stepped over.

Setting Breakpoints

The F2 key is used to toggle a breakpoint at the cursor position if the program has already been compiled. You can set a breakpoint if the program is paused at a breakpoint. You can also set a breakpoint in a program that is running at full speed. This will cause the program to break if the execution thread hits your breakpoint.

Watch Expressions

A watch expression is a C expression that is evaluated on command in the watch window. An expression is basically any type of C statement that can include operators, variables and function calls, but not statements that require multiple lines such as `for` or `switch`. You can have a list of watch expressions in the “Watches” window. If you are single stepping, then they are all evaluated on each step. You can also command the watch expression to be evaluated by using the <Ctrl+U> command. When a watch expression is evaluated at a breakpoint, it is evaluated as if the statement was at the beginning of the function where you are single stepping.

Costatements

A costatement is a Dynamic C extension that allows cooperative multitasking to be programmed by the user. Keywords, like `abort` and `waitfor`, are available to control multitasking operation from within costatements.

4. Language

Dynamic C is based on the C language. The programmer is expected to know programming methodologies and the basic principles of the C language. Dynamic C has its own set of libraries, which include user-callable functions. Please see the *Dynamic C Function Reference Manual* for detailed descriptions of these API functions. Dynamic C libraries are in source code, allowing the creation of customized libraries.

Before starting on your application, read through the rest of this chapter to review C-language features and understand the differences between standard C and Dynamic C.

4.1 C Language Elements

A Dynamic C program is a set of files consisting of one file with a `.c` extension and the requested library files. Each file is a stream of characters that compose statements in the C language. The language has grammar and syntax, that is, rules for making statements. Syntactic elements—often called tokens—form the basic elements of the C language. Some of these elements are listed in the table below.

Table 4-1 Language Elements

Syntactic Element	Description
punctuation	Symbols used to mark beginnings and endings
names	Words used to name data and functions
numbers	Literal numeric values
strings	Literal character values enclosed in quotes
directives	Words that start with # and control compilation
keywords	Words used as instructions to Dynamic C
operators	Symbols used to perform arithmetic operations

4.2 Punctuation Tokens

Punctuation serves as boundaries in C programs. The table below lists the punctuation tokens.

Table 4-2 Punctuation Marks and Tokens

Token	Description
`:`	Terminates a statement label.
`;`	Terminates a simple statement or a `do` loop.
`,`	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. Also used for arithmetic and logical sub expressions.
`{ }`	Begins and ends a compound statement, a function body, a structure or union body, or encloses a function chain segment.
`//`	Indicates that the rest of the line is a comment and is not compiled.
`/* ... */`	Comments are nested between the `/*` and `*/` tokens.

4.3 Data

Data (variables and constants) have type, size, structure, and storage class. Basic (aka primitive) data types are shown below.

Table 4-3 Dynamic C Basic Data Types

Data Type	Description
`char`	8-bit unsigned integer. Range: 0 to 255 (0xFF)
`int`	16-bit signed integer. Range: -32,768 to +32,767
`unsigned int`	16-bit unsigned integer. 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. 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 1 implicitly. (Rabbit controllers do not have floating-point hardware.) Range: 1.18 x 10^{-38} to 3.40 x 10^{38}
`enum`	Defines a list of named integer constants. The integer constants are signed and in the range: -32,768 to +32,767.

4.3.1 Data Type Limits

The following symbolic names for the hardcoded limits of the data types are defined in `limits.h`.

```
#define CHAR_BIT            8
#define UCHAR_MAX         255
#define CHAR_MIN            0
#define CHAR_MAX          255
#define MB_LEN_MAX          1

#define SHRT_MIN       -32768
#define SHRT_MAX        32767
#define USHRT_MAX       65535

#define INT_MIN        -32767
#define INT_MAX         32767
#define UINT_MAX        65535
#define LONG_MIN  -2147483647
#define LONG_MAX   2147483647
#define ULONG_MAX  4294967295
```

4.4 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

Cleanup_the_data_now     // These names are not distinct in DC 6.19
Cleanup_the_data_later   // but are distinct in all later versions.
```

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 Section 4.5, “Macros.”

```
#define READ   10
#define WRITE  20
#define ABS     0
#define REL     1
#define READ_ABS   READ + ABS
#define READ_REL   READ + REL
```

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

```
READ_ABS = 27;      // produces compiler error
```

To accomplish the above statement, do the following:

```
#undef READ_ABS
#define READ_ABS 27
```

4.5 Macros

Macros may be defined in Dynamic C by using `#define`. 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, up to 31 characters, to a fragment of text. Dynamic C then replaces the macro name with the text fragment wherever the name appears in the program. 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 such as in the following 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 the 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 it 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 of expansion will be

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

The ## operator concatenates the preceding character sequence with the following character sequence, deleting any white space in between. For example, given the macro

```
#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);
```

4.5.1 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, 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.

4.6 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 the 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   0xFFFFFFFA
```

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   000000630
```

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`

4.7 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 C does not have a string data type, it does have character arrays that serve the purpose. C does not have string operators, such as concatenate, but library functions `strcat()` and `strncat()` are available.

Strings are multibyte objects, and as such they are always referenced by their starting address, and usually by a `char*` variable. More precisely, arrays are always passed by address. Passing a pointer to a string is the same as passing the string. Refer to Section 4.15 for more information on pointers.

The following example illustrates typical use of strings.

```
const 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
```

4.7.1 String Concatenation

Two or more string literals are concatenated when placed next to each other. For example:

```
"Rabbits" "like carrots."
```

becomes

```
"Rabbits like carrots."
```

during compilation.

If the strings are on multiple lines, the macro continuation character must be used. For example:

```
"Rabbits"\
"don't like line dancing."
```

becomes

```
"Rabbits don't like line dancing."
```

during compilation.

4.7.2 Character Constants

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, a leading zero is optional
\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 character
`\\`	backslash	`\c`	the actual character `c`
`\'`	single quote	`\"`	double quote

Examples

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

4.8 Statements

Except for comments, everything in a C program is a statement. Almost 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. The Dynamic C text editor enforces a 512 byte limit on the length of a line. Similarly, the Dynamic C compiler is only guaranteed to parse up to 512 bytes for any single C statement.

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 braces { and }. A group of statements may be single statements and/or compound statements.

Comments (the `/*...*/` kind) may occur almost anywhere, even in the middle of a statement, as long as they begin with `/*` and end with `*/`.

4.9 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. Root data arrays are limited to a total of 32,767 elements.

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

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

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

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

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

4.10 Functions

The basic unit of a C application program is a function. Most functions accept parameters (a.k.a., 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.

A function may call another function, including itself (a recursive call). 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.

4.11 Prototypes

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

- Functions that have not been compiled may be called.
- Recursive functions may be written.
- 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, ... );
```

4.12 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 variable of & pointer to type COMPLEX .
```

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

```
typedef unsigned int node;
void NodeInit( node );                  // type name is informative
void NodeInit( unsigned int );          // 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 the end of a line.*/
#define SIZE 12                         // A symbolic constant defined.
int g, h;                               // Declare global integers.
float sumSquare( int, int );            // Prototypes for
void init();                            //     functions below.

main(){                                 // Program starts here.
   float x;                             // x is local to main.
   init();                              // Call a void function.
   x = sumSquare( g, h );               // x gets sumSquare value.
   printf("x = %f",x);                  // printf is a standard function.
}
void init(){                            // Void functions do things but
   g = 10;                              //     they return no value.
   h = SIZE;                            // Here, it uses the symbolic
}                                       //     constant defined above.
float sumSquare( int a, int b ){        // Integer arguments.
   float temp;                          // Local variables.
   temp = a*a + b*b;                    // Arithmetic statement.
   return( temp );                      // Return value.
}
/* and here is the end of the program */
```

The program above 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 explained this.

4.13 Aggregate Data Types

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

4.13.1 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";
```

4.13.2 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.

4.13.3 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
```

4.13.4 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;
```

4.14 Storage Classes

Variable storage can be `auto` or `static`. 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 default storage class is `auto`, but can be changed by using `#class static`. The default storage class can be superseded by the use of the keyword `auto` or `static` in a variable declaration.

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—available anywhere in the program—but there is no keyword in C to represent this fact. Global variables always have `static` storage.

4.15 Pointers

A pointer is a variable that holds the 16-bit logical address of another variable, a structure, or a function. Dynamic C does not currently support long pointers. The indirection operator (*) is used to declare a variable as a pointer. The address operator (&) is used to set the pointer to the address of a variable.

```
int *ptr_to_i;
int i;
ptr_to_i = &i;          // set pointer equal to the address of i
i = 10:                 // assign a value to i
j = *ptr_to_i;          // this sets j equal to the value in i
```

In this example, the variable `ptr_to_i` is a pointer to an integer. The statement `j = *ptr_to_i;` references the value of the integer by the use of the asterisk. Using correct pointer terminology, the statement *dereferences* the pointer `ptr_to_i`. Then `*ptr_to_i` and `i` have identical values.

Note that `ptr_to_i` and `i` do not have the same values because `ptr_to_i` is a pointer and `i` is an `int`. Note also that `*` has two meanings (not counting its use as a multiplier in others contexts) in a variable declaration such as `int *ptr_to_i;` the `*` means that the variable will be a pointer type, and in an executable statement `j = *ptr_to_i;` means "the value stored at the address contained in `ptr_to_i`."

Pointers may point to other pointers.

```
int *ptr_to_i;
int **ptr_to_ptr_to_i;
int i,j;
ptr_to_i = &i;                  // Set pointer equal to the address of i
ptr_to_ptr_to_i = &ptr_to_i;    // Set a pointer to the pointer
                                // to the address of i
i = 10;                         // Assign a value to i
j = **ptr_to_ptr_to_i;          // This sets j equal to the value in i.
```

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

```
float f[10], *p, *q;            // an array and some ptrs
p = &f;                         // point p to array element 0
q = p+5;                        // point q to array element 5
q++;                            // point q to array element 6
p = p + q;                      // illegal!
```

Because the `float` is a 4-byte storage element, the statement `q = p+5` sets the actual value of `q` to `p+20`. The statement `q++` adds 4 to the actual value of `q`. If `f` were an array of 1-byte characters, the statement `q++` adds 1 to `q`.

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 crash.

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 in Dynamic C. Use the compiler options command in the Options menu. Pointer checking will catch attempts to dereference a pointer to unallocated memory. However, if an uninitialized pointer happens to contain the address of a memory location that the compiler has already allocated, pointer checking will not catch this logic error. Because pointer checking is a run-time option, pointer checking adds instructions to code when pointer checking is used.

4.16 Pointers to Functions, Indirect Calls

Pointers to functions may be declared. When a function is called using a pointer to it, instead of directly, we call this an *indirect* call.

The syntax for declaring a pointer to a function is different than for ordinary pointers, and Dynamic C syntax for this is slightly different than the standard C syntax. Standard syntax for a pointer to a function is:

```
returntype (*name)( [argument list] );
```

for example:

```
int (*func1)(int a, int b);
void (*func2)(char*);
```

Dynamic C doesn't recognize the argument list in function pointer declarations. The correct Dynamic syntax for the above examples would be:

```
int (*func1)();
void (*func2)();
```

You can pass arguments to functions that are called indirectly by pointers, but the compiler will not check them for correctness. The following program shows some examples of using function pointers.

```
typedef int (*fnptr)();   // create pointer to function that returns an integer

main(){
   int x,y;
   int (*fnc1)();          // declare var fnc1 as a pointer to an int function.
   fnptr  fp2;             // declare var fp2 as pointer to an int function
   fnc1 = intfunc;         // initialize fnc1 to point to intfunc()
   fp2 = intfunc;          // initialize fp2 to point to the same function.

   x = (*fnc1)(1,2);       // call intfunc() via fnc1
   y = (*fp2)(3,4);        // call intfunc() via fp2

   printf("%d\n", x);
   printf("%d\n", y);
}
int intfunc(int x, int y){
   return x+y;
}
```

4.17 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.

4.18 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.

4.18.1 Loops

A `while` loop tests a condition at the start of the loop. As long as *expression* is true (non-zero), the loop body *(some statement(s))* will execute. If *expression* is initially false (zero), the loop body will not execute. The curly braces are necessary if there is more than one statement in the loop body.

```
while( expression ){
   some statement(s)
}
```

A `do` loop tests a condition at the end of the loop. As long as *expression* is true (non-zero) the loop body *(some statement(s))* will execute. A `do` loop executes at least once before its test. Unlike other controls, the `do` loop requires a semicolon at the end.

```
do{
   some statements
}while( expression );
```

The `for` loop is more complex: it sets an initial condition (*exp1*), evaluates a terminating condition (*exp2*), and provides a stepping expression (*exp3*) that is evaluated at the end of each iteration. Each of the three expressions is optional.

```
for( exp1 ; exp2 ; exp3 ){
   some statements
}
```

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 loop initially sets `i` to 0, 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 statement(s)}
```

Here, there is no initial condition, no end condition, and no stepping expression. The loop body *(some statement(s))* continues to execute endlessly. An endless loop can also be achieved with a `while` loop. This method is slightly less efficient than the `for` loop.

```
while(1)  { some statement(s) }
```

4.18.2 Continue and Break

Two keywords are available to help in the construction of loops: `continue` and `break`.

The `continue` statement causes the program control to skip unconditionally to the next pass of the loop. In the example below, if `bad` is true, *more statements* will not execute; control will pass back to the top of the `while` loop.

```
get_char();
while( ! EOF ){
   some statements
   if( bad ) continue;
   more statements
}
```

The `break` statement causes the program control to jump unconditionally out of a loop. In the example below, if `cond_RED` is true, *more statements* will not be executed and control will pass to the next statement after the ending curly brace of the `for` loop

```
for( i=0;i<n;i++ ){
   some statements
   if( cond_RED ) break;
   more statements
}
```

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
```

4.18.3 Branching

The `goto` statement is the simplest form of a 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
```

The colon at the end of the labels is required. In general, the use of the `goto` statement is discouraged in structured programming.

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 (non-zero). The program will ignore the `if` body when the condition is false (zero).

```
if( expression ){
   some statement(s)
}
```

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.

```
if( expression ){
   some statement(s)        // if true
}else{
   some statement(s)        // if false
}
```

The fullest form of the `if` statements produces a succession of tests.

```
if( expr1 ){
   some statements
}else if( expr2 ){
   some statements
}else if( expr3 ){
   some statements
   ...
}else{
   some statements
}
```

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/else 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.

```
switch( expression ){
   case const1 :
      statements1
      break;
   case const2 :
      statements2
      break;
   case const3 :
      statements3
      break;
   ...
   default:
      statementsDEFAULT
}
```

First the `switch` *expression* is evaluated. It must have an integer value. If one of the `const`$_N$ values 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.

The colons (`:`) after `case` and `default` are required.

4.19 Function Chaining

Function chaining 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, and other kinds of tasks on request. There are two directives, `#makechain` and `#funcchain`, and one keyword, `segchain` that create and control function chains:

`#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. The following example shows how to call a function chain that is named `recover`.

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

4.20 Global Initialization

Various hardware devices in a system need to be initialized, not only by setting variables and control registers, but often by complex initialization procedures. Dynamic C provides a specific function chain, `_GLOBAL_INIT`, for this purpose. Your program can add segments to the _GLOBAL_INIT function chain, as shown in the example below.

```
long my_func( char j );

main(){
   my_func(100);
}

long my_func(char j){
   int i;
   long array[256];

   // The GLOBAL_INIT section is automatically run once when the program starts up

   #GLOBAL_INIT{
      for( i = 0; i < 100; i++ ){
         array[i] = i*i;
      }
   }

   return array[j];    // only this code runs when the function is called
}
```

The special directive `#GLOBAL_INIT{ }` tells the compiler to add the code in the block enclosed in braces to the `_GLOBAL_INIT` function chain. Any number of `#GLOBAL_INIT` sections may be used in your code. The order in which they are called is indeterminate since it depends on the order in which they were compiled.

The `_GLOBAL_INIT` function chain is always called when your program starts up, so there is nothing special to do to invoke it. In addition, it may be called explicitly at any time in an application program with the statement:

```
_GLOBAL_INIT();
```

Make this call this with caution. All costatements and cofunctions will be initialized. See “Calling _GLOBAL_INIT()” on page 67 for more information.

4.21 Libraries

Dynamic C includes many libraries—files of useful functions in source code form. They are located in the `LIB` subdirectory where Dynamic C was installed. The default library file extension is `.LIB`. Dynamic C uses functions and data from library files and compiles them with an application program that is then downloaded to a controller or saved to a `.bin` file.

An application program (the default file extension is `.c`) consists of a source code file that contains a main function (called `main`) and usually other user-defined functions. Any additional source files are considered to be libraries (though they may have a `.c` extension) and are treated as such. The minimum application program is one source file, containing only

```
main(){
}
```

Libraries (both user defined and Z-World defined) 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 below. The `#use` directive is a replacement for the `#include` directive, which is not supported in Dynamic C. Any library that is to be #used in a Dynamic C program must be listed in the file `LIB.DIR`, or another `*.DIR` file specified by the user.

(Starting with version Dynamic C 7.05, a different `*.DIR` file may be specified by the user in the **Compiler Options** dialog box to facilitate working on multiple projects.)

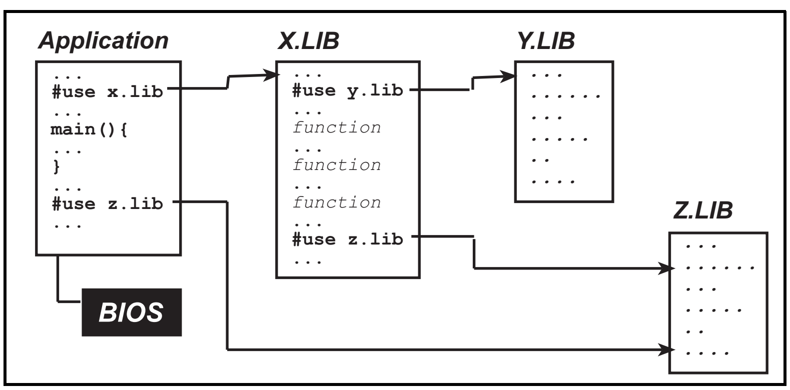


Figure 4-1 Nesting Files in Dynamic C

Most libraries needed by Dynamic C programs have a #use statement in the file `lib\default.h`.

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

4.22 Headers

The following table describes two kinds of headers used in Dynamic C libraries.

Table 4-4 Dynamic C Library Headers

Header Name	Description
Module headers	Make functions and global variables in the library known to Dynamic C.
Function Description headers	Describe functions. Function headers form the basis for function lookup help.

You may also notice some “Library Description” headers at the top of library files. These have no special meaning to Dynamic C, they are simply comment blocks.

4.23 Modules

A Dynamic C library typically contains several modules. Modules must be understood to write efficient custom libraries. Modules provide Dynamic C with the names of functions and variables within a library that may be referenced by files that have a `#use` directive for the library somewhere in the code.

Modules organize the library contents in such a way as to allow for smaller code size in the compiled application that uses the library. To create your own libraries, write modules following the guidelines in this section.

The scope of modules is global, but indeterminate compilation order makes the situation less than straightforward. Read this entire section carefully to understand module scope.

4.23.1 The Parts of a Module

A module has three parts: the key, the header, and the body. The structure of a module is:

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

A module begins with its `BeginHeader` comment and continues until either the next `BeginHeader` comment or the end of the file is encountered.

4.23.1.1 Module Key

The module key is ususally contained within the first line of the module header. It is a list of function and data names separated by commas. The list of names may continue on subsequent lines.

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

It is important to format the `BeginHeader` comment correctly, otherwise Dynamic C cannot find the contents of the module. The case of the word “beginheader” is unimportant, but it must be preceded by a forward slash, 3 astericks and one space (`/*** `). The forward slash must be the first character on the line. The `BeginHeader` comment must end with an asterick and a forward slash (`*/`).

The key tells the compiler which functions exist in the module so the compiler can exclude the module if names in the key are not referenced. Data declarations (constants, structures, unions and variables) as well as macros and function chains (both `#makechain` and `#funchain` statements) do not need to be named in the key if they are completely defined in the header, i.e, no `extern` declaration. They are fully known to the compiler by being completely defined in the module header. An important thing to remember is that variables declared in a header section will be allocated memory space unless the declaration is preceded with `extern` .

4.23.1.2 Module Header

Every line between the `BeginHeader` and `EndHeader` comments belongs to the header of the module. When a library is linked to an application (i.e., the application has the statement `#use “library_name”`), Dynamic C precompiles every header in the library, and only the headers.

With proper function prototypes and variable declarations, a module header ensures proper type checking throughout the application program. Prototypes, variables, structures, typedefs and macros declared in a header section will always be parsed by the compiler if the library is #used, and everything will have global scope. It is even permissible to put function bodies in header sections, but it’s not recommended because the function will be compiled with any application that #uses the library. Since variables declared in a header section will be allocated memory space unless the declaration is preceded with `extern`, the variable declaration should be in the module body instead of the header to save data space.

The scope of anything inside the module header is global; this includes compiler directives. Since the headers are compiled before the module bodies, the last one of a given type of directive encountered will be in effect and any previous ones will be forgotten.

Using compiler directives like `#class` or `#memmap` inside module headers is inadvisable. If it is important to set, for example, “#class auto” for some library modules and “#class static” for others, the appropriate directives should be placed inside the module body, not in the module header. Furthermore, since there is no guaranteed compilation order and compiler directives have global scope, when you issue a compiler directive to change default behavior for a particular module, at the end of the module you should issue another compiler directive to change back to the default behavior. For example, if a module body needs to have its storage class as static, have a “#class static” directive at the beginning of the module body and “#class auto” at the end.

4.23.1.3 Module 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 or header are referenced anywhere in the application. So keep modules small, don't put all the functions in a library into one module. If you look at the Dynamic C libraries you'll notice that many modules consist of one function. This saves on code size, because only the functions that are called are actually compiled into the application.

To further minimize waste, define code and data only in the body of a module. It is recommended that a module header contain only prototypes and `extern` declarations because they do not generate any code by themselves. That way, the compiler will generate code or allocate data *only* if the module is used by the application program.

4.23.2 Module Sample Code

There are many examples of modules in the `Lib` directory of Dynamic C. The following code will illustrate proper module syntax and show the scope of directives, functions and variables.

```
/*** BeginHeader ticks*/
   extern unsigned long ticks;
/*** EndHeader */
   unsigned long ticks;

/*** BeginHeader Get_Ticks */
   unsigned long Get_Ticks();
/*** EndHeader */
unsigned long Get_Ticks(){
   ...
}

/*** BeginHeader Inc_Ticks */
   void Inc_Ticks( int i );
/*** EndHeader */
#asm
Inc_Ticks::
   or   a
   ipset 1
   ...
   ipres
   ret
#endasm
```

There are 3 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. Although `Inc_Ticks` is an assembly language routine, it has a function prototype in the module header, allowing the compiler to check calls to it.

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 triggers compilation of the module body corresponding to `ticks` because the functions use the variable `ticks`.

```
/*** BeginHeader func_a */
int func_a();
#ifdef SECONDHEADER
   #define XYZ
#endif
/*** EndHeader */
int func_a(){
#ifdef SECONDHEADER
   printf ("I am function A.\n");
#endif
}
/*** BeginHeader func_b */
   int func_b();
   #define SECONDHEADER
/*** EndHeader */
   #ifdef XYZ
      #define FUNCTION_B
   #endif
int func_b() {
   #ifdef FUNCTION_B
      printf ("I am function B.\n");
   #endif
}
```

Let’s say the above file is named `mylibrary.lib`. If an application has the statement `#use “mylibrary.lib”` and then calls `func_b()`, will the printf statement be reached? The answer is no. The order of compilation for module headers is sequential from the beginning of the file, therefore, the macro `SECONDHEADER` is undefined when the first module header is parsed.

If an application #uses this library and then makes a call to `func_a()`, will that function’s print statement be reached? The answer is yes. Since all the headers were compiled first, the macro `SECONDHEADER` is defined when the first module body is compiled.

4.23.3 Important Notes

Remember that in a Dynamic C application there is only one file that contains `main()`. All other source files used by the file that contains `main()` are regarded as library files. Each library must be included in `LIB.DIR` (or a user defined replacement for `LIB.DIR`). Although Dynamic C uses `.LIB` as the library extension, you may use anything you like as long as the complete path is entered in your `LIB.DIR` file.

There is no way to define file scope variables in Dynamic C libraries.

4.24 Function Description Headers

Each user-callable function in a Z-World library has a descriptive header preceding the function to describe the function. Function headers are extracted by Dynamic C to provide on-line help messages.

The header is a specially formatted comment, such as the following example.

```
/* START FUNCTION DESCRIPTION *********************
WrIOport                      <IO.LIB>
SYNTAX: void WrIOport(int portaddr, int value);
DESCRIPTION:
Writes data to the specified I/O port.
PARAMETER1:  portaddr - register address of the port.
PARAMETER2:  value - data to be written to the port.

RETURN VALUE:  None
KEY WORDS: parallel port

SEE ALSO:  RdIOport
END DESCRIPTION ***********************************/
```

If this format is followed, user-created library functions will show up in the “Function Lookup” facility. Note that these sections are scanned in only when Dynamic C starts.

4.25 Support Files

Dynamic C has several support files that are necessary in building an application. These files are listed below.

Table 4-5 Dynamic C Support Files

File Name	Purpose of File
DCW.CFG	Contains configuration data for the target controller.
DC.HH	Contains prototypes, basic type definitions, #define, and default modes for Dynamic C. This file can be modified by the programmer.
DEFAULT.H	Contains a set of #use directives for each control product that Z-World ships. This file can be modified.
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 the libraries on the Dynamic C distribution disk. Any library that is to be used in a Dynamic C program must be listed in the file LIB.DIR, or another *.DIR file specified by the user. (Starting with version Dynamic C 7.05, a different *.DIR file may be specified by the user in the Compiler Options dialog to facilitate working on multiple projects.)
PROJECT.DCP DEFAULT.DCP	These files hold the default compilation environment that is shipped from the factory. DEFAULT.DCP may be modified, but not PROJECT.DCP. See Chapter 16 for details on project files.

5. Multitasking with Dynamic C

A *task* is an ordered list of operations to perform. In a multitasking environment, more than one task (each representing a sequence of operations) can *appear* to execute in parallel. In reality, a single processor can only execute one instruction at a time. If an application has multiple tasks to perform, multitasking software can usually take advantage of natural delays in each task to increase the overall performance of the system. Each task can do some of its work while the other tasks are waiting for an event, or for something to do. In this way, the tasks execute *almost* in parallel.

There are two types of multitasking available for developing applications in Dynamic C: *preemptive* and *cooperative*. In a cooperative multitasking environment, each well-behaved task voluntarily gives up control when it is waiting, allowing other tasks to execute. Dynamic C has language extensions, *costatements* and *cofunctions*, to support cooperative multitasking. Preemptive multitasking is supported by the *slice* statement, which allows a computation to be divided into small slices of a few milliseconds each, and by the µC/OS-II real-time kernel.

5.1 Cooperative Multitasking

In the absence of a preemptive multitasking kernel or operating system, a programmer given a real-time programming problem that involves running separate tasks on different time scales will often come up with a solution that can be described as a *big loop* driving state machines.

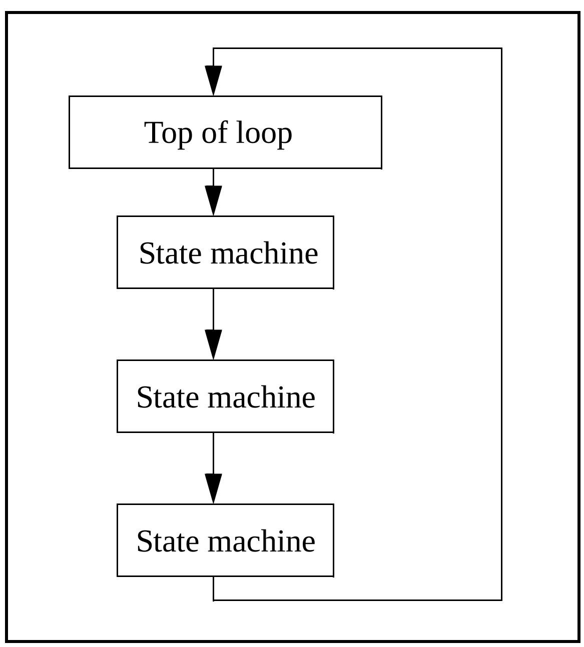


Figure 5-1. Big Loop

This means that the program consists of a large, endless loop—a big loop. Within the loop, tasks are accomplished by small fragments of a program that cycle through a series of states. The state is typically encoded as numerical values in C variables.

State machines can become quite complicated, involving a large number of state variables and a large number of states. The advantage of the state machine is that it avoids busy waiting, which is waiting in a loop until a condition is satisfied. In this way, one big loop can service a large number of state machines, each performing its own task, and no one is busy waiting.

The cooperative multitasking language extensions added to Dynamic C use the big loop and state machine concept, but C code is used to implement the state machine rather than C variables. The state of a task is remembered by a statement pointer that records the place where execution of the block of statements has been paused to wait for an event.

To multitask using Dynamic C language extensions, most application programs will have some flavor of this simple structure:

```
main() {
   int i;
   while(1) {          // endless loop for multitasking framework
      costate {        // task 1
         . . .         // body of costatement
      }
      costate {        // task 2
         ...           // body of costatement
      }
   }
}
```

5.2 A Real-Time Problem

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

Start:

1. Wait for a pushbutton to be pressed.
2. Turn on the first device.
3. Wait 60 seconds.
4. Turn on the second device.
5. Wait 60 seconds.
6. Turn off both devices.
7. Go back to the start.

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

5.2.1 Solving the Real-Time Problem with a State Machine

Here is what a state machine solution might look like.

```
task1state = 1;                    // initialization:
while(1){
   switch(task1state){
      case 1:
         if( buttonpushed() ){
            task1state=2;   turnondevice1();
            timer1 = time;          // time incremented every second
         }
         break;
      case 2:
         if( (time-timer1) >= 60L){
            task1state=3;   turnondevice2();
            timer2=time;
         }
         break;
      case 3:
         if( (time-timer2) >= 60L){
            task1state=1;   turnoffdevice1();
            turnoffdevice2();
         }
         break;
   }
   /* other tasks or state machines */
}
```

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

5.3 Costatements

Costatements are Dynamic C extensions to the C language which simplify implementation of state machines. Costatements are cooperative because their execution can be voluntarily suspended and later resumed. The body of a costatement is an ordered list of operations to perform -- a task. Each costatement has its own statement pointer to keep track of which item on the list will be performed when the costatement is given a chance to run. As part of the startup initialization, the pointer is set to point to the first statement of the costatement.

The statement pointer is effectively a state variable for the costatement or cofunction. It specifies the statement where execution is to begin when the program execution thread hits the start of the costatement.

All costatements in the program, except those that use pointers as their names, are initialized when the function chain `_GLOBAL_INIT` is called. `_GLOBAL_INIT` is called automatically by `premain` before `main` is called. Calling `_GLOBAL_INIT` from an application program will cause reinitialization of anything that was initialized in the call made by `premain`.

5.3.1 Solving the Real-Time Problem with Costatements

The Dynamic C costatement provides an easier way to control the tasks. It is relatively easy to add a task that checks for the use of an emergency stop button and then behaves accordingly.

```
while(1){
   costate{ ... }                        // task 1

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

The solution is elegant and simple. Note that the second costatement looks much like the original description of the problem. All the branching, nesting and variables within the task are hidden in the implementation of the costatement and its `waitfor` statements.

5.3.2 Costatement Syntax

costate [*name* [*state*]] { [*statement* | yield; | abort; | waitfor(*expression*);] . . .}

The keyword `costate` identifies the statements enclosed in the curly braces that follow as a costatement.

`name` 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`

Costatements can be named or unnamed. If `name` is absent the compiler creates an "unnamed" structure of type `CoData` for the costatement.

`state` can be one of the following:

- `always_on`

  The costatement is always active. This means the costatement will execute every time it is encountered in the execution thread, unless it is made inactive by `CoPause()`. It may be made active again by `CoResume()`.

- `init_on`

  The costatement is initially active and will automatically execute the first time it is encountered in the execution thread. The costatement becomes inactive after it completes (or aborts). The costatement can be made inactive by `CoPause()`.

If `state` is absent, a named costatement is initialized in a paused `init_on` condition. This means that the costatement will not execute until `CoBegin()` or `CoResume()` is executed. It will then execute once and become inactive again.

Unnamed costatements are `always_on`. You cannot specify `init_on` without specifying `name`.

5.3.3 Control Statements

`waitfor` **(** *expression* **);**

The keyword `waitfor` indicates a special `waitfor` statement and not a function call. The expression is computed each time `waitfor` is executed. If true (non-zero), execution proceeds to the next statement, otherwise a jump is made to the closing brace of the costatement or cofunction, with the statement pointer continuing to point to the `waitfor` statement. Any valid C function that returns a value can be used in a `waitfor` statement.

`yield`

The `yield` statement makes an unconditional exit from a costatement or a cofunction. Execution continues at the statement following `yield` the next time the costatement or cofunction is encountered.

`abort`

The `abort` statement causes the costatement or cofunction to terminate execution. If a costatement is `always_on`, the next time the program reaches it, it will restart from the top. If the costatement is not `always_on`, it becomes inactive and will not execute again until turned on by some other software.

A costatement can have as many C statements, including `abort`, `yield`, and `waitfor` statements, as needed. Costatements can be nested.

5.4 Advanced Costatement Topics

Each costatement has a structure of type `CoData`. This structure contains state and timing information. It also contains the address inside the costatement that will execute the next time the program thread reaches the costatement. A value of zero in the address location indicates the beginning of the costatement.

5.4.1 The CoData Structure

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

5.4.2 CoData Fields

CSState

The CSState field contains two flags, STOPPED and INIT. The possible flag values and their meaning are in the table below.

Table 5-1. Flags that specify the run status of a costatement

STOPPED	INIT	State of Costatement
yes	yes	Done, or has been initialized to run, but set to inactive. Set by CoReset().
yes	no	Paused, waiting to resume. Set by CoPause().
no	yes	Initialized to run. Set by CoBegin().
no	no	Running. 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 or cofunctions.

Last Location

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

These fields are zeroed whenever one of the following is true:

- the CoData structure is initialized by a call to _GLOBAL_INIT, CoBegin or CoReset
- the costatement is executed to completion
- the costatement is aborted.

Check Sum

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

First Time

The firsttime field is a flag that is used by a waitfor, or waitfordone statement. It is set to 1 before the statement is evaluated the first time. This aids in calculating elapsed time for the functions DelayMs, DelaySec, DelayTicks, IntervalTick, IntervalMs, and IntervalSec.

Content

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

Check Sum 2

The `ChkSum2` field is currently unused.

5.4.3 Pointer to CoData Structure

To obtain a pointer to a named costatement's CoData structure, do the following:

```
CoData    cost1;        // allocate memory for a CoData struct
CoData    *pcost1;

pcost1 = &cost1;        // get pointer to the CoData struct
...
CoBegin (pcost1);       // initialize CoData struct
costate pcost1 {        // pcost1 is the costatement name and also a
   ...                  //    pointer to its  CoData structure.
}
```

5.4.4 Functions for Use With Named Costatements

For detailed function descriptions, please see the *Dynamic C Function Reference Manual* or select Function Lookup/Insert from Dynamic C's Help menu (keyboard shortcut is <Ctrl-H>).

All of these functions are in `COSTATE.LIB`. Each one takes a pointer to a `CoData` struct as its only parameter.

isCoDone

`int isCoDone(CoData* p);`

This function returns true if the costatement pointed to by `p` has completed.

isCoRunning

`int isCoRunning(CoData* p);`

This function returns true if the costatement pointed to by `p` will run if given a continuation call.

CoBegin

`void CoBegin(CoData* p);`

This function initializes a costatement's `CoData` structure so that the costatement will be executed next time it is encountered.

CoPause

`void CoPause(CoData* p);`

This function will change `CoData` so that the associated costatement is paused. When a costatement is called in this state it does an implicit yield until it is released by a call from `CoResume` or `CoBegin`.

CoReset

`void CoReset(CoData* p);`

This function initializes a costatement's CoData structure so that the costatement will not be executed the next time it is encountered (unless the costatement is declared `always_on`.)

CoResume

`void CoResume(CoData* p);`

This function unpauses a paused costatement. The costatement will resume the next time it is called.

5.4.5 Firsttime Functions

In a function definition, the keyword `firsttime` causes the function to have an implicit first parameter: a pointer to the CoData structure of the costatement that calls it.

The following `firsttime` functions are defined in `COSTATE.LIB`. For more information see the *Dynamic C Function Reference Manual*. These functions should be called inside a `waitfor` statement because they do not yield while waiting for the desired time to elapse, but instead return 0 to indicate that the desired time has not yet elapsed.

```
DelayMs        IntervalMs
DelaySec       IntervalSec
DelayTicks     IntervalTick
```

User-defined `firsttime` functions are allowed.

5.4.6 Shared Global Variables

The variables `SEC_TIMER`, `MS_TIMER` and `TICK_TIMER` are shared, making them atomic when being updated. They are defined and initialized in `VDRIVER.LIB`. They are updated by the periodic interrupt and are used by `firsttime` functions. They should not be modified by an application program. Costatements and cofunctions depend on these timer variables being valid for use in `waitfor` statements that call functions that read them. E.g. the following statement will access `SEC_TIMER`.

```
waitfor(DelaySec(3));
```

5.5 Cofunctions

Cofunctions, like costatements, are used to implement cooperative multitasking. But, unlike costatements, they have a form similar to functions in that arguments can be passed to them and a value can be returned (but not a structure).

The default storage class for a cofunction's variables is `Instance`. An `instance` variable behaves like a `static` variable, i.e., its value persists between function calls. Each instance of an *Indexed Cofunction* has its own set of instance variables. The compiler directive `#class` does not change the default storage class for a cofunction's variables.

All cofunctions in the program are initialized when the function chain `_GLOBAL_INIT` is called. This call is made by `premain`.

5.5.1 Cofunction Syntax

A cofunction definition is similar to the definition of a C function.

```
cofunc|scofunc type [name][[dim]]([type arg1, ..., type argN])
   { [ statement | yield; | abort; | waitfor(expression);]... }
```

cofunc, scofunc

The keywords `cofunc` or `scofunc` (a single-user cofunction) identify the statements enclosed in curly braces that follow as a cofunction.

type

Whichever keyword (`cofunc` or `scofunc`) is used is followed by the data type returned (`void`, `int`, etc.).

name

A `name` can be any valid C name not previously used. This results in the creation of a structure of type `CoData` of the same name.

dim

The cofunction `name` may be followed by a dimension if an indexed cofunction is being defined.

cofunction arguments (arg1, . . ., argN)

As with other Dynamic C functions, cofunction arguments are passed by value.

cofunction body

A cofunction can have as many C statements, including `abort`, `yield`, `waitfor`, and `waitfordone` statements, as needed. Cofunctions can contain calls to other cofunctions.

5.5.2 Calling Restrictions

You cannot assign a cofunction to a function pointer then call it via the pointer.

Cofunctions are called using a `waitfordone` statement. Cofunctions and the `waitfordone` statement may return an argument value as in the following example.

```
int j,k,x,y,z;
j = waitfordone x = Cofunc1;
k = waitfordone{ y=Cofunc2(...); z=Cofunc3(...); }
```

The keyword `waitfordone` (can be abbreviated to the keyword `wfd`) must be inside a costatement or cofunction. Since a cofunction must be called from inside a `wfd` statement, ultimately a `wfd` statement must be inside a costatement.

If only one cofunction is being called by `wfd` the curly braces are not needed.

The `wfd` statement executes cofunctions and `firsttime` functions. When all the cofunctions and `firsttime` functions listed in the `wfd` statement are complete (or one of them aborts), execution proceeds to the statement following `wfd`. Otherwise a jump is made to the ending brace of the costatement or cofunction where the `wfd` statement appears and when the execution thread comes around again control is given back to `wfd`.

In the example above, `x`, `y` and `z` must be set by `return` statements inside the called cofunctions. Executing a return statement in a cofunction has the same effect as executing the end brace.

In the example above, the variable `k` is a status variable that is set according to the following scheme. If no abort has taken place in any cofunction, `k` is set to 1, 2, ..., n to indicate which cofunction inside the braces finished executing last. If an abort takes place, `k` is set to -1, -2, ..., -n to indicate which cofunction caused the abort.

5.5.2.1 Using the IX Register

Functions called from within a cofunction may use the IX register if they restore it before the cofunction is exited, which includes an exit via an incomplete `waitfordone` statement.

In the case of an application that uses the #useix directive, the IX register will be corrupted when any stack-variable using function is called from within a cofunction, or if a stack-variable using function contains a call to a cofunction.

5.5.3 CoData Structure

The CoData structure discussed in Section 5.4.1 applies to cofunctions; each cofunction has an associated CoData structure.

5.5.4 Firsttime Functions

The `firsttime` functions discussed in “Firsttime Functions” on page 53 can also be used inside cofunctions. They should be called inside a `waitfor` statement. If you call these functions from inside a `wfd` statement, no compiler error is generated, but, since these delay functions do not yield while waiting for the desired time to elapse, but instead return 0 to indicate that the desired time has not yet elapsed, the `wfd` statement will consider a return value to be completion of the `firsttime` function and control will pass to the statement following the `wfd`.

5.5.5 Types of Cofunctions

There are three types of cofunctions: simple, indexed and single-user. Which one to use depends on the problem that is being solved. A single-user, indexed cofunction is not valid.

5.5.5.1 Simple Cofunction

A simple cofunction has only one instance and is similar to a regular function with a costate taking up most of the function's body.

5.5.5.2 Indexed Cofunction

An indexed cofunction allows the body of a cofunction to be called more than once with different parameters and local variables. The parameters and the local variable that are not declared static have a special lifetime that begins at a first time call of a cofunction instance and ends when the last curly brace of the cofunction is reached or when an `abort` or `return` is encountered.

The indexed cofunction call is a cross between an array access and a normal function call, where the array access selects the specific instance to be run.

Typically this type of cofunction is used in a situation where N identical units need to be controlled by the same algorithm. For example, a program to control the door latches in a building could use indexed cofunctions. The same cofunction code would read the key pad at each door, compare the passcode to the approved list, and operate the door latch. If there are 25 doors in the building, then the indexed cofunction would use an index ranging from 0 to 24 to keep track of which door is currently being tested. An indexed cofunction has an index similar to an array index.

```
waitfordone{ ICofunc[n](...); ICofunc2[m](...); }
```

The value between the square brackets must be positive and less than the maximum number of instances for that cofunction. There is no runtime checking on the instance selected, so, like arrays, the programmer is responsible for keeping this value in the proper range.

5.5.5.2.1 Indexed Cofunction Restrictions

Costatements are not supported inside indexed cofunctions. Single user cofunctions can not be indexed.

5.5.5.3 Single User Cofunction

Since cofunctions are executing in parallel, the same cofunction normally cannot be called at the same time from two places in the same big loop. For example, the following statement containing two simple cofunctions will generally cause a fatal error.

```
waitfordone{ cofunc_nameA(); cofunc_nameA();}
```

This is because the same cofunction is being called from the second location after it has already started, but not completed, execution for the call from the first location. The cofunction is a state machine and it has an internal statement pointer that cannot point to two statements at the same time.

Single-user cofunctions can be used instead. They can be called simultaneously because the second and additional callers are made to wait until the first call completes. The following statement, which contains two single-user cofunctions, is okay.

```
waitfordone( scofunc_nameA(); scofunc_nameA();}
```

loopinit()

This function should be called in the beginning of a program that uses single-user cofunctions. It initializes internal data structures that are used by `loophead()`.

loophead()

This function should be called within the "big loop" in your program. It is necessary for proper single-user cofunction abandonment handling.

Example

```
// echoes characters
main() {
   int c;
   serXopen(19200);
   loopinit();
   while (1) {
      loophead();
      wfd c = cof_serAgetc();
      wfd cof_serAputc(c);
   }
   serAclose();
}
```

5.5.6 Types of Cofunction Calls

A `wfd` statement makes one of three types of calls to a cofunction.

5.5.6.1 First Time Call

A first time call happens when a `wfd` statement calls a cofunction for the first time in that statement. After the first time, only the original `wfd` statement can give this cofunction instance continuation calls until either the instance is complete or until the instance is given another first time call from a different statement.

5.5.6.2 Continuation Call

A continuation call is when a cofunction that has previously yielded is given another chance to run by the enclosing `wfd` statement. These statements can only call the cofunction if it was the last statement to give the cofunction a first time call or a continuation call.

5.5.6.3 Terminal Call

A terminal call ends with a cofunction returning to its `wfd` statement without yielding to another cofunction. This can happen when it reaches the end of the cofunction and does an implicit return, when the cofunction does an explicit return, or when the cofunction aborts.

5.5.6.4 Lifetime of a Cofunction Instance

This stretches from a first time call until its terminal call or until its next first time call.

5.5.7 Special Code Blocks

The following special code blocks can appear inside a cofunction.

everytime { *statements* }

This must be the first statement in the cofunction. It will be executed every time program execution passes to the cofunction no matter where the statement pointer is pointing. After the `everytime` statements are executed, control will pass to the statement pointed to by the cofunction's statement pointer.

abandon { *statements* **}**

This keyword applies to single-user cofunctions only and must be the first statement in the body of the cofunction. The statements inside the curly braces will be executed if the single-user cofunction is forcibly abandoned. A call to `loophead()` (defined in `COFUNC.LIB`) is necessary for abandon statements to execute.

Example

`Samples/COFUNC/ COFABAND.C` illustrates the use of `abandon`.

```
scofunc SCofTest(int i){
   abandon {
      printf("CofTest was abandoned\n");
   }
   while(i>0) {
      printf("CofTest(%d)\n",i);
      yield;
   }
}
main(){
   int x;
   for(x=0;x<=10;x++) {
      loophead();
      if(x<5) {
         costate {
            wfd SCofTest(1);        // first caller
         }
      }
      costate {
         wfd SCofTest(2);           // second caller
      }
   }
}
```

In this example two tasks in `main` are requesting access to `SCofTest`. The first request is honored and the second request is held. When `loophead` notices that the first caller is not being called each time around the loop, it cancels the request, calls the abandonment code and allows the second caller in.

5.5.8 Solving the Real-Time Problem with Cofunctions

```
for(;;){
   costate{                                  // task 1
      wfd emergencystop();
      for (i=0; i<MAX_DEVICES; i++)
         wfd turnoffdevice(i);
   }
   costate{                                  // task 2
      wfd x = buttonpushed();
      wfd turnondevice(x);
      waitfor( DelaySec(60L) );
      wfd turnoffdevice(x);
   }
   ...
   costate{ ... }                            // task n
}
```

Cofunctions, with their ability to receive arguments and return values, provide more flexibility and specificity than our previous solutions. Using cofunctions, new machines can be added with only trivial code changes. Making `buttonpushed()` a cofunction allows more specificity because the value returned can indicate a particular button in an array of buttons. Then that value can be passed as an argument to the cofunctions `turnondevice` and `turnoffdevice`.

5.6 Patterns of Cooperative Multitasking

Sometimes a task may be something that has a beginning and an end. For example, a cofunction to transmit a string of characters via the serial port begins when the cofunction is first called, and continues during successive calls as control cycles around the big loop. The end occurs after the last character has been sent and the `waitfordone` condition is satisified. This type of a call to a cofunctions might look like this:

```
waitfordone{ SendSerial("string of characters"); }
[ next statement ]
```

The next statement will execute after the last character is sent.

Some tasks may not have an end. They are endless loops. For example, a task to control a servo loop may run continuously to regulate the temperature in an oven. If there are a a number of tasks that need to run continuously, then they can be called using a single `waitfordone` statement as shown below.

```
costate {
   waitfordone { Task1(); Task2(); Task3(); Task4(); }
   [ to come here is an error ]
}
```

Each task will receive some execution time and, assuming none of the tasks is completed, they will continue to be called. If one of the cofunctions should abort, then the `waitfordone` statement will abort, and corrective action can be taken.

