PRODUCT MANUAL

Dynamic C®

Integrated C Development System

For Rabbit 2000 and 3000 Microprocessors

User's Manual

019-0125J

The latest revision of this manual is available on the Rabbit Web site, www.rabbit.com, for free, unregistered download.

Dynamic C User's Manual

Part Number 019-0125J • Printed in the U.S.A.

Digi International Inc.© 2007 - 2008 • All rights reserved.

Digi International Inc. reserves the right to make changes and improvements to its products without providing notice.

Trademarks

RabbitSys™ is a trademark of Digi International Inc.

Rabbit® and Dynamic C® are registered trademarks of Digi International Inc.

Windows® is a registered trademark of Microsoft Corporation

Table of Contents

1. Installing Dynamic C 1
   1.1 Requirements 1
   1.2 Assumptions 1
2. Introduction to Dynamic C.. 3
   2.1 The Nature of Dynamic C 3
      2.1.1 Speed.. 3
   2.2 Dynamic C Enhancements and Differences 4
   2.3 Rabbit and Z180 Comparison.. 6
3. Quick Tutorial 7
   3.1 Run DEMO1.C 8
      3.1.1 Single Stepping 9
      3.1.2 Watch Expression.. 9
      3.1.3 Breakpoint.. 9
      3.1.4 Editing the Program 10
   3.2 Run DEMO2.C 10
      3.2.1 Watching Variables Dynamically . 10
   3.3 Run DEMO3.C 11
      3.3.1 Cooperative Multitasking.. 11
   3.4 Run DEMO4.C 12
      3.4.1 Trace Macros.. 13
   3.5 Summary of Features.. 14
4. Language.. 15
   4.1 C Language Elements.. 15
   4.2 Punctuation Tokens.. 16
   4.3 Data 17
      4.3.1 Data Type Limits.. 17
   4.4 Names 18
   4.5 Macros 19
      4.5.1 Macro Operators # and ##.. 19
      4.5.2 Nested Macro Definitions 20
      4.5.3 Macro Restrictions 21
   4.6 Numbers 21
   4.7 Strings and Character Data 22
      4.7.1 String Concatenation.. 22
      4.7.2 Character Constants 23
   4.8 S tatements.. 24
   4.9 Declarations 24
   4.10 Functions.. 25
   4.11 Prototypes.. 25
   4.12 Type Definitions.. 26
   4.13 Aggregate Data Types.. 27
      4.13.1 Array 27
      4.13.2 Structure 28
      4.13.3 Union.. 28
      4.13.4 Composites.. 28
   4.14 Storage Classes 29
   4.15 Pointers 29
   4.16 Pointers to Functions, Indirect Calls........ 30
   4.17 Argument Passing.. 31
   4.18 Program Flow 32
      4.18.1 Loops 32
      4.18.2 Continue and Break 33
      4.18.3 Branching.. 34
   4.19 Function Chaining 36
   4.20 Global Initialization.. 37
   4.21 Libraries.. 38
      4.21.1 LIB.DIR 39
   4.22 Headers 39
   4.23 Modules 39
      4.23.1 The Parts of a Module.. 40
      4.23.2 Module Sample Code.. 42
      4.23.3 Important Notes 43
   4.24 Function Description Headers 44
   4.25 Support Files.. 44
5. Multitasking with Dynamic C 45
   5.1 Cooperative Multitasking.. 45
   5.2 A Real-Time Problem.. 46
      5.2.1 Solving the Real-Time Problem with a State Machine 47
   5.3 Costa tements 48
      5.3.1 Solving the Real-Time Problem with Costatements 48
      5.3.2 Costatement Syntax 49
      5.3.3 Control Statements.. 49
   5.4 Advanced Costatement Topics 51
      5.4.1 The CoData Structure 51
      5.4.2 CoData Fields 52
      5.4.3 Pointer to CoData Structure.. 53
      5.4.4 Functions for Use With Named Costatements 53
      5.4.5 Firsttime Functions 54
      5.4.6 Shared Global Variables.. 54
   5.5 Cofunctions 54
      5.5.1 Cofunction Syntax 54
      5.5.2 Calling Restrictions.. 55
      5.5.3 CoData Structure.. 56
      5.5.4 Firsttime Functions 56
      5.5.5 Types of Cofunctions 56
      5.5.6 Types of Cofunction Calls 58
      5.5.7 Special Code Blocks 59
      5.5.8 Solving the Real-Time Problem with Cofunctions 60
   5.6 Patterns of Cooperative Multitasking........ 60

5.7 T iming Considerations.............................61
5.7.1 waitfor Accuracy Limits..............61
5.8 Overview of Preemptive Multitasking......62
5.9 Slice Statements.......................................62
5.9.1 Slice Syntax.................................62
5.9.2 Usage...62
5.9.3 Restrictions..................................63
5.9.4 Slice Data Structure......................63
5.9.5 Slice Internals..............................63
5.10 µC/OS-II..66
5.10.1 Changes to µC/OS-II..................66
5.10.2 Tasking Aware Interrupt Service Routines (TA-ISR)............................68
5.10.3 Library Reentrancy.....................74
5.10.4 How to Get a µC/OS-II Application Running...75
5.10.5 Compatibility with TCP/IP.........80
5.10.6 Debugging Tips..........................81
5.11 Summary..81

6. Debugging with Dynamic C...........................83
6.1 Debugging Features Prior to Dynamic C 983
6.2 Debugging Features Introduced in Dynamic C 9...84
6.3 Debugging Tools......................................85
6.3.1 printf()..85
6.3.2 Software Breakpoints..................86
6.3.3 Single Stepping........................... 88
6.3.4 Watch Expressions.......................89
6.3.5 Evaluate Expressions...................90
6.3.6 Memory Dump.............................91
6.3.7 MAP File.....................................92
6.3.8 Execution Trace...........................94
6.3.9 Symbolic Stack Trace..................95
6.3.10 Assert Macro............................ 96
6.3.11 Miscellaneous Debugging Tools.96
6.4 Where to Look for Debugger Features.....99
6.4.1 Run and Inspect Menus..............100
6.4.2 Options Menu............................100
6.4.3 Window Menu............................100
6.5 Debug Strategies....................................101
6.5.1 Good Programming Practices.....101
6.5.2 Finding the Bug.........................102
6.6 Reference to Other Debugging Information . 104

7. The Virtual Driver..105
7.1 Defa ult Operation...................................105
7.2 Calling _GLOBAL_INIT()....................105
7.3 Global Timer Variables...........................106
7.3.1 Example: Timing Loop..............106
7.3.2 Example: Delay Loop................107
7.4 W atchdog Timers...................................108
7.4.1 Hardware Watchdog...................108
7.4.2 Virtual Watchdogs.....................108
7.5 Preemp tive Multitasking Drivers...........109

8. Run-Time Errors... 111
8.1 Run-Time Error Handling...................... 111
8.1.1 Error Code Ranges.................... 111
8.1.2 Fatal Error Codes...................... 112
8.2 User-Defined Error Handler.................. 113
8.2.1 Replacing the Default Handler... 113
8.3 Run-Time Error Logging....................... 114
8.3.1 Error Log Buffer........................ 114
8.3.2 Initialization and Defaults.......... 115
8.3.3 Configuration Macros................ 115
8.3.4 Error Logging Functions............ 116
8.3.5 Examples of Error Log Use........ 116

9. Memory Management.................................. 117
9.1 Memory Map... 117
9.1.1 Memory Mapping Control......... 118
9.1.2 Macro to Use Second Flash for Code 118
9.2 Extended Memory Functions................. 118
9.3 Code Placement in Memory.................. 118
9.4 Dynamic Memory Allocation................ 119

10. File Systems...121
10.1 FS2...121
10.1.1 General Usage.........................121
10.1.2 Application Requirements........123
10.1.3 File System API Functions.......126
10.1.4 Setting up and Partitioning the File System...127
10.1.5 File Identifiers.........................129
10.1.6 Skeleton Program Using FS2...131
10.2 FAT File System...................................132
10.2.1 Overview of FAT Documentation .. 133
10.2.2 Running Your First FAT Sample Program..133
10.2.3 More Sample Programs............140
10.2.4 FAT Operations.......................146
10.2.5 More FAT Information.............156

11. Using Assembly Language...........................163
11.1 Mixing Assembly and C.......................163
11.1.1 Embedded Assembly Syntax....163
11.1.2 Embedded C Syntax.................164
11.1.3 Setting Breakpoints in Assembly ... 164
11.2 Assembler and Preprocessor.................165
11.2.1 Comments................................165
11.2.2 Defining Constants...................165
11.2.3 Multiline Macros......................167
11.2.4 Labels......................................167
11.2.5 Special Symbols.......................167
11.2.6 C Variables..............................168
11.3 Stand-Alone Assembly Code................169
11.3.1 Stand-Alone Assembly Code in Extended Memory..........................169

11.3.2 Example of Stand-Alone Assembly Code ... 170
11.4 Embedded Assembly Code ... 170
11.4.1 The Stack Frame ... 170
11.4.2 Embedded Assembly Example 172
11.4.3 The Disassembled Code Window .. 173
11.4.4 Local Variable Access ... 174
11.5 C Calling Assembly ... 175
11.5.1 Passing Parameters ... 175
11.5.2 Location of Return Results ... 175
11.5.3 Returning a Structure ... 175
11.6 Assembly Calling C ... 177
11.7 Interrupt Routines in Assembly ... 178
11.7.1 Steps Followed by an ISR ... 178
11.7.2 Modifying Interrupt Vectors ... 179
11.8 Common Problems ... 184
12. Keywords ... 185
abandon ... 185
abort ... 185
align ... 186
always_on ... 186
anymem ... 186
asm ... 187
auto ... 187
bbram ... 187
break ... 188
c ... 188
case ... 188
char ... 189
cofunc ... 189
const ... 190
continue ... 191
costate ... 191
debug ... 191
default ... 192
do ... 192
else ... 192
enum ... 193
extern ... 193
firsttime ... 194
float ... 194
for ... 195
goto ... 195
if ... 196
init_on ... 196
int ... 197
interrupt ... 197
interrupt_vector ... 198
__lcall__ ... 200
long ... 200
main ... 200
nodebug ... 201
norst ... 201
nouseix ... 201
NULL ... 201
protected ... 202
register ... 202
return ... 203
root ... 203
scofunc ... 203
segchain ... 204
shared ... 204
short ... 205
size ... 205
sizeof ... 205
speed ... 205
static ... 206
struct ... 206
switch ... 207
typedef ... 207
union ... 208
unsigned ... 208
useix ... 208
void ... 209
volatile ... 209
waitfor ... 209
waitfordone (wfd) ... 210
while ... 210
xdata ... 211
xmem ... 212
xstring ... 212
yield ... 213
12.1 Compiler Directives ... 214
#asm ... 214
#class ... 214
#debug #nodebug ... 215
#define ... 215
#endasm ... 215
#fatal ... 215
#GLOBAL_INIT ... 216
#error ... 216
#funcchain ... 216
#if #elif #else #endif ... 217
#ifdef ... 217
#ifndef ... 218
#interleave #nointerleave ... 218
#makechain ... 218
#memmap ... 219
#pragma ... 219
#precompile ... 220
#undef ... 220
#use ... 220
#useix #nouseix ... 221
#warns ... 221
#warnt ... 221
#ximport ... 221
#zimport ... 222
13. Operators ... 223
13.1 Arithmetic Operators ... 224
+ ... 224
– ... 224
* ... 225
/ ... 225
++ ... 226
— ... 226
% ... 226

13.2 Assignment Operators 227
= 227
+= 227
-= 227
*= 227
/= 227
%= 227
<<= 227
>>= 228
&= 228
^= 228
|= 228
13.3 Bitwise Operators 228
<< 228
>> 228
& 229
^ 229
| 229
~ 229
13.4 Relational Operators 230
< 230
<= 230
> 230
>= 230
13.5 Equality Operators 231
== 231
!= 231
13.6 Logical Operators 231
&& 231
|| 231
! 232
13.7 Postfix Expressions 232
( ) 232
[ ] 232
. (dot) 232
-> 233
13.8 Reference/Dereference Operators 233
& 233
* 233
13.9 Conditional Operators 234
? : 234
13.10 Other Operators 235
(type) 235
sizeof 235
, ... 236

14. Graphical User Interface 237
14.1 Editing 237
14.2 Menus 238
14.2.1 Using Keyboard Shortcuts 238
14.2.2 File Menu 239
14.2.3 Edit Menu 241
14.2.4 Compile Menu 245
14.2.5 Run Menu 247
14.2.6 Inspect Menu 249
14.2.7 Options Menu 254
Environment Options 254
Editor Tab 254
Gutter & Margin Tab 257
Display Tab 259
Syntax Colors Tab 260
Code Templates Tab 261
Debug Windows Tab 262
Print/Alerts Tab 269
Project Options 270
Communications Tab 270
Compiler Tab 272
Debugger Tab 278
Defines Tab 281
Targetless Tab 283
14.2.8 Window Menu 286
14.2.9 Help Menu 292

15. Command Line Interface 295
15.1 Default States 295
15.2 User Input 295
15.3 Saving Output to a File 296
15.4 Command Line Switches 296
15.4.1 Switches Without Parameters ... 296
15.4.2 Switches Requiring a Parameter 304
15.5 Examples 312
15.6 Command Line RFU 313

16. Project Files 317
16.1 Project File Names 317
16.1.3 Active Project 317
16.2 Updating a Project File 318
16.3 Menu Selections 318
16.4 Command Line Usage 319

17. Hints and Tips 321
17.1 A User-Defined BIOS 321
17.2 Efficiency 322
17.2.1 Nodebug Keyword 322
17.2.2 In-line I/O 323
17.3 Run-time Storage of Data 323
17.3.1 User Block 324
17.3.2 Flash File System 324
17.3.3 WriteFlash2 324
17.3.4 Battery-Backed RAM 324
17.4 Root Memory Reduction Tips 325
17.4.1 Increasing Root Code Space 325
17.4.2 Increasing Root Data Space 327

Appendix A: Macros and Global Variables 329
Macros Defined by the Compiler 329
Macros Defined in the BIOS or Configuration Libraries 331
Global Variables 332
Exception Types 333
Rabbit Registers 333

Appendix B: Map File Generation 335
Grammar.. 335

Appendix C: Security Software & Utility Programs 337
Rabbit Embedded Security Pack 337
AES 337
SSL 337
Dynamic C Utilities................................... 338
Library File Encryption 338
File Compression Utility 339
Font and Bitmap Converter Utility 341
Rabbit Field Utility 341

Appendix D: Additional Documentation......... 347

Software License Agreement 349

Index 353

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

Dynamic C requires an IBM-compatible PC running Windows 2000 or later with at least one free COM or USB port.

Please note that Windows Vista is supported by Dynamic C out of the box if there is only one processor in the host PC or laptop. With multiple processors (a.k.a., dual cores) present in the host system, you must check Windows "Processor Affinity" setting in order to ensure Vista compatibility with Dynamic C. Technical note TN257 "Running Dynamic C with Windows Vista" has instructions for modifying the "Processor Affinity" setting. This technical note is available on the Rabbit website:

```
http://www.rabbit.com/support/techNotes_whitePapers.shtml#dcp
```

Starting with Dynamic C 9.60, the "Processor Affinity" setting is set automatically.

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.

Refer to one or both of the following texts for a full treatment of C:

- *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 Rabbit controllers and other controllers based on the Rabbit microprocessor.

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 C programs 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 and breakpoints. 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 9 introduces advanced debugging features such as execution and stack tracing. Execution tracing can be used to follow the execution of debuggable statements, including such information as function/file name, source code line and column numbers, action performed, time stamp of action performed and register contents. Stack tracing shows function call sequences and parameter values.

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 multitasking.

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 five 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 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. Rabbit 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 cooperative, 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 keywords that help protect data shared between different contexts (shared) or stored in battery-backed memory (protected).

- Dynamic C has a set of features that allow the programmer to make the fullest use of xmem (extended memory). The compiler supports a 1 MB physical address space.

  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 in the proper place, such as: root, xmem, and #memmap.

  See Chapter 9 for further details on memory management.

2.2.2 Dynamic C Differences

The main differences in Dynamic C are summarized in the list below and discussed in detail in Chapter 4. "Language" and Chapter 12. "Keywords".

- 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 Section 4.23, "Modules" for more information.

- 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. See Section 4.16 for more information

- Bit fields are not supported.

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

2.3 Rabbit and Z180 Comparison

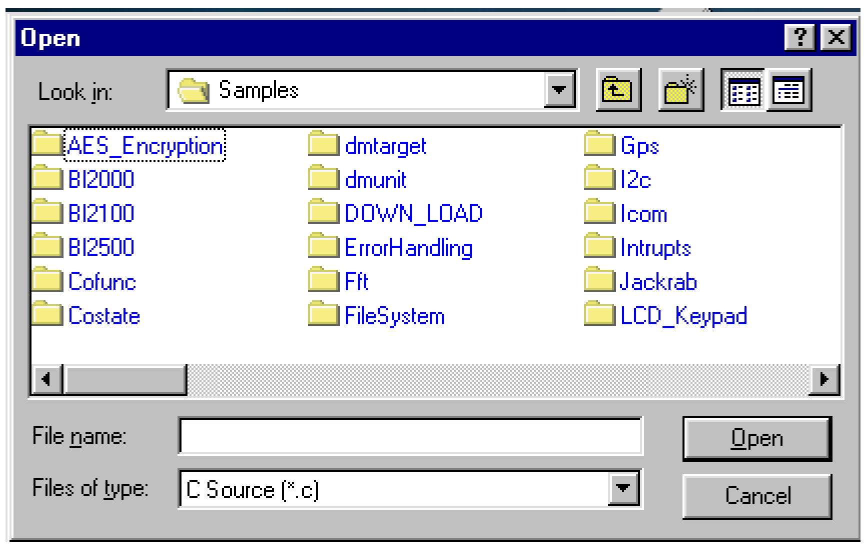
A major difference in the way Dynamic C interacts with a Rabbit-based board compared to a Z180 or 386EX board is that Dynamic C expects no BIOS kernel to be present on the target when it starts up. Dynamic C stores the BIOS kernel as a C source file. Dynamic C compiles and loads it to the Rabbit target 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 Rabbit 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 Rabbit.
- A major feature is the ability to program and debug over the Internet or local Ethernet. This requires either the use of a RabbitLink board, available alone or as an option with Rabbit-based development kits, or the use of RabbitSys.

3. Quick Tutorial

Sample programs are provided in the Dynamic C Samples folder, which is in the root directory where Dynamic C was installed. The Samples folder contains many subfolders, as shown in Figure 3.1. Sample programs are provided in source code format. You can open the source code file in Dynamic C and read the comment block at the top of the sample program for a description of its purpose and other details. Comments are also provided throughout the source code. This documentation, provided by the software engineers, is a rich source of information.

Figure 3.1 Screenshot of Samples Folder



The subfolders contain sample programs that illustrate the use of the various Dynamic C libraries. For example, 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. Besides its subfolders, the Samples folder also contains some sample programs to demonstrate various aspects of Dynamic C. For example, the sample program `Pong.c` demonstrates output to the Stdio window.