5.7 Timing Considerations

In most instances, costatements and cofunctions are grouped as periodically executed tasks. They can be part of a real-time task, which executes every *n* milliseconds as shown below using costatements.

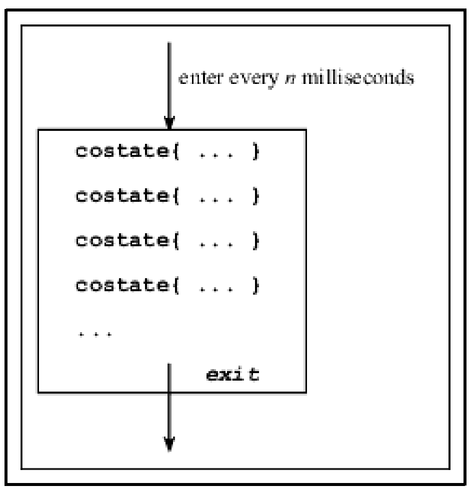


Figure 5-2. Costatement as Part of Real-Time Task

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

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

The consequences of such variations in the time between steps depends on the program's objective. Suppose that the typical delay between an event and the controller's response to the event is

25 ms, but under unusual circumstances the delay may reach 50 ms. An occasional slow response may have no consequences whatsoever. If a delay is added between the steps of a process where the time scale is measured in seconds, then the result may be a very slight reduction in throughput.

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

5.7.1 waitfor Accuracy Limits

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

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

5.8 Overview of Preemptive Multitasking

In a preemptive multitasking environment, tasks do not voluntarily relinquish control. Tasks are scheduled to run by priority level and/or by being given a certain amount of time.

There are two ways to accomplish preemptive multitasking using Dynamic C. The first way is µC/OS-II, a real-time, preemptive kernel that runs on the Rabbit microprocessor and is fully supported by Dynamic C. For more information see Chapter 18, “µC/OS-II.” The other way is to use `slice` statements.

5.9 Slice Statements

The `slice` statement, based on the costatement language construct, allows the programmer to run a block of code for a specific amount of time.

5.9.1 Slice Syntax

```
slice ([context_buffer,] context_buffer_size, time_slice)
   [name]{[statement|yield;|abort;|waitfor(expression);]}
```

`context_buffer_size`

This value must evaluate to a constant integer. The value specifies the number of bytes for the buffer `context_buffer`. It needs to be large enough for worst-case stack usage by the user program and interrupt routines.

`time_slice`

The amount of time in ticks for the slice to run. One tick = 1/1024 second.

name

When defining a named `slice` statement, you supply a context buffer as the first argument. When you define an unnamed `slice` statement, this structure is allocated by the compiler.

[statement | yield; | abort; | waitfor(expression);]

The body of a `slice` statement may contain:

- Regular C statements
- `yield` statements to make an unconditional exit.
- `abort` statements to make an execution jump to the very end of the statement.
- `waitfor` statements to suspend progress of the slice statement pending some condition indicated by the expression.

5.9.2 Usage

The `slice` statement can run both cooperatively and preemptively all in the same framework. A slice statement, like costatements and cofunctions, can suspend its execution with an `abort`, `yield`, or `waitfor`. It can also suspend execution with an implicit `yield` determined by the `time_slice` parameter that was passed to it.

A routine called from the periodic interrupt forms the basis for scheduling slice statements. It counts down the ticks and changes the `slice` statement's context.

5.9.3 Restrictions

Since a `slice` statement has its own stack, local auto variables and parameters cannot be accessed while in the context of a `slice` statement. Any functions called from the slice statement function normally.

Only one `slice` statement can be active at any time, which eliminates the possibility of nesting `slice` statements or using a `slice` statement inside a function that is either directly or indirectly called from a `slice` statement. The only methods supported for leaving a `slice` statement are completely executing the last statement in the `slice`, or executing an `abort`, `yield` or `waitfor` statement.

The `return`, `continue`, `break`, and `goto` statements are not supported.

Slice statements cannot be used with µC/OS-II or TCP/IP.

5.9.4 Slice Data Structure

Internally, the slice statement uses two structures to operate. When defining a named slice statement, you supply a context buffer as the first argument. When you define an unnamed slice statement, this structure is allocated by the compiler. Internally, the context buffer is represented by the SliceBuffer structure below.

```
struct SliceData {
   int time_out;
   void* my_sp;
   void* caller_sp;
   CoData codata;
}
struct SliceBuffer {
   SliceData slice_data;
   char stack[];              //  fills rest of the slice buffer
};
```

5.9.5 Slice Internals

When a slice statement is given control, it saves the current context and switches to a context associated with the slice statement. After that, the driving force behind the slice statement is the timer interrupt. Each time the timer interrupt is called, it checks to see if a slice statement is active. If a slice statement is active, the timer interrupt decrements the time_out field in the slice's SliceData. When the field is decremented to zero, the timer interrupt saves the slice statement's context into the SliceBuffer and restores the previous context. Once the timer interrupt completes, the flow of control is passed to the statement directly following the slice statement. A similar set of events takes place when the slice statement does an explicit yield/abort/waitfor.

5.9.5.1 Example 1

Two `slice` statements and a costatement will appear to run in parallel. Each block will run independently, but the `slice` statement blocks will suspend their operation after 20 ticks for `slice_a` and 40 ticks for `slice_b`. Costate a will not release control until it either explicitly yields, aborts, or completes. In contrast, `slice_a` will run for at most 20 ticks, then `slice_b` will begin running. Costate a will get its next opportunity to run about 60 ticks after it relinquishes control.

```
main () {
   int x, y, z;
   ...
   for (;;) {
      costate a {
         ...
      }
      slice(500, 20) {     // slice_a
         ...
      }
      slice(500, 40) {     // slice_b
         ...
      }
   }
}
```

5.9.5.2 Example 2

This code guarantees that the first slice starts on `TICK_TIMER` evenly divisible by 80 and the second starts on `TICK_TIMER` evenly divisible by 105.

```
main() {
   for(;;) {
      costate {
         slice(500,20) {                // slice_a
            waitfor(IntervalTick(80));
            ...
         }
         slice(500,50) {                // slice_b
            waitfor(IntervalTick(105);
            ...
         }
      }
   }
}
```

5.9.5.3 Example 3

This approach is more complicated, but will allow you to spend the idle time doing a low-priority background task.

```
main() {
   int time_left;
   long start_time;
   for(;;) {
      start_time = TICK_TIMER;
      slice(500,20) {                              // slice_a
         waitfor(IntervalTick(80));
            ...
      }
      slice(500,50) {                              // slice_b
         waitfor(IntervalTick(105));
            ...
      }
      time_left = 75-(TICK_TIMER-start_time);
      if(time_left>0) {
         slice(500,75-(TICK_TIMER-start_time)) {  // slice_c
            ...
         }
      }
   }
}
```

5.10 Summary

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

6. The Virtual Driver

Virtual Driver is the name given to some initialization services and a group of services performed by a periodic interrupt. These services are:

Initialization Services

- Call `_GLOBAL_INIT()`
- Initialize the global timer variables
- Start the Virtual Driver periodic interrupt

Periodic Interrupt Services

- Decrement software (virtual) watchdog timers
- Hitting the hardware watchdog timer
- Increment the global timer variables
- Drive uC/OS-II preemptive multitasking
- Drive slice statement preemptive multitasking

6.1 Default Operation

The user should be aware that by default the Virtual Driver starts and runs in a Dynamic C program without the user doing anything. This happens because before `main()` is called, a function called `premain()` is called by the Rabbit kernel (BIOS) that actually calls `main()`. Before `premain()` calls `main()`, it calls a function named `VdInit()` that performs the initialization services, including starting the periodic interrupt. If the user were to disable the Virtual Driver by commenting out the call to `VdInit()` in `premain()`, then none of the services performed by the periodic interrupt would be available. Unless the Virtual Driver is incompatible with some very tight timing requirements of a program and none of the services performed by the Virtual Driver are needed, it is recommended that the user not disable it.

6.2 Calling _GLOBAL_INIT()

`VdInit()` calls the function chanin `_GLOBAL_INIT()` which runs all `#GLOBAL_INIT` sections in a program. `_GLOBAL_INIT()` also initializes all of the CoData structures needed by costatements and cofunctions. If `VdInit()` is not called, users could still use costatements and cofunctions if the call to `VdInit()` was replaced by a call to `_GLOBAL_INIT()`, but the `DelaySec()` and `DelayMs()` functions often used with costatements and cofunctions in `waitfor` statements would not work because those functions depend on timer variables which are maintained by the periodic interrupt.

6.3 Global Timer Variables

SEC_TIMER, MS_TIMER and TICK_TIMER are global variables defined as shared unsigned long. These variables should never be changed by an application program. Among other things, the TCP/IP stack depends on the validity of the timer variables.

On initialization, SEC_TIMER is synchronized with the real-time clock. The date and time can be accessed more quickly by reading SEC_TIMER than by reading the real-time clock.

The periodic interrupt updates SEC_TIMER every second, MS_TIMER every millisecond, and TICK_TIMER 1024 times per second (the frequency of the periodic interrupt). These variables are used by the DelaySec, DelayMS and DelayTicks functions, but are also convenient for application programs to use for timing purposes. The following sample shows the use of MS_TIMER to measure the execution time in microseconds of a Dynamic C integer add. The work is done in a nodebug function so that debugging does not affect timing. For more information on the nodebug keyword, please see “nodebug” on page 157.

```
#define N 10000
main(){ timeit(); }
nodebug timeit(){
   unsigned long int T0;
   float T2,T1;
   int x,y;
   int i;

   T0 = MS_TIMER;
   for(i=0;i<N;i++) { }
   // T1 gives empty loop time
   T1=(MS_TIMER-T0);
   T0 = MS_TIMER;
   for(i=0;i<N;i++){ x+y;}
   // T2 gives test code execution time
   T2=(MS_TIMER-T0);
   // subtract empty loop time and convert to time for single pass
   T2=(T2-T1)/(float)N;
   // multiply by 1000 to convert ms. to us.
   printf("time to execute test code = %f us\n",T2*1000.0);
}
```

6.4 Watchdog Timers

Watchdog timers limit the amount of time your system will be in an unknown state.

6.4.1 Hardware Watchdog

The Rabbit CPU has one built-in hardware watchdog timer (WDT). The Virtual Driver hits this watchdog periodically. The following code fragment could be used to disable this WDT:

```
#asm
    ld a,0x51
ioi ld (WDTTR),a
    ld a,0x54
ioi ld (WDTTR),a
#endasm
```

However, it is recommended that the watchdog not be disabled. This prevents the target from entering an endless loop in software due to coding errors or hardware problems. If the Virtual Driver is not used, the user code should periodically call `hitwd()`.

When debugging a program, if the program is stopped at a breakpoint because the breakpoint was explicitly set, or because the user is single stepping, then the debug kernel hits the hardware watchdog periodically.

6.4.2 Virtual Watchdogs

There are 10 virtual WDTs available; they are maintained by the Virtual Driver. Virtual watchdogs, like the hardware watchdog, limit the amount of time a system is in an unknown state. They also narrow down the problem area to assist in debugging.

The function `VdGetFreeWd(count)` allocates and initializes a virtual watchdog. The return value of this function is the ID of the virtual watchdog. If an attempt is made to allocate more than 10 virtual WDTs, a fatal error occurs. In debug mode, this fatal error will cause the program to return with error code 250. The default run-time error behavior is to reset the board.

The ID returned by `VdGetFreeWd()` is used as the argument when calling `VdHitWd(ID)` or `VdReleaseWd(ID)` to hit or deallocate a virtual watchdog

The Virtual Driver counts down watchdogs every 62.5 ms. If a virtual watchdog reaches 0, this is fatal error code 247. Once a virtual watchdog is active, it should be reset periodically with a call to `VdHitWd(ID)` to prevent this. If count = 2 for a particular WDT, then `VdHitWd(ID)` will need to be called within 62.5 ms for that WDT. If count = 255, `VdHitWd(ID)` will need to be called within 15.94 seconds.

The Virtual Driver does not count down any virtual WDTs if the user is debugging with Dynamic C and stopped at a breakpoint.

6.5 Preemptive Multitasking Drivers

A simple scheduler for Dynamic C's preemptive slice statement is serviced by the Virtual Driver. The scheduling for µC/OS-II a more traditional full-featured real-time kernel, is also done by the Virtual Driver.

These two scheduling methods are mutually exclusive—slicing and µC/OS-II must not be used in the same program.

7. The Slave Port Driver

The Rabbit 2000 and the Rabbit 3000 have hardware for a slave port, allowing a master controller to read and write certain internal registers on the Rabbit. The library, `Slaveport.lib`, implements a complete master/slave protocol for the Rabbit slave port. Sample libraries, `Master_serial.lib` and `Sp_stream.lib` provide serial port and stream-based communication handlers using the slave port protocol.

7.1 Slave Port Driver Protocol

Given the variety of embedded system implementations, the protocol for the slave port driver was designed to make the software for the master controller as simple as possible. Each interaction between the master and the slave is initiated by the master. The master has complete control over when data transfers occur and can expect single, immediate responses from the slave.

7.1.1 Overview

1. Master writes to the command register after setting the address register and, optionally, the data register. These registers are internal to the slave.
2. Slave reads the registers that were written by the master.
3. Slave writes to command response register after optionally setting the data register. This also causes the SLAVEATTN line on the Rabbit slave to be pulled low.
4. Master reads response and data registers.
5. Master writes to the slave port status register to clear interrupt line from the slave.

7.1.2 Registers on the Slave

From the point of view of the master, the slave is an I/O device with four register addresses.

Table 7-1. The slave registers that are accessible by the master

Register Name	Internal Address of Register	Address of Register From Master's Perspective	Register Use
`SPD0R`	**0x20**	**0**	Command and response register
`SPD1R`	**0x21**	**1**	Address register
`SPD2R`	**0x22**	**2**	Optional data register
`SPSR`	**0x23**	**3**	Slave port status register. In this protocol the only bit used is for checking the command response register. Bit 3 is set if the slave has written to SPD0R. It is cleared when the master writes to SPSR, which also deasserts the SLAVEATTN line.

Accessing the same address (0, 1 or 2) uses two different registers, depending on whether the access was a read or a write. In other words, when writing to address 0, the master accesses a different location than when the it reads address 0.

Table 7-2. What happens when the master accesses a slave register

Register Address	Read	Write
0	Gets command response from slave	Sends command to slave, triggers slave response
1	Not used	Sets channel address to send command to
2	Gets returned data from slave	Sets data byte to send to slave
3	Gets slave port status (see below)	Clears slave response bit (see below)

The status port is a bit field showing which slave port registers have been updated. For the purposes of this protocol. Only bit 3 needs to be examined. After sending a command, the master can check bit 3, which is set when the slave writes to the response register. At this point the response and returned data are valid and should be read before sending a new command. Performing a dummy write to the status register will clear this bit, so that it can be set by the next response.

Pin assignments for both the Rabbit 2000 and the Rabbit 3000 acting as a slave are as follows:

Table 7-3. Pin assignments for the Rabbit acting as a slave

Pin	Function
PE7	/SCS chip select (active low to read/write slave port)
PB2	/SWR slave write (assert for write cycle)
PB3	/SRD slave read (assert for read cycle)
PB4	SA0 low address bit for slave port registers
PB5	SA1 high address bit for slave registers
PB7	/SLVATTN asserted by slave when it responds to a command. cleared by master write to status register
PA0-PA7	slave port data bus

For more details and read/write signal timing see the *Rabbit 2000 Microprocessor User's Manual* or the *Rabbit 3000 Microprocessor User's Manual*.

7.1.3 Polling and Interrupts

Both the slave and the master can use interrupt or polling for the slave. The parameter passed to `SPinit()` determines which one is used. In interrupt mode, the developer can indicate whether the handler functions for the channels are interruptible or non-interruptible.

7.1.4 Communication Channels

The Rabbit slave has 256 configurable channels available for communication. The developer must provide a handler function for each channel that is used. Some basic handlers are available in the library `Slave_Port.lib`. These handlers will be discussed later in this chapter.

When the slave port driver is initialized, a callback table of handler functions is set up. Handler functions are added to the callback table by `SPsetHandler()`.

7.2 Functions

`Slave_port.lib` provides the following functions:

SPinit

```
int SPinit ( int mode );
```

DESCRIPTION

This function initializes the slave port driver. It sets up the callback tables for the different channels. The slave port driver can be run in either polling mode where `SPtick()` must be called periodically, or in interrupt mode where an ISR is triggered every time the master sends a command. There are two version of interrupt mode. In the first, interrupts are reenabled while the handler function is executing. In the other, the handler function will execute at the same interrupt priority as the driver ISR.

PARAMETERS

mode
`0`: For polling
`1`: For interrupt driven (interruptible handler functions)
`2`: For interrupt driven (non-interruptible handler functions)

RETURN VALUE

`1`: Success
`0`: Failure

LIBRARY

SLAVE_PORT.LIB

SPsetHandler

```
int SPsetHandler ( char address, int (*handler)(), void
   *handler_params);
```

DESCRIPTION

This function sets up a handler function to process incoming commands from the master for a particular slave port address.

PARAMETERS

address	The 8-bit slave port address of the channel that corresponds to the handler function.
handler	Pointer to the handler function. This function must have a particular form, which is described by the function description for `MyHandler()` shown below. Setting this parameter to `NULL` unloads the current handler.
handler_params	Pointer that will be saved and passed to the handler function each time it is called. This allows the handler function to be parameterized for multiple cases.

RETURN VALUE

`1`: Success, the handler was set.
`0`: Failure.

LIBRARY

`SLAVE_PORT.LIB`

MyHandler

```
int MyHandler ( char command, char data_in, void *params );
```

DESCRIPTION

This function is a developer-supplied function and can have any valid Dynamic C name. Its purpose is to handle incoming commands from a master to one of the 256 channels on the slave port. A handler function must be supplied for every channel that is being used on the slave port.

PARAMETERS

command	This is the received command byte.
data_in	The optional data byte
params	The optional parameters pointer.

RETURN VALUE

This function must return an integer. The low byte must contains the response code and the high byte contains the returned data, if there is any.

LIBRARY

This is a developer-supplied function.

SPtick

```
void SPtick ( void );
```

DESCRIPTION

This function must be called periodically when the slave port is used in polling mode.

LIBRARY

SLAVE_PORT.LIB

SPclose

```
void SPclose( void );
```

DESCRIPTION

This function disables the slave port driver and unloads the ISR if one was used.

LIBRARY

SLAVE_PORT.LIB

7.3 Examples

The rest of the chapter describes some useful handlers.

7.3.1 Status Handler

`SPstatusHandler()`, available in `Slave_port.lib`, is an example of a simple handler to report the status of the slave. To set up the function as a handler on slave port address 12, do the following:

```
SPsetHandler (12, SPstatusHandler, &status_char);
```

Sending any command to this handler will cause it to respond with a `1` in the response register and the current value of `status_char` in the data return register.

7.3.2 Serial Port Handler

`Slave_port.lib` contains handlers for all four serial ports on the slave.
`Master_serial.lib` contains code for a master using the slave's serial port handler. This library illustrates the general case of implementing the master side of the master/slave protocol.

7.3.2.1 Commands to the Slave

Table 7-4. Commands that the master can send to the slave

Command	Command Description
1	Transmit byte. Byte value is in data register. Slave responds with 1 if the byte was processed or 0 if it was not.
2	Receive byte. Slave responds with 2 if has put a new received byte into the data return register or 0 if there were no bytes to receive.
3	Combined transmit/receive—a combination of the transmit and receive commands. The response will also be a logical OR of the two command responses.
4	Set baud factor, byte 1 (LSB). The actual baud rate is the baud factor multiplied by 300.
5	Set baud factor, byte 2 (MSB). The actual baud rate is the baud factor multiplied by 300.
6	Set port configuration bits
7	Open port
8	Close port
9	Get errors. Slave responds with 1 if the port is open and can return an error bitfield. The error bits are the same as for the function serAgetErrors() and are put in the data return register by the slave.
10, 11	Returns count of free bytes in the serial port write buffer. The two commands return the LSB and the MSB of the count respectively. The LSB(10) should be read first to latch the count.
12, 13	Returns count of free bytes in the serial port read buffer. The two commands return the LSB and the MSB of the count respectively. The LSB(12) should be read first to latch the count.
14, 15	Returns count of bytes currently in the serial port write buffer. The two commands return the LSB and the MSB of the count respectively. The LSB(14) should be read first to latch the count.
16, 17	Returns count of bytes currently in the serial port write buffer. The two commands return the LSB and the MSB of the count respectively. The LSB(16) should be read first to latch the count.

7.3.2.2 Slave Side of Protocol

To set up the serial port handler to connect serial port A to channel 5 , do the following:

```
SPsetHandler (5, SPserAhandler, NULL);
```

7.3.2.3 Master Side of Protocol

The following functions are in `Master_serial.lib`. They are for a master using a serial port handler on a slave.

cof_MSgetc

```
int cof_MSgetc(char address);
```

DESCRIPTION

Yields to other tasks until a byte is received from the serial port on the slave.

PARAMETERS

address	Slave channel address of the serial handler.

RETURN VALUE

Value of the received character on success.
`-1`: Failure.

LIBRARY

MASTER_SERIAL.LIB

cof_MSputc

```
void cof_MSputc(char address, char ch);
```

DESCRIPTION

Sends a character to the serial port. Yields until character is sent.

PARAMETERS

address	Slave channel address of serial handler.
ch	Character to send.

RETURN VALUE

`0`: Success, character was sent.
`-1`: Failure, character was not sent.

LIBRARY

MASTER_SERIAL.LIB

cof_MSread

```
int cof_MSread(char address, char *buffer, int length, unsigned
    long timeout);
```

DESCRIPTION

Reads bytes from the serial port on the slave into the provided buffer. Waits until at least one character has been read. Returns after buffer is full, or `timeout` has expired between reading bytes. Yields to other tasks while waiting for data.

PARAMETERS

address	Slave channel address of serial handler.
buffer	Buffer to store received bytes.
length	Size of buffer.
timeout	Time to wait between bytes before giving up on receiving anymore.

RETURN VALUE

`>0`: Bytes read.
`-1`: Failure.

LIBRARY

`MASTER_SERIAL.LIB`

cof_MSwrite

```
int cof_MSwrite(char address, char *data, int length);
```

DESCRIPTION

Transmits an array of bytes from the serial port on the slave. Yields to other tasks while waiting for write buffer to clear.

PARAMETERS

address	Slave channel address of serial handler.
data	Array to be transmitted.
length	Size of array.

RETURN VALUE

Number of bytes actually written or `-1` if error.

LIBRARY

`MASTER_SERIAL.LIB`

MSclose

```
int MSclose(char address);
```

DESCRIPTION

Closes a serial port on the slave.

PARAMETERS

address	Slave channel address of serial handler.

RETURN VALUE

`0`: Success.
`-1`: Failure.

LIBRARY

`MASTER_SERIAL.LIB`

MSgetc

`int MSgetc(char address);`

DESCRIPTION

Receives a character from the serial port.

PARAMETERS

`address` Slave channel address of serial handler.

RETURN VALUE

Value of received character.
`-1`: No character available.

LIBRARY

`MASTER_SERIAL.LIB`

MSgetError

`int MSgetError(char address);`

DESCRIPTION

Gets bitfield with any current error from the specified serial port on the slave. Error codes are:

```
SER_PARITY_ERROR
SER_OVERRUN_ERROR
```

PARAMETERS

`address` Slave channel address of serial handler.

RETURN VALUE

Number of bytes free: Success.
`-1`: Failure.

LIBRARY

`MASTER_SERIAL.LIB`

MSinit

`int MSinit(int io_bank);`

DESCRIPTION

Sets up the connection to the slave.

PARAMETERS

`io_bank`	The IO bank and chip select pin number for the slave device. This is a number from 0 to 7 inclusive.

RETURN VALUE

`1`: Success.

LIBRARY

`MASTER_SERIAL.LIB`

MSopen

`int MSopen(char address, unsigned long baud);`

DESCRIPTION

Opens a serial port on the slave, given that there is a serial handler at the specified address on the slave.

PARAMETERS

`address`	Slave channel address of serial handler.
`baud`	Baud rate for the serial port on the slave.

RETURN VALUE

`1`: Baud rate used matches the argument.
`0`: Different baud rate is being used.
`-1`: Slave port comm error occurred.

LIBRARY

`MASTER_SERIAL.LIB`

MSputc

`int MSputc(char address, char ch);`

DESCRIPTION

Transmits a single character through the serial port.

PARAMETERS

`address`	Slave channel address of serial handler.
`ch`	Character to send.

RETURN VALUE

`1`: Character sent.
`0`: Transmit buffer is full or locked.

LIBRARY

`MASTER_SERIAL.LIB`

MSrdFree

`int MSrdFree(char address);`

DESCRIPTION

Gets the number of bytes available in the specified serial port read buffer on the slave.

PARAMETERS

`address`	Slave channel address of serial handler.

RETURN VALUE

Number of bytes free: Success.
`-1`: Failure.

LIBRARY

`MASTER_SERIAL.LIB`

MSsendCommand

```
int MSsendCommand(char address, char command, char data,
   char *data_returned, unsigned long timeout);
```

DESCRIPTION

Sends a single command to the slave and gets a response. This function also serves as a general example of how to implement the master side of the slave protocol.

PARAMETERS

address	Slave channel address to send command to.
command	Command to be sent to the slave (see Section 7.3.2.1).
data	Data byte to be sent to the slave.
data_returned	Address of variable to place data returned by the slave.
timeout	Time to wait before giving up on slave response.

RETURN VALUE

≥0: Response code.
-1: Timeout occured before response.
-2: Nothing at that address (response = 0xff).

LIBRARY

MASTER_SERIAL.LIB

MSread

```
int MSread(char address, char *buffer, int size, unsigned long
   timeout);
```

DESCRIPTION

Receives bytes from the serial port on the slave.

PARAMETERS

address	Slave channel address of serial handler.
buffer	Array to put received data into.
size	Size of array (max bytes to be read).
timeout	Time to wait between characters before giving up on receiving any more.

RETURN VALUE

The number of bytes read into the buffer (behaves like `serXread()`).

LIBRARY

`MASTER_SERIAL.LIB`

MSwrFree

```
int MSwrFree(char address)
```

DESCRIPTION

Gets the number of bytes available in the specified serial port write buffer on the slave.

PARAMETERS

address	Slave channel address of serial handler.

RETURN VALUE

Number of bytes free: Success.
`-1`: Failure.

LIBRARY

`MASTER_SERIAL.LIB`

MSwrite

```
int MSwrite(char address, char *data, int length);
```

DESCRIPTION

Sends an array of bytes out the serial port on the slave (behaves like `serXwrite()`).

PARAMETERS

address	Slave channel address of serial handler.
data	Array of bytes to send.
length	Size of array.

RETURN VALUE

Number of bytes actually sent.

LIBRARY

MASTER_SERIAL.LIB

7.3.2.4 Sample Program for Master

This sample program, `/Samples/SlavePort/master_demo.c`, treats the slave like a serial port.

```
#use "master_serial.lib"
#define SP_CHANNEL 0x42

char* const test_str = "Hello There";

main(){
  char buffer[100];
  int read_length;
  MSinit(0);
  // comment this line out if talking to a stream handler
  printf("open returned:0x%x\n", MSopen(SP_CHANNEL, 9600));
  while(1)
  {
    costate
    {
      wfd{cof_MSwrite(SP_CHANNEL, test_str, strlen(test_str));}
      wfd{cof_MSwrite(SP_CHANNEL, test_str, strlen(test_str));}
    }
    costate
    {
      wfd{ read_length = cof_MSread(SP_CHANNEL, buffer, 99, 10); }
      if(read_length > 0)
      {
        buffer[read_length] = 0; //null terminator
        printf("Read:%s\n", buffer);
      }
      else if(read_length < 0)
      {
        printf("Got read error: %d\n", read_length);
      }
      printf("wrfree = %d\n", MSwrFree(SP_CHANNEL));
    }
  }
}
```

7.3.3 Byte Stream Handler

The library, `SP_STREAM.LIB`, implements a byte stream over the slave port. If the master is a Rabbit, the functions in `MASTER_SERIAL.LIB` can be used to access the stream as though it came from a serial port on the slave.

7.3.3.1 Slave Side of Stream Channel

To set up the function `SPShandler()` as the byte stream handler, do the following:

```
SPsetHandler (10, SPShandler, stream_ptr);
```

This sets up the stream to use channel 10 on the slave.

A sample program in Section 7.3.3.2 shows how to set up and initialize the circular buffers. An internal data structure, `SPStream`, keeps track of the buffers and a pointer to it is passed to `SPsetHandler()` and some of the auxiliary functions that supports the byte stream handler. This is also shown in the sample program.

7.3.3.1.1 Functions

These are the auxiliary functions that support the stream handler function, `SPShandler()`.

cbuf_init

```
void cbuf_init(char *circularBuffer, int dataSize);
```

DESCRIPTION

This function initializes a circular buffer.

PARAMETERS

circularBuffer	The circular buffer to initialize.
dataSize	Size available to data. The size must be 9 bytes more than the number of bytes needed for data. This is for internal bookkeeping.

LIBRARY

`RS232.LIB`

cof_SPSread

```
int cof_SPSread(SPStream *stream, void *data, int length,
   unsigned long tmout);
```

DESCRIPTION

Reads `length` bytes from the slave port input buffer or until `tmout` milliseconds transpires between bytes after the first byte is read. It will yield to other tasks while waiting for data. This function is non-reentrant.

PARAMETERS

stream	Pointer to the stream state structure.
data	Structure to read from slave port buffer.
length	Number of bytes to read.
tmout	Maximum wait in milliseconds for any byte from previous one.

RETURN VALUE

The number of bytes read from the buffer.

LIBRARY

SP_STREAM.LIB

cof_SPSwrite

```
int cof_SPSwrite(SPStream *stream, void *data, int length);
```

DESCRIPTION

Transmits `length` bytes to slave port output buffer.This function is non-reentrant.

PARAMETERS

stream	Pointer to the stream state structure.
data	Structure to write to slave port buffer.
length	Number of bytes to write.

RETURN VALUE

The number of bytes successfully written to slave port.

LIBRARY

SP_STREAM.LIB

SPSinit

```
void SPSinit( void );
```

DESCRIPTION

Initializes the circular buffers used by the stream handler.

LIBRARY

SP_STREAM.LIB

SPSread

int SPSread(SPStream *stream, void *data, int length, unsigned long tmout);

DESCRIPTION

Reads `length` bytes from the slave port input buffer or until `tmout` milliseconds transpires between bytes. If no data is available when this function is called, it will return immediately. This function will call `SPtick()` if the slave port is in polling mode.

This function is non-reentrant.

PARAMETERS

stream	Pointer to the stream state structure.
data	Buffer to read received data into.
length	Maximum number of bytes to read.
tmout	Time to wait between received bytes before returning.

RETURN VALUE

Number of bytes read into the data buffer

LIBRARY

SP_STREAM.LIB

SPSwrite

`int SPSwrite(SPSream *stream, void *data, int length)`

DESCRIPTION

This function transmits length bytes to slave port output buffer. If the slave port is in polling mode, this function will call `SPtick()` while waiting for the output buffer to empty. This function is non-reentrant.

PARAMETERS

`stream`	Pointer to the stream state structure.
`data`	Bytes to write to stream.
`length`	Size of write buffer.

RETURN VALUE

Number of bytes written into the data buffer.

LIBRARY

`SP_STREAM.LIB`

SPSwrFree

```
int SPSwrFree();
```

DESCRIPTION

Returns number of free bytes in the stream write buffer.

RETURN VALUE

Space available in the stream write buffer.

LIBRARY

SP_STREAM.LIB

SPSrdFree

```
int SPSrdFree();
```

DESCRIPTION

Returns the number of free bytes in the stream read buffer.

RETURN VALUE

Space available in the stream read buffer.

LIBRARY

SP_STREAM.LIB

SPSwrUsed

```
int SPSwrUsed();
```

DESCRIPTION

Returns the number of bytes currently in the stream write buffer.

RETURN VALUE

Number of bytes currently in the stream write buffer.

LIBRARY

`SP_STREAM.LIB`

SPSrdUsed

```
int SPSrdUsed();
```

DESCRIPTION

Returns the number of bytes currently in the stream read buffer.

RETURN VALUE

Number of bytes currently in the stream read buffer.

LIBRARY

`SP_STREAM.LIB`

7.3.3.2 Byte Stream Sample Program

This program, `/Samples/SlavePort/Slave_Demo.c`, runs on a slave and implements a byte stream over the slave port.

```
#class auto
#use "slave_port.lib"
#use "sp_stream.lib"
#define STREAM_BUFFER_SIZE 31
main()
{
   char buffer[10];
   int bytes_read;
   SPStream stream;
   // Circular buffers need 9 bytes for bookkeeping.
   char stream_inbuf[STREAM_BUFFER_SIZE + 9];
   char stream_outbuf[STREAM_BUFFER_SIZE + 9];
   SPStream *stream_ptr;
   // setup buffers
   cbuf_init(stream_inbuf, STREAM_BUFFER_SIZE);
   stream.inbuf = stream_inbuf;
   cbuf_init(stream_outbuf, STREAM_BUFFER_SIZE);
   stream.outbuf = stream_outbuf;
   stream_ptr = &stream;
   SPinit(1);
   SPsetHandler(0x42, SPShandler, stream_ptr);
   while(1)
   {
      bytes_read = SPSread(stream_ptr, buffer, 10, 10);
      if(bytes_read)
      {
         SPSwrite(stream_ptr, buffer, bytes_read);
      }
   }
}
```

8. Run-Time Errors

Compiled code generated by Dynamic C calls an exception handling routine for run-time errors. The exception handler supplied with Dynamic C prints internally defined error messages to a Windows message box when run-time errors are detected during a debugging session. When software runs stand-alone (disconnected from Dynamic C), such a run-time error will cause a watchdog timeout and reset. Run-time error logging is available for Rabbit-based target systems with battery-backed RAM.

8.1 Run-Time Error Handling

When a run-time error occurs, a call is made to `exception()`. The run-time error type is passed to `exception()`, which then pushes various parameters on the stack, and calls the installed error handler. The default error handler places information on the stack, disables interrupts, and enters an endless loop by calling the `_xexit` function in the BIOS. Dynamic C notices this and halts execution, reporting a run-time error to the user.

8.1.1 Error Code Ranges

The table below shows the range of error codes used by Dynamic C and the range available for a custom error handler to use. Please see section 8.2 on page 103 for more information on replacing the default error handler with a custom one.

Table 8-1. Dynamic C Error Types Ranges

Error Type	Meaning
0–127	Reserved for user-defined error codes.
128–255	Reserved for use by Dynamic C.

8.1.2 Fatal Error Codes

This table lists the fatal errors generated by Dynamic C.

Table 8-2. Dynamic C Fatal Errors

Error Type	Meaning
127 - 227	not used
228	Pointer store out of bounds
229	Array index out of bounds
230 - 233	not used
234	Domain error (for example, `acos(2)`)
235	Range error (for example, `tan(pi/2)`)
236	Floating point overflow
237	Long divide by zero
238	Long modulus, modulus zero
239	not used
240	Integer divide by zero
241	Unexpected interrupt
242	not used
243	Codata structure corrupted
244	Virtual watchdog timeout
245	XMEM allocation failed (xalloc call)
246	Stack allocation failed
247	Stack deallocation failed
248	not used
249	Xmem allocation initialization failed
250	No virtual watchdog timers available
251	No valid MAC address for board
252	Invalid cofunction instance
253	Socket passed as auto variable while running µC/OS-II
254	not used
255	

8.2 User-Defined Error Handler

Dynamic C allows replacement of the default error handler with a custom error handler. This is needed to add run-time error handling that would require treatment not supported by the default handler.

A custom error handler can also be used to change how existing run-time errors are handled. For example, the floating-point math libraries included with Dynamic C are written to allow for execution to continue after a domain or range error, but the default error handler halts with a run-time error if that state occurs. If continued execution is desired (the function in question would return a value of INF or whatever value is appropriate), then a simple error handler could be written to pass execution back to the program when a domain or range error occurs, and pass any other run-time errors to Dynamic C.

8.2.1 Replacing the Default Handler

To tell the BIOS to use a custom error handler, call this function:

```
void defineErrorHandler(void *errfcn)
```

This function sets the BIOS function pointer for run-time errors to the one passed to it.

When a run-time error occurs, `exception()` pushes onto the stack the information detailed in the table below.

Table 8-3. Stack setup for run-time errors

Address	Data at address
SP+0	Return address for error handler
SP+2	Error code
SP+4	Additional data (user-defined)
SP+6	XPC when `exception()` was called (upper byte)
SP+8	Address where `exception()` was called from

Then `exception()` calls the installed error handler. If the error handler passes the run-time error to Dynamic C (i.e. it is a fatal error and the system needs to be halted or reset), then registers must be loaded appropriately before calling the `_xexit` function.

Dynamic C expects the following values to be loaded:

Table 8-4. Register contents loaded by error handler before passing the error to Dynamic C

Register	Expected Value
H	XPC when `exception()` was called
L	Run-time error code
HL’	Address where `exception()` was called from

8.3 Run-Time Error Logging

Error logging is available as a BIOS enhancement for storing run-time exception history. It can be useful diagnosing problems in deployed Rabbit targets. To support error logging, the target must have battery-backed RAM.

8.3.1 Error Log Buffer

A circular buffer in extended RAM will be filled with the following information for each run-time error that occurs:

- The value of `SEC_TIMER` at the time of the error. This variable contains the number of seconds since 00:00:00 on January 1st 1980 if the real-time clock has been set correctly. This variable is updated by the periodic timer which is enabled by default. Z-World sets the real-time clock in the factory. When the BIOS starts on boards with batteries, it initializes `SEC_TIMER` to the value in the real-time clock.
- The address where the exception was called from. This can be traced to a particular function using the MAP file generated when a Dynamic C program is compiled.
- The exception type. Please see Table 8-2 on page 102 for a list of exception types.
- The value of all registers. This includes alternate registers, SP and XPC. This is a global option that is enabled by default.
- An 8-byte message. This is a global option that is disabled by default. The default error handler does nothing with this.
- A user-definable length of stack dump. This is a global option that is enabled by default.
- A one byte checksum of the entry.

8.3.1.1 Error Log Buffer Size

The size of the error log buffer is determined by the number of entries, the size of an entry, and the header information at the beginning of the buffer. The number of entries is determined by the macro `ERRLOG_NUM_ENTRIES` (default is 78). The size of each entry is dependent on the settings of the global options for stack dump, register dump and error message. The default size of the buffer is about 4K in extended RAM.

8.3.2 Initialization and Defaults

An initialization of the error log occurs when the BIOS is compiled, when cloning takes place or when the BIOS is loaded via the Rabbit Field Utility (RFU). By default, error logging is enabled with messages turned off, stack and register dumps turned on, and an error log buffer big enough for 78 entries.

The error log buffer contains header information as well as an entry for each run-time error. A debug start-up will zero out this header structure, but the run-time error entries can still be examined from Dynamic C using the static information in flash. The header is at the start of the error log buffer and contains:

- A status byte
- The number of errors since deployment
- The index of the last error
- The number of hardware resets since deployment
- The number of watchdog time-outs since deployment
- The number of software resets since deployment
- A checksum byte.

“Deployment” is defined as the first power up without the programming cable attached. Reprogramming the board through the programming cable, RFU, or RabbitLink and starting the program again without the programming cable attached is a new deployment.

8.3.3 Configuration Macros

These macros are defined at the top of `Bios/RabbitBios.c`.

ENABLE_ERROR_LOGGING

Default: 0. Disables error logging. Changing this to one in the BIOS enables error logging.

ERRLOG_USE_REG_DUMP

Default: 1. Include a register dump in log entries. Changing this to zero in the BIOS excludes the register dump in log entries.

ERRLOG_STACKDUMP_SIZE

Default: 16. Include a stack dump of size `ERRLOG_STACKDUMP_SIZE` in log entries. Changing this to zero in the BIOS excludes the stack dump in log entries.

ERRLOG_NUM_ENTRIES

Default: 78. This is the number of entries allowed in the log buffer.

ERRLOG_USE_MESSAGE

Default: 0. Exclude error messages from log entries. Changing this to one in the BIOS includes error messages in log entries The default error handler makes no use of this feature.

8.3.4 Error Logging Functions

The run-time error logging API consists of the following functions:

errlogGetHeaderInfo	Reads error log header and formats output.
errlogGetNthEntry	Loads `errLogEntry` structure with the Nth entry from the error log buffer. `errLogEntry` is a pre-allocated global structure.
errlogGetMessage	Returns a `NULL`-terminated string containing the 8 byte error message in `errLogEntry`.
errlogFormatEntry	Returns a `NULL`-terminated string containing basic information in `errLogEntry`.
errlogFormatRegDump	Returns a `NULL`-terminated string containing the register dump in `errLogEntry`.
errlogFormatStackDump	Returns a `NULL`-terminated string containing the stack dump in `errLogEntry`.
errlogReadHeader	Reads error log header into the structure `errlogInfo`.
ResetErrorLog	Resets the exception and restart type counts in the error log buffer header.

8.3.5 Examples of Error Log Use

To try error logging, follow the instructions at the top of the sample programs:

```
samples\ErrorHandling\Generate_runtime_errors.c
```

and

```
samples\ErrorHandling\Display_errorlog.c
```

9. Memory Management

Processor instructions can specify 16-bit addresses, giving a logical address space of 64K (65,536 bytes). Dynamic C supports a 1M physical address space (20-bit addresses).

An on-chip memory management unit (MMU) translates 16-bit addresses to 20-bit memory addresses. Four MMU registers (SEGSIZE, STACKSEG, DATASEG and XPC) divide and maintain the logical sections and map each section onto physical memory.

Any memory beyond the 16 bit address capability of the processor, whether flash or RAM, is called xmem and requires memory management techniques for access. In general, xmem flash access for program space is transparent to the user, but xmem accesses to RAM are not.

9.1 Memory Map

A typical Dynamic C memory mapping of logical and physical address space is shown in the figure below. The actual layout may be different depending on board type and compilation options. E.g., enabling separate I&D space will affect the memory map.

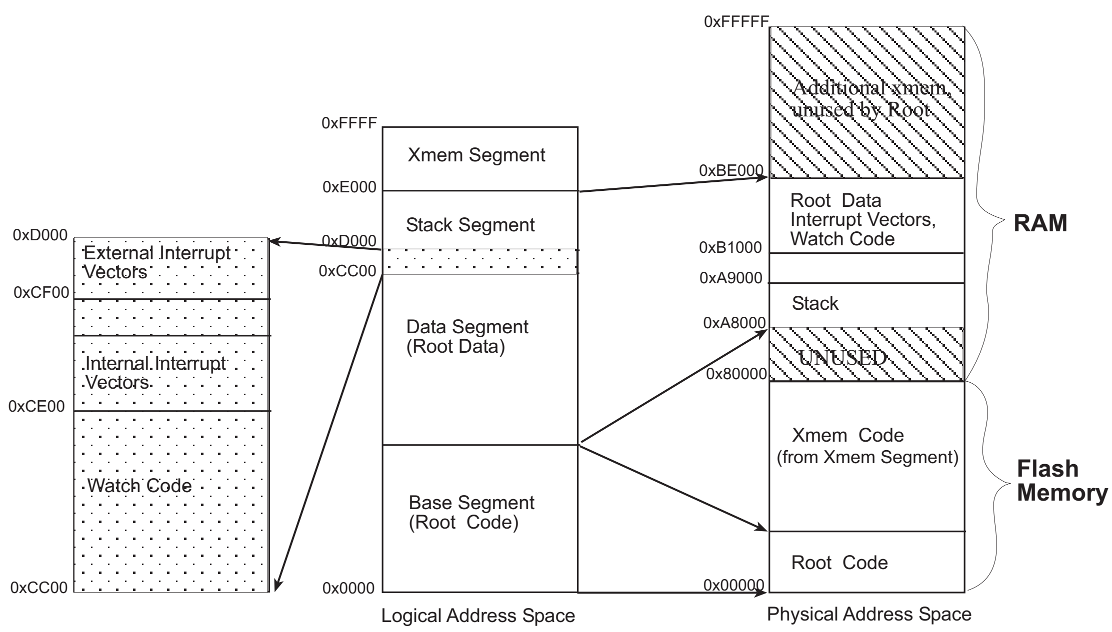


Figure 9-1. Dynamic C Memory Mapping

Figure 9-1 illustrates how the logical address space is divided and where code resides in physical memory. Both the static RAM and the flash memory are 128K in the diagram. Physical memory starts at address 0x00000 and flash memory is usually mapped to the same address. SRAM typically begins at address 0x80000.

If BIOS code runs from flash memory, the BIOS code starts in the root code section at address 0x00000 and fills upward. The rest of the root code will continue to fill upward immediately following the BIOS code. If the BIOS code runs from SRAM, the root code section, along with root data and stack sections, will start at address 0x80000.

9.1.1 Memory Mapping Control

The advanced user of Dynamic C can control how Dynamic C allocates and maps memory. For details on memory mapping, refer to any of the following:

- *Rabbit 2000 Microprocessor User's Manual*
- *Rabbit 3000 Microprocessor User's Manual*
- *Rabbit 3000 Designer's Handbook*
- Technical Note 202 "Rabbit Memory Management in a Nutshell"

All of the above documents are available at www.zworld.com.

9.2 Extended Memory Functions

A program can use many pages of extended memory. Under normal execution, code in extended memory maps to the logical address region E000H to FFFFH.

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.

To access xmem data, use function calls to exchange data between xmem and root memory. Use the Dynamic C functions `root2xmem()`, `xmem2root()` and `xmem2xmem()` to move blocks of data between logical memory and physical memory.

9.2.1 Code Placement in Memory

Code runs just as quickly in extended memory as it does in root memory, but calls to and returns from the functions in extended memory take a few extra machine cycles. Code placement in memory can be changed by the keywords `xmem` and `root`, depending on the type of code:

Pure Assembly Routines

Pure assembly functions may be placed in root memory or extended memory. Prior to Dynamic C version 7.10, pure assembly routines had to be in root memory.

C Functions

C functions may be placed in root memory or extended memory. Access to variables in C statements is not affected by the placement of the function. Dynamic C will automatically place C functions in extended memory as root memory fills. Short, frequently used functions may be declared with the `root` keyword 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.

All static variables, even those local to extended memory functions, are placed in root memory. Keep this in mind if the functions have many variables or large arrays. Root memory can fill up quickly.

10. The Flash File System

The Dynamic C file system, known as the filesystem mk II or simply as FS2, was designed to be used with a second flash memory or in SRAM.

FS2 allows:

- the ability to overwrite parts of a file.
- the simultaneous use of multiple device types.
- the ability to partition devices.
- efficient support for byte-writable devices.
- better performance tuning.
- a high degree of backwards compatibility with its predecessor.

> **NOTE:** Dynamic C's low-level flash memory access functions should not be used in the same area of the flash where the flash file system exists.

10.1 General Usage

The recommended use of a flash file system is for infrequently changing data or data rates that have writes on the order of tens of minutes instead of seconds. Rapidly writing data to the flash[i] could result in using up its write cycles too quickly. For example, consider a 256K flash with 64 blocks of 4K each. Using a flash with a maximum recommendation of 10,000 write cycles means a limit of 640,000 writes to the file system. If you are performing one write to the flash per second, in a little over a week you will use up its recommended lifetime.

Increase the useful lifetime and performance of the flash by buffering data before writing it to the flash. Accumulating 1000 single byte writes into one multi-byte write can extend the life of the flash by an average of 750 times. FS2 does not currently perform any in-memory buffering. If you write a single byte to a file, that byte will cause write activity on the device. This ensures that data is written to non-volatile storage as soon as possible. Buffering may be implemented within the application if possible loss of data is tolerable.