In the rest of this chapter we examine four sample programs in some detail.

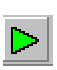
3.1 Run DEMO1.C

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 3.2 (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 Function Description window for `printf()`. You can do this with all functions in the Dynamic C libraries, including libraries you write yourself.

Figure 3.2 Sample Program DEMO1.C

C:\DC_960\Samples\DEMO1.C

```
/*****************************************************

     demo1.c
     Z-World, 2000

      Sample program for Dynamic C Premier tutorial
*******************************************************/

main() {

   int i, j;

   i = 0;

   while (1) {
      i++;

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

      printf("i = %d\n", i);
   }
}
```



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>.)

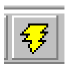
The value of the counter should be printed repeatedly to the Stdio window if everything went well. If this doesn’t work, review the following points:

- 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 pressing <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

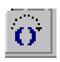
dialog box must be the same as the one the programming cable is connected to. (The Communications dialog box is accessed via the Communications tab of the Options | Project Options menu.)

- 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 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>. You will see the value of “i” displayed in the Stdio window repeatedly until program execution is halted.

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.

Starting with Dynamic C 9, you can also set breakpoints while in edit mode. Breakpoint information is not only retained when going back and forth from edit mode to debug mode, it is stored when a file is closed and restored when the file is re-opened.

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 theoriginal 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.

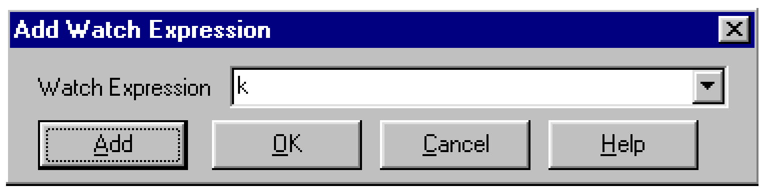
3.2 Run DEMO2.C

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."

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.

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.

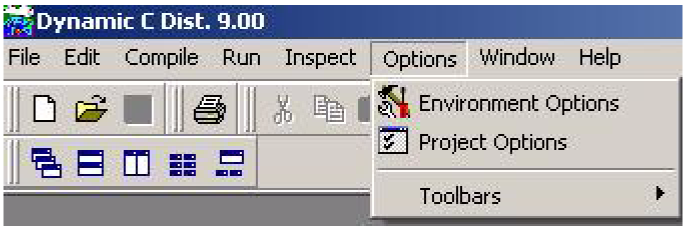
The elapsed time message is printed by the costatement starting at the line marked (2). Costatements need to be executed regularly, often at least every 25 ms. To accomplish this, the costatements are enclosed in a `while` loop. The `while` loop starts at (1) and ends at (6). The statement at (3) waits 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 for more details.

The second costatement in the program checks to see if an alpha-numeric 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 running.To 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 Run DEMO4.C

The sample program `DEMO4.C` uses execution tracing. This is one of the advanced debugging features introduced in Dynamic C 9. Tracing records program state information based on options you choose in the Debugger tab of the Project Options dialog. The information captured from the target by Dynamic C's tracing feature is displayed in the Trace window, available from the Window menu. To make the target send trace information, you must turn on tracing either from the INSPECT menu or from within your program using one of the macros described here.



To use this sample program, first go to the Debugger tab of the Project Options dialog, select Enable Tracing, and choose Full for the Trace Level. Click OK to save and close the dialog, then compile and run `DEMO4.C`. When the program finishes, the Trace window will open and you can examine its entries. The Trace window can be opened anytime after the program is compiled, but execution speed is slightly affected if the window is open while the program is running.

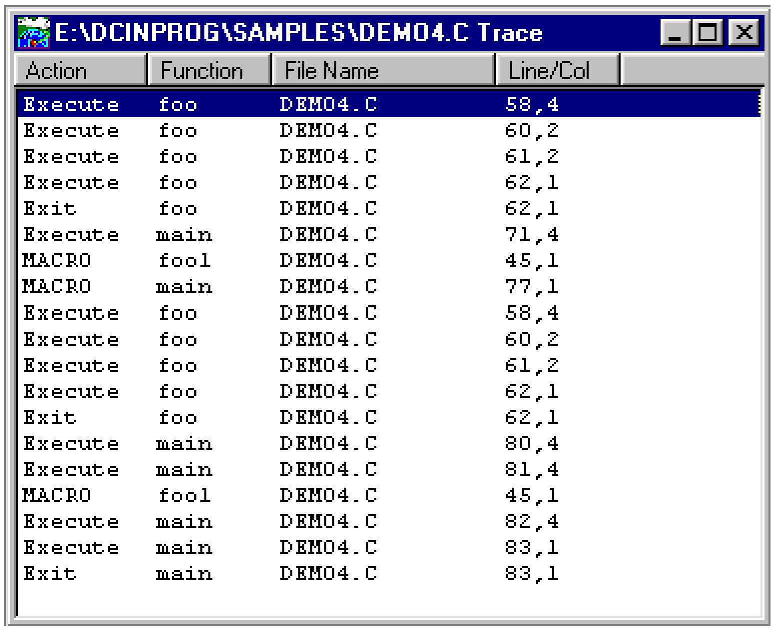
3.4.1 Trace Macros

Trace macros provide more fine-grained control than the menu options.

_TRACE

The `_TRACE` macro creates one entry in the trace buffer containing the program state information at the time the macro executes. It is useful if you want to monitor one statement closely rather than follow the flow of part of a program. In `Demo4.c`, `_TRACE` is executed at lines 45 and 77, as you can see in the screenshot in Figure 3.3.

Figure 3.3 Trace window contents after running Demo4.c



The `_TRACE` macro does not affect the `_TRACEON` and `_TRACEOFF` macros, and likewise is not affected by them. It will execute regardless of whether tracing is turned on or off. An interesting thing to note about `_TRACE` is that it generate a trace statement even when it appears in a `nodebug` function.

_TRACEON

The `_TRACEON` macro turns on tracing. This does not cause any information to be recorded by itself like the `_TRACE` macro, but rather causes a change of state within the debug kernel so that program state information is recorded for program and library statements executed thereafter, until the `_TRACEOFF` macro is executed or by menu command. Dynamic C captures the information you specified in the Project Options dialog and displays it in the Trace window.

In `Demo4.c`, `_TRACEON` is executed in the function `foo()`. Note that tracing is turned on in the second call to `foo1()` in `main()`, but that except for the `_TRACE` statement there are no trace statements for `foo1()`. This is because statements in nodebug functions are not traceable.

_TRACEOFF

The `_TRACEOFF` macro turns off tracing, starting with the next statement after it executes. Instances of the `_TRACE` macro will still execute, but tracing remains off until it is turned on by the `_TRACEON` macro or by menu command.

3.5 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 functions will take place. With F8 the function 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. Prior to Dynamic C 9, breakpoints could only be toggled while in run mode, either while stopped at a breakpoint or when the program ran at full speed. Starting with Dynamic C 9, breakpoints can be set in edit mode and retained when changing modes or closing the file.

Watch Expressions

A watch expression is a C expression that is evaluated on command in the Watches window. An expression is basically any type of C statement that can include operators, variables, structures 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 expressions 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.

Execution Tracing

Execution tracing allows you to follow the flow of your program's execution in real time instead of single stepping through it. The Trace window can show which statement was executed, what type of action it was, when it was executed, and the contents of the registers after executing it. You can also save the contents of the Trace window to a file.

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 main() function 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 Table 4-1.

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. Table 4-2 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 (a.k.a., 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 Dynamic C 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 a 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 code.

```
#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.

4.5.1 Macro Operators # and

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.

4.5.2 Nested Macro Definitions

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) doesn't 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);
```

4.5.3 Macro 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 code illustrates a typical use of strings.

```
const char * const 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
```

Note that both the pointer and the elements of the array are explicitly defined as const. Some versions of Dynamic C allowed the second const to be omitted. Current versions of the compiler generate an error unless the second const is included.

4.7.1 String Concatenation

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

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

becomes, during compilation:

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

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

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

becomes, during compilation:

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

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. Prior to Dynamic C 9.60, the compiler will parse up to 250 bytes for any single C statement in a ".c" or a ".lib" file. Starting with Dynamic C 9.60, the compiler will parse up to 64K bytes for any single C statement in a ".c" file; the 250 byte limit still exists for ".lib" files.

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.This was explained in Section 4.11.

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 the array.
```

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;
```

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.

The `register` type is reserved, but is not currently implemented. Dynamic C will change a variable to be of type `auto` if `register` is encountered. Even though the `register` keyword is not implemented, it still can not be used as a variable name or other symbol name. Its use will cause unhelpful error messages from the compiler.

4.15 Pointers

A pointer is a variable that holds the 16-bit logical address of another variable, a structure, or a function. 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 tab on the Options | Project 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 C 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. This means that the auto promotions provided by Dynamic C type checking will not occur, so values must be cast to the type that is expected or the size may not be correct. For example, if a function takes a long as a parameter, and you pass it a 16-bit integer value, it must be cast to type long in order for 4 bytes to be put onto the stack.

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 statement(s)
}
```

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 $\texttt{const}_N$ values matches the `switch` *expression*, the sequence of statements identified by the $\texttt{const}_N$ 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){
   static int i;
   static 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 storage class for variables used in a global initialization section must be static. Since the default storage class is auto, you must define variables as static in your application.

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 Section 7.2 for more information about calling `_GLOBAL_INIT()`.

4.21 Libraries

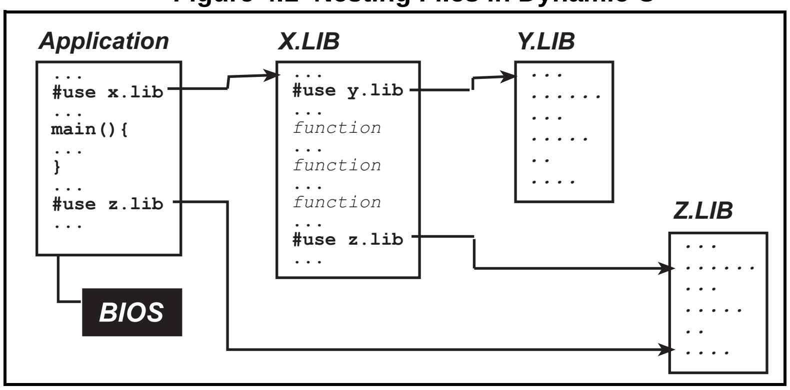
Dynamic C includes many libraries—files of useful functions in source code form. They are located in the \LIB directory where Dynamic C was installed. The default library file extension is `.LIB`. Dynamic C uses functions and data from library files and compiles them with an application program that 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 (those defined by you and those defined by Rabbit) 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.)

Figure 4.1 Nesting Files in Dynamic C



Most libraries needed by Dynamic C programs have #use statements in `lib\..\default.h`.

Section 4.23 explains how Dynamic C knows which functions and global variables in a library are available for use.

4.21.1 LIB.DIR

Any library that is to be #use'd in a Dynamic C program must be listed in the file `LIB.DIR`, or another `*.DIR` file specified by the user.

The lib.dir strategy starting with Dynamic C 9.30 allows naming a folder with optional mask(s). No mask implies *.* and multiple masks are separated by ";" so that "lib" and "lib*.*" both include all files and "`lib\*.lib;*.c;*.h*`" includes all files with extensions of `.lib`, `.c` and `.h`. Dynamic C generated file (e.g., `.mdl`, `.hxl`, etc.) are not parsed, which means they are excluded when using the wildcard mask.

Dynamic C now enforces unique file extension names regardless of path, so that "#use myfile.lib" can not use an unintended copy of `myfile.lib` as the list of pathnames included in `lib.dir` is searched for the first occurrence of that file extension. An error message naming both full paths will come up when trying to compile ANY program alerting the user of the infraction.

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 usually 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 asterisks and one space (`/*** `). The forward slash must be the first character on the line. The `BeginHeader` comment must end with an asterisk 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 three modules defined in this code. The first one is responsible for the variable `ticks`, the second and third modules define functions `Get_Ticks()` and `Inc_Ticks` that access the variable. 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 a `LIB.DIR` (or a user defined replacement for it). 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 Dynamic C 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 <Ctrl+H> feature if the library is listed in `lib.dir` or its replacement. 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 Rabbit ships. This file can be modified.
`LIB.DIR`	Contains pathnames for all libraries that will be known to Dynamic C. The programmer can add 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.
`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

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.

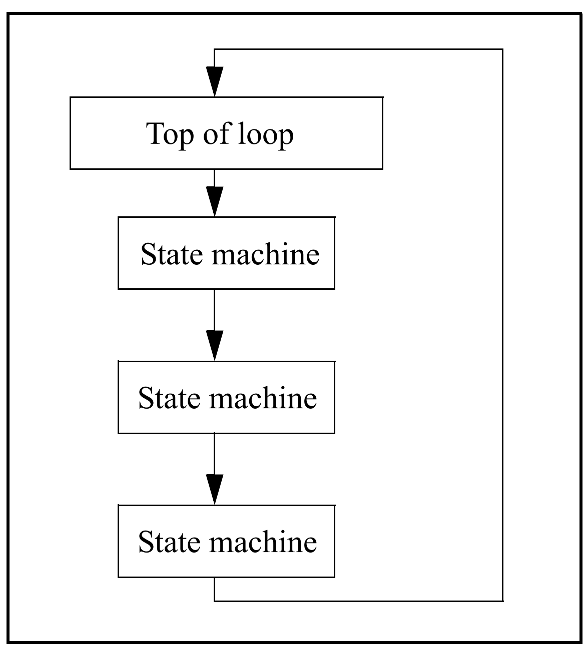
There are two types of multitasking available for developing applications in Dynamic C: *preemptive* and *cooperative*. In a cooperative multitasking environment, each well-behaved task voluntarily gives up control when it is waiting, allowing other tasks to execute. Dynamic C has language extensions, *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



Within this endless 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

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

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

`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 a costatement name.

5.3.3 Control Statements

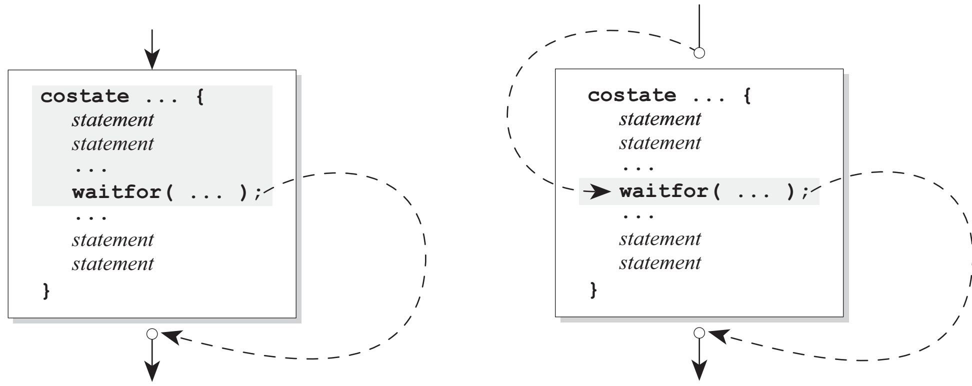
This section describes the control statements identified by the keywords: waitfor, yield and abort.

waitfor (expression);

The keyword `waitfor` indicates a special `waitfor` statement and not a function call. Each time `waitfor` is executed, *expression* is evaluated. 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.

Figure 5-2 shows the execution thread through a costatement when a `waitfor` evaluates to false. The diagram on the left side shows which statements are executed the first time through the costatement. The diagram on the right shows that when the execution thread again reaches the costatement the only statement executed is the `waitfor`. As long as the `waitfor` continues to evaluate to false, it will be the only statement executed within the costatement.

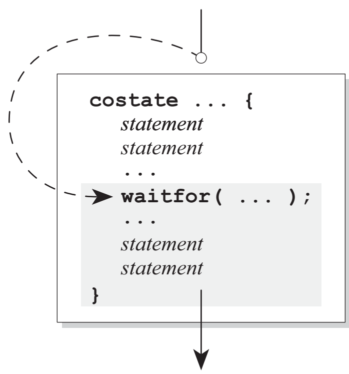
Figure 5-2 Execution thread when waitfor evaluates to false



(a) First Time   (b) Subsequent Times

Figure 5-3 shows the execution thread through a costatement when a `waitfor` evaluates to true.

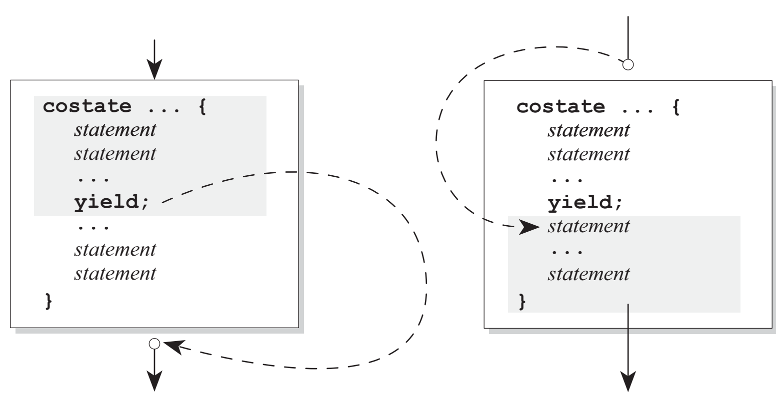
Figure 5-3 Execution thread when waitfor evaluates to true



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 by the execution thread.

Figure 5-4 Execution thread with yield statement

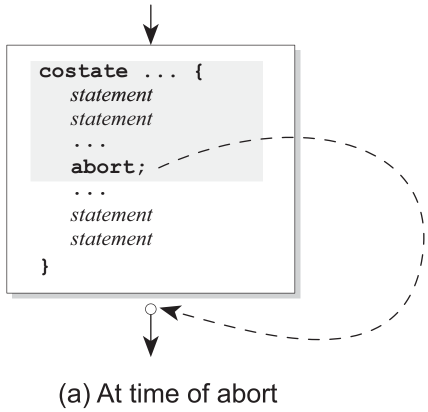


(a) Execution of yield   (b) Execution thread the next time the costate is encountered

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.

Figure 5-5 Execution thread with abort statement



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

This section describes the fields of the CoData structure.

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:

```
static CoData   cost1;           // allocate memory for a CoData struct
static 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.
}
```

The storage class of a named CoData structure must be `static`.

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.

`int isCoDone(CoData* p);`

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

`int isCoRunning(CoData* p);`

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

`void CoBegin(CoData* p);`

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

`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`.

`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`.

`void CoResume(CoData* p);`

This function unpauses a paused costatement. The costatement resumes 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. User-defined `firsttime` functions are allowed.

The following `firsttime` functions are defined in `COSTATE.LIB`.

```
DelayMs(), DelaySec(), DelayTicks()
IntervalMs(), IntervalSec(), IntervalTick()
```

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.

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. For example, 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 Costate Within a Cofunc

In all but trivial cases (where the costate is really not necessary), a costate within a cofunc causes execution problems ranging from never completing the cofunc to unexpected interrupts or target lockups. To avoid these problems, do not introduce costates with nested wfd cofuncs into a cofunc. If you find yourself coding such a thing, consider these alternatives:

1. Intermediate regular functions can be used between the cofuncs to isolate them.
2. A regular `waitfor(function)` can be substituted for the top level costate's `wfd` cofunction.
3. The nested costates with wfd cofuncs can be moved up into the body of the calling function, replacing the top-level costate with the wfd cofunc.

A compiler error will be generated if a costate is found within a cofunction.

5.5.2.2 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 54 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 calls to single-user cofunction, 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;
   serAopen(19200);
   loopinit();
   while (1) {
      loophead();
      costate {
         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. The lifetime of a cofunction instance stretches from a first time call until its terminal call or until its next first time call.

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.

5.5.7 Special Code Blocks

The following special code blocks can appear inside a cofunction.

everytime { *statements* }

This must be the first statement in the cofunction. The everytime statement block will be executed on every `cofunc` continuation call no matter where the statement pointer is pointing. After the everytime statement block is executed, control will pass to the statement pointed to by the cofunction's statement pointer.

The everytime statement block will not be executed during the initial `cofunc` entry call.

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

Cofunctions, with their ability to receive arguments and return values, provide more flexibility and specificity than our previous solutions.

```
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
}
```

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 cofunction 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.

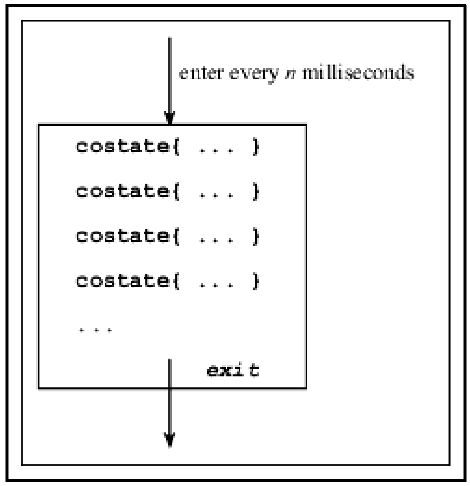
```
costate {
   waitfordone { Task1(); Task2(); Task3(); Task4(); }
   [ to come here is an error ]
}
```

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 5-6 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 via a Dynamic C construct called the "slice" statement (described in Section 5.9). The second way is µC/OS-II, a real-time, preemptive kernel that runs on the Rabbit microprocessor and is fully supported by Dynamic C (described in Section 5.10).

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 function called from the slice statement performs 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 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. With Dynamic C, there is no fee to pay for the "Object Code Distribution License" that is usually required for embedding µC/OS-II in a product.

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 Rabbit store, www.rabbit.com/store/, or at http://www.ucos-ii.com/.

The Dynamic C 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 will be in the `/Samples/UCos-II` directory. The file `Newv251.pdf` contains all of the features added since version 2.00 and `Relv251.pdf` contains release notes for version 2.51.

The remainder of this section discusses the following:

- Dynamic C enhancements to µC/OS-II
- Tasking aware ISRs
- Dynamic C library reentrancy
- How to get a µC/OS-II application running
- TCP/IP compatibility
- API function descriptions
- Debugging tips

5.10.1 Changes to µC/OS-II

Minor changes have been made to µC/OS-II to take full advantage of services provided by Dynamic C.

5.10.1.1 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()`. For example, to change the tick rate to 32 ticks per second:

```
#define OS_TICKS_PER_SEC 32
...
OSInit();
...
OSStart();
```

5.10.1.2 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. 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);
```

5.10.1.3 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.

5.10.2 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.

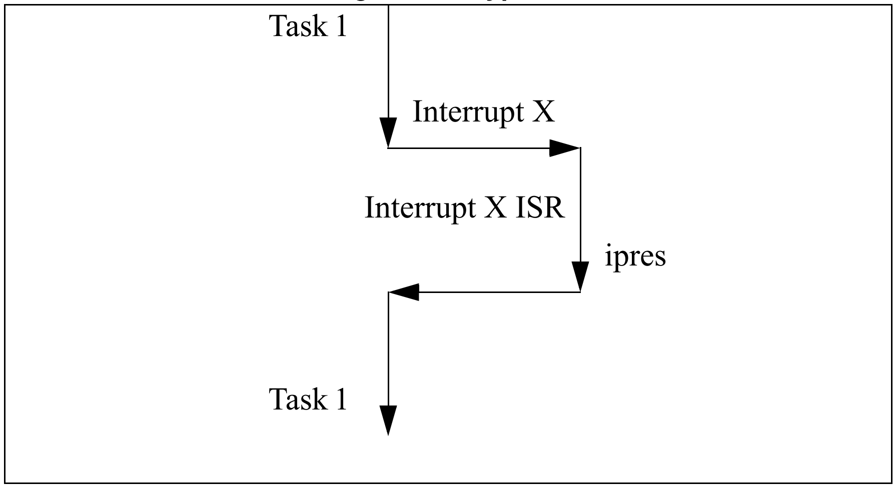
5.10.2.1 Interrupt Priority Levels

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.

5.10.2.2 Possible ISR Scenarios

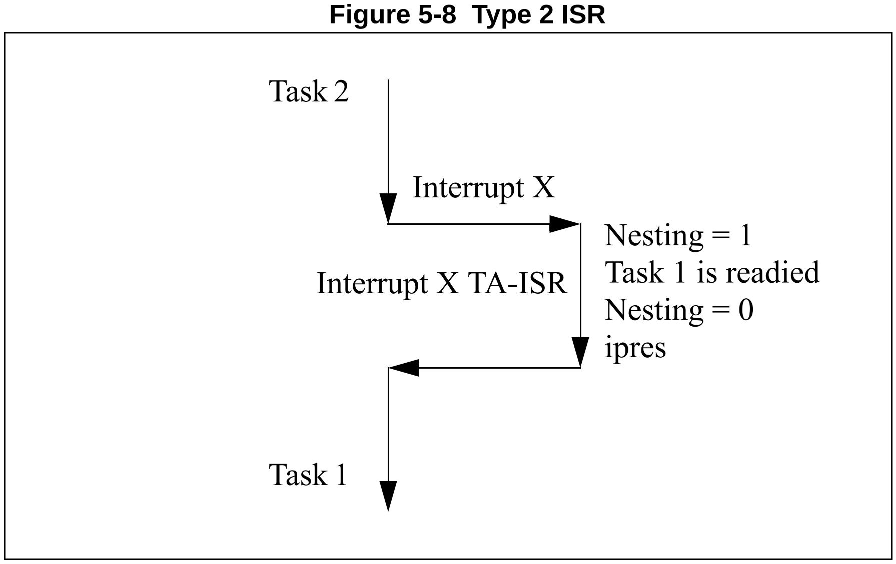
There are several different scenarios that must be considered when writing an ISR for use with µC/OS-II. Depending on the use of the ISR, it may or may not have to be written so that it is tasking aware. Consider the scenario in Figure 5-7. In this situation, the ISR for Interrupt X does not have to be tasking aware since it does not re-enable interrupts before completion and it does not post to a semaphore, mailbox, or queue.

Figure 5-7 Type 1 ISR



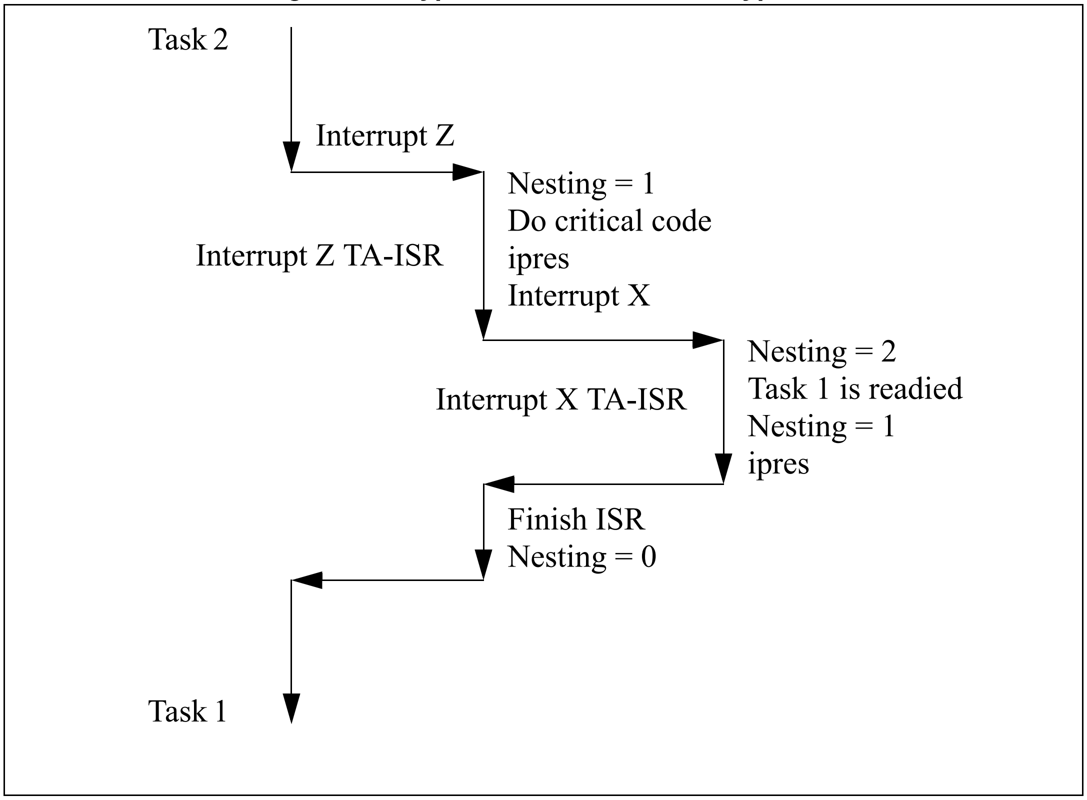
If, however, an ISR needs to signal a task to the ready state, then the ISR must be tasking aware. In the example in Figure 5-8, the TA-ISR increments the interrupt nesting counter, does the work necessary for the ISR, readies a higher priority task, decrements the nesting count, and returns to the higher priority task.

Figure 5-8 Type 2 ISR



It may seem as though the ISR in Figure 5-8 does not have to increment and decrement the nesting count. However, this is 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. Figure 5-9 shows an example of this situation.

Figure 5-9 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.

5.10.2.3 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.

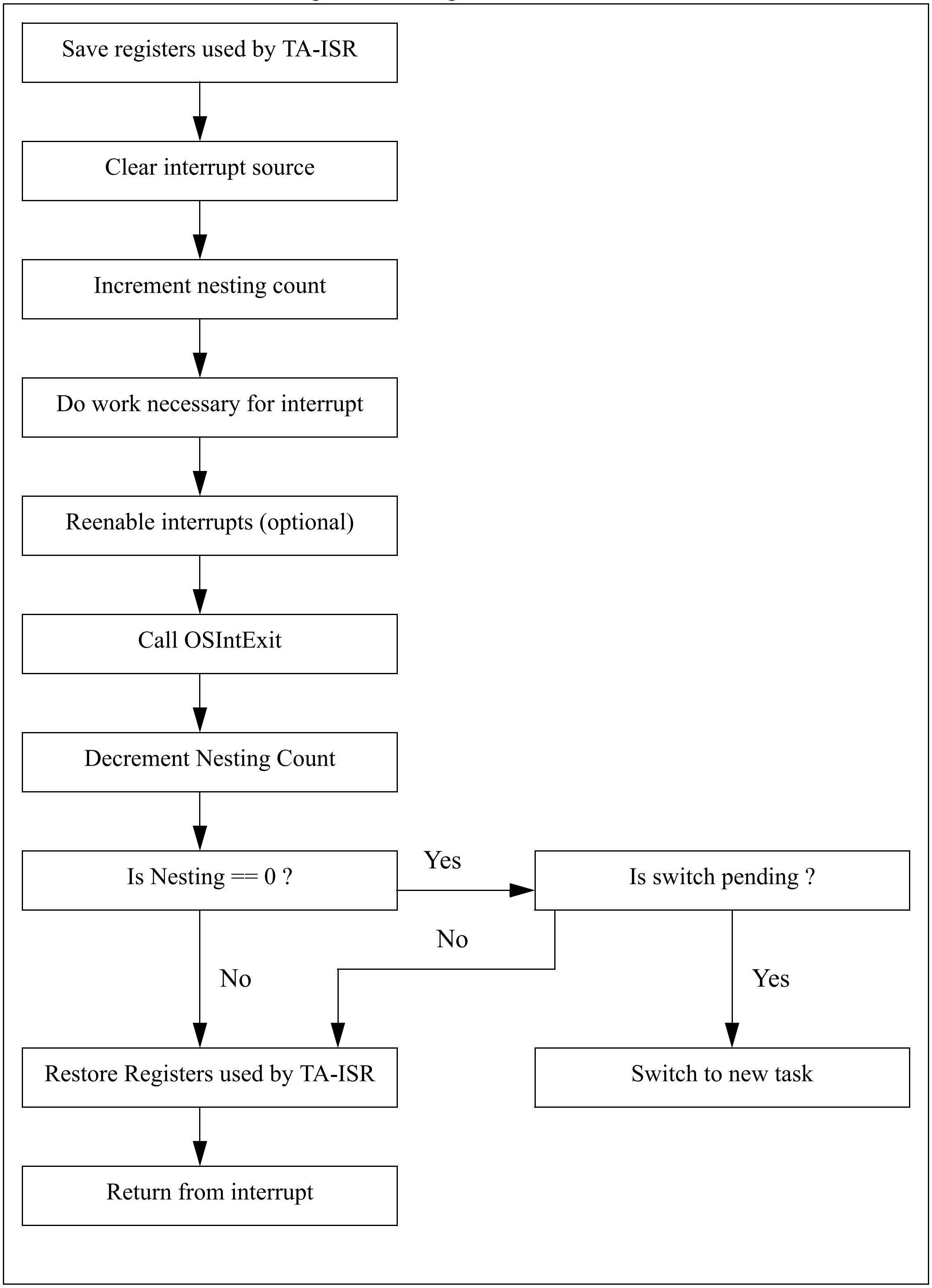
Table 5-2. Use of TA-ISR

	µC/OS-II Application		
	Type 1[a]	**Type 2[b]**	**Type 3[c]**
TA-ISR Required?	No	Yes	Yes

a. Type 1—Leaves interrupts disabled and does not signal task to ready state
b. Type 2—Leaves interrupts disabled and signals task to ready state
c. Type 3—Reenables interrupts before completion

Figure 5-10 shows the logical flow of a TA-ISR.

Figure 5-10 Logical Flow of a TA-ISR



5.10.2.3.1 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
```

5.10.3 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.

Table 5-3. Dynamic C Non-Reentrant Functions

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`[a]`, atoi1, 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

a. reentrant but sets the global `_xtoxErr` flag

The Dynamic C 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.

5.10.4 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).

5.10.4.1 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:

`OS_MAX_EVENTS`	Max number of events (semaphores, queues, mailboxes) Default is 10
`OS_MAX_TASKS`	Maximum number of tasks (less stat and idle tasks) Default is 10
`OS_MAX_QS`	Max number of queues in system Default is 2
`OS_MAX_MEM_PART`	Max number of memory partitions Default is 1
`OS_TASK_CREATE_EN`	Enable normal task creation Default is 1
`OS_TASK_CREATE_EXT_EN`	Disable extended task creation Default is 0
`OS_TASK_DEL_EN`	Disable task deletion Default is 0
`OS_TASK_STAT_EN`	Disable statistics task creation Default is 0
`OS_Q_EN`	Enable queue usage Default is 1

OS_MEM_EN — Disable memory manager
Default is 0

OS_MBOX_EN — Enable mailboxes
Default is 1

OS_SEM_EN — Enable semaphores
Default is 1

OS_TICKS_PER_SEC — Number of ticks in one second
Default is 64

STACK_CNT_256 — Number of 256 byte stacks (idle task stack)
Default is 1

STACK_CNT_512 — Number of 512-byte stacks
(task stacks + initial program stack)
Default is `OS_MAX_TASKS+1` (11)

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.

5.10.4.2 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 ten of the application tasks will each have a 1024 byte stack, ten will each have a 2048 byte stack, and an extra stack is declared for the statistics task.

5.10.4.3 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. Each task runs as an infinite loop and once this application is started, µC/OS-II will run indefinitely.

```
// 1. Explicitly use µC/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)
{
   OS_TCB data;
   INT8U  err;
   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 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);
   }
}
```

5.10.5 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()`.

5.10.5.1 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 about TCP/IP, please see the *Dynamic C TCP/IP User's Manual, Volumes 1 and 2,* available online at rabbit.com.

5.10.6 Debugging Tips

Single stepping may be limited to the currently running task by using the F8 key (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.

Pressing the F7 key (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.

5.11 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. Debugging with Dynamic C

This chapter is intended for anyone debugging Dynamic C programs. For the person with little to no experience, we offer general debugging strategies in Section 6.5. Both experienced and inexperienced Dynamic C users can refer to Section 6.3 to see the full set of tools, programs and functions available for debugging Dynamic C programs. Section 6.4 consolidates the information found in the GUI chapter regarding debugging features into an quicker-to-read table of GUI options. And lastly, Section 6.6 gives some good references for further study.

Dynamic C comes with robust capabilities to make debugging faster and easier. The debugger is highly configurable; it is easy to enable or disable the debugger features using the Project Options dialog.

6.1 Debugging Features Prior to Dynamic C 9

The following features are available prior to Dynamic C 9. They are summarized here, with links to more detailed descriptions.

- printf() - Display messages to the Stdio window (default) or redirect to a serial port. May also write to a file.
- Software Breakpoints - Stop execution, allow the available debug windows to be examined: Stack, Assembly, Dump and Register windows are always available.
- Single Stepping - Execute one C statement or one assembly statement. This is an extension of breakpoints, so again, the Stack, Assembly, Dump and Register windows are always available.
- Watch Expressions - Keep running track of any valid C expression in the application. Fly-over hints evaluate any watchable statement.
- Memory Dump - Displays blocks of raw values and their ASCII representation at any memory location (can also be sent to a file).
- MAP File - Shows a global view of the program: memory usage, mapping of functions, global/static data, parameters and local auto variables, macro listing and a function call graph.
- Assert Macro - This is a preventative measure, a kind of defensive programming that can be used to check assumptions before they are used in the code. This was introduced in Dynamic C 8.51.
- Blinking Lights - LEDs can be toggled to indicate a variety of conditions. This requires a signal line connected to an LED on the board.

6.2 Debugging Features Introduced in Dynamic C 9

Dynamic C 9 contains all the previous debugging tools and the additional ones listed here.

- Execution Trace - Traces at each statement, each function, or customer inserted points. Displays results in the Trace window. The options for execution tracing are configurable. This feature is disabled by default.

- Symbolic Stack Trace - Helps customers find out the path of the program at each single step or break point. By looking through the stack, it is possible to reconstruct the path and allow the customer to easily move backwards in the current call tree to get a better feeling for the current debugging context.

- Persistent Breakpoints - Persistent breakpoints mean the information is retained when transitioning back and forth from edit mode to debug mode and when a file is closed and re-opened.

- Enhanced Watch Expressions - The Watches window is now a tree structure capable of showing struct members. That is, all members of a structure become viewable as watch expressions when a structure is added, without having to add them each separately.

- Enhanced Memory Dumps - Changed data in the Memory Dump window is highlighted in reverse video or in customizable colors every time you single step in either C or assembly.

- Enhanced Mode Switching - Debug mode can be entered without a recompile and download. If the contents of the debugged program are edited, Dynamic C prompts for a recompile.

- Enhanced Stdio Window - The Stdio window is directly searchable.

Execution tracing is available with Dynamic C version 9. For more information on this debugging feature please see technical note TN253 “Execution Tracing.” All technical notes are available at rabbit.com.

6.3 Debugging Tools

This section describes the different tools available for debugging, including their pros and cons, as well as when you might want to use them, how to use them and an example of using them. The examples are suggestions and are not meant to be restrictive. While there may be some collaboration, bug hunting is largely a solitary sport, with different people using different tools and methods to solve the same problem.

6.3.1 printf()

The `printf()` function has always been available in Dynamic C, with output going to the Stdio window by default, and optionally to a file (by configuring the Stdio window contents to log to a file). The ability to redirect output to any one of the serial ports A, B, C or D was introduced in Dynamic C 7.25. In DC 8.51, serial ports E and F were added for the Rabbit 3000. See `Samples\stdio_serial.c` for instructions on how to use the serial port redirect. This feature is intended for debug purposes only.

The syntax for `printf()` is explained in detail in the *Dynamic C Function Reference Manual*, including a listing of allowable conversion characters.

Pros A `printf()` statement is quick, easy and sometimes all that is needed to nail down a problem.

You can use `#ifdef` directives to create levels of debugging information that can be conditionally compiled using macro definitions. This is a technique used by Rabbit engineers when developing Dynamic C libraries. In the library code you will see statements such as:

```
#ifdef LIBNAME_DEBUG
  printf(“Insert information here.\n”);
  ...
#endif
...
#ifdef LIBNAME_VERBOSE
  printf(“Insert more information.\n”);
...
#endif
```

By defining the above mentioned macro(s) you include the corresponding printf statements.

Cons The `printf()` function is so easy to use, it is easy to overuse. This can lead to a shortage of root memory. A solution to this that allows you to still have lots of printf strings is to place the strings in extended memory (xmem) using the keyword xdata and then call `printf()` with the conversion character “%ls.” An overuse of printf statements can also affect execution time.

Uses Use to check a program’s flow without stopping its execution.

`Example` There are numerous examples of using `printf()` in the programs provided in the Samples folder where you installed Dynamic C.

To display a string to the Stdio window place the following line of code in your application:

```
printf("Entering my_function().\n");
```

To do the same thing, but without using root memory:

```
xdata entering {“Entering my_function().”};
...
printf("%ls\n", entering);
```

6.3.2 Software Breakpoints

Software breakpoints have always been available in Dynamic C. They have been improved over several versions: the “Clear All Breakpoints” command was introduced in DC 7.10; the ability to set breakpoints in ISRs was introduced in DC 7.30; DC 9 introduces persistent breakpoints and the ability to set breakpoints in edit mode.

`Pros` Software breakpoints can be set on any C statement unless it is marked “nodebug” and in any “#asm debug” assembly block. Breakpoints let you run a program at full speed until the specified stopping point is reached. You can set multiple breakpoints in a program or even on the same line. They are easy to toggle on and off individually and can all be cleared with one command. You can choose whether to leave interrupts turned on (soft breakpoint) or not (hard breakpoint).

When stopped at a breakpoint, you can examine up-to-date contents in debug windows and choose other debugging features to employ, such as single stepping, dumping memory, fly-over watch expressions.

`Cons` To support large sector flash, breakpoint internals require that breakpoint overhead remain, even when the breakpoint has been toggled off. Recompile the program to remove this overhead.

When the debug keyword is added to an assembly block, relative jumps (which are limited to 128 bytes) may go out of range. If this happens, change the JR instruction to a JP instruction. Another solution is to embed a null C statement in the assembly code like so:

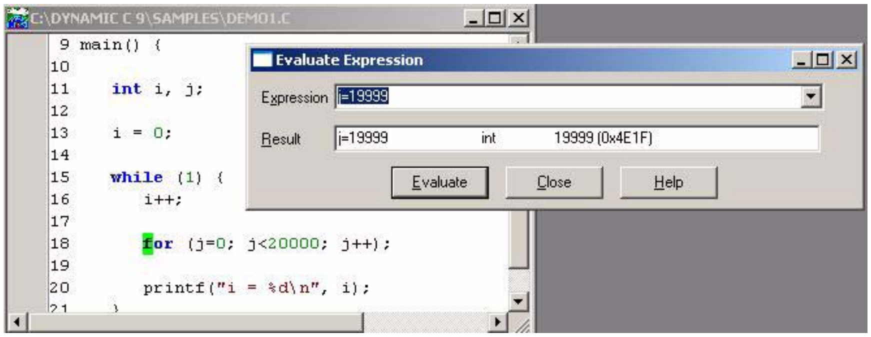
```
#asm
...
c ;        // Set a breakpoint on the semicolon
...
#endasm
```

`Uses` Use software breakpoints when you need to stop at a specified location to begin single stepping or to examine variables, memory locations or register values.

`Example` Open `Samples\Demo1.c`. If you are using DC 9, place the cursor on the word “for,” then press F2 to insert a breakpoint. Otherwise, press F5 to compile the program before setting the breakpoint. Now press F9. Every time you press F9 program execution will stop when it hits the start of the for loop. From here you can single step or look at a variety of information through debug windows.

For example, let us say there is a problem when you get to the limit of a `for` loop. You can use the Evaluate Expressions dialog to set the looping variable to a value that brings program execution to the exact spot that you want, as shown in this screenshot:

Figure 6.1 Altering the looping variable when stopped at a breakpoint



6.3.3 Single Stepping

Single stepping has always been available in Dynamic C. In version 7.10, the ability to single step on C statements with the Assembly window open was added.

`Pros` Single stepping allows you to closely supervise program execution at the source code level, either by C statement or assembly statement. This helps in tracing the logic of the program. You can single step any debuggable statement. Even Dynamic C library functions can be stepped into as long as they are not flagged as not available with the keyword `nodebug`.

`Cons` Single stepping is of limited use if interaction with an external device is being examined; an external device does not stop whatever it is doing just because the execution of the application has been restrained.

Also, single stepping can be very tedious if stepping through many instructions. Well-placed breakpoints might serve you better.

`Uses` Single stepping is typically used when you have isolated the problem and have stopped at the area of interest using a breakpoint.

`Example` To single step through a program instead of running at full execution speed, you must either set a breakpoint while in edit mode (if you have DC 9) or compile the program without running it.

To compile the program without running it, use the Compile menu option, the keyboard shortcut F5 or the toolbar menu button (pictured to the left of the Compile menu option).

F7, F8, Alt+F7 and Alt+F8 are the keyboard shortcuts for stepping through code. Use F7 if you want to step at the C statement level, but want to step into calls to debuggable functions. Use F8 instead if you want to step over function calls.

If the Assembly window is open, the stepping will be done by assembly instruction instead of by C statement if the feature “Enable instruction level single stepping” is checked on the Debugger tab of the Project Options dialog; otherwise, stepping is done by C statement regardless of the status of the Assembly window. If you have checked “Enable instruction level single stepping” but wish to continue to step by C statement when the Assembly window is open, use Alt+F7 or Alt+F8 instead of F7 or F8.

6.3.4 Watch Expressions

Like many other debugging features, watch expressions have been around since the beginning and have improved over time. Dynamic C 8.01 introduced the ability to evaluate watchable expressions using flyover hints. (The highlighted expression does not need to be set as a watch expression for evaluation in a flyover hint.) Dynamic C 9 introduced a new way of handling structures as watch expressions. Previously when you set a watch on a struct, its members had to be added separately and deliberately. Now they are set as watch expressions automatically with the addition of the struct.

Pros

Any valid C expression can be watched. Multiple expressions can be watched simultaneously. Once a watch is set on an expression, its value is updated in the Watches window whenever program execution is stopped.

The Watches window may be updated while the program is running (which will affect timing) by issuing the "Update Watch Window" command: use the Inspect menu, Ctrl+U or the toolbar menu button shown hereto update the Watches window.

You can use flyover hints to find out the value of any highlighted C expression when the program is stopped.

Cons

The scope of variables in watch expressions affects the value that is displayed in the Watches window. If the variable goes out of scope, its true value will not be displayed until it comes back into scope.

Keep in mind two additional things, which are not bad per se, but could be if they are used carelessly: Assignment statements in a watch expression will change the value of a variable every time watches are evaluated. Similarly, when a function call is in a watch expression, the function will run every time watches are evaluated.

Uses

Use a watch expression when the value of the expression is important to the behavior of the part of the program you are analyzing.

Example

Watch expressions can be used to evaluate complicated conditionals. A quick way to see this is to run the program `Samples\pong.c`. Set a breakpoint at this line

```
if (nx <= xl || nx >= xh)
```

within the function `pong()`. While the program is stopped, highlight the section of the expression you want evaluated. Use the watches flyover hint by hovering the cursor over the highlighted expression. It will be evaluated and the result displayed. You can see the values of, e.g., `nx` or `x1` or the result of the conditional expression `nx <= x1`, depending on what you highlight.

Keep in mind that when single stepping in assembly, the value of the watch expression may not be valid for variables located on the stack (all auto variables). This is because the debug kernel does not keep track of the pushesand pops that occur on the stack, and since watches of stack variables only make sense in the context of the pushes and pops that have happened, they will not always be accurate when assembly code is being single stepped.

6.3.5 Evaluate Expressions

The evaluate expression functionality was separated out from watch expressions in Dynamic C 8.01. It is a special case of a watch expression evaluation in that the evaluation takes place once, when the Evaluate button is clicked, not every time the Watches window is updated.

`Pros` Like watches, you can use the Evaluate Expression feature on any valid C expression. Multiple Evaluate Expression dialogs can be opened simultaneously.

`Cons` Can alter program data adversely if the change being made is not thought out properly

`Uses` This feature can be used to quickly and easily explore a variant of program flow.

`Example` Say you have an application that is supposed to treat the 100th iteration of a loop as a special case, but it does not. You do not want to set a breakpoint on the looping statement and hit F9 that many times, so instead you force the loop variable to equal 99 using the evaluate expression dialog. To do this compile the program without running it. Set a breakpoint at the start of the loop and then single step to get past the loop variable initialization. Open the Inspect menu and choose Evaluate Expression. Type in “j=99” and click on the Evaluate button. Now you are ready to start examining the program’s behavior.

6.3.6 Memory Dump

The Dump window was improved in Dynamic C 8.01 in several ways. For example, multiple dump windows can be active simultaneously, flyover hints make it easier to see the correct address, and three different types of dumps are allowed. Read the section titled, "Dump at Address," for more information on these and the other improvements made in version 8.01. In Dynamic C 9, dump windows were improved again. One improvement is that values that have changed are shown highlighted in reverse video or in customizable colors. Another improvement is that the value entered in the Memory Dump Setup dialog is the first address shown in the dump window. E.g., if you type in a logical address such as 74ec (all addresses are in hexadecimal), that will be the first address shown. Earlier versions of Dynamic C took a zero-based approach, meaning that the first address would be 74e0.

Pros Dump windows allow access to any memory location, beginning at any address. There are alignment options; the data can be viewed as bytes, words or double-words using a right-click menu.

Cons The Dump window does not contain symbolic information, which makes some information harder to decipher. There is the potential for increased debugging overhead if you open multiple dump windows and make them large.

Uses Use a dump window when you suspect memory is being corrupted. Or to watch string or numerical data manipulation proceed. String manipulation can easily cause memory corruption if you are not careful.

Example Consider the following code:

```
char my_array[10];
for (i=0; i<=10; i++){
   my_array[i] = 0xff;
}
```

If you do not have run-time checking of array indices enabled, this code will corrupt whatever is immediately following `my_array` in memory.

There is no run-time checking for string manipulation, so if you wrote something like the following in your application, memory would be corrupted when the null terminator for the string "1234" was written.

```
void foo () {
  int x;
  char str[4];

  x = 0xffff;
  strcpy(str,"1234");
}
```

Watching changes in a dump window will make the mistake more obvious in both of these situations, though in the former, turning on run-time checking for array indices in the Compiler tab of the Project Options dialog is easier.

6.3.7 MAP File

Map files have been generated for compiled programs since Dynamic C 7.02.

Pros
The map file is full of useful information. It contains,

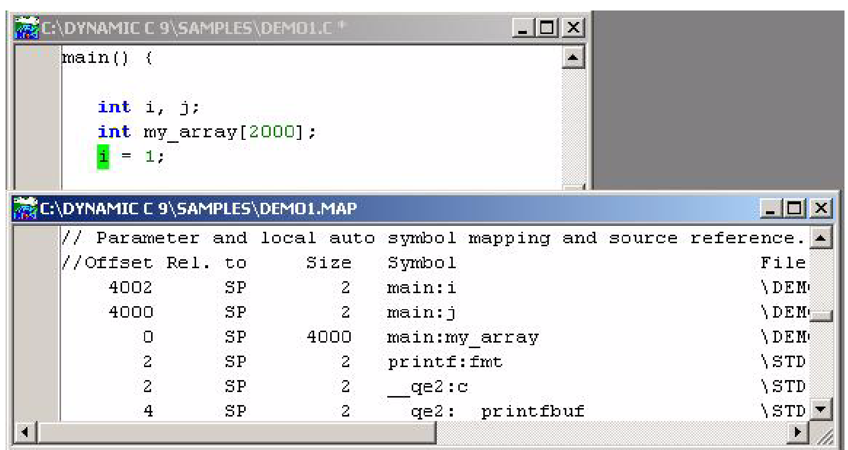
- location and size of code and data segments
- a list of all symbols used, their location, size and their file of origin
- a list of all macros used, their file of origin and the line number within that file where the macro is defined
- function call graph

A valid map file is produced after a successful compile, so it is available when a program crashes.

Cons
If the compile was not successful, for example you get a message that says you ran out of root code space, the map file will still be created, but will contain incomplete and possibly incorrect information.

Uses
Map files are useful when you want to gather more data or are trying to get a comprehensive overview of the program. A map file can help you make better use of memory in cases where you are running short or are experiencing stack overflow problems.

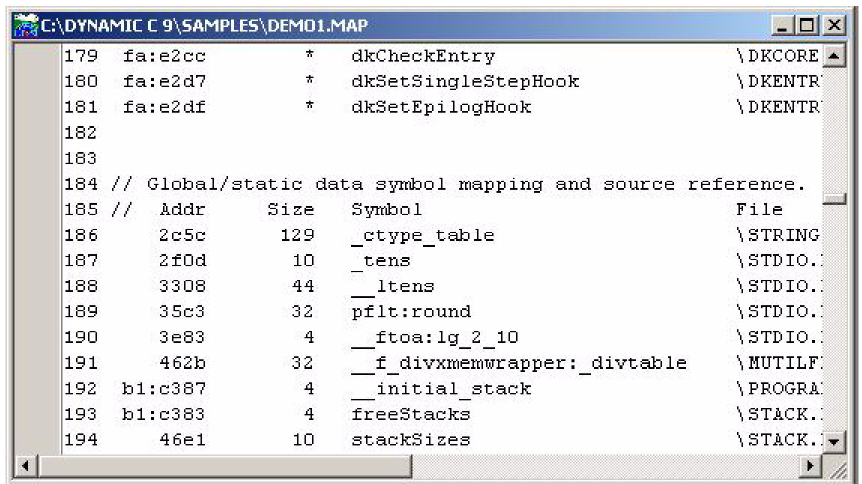
Example
Say you are pushing the limits of memory in your application and want to see where you can shave bytes. The map file contains sizes for all the data used in your program.The screen shot below shows some code and part of its map file. Maybe you meant to type "200" as the size for `my_array` and added a zero on the end by mistake. (This is a good place to mention that using hard-coded values is more prone to error than defining and using constants.)



Scanning the size column, the mistake jumps out at you more readily than looking at the code, maybe because you expect to see "200" and so your brain filters out the extra zero. For whatever reason, looking at the same information in a different format allows you to see more.

The size value for functions might not be accurate because it measures code distance. In other words, if a function spans a gap created with a *follows* action, the size reported for the function will be much greater than the actual number of bytes added to the program. The follows action is an advanced topic related to the subject of origin directives. See the *Rabbit 3000 Designer's Handbook* for a discussion of origin directives and action qualifiers.

The map file provides the logical and physical addresses of the program and its data. The screen shot below shows a small section of `demo1.map`. The left-most column shows line numbers, with addresses to their immediate right. Using the addresses we can reproduce the actions taken by the Memory Management Unit (MMU) of the Rabbit. Addresses with four-digits are both the logical and the physical address. That is because in the logical address space they are in the base segment, which always starts at zero in the physical address space. You can see this for yourself by opening two dump windows: one with a four-digit logical address and the second with that same four-digit number but with a leading zero, making it a physical address. The contents of the dump windows will be the same.



The addresses in the format xx:yyyy are physical addresses. For code xx is the XPC value, for data it is the value of DATASEG; yyyy is the PC value for both code and data. In the above map file you can see examples of both code and data addresses. Addresses in the format xx:yyyy are transformed by the MMU into a 5-digit physical address.

We will use the address fa:e64c to explain the actions of the MMU. It is really very simple if you can do hex arithmetic in your head or have a decent calculator. The MMU takes the XPC or DATASEG value, appends three zeros to it, then adds it to the PC value, like so:

```
fa000 + e64c = 10864c
```

A sixth digit in the result is ignored, leaving us with the value 0x0864c. This is the physical address. Again, you can check this in a couple of dump windows by typing in the 5-digit physical address for one window and the XPC:offset into another and seeing that the contents are the same.

6.3.8 Execution Trace

Execution tracing was introduced in Dynamic C 9. The program `Samples\demo4.c` demonstrates its use. Go to Section 3.4 for a full description of `demo4.c`.

There are basically three ways to toggle tracing during program execution. Two of them are similar: they require that tracing be enabled in the Debugger tab of the Project Options dialog and they do not trace in `nodebug` functions.

- GUI options: Opening the Inspect menu, you will see the “Stop Execution Tracing” and the “Start Execution Tracing” commands, along with their keyboard shortcuts and toolbar buttons. Use any of these methods to start and stop execution tracing while the program is running or while stopped at a breakpoint.
- `_TRACEON` and `_TRACEOFF`: Macros that are the equal to the GUI options

The third way does not require tracing to be enabled and it can be done in `nodebug` functions.

- `_TRACE`: A macro that causes itself, and only itself, to be traced.

Note that execution tracing is intrusive, slightly more so when the Trace window is open.

Pros The large amount of tracing information that may be saved on the host PC is available even if the program crashes. Tracing information fields can be turned on and off by the user on the Debugger tab of the Project Options dialog. The size of the trace buffer, which determines the number of trace entries, and whether to save the buffer to a file on program termination are also decided on the Debugger tab.

Cons Execution tracing alters the timing of a program because tracing information is inserted between every source statement that is executed. Therefore, execution tracing may not be useful in tracking down a timing related problem... it might even cause one.

Uses A good data gathering tool to use when you are not sure what is happening.

Example Say you have an application in which program flow deviates at some unknown point that is too tedious to detect by stepping. With execution tracing enabled, compile the program and click “Trace On” in the Inspect menu. Run the program and stop when the deviation is known or suspected to have occurred. Open the Trace window. You can now follow the execution at any point in the trace by double-clicking to source, or save to a file and grep for pertinent function calls or lines executed.

6.3.9 Symbolic Stack Trace

Dynamic C has always had the Stack window, but the Stack Trace window is new in Dynamic C 9. The old Stack window is still available to any compiled program, and being able to view the top 32 bytes of the stack could still be useful.

The Stack Trace window lets you see where you are and how you got there. It keeps a running depth value, telling you how many bytes have been pushed on the stack in the current program instance, or since the depth value reset button was clicked. The Stack Trace window only tracks stack-based variables, i.e., auto variables. The storage class for local variables can be either auto or static, specified through a modifier when the variable is declared or globally via the #class directive. Whatever the means, if a local variable is marked static it will not appear in the Stack Trace window.

Pros Provides a concise history of the call sequence and values of local variables and function arguments that led to the current breakpoint, all for a very small cost in execution time and BIOS memory.

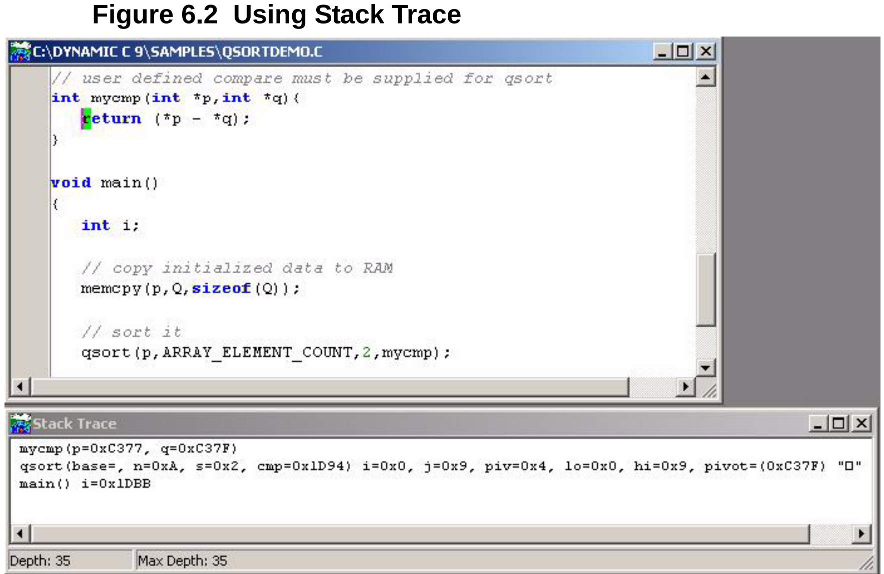
Cons Currently, the Stack Trace window can not trace the parameters and local variables in cofunctions. Also the contents of the window can not be saved after a program crash.

Uses Use stack tracing to capture the call sequence leading to a breakpoint and to see the values of functions arguments and local variables.

Example Say you have a function that is behaving badly. You can set a breakpoint in the function and use the Stack Trace window to examine the function call sequence. Examining the call sequence and the parameters being passed might give enough information to solve the problem.

The following screenshot shows an instance of `qsortdemo.c` and the Stack Trace window. Note that the call to `memcpy()` is not represented on the stack. The reason? Its stack activity had completed and program execution had returned to `main()` when the stack was traced at the breakpoint in the function `mycmp()`.

Figure 6.2 Using Stack Trace



6.3.10 Assert Macro

The assert macro was introduced in Dynamic C 8.51. The Dynamic C implementation of assert follows the ANSI standard for the NDEBUG macro, but differs in what the macro is defined to be so as to save code space (ANSI specifies that assert is defined as ((void)0) when NDEBUG is defined, but this generates a NOP in Dynamic C, so it is defined to be nothing).

Pros The assert macro is self-checking software. It lets you explicitly state something is true, and if it turns out to be false, the program terminates with an error message. At the time of this writing, this link contained an excellent write-up on the assert macro:

`http://www.embedded.com/story/OEG20010311S0021`

Cons Side effects can occur if the assert macro is not coded properly, e.g.,

```
assert(i=1)
```

will never trigger the assert and will change the value of the variable i; it should be coded as:

```
assert(i==1)
```

Uses Use the assert macro when you must make sure your assumption is accurate.

Example Check for a NULL pointer before using it.

```
void my_function (int * ptr){
   assert(ptr);
   ...
}
```

6.3.11 Miscellaneous Debugging Tools

Noted here are a number of other debugging tools to consider.

General Debug Windows

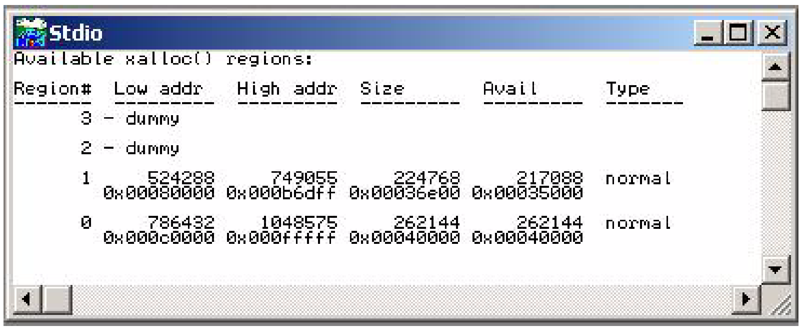
In addition to the debug windows we have discussed already, there are three other windows that are available when a program is compiled: the Assembly, Register and Stack windows. They are described in detail in Chapter 14, in the sections titled, Assembly (F10), Register Window and Stack (F12), respectively.

xalloc_stats()

Prints a table of physical addresses that are available for allocation in xmem via `xalloc()` calls. To display this information in the Stdio window, execute the statement:

```
xalloc_stats(0);
```

in your application or use Inspect | Evaluate Expression. The Stdio window will display something similar to the following:



A region is a contiguous piece of memory. Theoretically, up to four regions can exist; a region that is marked “dummy” is a region that does not exist. Each region is identified as “normal” or “BB RAM,” which refers to memory that is battery-backed.

SerialIO.exe

The utility `serialIO.exe` is located in `\Diagnostics\Serial_IO`. It is also in the file `SerialIO_1.zip`, available for download at the Rabbit website. This utility is a specialized terminal emulator program and comes with several diagnostic programs. The diagnostic programs test a variety of functionality, and allow the user to simulate some of the behavior of the Dynamic C download process.

The utility has a Help button that gives complete instructions for its use. The *Rabbit 3000 Designer's Handbook* in the chapter titled “Troubleshooting Tips for New Rabbit-Based Systems” explains some of the diagnostic programs that come with the serialIO utility. Understanding the information in this chapter will allow you to write your own diagnostic programs for the serialIO utility.

reset_demo.c

The sample program `Samples\reset_demo.c` demonstrates using the functions that check the reason for a reset: hard reset (power failure or pressing the reset button), soft reset (initiated by software), or a watchdog timeout.

Error Logging

Chapter 8, “Run-Time Errors,” describes the exception handling routine for run-time errors that is supplied with Dynamic C. The default handler may be replaced with a user-defined handler. Also error logging can be enabled by setting `ENABLE_ERROR_LOGGING` to 1 in the BIOS (prior to Dynamic C version 9.30) or in `ERRLOGCONFIG.LIB` (starting with DC 9.30). See Chapter 8 for more information.

Watchdogs

Ten virtual watchdogs are provided, in addition to the hardware watchdog(s) of the processor. Watchdogs, whether hardware or software, limit the amount of time a system is in an unknown state.

Virtual watchdogs are maintained by the Virtual Driver and described in Section 7.4.2. The sample program `Samples\VDRIVER\VIRT_WD.C` demonstrates the use of a virtual watchdog.

Compiler Options

The Compiler tab of the Project Options dialog contains several options that assist debugging. They are summarized here and fully documented starting on "Compiler Tab".

- List Files - When enabled, this option generates an assembly list file for each compile. The list file contains the same information and is in the same format as the contents of the Assembly window. List files can be very large.
- Run-Time Checking - Run-time checking of array indices and pointers are enabled by default.
- Type Checking - Compile-time checking of type options are enabled by default. There are three type checking options, labeled as: Prototype, Demotion and Pointer. Checking prototypes means that arguments passed in function calls are checked against the function prototype. Demotion checking means that the automatic conversion of a type to a smaller or less complex type is noted. Pointer checking refers to making sure pointers of different types being intermixed are cast properly.

  See the section titled, "Type Checking" on page 273 for more information.

Blinking Lights

Debugging software by toggling LEDs on and off might seem like a strange way to approach the problem, but there are a number of situations that might call for it. Maybe you just want to exercise the board hardware. Or, let us say you need to see if a certain piece of code was executed, but the board is disconnected from your computer and so you have no way of viewing printf output or using the other debugging tools. Or, maybe timing is an issue and directly toggling an LED with a call to `WrPortE()` or `BitWrPortE()` gives you the information you need without as much affect on timing.

The sample program `\Samples\LP3500\power.c` demonstrates how to use LEDs to communicate information.

6.4 Where to Look for Debugger Features

Debugger features are accessed from several different Dynamic C menus. The menu to look in depends on whether you want to enable, configure, view or use the debugger feature. This section identifies the various menus that deal with debugging. Table 6-1 summarizes the menus and debugging tools.

Table 6-1. Summary of Debug Tools and Menus

Name of Feature	Where Feature is Configured	Where Feature is Enabled	Where Feature is Toggled[a]
Execution Trace	Environment Options, Debug Windows tab Project Options, Debugger tab Right-click menu in the Trace window	Project Options, Debugger tab	Inspect Menu Programatically with macros
Symbolic Stack Trace	Environment Options, Debug Windows tab	Project Options, Debugger tab	Windows Menu
Software Breakpoints	Project Options, Debugger tab	Project Options, Debugger tab	Run Menu
Hardware Breakpoints	"Add | Edit Hardware breakpoint" dialog	Run menu's "Add/Edit Hardware Breakpoints" option	In "Add | Edit Hardware breakpoint" dialog, change check box, then click "Update" button
Single Stepping	No configuration options	Always enabled	Run Menu
Instruction Level Single Stepping	No configuration options	Project Options, Debugger tab	Run Menu
Watch Expressions	Environment Options, Debug Windows tab Project Options, Debugger tab	Project Options, Debugger tab	Inspect Menu
Evaluate Expression	No configuration options	This feature is enabled when Watch Expressions is enabled.	Inspect Menu
Map File	No configuration options	Always enabled	Automatically generated for compiled programs
Memory Dump	Environment Options, Debug Windows tab	Always enabled	Inspect Menu
Disassemble Code	Environment Options, Debug Windows tab	Always enabled	Inspect Menu
Assert Macro	Programatically	Programatically	Programatically
printf()	Programatically	Programatically	Programatically

Table 6-1. Summary of Debug Tools and Menus

Name of Feature	Where Feature is Configured	Where Feature is Enabled	Where Feature is Toggled[a]
Stdio, Stack and Register windows	Environment Options, Debug Windows tab	Always enabled	Windows Menu

a. Keyboard shortcuts and toolbar menu buttons are shown in the dropdown menus, along with their corresponding menu commands.

6.4.1 Run and Inspect Menus

The Run and Inspect menus are covered in detail in Section 14.2.5 and Section 14.2.6, respectively. These menus are where you can enable the use of several debugger features. The Run menu has options for toggling breakpoints and for single stepping. The Inspect menu has options for manipulating watch expressions, disassembling code and for dumping memory. For the most part, a debugger feature must be enabled before it can be selected in the Run or Inspect menus (or by its keyboard shortcut or toolbar menu button). Most debugger features are enabled by default in the Project Options dialog. The disassembled code and memory dump options are the exception, as they are always available to a compiled program.

6.4.2 Options Menu

From the Options menu in Dynamic C you can select Environment Options, Project Options or Toolbars, where you configure debug windows, enable debug tools or customize your toolbar buttons, respectively.

The Environment Options dialog has a tab labeled "Debug Windows." There are a number of configuration options available there. You can choose to have all or certain debug windows open automatically when a program compiles. You can choose font and color schemes for any debug window. More important than fonts and colors, you can configure most of the debug windows in ways specific to that window. For example, for the Assembly window you can alter which information fields are visible. See the section titled, "Debug Windows Tab" on page 262 for complete information on the specific options available for each window.

The Project Options dialog has a tab labeled "Debugger." This is where symbolic stack tracing, breakpoints, watch expressions and instruction level single stepping are enabled. These debugging tools must be enabled before they can be used. Some configuration options are also set on the Debugger tab. See the section titled, "Debugger Tab" on page 278, for complete information on the configuration options available on the Debugger tab.

The final menu selection on the Options menu is labeled, "Toolbars." This is where you choose the toolbars and the menu buttons that appear on the control bar. See the section titled, "Toolbars" on page 285, for instructions on customizing this area. Placing the menu buttons you use the most on the control bar is not really a debugging tool, but may make the task easier by offering some convenience.

6.4.3 Window Menu

The Window menu is where you can toggle display of debug windows. See Section 14.2.8 for more information. Another selection available from the Window menu is the Information window, which contains memory information and the status of the last compile. See "Information" on page 290 for full details.

6.5 Debug Strategies

Since bug-free code is a trade-off with time and money, we know that software has bugs[i]. This section discusses ways to minimize the occurrence of bugs and gives you some strategies for finding and eliminating them when they do occur.

6.5.1 Good Programming Practices

There is a big difference between "buggy code" and code that runs with near flawless precision. The latter program may have a bug, but it may be a relatively minor problem that only appears under abnormal circumstances. (This touches on the subject of testing, which are the actions taken specifically to find bugs, a larger discussion that is beyond the scope of this chapter.) This section discusses some time-tested methods that may improve your ability to write software with fewer bugs.

- **The Design:** The design is the solution to the problem that a program or function is supposed to solve. At a high level, the design is independent of the language that will be used in the implementation. Many questions must be asked and answered. What are the requirements, the boundaries, the special cases? These things are all captured in a well thought out design document. The design, written down, not just an idea floating in your head, should be rigorous, complete and detailed. There should be agreement and sign-off on the design before any coding takes place. The design underlies the code—it must come first. This is also the first part of creating full documentation.

- **Documentation**: Other documentation includes code comments and function description headers, which are specially formatted comments. Function description headers allow functions from libraries listed in `lib.dir` to be displayed in the Function Lookup option in Dynamic C's Help menu (or by using the keyboard shortcut Ctrl+H). See Section 4.24 for details on creating function description headers for user-defined library functions.

  Another way to comment code is by making the code self-documenting: Always choose descriptive names for functions, variables and macros. The brain only has so much memory capacity, why waste it up by requiring yourself to remember that cwl() is the function to call when you want to check the water level in yourfish tank; chk_h20_level(), for example, makes it easier to remember the function's purpose. Of course, you get very familiar with code while it is in development and so your brain transforms the letters "cwl" quite easily to the words "check water level." But years later when some esoteric bug appears and you have to dig into old code, you might be glad you took the time to type out some longer function names.

- **Modular Code**: If you have a function that checks the water level in the fish tank, don't have the same function check the temperature. Keep functions focused and as simple as possible.

---

i. For an account of what can happen when time and money constraints all but disappear, read "They Write the Right Stuff" by Charles Fishman.

- **Coding Standards**: The use of coding standards increases maintainability, portability and re-use of code. In Dynamic C libraries and sample programs[i] some of the standards are as follows:

  - Macros names are capitalized with an underscore separating words, e.g., `MY_MACRO`.
  - Function names start with a lowercase letter with an underscore or a capital letter separating words, e.g., `my_function()` or `myFunction()`.
  - Use parenthesis. Do not assume everyone has memorized the rules of precedence. E.g.,

```
y = a * b << c;     // this is legal
y = (a * b) << c;   // but this is more clear
```

  - Use consistent indenting. This increases readability of the code. Look in the Editor tab in the Environment Options dialog to turn on a feature that makes this automatic.
  - Use block comments (/*...*/) only for multiple line comments on the global level and line comments (//) inside functions, unless you really need to insert a long, multiple line comment. The reason for this is it is difficult to temporarily comment out sections of code using /*...*/ when debugging if the section being commented out has block comments, since block comments are not nestable.
  - Use Dynamic C code templates to minimize syntax errors and some typos. Look in the Code Templates tab in the Environment Options dialog to modify existing templates or create you own. Right click in an editor window and select Insert Code Template from the popup menu. This will bring up a scroll box containing all the available templates from which to choose.

- **Syntax Highlighting**: Many syntactic elements are visually enhanced with color or other text attributes by default. These elements are user-configurable from the Syntax Colors tab of the Environment Options dialog. This is more than mere lipstick. The visual representation of material can aid in or detract from understanding it, especially when the material is complex.

- **Revision Control System**: If your company has a code revision control systems in place, use it. In addition, when in development or testing stages, keep a known good copy of your program close at hand. That is, a compiles-and-runs-without-crashing copy of your program. Then if you make changes, improvements or whatever and then can't compile, you can go back to the known good copy.

6.5.2 Finding the Bug

When a program does not compile, or compiles, but when running behaves in unexpected ways, or perhaps worse, runs and then crashes, what do you do?

Compilation failures are caused by syntax errors. The compiler will generate messages to help you fix the problem. There may be a list of compiler error messages in the window that pops up. Fix the first one, then recompile. The other compile errors may disappear if they were not true syntax errors, but just the compiler being confused from the first syntax error.

During development, verify code as you progress. Develop code one function at a time. Do not wait until you are finished with your implementation before you attempt to compile and run it, unless it is a very short application. After a program is compiled, other types of bugs have a chance to reveal themselves. The rest of this section concentrates on how to find a bug.

i. Older libraries may not adhere strictly to these standards.

6.5.2.1 Reproduce the Problem

Keep an open mind. It might not be a bug in the software: you might have a bad cable connection, or something along those lines. Check and eliminate the easy things first. If you are reasonably sure that your hardware is in good working order, then it is time to debug the software.

Some bugs are consistent and are easy to reproduce, which means it will be easier to gather the information needed to solve the problem. Other bugs are more elusive. They might seem random, happening only on Wednesdays, or some other seemingly bizarre behavior. There are a number of reasons why a bug may be intermittent. Here are some common one:

- Memory corruption
    - uninitialized or incorrectly initialized pointers
    - buffer overflow
    - Stack overflow/underflow
- ISR modifying but not saving infrequently used register
- Interrupt latency
- Other borderline timing issues
- EMI

One of the difficulties of debugging is that the source of a bug and its effect may not appear closely related in the code. For example, if an array goes out of bounds and corrupts memory, it may not be a problem until much later when the corrupted memory is accessed.

6.5.2.2 Minimize the Failure Scenario

After you can reproduce the bug, create the shortest program possible that demonstrates the problem. Whatever the size of the code you are debugging, one way to minimize the failure scenario is a method called “binary search.” Basically, comment out half the code (more or less) and see which half of the program the bug is in. Repeat until the problem is isolated.

6.5.2.3 Other Things to Try

Get out of your cubicle. It is a well-known fact that there are times when simply walking over to a co-worker and explaining your problem can result in a solution. Probably because it is a form of data gathering. The more data you gather (up to a point), the more you know, and the more you know, the more your chances of figuring out the problem increase.

Stay in your cubicle. Log on and get involved in one of the online communities. There is a great Yahoo E-group dedicated to Rabbit and Dynamic C. Although Rabbit engineers will answer questions there, it is mostly the members of this group that solve problems for each other. To join this group go to:

```
http://tech.groups.yahoo.com/group/rabbit-semi/
```

Another good online source of information and help is the Rabbit bulletin board. Go to:

```
www.rabbit.com/support/bb/
```

If you are having trouble figuring out what is happening, remember to analyze the bug under various conditions. For example, run the program without the programming cable attached. Change the baud rate. Change the processor speed. Do bug symptoms change? If they do, you have more clues.

6.6 Reference to Other Debugging Information

There are many good references available. Here are a few of them:

- *Debugging Embedded Microprocessor Systems*, Stuart Ball
- *Writing Solid Code*, by Steve Macquire
- Websites: google, search on *debugging software*

At the time of this writing the following links provided some good information:

- http://www.embeddedstar.com/technicalpapers/content/d/embedded1494.html
- "They Write the Right Stuff" by Charles Fishman
  http://www.fastcompany.com/magazine/06/writestuff.html

7. 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

7.1 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.

7.2 Calling _GLOBAL_INIT()

`VdInit()` calls the function chain `_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.

7.3 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.

7.3.1 Example: Timing Loop

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.