10.1.1 Maximum File Size

The maximum file size for an individual file depends on the total file system size and the number of files present. Each file requires at least two sectors: at least one for data and always one for metadata (for information used internally). There also needs to be two free sectors to allow for moving data around.

FS2 supports a total of 255 files, but storing a large number of small files is not recommended. It is much more efficient to have a few large ones.

i. All other code, including ISRs, is suspended while writing to flash.

10.1.2 Two Flash Boards

By default, when a board has two flash devices, Dynamic C will use only the first flash for code. The second flash is available for the file system unless the BIOS macro `USE_2NDFLASH_CODE` has been uncommented. This macro allocates the second flash to hold program code. The use of `USE_2NDFLASH_CODE` is not compatible with FS2.

10.1.3 Using SRAM

The flash file system can be used with battery-backed SRAM. Internally, RAM is treated like a flash device, except that there is no write-cycle limitation, and access is much faster. The file system will work without the battery backup, but would, of course, lose all data when the power went off.

Currently, the maximum size file system supported in RAM is about 200k. This limitation holds true even on boards with a 512k RAM chip. The limitation involves the placement of BIOS control blocks in the upper part of the lower 256k portion of RAM.

To obtain more RAM memory, `xalloc()` may be used. If `xalloc()` is called first thing in the program, the same memory addresses will always be returned. This can be used to store non-volatile data is so desired (if the RAM is battery-backed), however, it is not possible to manage this area using the file system.

Using FS2 increases flexibility, with its capacity to use multiple device types simultaneously. Since RAM is usually a scarce resource, it can be used together with flash memory devices to obtain the best balance of speed, performance and capacity.

10.1.4 Wear Leveling

The current code has a rudimentary form of wear leveling. When you write into an existing block it selects a free block with the least number of writes. The file system routines copy the old block into the new block adding in the users new data. This has the effect of evening the wear if there is a reasonable turnover in the flash files.

10.1.5 Low-Level Implementation

For information on the low-level implementation of the flash file system, refer to the beginning of the library file `FS2.LIB`.

10.1.6 Multitasking and the File System

The file system is not re-entrant. If using preemptive multitasking, ensure that only one thread performs calls to the file system, or implement locking around each call.

When using µC/OS-II, FS2 must be initialized first; that is, `fs_init()` must be called before `OSInit()` in the application code.

10.2 Application Requirements

Application requirements for using FS2 are covered in this section, including:

- which library to use
- which drivers to use
- defaults and descriptions for configuration macros
- detailed instructions for using the first flash

10.2.1 Library Requirements

The file system library must be compiled with the application:

```
#use "FS2.LIB"
```

For the simplest applications, this is all that is necessary for configuration. For more complex applications, there are several other macro definitions that may be used before the inclusion of FS2.LIB. These are:

```
#define FS_MAX_DEVICES   3
#define FS_MAX_LX        4
#define FS_MAX_FILES    10
```

These specify certain static array sizes that allow control over the amount of root data space taken by FS2. If you are using only one flash device (and possibly battery-backed RAM), and are not using partitions, then there is no need to set FS_MAX_DEVICES or FS_MAX_LX.

For more information on partitioning, please see section 10.4, "Setting up and Partitioning the File System," on page 115.

10.2.2 FS2 Configuration Macros

FS_MAX_DEVICES

This macro defines the maximum physical media. If it is not defined in the program code, FS_MAX_DEVICES will default to 1, 2, or 3, depending on the values of FS2_USE_PROGRAM_FLASH, XMEM_RESERVE_SIZE and FS2_RAM_RESERVE.

FS_MAX_LX

This macro defines the maximum logical extents. You must increase this value by 1 for each new partition your application creates. It this is not defined in the program code it will default to FS_MAX_DEVICES.

For a description of logical extents please see section 10.4.2, "Logical Extents (LX)," on page 116.

FS_MAX_FILES

This macro is used to specify the maximum number of files that are allowed to coexist in the entire file system. Most applications will have a fixed number of files defined, so this parameter can be set to that number to avoid wasting root data memory. The default is 6 files. The maximum value for this parameter is 255.

FS2_RAM_RESERVE

This BIOS-defined macro determines the amount of space used for FS2 in RAM. If some battery-backed RAM is to be used by FS2, then this macro must be modified to specify the amount of RAM to reserve. The memory is reserved near the top of RAM. Note that this RAM will be reserved whether or not the application actually uses FS2.

Prior to Dynamic C 7.06 this macro was defined as the number of bytes to reserve and had to be a multiple of 4096. It is now defined as the number of blocks to reserve, with each block being 4096 bytes.

FS2_SHIFT_DOESNT_UPDATE_FPOS

If this macro is defined before the `#use fs2.lib` statement in an application, multiple file descriptors can be opened, but their current position will not be updated if `fshift()` is used.

FS2_USE_PROGRAM_FLASH

The number of kilobytes reserved in the first flash for use by FS2. The default is zero. The actual amount of flash used by FS2 is determined by the minimum of this macro and `XMEM_RESERVE_SIZE`.

XMEM_RESERVE_SIZE

This BIOS-defined macro is the number of bytes (which must be a multiple of 4096) reserved in the first flash for use by FS2 and possibly other customer-defined purposes. This is defined in the BIOS as 0x0000. Memory set aside with `XMEM_RESERVE_SIZE` will NOT be available for xmem code.

10.2.3 FS2 and Use of the First Flash

To use the first flash in FS2, follow these steps:

1. Define `XMEM_RESERVE_SIZE` (currently set to 0x0000 in the BIOS) to the number of bytes to allocate in the first flash for the file system.
2. Define `FS2_USE_PROGRAM_FLASH` to the number of KB (1024 bytes) to allocate in the first flash for the file system. Do this in the application code before `#use "fs2.lib"`.
3. Obtain the LX number of the first flash: Call `fs_get_other_lx()` when there are two flash memories; call `fs_get_flash_lx()` when there is only one.
4. If desired, create additional logical extents by calling the FS2 function `fs_setup()` to further partition the device. This function can also change the logical sector sizes of an extent. Please see the function description for `fs_setup()` in the *Dynamic C Function Reference Manual* for more information.

10.2.3.1 Example Code Using First Flash in FS2

If the target board has two flash memories, the following code will cause the file system to use the first flash:

```
FSLXnum flash1;                          //  logical extent number
File f;                                  //  struct for file information

flash1 = fs_get_other_lx();
if (flash1) {
   fs_set_lx(flash1, flash1);
   fcreate(&f, 10);
   . . .
}
```

To obtain the logical extent number for a one flash board, `fs_get_flash_lx()` must be called instead of `fs_get_other_lx()`.

10.3 File System API Functions

These functions are defined in `FS2.LIB`. For more information please see the *Dynamic C Function Reference Manual*.

Table 10-1. FS2 API

Command	Description
`fs_setup (FS2)`	Alters the initial default configuration.
`fs_init (FS2)`	Initialize the internal data structures for the file system.
`fs_format (FS2)`	Initialize flash and the internal data structures.
`lx_format`	Formats a specified logical extent (LX).
`fs_set_lx (FS2)`	Sets the default LX numbers for file creation.
`fs_get_lx (FS2)`	Returns the current LX number for file creation.
`fcreate (FS2)`	Creates a file and open it for writing.
`fcreate_unused (FS2)`	Creates a file with an unused file number.
`fopen_rd (FS2)`	Opens a file for reading.
`fopen_wr (FS2)`	Opens a file for writing (and reading).
`fshift`	Removes specified number of bytes from beginning of file.
`fwrite (FS2)`	Writes to a file starting at “current position.”
`fread (FS2)`	Reads from the current file pointer.
`fseek (FS2)`	Moves the read/write pointer.
`ftell (FS2)`	Returns the current offset of the file pointer.
`fs_sync (FS2)`	Flushes any buffers retained in RAM to the underlying hardware device.
`fflush (FS2)`	Flushes buffers retained in RAM and associated with the specified file to the underlying hardware device.
`fs_get_flash_lx (FS2)`	Returns the LX number of the preferred flash device (the 2nd flash if available).
`fs_get_lx_size (FS2)`	Returns the number of bytes of the specified LX.
`fs_get_other_lx (FS2)`	Returns LX # of the non-preferred flash (usually the first flash).
`fs_get_ram_lx (FS2)`	Return the LX number of the RAM file system device.
`fclose`	Closes a file.
`fdelete (FS2)`	Deletes a file.

10.3.1 FS2 API Error Codes

The library `ERRNO.LIB` contains a list of all possible error codes returnable by the FS2 API. These error codes mostly conform to POSIX standards. If the return value indicates an error, then the global variable `errno` may be examined to determine a more specific reason for the failure. The possible errno codes returned from each function are documented with the function.

10.4 Setting up and Partitioning the File System

This step merits some thought before plowing ahead. The context within which the file system will be used should be considered. For example, if the target board contains both battery-backed SRAM and a second flash chip, then both types of storage may be used for their respective advantages. The SRAM might be used for a small application configuration file that changes frequently, and the flash used for a large log file.

FS2 automatically detects the second flash device (if any) and will also use any SRAM set aside for the file system (if `FS2_RAM_RESERVE` is set).

10.4.1 Initial Formatting

The filesystem must be formatted when it is first used. The only exception is when a flash memory device is known to be completely erased, which is the normal condition on receipt from the factory. If the device contains random data, then formatting is required to avoid the possibility of some sectors being permanently locked out of use.

Formatting is also required if any of the logical extent parameters are changed, such as changing the logical sector size or re-partitioning. This would normally happen only during application development.

The question for application developers is how to code the application so that it formats the filesystem only the first time it is run. There are several approaches that may be taken:

- A special program that is loaded and run once in the factory, before the application is loaded. The special program prepares the filesystem and formats it. The application never formats; it expects the filesystem to be in a proper state.
- The application can perform some sort of consistency check. If it determines an inconsistency, it calls format. The consistency check could include testing for a file that should exist, or by checking some sort of "signature" that would be unlikely to occur by chance.
- Have the application prompt the end-user, if some form of interaction is possible.
- A combination of one or more of the above.
- Rely on a flash device being erased. This would be OK for a production run, but not suitable if battery-backed SRAM was being used for part of the filesystem.

10.4.2 Logical Extents (LX)

The presence of both “devices” causes an initial default configuration of two logical extents (a.k.a., LXs) to be set up. An LX is analogous to disk partitions used in other operating systems. It represents a contiguous area of the device set aside for file system operations. An LX contains sectors that are all the same size, and all contiguously addressable within the one device. Thus a flash device with three different sector sizes would necessitate at least three logical extents, and more if the same-sized sectors were not adjacent.

Files stored by the file system are comprised of two parts: one part contains the actual application data, and the other is a fixed size area used to contain data controlled by the file system in order to track the file status. This second area, called metadata, is analogous to a “directory entry” of other operating systems. The metadata consumes one sector per file.

The data and metadata for a file are usually stored in the same LX, however they may be separated for performance reasons. Since the metadata needs to be updated for each write operation, it is often advantageous to store the metadata in battery-backed SRAM with the bulk of the data on a flash device.

10.4.2.1 Specifying Logical Extents

When a file is created, the logical extent(s) to use for the file are defined. This association remains until the file is deleted. The default LX for both data and metadata is the flash device (LX #1) if it exists; otherwise the RAM LX. If both flash and RAM are available, LX #1 is the flash device and LX #2 is the RAM.

When creating a file, the associated logical extents for the data and the metadata can be changed from the default by calling `fs_set_lx()`. This functions takes two parameters, one to specify the LX for the metadata and the other to specify the LX for the data. Thereafter, all created files are associated with the specified LXs until a new call to `fs_set_lx()` is made. Typically, there will be a call to `fs_set_lx()` before each file is created, in order to ensure that the new file gets created with the desired associations. The file creation function, `fcreate()`, may be used to specify the LX for the metadata by providing a valid LX number in the high byte of the function’s second parameter. This will override any LX number set for the metadata in `fs_set_lx()`.

10.4.2.1.1 Further Partitioning

The initial default logical extents can be divided further. This must be done before calling `fs_init()`. The function to create sub-partitions is called `fs_setup()`. This function takes an existing LX number, divides that LX according to the given parameters, and returns a newly created LX number. The original partition still exists, but is smaller because of the division. For example, in a system with LX#1 as a flash device of 256K and LX#2 as 4K of RAM, an initial call to `fs_setup()` might be made to partition LX#1 into two equal sized extents of 128K each. LX#1 would then be 128K (the first half of the flash) and LX#3 would be 128K (the other half). LX#2 is untouched.

Having partitioned once, `fs_setup()` may be called again to perform further subdivision. This may be done on any of the original or new extents. Each call to `fs_setup()` in partitioning mode increases the total number of logical extents. You will need to make sure that `FS_MAX_LX` is defined to a high enough value that the LX array size is not exceeded.

While developing an application, you might need to adjust partitioning parameters. If any parameter is changed, FS2 will probably not recognize data written using the previous parameters. This problem is common to most operating systems. The "solution" is to save any desired files to outside the file system before changing its organization; then after the change, force a format of the file system.

10.4.3 Logical Sector Size

`fs_setup()` can also be used to specify non-default logical sector (LS) sizes and other parameters. FS2 allows any logical sector size between 64 and 8192 bytes, providing the LS size is an exact power of 2. Each logical extent, including sub-partitions, can have a different LS size. This allows some performance optimization. Small LSs are better for a RAM LX, since it minimizes wasted space without incurring a performance penalty. Larger LSs are better for bulk data such as logs. If the flash physical sector size (i.e. the actual hardware sector size) is large, it is better to use a correspondingly large LS size. This is especially the case for byte-writable devices. Large LSs should also be used for large LXs. This minimizes the amount of time needed to initialize the file system and access large files. As a rule of thumb, there should be no more than 1024 LSs in any LX. The ideal LS size for RAM (which is the default) is 128 bytes. 256 or 512 can also be reasonable values for some applications that have a lot of spare RAM.

Sector-writable flash devices require: LS size ≥ PS size. Byte-writable devices, however, may use any allowable logical sector size, regardless of the physical sector size.

Sample program `Samples\FileSystem\FS2DEMO2` illustrates use of `fs_setup()`. This sample also allows you to experiment with various file system settings to obtain the best performance.

FS2 has been designed to be extensible in order to work with future flash and other non-volatile storage devices. Writing and installing custom low-level device drivers is beyond the scope of this document, however see `FS2.LIB` and `FS_DEV.LIB` for hints.

10.5 File Identifiers

There are two ways to identify a particular file in the file system: file numbers and file names.

10.5.1 File Numbers

The file number uniquely identifies a file within a logical extent. File numbers must be unique within the entire file system. FS2 accepts file numbers in word format:

```
typedef word FileNumber
```

The low-order byte specifies the file number and the high-order byte specifies the LX number of the metadata (1 through number of LXs). If the high-order byte is zero, then a suitable "default" LX will be located by the file system. The default LX will default to 1, but will be settable via a `#define`, for file creation. For existing files, a high-order byte of zero will cause the file system to search for the LX that contains the file. This will require no or minimal changes to existing customer code.

Only the metadata LX may be specified in the file number. This is called a "fully-qualified" file number (FQFN). The LX number always applies to the file metadata. The data can reside on a different LX, however this is always determined by FS2 once the file has been created.

10.5.2 File Names

There are several functions in ZSERVER.LIB that can be used to associate a descriptive name with a file. The file must exist in the flash file system before using the auxiliary functions listed in the following table. These functions were originally intended for use with an HTTP or FTP server, so some of them take a parameter called servermask. To use these functions for file naming purposes only, this parameter should be SERVER_USER.

For a detailed description of these functions please refer to the *Dynamic C's TCP/IP User's Manual*, or use <Ctrl-H> in Dynamic C to use the Library Lookup feature.

Table 10-2. Flash File System Auxiliary Functions

Command	Description
sspec_addfsfile	Associate a name with the flash file system file number. The return value is an index into an array of structures associated with the named files.
sspec_readfile	Read a file represented by the return value of sspec_addfsfile into a buffer.
sspec_getlength	Get the length (number of bytes) of the file.
sspec_getfileloc	Get the file system file number (1- 255). Cast return value to FILENUMBER.
sspec_findname	Find the index into the array of structures associated with named files of the file that has the specified name.
sspec_getfiletype	Get file type. For flash file system files this value will be SSPEC_FSFILE.
sspec_findnextfile	Find the next named file in the flash file system, at or following the specified index, and return the index of the file.
sspec_remove	Remove the file name association.
sspec_save	Saves to the flash file system the array of structures that reference the named files in the flash file system.
sspec_restore	Restores the array of structures that reference the named files in the flash file system.

10.6 Skeleton Program Using FS2

The following program uses some of the FS2 API. It writes several strings into a file, reads the file back and prints the contents to the STDIO window.

```
#use "FS2.LIB"
#define TESTFILE 1

main()
{
   File file;
   static char buffer[256];

   fs_init(0, 0);

   if (!fcreate(&file, TESTFILE) && fopen_wr(&file,TESTFILE))
   {
      printf("error opening TESTFILE %d\n", errno);
      return -1;
   }

   fseek(&file, 0, SEEK_END);
   fwrite(&file,"hello",6);
   fwrite(&file,"12345",6);
   fwrite(&file,"67890",6);
   fseek(&file, 0, SEEK_SET);

   while(fread(&file,buffer,6)>0) {
      printf("%s\n",buffer);
   }

   fclose(&file);

}
```

For a more robust program, more error checking should be included. See the sample programs in the `Samples\FILESYSTEM` folder for more complex examples, including error checking, formatting, partitioning and other new features.

11. Using Assembly Language

This chapter gives the rules for mixing assembly language with Dynamic C code. A reference guide to the Rabbit Instruction Set is available from the **Help** menu of Dynamic C and is also documented in the *Rabbit 2000/3000 Microprocessor Instruction Reference Manual*.

11.1 Mixing Assembly and C

Dynamic C permits assembly language statements to be embedded in C functions and/or entire functions to be written in assembly language. C statements may also be embedded in assembly code. C-language variables may be accessed by the assembly code.

11.1.1 Embedded Assembly Syntax

Use the `#asm` and `#endasm` directives to place assembly code in Dynamic C programs. For example, the following function will add two 64-bit numbers together. 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`).

```
void eightadd( char *ch1, char *ch2 ){
#asm
   ld   hl,(sp+@SP+ch2)      ; get source pointer
   ex   de,hl                ; save in register DE
   ld   hl,(sp+@SP+ch1)      ; get destination pointer
   ld   b,8                  ; number of bytes
   xor  a                    ; clear carry
   loop:
   ld   a,(de)               ; ch2 source byte
   adc  a,(hl)               ; add ch1 byte
   ld   (hl),a               ; store result to ch1 address
   inc  hl                   ; increment ch1 pointer
   inc  de                   ; increment ch2 pointer
   djnz loop                 ; do 8 bytes
   ; ch1 now points to 64 bit result
#endasm
}
```

The keywords `debug` and `nodebug` can be placed on the same line as `#asm`. Assembly code blocks are `nodebug` by default. This saves space and unnecessary calls to the debugger kernel.

All blocks of assembly code within a C function are assembled in nodebug mode. The only exception to this is when a block of assembly code is explicitly marked with `debug`. Any blocks marked `debug` will be assembled in debug mode even if the enclosing C function is marked `nodebug`.

11.1.2 Embedded C Syntax

A C statement may be placed within assembly code by placing a “c” in column 1. Note that whichever registers are used in the embedded C statement will be changed.

```
#asm
InitValues::
c  start_time = 0;
c  counter = 256;
   ret
#endasm
```

11.1.3 Setting Breakpoints in Assembly

There are two ways to enable breakpoint support in a block of assembly code.

One way is to explicitly mark the assembly block as `debug` (the default condition is `nodebug`). This causes the insertion of “rst 0x28” instructions between each assembly instruction. These rst 0x28 instructions may cause jump relative (i.e., `jr`) instructions to go out of range, but this problem can be solved by changing the relative jump (`jr`) to an absolute jump (`jp`).

The other way to enable breakpoint support in a block of assembly code is to add a C statement before the desired assembly instruction. Note that the assembly code must be contained in a debug C function in order to enable C code debugging. Below is an example.

```
debug dummyfunction() {
#asm
function::
...
label:
...
c ;              //  add line of C code to permit a breakpoint before jump relative
jr nc, label
ret
#endasm
}
```

NOTE: Single stepping through assembly code is always allowed if the assembly window is open.

11.2 Assembler and Preprocessor

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:

Table 11-1. Operators Not Supported By The Assembler

Operator Symbol	**Operator Description**
`?:`	conditional
`[ ]`	array index
`.`	dot
`->`	points to
`*`	dereference

11.2.1 Comments

C-style comments are allowed in embedded assembly code. The assembler will ignore comments beginning with

`;` — text from the semicolon to the end of line is ignored.
`//` — text from the double forward slashes to the end of line is ignored.
`/* ... */` — text between slash-asterisk and asterisk-slash is ignored.

11.2.2 Defining Constants

Constants may be created and defined in assembly code with the assembly language keyword `db` (**d**efine **b**yte). `db` should be followed immediately by numerical values and strings separated by commas. For example, each of the following lines all define the string "`ABC`."

```
db 'A', 'B', 'C'
db "ABC"
db 0x41, 0x42, 0x43
```

The numerical values and characters in strings are used to initialize sequential byte locations.

If separate I&D space is enabled, assembly constants should either be put in their own assembly block with the `const` keyword or be done in C.

```
#asm const
   myrootconstants::
   db 0x40, 0x41, 0x42
#endasm
```

or

```
const char myrootconstants[] = {‘\x40’, ‘\x41’, ‘\x42’}
```

If separate I&D space is enabled, `db` places bytes in the base segment of the data space when it is used with `const`. If the `const` keyword is absent, i.e.,

```
#asm
   myrootconstants::
   db 0x40, 0x41, 0x42
#endasm
```

the bytes are placed somewhere in the instruction space. If separate I&D space is disabled (the default condition), the bytes are placed in the base segment (aka, root segment) interspersed with code.

Therefore, so that data will be treated as data when referenced in assembly code, the `const` keyword must be used when separate I&D space is enabled. For example, this won't work correctly without `const`:

```
#asm const
label::
   db 0x5a
#endasm
main(){
   ;
#asm
   ld a,(label)      // ld 0x5a to reg a
#endasm
}
```

The assembly language keyword `dw` defines 16-bit words, least significant byte first. The keyword `dw` should be followed immediately by numerical values:

```
dw 0x0123, 0xFFFF, xyz
```

This example defines three constants. The first two constants are literals, and the third constant is the address of variable `xyz`.

The numerical values initialize sequential word locations, starting at the current code address.

11.2.3 Multiline Macros

The Dynamic C preprocessor has a special feature to allow multiline macros in assembly code. The preprocessor expands macros before the assembler parses any text. Putting a `$\` at the end of a line inserts a new line in the text. This only works in assembly code. Labels and comments are not allowed in multiline macros.

```
#define SAVEFLAG  $\
   ld   a,b  $\
   push af  $\
   pop  bc
#asm
   ...
   ld  b,0x32
   SAVEFLAG
   ...
#endasm
```

11.2.4 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 (i.e., code before the `#asm` or after the `#endasm` directive).

Unless it is followed immediately by the assembly language 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.

11.2.5 Special Symbols

This table lists special symbols that can be used in an assembly language expression.

Table 11-2. Special Assembly Language Symbols

Symbol	Description
`@SP`	Indicates the amount of stack space (in bytes) used for stack-based variables. This does not include arguments.
`@PC`	Constant for the current code location. For example: `ld hl, @PC` loads the code address of the instruction. ld hl,@PC+3 loads the address after the instruction since it is a 3 byte instruction.
`@RETVAL`	Evaluates the offset from the *frame reference point* to the stack space reserved for the `struct` function returns. See Section 11.4.1.1 on page 129 for more information.
`@LENGTH`	Determines the next reference address of a variable plus its size.

11.2.6 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 or static 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.

The following list of processor register names are reserved and may not be used as C variable names in assembly: A, B, C, D, E, F, H, L, AF, HL, DE, BC, IX, IY, SP, PC, XPC, IP, IIR and EIR. Both upper and lower case instances are reserved.

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
   };
};
```

11.3 Stand-Alone Assembly Code

A stand-alone assembly function is one that is defined outside the context of a C language function.

A stand-alone assembly function has no `auto` variables and no formal parameters. It can, however, have arguments passed to it by the calling function. 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, unsigned int, char, pointer`), the primary register is HL (with register H containing the most significant byte). If the first argument has four bytes (`long, unsigned long, float`), the primary register is BC:DE (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, pushed last—are pushed on the stack.

C function values return in the primary register, if they have four or fewer bytes, either in HL or BC:DE.

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.

11.3.1 Stand-Alone Assembly Code in Extended Memory

Stand-alone assembly functions may be placed in extended memory by adding the `xmem` keyword as a qualifier to `#asm`, as shown below. Care needs be taken so that branch instructions do not jump beyond the current xmem window. To help prevent such bad jumps, the compiler limits xmem assembly blocks to 4096 bytes. Code that branches to other assembly blocks in xmem should always use `ljp` or `lcall`.

```
#asm xmem
main::
...
lcall fcn_in_xmem
...
lret
#endasm

#asm xmem
fcn_in_xmem::
...
lret
#endasm
```

11.3.2 Example of Stand-Alone Assembly Code

The stand-alone assembly function `foo()` can be called from a Dynamic C function.

```
int foo ( int );      // A function prototype can be declared for stand-alone
                      // assembly functions, which will cause the compiler
                      // to perform the appropriate type-checking.
main(){
   int i,j;
   i=1;
   j=foo(i);
}
#asm
foo::
...
ld hl,2            // The return value expected by main() is put
ret                   // in HL just before foo() returns
#endasm
```

The entire program can be written in assembly.

```
#asm
main::
...
ret
#endasm
```

11.4 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.

11.4.1 The Stack Frame

The purpose and structure of a *stack frame* should be understood before writing embedded assembly code. 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 for a particular function. If the IX register is used for a frame reference pointer, the previous value of IX is also kept in the stack frame. Figure 11.1 shows the general appearance of a stack frame.

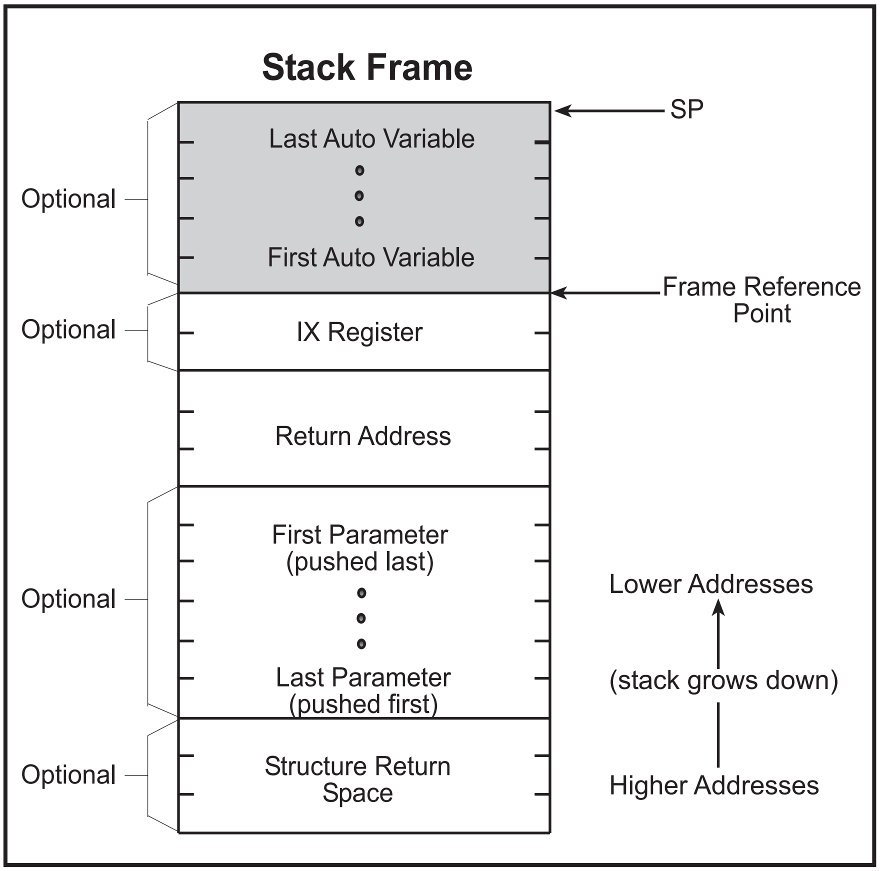


Figure 11.1. General Appearance of Assembly Code Stack Frame

The return address is always necessary. The presence of auto variables depends on the function definition. 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` variables. The assembler symbol `@SP` represents the size of this area.

11.4.1.1 The Frame Reference Point

The frame reference point is a location in the stack frame that immediately follows the function's return address. The IX register may be used as a pointer to this location by putting the keyword `useix` before the function, or the request can be specified globally by the compiler directive `#useix`. The default is `#nouseix`. If the IX register is used as a frame reference pointer, its previous value is pushed on the stack after the function's return address. The frame reference point moves to encompass the saved IX value.

11.4.2 Embedded Assembly Example

The purpose of the following sample program, `asm1.c`, is to show the different ways to access stack-based variables from assembly code.

```
void func(char ch, int i, long lg);

main(){
   char ch;
   int i;
   long lg;

   ch = 0x11;
   i = 0x2233;
   lg = 0x44556677L;

   func(ch,i,lg);
}

void func(char ch, int i, long lg){
   auto int x;
   auto int z;

   x = 0x8888;
   z = 0x9999;

#asm
   // This is equivalent to the C statement: x = 0x8888
   ld hl, 0x8888
   ld (sp+@SP+x), hl

   // This is equivalent to the C statement: z = 0x9999
   ld hl, 0x9999
   ld (sp+@SP+z), hl

   // @SP+i gives the offset of i from the stack frame on entry.
   // On the Rabbit, this is how HL is loaded with the value in i.
   ld   hl,(sp+@SP+i)

   // This works if func() is useix; however, if the IX register
   // has been changed by the user code, this code will fail.
   ld   hl,(ix+i)

   // This method works in either case because the assembler
   // adjusts the constant @SP, so changing the function to
   // nouseix with the keyword nouseix, or the compiler
   // directive #nouseix will not break the code. But, if SP has
   // been changed by user code, (e.g. a push) it won't work.
   ld   hl,(sp+@SP+lg+2)
   ld   b,h
   ld   c,L
   ld   hl,(sp+@SP+lg)
   ex   de,hl
#endasm
}
```

11.4.2.1 The Disassembled Code Window

A program may be debugged at the assembly level by opening the “Disassembled Code” window. Single stepping and breakpoints are supported in this window. When the “Disassembled Code” window is open, single stepping occurs instruction by instruction rather than statement by statement. The figure below shows the “Disassembled Code” window for the example code, `asm1.c`.

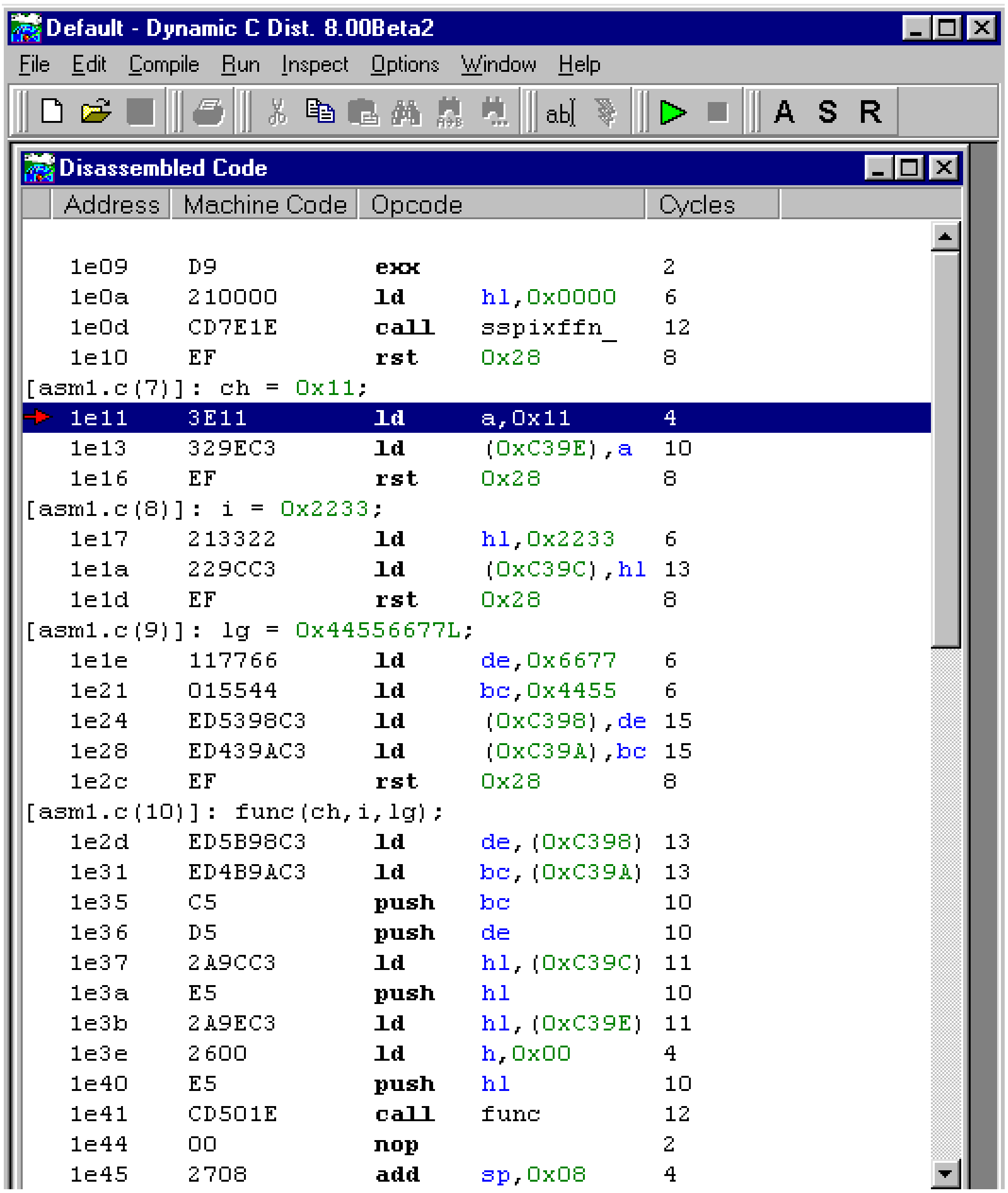


Figure 11.2. Disassembled Code Windows

11.4.2.2 Instruction Cycle Time

The “Disassembled Code” window shows the memory address on the far left, followed by the code bytes for the instruction at the address, followed by the mnemonics for the instruction. The last column shows the number of cycles for the instruction, assuming no wait states. The total cycle time for a block of instructions will be shown at the lowest row in the block in the cycle-time column, if that block is selected and highlighted with the mouse. The total assumes one execution per instruction, so the user must take looping and branching into consideration when evaluating execution times.

11.4.3 Local Variable Access

Accessing static local variables is simple because the symbol evaluates to the address directly. The following code shows, for example, how to load static variable `y` into HL.

```
ld hl,(y)              ; load hl with contents of y
```

11.4.3.1 Using the IX Register

Access to stack-based local variables is fairly inefficient. The efficiency improves if IX is used as a frame pointer. The arguments will have slightly different offsets because of the additional two bytes for the saved IX register value.

Now, access to stack variables is easier. Consider, for example, how to load `ch` into register A.

```
ld  a,(ix+ch)          ; a ← ch
```

The IX+offset load instruction takes 9 clock cycles and opcode is three bytes. If the program needs to load a four-byte variable such as **`lg`**, the IX+offset instructions are as follows.

```
ld hl,(ix+lg+2)          ; load LSB of lg
ld b,h                   ; longs are normally stored in BC:DE
ld c,L
ld hl,(ix+lg)            ; load MSB of lg
ex de,hl
```

This takes a total of 24 cycles.

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 accessing variables out of this range. The `@SP` symbol may be used even if IX is the frame reference pointer.

11.4.3.2 Functions in Extended Memory

If the `xmem` keyword is present, Dynamic C compiles the function to extended memory. Otherwise, Dynamic C determines where to compile the function. Functions compiled to extended memory have a 3-byte return address instead of a 2-byte return address.

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.

11.5 C Calling Assembly

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.

11.5.1 Passing Parameters

When a program calls a function from C, it puts the first argument into HL (if it has one or two bytes) with register H containing the most significant byte. If the first argument has four bytes, it goes in BC:DE (with register B containing the most significant byte). Only the first argument is put into the primary register, while *all* arguments—including the first, pushed last—are pushed on the stack.

11.5.2 Location of Return Results

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, unsigned int, char`, or a pointer, return the result in HL (register H contains the most significant byte). If the result is a `long, unsigned long,` 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 stand-alone assembly routine, however, must load the primary register with the return value before the `ret` instruction.

11.5.2.1 Returning a Structure

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). Dynamic C functions containing embedded assembly code may use a C `return` statement to return a value. A stand-alone assembly routine, however, must store the return value in the structure return space on the stack before returning.

Inline 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.

11.6 Assembly Calling C

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, unsigned int, char`, and pointers, or BCDE for `long, unsigned long,` 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 a `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.

11.7 Interrupt Routines in Assembly

Interrupt Service Routines (ISRs) may be written in Dynamic C (declared with the keyword `interrupt`). But since an assembly routine may be more efficient than the equivalent C function, assembly is more suitable for an ISR. Even if the execution time of an ISR is not critical, the latency of one ISR may affect the latency of other ISRs.

Either stand-alone assembly code or embedded assembly code may be used for ISRs. The benefit of embedding assembly code in a C-language ISR 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 stand-alone assembly routine needs to save and restore only the registers it uses.

11.7.1 Steps Followed by an ISR

The CPU loads the Interrupt Priority register (IP) with the priority of the interrupt before the ISR is called. This effectively turns off interrupts that are of the same or lower priority. Generally, the ISR performs the following actions:

1. Save all registers that will be used, i.e. push them on the stack. Interrupt routines written in C save all registers automatically. Stand-alone assembly routines must push the registers explicitly.
2. Determine the cause of the interrupt. Some devices map multiple causes to the same interrupt vector. An interrupt handler must determine what actually caused the interrupt.
3. Remove the cause of the interrupt.
4. 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. Stand-alone assembly routines must pop the registers explicitly.
6. Restore the interrupt priority level so that other interrupts can get the attention of the CPU. ISRs written in C restore the interrupt priority level automatically when the function returns. However, stand-alone assembly ISRs must restore the interrupt priority level explicitly by calling `ipres`.

   The interrupt priority level must be restored 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. There are three types of interrupt returns: `ret, reti`, and `retn`.

The value in IP is shown in the status bar at the bottom of the Dynamic C window. If a breakpoint is encountered, the IP value shown on the status bar reflects the saved context of IP from just before the breakpoint.

11.7.2 Modifying Interrupt Vectors

Prior to Dynamic C 7.30, interrupt vector code could be modified directly. By reading the internal and external interrupt registers, IIR and EIR, the location of the vector could be calculated and then written to because it was located in RAM. This method will not work if separate I&D space is enabled because the vectors must be located in flash. To accommodate separate I&D space, the way interrupt vectors are set up and modified has changed slightly. Please see the *Rabbit 3000 Designer's Handbook* for detailed information about how the interrupt vectors are set up. This section will discuss how to modify the interrupt vectors after they have been set up.

For backwards compatibility, "modifiable" vector relays are provided in RAM. In C, they can be accessed through the SetVectIntern and SetVectExtern functions. In assembly, they are accessed through `INTVEC_BASE` + <vector offset> or `XINTVEC_BASE` + <vector offset>. The values for <vector offset> are defined in `sysio.lib`, and are listed here for convenience.

Table 11-3. Internal Interrupts and their offset from `INTVEC_BASE`

PERIODIC_OFS	SERA_OFS
RST10_OFS	SERB_OFS
RST18_OFS	SERC_OFS
RST20_OFS	SERD_OFS
RST28_OFS	SERE_OFS
RST38_OFS	SERF_OFS
SLAVE_OFS	QUAD_OFS
TIMERA_OFS	INPUTCAP_OFS
TIMERB_OFS	

Table 11-4. External Interrupts and their offset from `XINTVEC_BASE`

EXT0_OFS
EXT1_OFS

The following example from RS232.LIB illustrates the new I&D space compatible way of modifying interrupt vectors.

The following code fragment to set up the interrupt service routine for the periodic interrupt from Dynamic C 7.25 is **not compatible** with separate I&D space:

```
#asm xmem
   ;*** Old method ***
   ld a,iir                    ; get the offset of interrupt table
   ld h,a
   ld l,0x00
   ld iy,hl
   ld (iy),0c3h                ; jp instruction entry
   inc iy
   ld hl,periodic_isr          ; set service routine
   ld (iy),hl
#endasm
```

The following code fragment shows an I&D space compatible method for setting up the ISR for the periodic interrupt in Dynamic C 7.30:

```
#asm xmem
   ;*** New method ***
   ld a, 0xc3                          ;jp instruction entry
   ld hl, periodic_isr                 ;set service routine
   ld (INTVEC_BASE+PERIODIC_OFS), a    ;write to the interrupt table
   ld (INTVEC_BASE+PERIODIC_OFS+1), hl
#endasm
```

When separate I&D space is enabled, INTVEC_BASE points to a proxy interrupt vector table in RAM that is modifiable. The code above assumes that the actual interrupt vector table pointed to by the IIR is set up to point to the proxy vector. When separate I&D space is disabled, INTVEC_BASE and the IIR point to the same location. The code above is an example only, the default configuration for the periodic interrupt is **not** modifiable.

The following example from RS232.LIB illustrates the new I&D space compatible way of modifying interrupt vectors.

The following function serAclose() from Dynamic C 7.25, is not compatible with separate I&D space:

```
#asm xmem
serAclose::
   ld a,iir                          ; hl=spaisr_start, de={iir,0xe0}
   ld h,a
   ld l,0xc0
   ld a,0xc9                         ; ret in first byte
   ipset 1
   ld (hl),a
   ld a,0x00                         ; disable interrupts for port
   ld (SACRShadow), a
   ioi ld (SACR), a
   ipres
   lret
#endasm
```

This version of serAclose() in Dynamic C 7.30 is compatible with separate I&D space:

```
#asm xmem
serAclose::
   ld a, 0xc9
   ipset 1
   ld (INTVEC_BASE + SERA_OFS), a    ; ret in first byte of spaisr_start
   ld a, 0x00                        ; disable interrupts for port
   ld (SACRShadow),a
ioi ld (SACR),a
   ipres
   lret
#endasm
```

If separate I&D space is enabled, using the modifiable interrupt vector proxy in RAM adds about 80 clock cycles of overhead to the execution time of the ISR. To avoid that, the preferred way to set up interrupt vectors is to use the new keyword, `interrupt_vector`, to set up the vector location at compile time.

When compiling with separate I&D space, modify applications that use `SetVectIntern()`, `SetVectExtern2000()` or `SetVectExtern3000()` to use `interrupt_vector` instead.

The following code, from `/Samples/TIMERB/TIMER_B.C`, illustrates the change that should be made.

```
void main()
{
   . . .
#if __SEPARATE_INST_DATA__
   interrupt_vector timerb_intvec timerb_isr;
#else
   SetVectIntern(0x0B, timerb_isr);     // set up ISR
#endif
   . . .
}
```

If `interrupt_vector` is used multiple times for the same interrupt vector, the last one encountered by the compiler will override all previous ones.

`interrupt_vector` is syntactic sugar for using the origin directives and assembly code. For example, the line:

```
interrupt_vector timerb_intvec timerb_isr;
```

is equivalent to:

```
#rcodorg timerb_intvec apply
#asm
   jp timerb_isr
#endasm
#rcodorg rootcode resume
```

The following table lists the defined interrupt vector names that may be used with `interrupt_vector`, along with their ISRs.

Table 11-5. Interrupt Vector and ISR Names

Interrupt Vector Name	ISR Name	Default Condition
periodic_intvec	periodic_isr	Fast and nonmodifiable
rst10_intvec	User defined name	User defined
rst18_intvec	These interrupt vectors and their ISRs should never be altered by the user because they are reserved for the debug kernel.	
rst20_intvec		
rst28_intvec		
rst38_intvec	User defined name	User defined
slave_intvec	slave_isr	Fast and nonmodifiable
timera_intvec	User defined name	User defined
timerb_intvec	User defined name	User defined
sera_intvec[a]	DevMateSerialISR	Fast and nonmodifiable
	spa_isr	User defined
serb_intvec	spb_isr	User defined
serc_intvec	spc_isr	
serd_intvec	spd_isr	
sere_intvec	spe_isr	
serf_intvec	spf_isr	
inputcap_intvec	User defined name	
quad_intvec	qd_isr	
ext0_intvec	User defined name	
ext1_intvec	User defined name	

a. Please note that this ISR shares the same interrupt vector as `DevMateSerialISR`. Using `spa_isr` precludes Dynamic C from communicating with the target.

11.8 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 inline 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
```

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.

12. Keywords

A keyword is a reserved word in C that represents a basic C construct. It cannot be used for any other purpose.

abandon

Used in single-user cofunctions, `abandon{}` must be the first statement in the body of the cofunction. The statements inside the curly braces will be executed only if the cofunction is forcibly abandoned and if a call to `loophead()` is made in `main()` before calling the single-user cofunction. See `Samples\Cofunc\Cofaband.c` for an example of abandonment handling.

abort

Jumps out of a costatement.

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

align

Used in assembly blocks, the `align` keyword outputs a padding of nops so that the next instruction to be compiled is placed at the boundary based on `VALUE`.

```
#asm
...
align <VALUE>
...
#endasm
```

`VALUE` can have any (positive) integer expression or the special operands `even` and `odd`. The operand `even` aligns the instruction on an even address, and `odd` on an odd address. Integer expressions align on multiples of the value of the expression.

Some examples:

```
align odd   ; This aligns on the next odd address
align 2     ; Aligns on a 16-bit (2-byte) boundary
align 4     ; Aligns on a 32-bit (4-byte) boundary
align 100h ; Aligns the code to the next address that is evenly divisible by 0x100
align sizeof(int)+4 ; Complex expression, involving sizeof and integer constant
```

Note that integer expressions are treated the same way as operand expressions for other asm operators, so variable labels are resolved to their addresses, not their values.

always_on

The costatement is always active. (Unnamed costatements are always on.)

anymem

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

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

asm

Use in Dynamic C code to insert one assembly language instruction. If more than one assembly instruction is desired use the compiler directive `#asm` instead.

```
int func() {
   int x,y,z;
   asm ld hl,0x3333
   ...
}
```

auto

A functions's local variable is located on the system stack and exists as long as the function call does.

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

bbram

Identifies a variable to be placed into a second data area reserved for battery-backed RAM with boards with more than one RAM device. Generally, the battery-backed RAM is attached to CS1 due to the low-power requirements. In the case of a reset or power failure, the value of a `bbram` variable is preserved, but not atomically like with `protected` variables. No software check is possible to ensure that the RAM is battery-backed. This requirement must be enforced by the user.

If interested, please see the *Rabbit 3000 Microprocessor Designer's Handbook* for information on how the second data area is reserved.

On boards with a single RAM, `bbram` variables will be treated the same as normal root variables. No warning will be given; the `bbram` keyword is simply ignored when compiling to boards with a single RAM.

break

Jumps out of a loop, `if`, or `case` statement.

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

c

Use in assembly block to insert one Dynamic C instruction.

```
#asm
InitValues::
c  start_time = 0;
c  counter = 256;
   ld    hl,0xa0;
   ret
#endasm
```

case

Identifies the next case in a `switch` statement.

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

char

Declares a variable or array element as an unsigned 8-bit character.

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

const

This keyword declares that a value will be stored in flash, thus making it unavailable for modification. `const` is a type qualifier and may be used with any static or global type specifier (`char`, `int`, `struct`, etc.). The `const` qualifier appears before the type unless it is modifying a pointer. When modifying a pointer, the `const` keyword appears after the "`*`."

In each of the following examples, if `const` was missing the compiler would generate a trivial warning. Warnings for `const` can be turned off by changing the compiler options to report serious warnings only. Note that `const` is not currently permitted with return types, automatic locals or parameters and does not change the default storage class for cofunctions.

Example 1:

```
// ptr_to_x is a constant pointer to an integer
int x;
int * const cptr_to_x = &x;
```

Example 2:

```
// cptr_to_i is a constant pointer to a constant integer
const int i = 3;
const int * const cptr_to_i = &i;
```

Example 3:

```
// ax is a constant 2 dimensional integer array
const int ax[2][2] = {{2,3}, {1,2}};
```

Example 4:

```
struct rec {
   int a;
   char b[10];
};
// zed is a constant struct
const struct rec zed = {5, "abc"};
```

Example 5:

```
// cptr is a constant pointer to an integer
typedef int * ptr_to_int;
const ptr_to_int cptr = &i;
// this declaration is equivalent to the previous one
int * const cptr = &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.

debug

Indicates a function is to be compiled in debug mode. This is the default case for Dynamic C functions with the exception of pure assembly language functions.

Library functions compiled in debug mode can be single stepped into, and breakpoints can be set in them.

```
debug int func(){
     ...
}
#asm debug
     ...
#endasm
```

default

Identifies the default case in a switch statement. The default case is optional. It executes only when the switch expression does not match any other case.

```
switch( expression ){
    case const1:
        ...
    case const2:
        ...
    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

The false branch of an `if` statement.

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

enum

Defines a list of named integer constants:

```
enum foo {
   white,              // default is 0 for the first item
   black,              // will be 1
   brown,              // will be 2
   spotted = -2,       // will be -2
   striped,            // will be -3
};
```

An `enum` can be declared in local or global scope. The tag `foo` is optional; but it allows further declarations:

```
enum foo rabbits;
```

To see a colorful sample of the enum keyword, run `/samples/enum.c`.

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

`firsttime` in front of a function body declares the function to have an implicit `*CoData` parameter as the first parameter. This parameter should not be specified in the call or the prototype, but only in the function body parameter list. The compiler generates the code to automatically pass the pointer to the `CoData` structure associated with the costatement from which the call is made. A `firstime` function can only be called from inside of a costatement, cofunction, or slice statement. The `DelayTick` function from `COSTATE.LIB` below is an example of a `firsttime` function.

```
firsttime nodebug int DelayTicks(CoData *pfb, unsigned int ticks)
{
   if(ticks==0) return 1;
   if(pfb->firsttime){
      fb->firsttime=0;
      /* save current ticker */
      fb->content.ul=(unsigned long)TICK_TIMER;
   }
   else if (TICK_TIMER - pfb->content.ul >= ticks)
      return 1;
   return 0;
}
```

float

Declares variables, function return values, or arrays, 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:
```

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` is optional and executes only when none of the `if` or `else if` expressions are true (non-zero).

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).

int

Declares variables, function return values, or array elements to be 16-bit integers. 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 (ISR). All registers, including alternates, are saved when an interrupt function is called and restored when the interrupt function returns. Writing ISRs in C is *never* recommended, especially when timing is critical.

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

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

interrupt_vector

This keyword, intended for use with separate I&D space, sets up an interrupt vector at compile time. This is its syntax:

```
interrupt_vector <INT_VECTOR_NAME> <ISR_NAME>
```

A list of `INT_VECTOR_NAMEs` and ISR_NAMEs is found in Table 11-5 on page 141. The following code fragment illustrates how `interrupt_vector` is used.

```
// Set up an Interrupt Service Routine for Timer B
#asm
   timerb_isr::
   ; ISR code
   ...
   ret
#endasm

main() {
   // Variables
   ...
   // Set up ISR
   interrupt_vector timerb_intvec timerb_isr; // Compile time setup

   // Code
   ...
}
```

`interrupt_vector` overrides run time setup. For run time setup, you would replace the `interrupt_vector` statement above with:

```
#rcodorg <INT_VEC_NAME> apply

#asm
   INTVEC_RELAY_SETUP(timerb_intvec + TIMERB_OFS)

#endasm

#rcodorg rootcode resume
```

This results in a slower interrupt (80 clock cycles are added), but an interrupt vector that can be modified at run time. Interrupt vectors that are set up using `interrupt_vector` are fast, but can't be modified at run time since they are set at compile time.

long

Declares variables, function return values, or array elements to be 32-bit integers. 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. (`main` is actually not a keyword, but is a function name.)

nodebug

Indicates a function is not compiled in debug mode. This is the default for assembly blocks.

```
nodebug int func(){
     ...
}
#asm nodebug
     ...
#endasm
```

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

norst

Indicates that a function does not use the `RST` instruction for breakpoints.

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

nouseix

Indicates a function does not use the IX register as a stack frame reference pointer. This is the default case.

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

NULL

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

protected

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. Battery-backed RAM is required for this operation..

```
main(){
    protected int state1, state2, state3;
        ...
    _sysIsSoftReset();        // restore any protected variables
}
```

The call to `_sysIsSoftReset` checks to see if the previous board reset was due to the compiler restarting the program (i.e. a soft reset). If so, then it initializes the protected variable flags and calls `sysResetChain()`, a function chain that can be used to initialize any protected variables or do other initialization. If the reset was due to a power failure or watchdog time-out, then any protected variables that were being written when the reset occurred are restored.

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 multibyte variable (such as type `int` or `float`). The variable might be only partially written at its next use.Declaring a multibyte variable *shared* means that changes to the variable are atomic, i.e., interrupts are disabled while the variable is being changed. You may declare a multibyte variable as both shared and protected.

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. This keyword is semantically meaningful in function prototypes and produces more efficient code when used. Its use must be consistent between the prototype and the function definition.

```
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`. Using `segchain` is equivalent to using the `#GLOBAL_INIT` directive. 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 multibyte variable (such as a `float`) are atomic. Interrupts are disabled when 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. `sizeof()` can be used inside of assembly blocks.

```
int list[] = { 10, 99, 33, 2, -7, 63, 217 };
   ...
x = sizeof(list);            // x will be assigned 14
```

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 `auto` by default.

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

struct

This keyword introduces a structure declaration, which defines a type.

```
struct {
   ...
   int x;
   int y;
   int z;
} thing1;                   //  defines the variable thing1 to be a struct

struct speed{
   int x;
   int y;
   int z;
};                          //  declares a struct type named speed

struct speed thing2;        //  defines variable thing2 to be of type speed
```

Structure declarations can be nested.

```
struct {
   struct speed slow;
   struct speed slower;
} tortoise;               //  defines the variable tortoise to be a nested struct

struct rabbit {
   struct speed fast;
   struct speed faster;
};                          //  declares a nested struct type named rabbit
struct rabbit chips;        //  defines the variable chips to be of type rabbit
```

switch

Indicates the start of a `switch` statement.

```
switch( expression ){
   case const1:
      ...
      break;
   case const2:
      ...
      break;
   case const3:
      ...
      break
   default :
      ...
}
```

The `switch` statement may contain any number of cases. The constants of the case statements are compared with *`expression`*. If there is a match, the statements for that case execute. The `default` case, if it is present, executes if none of the constants of the case statements match *`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 also execute, regardless of whether their constants match the `switch` expression.

typedef

This keyword provides a way to create new names for existing data types.

```
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, slave1, slave2;
```

union

Identifies a variable that can contain objects of different types and sizes at different times. 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 #nouseix`.

waitfor

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

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

waitfordone (wfd)

The `waitfordone` keyword can be abbreviated as `wfd`. It is part of Dynamic C's cooperative multitasking constructs. Used inside a costatement or a cofunction, it executes cofunctions and `firsttime` functions. When all the cofunctions and `firsttime` functions in the `wfd` statement are complete, or one of them aborts, execution proceeds to the statement following `wfd`. Otherwise a jump is made to the ending brace of the costatement or cofunction where the `wfd` statement appears; when the execution thread comes around again, control is given back to the `wfd` statement.

The `wfd` statements below are from `Samples\cofunc\cofterm.c`

```
x=wfd login();                    // wfd with one cofunction

wfd {                             // wfd with several cofunctions
   clrscr();
   putat(5,5,"name:");
   putat(5,6,"password:");
   echoon();
}
```

As shown, `wfd` may return an argument.

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

Declares a block of data in extended flash memory.

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

The 20-bit physical address of the block is assigned to `name` by the compiler as an unsigned long variable. The amount of memory allocated depends on the data type. Each `char` is allocated one byte, and each `int` is allocated two bytes. If an integer fits into one byte, it is still allocated two bytes. Each `float` and `long` cause four bytes to be allocated.

The value list may include constant expressions of type `int`, `float`, `unsigned int`, `long`, `unsigned long`, `char`, and (quoted) strings. For example:

```
xdata name1 {'\x46','\x47','\x48','\x49','\x4A','\x20','\x20'};
xdata name2 {'R','a','b','b','i','t'};
xdata name3 {" Rules!  "};
xdata name4 {1.0,2.0,(float)3,40e-01,5e00,.6e1};
```

The data can be viewed directly in the dump window by doing a physical memory dump using the 20-bit address of the xdata block. See `Samples\Xmem\xdata.c` for more information.

xmem

Indicates that a function is to be placed in extended memory. This keyword is semantically meaningful in function prototypes. Good programing style dictates its use be consistent between the prototype and the function definition. That is, if a function is defined as:

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

the function prototype should be:

```
xmem int func();
```

Any of the following will put the function in xmem:

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

or

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

or

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

In addition to flagging individual functions, the xmem keyword can be used with the compiler directive #memmap to send all functions not declared as root to extended memory.

```
#memmap xmem
```

This construct is helpful if an application is running out of root code space. Another strategy is to use separate I&D space. Using both `#memmap xmem` and I&D space is not advised and might cause an application to run out of xmem, depending on the size of the application and the size of the flash.

xstring

Declares a table of strings in extended memory. The strings are allocated in flash memory at compile time which means they can not be rewritten directly.

The table entries are 20-bit physical addresses. The `name` of the table represents the 20-bit physical address of the table; this address is assigned to `name` by the compiler.

```
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 {
      ...
      yield;
      ...
   }
   ...
}
```

12.1 Compiler Directives

Compiler 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).

There are some compiler directives used to decide where to place code and data in memory. They are called origin directives and include `#rcodorg`, `#rvarorg` and `#xcodorg`. A detailed description of origin directives may be found in the *Rabbit 3000 Designer's Handbook* (look in the index under "origin directives").

#asm

Syntax: `#asm` *`options`*

Begins a block of assembly code. The available options are:

- `const`: When seperate I&D space is enabled, assembly constants should be placed in their own assembly block (or done in C). For more information, see Section 11.2.2, "Defining Constants."
- `debug`: Enables debug code during assembly.
- `nodebug`: Disables debug code during assembly. This is the default condition. It is still possible to single step through assembly code as long as the assembly window is open.
- `xmem`: Places a block of code into extended memory, overriding any previous memory directives. The block is limited to 4KB.

If the `#asm` block is unmarked, it will be compiled to root.

#class

Syntax: `#class` *`options`*

Controls the storage class for local variables. The available options are:

- `auto`: Place local variables on the stack.
- `static`: Place local variables in permanent, fixed storage.

The default storage class is `auto`.

#debug
#nodebug

Enables or disables `debug` code compilation. `#debug` is the default condition. These directives override the `debug` and `nodebug` keywords used on function declarations or assembly blocks. `#nodebug` prevents RST 28h instructions from being inserted between C statements and assembly instructions.

> **NOTE:** These directives do nothing if they are inside of a function. This is by design. They are meant to be used at the top of an application file.

#define

Syntax: `#define` *`name text`* or `#define` *`name (parameters...) text`*

Defines a macro with or without parameters according to ANSI standard. A macro without parameters may be considered a symbolic constant. Supports the `#` and `##` macro operators. Macros can have up to 32 parameters and can be nested to 126 levels.

#endasm

Ends a block of assembly code.

#fatal

Syntax: `#fatal "..."`

Instructs the compiler to act as if a fatal error. The string in quotes following the directive is the message to be printed

#GLOBAL_INIT

Syntax: `#GLOBAL_INIT` { *`variables`* }

`#GLOBAL_INIT` sections are blocks of code that are run once before `main()` is called. They should appear in functions after variable declarations and before the first executable code. If a local static variable must be initialized once only before the program runs, it should be done in a `#GLOBAL_INIT` section, but other inititialization may also be done. For example:

```
// This function outputs and returns the number of times it has been called.
int foo(){
   char count;
   #GLOBAL_INIT{
      // initialize count
      count = 1;
      // make port A output
      WrPortI(SPCR,SPCRShadow,0x84);
   }
   // output count
   WrPortI(PADR,NULL,count);
   // increment and return count
   return ++count;
}
```

#error

Syntax: `#error` "…"

Instructs the compiler to act as if an error was issued. The string in quotes following the directive is the message to be printed

#funcchain

Syntax: `#funcchain` *`chainname name`*

Adds a function, or another function chain, to a function chain.

#if
#elif
#else
#endif

Syntax: `#if` *`constant_expression`*
`#elif` *`constant_expression`*
`#else`
`#endif`

These directives control conditional compilation. Combined, they 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. Code belonging to the other choices is ignored.

```
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 if all values for *`constant_expression`* are false.

#ifdef

Syntax: `#ifdef` *`name`*

This directive enables code compilation if *`name`* has been defined with a `#define` directive. This directive must have a matching `#endif`.

#ifndef

Syntax: `#ifndef` *`name`*

This directive enables code compilation if *`name`* has not been defined with a `#define` directive. This directive must have a matching `#endif`.

#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

Syntax: `#KILL` *`name`*

To redefine a symbol found in the BIOS of a controller, first `KILL` the prior *`name`*.

#makechain

Syntax: `#makechain` *`chainname`*

Creates a function chain. When a program executes the function chain named in this directive, all of the functions or segments belonging to the function chain execute.

#memmap

Syntax: `#memmap` *`options`*

Controls the default memory area for functions. The following options are available.

- **`anymem NNNN`**: When code comes within NNNN bytes of the end of root code space, start putting it in xmem. Default memory usage is `#memmap anymem 0x2000`.
- **`root`**: All functions not declared as `xmem` go to root memory.
- **`xmem`**: C functions not declared as `root` go to extended memory. Assembly blocks not marked as `xmem` go to root memory. See the description for `xmem` for more information on this keyword.

#pragma

Syntax: `#pragma nowarn [warnt|warns]`

Trivial warnings (`warnt`) or trivial and serious warnings (`warns`) for the next physical line of code are not displayed in the Compiler Messages window. The argument is optional; default behavior is `warnt`.

Syntax: `#pragma nowarn [warnt|warns] start`

Trivial warnings (`warnt`) or trivial and serious warnings (`warns`) are not displayed in the Compiler Messages window until the `#pragma nowarn end` statement is encountered. The argument is optional; default behavior is `warnt`. `#pragma nowarn` cannot be nested.

#precompile

Allows library functions in a comma separated list to be compiled immediately after the BIOS.

The `#precompile` directive is useful for decreasing the download time when developing your program. Precompiled functions will be compiled and downloaded with the BIOS, instead of each time you compile and download your program. The following limitations exist:

- Precompile functions must be defined `nodebug`.
- Any functions to be precompiled must be in a library, and that library must be included either in the BIOS using a `#use`, or recursively included by those libraries.
- Internal BIOS functions will precompile, but will not result in any improvement.
- Libraries that require the user to define parameters before being used can only be precompiled if those parameters are defined before the `#precompile` statement. An example of this is included in `precompile.lib`.
- Function chains and functions using segment chains cannot be precompiled.
- Precompiled functions will be placed in extended memory, unless specifically marked `root`.
- All dependencies must be resolved (Macros, variables, other functions, etc.) before a function can be precompiled. This may require precompiling other functions first.

See `precompile.lib` for more information and examples.

#undef

Syntax: #undef *identifier*

Removes (undefines) a defined macro.

#use

Syntax: #use *pathname*

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. #nouseix is the default.

Note that when the IX register is used as a stack frame reference pointer, it is corrupted when any stack-variable using function is called from within a cofunction, or if a stack-variable using function contains a call to a cofunction.

#warns

Syntax: #warns "..."

Instructs the compiler to act as if a serious warning was issued. The string in quotes following the directive is the message to be printed.

#warntasmine

Syntax: #warnt "..."

Instructs the compiler to act as if a trivial warning was issued. The string in quotes following the directive is the message to be printed.

#ximport

Syntax: `#ximport "filename" symbol`

This compiler directive places the length of *filename* (stored as a `long`) and its binary contents at the next available place in xmem flash. *filename* is assumed to be either relative to the Dynamic C installation directory or a fully qualified path. *symbol* is a compiler generated macro that gives the physical address where the length and contents were stored.

The sample program `ximport.c` illustrates the use of this compiler directive.

#zimport

Syntax: `#zimport "filename" symbol`

This compiler directive extends the functionality of `#ximport` to include file compression by an external utility. *filename* is the input file (and must be relative to the Dynamic C installation directory or be a fully qualified path) and *symbol* represents the 20-bit physical address of the downloaded file.

The external utility supplied with Dynamic C is `zcompress.exe`. It outputs the compressed file to the same directory as the input file, appending the extension `.DCZ`. E.g., if the input file is named `test.txt`, the output file will be named `test.txt.dcz`. The first 32 bits of the output file contains the length (in bytes) of the file, followed by its binary contents. The most significant bit of the length is set to one to indicate that the file is compressed.

The sample program `zimport.c` illustrates the use of this compiler directive. Please see Appendix C.2 for further information regarding file compression and decompression.

13. 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 = 0x17;        // low byte of x 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.

13.1 Arithmetic Operators

+

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, `*` indicates that the following item is a pointer. When used as an indirection operator in an expression, `*` provides the value at the address specified by a pointer.

```
int *p;                  // p is a pointer to an integer
const 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 pointers to integers
int (*list)[10]          // pointer to array of 10 integers
float** y;               // pointer to a pointer 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
```

/

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

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

++

Pre- or post-increment 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. 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.

%

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

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

13.2 Assignment Operators

=

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
```

13.3 Bitwise Operators

<<

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:

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

The least significant bits of the operand are lost; the vacated bits become zero for unsigned variables and are sign-extended for signed variables.

&

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
```

^

Bitwise exclusive OR. A binary operator, this performs the bitwise XOR of two integer (8-bit, 16-bit or 32-bit) values.

```
int i = 0xFFF0;
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 0xFFF0
```

~

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
```

13.4 Relational Operators

<

Less than. This binary (relational) operator yields a Boolean value. The result is 1 if the left operand is less than 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 is less than or equal to 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 is greater than 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 is greater than or equal to the right operand, and 0 otherwise.

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

13.5 Equality Operators

==

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 is not equal to the right operand, and 0 otherwise.

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

13.6 Logical Operators

&&

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 non-zero, the result is 1 (TRUE). Otherwise, the result is 0 (FALSE).

!

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 non-zero. The NOT operator result is 1 if the operand is 0. The result is 0 otherwise.

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

13.7 Postfix Expressions

( )

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. All array subscripts count from 0.

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

. (dot)

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;                  // p is a pointer to structure
   ...
m = p->x;                  // reference to structure element
```

13.8 Reference/Dereference Operators

&

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
```

`*`

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

```
int *p;              // p is a pointer to an 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
```

13.9 Conditional Operators

Conditional operators are a three-part operation unique to the C language. The operation has three operands and the two operator symbols `?` and `:`.

`? :`

If the first operand evaluates true (non-zero), 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.

13.10 Other Operators

(type)