```
#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 milliseconds to microseconds.
   printf("time to execute test code = %f us\n",T2*1000.0);
}
```

7.3.2 Example: Delay Loop

An important detail about MS_TIMER is that it overflows ("rolls over") approximately every 49 days, 17 hours. This behavior causes the following delay loop code to fail:

```
/* THIS CODE WILL FAIL!! */
endtime = MS_TIMER + delay;
while (MS_TIMER < endtime) {
   //do something
}
```

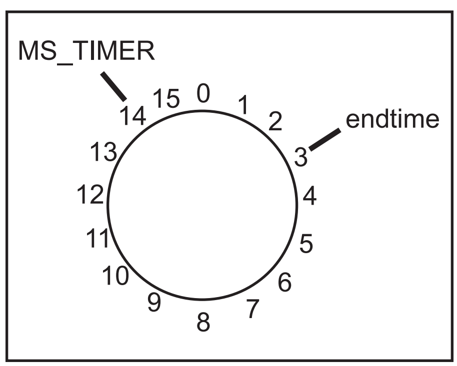
If "MS_TIMER + delay" overflows, this returns immediately. The correct way to code the delay loop so that an overflow of MS_TIMER does not break it, is this:

```
endtime = MS_TIMER + delay;
while ((long)MS_TIMER - endtime < 0) {
   //do something
}
```

The interval defined by the subtraction is always correct. This is true because the value of the interval is based on the values of MS_TIMER and "endtime" relative to one another, so the actual value of these variable does not matter.

One way to conceptualize why the second code snippet is always correct is to consider a number circle like the one in Figure 7.1. In this example, delay=5. Notice that the value chosen for MS_TIMER will "roll over" but that it is only when MS_TIMER equals or is greater than "endtime" that the while loop will evaluate to false.

Figure 7.1 "delay=5"



Another important point to consider is that the interval is cast to a signed number, which means that any number with the high bit set is negative. This is necessary in order for the interval to be less than zero when MS_TIMER is a large number.

7.4 Watchdog Timers

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

7.4.1 Hardware Watchdog

The Rabbit CPU has one built-in hardware watchdog timer[i]. The Virtual Driver hits the watchdog timer (WDT) 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. The watchdog 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.

7.4.2 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)` to hit a virtual watchdog or `VdReleaseWd(ID)` to deallocate it.

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.

---

i. Starting with the Rabbit 3000A, Rabbit microprocessors have secondary hardware watchdog timers. See the user's manual for your Rabbit processor for details, e.g., the *Rabbit 3000 Microprocessor User's Manual*.

7.5 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.

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. Table 8-1 is valid prior to Dynamic C version 9.30. Starting with DC 9.30, the file `errmsg.ini` located in the root directory of Dynamic C can be edited to add descriptions for user-defined run-time errors that will be displayed by Dynamic C should the error occur.

For example, if the following entry is made in `errmsg.ini`:

```
// My custom errors
800=My own run-time error message
```

Calling “exit(-800)” in an application or library will cause Dynamic C to report “My own run-time error message” in a message box.

Table 8-1. Dynamic C Error Types Ranges (prior to DC 9.30)

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

Please see Section 8.2 for information on replacing the default error handler with a custom one.

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. The wide range of error logs available with RabbitSys obviates the need for the default error logging described here.

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. Rabbit 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 112 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.

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 disabled.

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 using the programming cable, the RFU, or a RabbitLink board and starting the program again without the programming cable attached is a new deployment.

8.3.3 Configuration Macros

The macros listed below are defined at the top of `Bios/RabbitBios.c` prior to Dynamic C version 9.30 and in `Lib\..\BIOSLIB\errlogconfig.lib` thereafter. To change from the defaults you must edit the #define statement either in the BIOS or the configuration library, depending on your version of Dynamic C.

ENABLE_ERROR_LOGGING

Default: 0. Disables error logging. Changing this to "1" enables error logging.

ERRLOG_USE_REG_DUMP

Default: 1. Include a register dump in log entries. Changing this to zero 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 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 "1" includes 8 byte 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 physical address space of 1 MB on all Rabbit-based boards. Dynamic C has been verified to work with Rabbit-based boards with 4.5 MB of memory.

An on-chip memory management unit (MMU) translates 16-bit addresses to 20-bit memory addresses for Rabbit 2000- and 3000-based boards. 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.

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 the Rabbit processor used, the board type and which compilation options are selected. For example, enabling separate I&D space will affect the memory map.

Figure 9.1 Dynamic C Memory Mapping with a Rabbit 2000- or 3000-Based Board

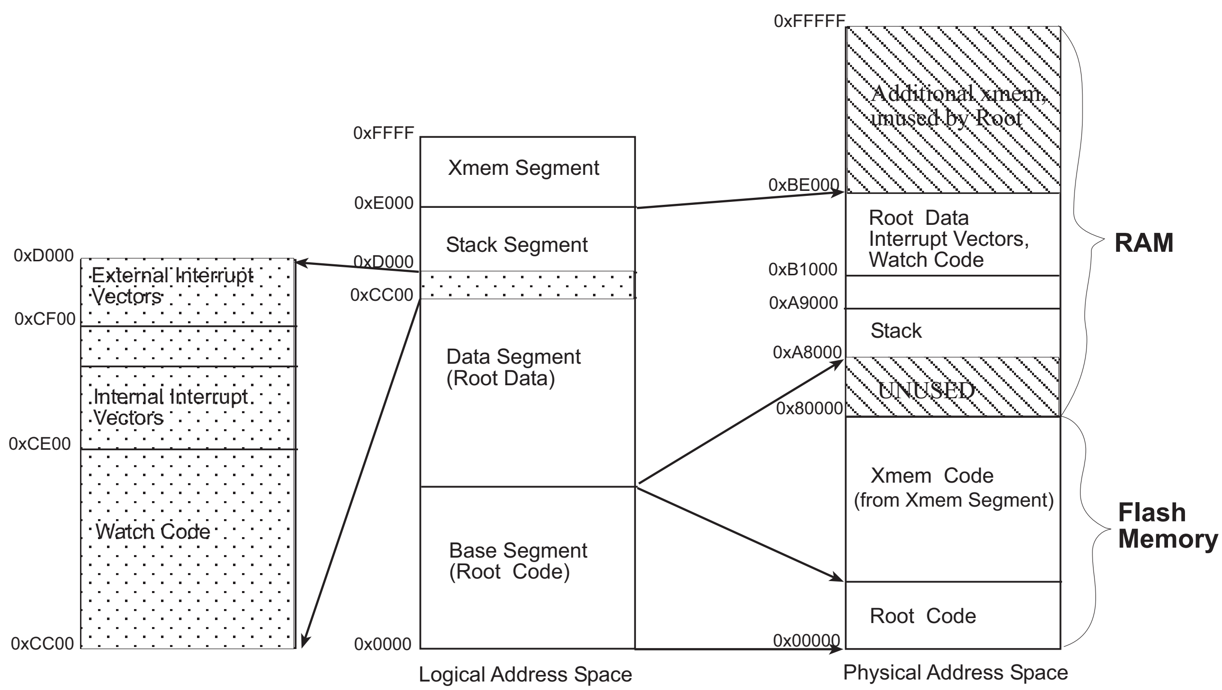


Figure 9.1 illustrates how the logical address space is divided and where code resides in physical memory. Both the static RAM and the flash memory are 128K in the diagram. Physical memory starts at address 0x00000 and flash memory is usually mapped to the same address. SRAM typically begins at address 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 Rabbit microprocessor user's manuals or designer's handbooks. You can also refer to one of our technical notes: TN202, "Rabbit Memory Management in a Nutshell." All of these documents are available at:

`www.rabbitsemiconductor.com/docs/`

9.1.2 Macro to Use Second Flash for Code

The macro `USE_2NDFLASH_CODE` can be uncommented in the file `sysconfig.lib` to cause the compiler to use a second available flash for xmem code.

9.2 Extended Memory Functions

A program can use many pages of extended memory (xmem). Under normal execution, code in xmem maps to the logical address region 0xE000 to 0xFFFF. Use the Dynamic C functions `root2xmem()`, `xmem2root()` and `xmem2xmem()`to move blocks of data between logical memory and physical memory.

9.3 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.

9.4 Dynamic Memory Allocation

Dynamic C 9 introduces the ability for an application to allocate a pool of memory at compile time for dynamic allocation and deallocation of fixed-size blocks at run time. A pool can be located in root or extended memory. Descriptions for all API functions for dynamic memory allocation are in the *Dynamic C Function Reference Manual*. Or use Function Lookup from the Help menu (or Ctrl+H) to gain quick access to the function descriptions from within Dynamic C.

Read the comments at the top of `\LIB\..\POOL.LIB` for a description of how to use dynamic memory allocation in Dynamic C.

10. File Systems

This chapter describes two separate file systems that can be used on Rabbit-based systems. The file system described in Section 10.1 works with all versions of Dynamic C for the Rabbit 2000 and 3000 microprocessors. The FAT file system described in Section 10.2 requires Dynamic C 8.51 or later. There have been several updates to the FAT file system to include additional flash devices.

FAT version 1.02 supports SPI-based serial flash devices. FAT versions 2.01 and 2.05 also support SPI-based serial flash devices and require Dynamic C 9.01 or later. FAT version 2.05 introduces support for NAND flash devices. FAT version 2.10 extends µC/OS-II compatibility to make the FAT API reentrant from multiple tasks. FAT version 2.13 adds support for SD cards and requires Dynamic C 10.21 or later. In all versions of the FAT, a battery-backed write-back cache reduces wear on the flash device and a round-robin cluster assignment helps spread the wear over its surface.

10.1 FS2

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 on Rabbit 2000- or 3000-based boards.

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
- all necessary run-time data to be reconstructed on power up

> **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.

10.1.1 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.

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

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.

10.1.1.1 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 per file to allow for moving data around.

Here is a formula you can use to determine how many bytes to allocate for the total file system (assuming all files are the same size):

```
Bytes = (Nbr_of_files * file_size * 1.14 ) + (Nbr_of_files * 128) +
        (2 * 128)
```

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.

10.1.1.2 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 macro `USE_2NDFLASH_CODE` is defined in the application by adding it to the Defines tab of the Project Options dialog box (for instructions see “Defines Tab” on page 281). This macro allocates the second flash to hold program code. The use of `USE_2NDFLASH_CODE` is not compatible with FS2.

10.1.1.3 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.

10.1.1.4 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 filesystem routines copy the old block into the new block adding in the user's new data. This has the effect of evening the wear if there is a reasonable turnover in the flash files. This goes for the data as well as the metadata.

10.1.1.5 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`.

10.1.1.6 Multitasking and FS2

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.

10.1.2 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

10.1.2.1 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 10.1.4 "Setting up and Partitioning the File System".

10.1.2.2 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 10.1.4.2.

FS_MAX_FILES

This macro is used to specify the maximum number of files 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_DISALLOW_GENERIC_FLASH

This macro is used to prevent FS2 from mistakenly attempting to recover a nonexistent file system on the "generic" (second) flash, or to prevent RAM corruption caused by _GetFlashID() when flash is not mapped into memory at all.

FS2_DISALLOW_PROGRAM_FLASH

This macro is used to prevent FS2 from mistakenly attempting to recover a nonexistent file system on the "program" (first) flash, or to prevent RAM corruption caused by _GetFlashID() when flash is not mapped into memory at all.

FS2_RAM_RESERVE

This 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.

This macro is defined in the BIOS prior to Dynamic C version 9.30 and in memconfig.lib thereafter.

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. If not defined in an application, it defaults to zero, meaning that the first flash is not used by FS2. The actual amount of flash used by FS2 is determined by the minimum of this macro and XMEM_RESERVE_SIZE.

XMEM_RESERVE_SIZE

This 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 as 0x0000. Memory set aside with XMEM_RESERVE_SIZE will NOT be available for xmem code.

This macro is defined in the BIOS prior to Dynamic C version 9.30 and in memconfig.lib thereafter.

10.1.2.3 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) 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[i] 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.

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().

i. For a description of logical extents please see Section 10.1.4.2, “Logical Extents (LX),” on page 128.

10.1.3 File System API Functions

These functions are defined in FS2.LIB. For more information please see the *Dynamic C Function Reference Manual* or from within Dynamic C you can use the Function Lookup feature, with its convenient Ctrl+H shortcut that will take you directly to a function's description if the cursor is on its name in the active edit window.

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.

10.1.3.1 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.

10.1.4 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).

10.1.4.1 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.

10.1.4.2 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 controlled by the file system containing data that tracks 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.

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; doing so ensures 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()`.

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.

10.1.4.3 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 $\geq$ 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 so it will 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.

10.1.5 File Identifiers

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

10.1.5.1 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.

10.1.5.2 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 TCP/IP User's Manual, Vol 2*, or use keyboard shortcut 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.

10.1.6 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.

10.2 FAT File System

Dynamic C 8.51 introduced a FAT (File Allocation Table) file system. The small footprint of this well-defined industry-standard file system makes it ideal for embedded systems. The Dynamic C implementation of FAT has a directory structure that can be accessed with either Unix or DOS style paths. The standard directory structure allows monitoring, logging, Web browsing, and FTP updates of files.

The FAT filesystem is included with Dynamic C starting with version 9.60. In earlier versions of Dynamic C, FAT was sold separately.

FAT version 1.02 supports SPI-based serial flash devices. FAT versions 2.01 and 2.05 also support SPI-based serial flash devices and require Dynamic C 9.01 or later. FAT version 2.05 introduces support for NAND flash devices. FAT version 2.10 extends µC/OS-II compatibility to make the FAT API reentrant from multiple tasks. FAT version 2.13 adds support for SD cards and requires Dynamic C 9.60 or later. In all versions of the FAT, a battery-backed write-back cache reduces wear on the flash device and a round-robin cluster assignment helps spread the wear over its surface.

Please be sure check the Rabbit website for software patches and updates to Dynamic C, the FAT filessystem, and for your specific hardware:

`www.rabbit.com/support/downloads/`

The FAT library can be used in either *blocking* or *non-blocking* mode and supports both FAT12 and FAT16. (See Section 10.2.5.3.1 for more information on these FAT types.)

Let's define some terms before continuing.

- A *device* is a single physical hardware item such as a hard drive, a serial flash or a NAND flash. E.g., one serial flash is a single *device*. The device, in turn, can host one to four *partitions*.
- A *partition* is a range of logical sectors on a device. A real-world example of a partition is what is commonly known as the C drive on a PC.
- A *driver* is the software interface that handles the hardware-specific aspects of any communication to or from the device.
- *Blocking* is a term that describes a function's behavior in regards to completion of the requested task. A blocking function will not return until it has completely finished its task. In contrast, a *non-blocking* function will return to its calling function before the task is finished if it is waiting for something. A non-blocking function can return a code that indicates it is not finished and should be called again. Used in conjunction with cooperative multitasking, non-blocking functions allow other processes to proceed while waiting for hardware resources to finish or become available.

Operations performed by the Dynamic C FAT implementation are:

- Formatting and partitioning of devices
- Formatting partitions
- File operations: create, open, close, delete, seek, read and write
- Directory[i] operations: create, read and delete
- Labels: create and delete

i. We use the terms *directory* and *subdirectory* somewhat interchangeably. The exception is the root directory—it is never called a subdirectory. Any directory below the root directory may be referred to as a directory or a subdirectory.

10.2.1 Overview of FAT Documentation

A sample program is reviewed in Section 10.2.2. Two additional sample programs, one for use with the blocking mode of the FAT and the other for use with the non-blocking mode are described in Section 10.2.3. Then Section 10.2.4 gives detailed descriptions of the various FAT file system functions (formatting, opening, reading, writing, and closing). Short, focused examples accompany each description. There is some general information about FAT file systems and also some web links for further study in Section 10.2.5.

> **NOTE:** All error codes returned from the Dynamic C FAT file system are defined in `LIB/.../FILESYSTEM/ERRNO.LIB`.

10.2.2 Running Your First FAT Sample Program

To run FAT samples, you need a Rabbit-based board with a supported flash type, such as the SPI-based serial flash device available on the RCM3300 or the RCM3700. FAT versions 2.01 and 2.05 require Dynamic C 9.01 or later. FAT version 2.05 extends the list of supported flash types to include NAND flash devices, such as those on the RCM3360 and 3370. FAT version 2.13 requires Dynamic C 9.60 or later and adds support for SD cards, available on the RCM3900 and 3910.

The board must be powered up and connected to a serial port on your PC through the programming cable to download a sample program.

In this section we look at `fat_create.c`, which demonstrate the basic use of the FAT file system. If you are using a serial or NAND flash device that has not been formatted or a removable device that was not formatted using Dynamic C, you must run `Samples\FileSystem\Fmt_Device.c` before you can run any other sample FAT program. The program, `Fmt_Device.c`, creates the default configuration of one partition that takes up the entire device.

If you are using an SD card, run `Fmt_Device.c` to remove the factory FAT32 partition and create a FAT16 partition. Be aware that although multiple partitions are possible on removable cards, most PC's will not support cards formatted in this fashion.

If you are using a removable NAND flash (XD cards), running `Fmt_Device.c` causes the device to no longer be usable without the Rabbit-based board or the Rabbit USB Reader for XD cards. Insert the NAND flash device into a USB-based flash card reader and format it to regain this usability. Note that this will only work if you have ***not*** defined the macro `NFLASH_CANERASEBADBLOCKS`. Defining this macro in a running application destroys proprietary information on the first block of the device, making it difficult to regain the usability of the NAND device when used without the Rabbit-based board.

If you are using FAT version 2.01 or later, you do not have to run `Fmt_Device.c` if you initialize the FAT file system with a call to `fat_AutoMount()` instead of `fat_Init()`. The function `fat_AutoMount()` can optionally format the device if it is unformatted; however, `fat_AutoMount()` will not erase and overwrite a factory-formatted removable device such as an SD card. If you are using an SD card, run `Fmt_Device.c` or erase the first three pages with the appropriate flash utitity (`sdflash_inspect.c` or `nflash_inspect.c`).

After the device has been formatted, open `Samples\FileSystem\fat_create.c`. Compile and run the program by pressing function key F9.

In a nutshell, `fat_create.c` initializes FAT, then creates a file, writes "Hello world!" to it, and then closes the file. The file is re-opened and the file is read, which displays "Hello world!" in the Dynamic C Stdio window. Understanding this sample will make writing your own FAT application easier.

The sample program has been broken into two functional parts for the purpose of discussion. The first part deals with getting the file system up and running. The second part is a description of writing and reading files.

10.2.2.1 Bringing Up the File System

We will look at the first part of the code as a whole, and then explain some of its details.

File Name: `Samples\FileSystem\fat_create.c`