The `cast` operator converts one data type to another. A floating-point value is 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
```

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" };

#define array_size sizeof(record)*100  // number of bytes in array
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.

,

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.

The table below shows the operator precedence, from highest to lowest. All operators grouped together have equal precedence.

Table 13-1. Operator Precedence

<table>
<tr><th>Operators</th><th>Associativity</th><th>Function</th></tr>
<tr><td>() [] -> .</td><td>left to right</td><td>member</td></tr>
<tr><td>! ~ ++ --
(type) * & sizeof</td><td>right to left</td><td>unary</td></tr>
<tr><td>* / %</td><td>left to right</td><td>multiplicative</td></tr>
<tr><td>+ -</td><td>left to right</td><td>additive</td></tr>
<tr><td><< >></td><td>left to right</td><td>bitwise</td></tr>
<tr><td>< <= > >=</td><td>left to right</td><td>relational</td></tr>
<tr><td>== !=</td><td>left to right</td><td>equality</td></tr>
<tr><td>&</td><td>left to right</td><td>bitwise</td></tr>
<tr><td>^</td><td>left to right</td><td>bitwise</td></tr>
<tr><td>|</td><td>left to right</td><td>bitwise</td></tr>
<tr><td>&&</td><td>left to right</td><td>logical</td></tr>
<tr><td>||</td><td>left to right</td><td>logical</td></tr>
<tr><td>? :</td><td>right to left</td><td>conditional</td></tr>
<tr><td>= *= /= %= += -=
<<= >>= &= ^= |=</td><td>right to left</td><td>assignment</td></tr>
<tr><td>, (comma)</td><td>left to right</td><td>series</td></tr>
</table>

14. Graphical User Interface

Dynamic C can be used to edit source files, compile and run programs, and choose options for these activities using pull-down menus or keyboard shortcuts. There are two modes: *edit mode* and *run mode* (run mode is also known as *debug mode*). Various debugging windows can be viewed in run mode. Programs can compile directly to a target controller for debugging in RAM or flash. Programs can also be compiled to a `.bin` file, with or without a controller connected to the PC.

To debug a program, a controller must be connected to the PC, either directly via a programming cable or indirectly via an Ethernet connection and a RabbitLink board. Multiple instances of Dynamic C can run simultaneously. This means multiple debugging sessions are possible over different serial ports. This is useful for debugging boards that are communicating among themselves.

14.1 Editing

A file is displayed in a text window when it is opened or created. More than one text window may be open. If the same file is in multiple windows, any changes made to the file in one window will be reflected in all text windows that display that file. Dynamic C supports normal Windows text editing operations.

A mouse (or other pointing device) may be used to position the text cursor, select text, or extend a text selection. The keyboard may be used to do these same things. Text may be scrolled using the arrow keys, the PageUp and PageDown keys, and the Home and End keys. The up, down, left and right arrow keys move the cursor in the corresponding direction.

The Ctrl key works in conjunction with the arrow keys this way

Ctrl+Left	Move cursor to previous word.
Ctrl+Right	Move cursor to next word.
Ctrl+Up	Move editor window up, text moves down one line. Cursor is not moved.
Ctrl+Down	Move editor window down, text moves up one line. Cursor is not moved.

The Home key may be used alone or with other keys.

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.

The End key may be used alone or with other keys.

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.

14.2 Menus

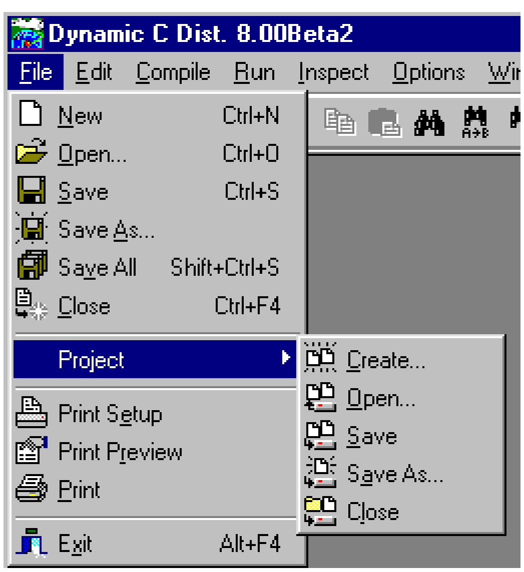
Dynamic C's main menu has 8 command menus, as well as the standard Windows system menus.

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 the Library Function Lookup option). 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. For example, press <Alt+F> to activate the FILE menu.

14.2.1 File Menu

Click the menu title or press <Alt+F> to select the FILE menu.



New <Ctrl+N>
Creates a blank, untitled program in a new window, called the text window or the editor window. If you right click anywhere in the text window a popup menu will appear. It is available as a convenience for accessing some frequently used commands.

Open <Ctrl+O>
Presents a dialog box to specify the name of a file to open. To select a file, type in the file name (pathnames may be entered), or browse and select it. 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. Multiple files can be selected by either holding down <Ctrl> then clicking the left mouse on each filename you want to open, or by dragging the selection rectangle over multiple filenames.

Save <Ctrl+S>

The Save command updates an open file to reflect changes made since the last time the file was saved. If the file has not been saved before (i.e., the file is a new untitled file), the Save As dialog will appear to prompt for a name. Use the Save command often while editing to protect against loss during power failures or system crashes.

Save As

Presents a dialog box to save the file under a new name. To select a file name, type it in the File name field. The file will be saved in the folder displayed in the Save in field. You may, of course, browse to another location. You may also select an existing file. Dynamic C will ask you if you wish to replace the existing file with the new one.

Save All <Shift+Ctrl+S>

This command saves all modified files that are currently open.

Close <Ctrl+F4>

Closes the active editor window. The active window may also be closed by double-clicking on its system menu. If there is an attempt to close a modified file, Dynamic C will ask you if you wish to save the changes. 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. Choosing Cancel results in a return to Dynamic C with no action taken.

Project

Allows a project file to be created, opened, saved, saved as a different name and closed. See Chapter 16 for all the details on project files.

Print Setup

Displays the Page Setup dialog box. Margins, page orientation, page numbers and header and footer properties are all chosen here.

The Printer Setup button is in the bottom left of the dialog box. It brings up the Print Setup dialog box, which allows a printer to be selected. The Network button allows printers to be added or removed from the list of printers.

Print Preview

Displays whichever file is in the active editor window in the Preview Form window, showing how the text will look when it is printed. You can search and navigate through the printable pages and bring up the Print dialog box.

Print

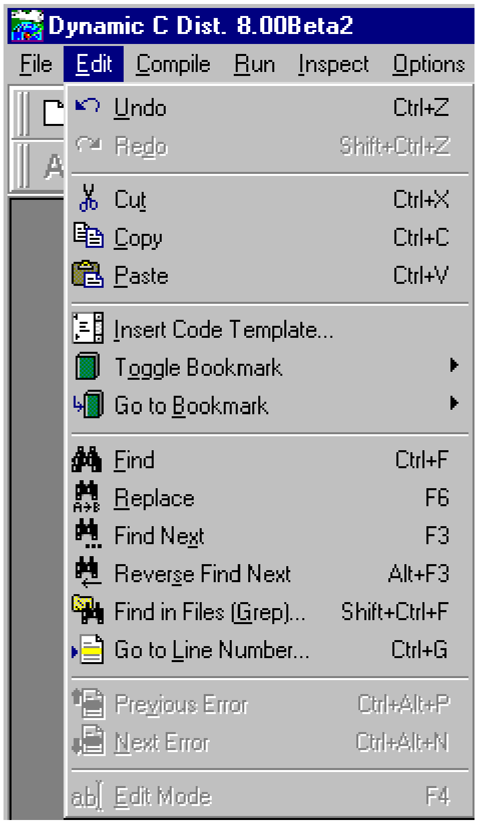
Brings up the Print dialog box, which allows you to choose a printer. Only text in an editor window may be printed. To print the contents of debug windows, the text must be copied and pasted to an editor window. As many copies of the text as needed may be printed. If more than one copy is requested, the pages may be collated or uncollated.

Exit <Alt+F4>

Close Dynamic C after prompting to save any unsaved changes to open files.

14.2.2 Edit Menu

Click the menu title or press <Alt+E> to select the EDIT menu.



Undo <Ctrl+Z>

This option undoes recent changes in the active edit window. The command may be repeated several times to undo multiple changes. Undo operations have unlimited depth. Two types of undo are supported—applied to a single operation and applied to a group of the same operations (2 continuous deletes are considered a single operation.

Dynamic C only discards undo information if the “Undo after save” option is unchecked in the Editor dialog under Environment Options.

Redo <Shift+Ctrl+Z>

Redoes changes recently undone. This command only works immediately after one or more Undo operations.

Cut <Ctrl+X>

Removes selected text and saves to the clipboard.

Copy <Ctrl+C>

Makes a copy of text selected in a file or in a debug window. The text is saved on the clipboard.

Paste <Ctrl+V>

Pastes text from the clipboard to the current insertion point. Nothing can be pasted in a debugging window. The contents of the clipboard may be pasted virtually anywhere, repeatedly (as long as nothing new is cut or copied into the clipboard), in the same or other source files, or even in word processing or graphics program documents.

Insert Code Template <Ctrl+J>

Opens the code template list at the current cursor location. Clicking on a list entry or pressing <Enter> inserts the selected template at the cursor location in the active edit window. The arrow keys may be used to scroll the list. Pressing the first letter of the name of a code template selects the first template whose name starts with that letter. Pressing the same letter again will go to the next template whose name starts with that letter. Continuing to press the same letter cycles through all the templates whose name starts with that letter.

To create, edit or remove templates from the code template list, go to Environment Options and click on the Code Templates tab.

Toggle Bookmark

Toggle one of 10 bookmarks in the active edit window.

Go to Bookmark

Go to one of 10 bookmarks in the active edit window. Executing this command again will take you back to the location you were at before going to the bookmarked location.

Find <Ctrl F>

Finds first occurrence of specified text. Text may be specified by selecting it prior to opening the Find dialog box if the option "Find text at cursor" is checked in the Editor dialog under Environment Options. Only one word may be selected; if more than one word is selected, the last word selected appears as the entry for the search text. More than one word of text may be specified by typing it in or selecting it from the available history of search text.

There are several ways to narrow or broaden the search criteria using the Find dialog box. For example, if Case sensitive is unchecked, then "Switch" and "SWITCH" would match the search text "switch." If Whole words only is checked, then the search text "switch" would not match "switches." Selecting Entire scope will cause the whole document to be searched. If Selected text is chosen and "Persistent blocks" was checked in the Editor dialog under Environment Options, the search will take place only in the selected text.

Replace <F6>

Finds and replaces the specified text. Text may be specified by selecting it prior to opening the Replace Text dialog box. Only one word may be selected; if more than one word is selected, the last word selected appears as the entry for the search text. More than one word of text may be specified by typing it in or selecting it from the available history of search text. The replacement text is typed or selected from the available history of replacement text.

As with the Find dialog box, there are several ways to narrow or broaden the search criteria. An important option is Prompt on replace. If this is unchecked, Dynamic C will not prompt before making the replacement, which could be dangerous in combination with the choice to Replace All.

Find Next <F3>

Once search text has been specified with the Find or Replace commands, the Find Next command 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 replace.

Reverse Find Next <Alt+F3>

Behaves the same as Find Next except in the opposite direction. If Find Next is searching forward in the file, Reverse Find Next will search backwards, and vice versa.

Find in Files (Grep)... <Shift+Ctrl+F>

This option searches for text in the currently open file(s) or in any directory (optionally including subdirectories) specified. Standard Unix-style regular expressions are used.

A window with the search results is displayed with an entry for each match found. Double-clicking on an entry will open the corresponding file and place the cursor on the search string in that file. Multiple filetypes can be separated by semicolons. For example, entering `C:\mydirectory\*.lib;*.c` will search all `.lib` and `.c` files in `mydirectory`.

Go to Line Number

Positions the insertion point at the beginning of the specified line.

Previous Error <Ctrl+Alt+P>

Locates the previous compilation error in the source code. Any error messages will be displayed in a list in the Compiler Messages window after a program is compiled. Dynamic C selects the previous error in the list and displays the offending line of code in the text window.

Next Error <Ctrl+Alt+N>

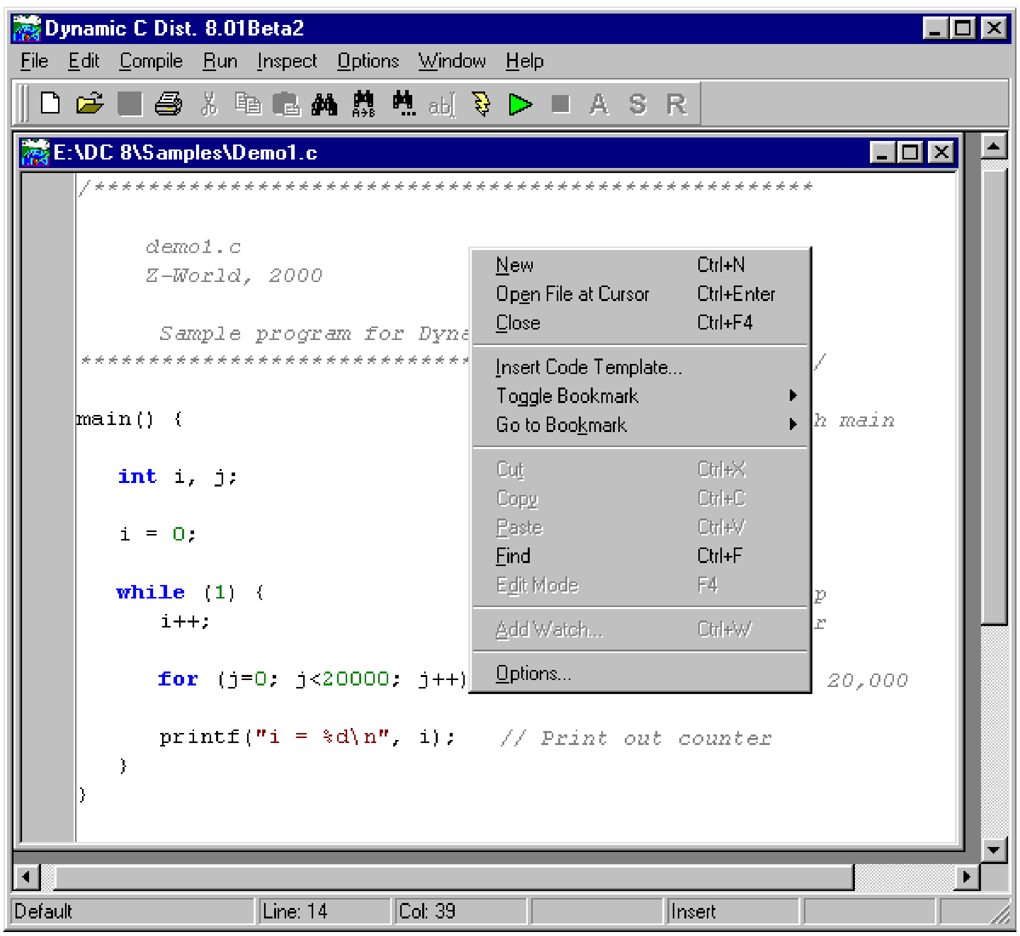
Locates the next compilation error in the source code. Any error messages will be displayed in a list in the Compiler Messages window after a program is compiled. Dynamic C selects the next error in the list and displays the offending line of code in the text window.

Edit Mode <F4>

Switches to edit mode from run mode. (Run mode aka debug mode.) After successful compilation or execution, no changes to the file are allowed unless Edit Mode is selected. If the compilation fails or a runtime error occurs, Dynamic C comes back already in edit mode.

14.2.2.1 Editor Window Popup Menu

Right click anywhere in the editor window and a popup menu will appear.



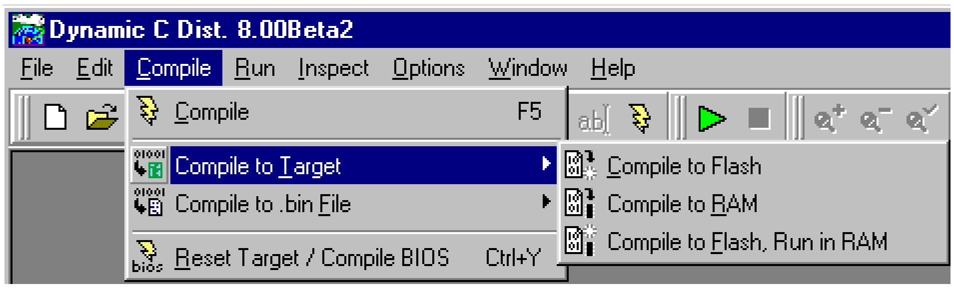
All of the menu options, with the exception of Open File at Cursor, are available from the main menu, e.g., New is an option in the File menu and was described earlier with the other options for that menu.

Open File at Cursor <Ctrl+Enter>

Attempts to open the file whose name is under the cursor. The file will be opened in a new editor window, if the file name is listed in `lib.dir` as either an absolute path or a path relative to the Dynamic C root directory or if the file is in Dynamic C's root directory. As a last resort, an Open dialog box will appear so that the file may be manually chosen.

14.2.3 Compile Menu

Click the menu title or press <Alt+C> to select the COMPILE menu.



Compile <F5>

Compiles a program and loads it to the target or to a .bin file. When you press <F5> or select Compile from the Compile menu, the active file will be compiled according to the current compiler options. Compiler options are set in the dialog box accessed by the Compiler tab in the menu selection Options | Project Options. When compiling directly to the target, Dynamic C queries the attached target for board information and creates macros to automatically configure the BIOS and libraries.

Any compilation errors are listed in the automatically activated Compiler Messages window. Press <F1> to obtain more information for any error message that is high-lighted in this window.

Compile to Target

Expands to one of 3 choices:

- Compile to Flash
- Compile to RAM
- Compile to Flash, Run in RAM

These options override any BIOS Memory Setting choice made from the Compiler tab in the menu selection Options | Project Options.

Compile to .bin File

Compiles a program and writes the image to a `.bin` file. There are 2 choices available with this option, Compile to Flash and Compile to Flash, Run in Ram:

The target configuration used in the compile is determined in Options | Project Options, Compiler tab. From there, under Default Compile Mode you can choose to use the attached target or a defined target configuration. The defined target configuration is accessed by clicking on the Targetless tab which will reveal three additional tabs: RTI File, Specify Parameters and Board Selection. To learn more about these tabs see page 231.

The `.bin` file may be used with a device programmer to program multiple targets; or the Rabbit Field Utility (RFU) can be used to load the `.bin` file to the target.

If you are creating special a program such as a cold loader that starts at address 0x0000 you can exclude the BIOS from being compiled into the `.bin` file by unchecking the option to include it. This is done by choosing Options | Project Options | Compiler and clicking on the Advanced... button.

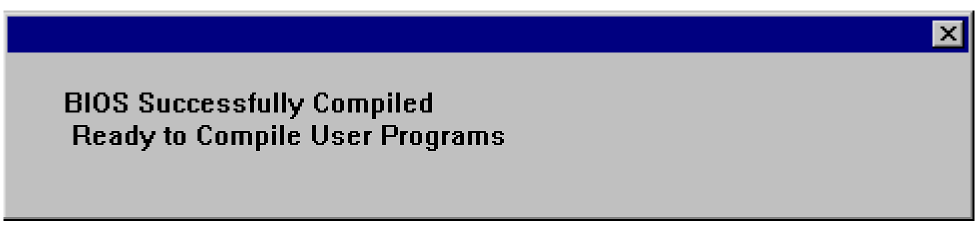
In addition to the .bin file, several other files are generated with this compile option. For example, if you compile `demo1.c` to a `.bin` file, the following files will be in the same folder as `demo1.c`:

- `DEMO1.bak` - backup of the application source file (made at compile time, when this option is enabled).
- `demo1.bdl` - binary image download file (used when loading the application to a connected target).
- `DEMO1.brk` - debugger breakpoints information.
- `demo1.hdl` - no longer used.
- `demo1.hex` - simple Intel HEX format output image file; the serial DLM samples download a DLP's HEX file and load the image to flash.
- `DEMO1.map` - the application's code/data map file (`RabbitBios.map` is also generated, separately). For more information on the map file, see Appendix B Appendix Title
- `DEMO1.rom` - ROM "output" file, containing redundant addresses (due to fixups); it's used to generate the BDL, BIN, HEX, and HDL files.

Reset Target / Compile BIOS <Ctrl+Y>

This option reloads the BIOS to RAM or flash, depending on the choice made under BIOS Memory Setting in the Compiler dialog (viewable from Options | Project Options).

The following message will appear upon successful compilation and loading of BIOS code.



14.2.4 Run Menu

Click the menu title or press <Alt+R> to select the RUN menu.

Run <F9>

Starts program execution from the current breakpoint. Registers are restored, including interrupt status, before execution begins. If in Edit mode, the program is compiled and downloaded.

Stop <Ctrl+Q>

The Stop command stops the program at the current point of execution. Usually, the debugger cannot stop within nodebug code. On the other hand, the target can be stopped at an `RST 028h` instruction if an `RST 028h` assembly code is inserted as inline assembly code in `nodebug` code. However, the debugger will never be able to find and place the execution cursor in `nodebug` code.

Run w/ No Polling <Alt+F9>

This command is identical to the Run command, with one exception. The PC polls the target every 3 seconds by default to determine if the target has crashed. When debugging via RabbitLink, polling is used to make the RabbitLink keep its connection to the PC open. Polling does have some overhead, but it is very minimal. If debugging ISRs, it may be helpful to disable polling.

Trace Into <F7>

Executes one C statement (or one assembly language instruction if the assembly window is displayed) with descent into functions. If `nodebug` is in effect and the Assembly window is closed, execution continues until code compiled without the `nodebug` keyword is encountered.

Step Over <F8>

Executes one C statement (or one assembly language instruction if the assembly window is displayed) without descending into functions.

Source Trace Into <Alt+F7>

Executes one C statement with descent into functions when the assembly window is open. If `nodebug` is in effect, execution continues until code compiled without the `nodebug` keyword is encountered.

Source Step Over <Alt+F8>

Executes one C statement without descending into functions when the assembly window is open.

Toggle Breakpoint <F2>

Toggles a regular (“soft”) breakpoint at the current cursor location. Soft breakpoints do not affect the interrupt state at the time the breakpoint is encountered, whereas hard breakpoints do.

Toggle Hard Breakpoint <Alt+F2>

Toggles a hard breakpoint at the current cursor location. A hard breakpoint differs from a soft breakpoint in that interrupts are disabled when the hard breakpoint is reached.

Clear All Breakpoints <Ctrl+A>

Self explanatory.

Poll Target <Ctrl+L>

This menu option used to be named Toggle Polling. A check mark indicates that Dynamic C will poll the target. The absence of a check mark indicates that Dynamic C will not poll the target. This differs from Toggle Polling in that Dynamic C will not restart polling without the user explicitly requesting it.

Reset Program <Ctrl+F2>

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 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. That is, if your code is corrupted, the reset will not be enough; you will have to reload the program to the target.

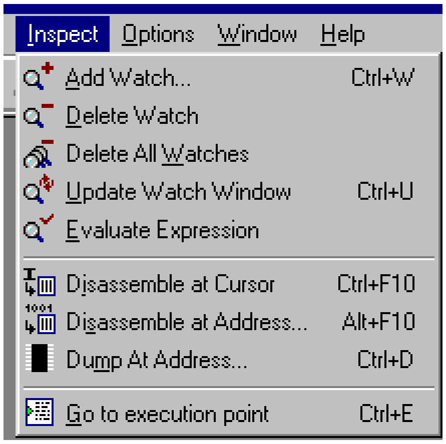
Close Connection

If using a serial connection, disconnects the programming serial port between PC and target so that the target serial port is accessible to other applications.

If using a TCP/IP connection, closes the socket between the PC and the RabbitLink.

14.2.5 Inspect Menu

Click the menu title or press <Alt+I> to open the INSPECT menu.

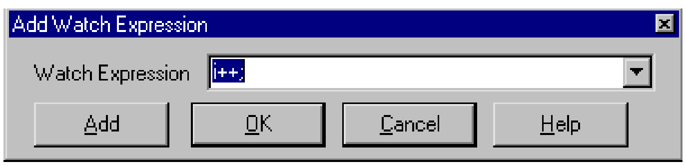


The INSPECT menu provides commands to manipulate watch expressions, view disassembled code, and produce hexadecimal memory dumps. The INSPECT menu commands and their functions are described here.

Add Watch <Ctrl+W>

This command displays the Add Watch Expression dialog. Enter watch expressions with this dialog box.

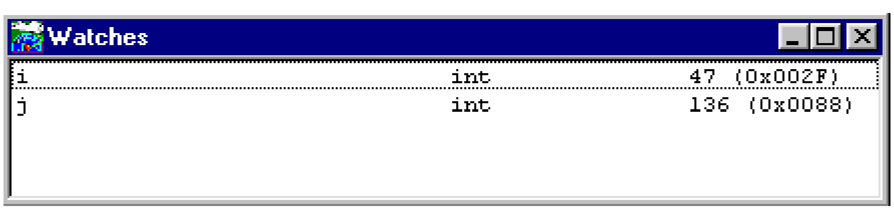
A watch expression may be any valid C expression, including assignments, function calls, and preprocessor macros. (Do not include a semicolon at the end of the expression.) If the watch expression is successfully compiled, it and its outcome will appear in the Watches window.



If the cursor in the active window is positioned over a variable or function name, that name will appear in the Watch Expression text box when the Add Watch Expression dialog box appears. Clicking the Add button will add the given watch expression to the watch list, and will leave the Add Watch Expression dialog open so that more watches can be added. Clicking the OK button will add the given watch expression to the watch list, and close the Add Watch Expression dialog.

To add a local variable to the Watch window, the target controller's program counter (PC) must point to the function where the local variable is defined. If the PC points outside the function, an error message will display when Add or OK is pressed, stating that the variable is out of scope or not declared.

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



If the evaluation of a watch expression causes a run-time exception, the exception will be ignored and the value displayed in the Watches window for the watch expression will be undefined.

Delete Watch

Removes highlighted entry from the Watches window.

Delete All Watches

Removes all entries from the Watches window.

Update Watch Window <Ctrl+U>

Forces expressions in the Watches window to be evaluated. If the target is running nodebug code, the Watches window will not be updated, and the PC will lose communication with the target. Inserting an `RST 028h` instruction into frequently executed nodebug code will allow the watch window to be updated while running in nodebug code. Normally the Watch window is updated every time the execution cursor is changed, that is when a single step, a breakpoint, or a stop occurs in the program.

Evaluate Expression

Brings up the Evaluate Expression dialog where you can enter a single expression in the Expression dialog. The result is displayed in the Result text box when Evaluate is clicked. Multiple Evaluate Expression dialogs can be active at the same time.

Disassemble at Cursor <Ctrl+F10>

Loads, disassembles and displays the code at the current editor cursor location. This command does not work in user application code declared as `nodebug`. Also, this command does not stop the execution on the target.

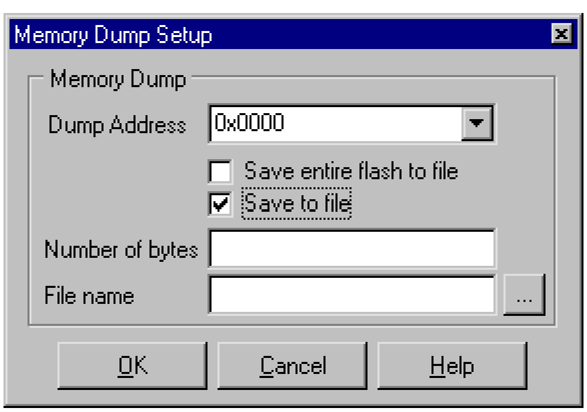
Disassemble at Address <Alt+F10>

Brings up the Disassemble at Address dialog where you can enter an address at which to begin disassembly. The format of the address is either the logical address specified as a hex number (0xnnnn or just nnnn) or as an xpc:offset pair separated by a colon (nn:mmmm).

The Disassembled Code window displays the result. See “Assembly” on page 235 for details about this window.

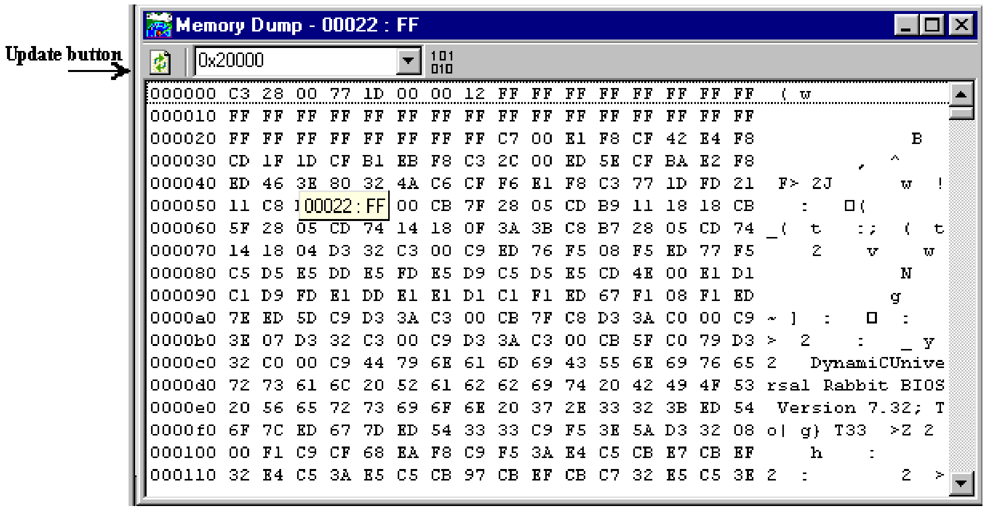
Dump at Address <Ctrl+D>

Allows blocks of raw values in any memory location to be looked at. Values can be displayed on the screen or written to a file. If separate I&D space is enabled, you can choose which logical space to examine: instruction space or data space.



When writing to a file, the option Save to File requires a file pathname and the number of bytes to dump. The option Save Entire Flash to File requires a file pathname. If you are running in RAM, then it will be RAM that is saved to a file, not Flash, because this option simply starts dumping physical memory at address zero.

When displaying on a screen, the Memory Dump window is opened. A typical screen display appears below. Although the cursor is not visible in this screen capture, it is hovering over logical memory location 0x0022, which has a value of 0xFF. This information is given in the fly-over text and also in the titlebar. Either or both of these options may be disabled by right clicking in the Memory Dump window or in the Options | Environment Options, Debug Windows tab, under Specific Preferences for the Memory Dump window.



The Memory Dump window may be scrolled. Scrolling causes the contents of other memory addresses to appear in the window. Hotkeys ArrowUp, ArrowDown, PageUp, PageDown are active in the Memory Dump window. The window always displays as many lines of 16 bytes and their ASCII equivalent as will fit in the window.

Values in the Dump window are updated automatically either when Dynamic C stops or comes to a breakpoint. Updates only occur if the window is updateable. This can be set either by right clicking in the Memory Dump window and toggling the updateable menu item, or by clicking on the Debug Windows tab in Options | Environment Options. Select Memory Dump under Specific Preferences, then check the option “Allow automatic updates.” The Memory Dump window can be updated at any time by clicking the Update button on the tool bar or by right clicking and choosing Update from the popup menu.

The Memory Dump window is capable of displaying three different types of dumps. A dump of a logical address ([0x]mmmm) will result in a 64k scrollable region (0x0000 - 0xffff). A dump of a physical address ([0x]mmmmm) will result in a dump of a 1M region (0x00000 - 0xfffff). A dump of an xpc:offset address (nn:mmmm) will result in either a 4k, 64k, or 1M dump range depending on the option set on the Debug Windows tab under Options | Environment Options.

Any number of dump windows may be open at the same time. The type of dump or dump region for a dump window can be changed by entering a new address in the toolbar’s text entry area. To the right of the this area is a button that, when clicked, will cause the address in the text entry area to be the first address in the Dump window. The toolbar for a dump window may be hidden or visible.

Goto execution point <Ctrl+E>

When stopped in debug mode, this option places the cursor at the statement or instruction that will execute next.

14.2.6 Options Menu

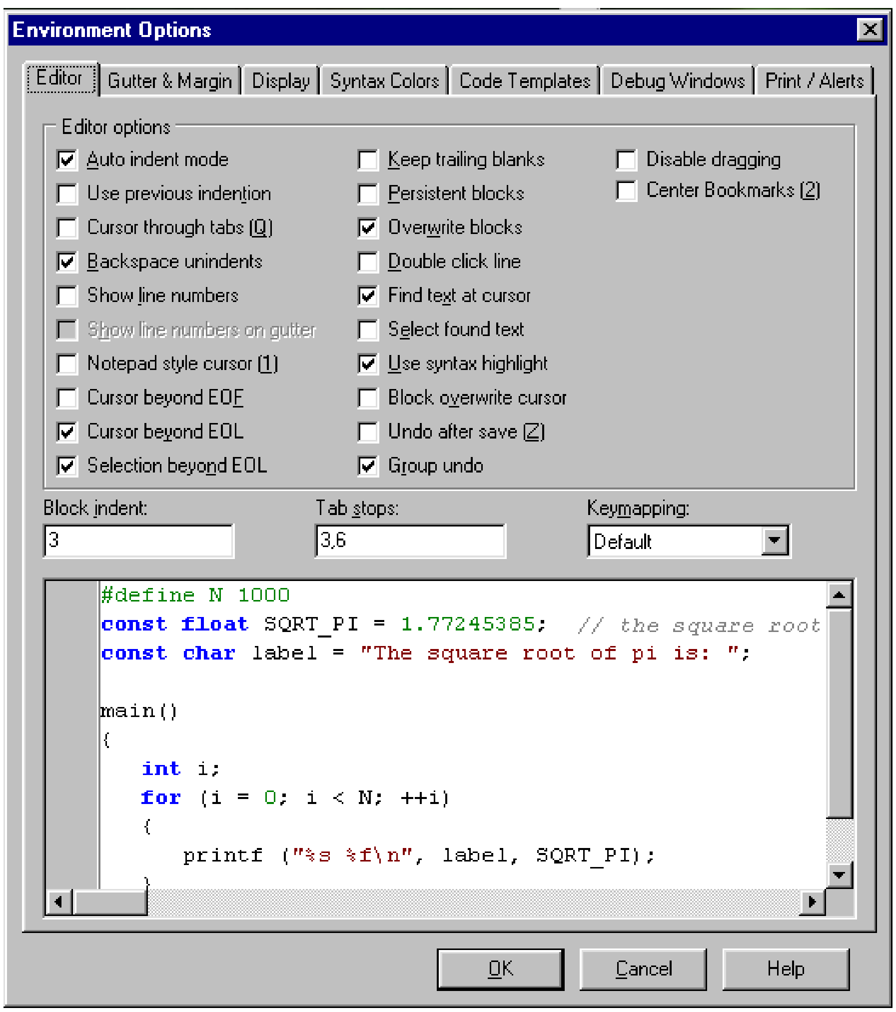
Click the Options menu title or press <Alt+O> to select the Options menu.

Environment Options

Dynamic C comes with a built-in, full-featured text editor. It may be customized to suit your style using the Environment Options dialog box. The dialog box has tabs for various aspects of the editor.

Editor Tab

Click on the Editor tab to display the following dialog. Installation defaults are shown.



The Editor options are detailed here. All actions taken are immediately reflected in the text area at the bottom of the dialog, and in any open editor windows.

Auto indent mode

Checking this causes a new line to match the indentation of the previous line.

Use previous indention

Uses the same characters for indentation that were used for the last indentation. If the last indentations was 2 tabs and 4 spaces, the next indentation will use the same combination of whitespace characters.

Cursor through tabs

With this option checked, the right and left arrow keys will move the cursor through the logical spaces of a tab character. If this is unchecked the cursor will move the entire length of the tab character.

Backspace unindents

Check this to backspace through indentation levels. If this is unchecked, the backspace will move one character at a time.

Show line numbers

Check this to display line numbers in the text window. This must be checked to activate the option Show line numbers on gutter.

Show line numbers on gutter

If gutters are visible, check this to display line numbers in the gutter.

Notepad style cursor

Checking this causes the cursor to behave similar to Notepad.

Cursor beyond EOF

Check this option to move the cursor past the end of the file.

Cursor beyond EOL

Check this option to move the cursor past the end of the line.

Selection beyond EOL

Check this option to select text beyond the end of the line.

Keep trailing blanks

Check this option to keep extra spaces and tabs at the end of a line when a new line is started.

Persistent blocks

Check this option to keep selected text selected when you move the cursor using the arrow keys. Using the mouse to move the cursor will deselect the block of text. Using menu commands or keyboard shortcuts will affect the entire block of selected text. For example, pressing <Ctrl+X> will cut the selected block. But pressing the delete key will only delete one character to the right of the cursor. If this option was unchecked, pressing the delete key would delete all the selected text.

If this option is checked and the Find or Replace dialog is opened with a piece of text selected in the active edit window, the search scope will default to that bit of selected text only.

Overwrite blocks

Check this option to enable overwriting a selected block of text by pressing a key on the keyboard. The block of text may be overwritten with any character, including whitespaces or by pressing delete or backspace.

Double click line

Check this option to allow an entire line to be selected when you double click at any position in the line. When this option is unchecked, double clicking will select the closest word to the left of the cursor.

Find text at cursor

When either the Search or Replace dialogs are opened, if this option is checked the word at the cursor location in the active editor window will be placed into the “Text to Find” edit box. If this option is unchecked, the edit box will contain the last search string.

Select found text

The color of found text can be set in Options | Environment Options, on the Syntax Colors page. Select “Search Match” from the Element list box, then set the foreground and background colors.

If this box is unchecked the Search Match color scheme will be used when a match is found, but the text will not be selected for copy or delete operations. If this option is checked, the matched text will automatically be selected so that it may be copied or deleted.

Use syntax highlight

Check this option to enable the Display and Syntax Color choices to be active.

Block overwrite cursor

Check this option to show the cursor as a block when an editor is placed in overwrite mode.

Undo after save

Check this option to enable undo operations after a file has been saved. With this option unchecked, the undo list for a file is erased each time the file is saved.

Group undo

Check this option to undo changes one group at a time. With this option unchecked, each operation is undone individually.

Disable dragging

Checking this option disables drag and drop operations: i.e., the ability to move selected text by pressing down the left mouse button and dragging the text to a new location.

Center Bookmarks

Check this option so that when you jump to a bookmark it is centered in the editor window.

Block indent

The number of spaces used when a selected block is indented using <Ctrl+k+i> or unindented using <Ctrl+k+u>.

Tab stops

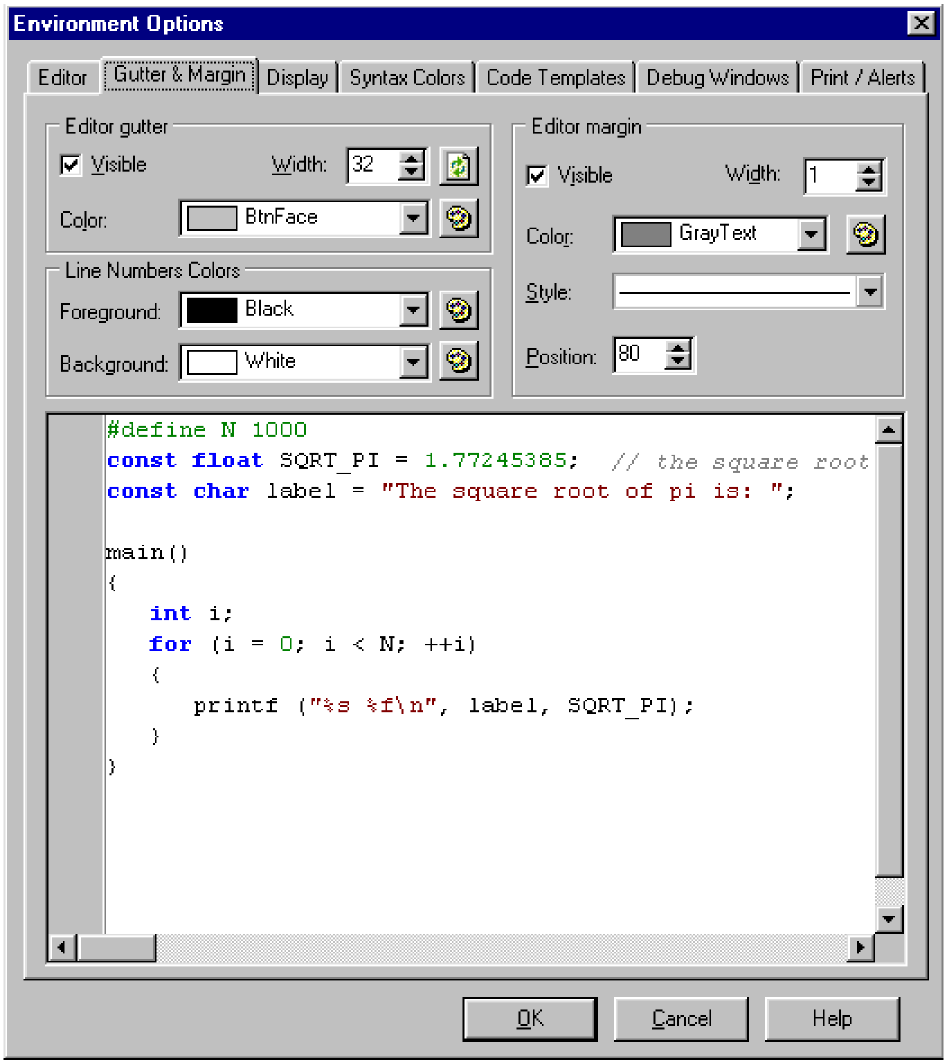
This is a comma separated list of numbers which indicate the number of spaces per tab stop. If only one number is entered, say "3," then the first tab stop is 3 spaces, as is each additional tab stop. Every additional number in the list indicates the number of spaces for all subsequent tabs. E.g., if the list consists of "3,6,12" the first tab stop is 3 spaces, the second tab stop is 3 more spaces and all subsequent tab stops are 6 spaces.

Keymapping

The keyboard has 5 different default key mappings: Default, Classic, Brief, Epsilon and Visual Studio. Change the keymapping with this pulldown menu.

Gutter & Margin Tab

Click on the Gutter & Margin tab to display the following dialog.



Editor gutter

Check the Visible box to create a gutter in the far left side of the text window. Use the Width scroll bar to set the width of the gutter in pixels. The button to the right updates the width parameter. Changing the width and clicking on OK at the bottom of the dialog does not update the gutter width; you must click on the button. Use the Color pulldown menu to set the color. The button to the right brings up more color choices.

Editor margin

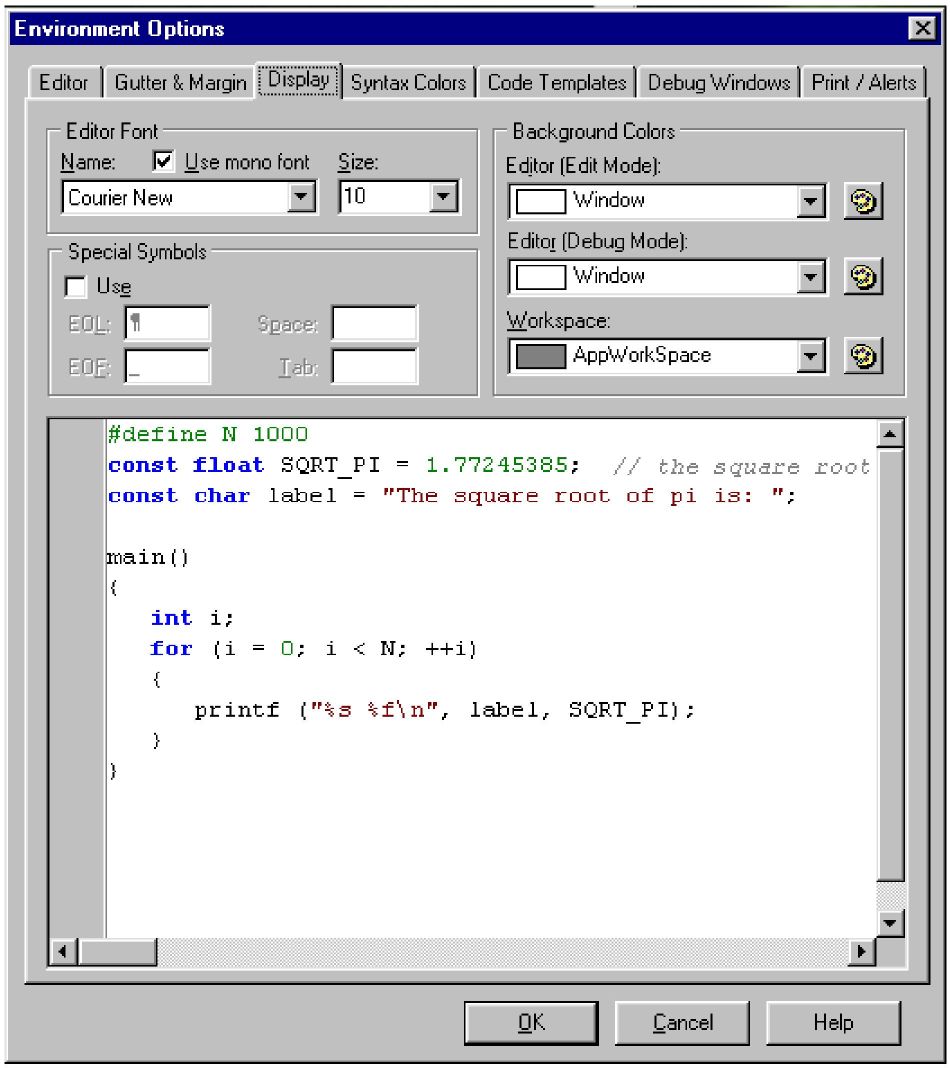
Check the Visible box to create a right-hand margin in the text window. Use the Width scroll bar and the Color pulldown menu to set the like-named attributes of the margin line. The Style pulldown menu displays the line choices available: a solid line and various dashed lines. The Position scroll box is used to place the margin at the desire location in the text window.

Line Number Colors

If line numbers are set to visible and are not placed on the gutter, the Foreground color will set the color of the line numbers and the Background color will set the color on which the line numbers appear.

Display Tab

Click on the Display tab to display the following dialog.



Editor Font

This area of the dialog box is for choosing the font style and size. Check Use mono font for fixed spacing with each character; note that this option limits the available font styles.

Special Symbols

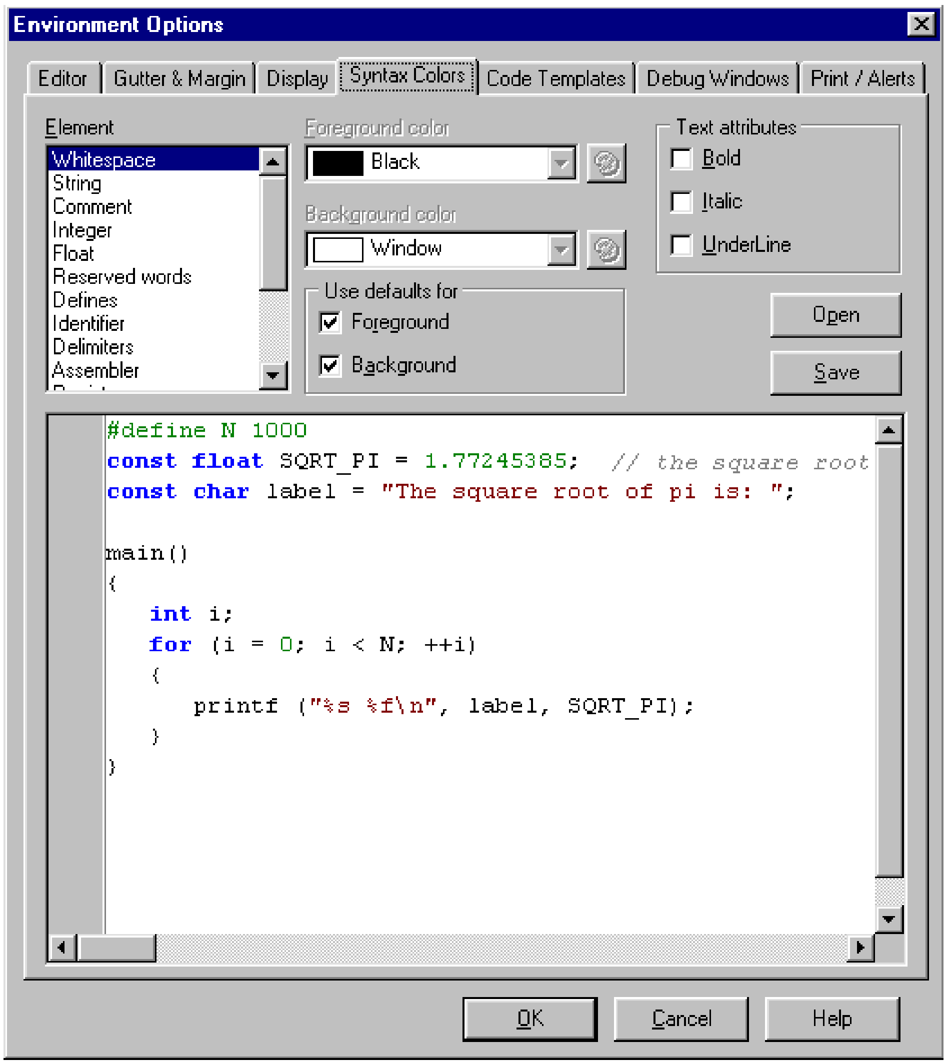
Check Use to view end of line, end of file, space and/or tab symbols in the editor window.

Background Colors

This area of the dialog box is for choosing background colors for editor windows and the main Dynamic C workspace. The editor window can have a different background color in edit mode than it does in run mode. Each pulldown menu has an icon to the right that brings up additional color choices.

Syntax Colors Tab

Click on the Syntax Colors tab to display the following dialog.



Element

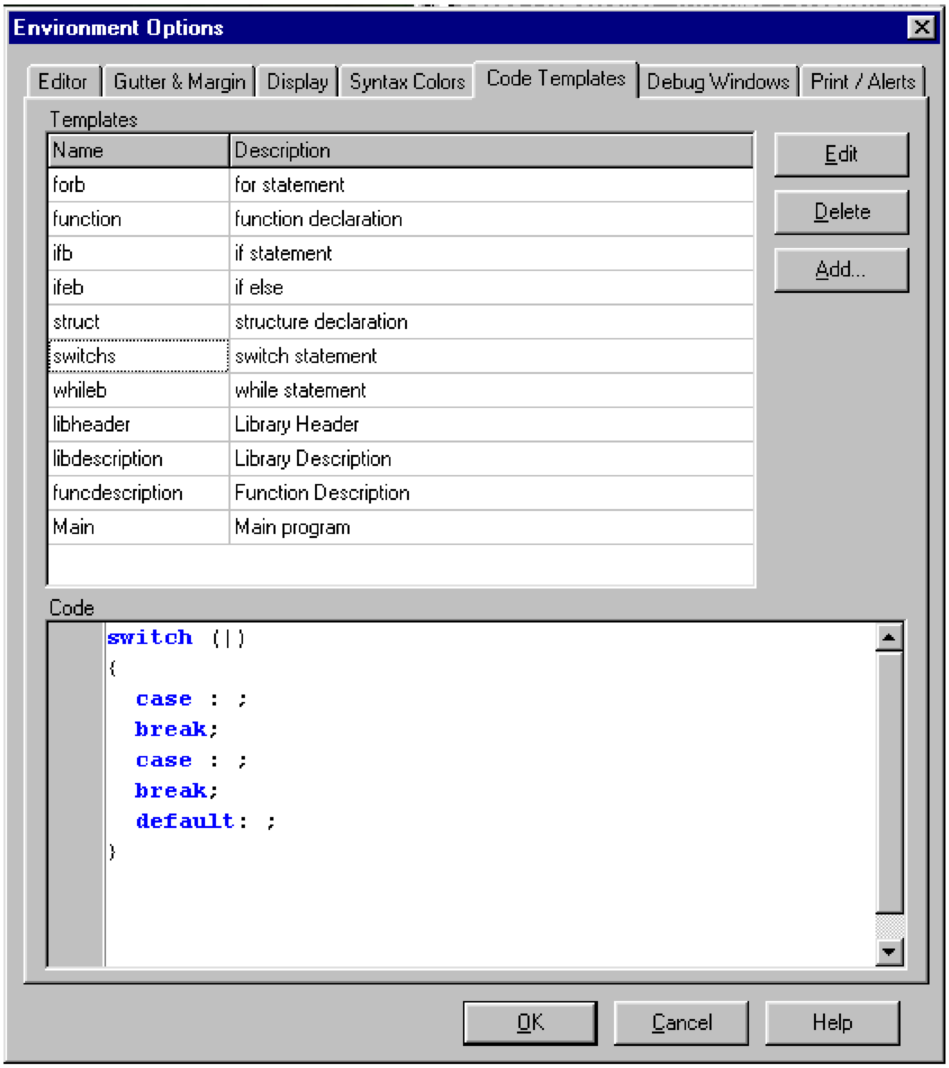
In this text box are the different elements that may be in a file (strings, comments, integers, etc.). For each one you may choose a foreground and a background color. You may also opt to use the default colors: black for foreground and white for background. In the Text attribues area of the dialog box, you may set Bold, Italic and/or Underline for the any of the elements.

Open / Save Buttons

These buttons load and save color styles into files with a .rgb extension. Clicking the Open button will bring up an Open File dialog box, where you choose a .rgb file that will set all of the syntax colors. There is a subdirectory titled Schemes under the root Dynamic C directory that has some predefined color schemes that can be used. Opening a .rgb file makes its colors immediately active in all open editor windows. If you close the Environment Options window without saving the changes, the colors will go back to whatever they were before you opened the .rgb file.

Code Template Tab

Click the Code Template tab to display the following dialog.

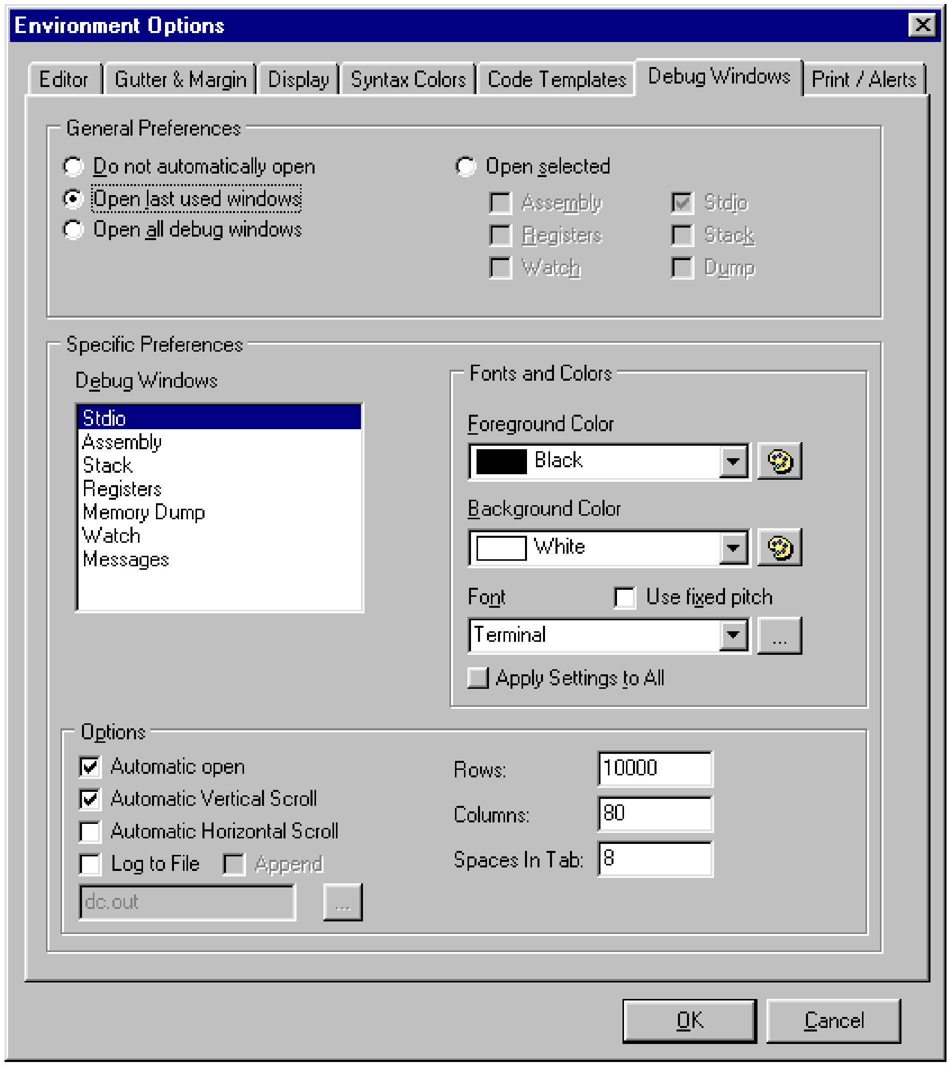


As you can see, there are several predefined templates. The Edit and Delete buttons allow the like-named operations on existing templates. The Add button gives the ability to create custom templates.

To bring up the list of defined templates, Dynamic C must be in edit mode. Then you must do one of the following: press <Ctrl+j> or right click in the editor window and choose “Insert Code Template” from the popup menu or choose the Edit command menu and select “Insert Code Template.” Clicking on the desired template name inserts that template at the cursor location.

Debug Windows Tab

Click on the Debug Windows tab to display the following dialog. Here is where you change the behavior and appearance of Dynamic C debug windows.



Under General Preferences is where you decide which debug windows will be opened after a successful compile. You may choose one of the radio buttons in this category. Selecting “Open last used windows” makes Dynamic C 8 act like Dynamic C 7.x.

Under Specific Preferences is where you customize each window. Colors and fonts are chosen here, as well as other options.

Stdio Window

The previous screen shows the options available for the Stdio window[i]. They are described here. You may modify or check as many as you would like.

Automatic open

Check this to open the Stdio window the first time `printf()` is encountered.

Automatic Vertical Scroll

Check this to force vertical scroll when text is displayed outside the view of the window. If this option is unchecked, the text display doesn't change when the bottom of the window is passed; you have to use the scroll bar to see text beyond the bottom of the window.

Automatic Horizontal Scroll

Check this to force horizontal scroll when text is displayed outside the view of the window.

Log to File

Check this to direct output to a file. If the file does not exist it will be created. If it does exist it will be overwritten unless you also check the option to append the file.

Rows

Specifies the maximum number of rows that can hold Stdio data.

Columns

Specifies the maximum number of columns that can hold Stdio data. When the maximum column is reached, output automatically wraps to the next row.

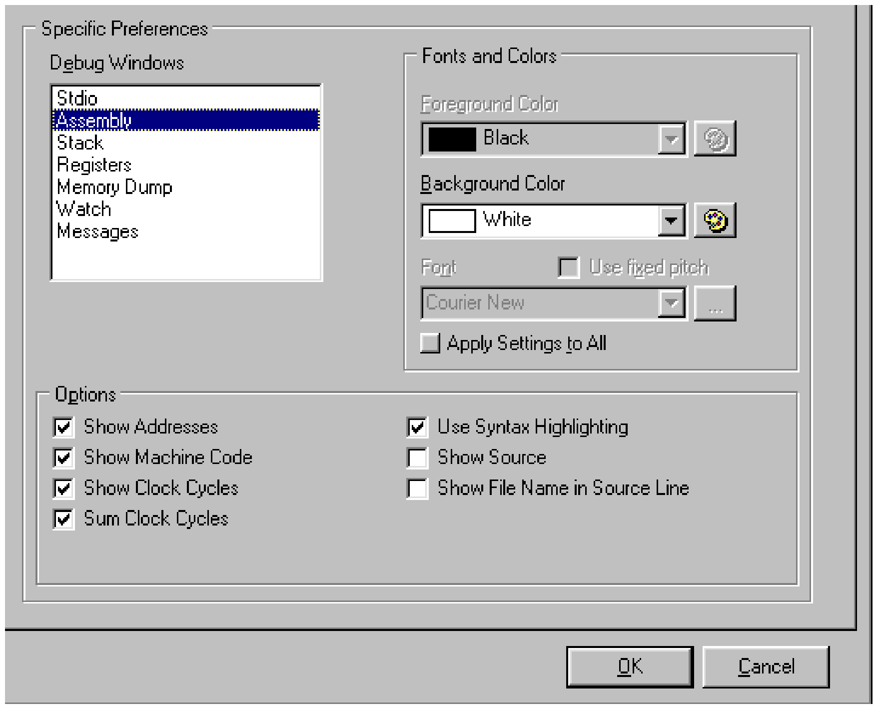
Spaces In Tab

Tab stops display as the number of spaces specified here.

i. The macro `STDIO_DEBUG_SERIAL` may be defined to redirect Stdio output to a designated serial port—A, B, C or D. For more information, please see the sample program `Samples/STDIO_SERIAL.C`.

Assembly Window

The Assembly window displays the disassembled code from the program just compiled. All but the opcode information may be toggled off and on using the checkboxes pictured below. For more information about this window see Section 11.4.2.1 on page 131.



Show Addresses
: Check this to show the logical address of the instruction in the far left column.

Show Machine Code
: Check this to show the hexidecimal number corresponding to the opcode of the instruction.

Show Clock Cycles
: Check this to show the number of clock cycles needed to execute the instruction in the far right column. Zero wait states is assumed. Two numbers are shown for conditional return instructions. The first is the number of cycles if the return is executed, the second is the number of cycles if the return is not executed.

Sum Clock Cycles
: Check this to total the clock cycles for a block of instructions. The block of instructions must be selected and highlighted using the mouse. The total is displayed to the right of the number of clock cycles of the last instruction in the block. This value assumes one execution per instruction, so looping and branching must be considered separately.

Use Syntax Highlighting

Toggle syntax highlighting. Click on the Syntax tab to set the different colors.

Show Source

Check this to display the Dynamic C statement corresponding to the assembly code.

Show File Name in Source Line

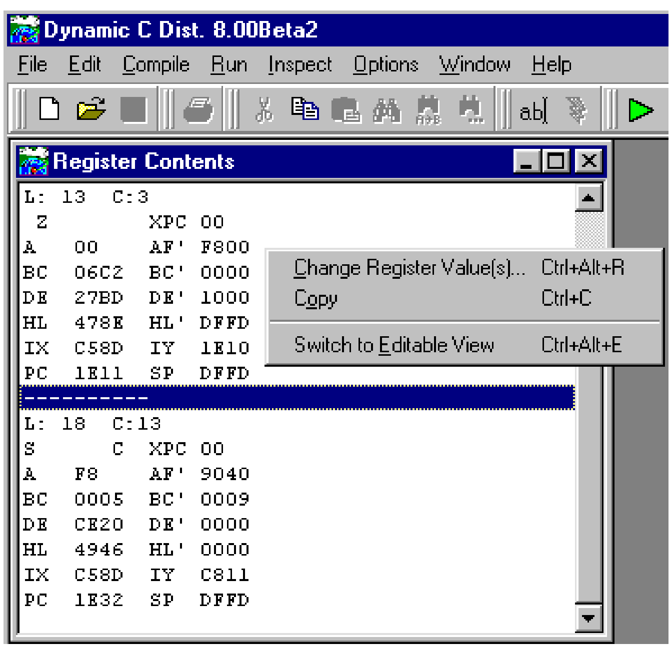
Check this to prepend the file name to the Dynamic C statements corresponding to the assembly code.

Register

For this window you must choose one of the following conditions: "Show register history" or "Show registers as editable." When the Register Contents window opens it will be in editable mode by default. Selecting "Show Register history" will override the default setting.

Show register history

In this mode, a snapshot of the register and flag values is displayed every time program execution stops. The line (L:) and column (C:) of the cursor is noted, followed by the register and flag values. The window is scrollable and sections may be selected with the mouse, then copied and pasted.

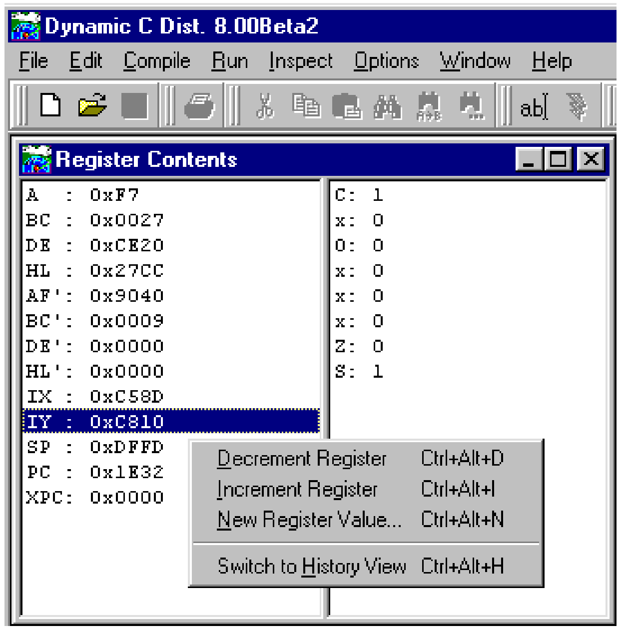


A click of the right mouse button brings up the menu pictured above. Choosing Change Register Value(s)... brings up a dialog where you can enter new values for any of the registers, except SP, PC and XPC.

Show registers as editable

In this mode, you can increment or decrement most of the registers, all but the SP, PC and XPC registers.

This screen shows the Register Contents window in editable mode. It is divided into registers on the left and flags on the right.

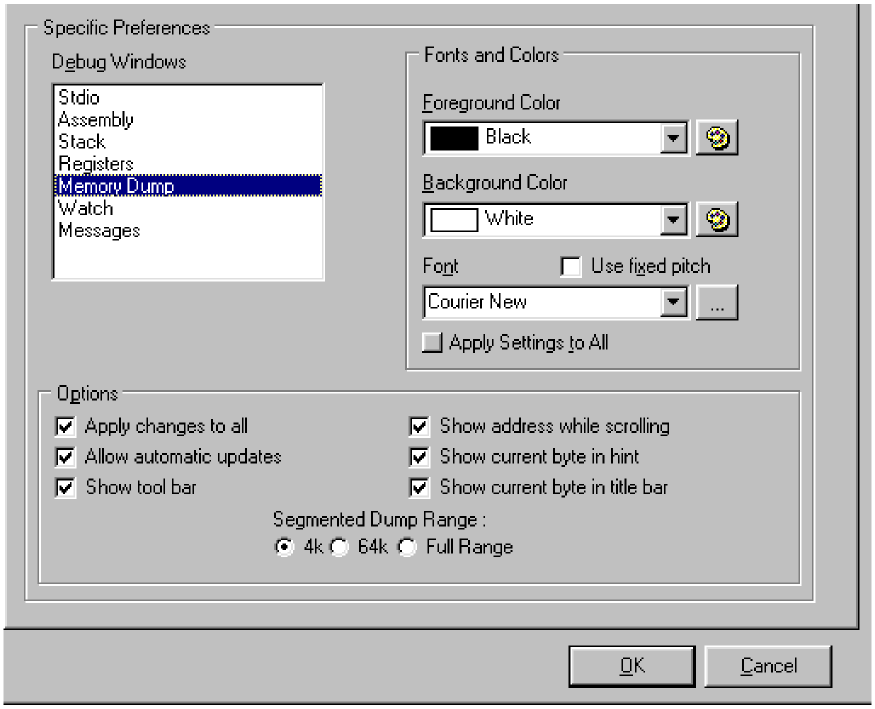


A click of the right mouse button on the register side will bring up the menu pictured here. You can switch to history view or change register values for all but the SP, PC and XPC registers.

The option New Register Value will bring up a dialog to enter the new register value. Hex values must have "0x" prepended to the value. Values without a leading "0x" are treated as decimal.

A click of the right mouse button on the flags side of the window will bring up a menu that lets you toggle the selected flag (Ctrl+Alt+T) or switch to history view (Ctrl+Alt+H).

Memory Dump



Apply changes to all

Changes made in this dialog will be applied to all memory dump windows.

Allow automatic updates

The memory dump window will be updated every time program execution stops (breakpoint, single step, etc.).

Show tool bar

Each dump window has the option of a tool bar that has a button for updating the dumped region and a text entry box to enter a new starting dump address.

Show address while scrolling

While using the scroll bar, a small popup box appears to the right of the scroll bar and displays the address of the first byte in the window. This allows you to know exactly where you are as you scroll.

Show current byte in hint

The address and value of the byte that is under the cursor is displayed in a small popup box.

Show current byte in title bar

The address and value of the byte that is under the cursor is displayed in the title bar.

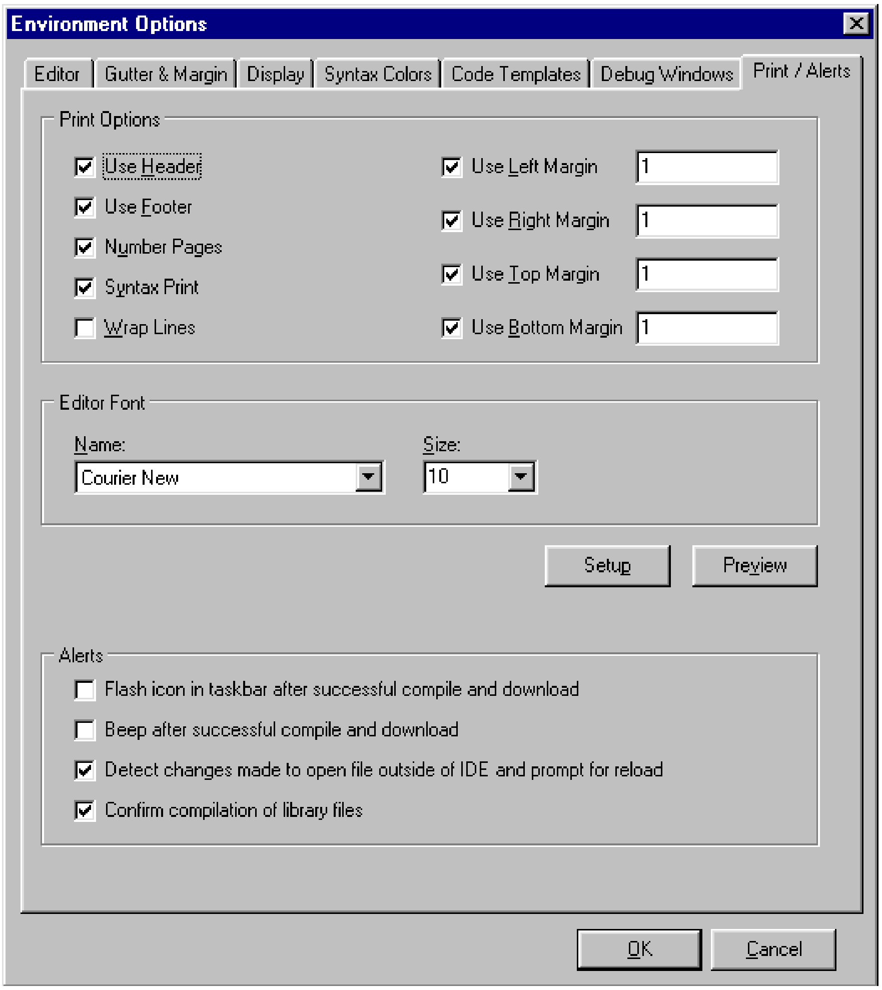
Segmented Dump Range

The memory dump window can display 3 different types of dumps. A dump of a logical address will result in a 64k scrollable region (0x0000 - 0xffff). A dump of a physical address will result in a dump of a 1M region (0x00000 - 0xfffff). A dump of an xpc:offset address will result in either a 4k, 64k or 1M dump range, depending on how this option is set.

If a 4k or 64k range is selected, the dump window will dump a 4k or 64k chunk of memory using the given xpc. If “Full Range” is selected, the window will dump 00:0000 - ff:ffff. To increment or decrement the xpc, use the “+’ and “-” buttons located below and above the scroll bar. These buttons are visible only for an xpc:offset dump where the range is either 4k or 64k.

Print/Alerts Tab

Click on the Print/Alerts tab to display the following dialog. You may access both the Page Setup dialog and Print Preview from here.



The Page Setup dialog works in conjunction with the Print/Alerts dialog. The Page Setup dialog is where you define the attributes of headers, footers, page numbering and margins for the printed page. The Print/Alerts dialog is where you enable and disable these settings. You may also change the font family and font size that will be used by the printer. This does not apply to the fonts used for headers and footers, those are defined in the Page Setup dialog.

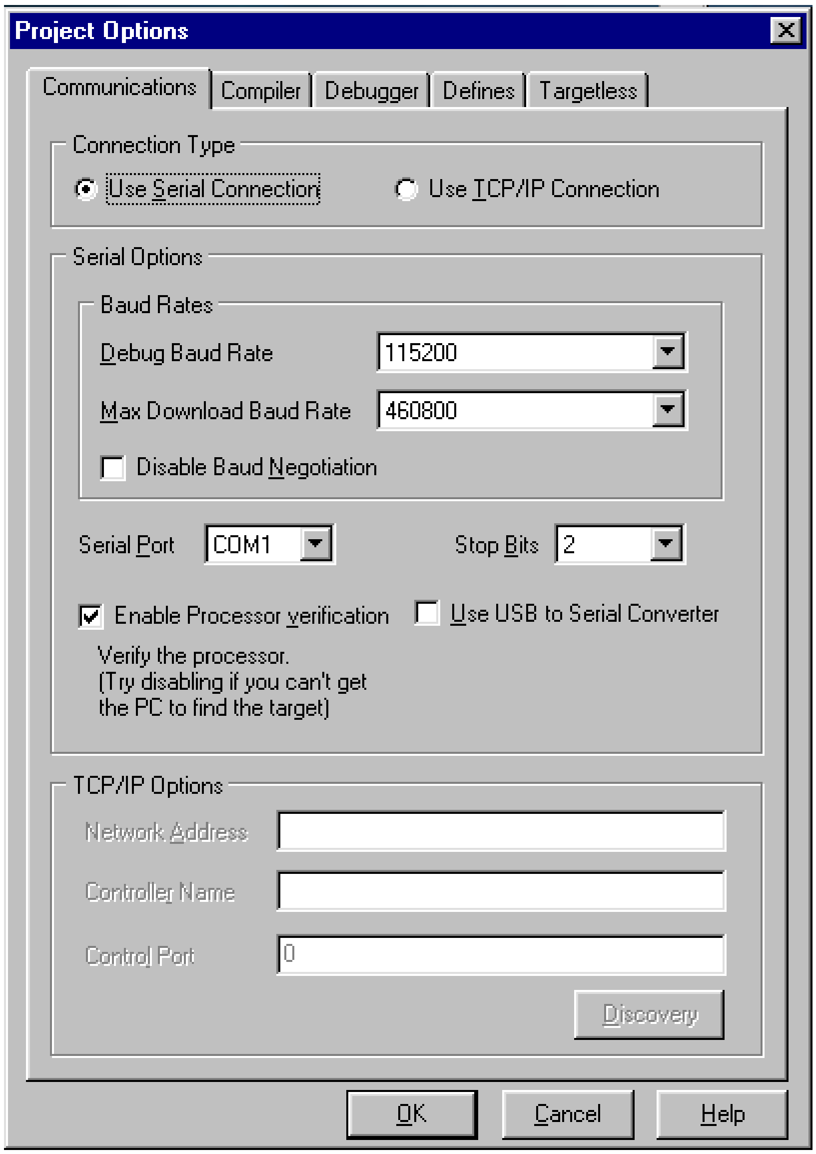
There are 4 checkboxes in the Alerts area of this dialog. The first 2 signal a successful compile and download, one with a visual signal, the other auditory. The 3rd checkbox detects if a file that is currently open in Dynamic C has been modified by an external source, i.e., a 3rd party editor; and if checked, will bring up a dialog box asking if you want to reload the modified file so that Dynamic C is working with the most current version. The last checkbox, if checked, causes Dynamic C to query when an attempt is made to compile a library file to make sure that is what is desired.

You may choose zero or more of these alerts.

Project Options

Settings used by Dynamic C to communicate with a target, and to compile and run programs are accessible by using the Project Options dialog box. The dialog box has tabs for various aspects of communicating with the target, the BIOS and the compiler.

Communications Tab



Connection Type

Choose either a serial connection or a TCP/IP connection.

Serial Options

This is where you setup for serial communication. The following options are available when the Use Serial Connection radio button is selected.

Debug Baud Rate

This defaults to 115200 bps. It is the baud rate used for target communications after the program has been downloaded.

Max Download Baud Rate

When baud negotiation is enabled, Dynamic C will start out at the selected baud rate and work downwards until it reaches one both it and the target can handle.

Disable Baud Negotiation

Dynamic C negotiates a baud rate for program download. (This helps with USB or anyone who happens to have a high-speed serial port.) This default behavior may be disabled by checking the Disable Baud Negotiation checkbox. When baud negotiation is disabled, the program will download at 115k baud or 56k baud only. When enabled, it will download at speeds up to 460k baud, as specified by Max Download Baud Rate.

Serial Port

This is the COM port of the PC that is connected to the target. It defaults to COM1.

Stop Bits

The number of stop bits used by the serial drivers. Defaults to 2.

Enable Processor Verification

Processor detection is enabled by default. The connection is normally checked with a test using the Data Set Ready (DSR) line of the PC serial connection. If the DSR line is not used as expected, a false error message will be generated in response to the connection check.

To bypass the connection check, uncheck the Enable Processor Verification checkbox. This allows custom designed systems to not connect the STATUS pin to the programming port. Also disabling the connection check allows non-standard PC ports or USB converters which might not implement the DSR line to work.

Use USB to Serial Converter

Check this checkbox if a USB to serial converter cable is being used. Dynamic C will then attempt to compensate for abnormalities in USB converter drivers. This mode makes the communications more USB/RS232 converter friendly by allowing higher download baud rates and introducing short delays at key points in the loading process. Checking this box may also help non-standard PC ports to work properly with Dynamic C.

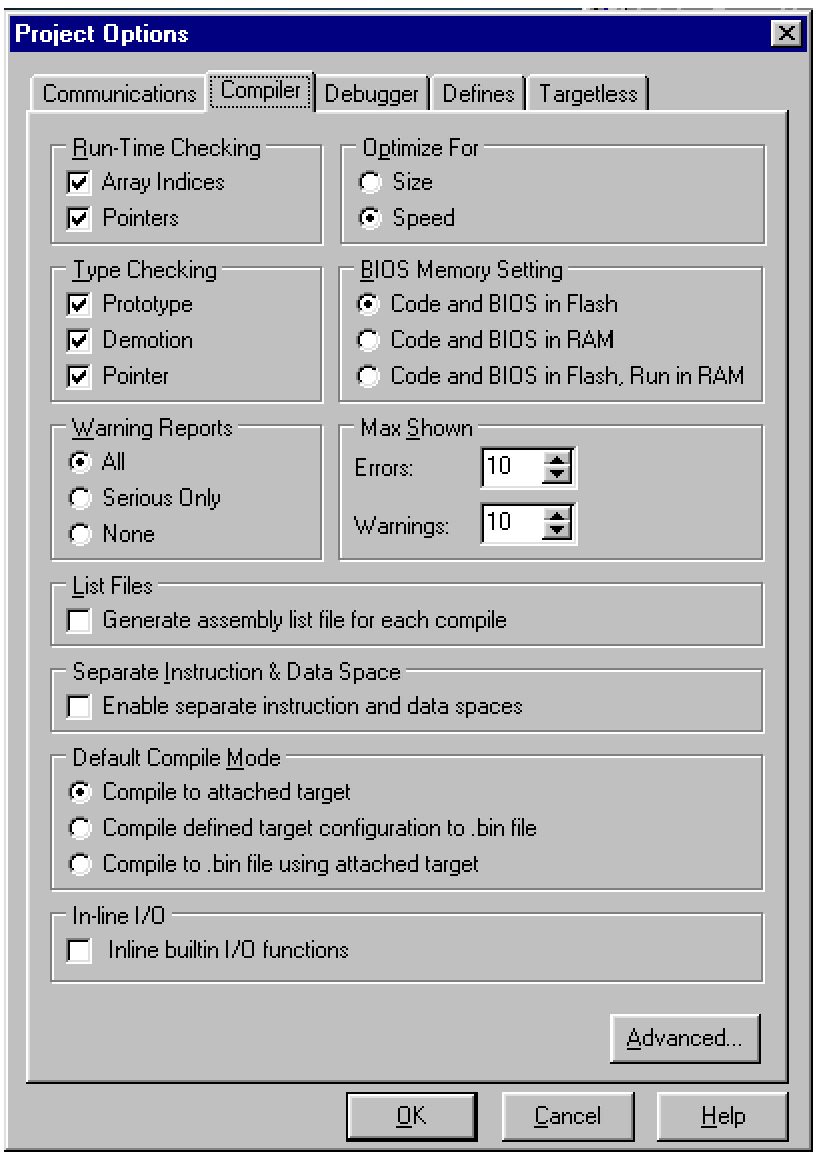
TCP/IP Options

In order to program and debug a controller across a TCP/IP connection, the Network Address field must have the IP address of either the Z-World RabbitLink board that is attached to the controller, or the IP address of a controller that has its own Ethernet interface.

To accept control commands from Dynamic C, the Control Port field must be set to the port used by the Ethernet-enabled controller. The Controller Name is for informational purposes only. The Discovery button makes Dynamic C broadcast a query to any RabbitLinks attached to the network. Any RabbitLinks that respond to the broadcast can be selected and their information will be placed in the appropriate fields.

Compiler Tab

Click on the Compiler tab to display the following dialog.



Run-Time Checking

These options, if checked, can allow a fatal error at run-time. They also increase the amount of code and cause slower execution, but they can be valuable debugging tools.

- Array Indices—Check 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 as not writable. Locations marked not writable include the *entire* root code segment. This feature adds code for every pointer reference.

Type Checking

This menu item allows the following choices:

- 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 arguments must match those defined in the prototype. Z-World recommends prototype checking because it identifies likely run-time problems. To use this feature fully, 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 scalar types is:

  ```
  char
  unsigned int
  int
  unsigned long
  long
  float
  ```

  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 warnings are considered non-serious as far as warning reports are concerned.
- 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 warnings are considered non-serious as far as warning reports are concerned.

Warning Reports

This 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 warning is a potential run-time bug. Demotions (such as converting a `long` to an `int`) are considered non-serious with regard to warning reports.

Optimize For

Allows for optimization of the program for size or 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 (where 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.

BIOS Memory Setting

A single, default BIOS source file that is defined in the system registry when installing Dynamic C is used for both compiling to RAM and compiling to flash. Dynamic C defines a preprocessor macro, `_FLASH_`, `_RAM_` or `_FAST_RAM_` depending on which of the following options is selected. This macro is used to determine the relevant sections of code to compile for the corresponding memory type.

- Code and BIOS in Flash—If you select this option, the compiler will load the BIOS to flash when cold-booting, and will compile the user program to flash where it will normally reside.
- Code and BIOS in RAM—If you select this option, the compiler will load the BIOS to RAM on cold-booting and compile the user program to RAM. This option is useful if you want to use breakpoints while you are debugging your application, but you don't want interrupts disabled while the debugger writes a breakpoint to flash (this can take 10 ms to 20 ms or more, depending on the flash type used). It is also possible to have a target that only has RAM for use as a slave processor, but this requires more than checking this option because hardware changes are necessary that in turn require a special BIOS and cold-loader.
- Code and BIOS in Flash, Run in RAM—If you select this option, the compiler will load the BIOS to flash when cold-booting, compile the user program to flash, and then the BIOS will copy the flash image to the fast RAM attached to CS2. This option supports a CPU running at a high clock speed (anything above 29 MHz).

  This is the same as the command line compiler `-mfr` option.

Max Shown

This limits the number of error and warning messages displayed after compilation.

List Files

Checking this option generates an assembly list file for each compile. A list file contains the assembly code generated from the source file.

The list file is placed in the same directory as your program, with the name `<Program Name>.LST`. The list file has the same format as the Disassembled Code window. Each C statement is followed by the generated assembly code. Each line of assembly code is broken down into memory address, machine code, opcode and number of clock cycles. See page 235 for a screen shot of the Disassembled Code window.

Separate Instruction and Data Space

When checked, this option enables separate I&D space, doubling the amount of root code and root data space available.

Please note that if you are compiling to a 128K RAM, there is only about 12K available for user code when separate I&D space is enabled.

Default Compile Mode

One of the following options will be used when Compile | Compile is selected from the main menu of Dynamic C or when the keyboard shortcut <F5> is used. The setting shown here may be overridden by choosing a different option in the Compile menu.

- Compile to attached target - a program is compiled and loaded to the attached target.
- Compile defined target configuration to .bin file - a program is compiled and the image written to a .bin file. The target configuration used in the compile is taken from the parameters specified in Options | Project Options. The Targetless tab allows you to choose an already defined board type or you may define one of your own.
- Compile to .bin file using attached target - a program is compiled and the image written to a .bin file using the parameters of the attached controller.

In-line I/O

If checked, the built-in I/O functions (`WrPortI()`, `RdPortI()`, `BitWrPortI()` and `BitRdPortI()`) will have efficient inline code generated instead of function calls if all arguments are constants, with the exception of the 3rd parameter of `BitWrPortI()` and `WrPortI()`, which may be any valid expression.

If this box is checked, but a call to one of the aforementioned functions is made with non-constant arguments, (with the exception of the 3rd parameter for the 2 write functions) then a normal function call is generated.

Advanced... Button

Click on this button to reveal the Advanced Compiler Options dialog. The options are:

Default Project Source File

Use this option to set a default source file for your project. If this box is checked, then when you compile, the source file named here will be used and not the file that is in the active editor window. If the file named here is not open, it will be opened into a new editor window, which will be the new active editor window.

User Defined BIOS File

Use this option to change from the default BIOS to a user-specified file. Enter or select the file using the browse button/text box underneath this option. The check box labeled use must be selected or else the default file BIOS defined in the system registry will be used. Note that a single BIOS file can be made for compiling both to RAM and flash by using the preprocessor macros `_FLASH_` or `_RAM_`. These two macros are defined by the compiler based on the currently selected radio button in the BIOS Memory Setting group box.

User Defined Lib Directory File

The Library Lookup information retrieved with <Ctrl+H> is parsed from the libraries found in the `lib.dir` file, which is part of the Dynamic C installation. Checking the Use box for User Defined Libraries File, allows the parsing of a user-defined replacement for `lib.dir` when Dynamic C starts. Library files must be listed in `lib.dir` (or its replacement) to be available to a program.

If the function description headers are formatted correctly (See “Function Description Headers” on page 43.), the functions in the libraries listed in the user-defined replacement for `lib.dir` will be available with <Ctrl+H> just like the user-callable functions that come with Dynamic C.

This is the same as the command line compiler `-lf` option.

Watch Code

Allow any expressions in watch expressions

This option causes any compilation of a user program to pull in all the utility functions used for expression evaluation.

Restricting watch expressions (may save root code space)

Choosing this option means only utility code already used in the application program will be compiled.

Debug Instructions and BIOS Inclusion

Include RST 28 instructions

If this is checked, the debug and nodebug keywords and compiler directives work as normal. Debug code consists mainly of RST 28h instructions inserted after every C statement. This option also controls the definition of a compiler-defined macro symbol, DEBUG_RST. If the menu item is checked, then DEBUG_RST is set to one, otherwise it is zero.

If the option is not checked, the compiler marks all code as nodebug and debugging is not possible.

The only reason to check this option if debugging is finished and the program is ready to be deployed, is to allow some current (or planned) diagnostic capability of the Rabbit Field Utility (RFU) to work in a deployed system. This option affects both code compiled to .bin files and code compiled to the target. To run the program after compiling to the target with this option, disconnect the target from the programming port and reset the target CPU.

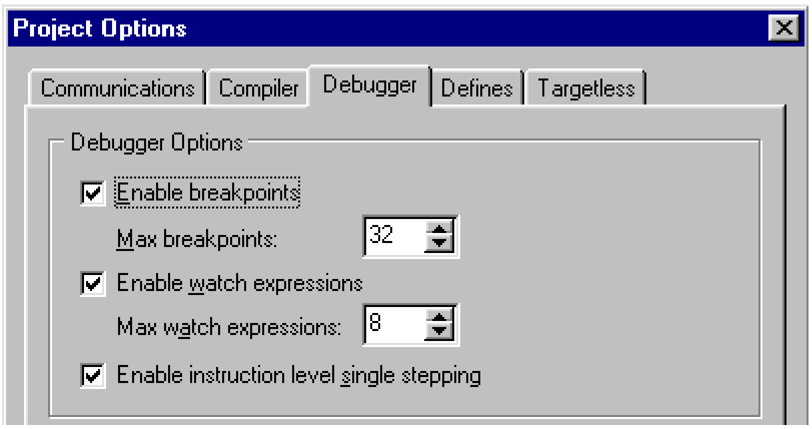
Include BIOS

If this is checked, the BIOS, as well as the user program, will be included in the `.bin` file. If you are creating a special program such as a cold loader that starts at address 0x0000, then this option should be unchecked.

This option is not available when you are compiling a program to the attached target controller.

Debugger Tab

Click on the Debugger tab to display the following dialog. This is where you can disable parts of the debug kernel to save room if there are tight code space requirements.



Enable Breakpoints

If this box is checked, the debug kernel will be able to toggle breakpoints on and off and will be able to stop at set breakpoints. This is where you set the maximum number of breakpoints the debug kernel will support. The debug kernel uses a small amount of root RAM for each breakpoint, so reducing the number of breakpoints will slightly reduce the amount of root RAM used.

If this box is unchecked, the debug kernel will be compiled without breakpoint support and the user will receive an error message if they attempt to add a breakpoint.

Enable Watch Expressions

If this is checked, watch expressions will be enabled. This is where you set the maximum number of watch expressions the debug kernel will support. The debug kernel uses a small amount of root RAM for evaluating each watch expression, so reducing the amount of watches will slightly reduce the amount of root RAM used.

With it unchecked, the debug kernel will be compiled without watch expressions support and the user will receive an error message if they attempt to add a watch expression.

Enable Instruction Level Single Stepping

If this is checked when the assembly window is open, single stepping will be by instruction rather than by C statement. Unchecking this box will disable instruction level single stepping on the target and, if the assembly window is open, the debug kernel will step by C statement.

Defines Tab

The Defines tab brings up a dialog box with a window for entering (or modifying) a list of defines that are global to any source file programs that are compiled and run. The macros that are defined here are seen by the BIOS during its compilation.

Syntax:

DEFINITION[DELIMETER DEFINITION[DELIMETER DEFINITION[...]]]

DEFINITION: MACRONAME[[WS]=[WS]VALUE]

DELIMETER: ';' or 'newline'

MACRONAME: the same as for a macro name in a source file

WS: [SPACE[SPACE[...]]]

VALUE: CHR[CHR[...]]

CHR: any character except the delimeter character '`;`', which is entered as the character pair "`\;`"

Notes:

- Do not continue a definition in this window with '`\`', simply continue typing as a long line will wrap.
- In this window hitting the Tab key will not enter a tab character (`\t`), but will tab to the OK button.
- The command line compiler honors all macros defined in the project file that it is directed to use with the project file switch, `-pf`, or `default.dcp` if `-pf` is not used. See command line compiler documentation.
- A macro redefined on the command line will supercede the definition read from the project file.

Examples and file equivalents:

Example:

```
DEF1;MAXN=10;DEF2
```

Equivalent:

```
#define DEF1
#define MAXN 10
#define DEF2
```

Example:

```
DEF1
MAXN = 10
DEF2
```

Equivalent:

```
#define DEF1
#define MAXN 10
#define DEF2
```

Example:

```
STATEMENT = A + B = C\;;DEF1=10
```

Equivalent:

```
#define STATEMENT A + B = C;
#define DEF1 10
```

Example:

```
STATEMENT = A + B = C\;
FORMATSTR = "name = %s\n"
DEF1=10
```

Equivalent:

```
#define STATEMENT A + B = C;
#define FORMATSTR "name = %s\n"
#define DEF1 10
```

Targetless Tab

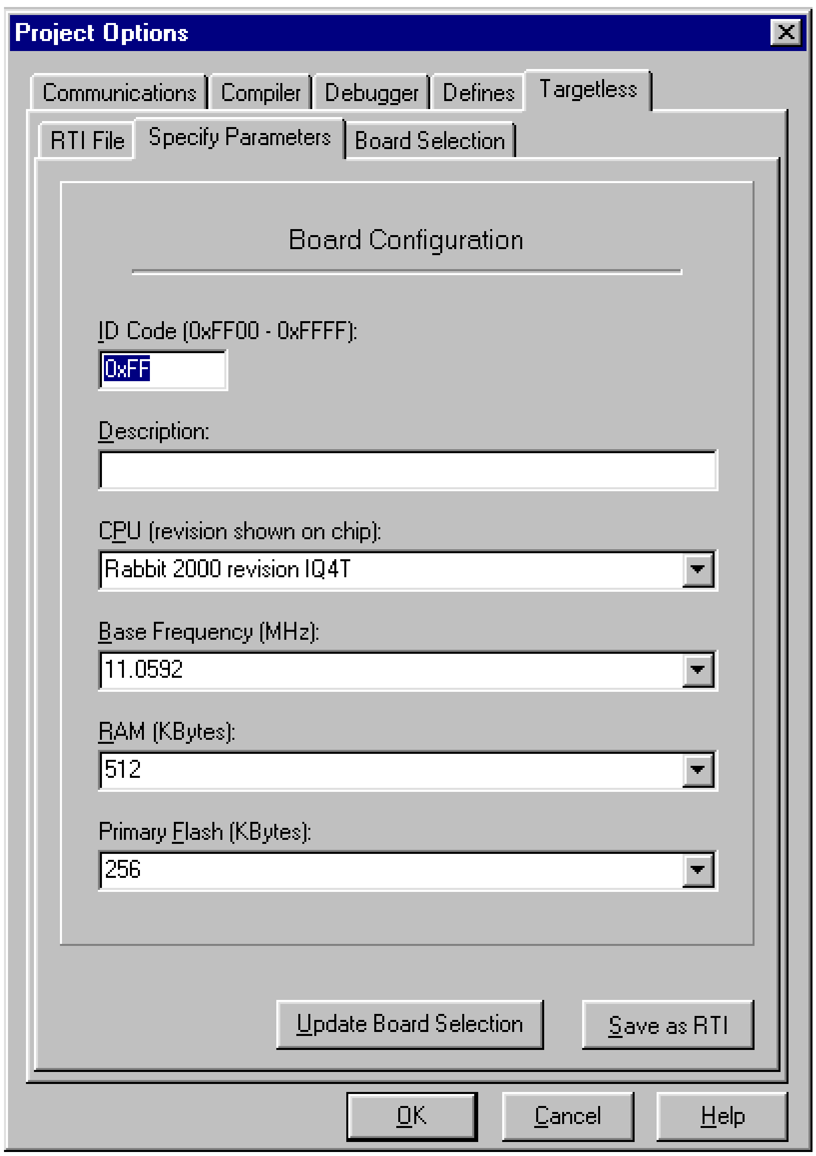
Click on the Targetless tab to reveal 3 additional tabs: RTI File, Specify Parameters and Board Selection.

RTI File

Click on this tab to open a Rabbit Target Information (RTI) file for viewing. The file is read-only. You may not edit RTI files, but you may create one by selecting an entry in the Board Selection list and clicking on the button Save as RTI. Or you may define a board configuration in the Specify Parameters dialog and then save the information in an RTI file. Details follow.

Specify Parameters

This is where you may define the parameters of a controller for later use in targetless compilations.

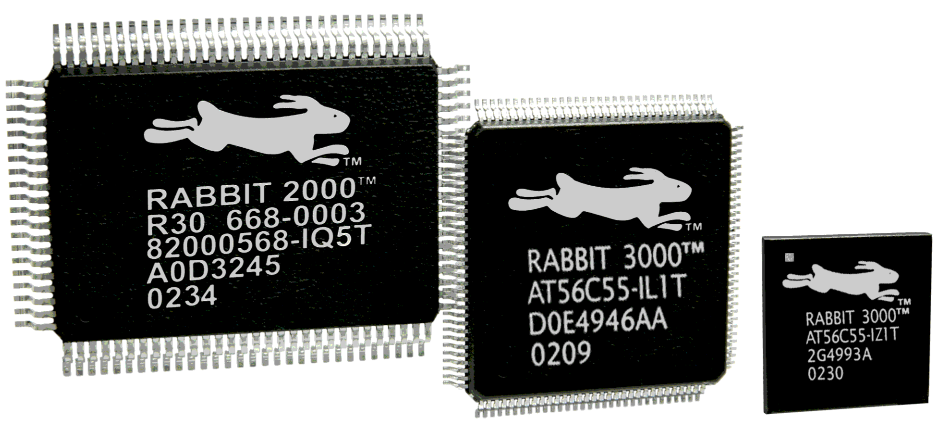


The result may be saved to a RTI file for later use, or the result may be saved to the list of board configurations. This list is viewable from the Board Selection tab. The highlighted entry in the list of board configurations is the one that will be used when the compilation uses a defined target configuration, that is, when the Default Compile Mode on the Compiler tab is set to “Compile defined target configuration to .bin file” and Compile or Compile to .bin file is chosen from the Compile menu.

If you save to the list of board configurations by clicking on the button Update Board Selection, then you must fill in all fields of the dialog. The baud rate, calculated from the value in the Base Frequency (MHz) field, only applies to debugging. The fastest baud rate for downloading is negotiated between the PC and the target.

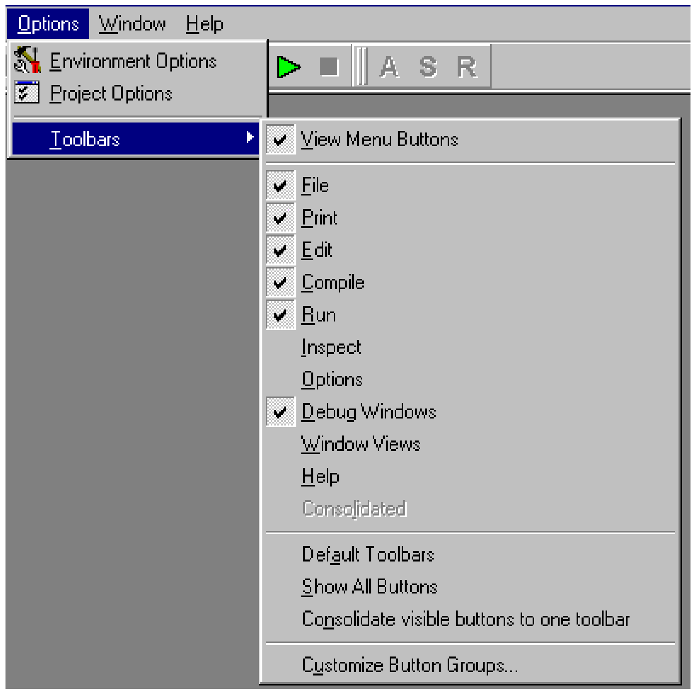
To save to an RTI file only requires an entry in the CPU field. Please see Technical Note 231 for information on the specifics of the Rabbit CPU revisions.

The correct choice for the CPU field is found on the chip itself. The information is printed on the 3rd line from the top on the Rabbit 2000 and the 2nd line from the top on the Rabbit 3000. The Rabbit 2000 revision is IQ#T, where # is the revision number and the letters are associated information. The Rabbit 3000 revision is IL#T or IZ#T, where # is the revision number and the letters are associated information.



Toolbars

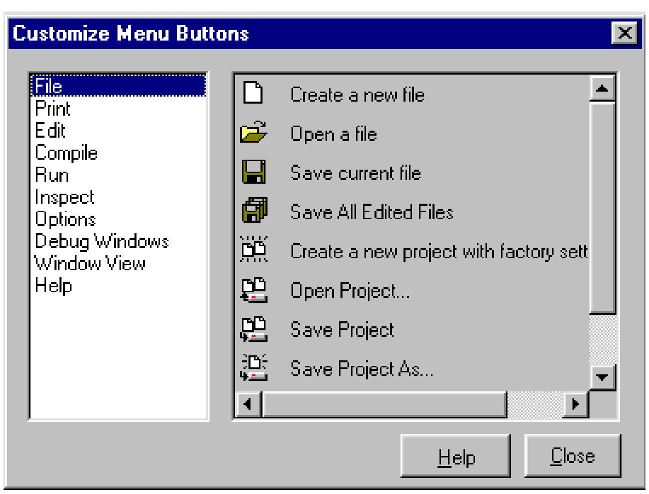
Selecting this menu item reveals a list of all menu button groups, i.e., the groups of icons that appear in toolbars beneath the title bar and the main menu items (File, Edit, …). This area is called the control bar. Uncheck View Menu Buttons to remove the control bar from the Dynamic C window. Any undocked toolbars (i.e., toolbars floating off the control bar) will still be visible. You undock a toolbar by placing the cursor on the 2 vertical lines on the left side of the toolbar and dragging it off the control bar.



Each menu button group (File, Edit, Compile, Run, Options, Watch, Debug Window, WindowView and Help) has a checkbox for choosing whether or not to make its toolbar visible on the control bar.

To quickly return to only showing the icons visible by default, select Default Toolbars.

Select the option, Consolidate visible buttons to one toolbar to do exactly that—create one toolbar containing all visible icons. Doing this, enables the option Consolidated, which toggles the visibility of the consolidated toolbar, even when it is undocked from the control bar.

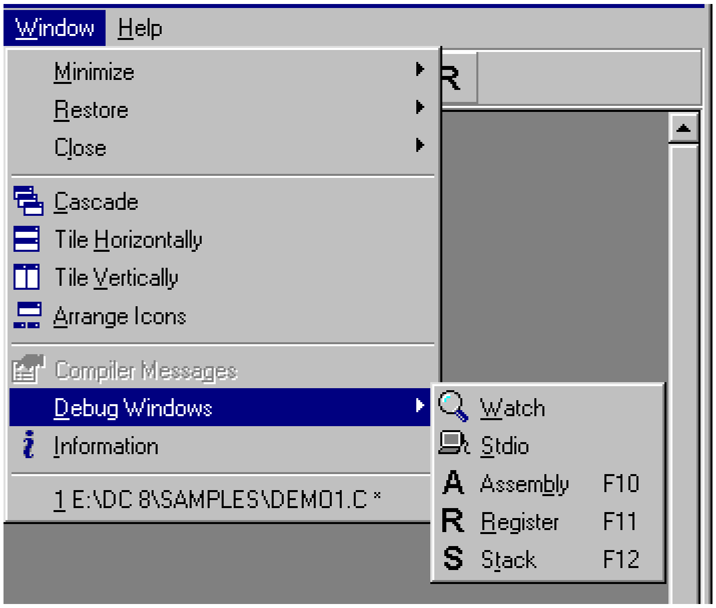


.Select Customize Button Groups to bring up the Customize Menu Buttons window. This window allows you to change which buttons are associated with which button group on the toolbar. Choose a button group on the left side of the window; this causes the icons for the buttons in that group to display on the right side of the window. Click and drag an icon from the right side of the window to the desired button group on the toolbar.

To remove an icon from its button group, click and drag the icon off the toolbar or to another button group on the toolbar. The Customize Menu Buttons window must be open to change the position of an icon on the toolbar.

14.2.7 Window Menu

Click the menu title or press <Alt+W> to display the Window menu.



You can choose to minimize, restore or close all open windows or just the open debug window or just the open editor windows. The second group of items is a set of standard Windows commands that allow the application windows to be arranged in an orderly way.

The Compiler Messages option is a toggle for displaying that window. This is only available if an error or warning occurred during compilation.

The Debug Windows option opens a secondary menu, whose items are toggles for displaying the like-named debug windows. You can scroll these windows to view larger portions of data, or copy information from these windows and paste the information as text anywhere. More information is given below for each window.

At the bottom of the Window menu is a list of current windows, including source code windows. Click on one of these items to bring its window to the front.

Watch

Select Watch to activate or deactivate the Watches window. The Add Watch command on the INSPECT menu will do this too. The Watches window displays the results whenever Dynamic C evaluates watch expressions.

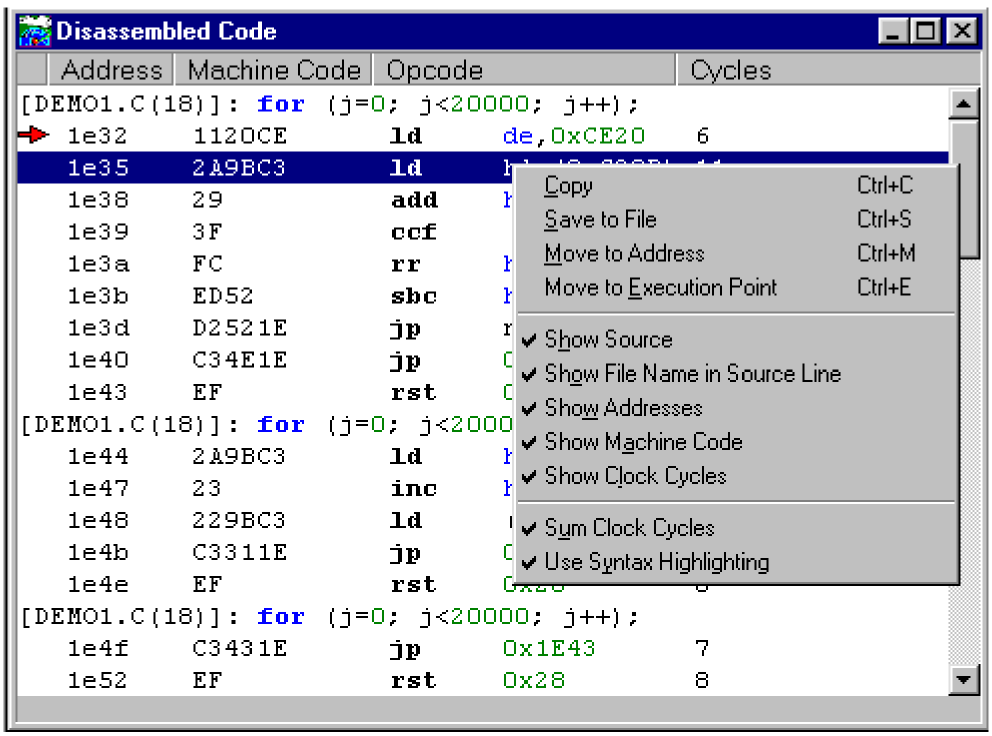
Stdio

Select Stdio to activate or deactivate 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 if it is not already open, unless “Automatic open” is unchecked in the Debug Windows dialog in Options | Environment Options.

Assembly

Select Assembly to activate or deactivate the Disassembled Code window. The Disassembled Code window (aka., the Assembly window) displays machine code generated by the compiler in assembly language format.

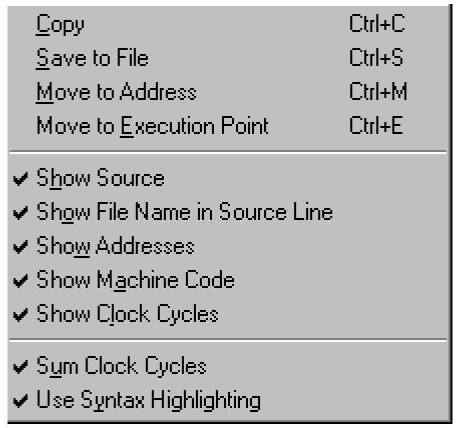
The Disassemble at Cursor or Disassemble at Address commands from the Inspect menu also activate the Disassembled Code window.



The Disassembled Code window displays Dynamic C statements followed by the assembly instructions for that statement. Each instruction is represented by the memory address on the far left, followed by the code bytes for the instruction at that address, followed by the mnemonics for the instruction. The last column shows the number of cycles for the instruction, assuming no wait states. The total cycle time for a block of instructions will be shown at the lowest row in the block in the cycle-time column, if that block is selected and highlighted with the mouse. The total assumes one execution per instruction, so the user must take looping and branching into consideration when evaluating execution times.

Use the mouse to select several lines in the Assembly window, and the total cycle time for the instructions that were selected will be displayed to the lower right of the selection. If the total includes an asterisk, that means an instruction with an indeterminate cycle time was selected, such as `ldir` or `ret nz`.

Right click anywhere in the Disassembled Code window to display the following popup menu:



Copy

Copies selected text in the Disassembled Code window to the clipboard.

Save to File

Opens the Save As dialog to save text selected in the Disassembled Code window to a file. If you do not specify an extension, `.dasm` will be appended to the file name.

Move to Address

Opens the Disassemble at Address dialog so you can enter a new address.

Move to Execution Point

Highlights the assembly instruction that will execute next and displays it in the Disassembled Code window.

All but the last menu option of the remaining items in the popup menu toggle what is displayed in the Disassembled Code window. The last menu option, Use Syntax Highlighting, displays the colors that were set for the editor window in the Disassembled Code window as well.

To resize a column in the assembly window, move the mouse pointer to one of the vertical bars that is between each of the column headers. For instance, if you move the mouse pointer between "Address" and "Machine Code," the pointer will change from an arrow to a vertical bar with arrows pointing to the right and left. Hold the left mouse button down and drag to the right or left to grow or shrink the column.

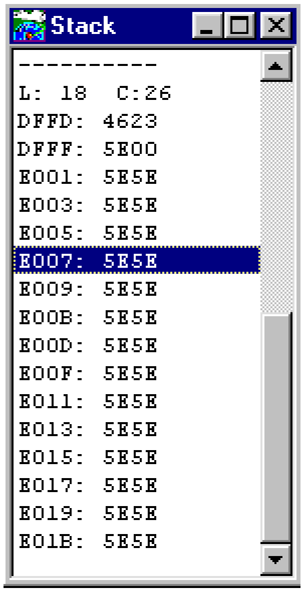
Registers

Select Registers to activate or deactivate the Register window. This window displays the processor register set, including the status register. Letter codes indicate the bits of the status register (F register). The window also shows the source-code line and column at which the snapshot of the register was taken.

It is possible to scroll back to see the progression of successive register snapshots. Register values may be changed when program execution is stopped Registers PC, XPC, and SP may not be edited as this can adversely affect program flow and debugging.

See “Register” on page 216 for more details on this window.

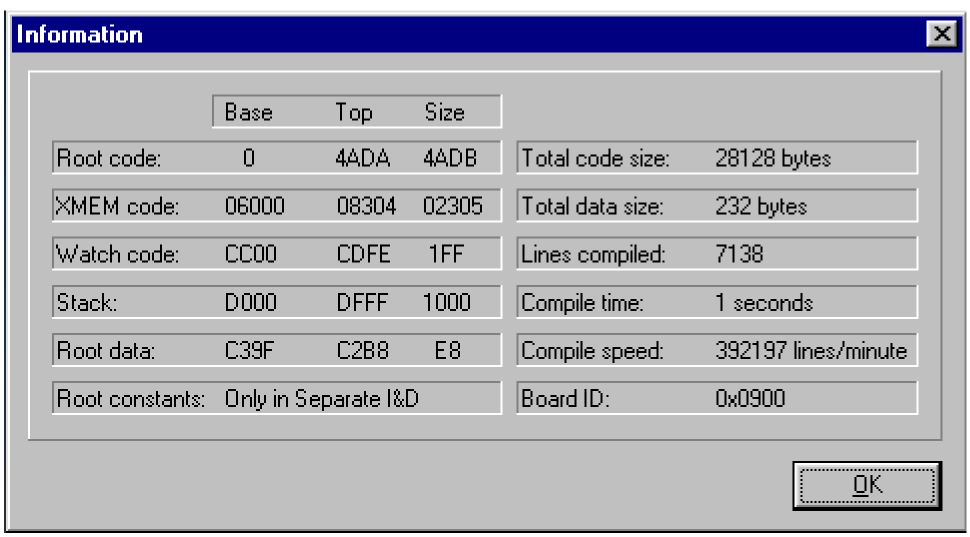
Stack



Click the Stack command to activate or deactivate the Stack window. The Stack window displays the top 32 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 progression of successive stack snapshots.

Information

Select the Information menu option to activate the Information window.



The Information window displays how the memory is partitioned and how well the compilation went.

14.2.8 Help Menu

Click the menu title or press <Alt+H> to select the HELP menu. The choices are given below:

Online Documentation

Opens a browser page and displays a file with links to other manuals. When installing Dynamic C from CD, this menu item points to the hard disk; after a Web upgrade of Dynamic C, this menu item optionally points to the Web.

Keywords

Opens a browser page and displays an HTML file of Dynamic C keywords, with links to their descriptions in this manual.

Operators

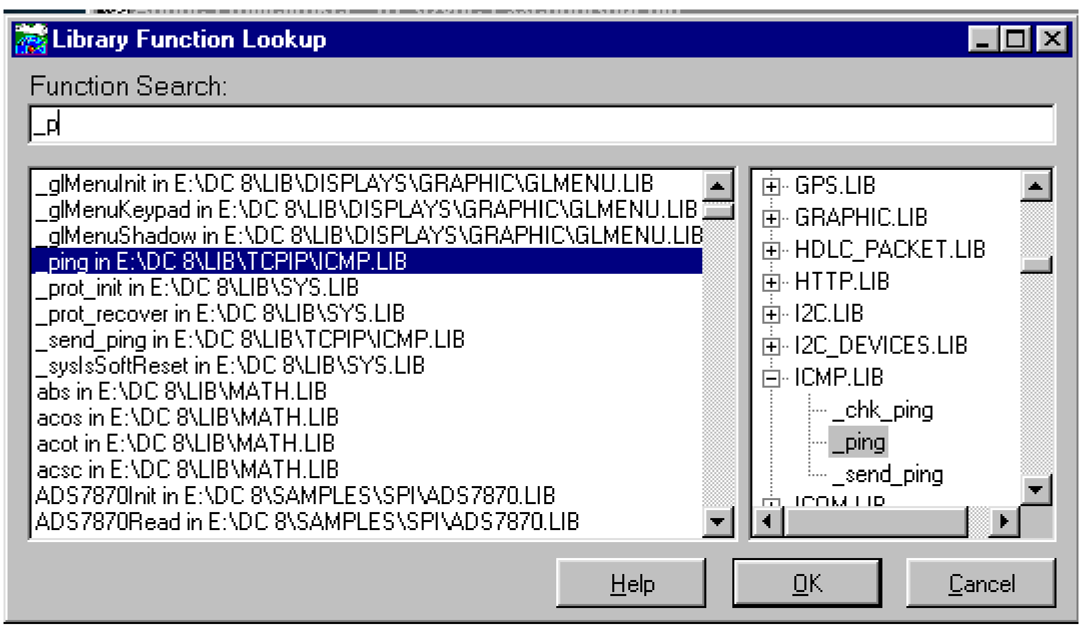
Opens a browser page and displays an HTML file of Dynamic C operators, with links to their descriptions in this manual.

HTML Function Reference

Opens a browser page and displays an HTML file that has two links, one to Dynamic C functions listed alphabetically, the other to the functions listed by functional group. Each function listed is linked to its description in the *Dynamic C Function Reference Manual*.

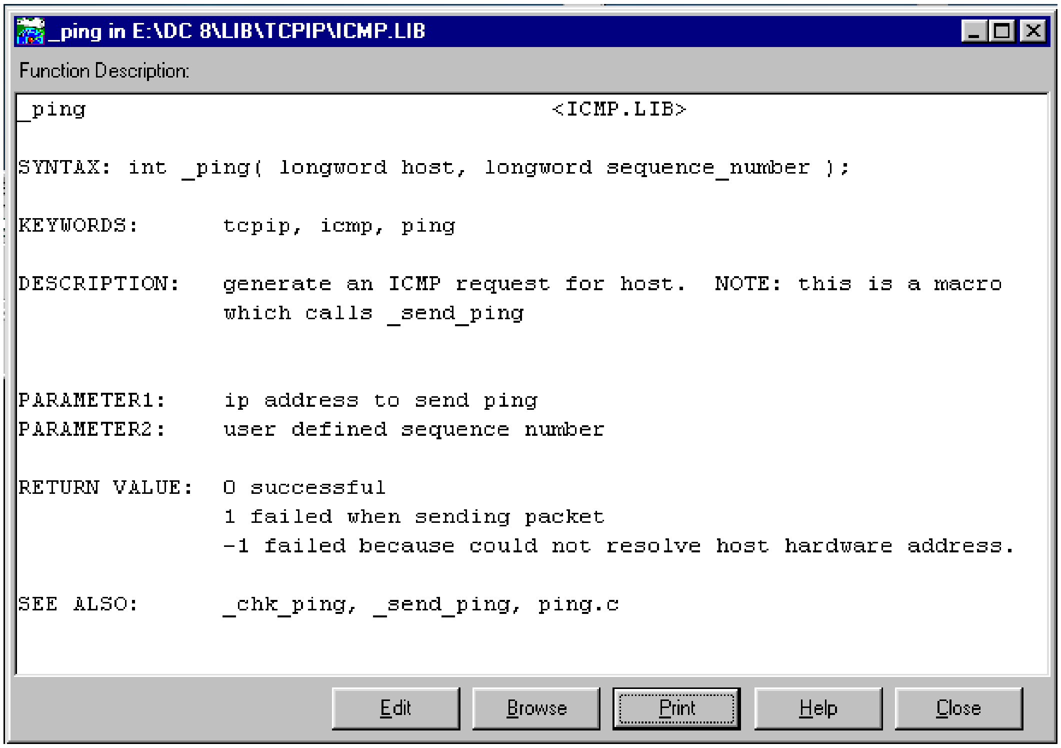
Function Lookup

Displays descriptions for library functions. The keyboard shortcut is <Ctrl+H>.



Choosing a function is done in one of several ways. You may type the function name in the Function Search entry box. Notice how both scroll areas underneath the entry box display the first function that matches what you type. The functions to the left are listed alphabetically, while those on the right are arranged in a tree format, displaying the libraries alphabetically with their functions collapsed underneath. You may scroll either of these two areas and have whatever you select in one area reflected in the other area and in the text entry box. Click OK or press <Enter> to bring up the Function Description window.

If the cursor is on a function when Help | Function Lookup is selected (or when <Ctrl+H> is pressed) then the Library Function Lookup dialog is skipped and the Function Description window appears directly.



If you click the Edit button, the Function Description window will close and the library that contains the function that was in the window will open in an editor window. The cursor will be placed at the function description of interest.

Clicking on the Browse button will open the Library Function Lookup window to allow you to search for a new function description. Multiple Function Description windows may be open at the same time.

Instruction Set Reference

Invokes an on-line help system and displays the alphabetical list of instructions for the Rabbit 2000 microprocessor and the Rabbit 3000 microprocessor.

I/O Registers

Invokes an on-line help system that provides the bit values for all of the Rabbit I/O registers.

Keystrokes

Invokes an on-line help system and displays the keystrokes page. Although a mouse or other pointing device may be convenient, Dynamic C also supports operation entirely from the keyboard.

Contents

Invokes an on-line help system and displays the contents page. From here view explanations of various features of Dynamic C.

Tech Support

Opens a browser window to the Rabbit Semiconductor Technical Support Center web page, which contains links to user forums, downloads for Dynamic C and information about 3rd party software vendors and developers.

Register Dynamic C

Allows you to register your copy of Dynamic C. A dialog is opened for entering your Dynamic C serial number. From there you will be guided through the very quick registration process.

Tip of the Day

Brings up a window displaying some useful information about Dynamic C. There is an option to scroll to another screen of Dynamic C information and an option to disable the feature. This is the same window that is displayed when Dynamic C initializes.

About

The About command displays the Dynamic C version number and the copyright notice.

15. Command Line Interface

The Dynamic C command line compiler (`dccl_cmp.exe`) performs the same compilation and program execution as its GUI counterpart (`dcrabxx.exe`), but is invoked as a console application from a DOS window. It is called with a single source file program pathname as the first parameter, followed by optional case-insensitive switches that alter the default conditions under which the program is run. The results of the compilation and execution, all errors, warnings and program output, are directed to the console window and are optionally written or appended to a text file.

15.1 Default States

The command line compiler uses the values of the environment variables that are in the project file indicated by the **-pf** switch, or if the **-pf** switch is not used, the values are taken from `default.dcp`. For more information, please see Chapter 16, "Project Files" on page 259.

The command line compiler will compile and run the specified source file. The exception to this is when the project file "Default Compile Mode" is one of the options which compiles to a .bin file, in which case the command line compiler will not run the program but will only compile the source to a .bin file. Command line help displayed to the console with

```
dccl_cmp
```

gives a summary of switches with defaults from the default project file, `default.dcp`, and

```
dccl_cmp -pf specified_project_name.dcp
```

gives a summary of switches with defaults from the specified project file. All project options including the default compile mode can be overridden with the switches described in Section 15.4.

15.2 User Input

Applications requiring user input must be called with the **-i** option:

```
dccl_cmp myProgram.c -i myProgramInputs.txt
```

where `myProgramInputs.txt` is a text file containing the inputs as separate lines, in the order in which `myProgram.c` expects them.

15.3 Saving Output to a File

The output consists of all program printf's as well as all error and warning messages.

Output to a file can be accomplished with the **-o** option