```
#define FAT_BLOCK                               // use blocking mode
#use "fat.lib"                                  // of FAT library

FATfile my_file;                                // get file handle
char buf[128];                                  // 128 byte buffer for read/write of file

int main(){
   int i;
   int rc;                                      // Check return codes from FAT API
   long prealloc;                               // Used if the file needs to be created.
   fat_part *first_part;                        // Use the first mounted FAT partition.

   rc = fat_AutoMount( FDDF_USE_DEFAULT );

   first_part = NULL;
   for(i=0;i < num_fat_devices * FAT_MAX_PARTITIONS; ++i)
   {                                            // Find the first mounted partition
      if ((first_part = fat_part_mounted[i]) != NULL) {
         break;                                 // Found mounted partition, so use it
      }
   }

   if (first_part == NULL) {                    // Check if mounted partition was found
      rc = (rc < 0) ? rc : -ENOPART;            // None found, set rc to a FAT error code
   } else{
      printf("fat_AutoMount() succeeded with return code %d.\n", rc);
      rc = 0;                                   // Found partition; ignore error, if any
   }

   if (rc < 0){                                 // negative values indicate error
      if (rc == -EUNFORMAT)
         printf("Device not Formatted, Please run Fmt_Device.c\n");
      else
         printf("fat_AutoMount() failed with return code %d.\n", rc);
      exit(1);
   }   // OK, file system exists and is ready to access. Let's create a file.
```

The first two statements:

```
#define FAT_BLOCK
#use "fat.lib"
```

cause the FAT library to be used in blocking mode.

FAT version 2.01 introduces a configuration library that chooses initialization settings based on the board type.The statement `#use “fat.lib”` brings in this configuration library, which in turn brings in the appropriate device driver library. The following table lists the device drivers that are available in the different FAT versions.

Table 11.

FAT Version	Device Driver
1.02, 2.01	`sflash_fat.lib`
2.05	`sflash_fat.lib` `nflash_fat.lib`

2.13	`sflash_fat.lib`
	`nflash_fat.lib`
	`SD_fat.lib`

Defining the macro `_DRIVER_CUSTOM` notifies `fat_config.lib` that a custom driver or hardware configuration is being used. For more information on how this works, see Section 10.2.5

Next some static variables are declared: a file structure to be used as a handle to the file that will be created and a buffer that will be used for reading and writing the file.

Now we are in `main()`. First there are some variable declarations: the integer `rc` is for the code returned by the FAT API functions. This code should always be checked, and *must* be checked if the non-blocking mode of the FAT is used. The descriptions for each function list possible return codes.

The variable `prealloc` stores the number of bytes to reserve on the device for use by a specific file. These clusters are attached to the file and are not available for use by any other files. This has some advantages and disadvantages. The obvious disadvantage is that it uses up space on the device. Some advantages are that having space reserved means that a log file, for instance, could have a portion of the drive set aside for its use only. Another advantage is that if you are transferring a known amount of information to a file, pre-allocation not only sets aside the space so you know you will not get half way through and run out, but it also makes the writing process a little faster as the allocation of clusters has already been dealt with so there is no need to spend time getting another cluster.

This feature should be used with care as pre-allocated clusters do not show up on directory listings until data is actually written to them, even though they have locked up space on the device. The only way to get unused pre-allocated clusters back is to delete the file to which they are attached, or use the `fat_truncate()` or `fat_split()` functions to trim or split the file. In the case of `fat_split()`, the pre-allocated space is not freed, but rather attached to the new file created in the split.

Lastly, a pointer to a partition structure is declared with the statement:

```
fat_part *first_part;
```

This pointer will be used as a handle to an active partition. (The `fat_part` structure and other data structures needed by the FAT file system are discussed in `fat_AutoMount()`.) The partition pointer will be passed to API functions, such as `fat_open()`.

Now a call is made to `fat_AutoMount()`. This function was introduced in FAT version 2.01 as a replacement for `fat_Init()`. Whereas `fat_Init()` can do all the things necessary to ready the first partition on the first device for use, it is limited to that. The function `fat_AutoMount()` is more flexible because it uses data from the configuration file `fat_config.lib` to identify FAT partitions and to optionally ready them for use, depending on the flags parameter that is passed to it. The flags parameter is described in the function description for `fat_AutoMount()`.

For this sample program, we are interested in the first usable FAT partition. The `for` loop after the call to `fat_AutoMount()` finds the partition, if one is available.

The `for` loop allows us to check every possible partition by using `num_fat_devices`, which is the number of configured devices, and then multiplying the configured devices by the maximum number of allowable partitions on a device, which is four. The `for` loop also makes use of `fat_part_mounted`, an array of pointers to partition structures that is populated by the `fat_autoMount()` call.

10.2.2.2 Using the File System

The rest of `fat_create.c` demonstrates how to use the file system once it is up and running.

File Name: `Samples\FileSystem\fat_create.c`

```
   prealloc = 0;

   rc = fat_Open( first_part, "HELLO.TXT", FAT_FILE, FAT_CREATE,
                  &my_file, &prealloc );

   if (rc < 0) {
      printf("fat_Open() failed with return code %d\n", rc);
      exit(1);
   }
   rc = fat_Write( &my_file, "Hello, world!\r\n", 15 );

   if (rc < 0) {
      printf("fat_Write() failed with return code %d\n", rc);
      exit(1);
   }
   rc = fat_Close(&my_file);
   if (rc < 0) {
      printf("fat_Close() failed with return code %d\n", rc);
   }

   rc = fat_Open( first_part, "HELLO.TXT",FAT_FILE, 0, &my_file,
                  NULL);

   if (rc < 0) {
      printf("fat_Open() (for read) failed, return code %d\n", rc);
      exit(1);
   }
   rc = fat_Read( &my_file, buf, sizeof(buf));
   if (rc < 0) {
      printf("fat_Read() failed with return code %d\n", rc);
   }
   else {
      printf("Read %d bytes:\n", rc);
      printf("%*.*s", rc, rc, buf);  // Print a string which is not NULL terminated
      printf("\n");
   }
   fat_UnmountDevice( first_part->dev );
   printf("All OK.\n");
   return 0;
}
```

The call to `fat_Open()` creates a file in the root directory and names it `HELLO.TXT`. A file must be opened before you can write or read it.

```
rc = fat_Open(first_part, "HELLO.TXT", FAT_FILE, FAT_CREATE,
              &my_file, &prealloc);
```

The parameters are as follows:

- `first_part` points to the partition structure initialized by `fat_AutoMount()`.
- `"HELLO.TXT"` is the file name, and is always an absolute path name relative to the root directory. All paths in Dynamic C must specify the full directory path explicitly.
- `FAT_FILE` identifies the type of object, in this case a file. Use `FAT_DIR` to open a directory.
- `FAT_CREATE` creates the file if it does not exist. If the file does exist, it will be opened, and the position pointer will be set to the start of the file. If you write to the file without moving the position pointer, you will overwrite existing data.

  Use `FAT_OPEN` instead of `FAT_CREATE` if the file or directory should already exist. If the file does not exist, you will get an `-ENOENT` error.

  Use `FAT_MUST_CREATE` if you know the file does not exist. This is a fail-safe way to avoid opening and overwriting an existing file since an `-EEXIST` error is returned if you attempt to create a file that already exists.
- `&my_file` is a file handle that points to an available file structure. It will be used for this file until the file is closed.
- `&prealloc` points to the number of bytes to allocate for the file. You do not want to pre-allocate any more than the minimum number of bytes necessary for storage, and so `prealloc` was set to 0. You could also use `NULL` instead of `prealloc` and `prealloc = 0`.

Next, the sample program writes the data "Hello, world!\r\n" to the file.

```
fat_Write( &my_file, "Hello, world!\r\n", 15 );
```

The parameters are as follows:

- `&my_file` is a pointer to the file handle opened by `fat_Open()`.
- `“Hello, world!\r\n”` is the data written to the file. Note that `\r\n` (carriage return, line feed) appears at the end of the string in the call. This is essentially a FAT (or really, DOS) convention for text files. It is good practice to use the standard line-end conventions. (If you just use `\n`, the file will read just fine on Unix systems, but some DOS-based programs may have difficulties.)
- `15` is the number of characters to write. Be sure to select this number with care since a value that is too small will result in your data being truncated, and a value that is too large will append any data that already exists beyond your new data.

The file is closed to release the file handle to allow it to be used to identify a different file.

```
rc = fat_Close( &my_file );
```

The parameter `&my_file` is a handle to the file to be closed. Remember to check for any return code from `fat_Close()` since an error return code may indicate the loss of data.

The file must be opened for any further work, even though `&my_file` may still reference the desired file. The file must be open to be active, so we call `fat_Open()` again. Now the file can be read.

```
rc = fat_Read( &my_file, buf, sizeof(buf));
```

The function `fat_Read()` returns the number of characters actually read. The parameters are as follows:

- `&my_file` is a handle to the file to be read.
- `buf` is a buffer for reading/writing the file that was defined at the beginning of the program.
- `sizeof(buf)` is the number of bytes to be read into `buf`. It does not have to be the full size of the buffer

Characters are read beginning at the current position of the file. (The file position can be changed with the `fat_Seek()` function.) If the file contains fewer than `sizeof(buf)` characters from the current position to the end-of-file marker (EOF), the transfer will stop at the EOF. If the file position is already at EOF, 0 is returned. The maximum number of characters read is 32767 bytes per call.

The file can now be closed. Call `fat_UnmountDevice()`[i] rather than simply calling `fat_Close()` to ensure that any data stored in cache will be written to the device. With a write-back cache, writes are delayed until either:

- all cache buffers are full and a new FAT read request requires a "dirty" cache buffer to be written out before the read can take place, or
- cache buffers for a partition or a device are being flushed due to an unmount call or explicit flush call.

Calling `fat_UnmountDevice()` will close all open files and unmount all mounted FAT partitions. This is the safest way to shut down a device. The parameter `first_part->dev` is a handle to the device to be unmounted.

```
fat_UnmountDevice( first_part->dev );
```

> **NOTE:** A removable device must be unmounted in order to flush its data before removal. Failure to unmount any partition on a device that has been written to could corrupt the file system.

i. Call fat_UnmountPartition() when using a FAT version prior to v2.06.

10.2.3 More Sample Programs

This section studies blocking sample `FAT_SHELL.C` and non-blocking sample `FAT_NB_Costate.c` More sample programs are in the Dynamic C folder `Samples\FileSystem\FAT`. For example, there is `udppages.c`, an application that shows how to combine HTTP, FTP and zserver functionality to create web content than can be updated via FTP.

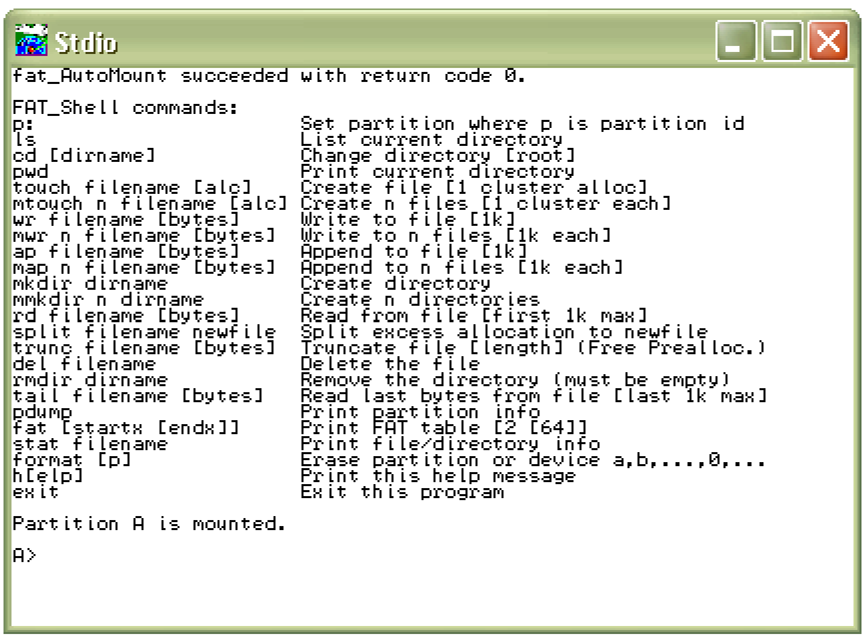
As described in Section 10.2.2, you will need a target board or core module with a supported flash device, powered up and connected to a serial port on your PC through the programming cable.

10.2.3.1 Blocking Sample

The sample program `Samples\FileSystem\FAT_SHELL.C` allows you to use the FAT library by entering DOS-like or Unix-like commands. To run this sample, open Dynamic C, then open `FAT_SHELL.C`. Compile and run `FAT_SHELL.C` by pressing F9. If the flash device has not been formatted and partitioned, `FAT_SHELL.C` will format and partition the flash device, and then you will be prompted to run `FAT_SHELL.C` again (just press F9 when prompted). A display similar to the one shown in Figure 1 will open in the Dynamic C Stdio window.

Optional parameters are denoted by the square braces [and] following the command name. The [alc] after “touch” and “mtouch” indicates an optional allocation amount in bytes. The square braces in the description indicate the default value that will be used if the optional parameter is not given.

Figure 1. List of Shell Commands



You can type “`h`” and press enter at any time to display the FAT shell commands.

In the following examples the commands that you enter are shown in boldface type. The response from the shell program is shown in regular typeface.

```
> ls
Listing '' (dir length 16384)
   hello.txt rhsvdA len=15       clust=2
>
```

This shows the HELLO.TXT file that was created using the FAT_CREATE.C sample program. The file length is 15 bytes. Cluster 2 has been allocated for this file. The "ls" command will display up to the first six clusters allocated to a file.

The flag, rhsvdA, displays the file or directory attributes, with upper case indicating that the attribute is turned on and lower case indicating that the attribute is turned off. In this example, the archive bit is turned on and all other attributes are turned off.

These are the six attributes:

r - read-only	**v** - volume label
h - hidden file	**d** - directory
s - system	**a** - archive

To create a directory named DIR1, do the following:

```
> mkdir dir1
Directory '/dir1' created with 1024 bytes
>
```

This shows that DIR1 was created, and is 1024 bytes (size may vary by flash type).

Now, select DIR1:

```
> cd dir1
PWD = '/dir1'
>
```

Add a new file called RABBIT.TXT:

```
> touch rabbit.txt
File '/dir1/rabbit.txt' created with 1024 bytes
>
```

Note that the file name was appended to the current directory. Now we can write to RABBIT.TXT. The shell program has predetermined characters to write, and does not allow you to enter your own data.

```
> wr rabbit.txt
File '/dir1/rabbit.txt' written with 1024 bytes out of 1024
>
```

To see what was written, use the “rd” command.

```
> rd rabbit.txt

rabbit.txt 1024 The quick brown fox jumps over the lazy dog
rabbit.txt 1024 The quick brown fox jumps over the lazy dog
.
.
rab

Read 1024 bytes out of 1024
>
```

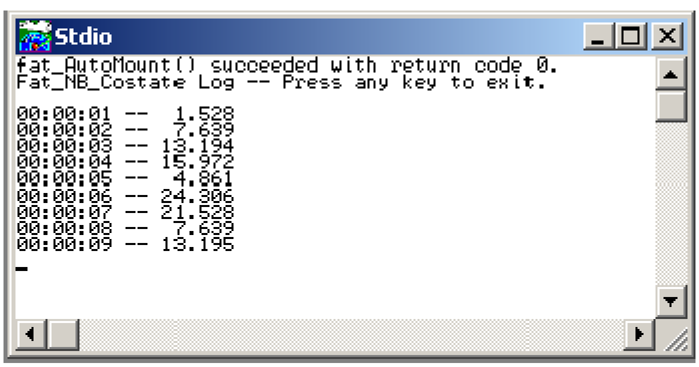
10.2.3.2 Non-Blocking Sample

To use the FAT file system in non-blocking mode, do not include the statement `#define FAT_BLOCK` in your application. The program interface to the library is the same as the blocking version, with the exception of the return code `-EBUSY` from many of the API functions.

The sample program `Fat_NB_Costate.c` in the `Samples\FileSystem` folder is an example of a non-blocking application. To view the code in its entirety, open it in Dynamic C. The following discussion will not examine every line of code, but will focus on what shows the non-blocking nature of the FAT library and how the application takes advantage of it.

Run Fat_`NB_Costate.c` and after 10 seconds the Stdio window will show something similar to the following:

Figure 2. Screen Shot of Fat_NB_Costate.c Running



Each line is an entry into a file that is stored in the FAT file system. The file is appended once every second and read and displayed once every ten seconds. In addition to the file system use and the screen output, if you are using an RCM3300, RCM3700 or PowerCore FLEX development board, the application blinks the LED on your board.

The code preceding `main()` brings in the required library and declares the file structure. And, as expected, there is no `#define` for the macro `FAT_BLOCK`. At the start of `main()` some system variable are created and initialized. This is followed by the code to bring up the FAT file system, which is similar to what we examined in Section 10.2.2.1 when looking at `fat_create.c`, with two essential differences. One, since we have initialized the FAT to be in non-blocking and we are making some calls to FAT functions that must return before we can continue, we must wait for the return.

A while loop accomplishes our goal of blocking on the function call until it returns something other than busy.

```
while ((rc = fat_Open( first_part, name, FAT_FILE, FAT_MUST_CREATE,
   &file, &alloc)) == -EBUSY);
```

The second difference from our earlier sample is the statement right before fat_Open():

```
file.state = 0;
```

This is required before opening a file when using non-blocking mode in order to indicate that the file is not in use. Only do this once. After you have opened the file, do not alter the contents of the file structure.

If fat_Open() succeeds we can go into the non-blocking section of the program: three costatements inside an endless while loop. The benefit of using the non-blocking mode of the FAT file system is realized when using costatements, an extension of Dynamic C that implements cooperative multitasking. Instead of waiting while a function finishes its execution, the application can accomplish other tasks.

10.2.3.2.1 Costatement that Writes a File

The first costate is named putdata. It waits for one second and then creates a string to timestamp the entry of a randomly generated number that is then appended to a file.