```
dccl_cmp myProgram.c -i myProgramInputs.txt -o myOutputs.txt
```

where `myOutputs.txt` is overwritten if it exists or is created if it does not exist.

If the **-oa** option is used, `myOutputs.txt` is appended if it exists or is created if it does not.

15.4 Command Line Switches

Each switch must be separated from the others on the command line with at least one space or tab. Extra spaces or tabs are ignored. The parameter(s) required by some switches must be added as separate text immediately following the switch. Any of the parameters requiring a pathname, including the source file pathname, can have imbedded spaces by enclosing the pathname in quotes.

15.4.1 Switches Without Parameters

-b

Description: Use compile mode: Compile to .bin file using attached target.

Factory Default: Compile mode: Compile to attached target.

GUI Equivalent: Compile program (F5) with Default Compile Mode set to "Compile to .bin file using attached target" in Compiler tab of Project Options dialog.

-bf-

Description: Undo user-defined BIOS file specification.

Factory Default: None.

GUI Equivalent: This is an advanced setting, viewable by clicking on the “Advanced” radio button at the bottom of the Compiler tab of Project Options dialog. Uncheck the “User Defined BIOS File” checkbox.

-br

Description: Use compile mode: Compile defined target configuration to .bin file

Factory Default: Compile mode: Compile to attached target.

GUI Equivalent: Compile program (F5) with Default Compile Mode set to "Compile defined target configuration to .bin file" in Compiler tab of Project Options dialog.

-h+

Description: Print program header information.

Factory Default: No header information will be printed.

GUI Equivalent: None.

Example: `dccl_cmp samples\demo1.c -h -o myoutputs.txt`

Header text preceding output of program:

****

4/5/01 2:47:16 PM
dccl_cmp.exe, Version 7.10P - English
samples\demo1.c
Options: -h+ -o myoutputs.txt
Program outputs:

Note: Version information refers to `dcwd.exe` with the same compiler core.

-h-

Description: Disable printing of program header information.

Factory Default: No header information will be printed.

GUI Equivalent: None.

-id+

Description: Enable separate instruction and data space.

Factory Default: Separate I&D space is disabled.

GUI Equivalent: Check “Separate Instruction & Data Space” in Project Options | Compiler.

-id-

Description: Disable separate instruction and data space.

Factory Default: Separate I&D space is disabled.

GUI Equivalent: Uncheck “Separate Instruction & Data Space” in the Project Options | Compiler dialog box.

-ini

Description: Generates inline code for `WrPortI()`, `RdPortI()`, `BitWrPortI()` and `BitRdPortI()` if all arguments are constants.

Factory Default: No inline code is generated for these functions.

GUI Equivalent: Check “Inline builin I/O functions” in the Project Options | Compiler dialog box.

-lf-

Description: Undo Library Directory file specification.

Factory Default: No Library Directory file is specified.

GUI Equivalent: This is an advanced setting, viewable by clicking on the “Advanced” radio button at the bottom of the Project Options | Compiler dialog box. Uncheck “User Defined Lib Directory File.”

-mf

Description: Memory BIOS setting: Flash.

Factory Default: Memory BIOS setting: Flash.

GUI Equivalent: Select “Code and BIOS in Flash” in the Project Options | Compiler dialog box.

-mfr

Description: The BIOS and code are compiled to flash, and then the BIOS copies the flash image to RAM to run the code.

Factory Default: Memory BIOS setting: Flash

GUI Equivalent: Select “Code and BIOS in Flash, Run in RAM” in the Project Options | Compiler dialog box.

-mr

Description: Memory BIOS setting: RAM.

Factory Default: Memory BIOS setting: Flash.

GUI Equivalent: Select “Code and BIOS in RAM” in the Project Options | Compiler dialog box.

-n

Description: Null compile for errors and warnings without running the program. The program will be downloaded to the target.

Factory Default: Program is run.

GUI Equivalent: Select Compile | Compile or use the keyboard shortcut <F5>.

-r

Description: Use compile mode: Compile to attached target.

Factory Default: Compile mode: Compile to attached target.

GUI Equivalent: Run program (F9)

-rb+

Description: Include BIOS when compiling to a file.

Factory Default: BIOS is included if compiling to a file.

GUI Equivalent: This is an advanced setting, viewable by clicking on the “Advanced” radio button at the bottom of the Project Options | Compiler dialog box. Check “Include BIOS.”

-rb-

Description: Do not include BIOS when compiling to a file.

Factory Default: BIOS is included if compiling to a file.

GUI Equivalent: This is an advanced setting, viewable by clicking on the “Advanced” radio button at the bottom of the Project Options | Compiler dialog box. Uncheck “Include BIOS.”

-rd+

Description: Include debug code when compiling to a file.

Factory Default: RST 28 instructions are included

GUI Equivalent: This is an advanced setting, viewable by clicking on the "Advanced" radio button at the bottom of the Project Options | Compiler dialog box. Check "Include RST 28 instructions."

-rd-

Description: Do not include debug code when compiling to a file. This option is ignored if not compiling to a file.

Factory Default: RST 28 instructions are included.

GUI Equivalent: This is an advanced setting, viewable by clicking on the "Advanced" radio button at the bottom of the Project Options | Compiler dialog box. Uncheck "Include RST 28 instructions."

-ri+

Description: Enable runtime checking of array indices.

Factory Default: Runtime checking of array indices is performed.

GUI Equivalent: Check "Array Indices" in the Project Options | Compiler dialog box.

-ri-

Description: Disable runtime checking of array indices.

Factory Default: Runtime checking of array indices is performed.

GUI Equivalent: Uncheck "Array Indices" in the Project Options | Compiler dialog box.

-rp+

Description: Enable runtime checking of pointers.

Factory Default: Runtime checking of pointers is performed.

GUI Equivalent: Check "Pointers" in the Project Options | Compiler dialog box.

-rp-

Description: Disable runtime checking of pointers.

Factory Default: Runtime checking of pointers is performed.

GUI Equivalent: Uncheck “Pointers” in the Project Options | Compiler dialog box.

-rw+

Description: Restrict watch expressions—may save root code space.

Factory Default: Allow any expressions in watch expressions.

GUI Equivalent: This is an advanced setting, viewable by clicking on the “Advanced” radio button at the bottom of the Project Options | Compiler dialog box. Check “Restrict watch expressions . . .”

-rw-

Description: Don’t restrict watch expressions.

Factory Default: Allow any expressions in watch expressions.

GUI Equivalent: This is an advanced setting, viewable by clicking on the “Advanced” radio button at the bottom of the Project Options | Compiler dialog box. Check “Allow any expressions in watch expressions”

-sp

Description: Optimize code generation for speed.

Factory Default: Optimize for speed.

GUI Equivalent: Choose “Speed” in the Project Options | Compiler dialog box.

-sz

Description: Optimize code generation for size.

Factory Default: Optimize for speed.

GUI Equivalent: Choose “Size” in the Project Options | Compiler dialog box.

-td+

Description: Enable type demotion checking.

Factory Default: Type demotion checking is performed.

GUI Equivalent: Check “Demotion” in the Project Options | Compiler dialog box.

-td-

Description: Disable type demotion checking.

Factory Default: Type demotion checking is performed.

GUI Equivalent: Uncheck “Demotion” in the Project Options | Compiler dialog box. .

-tp+

Description: Enable type checking of pointers.

Factory Default: Type checking of pointers is performed.

GUI Equivalent: Check “Pointer” in the Project Options | Compiler dialog box.

-tp-

Description: Disable type checking of pointers.

Factory Default: Type checking of pointers is performed.

GUI Equivalent: Uncheck “Pointer” in the Project Options | Compiler dialog box.

-tt+

Description: Enable type checking of prototypes.

Factory Default: Type checking of prototypes is performed.

GUI Equivalent: Check “Prototype” in the Project Options | Compiler dialog box.

-tt-

Description: Disable type checking of prototypes.

Factory Default: Type checking of prototypes is performed.

GUI Equivalent: Uncheck “Prototype” in the Project Options | Compiler dialog box..

-vp+

Description: Verify the processor by enabling a DSR check. This should be disabled if a check of the DSR line is incompatible on your system for any reason.

Factory Default: Processor verification is enabled.

GUI Equivalent: Check "Enable Processor verification" in the Project Options | Communications dialog box.

-vp-

Description: Assume a valid processor is connected.

Factory Default: Processor verification is enabled.

GUI Equivalent: Uncheck "Enable Processor verification" in the Project Options | Communications dialog box.

-wa

Description: Report all warnings.

Factory Default: All warnings reported.

GUI Equivalent: Select "All" under "Warning Reports" in the Project Options | Compiler dialog box.

-wn

Description: Report no warnings.

Factory Default: All warnings reported.

GUI Equivalent: Select "None" under "Warning Reports" in the Project Options | Compiler dialog box.

-ws

Description: Report only serious warnings.

Factory Default: All warnings reported.

GUI Equivalent: Select "Serious Only" under "Warning Reports" in the Project Options | Compiler dialog box.

15.4.2 Switches Requiring a Parameter

-bf BIOSFilePathname

Description: Compile using a BIOS file found in `BIOSFilePathname`.

Factory Default: `\Bios\RabbitBios.c`

GUI Equivalent: This is an advanced setting, viewable by clicking on the "Advanced" radio button at the bottom of the Project Options | Compiler dialog box. Check the box under "User Defined BIOS File" and then fill in the pathname for the new BIOS file.

Example: `dccl_cmp myProgram.c -bf MyPath\MyBIOS.lib`

-clf ColdLoaderFilePathname

Description: Compile using cold loader file found in `ColdLoaderFilePathname`.

Factory Default: `\Bios\ColdLoad.bin`

GUI Equivalent: None.

Example: `dccl_cmp myProgram.c -clf MyPath\MyColdloader.bin`

-d MacroDefinition

Description: Define macros and optionally equate to values. The following rules apply and are shown here with examples and equivalent `#define` form:

Separate macros with semicolons.

```
dccl_cmp myProgram.c -d DEF1;DEF2
#define DEF1
#define DEF2
```

A defined macro may be equated to text by separating the defined macro from the text with an equal sign (=).

```
dccl_cmp myProgram.c -d DEF1=20;DEF2
#define DEF1 20
#define DEF2
```

Macro definitions enclosed in quotation marks will be interpreted as a single command line parameter.

```
dccl_cmp myProgram.c -d “DEF1=text with spaces;DEF2”
#define DEF1 text with spaces
#define DEF2
```

A backslash preceding a character will be kept except for semicolon, quote and backslash, which keep only the character following the backslash. An escaped semicolon will not be interpreted as a macro separator and an escaped quote will not be interpreted as the quote defining the end of a command line parameter of text.

```
dccl_cmp myProgram.c -d DEF1=statement\;;ESCQUOTE=\\\”
#define DEF1 statement;
#define ESCQUOTE \”
dccl_cmp myProg.c -d “FSTR = \”Temp = %6.2F DEGREES C\n\””
#define FSTR “Temp = %6.2f degrees C\n”
```

Factory Default: None.

GUI Equivalent: Select the Defines tab from Project Options.

-d- MacroToUndefine

Description: Undefines a macro that might have been defined in the project file. If a macro is defined in the project file read by the command line compiler and the same macro name is redefined on the command line, the command line definition will generate a warning. A macro previously defined must be undefined with the **-d-** switch before redefining it. Undefining a macro that has not been defined has no consequence and so is always safe although possibly unnecessary. In the example, all compilation settings are taken from the project file specified except that now the macro `MAXCHARS` was first undefined before being redefined.

Factory Default: None.

GUI Equivalent: None.

Example:

```
dccl_cmp myProgram.c -pf myproject -d- MAXCHARS -d
  MAXCHARS=512
```

-eto EthernetResponseTimeout

Description: Time in milliseconds Dynamic C waits for a response from the target on any retry while trying to establish Ethernet communication.

Factory Default: 8000 milliseconds.

GUI Equivalent: None.

Example:

```
dccl_cmp myProgram.c -eto 6000
```

-i InputsFilePathname

Description: Execute a program that requires user input by supplying the input in a text file. Each input required should be entered into the text file exactly as it would be when entered into the Stdio Window in `dcwd.exe`. Extra input is ignored and missing input causes `dccl_cmp` to wait for keyboard input at the command line.

Factory Default: None.

GUI Equivalent: Using `-i` is like entering inputs into the Stdio Window.

Example

```
dccl_cmp myProgram.c -i MyInputs.txt
```

-lf LibrariesFilePathname

Description: Compile using a file found in LibrariesFilePathname which lists all libraries to be made available to your programs.

Factory Default: Lib.dir.

GUI Equivalent: This is an advanced setting, viewable by clicking on the “Advanced” radio button at the bottom of the Project Options | Compiler dialog box. Check the box under “User Defined Lib Directory File” and then fill in the pathname for the new Lib.dir.

Example
```
dccl_cmp myProgram.c -lf MyPath\MyLibs.txt
```

-ne maxNumberOfErrors

Description: Change the maximum number of errors reported.

Factory Default: A maximum of 10 errors are reported.

GUI Equivalent: Enter the maximum number of errors to report under “Max Shown” in the Project Options | Compiler dialog box.

Example: Allows up to 25 errors to be reported:
```
dccl_cmp myProgram.c -ne 25
```

-nw maxNumberOfWarnings

Description: Change the maximum number of warnings reported.

Factory Default: A maximum of 10 warnings are reported.

GUI Equivalent: Enter the maximum number of warnings to report under “Max Shown” in the Project Options | Compiler dialog box.

Example: Allows up to 50 warnings to be reported:
```
dccl_cmp myProgram.c -nw 50
```

-o OutputFilePathname

Description: Write header information (if specified with `-h`) and all program errors, warnings and outputs to a text file. If the text file does not exist it will be created, otherwise it will be overwritten.

Factory Default: None.

GUI Equivalent: Go to Option | Environment Options and select the Debug Windows tab. Under “Specific Preferences” select “Stdio” and check “Log to File” under “Options.”

Example

```
dccl_cmp myProgram.c -o MyOutput.txt
dccl_cmp myProgram.c -o MyOutput.txt -h
dccl_cmp myProgram.c -h -o MyOutput.txt
```

-oa OutputFilePathname

Description: Append header information (if specified with `-h`) and all program errors, warnings and outputs to a text file. If the text file does not exist it will be created, otherwise it will be appended.

Factory Default: None.

GUI Equivalent: Go to Option | Environment Options and select the Debug Windows tab. Under “Specific Preferences” select “Stdio” and check “Log to File” under “Options,” then check “Append” and specify the filename.

Example

```
dccl_cmp myProgram.c -oa MyOutput.txt
```

-pbf PilotBIOSFilePathname

Description: Compile using a pilot BIOS found in `PilotBIOSFilePathname`.

Factory Default: `\Bios\Pilot.bin`

GUI Equivalent: None.

Example:

```
dccl_cmp myProgram.c -pbf MyPath\MyPilot.bin
```

-pf projectFilePathname

Description: Specify a project file to read before the command line switches are read. The environment settings are taken from the project file specified with **-pf**, or `default.dcp` if no other project file is specified. Any switches on the command line, regardless of their position relative to the **-pf** switch, will override the settings from the project file.

Factory Default: The project file `default.dcp`.

GUI Equivalent: Select File | Project | Open...

Example

```
dccl_cmp myProgram.c -ne 25 -pf myProject.dcp
dccl_cmp myProgram.c -ne 25 -pf myProject
```

Note: The project file extension, `.dcp`, may be omitted.

-pw TCPPassPhrase

Description: Enter the passphrase required for your TCP/IP connection. If no passphrase is required this option need not be used.

Factory Default: No passphrase.

GUI Equivalent: Enter the passphrase required at the dialog prompt when compiling over a TCP/IP connection

Example:

```
dccl_cmp myProgram.c -pw “My passphrase”
```

-ret Retries

Description: The number of times Dynamic C attempts to establish communication if the given timeout period expires.

Factory Default: 3

GUI Equivalent: None.

Example:

```
dccl_cmp myProgram.c -ret 5
```

-rf RTIFilePathname

Description: Compile to a .bin file using targetless compilation parameters found in RTIFilePathname. The resulting compiled file will have the same pathname as the source (`.c`) file being compiled, but with a `.bin` extension.

Factory Default: None.

GUI Equivalent:

Example:

```
dccl_cmp myProgram.c -rf MyTCparameters.rti
dccl_cmp myProgram.c -rf "My Long Pathname\MyTCpa-
  rameters.rti"
```

-rti BoardID:CpuID:CrystalSpeed:RAMSize:FlashSize

Description: Compile to a `.bin` file using parameters defined in a colon separated format of BoardID:CpuID:CrystalSpeed:RAMSize:FlashSize. The resulting compiled file will have the same pathname as the source (`.c`) file being compiled, but with a `.bin` extension.

BoardID - Hex integer

CpuID - 2000r# or 3000r# where # is the revision number of the CPU.

2000r0: corresponds to IQ2T[a]
2000r1: corresponds to IQ3T
2000r2: corresponds to IQ4T
2000r3: corresponds to IQ5T
3000r0: corresponds to IL1T or IZ1T
3000r1: corresponds to IL2T

For backward compatibility, we also support:

2000: corresponds to IQ2T
3000: corresponds to IL1T or IZ1T

CrystalSpeed - Base frequency, decimal floating point, in MHz

RAMSize - Decimal, in KBytes

FlashSize - Primary flash, decimal, in KBytes.

Factory Default: None.

GUI Equivalent: Select Options | Project Options | Targetless | Board Selection and choose a board from the list; then select Compile | Compile to .bin File | Compile to Flash

Example:

```
dccl_cmp myProgram.c -rti
  0x0700:2000r3:11.0592:512:256
```

a. IQ*, IL* and IZ* are explained on page 232.

-s Port:Baud:Stopbits

Description: Use serial transmission with parameters defined in a colon separated format of Port:Baud:Stopbits:BackgroundTx.

Port: 1, 2, 3, 4, 5, 6, 7, 8

Baud: 110, 150, 300, 600, 1200, 2400, 4800, 9600, 12800, 14400, 19200, 28800, 38400, 57600, 115200, 128000, 230400, 256000

Stopbits: 1, 2

Include all serial parameters in the prescribed format even if only one is being changed.

Factory Default: 1:115200:1:0

GUI Equivalent: Select the Communications tab of Project Options. Select the “Use Serial Connection” radio button.

Example: Changing port from default of 1 to 2:

```
dccl_cmp myProgram.c -s 2:115200:1:0
```

-sto SerialResponseTimeout

Description: Time in milliseconds Dynamic C waits for a response from the target on any retry while trying to establish serial communication.

Factory Default: 300 ms.

GUI Equivalent: None.

Example:

```
dccl_cmp myProgram.c -sto 400
```

-t NetAddress:TcpName:TcpPort

Description: Use TCP with parameters defined in a contiguous colon separated format of NetAddress:TcpName:TcpPort. Include all parameters even if only one is being changed.

netAddress: n.n.n.n

tcpName: Text name of TCP port

tcpPort: decimal number of TCP port

Factory Default: None.

GUI Equivalent: Select the Communications tab of Project Options. Select the “Use TCP/IP Connection” radio button.

Example:

```
dccl_cmp myProgram.c -t 10.10.6.138:TCPName:4244
```

15.5 Examples

The following examples illustrate using multiple command line switches at the same time. If the switches on the command line are contradictory, such as `-mr` and `-mf`, the last switch (read left to right) will be used.

15.5.1 Example 1

In this example, all current settings of `default.dcp` are used for the compile.

```
dccl_cmp samples\timerb\timerb.c
```

15.5.2 Example 2

In this example, all settings of `myproject.dcp` are used, except `timer_b.c` is compiled to `timer_b.bin` instead of to the target and warnings or errors are written to `myouputs.txt`.

```
dccl_cmp samples\timerb\timer_b.c -o myoutputs.txt -b -pf
  myproject
```

15.5.3 Example 3

These examples will compile and run **`myProgram.c`** with the current settings in `default.dcp` but using different defines, displaying up to 50 warnings and capture all output to one file with a header for each run.

```
dccl_cmp myProgram.c -d MAXCOUNT=99 -nw 50 -h -o myOutput.txt
dccl_cmp myProgram.c -d MAXCOUNT=15 -nw 50 -h -oa myOutput.txt
dccl_cmp myProgram.c -d MAXCOUNT=15 -d DEF1 -nw 50 -h -oa
  myOutput.txt
```

The first run could have used the `-oa` option if `myOutput.txt` were known to not initially exist. `myProgram.c` presumably uses a constant `MAXCOUNT` and contains one or more compiler directives that react to whether or not `DEF1` is defined.

16. Project Files

In Dynamic C, a project is an environment that consists of opened source files, a BIOS file, available libraries, and the conditions under which the source files will be compiled. Projects allow different compilation environments to be separately maintained.

16.1 Project File Names

A project maintains a compilation environment in a file with the extension `.dcp`.

16.1.1 Factory.dcp

The environment originally shipped from the factory is kept in a project file named `factory.dcp`. If Dynamic C cannot find this file, it will be recreated automatically in the Dynamic C exe path. The factory project can be opened at any time and the environment changed and saved to another project name, but `factory.dcp` will not be changed by Dynamic C.

16.1.2 Default.dcp

This default project file is originally a copy of `factory.dcp` and will be automatically recreated as such in the exe path if it cannot be found when Dynamic C opens. The default project will automatically become the active project with **File | Project... | Close**.

The default project is special in that the command line compiler will use it for default values unless another project file is specified with the **-pf** switch, in which case the settings from the indicated project will be used.

Please see chapter 15, “Command Line Interface” starting on page 241 for more details on using the command line compiler.

16.1.3 Active Project

Whenever a project is selected, the current project related data is saved to the closing project file, the new project settings become active, and the (possibly new) BIOS will automatically be recompiled prior to compiling a source file in the new environment.

The active project can be `factory.dcp`, `default.dcp` or any project you create with **File | Project... | Save As...** When Dynamic C opens, it retrieves the last used project, or the default project if being opened for the first time or if the last used project cannot be found.

If a project is closed with the **File | Projects... | Close** menu option, the default project, `default.dcp`, becomes the active project.

The active project file name, without path or extension, is always shown in the leftmost panel of the status bar at the bottom of the Dynamic C main window and is prepended to the Dynamic C version in the title bar except when the active project is the default project.

Changes made to the compilation environment of Dynamic C are automatically updated to the active project, unless the active project is `factory.dcp`.

16.2 Updating a Project File

Unless the active project is `factory.dcp`, changes made in the Project Options dialog will cause the active project file to be updated immediately:

Opening or closing files will not immediately update the active project file. The project file state of the recently used files appearing at the bottom of the **File** menu selection and any opened files in edit windows will only by updated when the project closes or when **File | Projects... | Save** is selected. The Message, Assembly, Memory Dump, Registers and Stack debug windows are not edit windows and will not be saved in the project file if you exit Dynamic C while debugging.

16.3 Menu Selections

The menu selections for project files are available in the **File** menu. The choices are the familiar ones: **Create...**, **Open...**, **Save**, **Save As...** and **Close**.

Choosing **File | Project | Open...** will bring up a dialog box to select an existing project filename to become the active project. The environment of the previous project is saved to its project file before it is replaced (unless the previous project is `factory.dcp`). The BIOS will automatically be recompiled prior to the compilation of a source file within the new environment, which may have a different library directory file and/or a different BIOS file.

Choosing **File | Project... | Save** will save the state of the environment to the active project file, including the state of the recently used filelist and any files open in edit windows. This selection is greyed out if the active project is `factory.dcp`. This option is of limited use since any project changes will be updated immediately to the file and the state of the recently used filelist and open edit windows will be updated when the project is closed for any reason.

Choosing **File | Project... | Save as...** will bring up a dialog box to select a project file name. The file will be created or, if it exists, it will be overwritten with the current environment settings. This environment will also be saved to the active project file before it is closed and its copy (the newly created or overwritten project file) will become active.

Choosing **File | Project... | Close** first saves the environment to the active project file (unless the active project is `factory.dcp`) and then loads the Dynamic C default project, `default.dcp`, as the active project. As with **Open...**, the BIOS will automatically be recompiled prior to the compilation of a source file within the new environment. The new environment may have a different library directory file and/or a different BIOS file.

16.4 Command Line Usage

When using the command line compiler, `dccl_cmp.exe`, a project file is always read. The default project, `default.dcp`, is used automatically unless the project file switch, **-pf**, specifies another project file to use. The project settings are read by the command line compiler first even if a **-pf** switch comes after the use of other switches, and then all other switches used in the command line are read, which may modify any of the settings specified by the project file.

The default behavior given for each switch in the command line documentation is with reference to the `factory.dcp` settings, so the user must be aware of the default state the command line compiler will actually use. The settings of `default.dcp` can be shown by entering `dccl_cmp` alone on the command line. The defaults for any other project file can be shown by following `dccl_cmp` by a the project file switch without a source file.

```
dccl_cmp
```

shows the current state of all `default.dcp` settings

```
dccl_cmp -pf myProject
```

shows the current state of all `myProject.dcp` settings

```
dccl_cmp myProgram.c -ne 25 -pf myProject
```

reads `myProject.dcp` then compiles and runs `myProgram.c` but with 25 errors maximum shown.

The command line compiler, unlike Dynamic C, never updates the project file it uses. Any changes desired to a project file to be used by the command line compiler can be made within Dynamic C or changed by hand with an editor.

Making changes by hand should be done with caution, using an editor which does not introduce carriage returns or line feeds with wordwrap, which may be a problem if the global defines or any file pathnames are lengthy strings. Be careful when changing by hand not to change any of the section names in brackets or any of the key phrases up to and including the '='.

If a macro is defined on the command line with the **-d** switch, any value that may have been defined within the project file used will be overwritten without warning or error. Undefining a macro with the **-d-** switch has no consequence if it was not previously defined.

17. Hints and Tips

This chapter offers hints on how to speed up an application and how to store persistent data at run time.

17.1 Efficiency

There are a number of methods that can be used to reduce the size of a program, or to increase its speed. Let's look at the events that occur when a program enters a function.

- The function saves IX on the stack and makes IX the stack frame reference pointer (if the program is in the `useix` mode).
- The function creates stack space for `auto` variables.
- The function sets up stack corruption checks if stack checking is enabled (on).
- The program notifies Dynamic C of the entry to the function so that single stepping modes can be resolved (if in debug mode).

The last two consume significant execution time and are eliminated when stack checking is disabled or if the debug mode is off.

17.1.1 Nodebug Keyword

When the PC is connected to a target controller with Dynamic C running, the normal code and debugging features are enabled. Dynamic C places an `RST 28H` instruction at the beginning of each C statement to provide locations for breakpoints. This allows the programmer to single step through the program or to set breakpoints. (It is possible to single step through assembly code at any time.) During debugging there is additional overhead for entry and exit bookkeeping, and for checking array bounds, stack corruption, and pointer stores. These "jumps" to the debugger consume one byte of code space and also require execution time for each statement.

At some point, the Dynamic C program will be debugged and can run on the target controller without the Dynamic C debugger. This saves on overhead when the program is executing. The `nodebug` keyword is used in the function declaration to remove the extra debugging instructions and checks.

```
nodebug int myfunc( int x, int z ){
    ...
}
```

If programs are executing on the target controller with the debugging instructions present, but without Dynamic C attached, the call to the function that handles RST 28H instructions in the vector table will be replaced by a simple ret instruction for Rabbit 2000 based targets. For Rabbit 3000 based targets, the RST 28H instruction is treated as a NOP by the processor when in debug mode. The target controller will work, but its performance will not be as good as when the nodebug keyword is used.

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 directive `#nodebug` anywhere within the program to enable `nodebug` for all statements following the directive. The `#debug` directive has the opposite effect.

Assembly code blocks are nodebug by default, even when they occur inside C functions that are marked debug, therefore using the `nodebug` keyword with the `#asm` directive is usually unnecessary.

17.1.2 In-line I/O

The built-in I/O functions (`WrPortI()`, `RdPortI()`, `BitWrPortI()` and `BitRdPortI()`) can be generated as efficient in-line code instead of function calls. All arguments must be constant. A normal function call is generated if the I/O function is called with any non-constant arguments. To enable in-line code generation for the built-in I/O functions check the option “Inline builtin I/O functions” in the Compiler dialog, which is accessible by clicking the Compiler tab in the Project Options dialog.

17.2 Run-time Storage of Data

Data that will never change in a program can be put in flash by initializing it in the declarations. The compiler will put this data in flash. See the description of the `const`, `xdata`, and `xstring` keywords for more information.

If data must be stored at run-time and persist between power cycles, there are several ways to do this using Dynamic C functions:

- **User Block -** Recommended method for storing non-file data. This is where calibration constants for boards with analog I/O are stored in the factory. Space here is limited to as small as `8K-sizeof(SysIDBlock)` bytes, or less if there are calibration constants.
- **Flash File System -** The file system is best for storing data that must be organized into files, or data that won’t fit in the User block. It is best used on a second flash chip. It is not possible to use a second flash for both extra program code that doesn’t fit into the first flash, and the file system. The macro `USE_2NDFLASH_CODE` must be uncommented in the BIOS to allow programs to grow into the second flash; this precludes the use of the file system.
- **WriteFlash2 -** This function is provided for writing arbitrary amounts of data directly to arbitrary addresses in the second flash.
- **Battery-Backed RAM -** Storing data here is as easy as assigning values to global variables or local static variables. The file system can also be configured to use RAM. The important question is, what will you do when your battery runs out?

17.2.1 User Block

The User block is an area near the top of flash reserved for run-time storage of persistent data and calibration constants. The size of the User block can be read in the global structure member `SysIDBlock.userBlockSize`. The functions `readUserBlock()` and `writeUserBlock()` are used to access the User block. These function take an offset into the block as a parameter. The highest offset available to the user in the User block will be

```
SysIDBlock.userBlockSize-1
```

if there are no calibration constants, or

```
DAC_CALIB_ADDR-1
```

if there are.

See the *Rabbit 3000 Designer's Handbook* or the *Rabbit 2000 Designer's Handbook* for more details about the User block.

17.2.2 Flash File System

For a complete discussion of the file system, please see "The Flash File System" on page 109.

17.2.3 WriteFlash2

See the *Dynamic C Function Reference Manual* for a complete description.

> **NOTE:** There is a `WriteFlash()` function available for writing to the first flash, but its use is highly discouraged for reasons of forward source and binary compatibility should flash sector configuration change drastically in a product. See Technical Notes 216 and 217 for more information on flash compatibility issues.

17.2.4 Battery-Backed RAM

Static variables and global variables will always be located at the same addresses between power cycles and can only change locations via recompilation. The file system can be configured to use RAM also. While there may be applications where storing persistent data in RAM is acceptable, for example a data logger where the data gets retrieved and the battery checked periodically, keep in mind that a programming error such as an uninitialized pointer could cause RAM data to be corrupted.

`xalloc()` will allocate blocks of RAM in extended memory. It will allocate the blocks consistently from the same physical address if done at the beginning of the program and the program is not recompiled.

17.3 Root Memory Reduction Tips

Customers with programs that are near the limits of root code and/or root data space usage will be interested in these tips for saving root space. For more help, see Technical Note TN238 “Rabbit Memory Usage Tips.” This document is available on our website: `www.zworld.com`, and also by choosing Online Documentation from within the Help menu of Dynamic C.

17.3.1 Increasing Root Code Space

Increasing the available amount of root code space may be done in the following ways:

- **Enable Separate Instruction and Data Space**

  A hardware memory management scheme that uses address line inversion to double the amount of logical address space in the base and data segments is enabled on the Compiler tab of the Options | Project Options dialog. Enabling separate I&D space doubles the amount of root cod and root data available for an application program.

- **Use #memmap xmem**

  This will cause C functions that are not explicitly declared as “root” to be placed in xmem. Note that the only reason to locate a C function in root is because it modifies the XPC register (in embedded assembly code), or it is an ISR. The only performance difference in running code in xmem is in getting there and returning. It takes a total of 12 additional machine cycles because of the differences between `call/lcall`, and `ret/lret`.

- **Increase DATAORG**

  Root code space can be increased by increasing `DATAORG` in `RabbitBios.c` in increments of 0x1000. `DATAORG` is the beginning logical address for the data segment. The default is 0x3000 when separate I&D space is enabled, and 0x6000 otherwise. It can be changed to as high as 0xB000.

  Be aware that increasing `DATAORG` reduces the amount of root data space.

- **Reduce usage of root constants and string literals**

  Shortening literal strings and reusing them will save root space. The compiler automatically reuses identical string literals.

  These two statements :

  ```
  printf ("This is a literal string");
  sprintf (buf, "This is a literal string");
  ```

  will share the same literal string space whereas:

  ```
  sprintf (buf, "this is a literal string");
  ```

  will use its own space since the string is different.

- **Use xdata to declare large tables of initialized data**

  If you have large tables of initialized data, consider using the keyword `xdata` to declare them. The disadvantage is that data cannot be accessed directly with pointers. The function `xmem2root()` allows xdata to be copied to a root buffer when needed.

```
// This uses root code space
const int root_tbl[8]={300,301,302,103,304,305,306,307};
// This does not
xdata xdata_table {300,301,302,103,304,305,306,307};
main(){
    // this only uses temporary stack space
    auto int table[8];
    xmem2root(table, xdata_table, 16);
    // now the xmem data can be accessed via a 16 bit pointer into the table
}
```

  Both methods, `const` and `xdata`, create initialized data in flash at compile time, so the data cannot be rewritten directly.

- **Use xstring to declare a table of strings**

  The keyword `xstring` declares a table of strings in extended flash memory. The disadvantage is that the strings cannot be accessed directly with pointers, since the table entries are 20-bit physical addresses. As illustrated above, the function `xmem2root()` may be used to store the table in temporary stack space.

```
// This uses root code space
const char * name[] = {"string_1", . . . "string_n"};

// This does not
xstring name {"string_1", . . . "string_n"};
```

  Both methods, `const` and `xstring`, create initialized data in flash at compile time, so the data cannot be rewritten directly.

- **Turn off selected debugging features**

  Watch expressions, breakpoints, and single stepping can be selectively disabled on the Debugger tab of Project Options to save some root code space.

- **Place assembly language code into xmem**

  Pure assembly language code functions can go into xmem.

```
#asm
foo_root::
   [some instructions]
   ret
#endasm
```

  The same function in xmem:

```
#asm xmem
foo_xmem::
   [some instructions]
   lret      ; use lret instead of ret
#endasm
```

  The correct calls are `call foo_root` and `lcall foo_xmem`. If the assembly function modifies the XPC register with

```
LD XPC, A
```

  it should not be placed in xmem. If it accesses data on the stack directly, the data will be one byte away from where it would be with a root function because `lcall` pushes the value of XPC onto the stack.

17.3.2 Increasing Root Data Space

Increasing the available amount of root data space may be done in the following ways:

- **Enable Separate Instruction and Data Space**

  A hardware memory management scheme that uses address line inversion to double the amount of logical address space in the base and data segments is enabled on the Compiler tab of the Options | Project Options dialog. Enabling separate I&D space doubles the amount of root code and root data available for an application program.

- **Decrease DATAORG**

  Root data space can be increased by decreasing `DATAORG` in `RabbitBios.c` in increments of 0x1000. At the time of this writing, RAM compiles should be done with no less than the default value of `DATAORG` when separate I&D space is disabled. This restriction is to ensure that the pilot BIOS does not overwrite itself. The default is 0x6000.

  Be aware that decreasing `DATAORG` reduces the amount of root code space.

- **Use xmem for large RAM buffers**

  `xalloc()` can be used to allocate chunks of RAM in extended memory. The memory cannot be accessed by a 16 bit pointer, so using it can be more difficult. The functions `xmem2root()` and `root2xmem()` are available for moving from root to xmem and xmem to root. Large buffers used by Dynamic C libraries are already allocated from RAM in extended memory.

18. µC/OS-II

µC/OS-II is a simple, clean, efficient, easy-to-use real-time operating system that runs on the Rabbit microprocessor and is fully supported by the Dynamic C development environment. It is an add-on module that may be purchased from our website: www.zworld.com.

µC/OS-II is capable of intertask communication and synchronization via the use of semaphores, mailboxes, and queues. User-definable system hooks are supplied for added system and configuration control during task creation, task deletion, context switches, and time ticks. For more information on µC/OS-II, please refer to Jean J. Labrosse's book, *MicroC/OS-II, The Real-Time Kernel* (ISBN: 0-87930-543-6). The data structures (e.g., Event Control Block) referenced in the Dynamic C µC/OS-II function descriptions are fully explained in Labrosse's book. It can be purchased at the Z-World store, www.zworld.com/store/home.html, or at http://www.ucos-ii.com/.

The Rabbit version of µC/OS-II has the new features and API changes available in version 2.51 of µC/OS-II. The documentation for these changes is included with Dynamic C in `Samples/UCos-II`. The file `Newv251.pdf` contains all of the features added since version 2.00 and `Relv251.pdf` contains release notes for version 2.51.

18.1 Changes to µC/OS-II

To take full advantage of services provided by Dynamic C, minor changes have been made to µC/OS-II.

18.1.1 Ticks per Second

In most implementations of µC/OS-II, `OS_TICKS_PER_SEC` informs the operating system of the rate at which `OSTimeTick` is called; this macro is used as a constant to match the rate of the periodic interrupt. In µC/OS-II for the Rabbit, however, changing this macro will *change* the tick rate of the operating system set up during `OSInit`. Usually, a real-time operating system has a tick rate of 10 Hz to 100 Hz, or 10–100 ticks per second. Since the periodic interrupt on the Rabbit occurs at a rate of 2 kHz, it is recommended that the tick rate be a power of 2 (e.g., 16, 32, or 64). Keep in mind that the higher the tick rate, the more overhead the system will incur.

In the Rabbit version of µC/OS-II, the number of ticks per second defaults to 64. The actual number of ticks per second may be slightly different than the desired ticks per second if `TicksPerSec` does not evenly divide 2048.

Changing the default tick rate is done by simply defining `OS_TICKS_PER_SEC` to the desired tick rate before calling `OSInit()`. E.g. to change the tick rate to 32 ticks per second:

```
#define OS_TICKS_PER_SEC 32
...
OSInit();
...
OSStart();
```

18.1.2 Task Creation

In a µC/OS-II application, stacks are declared as static arrays, and the address of either the top or bottom (depending on the CPU) of the stack is passed to `OSTaskCreate`. In a Rabbit-based system, the Dynamic C development environment provides a superior stack allocation mechanism that µC/OS-II incorporates. Rather than declaring stacks as static arrays, the number of stacks of particular sizes are declared, and when a task is created using either `OSTaskCreate` or `OSTaskCreateExt`, only the size of the stack is passed, not the memory address. This mechanism allows a large number of stacks to be defined without using up root RAM.

There are five macros located in `ucos2.lib` that define the number of stacks needed of five different sizes. In order to have three 256 byte stacks, one 512 byte stack, two 1024 byte stacks, one 2048 byte stack, and no 4096 byte stacks, the following macro definitions would be used:

```
#define STACK_CNT_256       3      // number of 256 byte stacks
#define STACK_CNT_512       1      // number of 512 byte stacks
#define STACK_CNT_1K        2      // number of 1K stacks
#define STACK_CNT_2K        1      // number of 2K stacks
#define STACK_CNT_4K        0      // number of 4K stacks
```

These macros can be placed into each µC/OS-II application so that the number of each size stack can be customized based on the needs of the application. Suppose that an application needs 5 tasks, and each task has a consecutively larger stack. The macros and calls to `OSTaskCreate` would look as follows

```
#define STACK_CNT_256      2      // number of 256 byte stacks
#define STACK_CNT_512      2      // number of 512 byte stacks
#define STACK_CNT_1K       1      // number of 1K stacks
#define STACK_CNT_2K       1      // number of 2K stacks
#define STACK_CNT_4K       1      // number of 4K stacks

OSTaskCreate(task1, NULL, 256, 0);
OSTaskCreate(task2, NULL, 512, 1);
OSTaskCreate(task3, NULL, 1024, 2);
OSTaskCreate(task4, NULL, 2048, 3);
OSTaskCreate(task5, NULL, 4096, 4);
```

Note that `STACK_CNT_256` is set to 2 instead of 1. µC/OS-II always creates an idle task which runs when no other tasks are in the ready state. Note also that there are two 512 byte stacks instead of one. This is because the program is given a 512 byte stack. If the application utilizes the µC/OS-II statistics task, then the number of 512 byte stacks would have to be set to 3. (Statistic task creation can be enabled and disabled via the macro `OS_TASK_STAT_EN` which is located in `ucos2.lib`). If only 6 stacks were declared, one of the calls to `OSTaskCreate` would fail.

If an application uses OSTaskCreateExt, which enables stack checking and allows an extension of the Task Control Block, fewer parameters are needed in the Rabbit version of µC/OS-II. Using the macros in the example above, the tasks would be created as follows:

```
OSTaskCreateExt(task1, NULL, 0, 0, 256, NULL, OS_TASK_OPT_STK_CHK |
   OS_TASK_OPT_STK_CLR);
OSTaskCreateExt(task2, NULL, 1, 1, 512, NULL, OS_TASK_OPT_STK_CHK |
   OS_TASK_OPT_STK_CLR);
OSTaskCreateExt(task3, NULL, 2, 2, 1024, NULL, OS_TASK_OPT_STK_CHK |
   OS_TASK_OPT_STK_CLR);
OSTaskCreateExt(task4, NULL, 3, 3, 2048, NULL, OS_TASK_OPT_STK_CHK |
   OS_TASK_OPT_STK_CLR);
OSTaskCreateExt(task5, NULL, 4, 4, 4096, NULL, OS_TASK_OPT_STK_CHK |
   OS_TASK_OPT_STK_CLR);
```

18.1.3 Restrictions

At the time of this writing, µC/OS-II for Dynamic C is not compatible with the use of slice statements. Also, see the function description for OSTimeTickHook() for important information about preserving registers if that stub function is replaced by a user-defined function.

Due to Dynamic C's stack allocation scheme, special care should be used when posting messages to either a mailbox or a queue. A message is simply a void pointer, allowing the application to determine its meaning. Since tasks can have their stacks in different segments, auto pointers declared on the stack of the task posting the message should not be used since the pointer may be invalid in another task with a different stack segment.

18.2 Tasking Aware Interrupt Service Routines (TA-ISR)

Special care must be taken when writing an interrupt service routine (ISR) that will be used in conjunction with µC/OS-II so that µC/OS-II scheduling will be performed at the proper time.

18.2.1 Interrupt Priority Levels

µC/OS-II for the Rabbit reserves interrupt priority levels 2 and 3 for interrupts outside of the kernel. Since the kernel is unaware of interrupts above priority level 1, interrupt service routines for interrupts that occur at interrupt priority levels 2 and 3 should not be written to be tasking aware. Also, a µC/OS-II application should only disable interrupts by setting the interrupt priority level to 1, and should never raise the interrupt priority level above 1.

18.2.2 Possible ISR Scenarios

There are several different scenarios that must be considered when writing an ISR for use with µC/OS-II. Depending on the use of the ISR, it may or may not have to be written so that it is tasking aware. Consider the scenario in the Figure below. In this situation, the ISR for Interrupt X does not have to be tasking aware since it does not re-enable interrupts before completion and it does not post to a semaphore, mailbox, or queue.

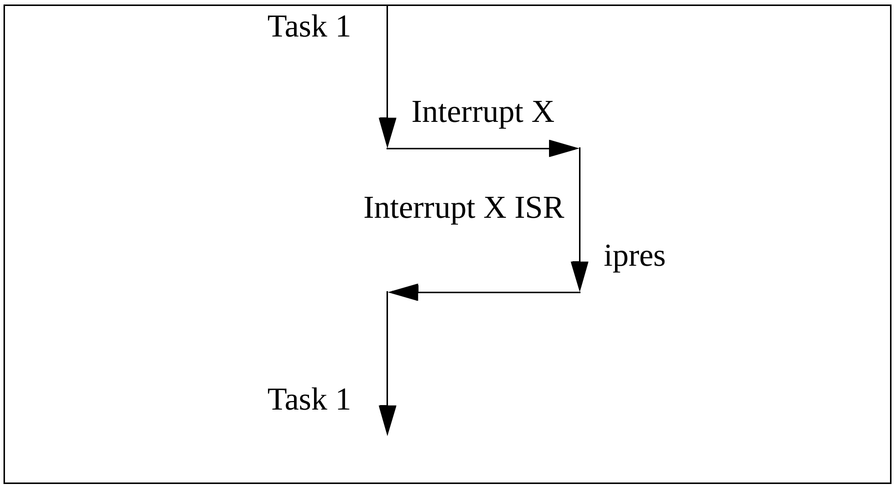


Figure 1. Type 1 ISR

If, however, an ISR needs to signal a task to the ready state, then the ISR must be tasking aware. In the example in the Figure below, the TA-ISR increments the interrupt nesting counter, does the work necessary for the ISR, readies a higher priority task, decrements the nesting count, and returns to the higher priority task.

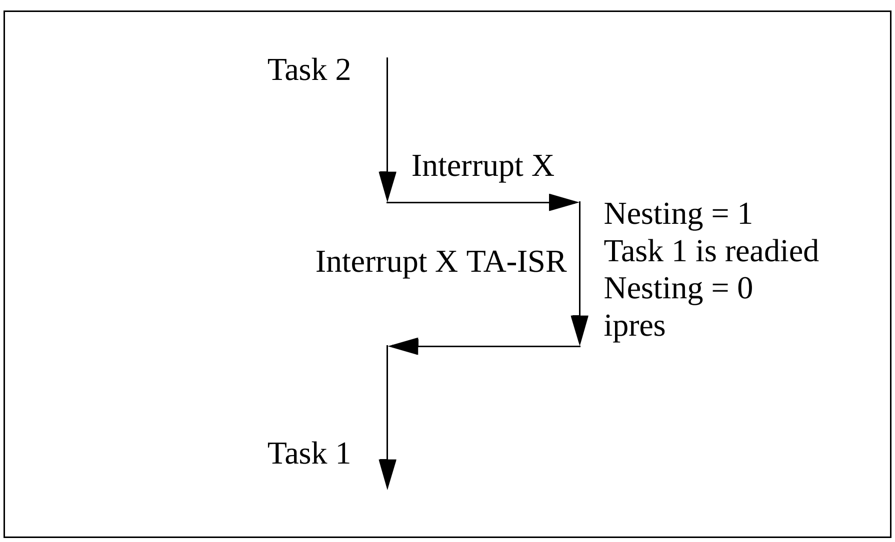


Figure 2. Type 2 ISR

It may seem as though the ISR in this Figure does not have to increment and decrement the nesting count. This is, however, very important. If the ISR for Interrupt X is called during an ISR that re-enables interrupts before completion, scheduling should not be performed when Interrupt X completes; scheduling should instead be deferred until the least nested ISR completes. The next Figure shows an example of this situation.

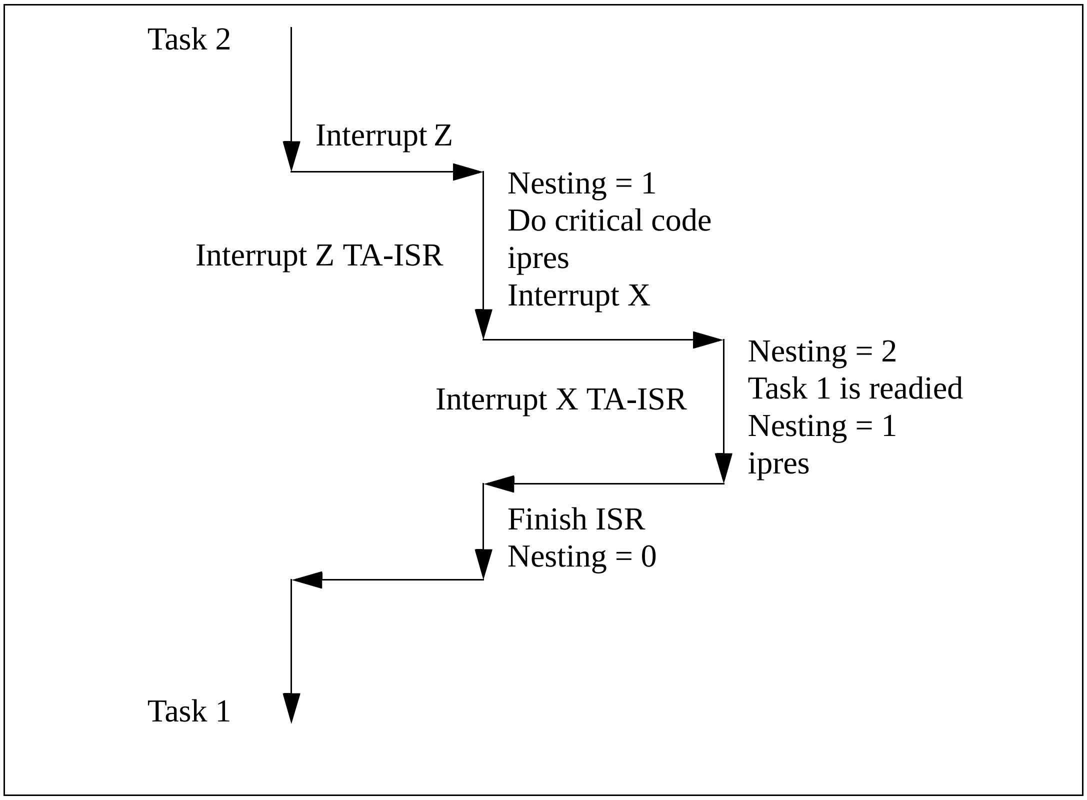


Figure 3. Type 2 ISR Nested Inside Type 3 ISR

As can be seen here, although the ISR for interrupt Z does not signal any tasks by posting to a semaphore, mailbox, or queue, it must increment and decrement the interrupt nesting count since it re-enables interrupts (`ipres`) prior to finishing all of its work.

18.2.3 General Layout of a TA-ISR

A TA-ISR is just like a standard ISR except that it does some extra checking and house-keeping. The following table summarizes when to use a TA-ISR.

Table 18-1. Use of TA-ISR

	µC/OS-II Application		
	Type 1[1]	**Type 2**[2]	**Type 3**[3]
TA-ISR Required?	No	Yes	Yes

1. Type 1—Leaves interrupts disabled and does not signal task to ready state
2. Type 2—Leaves interrupts disabled and signals task to ready state
3. Type 3—Reenables interrupts before completion

The following Figure shows the logical flow of a TA-ISR.

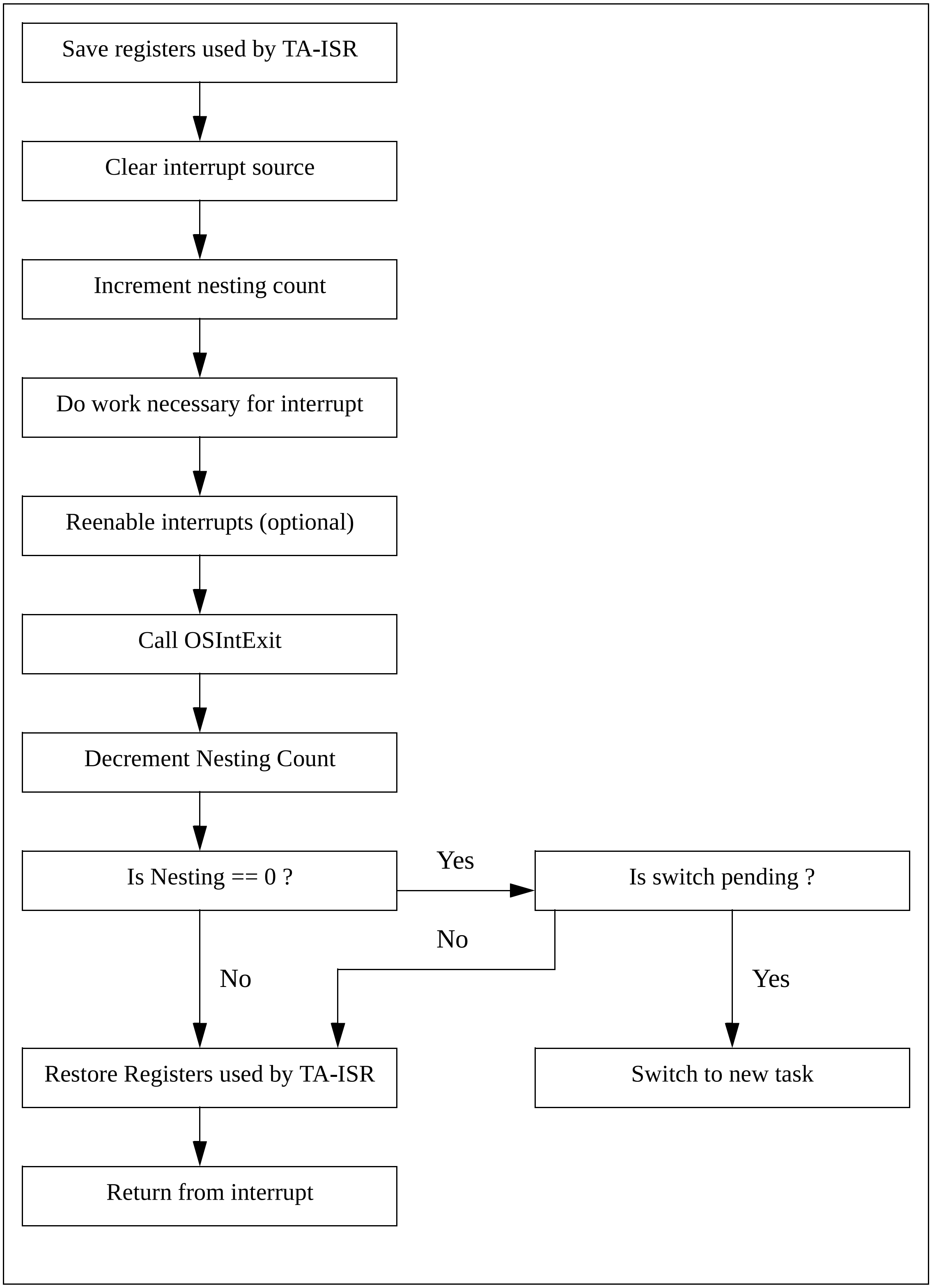


Figure 4. Logical Flow of a TA-ISR

18.2.3.1 Sample Code for a TA-ISR

Fortunately, the Rabbit BIOS and libraries provide all of the necessary flags to make TA-ISRs work. With the code found in `Listing 1`, minimal work is needed to make a TA-ISR function correctly with µC/OS-II. TA-ISRs allow µC/OS-II the ability to have ISRs that communicate with tasks as well as the ability to let ISRs nest, thereby reducing interrupt latency.

Just like a standard ISR, the first thing a TA-ISR does is to save the registers that it is going to use (1). Once the registers are saved, the interrupt source is cleared (2) and the nesting counter is incremented (3). Note that `bios_intnesting` is a global interrupt nesting counter provided in the Dynamic C libraries specifically for tracking the interrupt nesting level. If an `ipres` instruction is executed (4) other interrupts can occur before this ISR is completed, making it necessary for this ISR to be a TA-ISR. If it is possible for the ISR to execute before µC/OS-II has been fully initialized and started multi-tasking, a check should be made (5) to insure that µC/OS-II is in a known state, especially if the TA-ISR signals a task to the ready state (6). After the TA-ISR has done its necessary work (which may include making a higher priority task than is currently running ready to run), `OSIntExit` must be called (7). This µC/OS-II function determines the highest priority task ready to run, sets it as the currently running task, and sets the global flag `bios_swpend` if a context switch needs to take place. Interrupts are disabled since a context switch is treated as a critical section (8). If the TA-ISR decrements the nesting counter and the count does not go to zero, then the nesting level is saved in `bios_intnesting` (9), the registers used by the TA-ISR are restored, interrupts are re-enabled (if not already done in (4)), and the TA-ISR returns (12). However, if decrementing the nesting counter in (9) causes the counter to become zero, then `bios_swpend` must be checked to see if a context switch needs to occur (10). If a context switch is not pending, then the nesting level is set (9) and the TA-ISR exits (12). If a context switch is pending, then the remaining context of the previous task is saved and a long call, which insures that the `xpc` is saved and restored properly, is made to `bios_intexit` (11). `bios_intexit` is responsible for switching to the stack of the task that is now ready to run and executing a long call to switch to the new task. The remainder of (11) is executed when a previously preempted task is allowed to run again.

`Listing 1`

```
#asm
taskaware_isr::
   push   af                        ;push regs needed by isr      (1)
   push   hl                        ;clear interrupt source       (2)
   ld     hl,bios_intnesting        ;increase the nesting count   (3)
   inc    (hl)
   ; ipres (optional)                                             (4)
   ; do processing necessary for interrupt
   ld     a,(OSRunning)             ;MCOS multitasking yet?       (5)
   or     a
   jr     z,taisr_decnesting

   ; possibly signal task to become ready                         (6)
   call   OSIntExit                 ;sets bios_swpend if higher
                                    ; prio ready                  (7)
```

```
taisr_decnesting:
   push   ip                                              (8)
   ipset  1

   ld     hl,bios_intnesting       ; nesting counter == 1?
   dec    (hl)                                            (9)
   jr     nz,taisr_noswitch

   ld     a,(bios_swpend)          ; switch pending?      (10)
   or     a
   jr     z,taisr_noswitch

   push   de                                              (11)
   push   bc
   ex     af,af'
   push   af
   exx
   push   hl
   push   de
   push   bc
   push   iy

   lcall  bios_intexit

   pop    iy
   pop    bc
   pop    de
   pop    hl
   exx
   pop    af
   ex     af,af'
   pop    bc
   pop    de

taisr_noswitch:
   pop   ip

taisr_done:
   pop    hl                                              (12)
   pop    af
   ipres
   ret
#endasm
```

18.3 Library Reentrancy

When writing a µC/OS-II application, it is important to know which Dynamic C library functions are non-reentrant. If a function is non-reentrant, then only one task may access the function at a time, and access to the function should be controlled with a µC/OS-II semaphore. The following is a list of Dynamic C functions that are non-reentrant.

Library	Non-reentrant Functions
MATH.LIB	randg, randb, rand
RS232.LIB	All
RTCLOCK.LIB	write_rtc, tm_wr
STDIO.LIB	kbhit, getchar, gets, getswf, selectkey
STRING.LIB	atof[1], atoi[1], strtok
SYS.LIB	clockDoublerOn, clockDoublerOff, useMainOsc, useClockDivider, use32kHzOsc
VDRIVER.LIB	VdGetFreeWd, VdReleaseWd
XMEM.LIB	WriteFlash
JRIO.LIB	digOut, digOn, digOff, jrioInit, anaIn, anaOut, cof_anaIn
JR485.LIB	All

1. reentrant but sets the global `_xtoxErr` flag

The serial port functions (`RS232.LIB` functions) should be used in a restricted manner with µC/OS-II. Two tasks can use the same port as long as both are not reading, or both are not writing; i.e., one task can read from serial port X and another task can write to serial port X at the same time without conflict.

18.4 How to Get a µC/OS-II Application Running

µC/OS-II is a highly configureable, real-time operating system. It can be customized using as many or as few of the operating system's features as needed. This section outlines:

- The configuration constants used in µC/OS-II
- How to override the default configuration supplied in `UCOS2.LIB`
- The necessary steps to get an application running

It is assumed that the reader has a familiarity with µC/OS-II or has a µC/OS-II reference (*MicroC/OS-II, The Real-Time Kernel* by Jean J. Labrosse is highly recommended).

18.4.1 Default Configuration

µC/OS-II usually relies on the include file `os_cfg.h` to get values for the configuration constants. In the Dynamic C implementation of µC/OS-II, these constants, along with their default values, are in `os_cfg.lib`. A default stack configuration is also supplied in `os_cfg.lib`. µC/OS-II for the Rabbit uses a more intelligent stack allocation scheme than other µC/OS-II implementations to take better advantage of unused memory.

The default configuration allows up to 10 normally created application tasks running at 64 ticks per second. Each task has a 512-byte stack. There are 2 queues specified, and 10 events. An event is a queue, mailbox or semaphore. You can define any combination of these three for a total of 10. If you want more than 2 queues, however, you must change the default value of `OS_MAX_QS`.

Some of the default configuration constants are:

```
// Maximum number of events (semaphores, queues, mailboxes)
#define OS_MAX_EVENTS 10

// Maximum number of tasks (less stat and idle tasks)
#define OS_MAX_TASKS 10

// Maximum number of queues in system
#define OS_MAX_QS 2

// Maximum number of memory partitions
#define OS_MAX_MEM_PART 1

// Enable normal task creation
#define OS_TASK_CREATE_EN 1

// Disable extended task creation
#defineOS_TASK_CREATE_EXT_EN 0

// Disable task deletion
#define OS_TASK_DEL_EN 0

// Disable statistics task creation
#define OS_TASK_STAT_EN 0

// Enable queue usage
#define OS_Q_EN 1

// Disable memory manager
#define OS_MEM_EN 0

// Enable mailboxes
#define OS_MBOX_EN 1
```

```
// Enable semaphores
#define OS_SEM_EN 1
// number of ticks in one second
#define OS_TICKS_PER_SEC 64
// number of 256 byte stacks (idle task stack)
#define STACK_CNT_256 1
// number of 512-byte stacks (task stacks + initial program stack)
#define STACK_CNT_512 OS_MAX_TASKS+1
```

If a particular portion of µC/OS-II is disabled, the code for that portion will not be compiled, making the overall size of the operating system smaller. Take advantage of this feature by customizing µC/OS-II based on the needs of each application.

18.4.2 Custom Configuration

In order to customize µC/OS-II by enabling and disabling components of the operating system, simply redefine the configuration constants as necessary for the application.

```
#define OS_MAX_EVENTS          2
#define OS_MAX_TASKS          20
#define OS_MAX_QS              1
#define OS_MAX_MEM_PART       15
#define OS_TASK_STAT_EN        1
#define OS_Q_EN                0
#define OS_MEM_EN              1
#define OS_MBOX_EN             0
#define OS_TICKS_PER_SEC      64
```

If a custom stack configuration is needed also, define the necessary macros for the counts of the different stack sizes needed by the application.

```
#define STACK_CNT_256 1   // idle task stack
#define STACK_CNT_512 2   // initial program + stat task stack
#define STACK_CNT_1K 10   // task stacks
#define STACK_CNT_2K 10   // number of 2K stacks
```

In the application code, follow the µC/OS-II and stack configuration constants with a `#use "ucos2.lib"` statement. This ensures that the definitions supplied outside of the library are used, rather than the defaults in the library.

This configuration uses 20 tasks, two semaphores, up to 15 memory partitions that the memory manager will control, and makes use of the statistics task. Note that the configuration constants for task creation, task deletion, and semaphores are not defined, as the library defaults will suffice. Also note that 10 of the application tasks will each have a 1024 byte stack, 10 will each have a 2048 byte stack, and an extra stack is declared for the statistics task.

18.4.3 Examples

The following sample programs demonstrate the use of the default configuration supplied in UCOS2.LIB and a custom configuration which overrides the defaults.

Example 1

In this application, ten tasks are created and one semaphore is created. Each task pends on the semaphore, gets a random number, posts to the semaphore, displays its random number, and finally delays itself for three seconds.

Looking at the code for this short application, there are several things to note. First, since µC/OS-II and slice statements are mutually exclusive (both rely on the periodic interrupt for a "heartbeat"), #use "ucos2.lib" must be included in every µC/OS-II application (1). In order for each of the tasks to have access to the random number generator semaphore, it is declared as a global variable (2). In most cases, all mailboxes, queues, and semaphores will be declared with global scope. Next, OSInit() must be called before any other µC/OS-II function to ensure that the operating system is properly initialized (3). Before µC/OS-II can begin running, at least one application task must be created. In this application, all tasks are created before the operating system begins running (4). It is perfectly acceptable for tasks to create other tasks. Next, the semaphore each task uses is created (5). Once all of the initialization is done, OSStart() is called to start µC/OS-II running (6). In the code that each of the tasks run, it is important to note the variable declarations. The default storage class in Dynamic C is static, so to ensure that the task code is reentrant, all are declared auto (7). Each task runs as an infinite loop and once this application is started, µC/OS-II will run indefinitely.

```
// 1. Explicitly use uC/OS-II library
#use "ucos2.lib"

void RandomNumberTask(void *pdata);

// 2. Declare semaphore global so all tasks have access
OS_EVENT* RandomSem;

void main()
{
   int i;

   // 3. Initialize OS internals
   OSInit();

   for(i = 0; i < OS_MAX_TASKS; i++)
      // 4. Create each of the system tasks
      OSTaskCreate(RandomNumberTask, NULL, 512, i);

   // 5. semaphore to control access to random number generator
   RandomSem = OSSemCreate(1);

   // 6. Begin multitasking
   OSStart();
}

void RandomNumberTask(void *pdata)
{
   // 7. Declare as auto to ensure reentrancy.
   auto OS_TCB data;
   auto INT8U  err;
   auto INT16U RNum;

   OSTaskQuery(OS_PRIO_SELF, &data);
   while(1)
   {
      // Rand is not reentrant, so access must be controlled via a semaphore.
      OSSemPend(RandomSem, 0, &err);
      RNum = (int)(rand() * 100);
      OSSemPost(RandomSem);
      printf("Task%d's random #: %d\n",data.OSTCBPrio,RNum);

      // Wait 3 seconds in order to view output from each task.
      OSTimeDlySec(3);
   }
}
```

Example 2

This application runs exactly the same code as Example 1, except that each of the tasks are created with 1024 byte stacks. The main difference between the two is the configuration of µC/OS-II.

First, each configuration constant that differs from the library default is defined. The configuration in this example differs from the default in that it allows only two events (the minimum needed when using only one semaphore), 20 tasks, no queues, no mailboxes, and the system tick rate is set to 32 ticks per second (1). Next, since this application uses tasks with 1024 byte stacks, it is necessary to define the configuration constants differently than the library default (2). Notice that one 512 byte stack is declared. Every Dynamic C program starts with an initial stack, and defining `STACK_CNT_512` is crucial to ensure that the application has a stack to use during initialization and before multi-tasking begins. Finally `ucos2.lib` is explicitly used (3). This ensures that the definitions in (1 and 2) are used rather than the library defaults. The last step in initialization is to set the number of ticks per second via `OSSetTicksPerSec` (4).

The rest of this application is identical to example 1 and is explained in the previous section.

```
// 1. Define necessary configuration constants for uC/OS-II
#define OS_MAX_EVENTS          2
#define OS_MAX_TASKS          20
#define OS_MAX_QS              0
#define OS_Q_EN                0
#define OS_MBOX_EN             0
#define OS_TICKS_PER_SEC      32

// 2. Define necessary stack configuration constants
#define STACK_CNT_512 1               // initial program stack
#define STACK_CNT_1K OS_MAX_TASKS    // task stacks

// 3. This ensures that the above definitions are used
#use "ucos2.lib"

void RandomNumberTask(void *pdata);

// Declare semaphore global so all tasks have access
OS_EVENT* RandomSem;

void main(){
   int i;

   // Initialize OS internals
   OSInit();

   for(i = 0; i < OS_MAX_TASKS; i++){
      // Create each of the system tasks
      OSTaskCreate(RandomNumberTask, NULL, 1024, i);
   }
   // semaphore to control access to random number generator
   RandomSem = OSSemCreate(1);

   // 4. Set number of system ticks per second
   OSSetTicksPerSec(OS_TICKS_PER_SEC);

   // Begin multi-tasking
   OSStart();
}
```

```
void RandomNumberTask(void *pdata)
{
   // Declare as auto to ensure reentrancy.
   auto OS_TCB data;
   auto INT8U  err;
   auto INT16U RNum;

   OSTaskQuery(OS_PRIO_SELF, &data);
   while(1)
   {
      // Rand is not reentrant, so access must be controlled via a semaphore.
      OSSemPend(RandomSem, 0, &err);
      RNum = (int)(rand() * 100);
      OSSemPost(RandomSem);
      printf("Task%02d's random #: %d\n",data.OSTCBPrio,RNum);
      // Wait 3 seconds in order to view output from each task.
      OSTimeDlySec(3);
   }
}
```

18.5 Compatibility with TCP/IP

The TCP/IP stack is reentrant and may be used with the µC/OS real-time kernel. The line

```
#use ucos2.lib
```

must appear before the line

```
#use dcrtcp.lib
```

A call to `OSInit()` must be made before calling `sock_init()`.

18.5.1 Socket Locks

Each socket used in a µC/OS-II application program has an associated socket lock. Each socket lock uses one semaphore of type `OS_EVENT`. Therefore, the macro `MAX_OS_EVENTS` must take into account each of the socket locks, plus any events that the application program may be using (semaphores, queues, mailboxes, event flags, or mutexes).

Determining `OS_MAX_EVENTS` may get a little tricky, but it isn't too bad if you know what your program is doing. Since `MAX_SOCKET_LOCKS` is defined as:

```
#define MAX_SOCKET_LOCKS (MAX_TCP_SOCKET_BUFFERS +
  MAX_UDP_SOCKET_BUFFERS)
```

`OS_MAX_EVENTS` may be defined as:

```
#define OS_MAX_EVENTS MAX_TCP_SOCKET_BUFFERS +
  MAX_UDP_SOCKET_BUFFERS + 2 + z
```

The constant “2” is included for the two global locks used by TCP/IP, and `z` is the number of `OS_EVENTS` (semaphores, queues, mailboxes, event flags, or mutexes) required by the program.

If either `MAX_TCP_SOCKET_BUFFERS` or `MAX_UDP_SOCKET_BUFFERS` is not defined by the application program prior to the #use statements for `ucos.lib` and `dcrtcp.lib` default values will be assigned.

If `MAX_TCP_SOCKET_BUFFERS` is not defined in the application program, it will be defined as `MAX_SOCKETS`. If, however, `MAX_SOCKETS` is not defined in the application program, `MAX_TCP_SOCKET_BUFFERS` will be 4.

If `MAX_UDP_SOCKET_BUFFERS` is not defined in the application program, it will be defined as 1 if `USE_DHCP` is defined, or 0 otherwise.

For more information regarding TCP/IP, please see the *Dynamic C TCP/IP User's Manual,* available online at zworld.com or rabbitsemiconductor.com.

18.6 Debugging Tips

Single stepping may be limited to the currently running task by using **F8** (Step over). If the task is suspended, single stepping will also be suspended. When the task is put back in a running state, single stepping will continue at the statement following the statement that suspended execution of the task.

Hitting **F7** (Trace into) at a statement that suspends execution of the current task will cause the program to step into the next active task that has debug information. It may be useful to put a watch on the global variable `OSPrioCur` to see which task is currently running.

For example, if the current task is going to call `OSSemPend()` on a semaphore that is not in the signaled state, the task will be suspended and other tasks will run. If **F8** is pressed at the statement that calls `OSSemPend()`, the debugger will not single step in the other running tasks that have debug information; single stepping will continue at the statement following the call to `OSSemPend()`. If **F7** is pressed at the statement that calls `OSSemPend()` instead of **F8**, the debugger will single step in the next task with debug information that is put into the running state.

Appendix A: Macros and Global Variables

This appendix contains descriptions of macros and global variables available in Dynamic C. This is not an exhaustive list.

A.1 Compiler-Defined Macros

The macros in the following table are defined internally. Default values are given where applicable, as well as directions for changing values.

Table A-2. Macros Defined by the Compiler

Macro Name	Definition and Default
BIOSBAUD	This is the debug baud rate. The baud rate can be changed in the Communications tab of Project Options.
_BOARD_TYPE_	This is read from the System ID block or defaulted to 0x100 (the BL1810 JackRabbit board) if no System ID block is present. This can be used for conditional compilation based on board type. Board types are listed in `boardtypes.lib`.
_CPU_ID_	This macro identifies the CPU type, e.g. R3000 is the Rabbit 3000 microprocessor.
CC_VER	Gives the Dynamic C version in hex, i.e. version 7.05 is 0x0705.
DC_CRC_PTR	Reserved.
__DATE__	The compiler substitutes this macro with the date that the file was compiled (either the BIOS or the `.c` file). The character string literal is of the form *`Mmm dd yyyy`*. The days of the month are as follows: "Jan," "Feb," "Mar," "Apr," "May," "Jun," "Jul," "Aug," "Sep," "Oct," "Nov," "Dec." There is a space as the first character of *`dd`* if the value is less than 10.
DEBUG_RST	Go to the Compiler tab of Project Options and click on the "Advanced" button at the bottom of the dialog box. Check "Include RST 28 instructions" to set `DEBUG_RST` to `1`. Debug code will be included even if `#nodebug` precedes the main function in the program.
__FILE__	The compiler substitutes this macro with the current source code file name as a character string literal.

Table A-2. Macros Defined by the Compiler

Macro Name	Definition and Default
_FAST_RAM_	These are used for conditional compilation of the BIOS to distinguish between the three options: • compiling to and running in flash • compiling to and running in RAM • compiling to flash and running in RAM The choice is made in the Compiler tab of Project Options. The default is compiling to and running in flash. The BIOS defines `FAST_RAM_COMPILE`, `FLASH_COMPILE` and `RAM_COMPILE`. These macros are defined to 0 or 1 as opposed to the corresponding compiler-defined macros which are either defined or not defined. This difference makes possible statements such as: `#if FLASH_COMPILE || FAST_RAM_COMPILE` Setting `FAST_RAM_COMPILE` limits the flash file system size to the smaller of the following two values: 256K less the SystemID/User Blocks reserved area; the sum of the completely available flash sectors between the application code/constants and the SystemID/User Blocks reserved area.
FLASH	
RAM	
_FLASH_SIZE_	These are used to set the MMU registers and code and data sizes available to the compiler. The values of the macros are the number of 4K blocks of memory available.
_RAM_SIZE_	
__LINE__	The compiler substitutes this macro with the current source code line number as a decimal constant.
NO_BIOS	Boolean value. Tells the compiler whether or not to include the BIOS when compiling to a .bin file. This is an advanced compiler option accessible by clicking the “Advanced” button on the Compiler tab in Project Options.
_TARGETLESS_COMPILE_	Boolean value. It defaults to 0. Set it by selecting “Compile defined target configuration to .bin file” under “Default Compile Mode,” in the Compiler tab of Project Options.
__TIME__	The compiler substitutes this macro with the time that the file (BIOS or `.c`) was compiled. The character string literal is of the form *`hh:mm:ss`*.

A.2 Global Variables

These variables may be read by any Dynamic C application program.

dc_timestamp

This internally-defined long is the number of seconds that have passed since 00:00:00 January 1, 1980, Greenwich Mean Time (GMT) adjusted by the current time zone and daylight savings of the PC on which the program was compiled. The recorded time indicates when the program finished compiling.

```
printf("The date and time: %lx\n", dc_timestamp);
```

OPMODE

This is a char. It can have the following values:

- 0x88 = debug mode
- 0x80 = run mode

SEC_TIMER

This unsigned long variable is initialized to the value of the real-time clock (RTC). If the RTC is set correctly, this is the number of seconds that have elapsed since the reference date of January 1, 1980. The periodic interrupt updates `SEC_TIMER` every second. This variable is initialized by the Virtual Driver when a program starts.

MS_TIMER

This unsigned long variable is initialized to zero. The periodic interrupt updates `MS_TIMER` every millisecond. This variable is initialized by the Virtual Driver when a program starts.

TICK_TIMER

This unsigned long variable is initialized to zero. The periodic interrupt updates `TICK_TIMER` 1024 times per second. This variable is initialized by the Virtual Driver when a program starts.

A.3 Exception Types

These macros are defined in `errors.lib`:

```
#define ERR_BADPOINTER             228
#define ERR_BADARRAYINDEX          229
#define ERR_DOMAIN                 234
#define ERR_RANGE                  235
#define ERR_FLOATOVERFLOW          236
#define ERR_LONGDIVBYZERO          237
#define ERR_LONGZEROMODULUS        238
#define ERR_BADPARAMETER           239
#define ERR_INTDIVBYZERO           240
#define ERR_UNEXPECTEDINTRPT       241
#define ERR_CORRUPTEDCODATA        243
#define ERR_VIRTWDOGTIMEOUT        244
#define ERR_BADXALLOC              245
#define ERR_BADSTACKALLOC          246
#define ERR_BADSTACKDEALLOC        247
#define ERR_BADXALLOCINIT          249
#define ERR_NOVIRTWDOGAVAIL        250
#define ERR_INVALIDMACADDR         251
#define ERR_INVALIDCOFUNC          252
```

A.4 Rabbit 2000/3000 Internal registers

Macros are defined for all of the Rabbit's I/O registers. A listing of these register macros can be found in the *Rabbit 2000 Microprocessor User's Manual* and the *Rabbit 3000 MIcroprocessor User's Manual*.

A.4.1 Shadow Registers

Shadow registers exist for many of the I/O registers. They are character variables defined in the BIOS. The naming convention for shadow registers is to append the word `Shadow` to the name of the register. For example, the global control status register, `GCSR`, has a corresponding shadow register named `GCSRShadow`.

The purpose of the shadow registers is to allow the program to reference the last value programmed to the actual register. This is needed because a number of the registers are write only.

Appendix B: Map File Generation

All symbol information is put into a single file. The map file has three sections: a memory map section, a function section, and a globals section.

The map file format is designed to be easy to read, but with parsing in mind for use in program down-loaders and in other possible future utilities (for example, an independent debugger). Also, the memory map, as defined by the `#org` statements, will be saved into the map file.

Map files are generated in the same directory as the file that is compiled. If compilation is not successful, the contents of the map file are not reliable.

B.1 Grammar

<mapfile>: <memmap section> <function section> <global section>

<memmap section>: <memmapreg>+

<memmapreg>: <register var> = <8-bit const>

<register var>: **XPC**|**SEGSIZE**|**DATASEG**

<function section>: <function descripton>+

<function description>: <identifier> <address> <size>

<address>: <logical address> | <physical address>

<logical address>: <16-bit constant>

<physical address: <8-bit constant>**:**<16-bit constant>

<size>: <20-bit constant>

<global section>: <global description>+

<global description>: <scoped name> <address>

<scoped name>: <global>| <local static>

<global>: <identifier>

<local static>: <identifier>**:**<identifier>

Comments are C++ style (// only).

Appendix C: Utility Programs

This appendix documents the utility programs available from Z-World. All of these utilities are easy to use.

- Library File Encryption Utility: sold separately as an add-on module. It may be obtained on our website: www.zworld.com.
- File Compression Utility: bundled with Dynamic C.
- Font and Bitmap Converter: bundled with Dynamic C.
- Rabbit Field Utility (RFU): bundled with Dynamic C, source code sold separately.

C.1 Library File Encryption Utility

The Library File Encryption Utility allows distribution of sensitive runtime library files. Complete instructions are available by clicking on the Help button within the utility, `Encrypt.exe`. Context-sensitive help is accessed by positioning the cursor over the desired subject and hitting <F1>.

The encrypted library files compile normally, but cannot be read with an editor. The files will be automatically decrypted during Dynamic C compilation, but users of Dynamic C will not be able to see any of the decrypted contents except for function descriptions for which a public interface is given. An optional user-defined copyright notice is put at the beginning of an encrypted file.

C.2 File Compression Utility

Dynamic C has a compression utility feature. The default utility implements an LZSS style compression algorithm. Support libraries to decompress files achieve a throughput of 10 KB/s to 20 KB/s (number of bytes in uncompressed file/time to decompress entire file using `ReadCompressedFile()`) depending upon file size and compression ratio.

The `#zimport` compiler directive performs a standard `#ximport`, but compresses the file by invoking the compression utility before emitting the file to the target. Support libraries allow the compressed file to be decompressed on-the-fly. Compression ratios of 50% or more for text files can be achieved , thus freeing up valuable xmem space. The compression library is thread safe.

For details on compression ratios, memory usage and performance, please see Technical Note 234, “File Compression (Using #zimport)” available on our website, at www.zworld.com.

C.2.1 Using the File Compression Utility

The utility is invoked by Dynamic C during compile time when `#zimport` is used. The keyword `#zimport` will compress any file. Of course some files are already in a compressed format, for example jpeg files, so trying to compress them further is not useful and may even cause the resulting compressed file to be larger than the original file. (The original file is not modified by the compression utility nor by the support libraries.) The compression of FS2 files is a special case. Instead of using `#zimport`, `#ximport` is used along with the function `CompressFile()`.

Compressed files are decompressed on-the-fly using `ReadCompressedFile()`. Compressed FS2 files may also be decompressed on-the-fly by using `ReadCompressedFile()`. In addition, an FS2 file may be decompressed into a new FS2 file by using `DecompressFile()`.

There are 3 sample programs to illustrate the use of file compression

- `Samples/zimport/zimport.c`: demonstrates `#zimport`
- `Samples/zimport/zimport_fs2.c`: demonstrates file compression in combination with the file system
- `Samples/tcpip/http/zimport.c`: demonstrates file compression support using the http server

C.2.1.1 File Compression/Decompression API

The file compression API consists of 7 functions, 3 of which are of prime importance:

`OpenInputCompressedFile()` - open a compressed file for reading or open an uncompressed `#ximport` file for compression.

`CloseInputCompressedFile()` - close input file and deallocate memory buffers.

`ReadCompressedFile()` - perform on-the-fly decompression.

The remaining 4 functions are included for compression support for FS2 files:

`OpenOutputCompressedFile()` - open FS2 file for use with `CompressFile()`.

`CloseOutputCompressedFile()` - close file and deallocate memory buffers.

`CompressFile()` - compress an FS2 file, placing the result in a second FS2 file.

`DecompressFile()` - decompress an FS2 file, placing the result in a second FS2 file.

Complete descriptions are available for these functions in the *Dynamic C Function Reference Manual* and also via the Function Lookup facility (Ctrl+H or Help menu).

There are several macros associated with the file compression utility:

- `ZIMPORT_MASK` - Used to determine if the imported file is compressed (`#zimport`) or not (`#ximport`).
- `OUTPUT_COMPRESSION_BUFFERS` (default = 0) - Number of 24K buffers for compression (compression also requires a 4K input buffer, which is allocated automatically for each output buffer that is defined).
- `INPUT_COMPRESSION_BUFFERS` (default = 1) Number of 4KB internal buffers (in RAM) used for decompression.

Each compressed file has an associated file descriptor of type ZFILE. All fields in this structure are used internally and must not be changed by an application program.

C.2.2 Replacing the File Compression Utility

Users can use their own compression utility, replacing the one provided. If the provided compression utility is replaced, the following support libraries will also need to be replaced: `zimport.lib`, `lzss.lib` and `bitio.lib`. They are located in `lib/zimport/`. The default compression utility, `Zcompress.exe`, is located in Dynamic C's root directory. The utility name is defined by a key in the current project file:

```
[Compression Utility]
Zimport External Utility=Zcompress.exe
```

To replace `Zcompress.exe` as the utility used by Dynamic C for compression, open your project file and edit the filename.

The compression utility must reside in the same directory as the Dynamic C compiler executable. Dynamic C expects the program to behave as follows:

- Take as input a file name relative to the Dynamic C installation directory or a fully qualified path.
- Produce an output file of the same name as the input file with the extension .DCZ at the end. E.g., `test.txt` becomes `test.txt.dcz`.
- Exit with zero on success, non-zero on failure.

If the utility does not meet these criteria, or does not exist, a compile-time error will be generated.

C.3 Font and Bitmap Converter Utility

The Font and Bitmap Converter converts Windows fonts and monochrome bitmaps to a library file format compatible with Z-World's Dynamic C applications and graphical displays. Non-Roman characters may also be converted by applying the monochrome bitmap converter to their bitmaps.

Double-click on the `fmbcnvtr.exe` file in the Dynamic C directory. Select and convert existing fonts or bitmaps. Complete instructions are available by clicking on the Help button within the utility.

When complete, the converted file is displayed in the editing window. Editing may be done, but probably won't be necessary. Save the file as `whatever.lib`: the name of your choice.

Add the file to applications with the statement:

```
#use whatever.lib          // remember to add this filename to lib.dir
```

or by cut and pasting from `whatever.lib` directly into the application file.

C.4 Rabbit Field Utility

The Rabbit Field Utility (RFU) will load a binary file created by Dynamic C to a Rabbit-based controller. It can be used to load a program to a controller without Dynamic C present on the host computer, and without recompiling the program each time it is loaded to a controller.

The Dynamic C installation created a desktop icon for the RFU. The executable file, `rfu.exe`, can be found in the subdirectory named "Utilities" where Dynamic C was installed. Complete instructions are available by clicking on the Help button within the utility. The Help document details setup information, the file menu options and BIOS requirements. The source code for the RFU may be purchased separately; it is available on our website, at www.zworld.com.

There is also a command line version of the RFU. On the command line specify:

```
clRFU SourceFilePathName [options]
```

where `SourceFilePathName` is the path name of the `.bin` file to load to the connected target. The options are as follows:

-s port:baudrate

Description: Select the comm port and baud rate for the serial connection.

Default: COM1 and 115,200 bps

RFU GUI Equivalent: From the Setup | Communications dialog box, choose values from the Baud Rate and Comm Port drop-down menus.

Example: **`clRFU myProgram.bin -s 2:115200`**

-t ipAddress:tcpPort

Description: Select the IP address and port.

Default: Serial Connection

RFU GUI Equivalent: From the Setup | Communications dialog box, click on "Use TCP/IP Connection," then type in the IP address and port for the controller that is receiving the `.bin` file or use the "Discover" radio button.

Example: **`clRFU myProgram.bin -t 10.10.1.100:4244`**

-v

Description: Causes the RFU version number and additional status information to be displayed.

Default: Only error messages are displayed.

RFU GUI Equivalent: Status information is displayed by default and there is no option to turn it off.

Example: **`clRFU myProgram.bin -v`**

-cl ColdLoaderPathName

Description: Select a new initial loader.

Default: `\bios\coldload.bin`

RFU GUI Equivalent: From the Setup | Boot Strap Loaders dialog box, type in a pathname or click on the ellipses radio button to browse for a file.

Example: **`clRFU myProgram.bin -cl myInitialLoader.c`**

-pb PilotBiosPathName

Description: Select a new secondary loader.

Default: `\bios\pilot.bin`

RFU GUI Equivalent: From the Setup | Boot Strap Loaders dialog box, type in a pathname or click on the ellipses radio button to browse for a file.

Example: **`clRFU myProgram.bin -pb mySecondaryLoader.c`**

-d

Description: Run Ethernet discovery. Don't load the `.bin` file. This option is for information gathering and must appear by itself with no other options and no binary image file name.

RFU GUI Equivalent: From the Setup | Communications dialog box, click on the "Use TCP/IP Connection" radio button, then on the "Discover" button.

Example: **`clRFU -d`**

Dynamic C User's Manual

Part Number 019-0125–B • Printed in U.S.A.

Notice to Users

Z-WORLD PRODUCTS ARE NOT AUTHORIZED FOR USE AS CRITICAL COMPONENTS IN LIFE-SUPPORT DEVICES OR SYSTEMS UNLESS A SPECIFIC WRITTEN AGREEMENT REGARDING SUCH INTENDED USE IS ENTERED INTO BETWEEN THE CUSTOMER AND Z-WORLD PRIOR TO USE. Life-support devices or systems are devices or systems intended for surgical implantation into the body or to sustain life, and whose failure to perform, when properly used in accordance with instructions for use provided in the labeling and user's manual, can be reasonably expected to result in significant injury.

No complex software or hardware system is perfect. Bugs are always present in a system of any size. In order to prevent danger to life or property, it is the responsibility of the system designer to incorporate redundant protective mechanisms appropriate to the risk involved.

Trademarks

Dynamic C® is a registered trademark of Z-World Inc.

Windows® is a registered trademark of Microsoft Corporation

Z-World, Inc.

2900 Spafford Street
Davis, California 95616-6800
USA

Telephone: (530) 757-3737
Fax: (530) 757-3792
www.zworld.com

Z-WORLD SOFTWARE END USER LICENSE AGREEMENT

IMPORTANT-READ CAREFULLY: BY INSTALLING, COPYING OR OTHERWISE USING THE ENCLOSED Z-WORLD,INC. ("Z-WORLD") DYNAMIC C SOFTWARE, WHICH INCLUDES COMPUTER SOFTWARE ("SOFTWARE") AND MAY INCLUDE ASSOCIATED MEDIA, PRINTED MATERIALS, AND "ONLINE" OR ELECTRONIC DOCUMENTATION ("DOCUMENTATION"), YOU (ON BEHALF OF YOURSELF OR AS AN AUTHORIZED REPRESENTATIVE ON BEHALF OF AN ENTITY) AGREE TO ALL THE TERMS OF THIS END USER LICENSE AGREEMENT ("LICENSE") REGARDING YOUR USE OF THE SOFTWARE. IF YOU DO NOT AGREE WITH ALL OF THE TERMS OF THIS LICENSE, DO NOT INSTALL, COPY OR OTHERWISE USE THE SOFTWARE AND IMMEDIATELY CONTACT Z-WORLD FOR RETURN OF THE SOFTWARE AND A REFUND OF THE PURCHASE PRICE FOR THE SOFTWARE.

We are sorry about the formality of the language below, which our lawyers tell us we need to include to protect our legal rights. If You have any questions, write or call Z-World at (530) 757-4616, 2900 Spafford Street, Davis, California 95616.

1. **Definitions**. In addition to the definitions stated in the first paragraph of this document, capitalized words used in this License shall have the following meanings:

   1.1 "Qualified Applications" means an application program developed using the Software and that links with the development libraries of the Software.

   1.1.1 "Qualified Applications" is amended to include application programs developed using the Softools WinIDE program for Rabbit processors available from Softools, Inc.

   1.1.2 The MicroC/OS-II (µC/OS-II) library and sample code released with any version of Dynamic C, and the Point-to-Point Protocol (PPP) library released prior to Dynamic C version 7.32 are not included in this amendment.

   1.1.3 Excluding the exceptions in 1.1.2, library and sample code provided with the Software may be modified for use with the Softools WinIDE program in Qualified Systems as defined in 1.2. All other Restrictions specified by this license agreement remain in force.

   1.2 "Qualified Systems" means a microprocessor-based computer system which is either (i) manufactured by, for or under license from Z-WORLD, or (ii) based on the Rabbit 2000 microprocessor or the Rabbit 3000 microprocessor. Qualified Systems may not be (a) designed or intended to be re-programmable by your customer using the Software, or (b) competitive with Z-WORLD products, except as otherwise stated in a written agreement between Z-World and the system manufacturer. Such written agreement may require an end user to pay run time royalties to Z-World.

2. **License**. Z-WORLD grants to You a nonexclusive, nontransferable license to (i) use and reproduce the Software, solely for internal purposes and only for the number of users for which You have purchased licenses for (the "Users") and not for redistribution or resale; (ii) use and reproduce the Software solely to develop the Qualified Applications; and (iii) use, reproduce and distribute, the Qualified Applications, in object code only, to end users solely for use on Qualified Systems; provided, however, any agreement entered into between You and such end users with respect to a Qualified Application is no less protective of Z-Worlds intellectual property rights than the terms and conditions of this License. (iv) use and distribute with Qualified Applications and Qualified Systems the program files distributed with Dynamic C named **`RFU.EXE`**, **`PILOT.BIN`**, and **`COLDLOAD.BIN`** in their unaltered forms.
3. **Restrictions**. Except as otherwise stated, You may not, nor permit anyone else to, decompile, reverse engineer, disassemble or otherwise attempt to reconstruct or discover the source code of the Software, alter, merge, modify, translate, adapt in any way, prepare any derivative work based upon the Software, rent, lease network, loan, distribute or otherwise transfer the Software or any copy thereof. You shall not make copies of the copyrighted Software and/or documentation without the prior written permission of Z-WORLD; provided that, You may make one (1) hard copy of such documentation for each User and a reasonable number of back-up copies for Your own archival purposes. You may not use copies of the Software as part of a benchmark or comparison test against other similar products in order to produce results strictly for purposes of comparison. The Software contains copyrighted material, trade secrets and other proprietary material of Z-WORLD and/or its licensors and You must reproduce, on each copy of the Software, all copyright notices and any other proprietary legends that appear on or in the original copy of the Software. Except for the limited license granted above, Z-WORLD retains all right, title and interest in and to all intellectual property rights embodied in the Software, including but not limited to, patents, copyrights and trade secrets.
4. **Export Law Assurances**. You agree and certify that neither the Software nor any other technical data received from Z-WORLD, nor the direct product thereof, will be exported outside the United States or re-exported except as authorized and as permitted by the laws and regulations of the United States and/or the laws and regulations of the jurisdiction, (if other than the United States) in which You rightfully obtained the Software. The Software may not be exported to any of the following countries: Cuba, Iran, Iraq, Libya, North Korea, Sudan, or Syria.
5. **Government End Users**. If You are acquiring the Software on behalf of any unit or agency of the United States Government, the following provisions apply. The Government agrees: (i) if the Software is supplied to the Department of Defense ("DOD"), the Software is classified as "Commercial Computer Software" and the Government is acquiring only "restricted rights" in the Software and its documentation as that term is defined in Clause 252.227-7013(c)(1) of the DFARS; and (ii) if the Software is supplied to any unit or agency of the United States Government other than DOD, the Government's rights in the Software and its documentation will be as defined in Clause 52.227-19(c)(2) of the FAR or, in the case of NASA, in Clause 18-52.227-86(d) of the NASA Supplement to the FAR.

6. **Disclaimer of Warranty**. You expressly acknowledge and agree that the use of the Software and its documentation is at Your sole risk. THE SOFTWARE, DOCUMENTATION, AND TECHNICAL SUPPORT ARE PROVIDED ON AN "AS IS" BASIS AND WITHOUT WARRANTY OF ANY KIND. Information regarding any third party services included in this package is provided as a convenience only, without any warranty by Z-WORLD, and will be governed solely by the terms agreed upon between You and the third party providing such services. Z-WORLD AND ITS LICENSORS EXPRESSLY DISCLAIM ALL WARRANTIES, EXPRESS, IMPLIED, STATUTORY OR OTHERWISE, INCLUDING BUT NOT LIMITED TO THE IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NON-INFRINGEMENT OF THIRD PARTY RIGHTS. Z-WORLD DOES NOT WARRANT THAT THE FUNCTIONS CONTAINED IN THE SOFTWARE WILL MEET YOUR REQUIREMENTS, OR THAT THE OPERATION OF THE SOFTWARE WILL BE UNINTERRUPTED OR ERROR-FREE, OR THAT DEFECTS IN THE SOFTWARE WILL BE CORRECTED. FURTHERMORE, Z-WORLD DOES NOT WARRANT OR MAKE ANY REPRESENTATIONS REGARDING THE USE OR THE RESULTS OF THE SOFTWARE IN TERMS OF ITS CORRECTNESS, ACCURACY, RELIABILITY OR OTHERWISE. NO ORAL OR WRITTEN INFORMATION OR ADVICE GIVEN BY Z-WORLD OR ITS AUTHORIZED REPRESENTATIVES SHALL CREATE A WARRANTY OR IN ANY WAY INCREASE THE SCOPE OF THIS WARRANTY. SOME JURISDICTIONS DO NOT ALLOW THE EXCLUSION OF IMPLIED WARRANTIES, SO THE ABOVE EXCLUSION MAY NOT APPLY TO YOU.
7. **Limitation of Liability**. YOU AGREE THAT UNDER NO CIRCUMSTANCES, INCLUDING NEGLIGENCE, SHALL Z-WORLD BE LIABLE FOR ANY INCIDENTAL, SPECIAL OR CONSEQUENTIAL DAMAGES (INCLUDING DAMAGES FOR LOSS OF BUSINESS PROFITS, BUSINESS INTERRUPTION, LOSS OF BUSINESS INFORMATION AND THE LIKE) ARISING OUT OF THE USE AND/OR INABILITY TO USE THE SOFTWARE, EVEN IF Z-WORLD OR ITS AUTHORIZED REPRESENTATIVE HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES. SOME JURISDICTIONS DO NOT ALLOW THE LIMITATION OR EXCLUSION OF LIABILITY FOR INCIDENTAL OR CONSEQUENTIAL DAMAGES SO THE ABOVE LIMITATION OR EXCLUSION MAY NOT APPLY TO YOU. IN NO EVENT SHALL Z-WORLDS TOTAL LIABILITY TO YOU FOR ALL DAMAGES, LOSSES, AND CAUSES OF ACTION (WHETHER IN CONTRACT, TORT, INCLUDING NEGLIGENCE, OR OTHERWISE) EXCEED THE AMOUNT PAID BY YOU FOR THE SOFTWARE.
8. **Termination**. This License is effective for the duration of the copyright in the Software unless terminated. You may terminate this License at any time by destroying all copies of the Software and its documentation. This License will terminate immediately without notice from Z-WORLD if You fail to comply with any provision of this License. Upon termination, You must destroy all copies of the Software and its documentation. Except for Section 2 ("License"), all Sections of this Agreement shall survive any expiration or termination of this License.

9. **General Provisions**. No delay or failure to take action under this License will constitute a waiver unless expressly waived in writing, signed by a duly authorized representative of Z-WORLD, and no single waiver will constitute a continuing or subsequent waiver. This License may not be assigned, sublicensed or otherwise transferred by You, by operation of law or otherwise, without Z-WORLD's prior written consent. This License shall be governed by and construed in accordance with the laws of the United States and the State of California, exclusive of the conflicts of laws principles. The United Nations Convention on Contracts for the International Sale of Goods shall not apply to this License. If for any reason a court of competent jurisdiction finds any provision of this License, or portion thereof, to be unenforceable, that provision of the License shall be enforced to the maximum extent permissible so as to affect the intent of the parties, and the remainder of this License shall continue in full force and effect. This License constitutes the entire agreement between the parties with respect to the use of the Software and its documentation, and supersedes all prior or contemporaneous understandings or agreements, written or oral, regarding such subject matter. There shall be no contract for purchase or sale of the Software except upon the terms and conditions specified herein. Any additional or different terms or conditions proposed by You or contained in any purchase order are hereby rejected and shall be of no force and effect unless expressly agreed to in writing by Z-WORLD. No amendment to or modification of this License will be binding unless in writing and signed by a duly authorized representative of Z-WORLD.

Index

L

M

N

O

P

Q

R

S