```
while (1){
   costate putdata always_on
   {
      waitfor (DelaySec(1));       // Wait for one second to elapse
```

Note that the always_on keyword is used. This is required when using a named costatement to force it to execute every time it is encountered in the execution thread (unless it is made inactive by a call to CoPause()).

It is easy to suspend execution within a costate by using the waitfor keyword. The costate will relinquish control if the argument to waitfor (in this case a call to DelaySec()) evaluates to FALSE. The next time the execution thread reaches putdata, waitfor will be called again. This will go on until DelaySec() returns TRUE, i.e., when one second has elapsed from the time DelaySec() was first called from within waitfor.

After the one second delay, the string to write to the file is placed in a buffer and a looping variable and position pointer are initialized.

```
sprintf(obuf, "%02d:%02d:%02d -- %6.3f \n", h, m, s, (25.0 * rand()));
ocount = 0;
optr = obuf;
```

Before the buffer contents can be written to a file in the FAT file system, we must ensure that no collisions occur since there is another costate that will attempt to read the file every ten seconds. A file can not be read from and written to at the same time. In the following code the waitfor keyword is used with the global variable filestate (defined at the top of the application) to implement a locking mechanism. As soon as the file becomes available for putdata, it is marked unavailable for showdata.

```
waitfor (filestate == 0);       // Wait until file is available
filestate = 1;                  // Show file is being updated
```

The next block of code appends the latest entry into the file that was opened at the start of the application.

```
   while (ocount < REC_LEN){    // Loop until entire record is written
      waitfor((rc = fat_Write(&file, optr, REC_LEN - ocount))!= -EBUSY);
      if (rc < 0){
         printf("fat_Write: rc = %d\n",rc);
         while ((rc = fat_UnmountDevice(first_part->dev)) == -EBUSY);
         return rc;
      }
      optr += rc;               // Move output pointer
      ocount += rc;             // Add number of characters written
   }
   filestate = 0;               // Show file is idle
}
```

Again, `waitfor` is used to voluntarily relinquish control, this time while waiting for the write function to complete. If an error occurs during the write operation the device is unmounted and the application exits. Otherwise the loop counter and the buffer position pointer are advanced by the number of bytes actually written. Since this can be less than the requested number of bytes, it is best to check in a loop such as the `while` loop shown in `putdata`.

The last action taken by `putdata` is to reset `filestate`, indicating that the open file is available.

10.2.3.2.2 Costatement that Reads and Displays a File

The costatement named `showdata` waits for ten seconds. Then it waits for the open file to be available, and when it is, immediately marks it as unavailable.

```
costate showdata always_on{
   waitfor (DelaySec(10));
   waitfor (filestate == 0);
   filestate = 2;
```

The next statement modifies the internal file position pointer. The first time this costate runs, `readto` is zero, meaning the position pointer is at the first byte of the file. The variable `readto` is incremented every time a record is read from the file, allowing `showdata` to always know where to seek to next.

```
   waitfor (fat_Seek(&file, readto, SEEK_SET) != -EBUSY);
```

The rest of `showdata` is a `while` loop inside of a `while` loop. The inner `while` loop is where each record is read from the file into the buffer and then displayed in the Stdio window with the `printf()` call. Since `fat_Read()` may return less than the requested number of bytes, the `while` loop is needed to make sure that the function will be called repeatedly until all bytes have been read. When the full record has been read, it will then be displayed to the Stdio window.

The outer `while` loop controls when to stop reading records from the file. After the last record is read, the `fat_Read()` function is called once more, returning an end-of-file error. This causes the `if` statements that are checking for this error to return `TRUE`, which resets `filestate` to zero, breaking out of the outer `while` loop and freeing the lock for the `putdata` costatement to use.

```
while (filestate){
   icount = 0;
   iptr = ibuf;
   while (icount < REC_LEN) {
      waitfor((rc = fat_Read(&file, iptr, REC_LEN-icount)) != -EBUSY);
      if (rc < 0)
      {
         if (rc == -EEOF)
         {
            filestate = 0;
            break;
         }
         printf("fat_Read: rc = %d\n",rc);
         while ((rc=fat_UnmountDevice(first_part->dev)) == -EBUSY);
         return rc;
      }
      iptr += rc;
      icount += rc;
   }                              // end of inner while loop
   if (filestate)
   {
      printf("%s", ibuf);
      readto += REC_LEN;
   }
}                                 // end of outer while loop
```

The other costatement in the endless `while` loop is the one that blinks the LED. It illustrates that while using the file system in non-blocking mode, there is still plenty of time for other tasks.

10.2.4 FAT Operations

There are some basic groups of operations involved in using the Dynamic C FAT library. These are described at length in the following sections.

Section 10.2.4.1 "Format and Partition the Device"

- Default Partitioning
- Creating Multiple FAT Partitions
- Preserving Existing Partitions

Section 10.2.4.2 "File and Directory Operations"

- Open and Close Operations
- Read and Write Operations
- Going to a Specified Position in a File
- Creating Files and Subdirectories
- Reading Directories
- Deleting Files and Directories

10.2.4.1 Format and Partition the Device

The flash device must be formatted before its first use. Formatting it after its first use may destroy information previously placed on it.

10.2.4.1.1 Default Partitioning

As a convenience, `Samples/FileSystem/Fmt_Device.c` is provided to format the flash device. This program can format individual FAT 12/16 partitions, or can format all FAT 12/16 partitions found on a device. If no FAT 12/16 partitions are found, it offers the option of erasing the entire device and formatting it with a single FAT 16 partition. Be aware that this will destroy any data on the device, including that contained on FAT 32 partitions. This is an easy way to format new media that may contain an empty FAT32 partition spanning the entire device, such as a new SD or XD card.

After the device has been formatted with `Fmt_Device.c`, an application that wants to use the FAT file system just has to call the function `fat_Init()` (replaced in FAT version 2.01) or `fat_AutoMount()`. If you are calling `fat_AutoMount()` refer to Section 10.2.2.1 for an example of its use. Note that if you call `fat_AutoMount()` using the configuration flag `FDDF_DEV_FORMAT`, you may not need to run `Fmt_Device.c`.

10.2.4.1.2 Creating Multiple Partitions

To create multiple partitions on the flash device use the sample program `FAT_Write_MBR.c`, which will allow you to easily create as many as four partitions. This program does require that the device be "erased" before being run. This can be done with the appropriate sample program: `sdflash_inspect.c`, `sflash_inspect.c` or `nflash_inspect.c`. You only need to clear the first three pages on SD cards or serial flash, or the first page on NAND flash or XD cards. Once this is done, run FAT_Write_MBR and it will display the total size of the device in MegaBytes and allow you to specify the size of each partition until all the space is used. If you specify an amount larger than the space remaining, then all remaining space will be used for that partition. Once all space is specified, it will ask approval to write the new partition structure. This utility does not format the partitions, it merely creates their definitions. Run `Fmt_device.c` afterwards and use the 0 or 1 option to format the full device and all partitions will be formatted. Be forewarned that on removable media, using multiple partitions will typically make the device unusable with PC readers.

The sample program FAT_Write_MBR.c is distributed with FAT version 2.13. It is also compatible with FAT versions 2.01, 2.05 and 2.10. If you have one of these earlier versions of the FAT and would like a copy of FAT_Write_MBR.c, please contact Technical Support either by email to support@rabbitsemiconductor.com or using the online form available on the Rabbit website: www.rabbitsemiconductor.com/support/questionSubmit.shtml.

There is a way to create multiple partitions without using the utility FAT_Write_MBR.c; this auxiliary method is explained in Section 10.2.5.3.5.

10.2.4.1.3 Preserving Existing Partitions

If the flash device already has a valid partition that you want to keep, you must know where it is so you can fit the FAT partition onto the device. This requires searching the partition table for both available partitions and available space. An available partition has the partsecsize field of its mbr_part entry equal to zero.

Look in lib/.../RCM3300/RemoteApplicationUpdate/downloadmanager.lib for the function dlm_initserialflash() for an example of searching through the partition table for available partitions and space. See the next section for more information on the download manager (DLM) and how to set up coexisting partitions.

10.2.4.1.4 FAT and DLM Partitions

The RabbitCore RCM3300 comes with a download manager utility that creates a partition on a serial flash device, which is then used by the utility to remotely update an application. You can set up a device to have both a DLM partition and a FAT partition.

Run the program Samples/RCM3300/RemoteApplicationUpdate/DLM_FAT_FORMAT.C. This program must be run on an unformatted serial flash, i.e., a flash with no MBR. To remove an existing MBR, first run the program Samples/RCM3300/SerialFlash/SFLASH_INSPECT.C to clear the first three pages.

The program DLM_FAT_FORMAT.C will set aside space for the DLM partition and use the rest of the device to create a FAT partition. Then, when you run the DLM software, it will be able to find space for its partition and will coexist with the FAT partition. This shows the advantage to partitions: Partitions set hard boundaries on the allocation of space on a device, thus neither FAT nor the DLM software can take space from the other.

10.2.4.2 File and Directory Operations

The Dynamic C FAT implementation supports the basic set of file and directory operations. Remember that a partition must be mounted before it can be used with any of the file, directory or status operations.

10.2.4.2.1 Open and Close Operations

The `fat_Open()` function opens a file or a directory. It can also be used to create a file or a directory. When using the non-blocking FAT, check the return code and call it again with the same arguments until it returns something other than `-EBUSY`..

```
rc = fat_Open(my_part, "DIR\\FILE.TXT", FAT_FILE, FAT_CREATE,
              &my_file, &prealloc);
```

The first parameter, `my_part`, points to a partition structure. This pointer must point to a mounted partition. Some of the sample programs, like `fat_create.c`, declare a local pointer and then search for a partition pointer in the global array `fat_part_mounted[]`. Other sample programs, like `fat_shell.c`, define an integer to be used as an index into `fat_part_mounted[]`. Both methods accomplish the same goal of gaining access to a partition pointer.

The second parameter contains the file name, including the directory (if applicable) relative to the root directory. All paths in Dynamic C must specify the full directory path explicitly, e.g., `DIR1\\FILE.EXT` or `DIR1/FILE.EXT`. The direction of the slash in the pathname is a backslash by default. If you use the default backslash for the path separator, you must always precede it with another backslash, as shown in the above call to `fat_Open()`. This is because the backslash is an escape character in a Dynamic C string. To use the forward slash as the path separator, define the macro `FAT_USE_FORWARDSLASH` in your application (or in `FAT.LIB` to make it the system default).

The third parameter determines whether a file or directory is opened (`FAT_FILE` or `FAT_DIR`).

The fourth parameter is a flag that limits `fat_Open()` to the action specified. `FAT_CREATE` creates the file (or directory) if it does not exist. If the file does exist, it will be opened, and the position pointer will be set to the start of the file. If you write to the file without moving the position pointer, you will overwrite existing data. Use `FAT_MUST_CREATE` if you know the file does not exist; this last option is also a fail-safe way to avoid opening and overwriting an existing file since an `-EEXIST` error message will be returned if you attempt to create a file that already exists.

The fifth parameter, `&my_file`, is an available file handle. After a file or directory is opened, its handle is used to identify it when using other API functions, so be wary of using local variables as your file handle.

The final parameter is an initial byte count if the object needs to be created. It is only used if the `FAT_CREATE` or `FAT_MUST_CREATE` flag is used and the file or directory does not already exist. The byte count is rounded up to the nearest whole number of clusters greater than or equal to 1. On return, the variable `prealloc` is updated to the number of bytes allocated. Pre-allocation is used to set aside space for a file, or to speed up writing a large amount of data as the space allocation is handled once.

Pass NULL as the final parameter to indicate that you are opening the file for reading or that a minimum number of bytes needs to be allocated to the file at this time. If the file does not exist and you pass NULL, the file will be created with the minimum one cluster allocation.

Once you are finished with the file, you must close it to release its handle so that it can be reused the next time a file is created or opened.

```
rc = fat_Close(&my_file);
```

Remember to check the return code from fat_Close() since an error return code may indicate the loss of data. Once you are completely finished, call fat_UnmountDevice() to make sure any data stored in the cache is written to the flash device.

10.2.4.2.2 Read and Write Operations

Use fat_Read() to read a file.

```
rc = fat_Read(&my_file, buf, sizeof(buf));
```

The first parameter, &my_file, is a pointer to the file handle already opened by fat_Open(). The parameter buf points to a buffer for reading the file. The sizeof(buf) parameter is the number of bytes to be read into the buffer. It does not have to be the full size of the buffer. If the file contains fewer than sizeof(buf) characters from the current position to the end-of-file marker (EOF), the transfer will stop at the EOF. If the file position is already at the EOF, 0 is returned. The maximum number of characters read is 32767 bytes per call.

The function returns the number of characters read or an error code. Characters are read beginning at the current position of the file. If you have just written to the file that is being read, the file position pointer will be where the write left off. If this is the end of the file and you want to read from the beginning of the file you must change the file position pointer. This can be done by closing the file and reopening it, thus moving the position pointer to the start of the file. Another way to change the position pointer is to use the fat_Seek() function. This function is explained in Section 10.2.4.2.3.

Use fat_ReadDir() to read a directory. This function is explained in Section 10.2.4.2.5.

Use fat_Write() or fat_xWrite() to write to a file. The difference between the two functions is that fat_xWrite() copies characters from a string stored in extended memory.

```
rc = fat_Write(&my_file, "Write data\r\n", 12);
```

The first parameter, &my_file, is a pointer to the file handle already opened by fat_Open(). Because fat_Open() sets the position pointer to the start of the file, you will overwrite any data already in the file. You will need to call fat_Seek() if you want to start the write at a position other than the start of the file (see Section 10.2.4.2.3).

The second parameter contains the data to write to the file. Note that \r\n (carriage return, line feed) appear at the end of the string in the function. This is essentially a FAT (or really, DOS) convention for text files. It is good practice to use these standard line-end conventions. (If you only use \n, the file will read just fine on Unix systems, but some DOS-based programs may have difficulties.) The third parameter specifies the number of characters to write. Select this number with care since a value that is too small will result in your data being truncated, and a value that is too large will append any data that already exists beyond your new data.

Remember that once you are finished with a file you must close it to release its handle. You can call the fat_Close() function, or, if you are finished using the file system on a particular partition, call fat_UnmountPartition(), which will close any open files and then unmount the partition. If you are finished using the device, it is best to call fat_UnmountDevice(), which will close any open FAT files on the device and unmount all mounted FAT partitions. Unmounting the device is the safest method for shutting down after using the device.

10.2.4.2.3 Going to a Specified Position in a File

The position pointer is at the start of the file when it is first opened. Two API functions, `fat_Tell()` and `fat_Seek()`, are available to help you with the position pointer.

```
fat_Tell(&my_file, &pos);
fat_Seek(&my_file, pos, SEEK_SET);
```

The `fat_Tell()` function does not change the position pointer, but reads its value (which is the number of bytes from the beginning of the file) into the variable pointed to by `&pos`. Zero indicates that the position pointer is at the start of the file. The first parameter, `&my_file`, is the file handle already opened by `fat_Open()`.

The `fat_Seek()` function changes the position pointer. Clusters are allocated to the file if necessary, but the position pointer will not go beyond the original end of file (EOF) unless doing a `SEEK_RAW`. In all other cases, extending the pointer past the original EOF will preallocate the space that would be needed to position the pointer as requested, but the pointer will be left at the original EOF and the file length will not be changed. If this occurs, the error code `-EEOF` is returned to indicate the space was allocated but the pointer was left at the EOF. If the position requires allocating more space than is available on the device, the error code `-ENOSPC` is returned.

The first parameter passed to `fat_Seek()` is the file handle that was passed to `fat_Open()`. The second parameter, `pos`, is a long integer that may be positive or negative. It is interpreted according to the value of the third parameter. The third parameter must be one of the following:

- `SEEK_SET` - `pos` is the byte position to seek, where 0 is the first byte of the file. If `pos` is less than 0, the position pointer is set to 0 and no error code is returned. If pos is greater than the length of the file, the position pointer is set to EOF and error code `-EEOF` is returned.
- `SEEK_CUR` - seek `pos` bytes from the current position. If `pos` is less than 0 the seek is towards the start of the file. If this goes past the start of the file, the position pointer is set to 0 and no error code is returned. If pos is greater than 0 the seek is towards EOF. If this goes past EOF the position pointer is set to EOF and error code `-EEOF` is returned.
- `SEEK_END` - seek to `pos` bytes from the end of the file. That is, for a file that is x bytes long, the statement:

  ```
  fat_Seek (&my_file, -1, SEEK_END);
  ```

  will cause the position pointer to be set at x-1 no matter its value prior to the seek call. If the value of pos would move the position pointer past the start of the file, the position pointer is set to 0 (the start of the file) and no error code is returned. If `pos` is greater than or equal to 0, the position pointer is set to EOF and error code `-EEOF` is returned.
- `SEEK_RAW` - is similar to `SEEK_SET`, but if `pos` goes beyond EOF, using `SEEK_RAW` will set the file length and the position pointer to `pos`. This adds whatever data exists on the allocated space onto the end of the file..

10.2.4.2.4 Creating Files and Subdirectories

While the `fat_Open()` function is versatile enough to not only open a file but also create a file or a subdirectory, there are API functions specific to the tasks of creating files and subdirectories.

The `fat_CreateDir()` function is used to create a subdirectory one level at a time.

```
rc = fat_CreateDir(my_part, "DIR1");
```

The first parameter, `my_part`, points to a partition structure. This pointer must point to a mounted partition. Some of the sample programs, like `fat_create.c`, declare a local pointer and then search for a partition pointer in the global array `fat_part_mounted[]`. Other sample programs, like `fat_shell.c`, define an integer to be used as an index into `fat_part_mounted[]`. Both methods accomplish the same goal of gaining access to a partition pointer.

The second parameter contains the directory or subdirectory name relative to the root directory. If you are creating a subdirectory, the parent directory must already exist.

Once `DIR1` is created as the parent directory, a subdirectory may be created, and so on.

```
rc = fat_CreateDir(my_part, "DIR1/SUBDIR");
```

Note that a forward slash is used in the pathname instead of a backslash. Either convention may be used. The backslash is used by default. To use a forward slash instead, define `FAT_USE_FORWARDSLASH` in your application or in `FAT.LIB`.

A file can be created using the `fat_CreateFile()` function. All directories in the path must already exist.

```
rc = fat_CreateFile(my_part, "DIR1/SUBDIR/FILE.TXT", &prealloc,
   &my_file);
```

The first parameter, `my_part`, points to the static partition structure set up by `fat_AutoMount()`.

The second parameter contains the file name, including the directories (if applicable) relative to the root directory. All paths in the FAT library are specified relative to the root directory.

The third parameter indicates the initial number of bytes to pre-allocate. At least one cluster will be allocated. If there is not enough space beyond the first cluster for the requested allocation amount, the file will be allocated with whatever space is available on the partition, but no error code will be returned. If no clusters can be allocated, the `-ENOSPC` error code will return. Use NULL to indicate that no bytes need to be allocated for the file at this time. Remember that pre-allocating more than the minimum number of bytes necessary for storage will reduce the available space on the device.

The final parameter, `&my_file`, is a file handle that points to an available file structure. If NULL is entered, the file will be closed after it is created.

10.2.4.2.5 Reading Directories

The `fat_ReadDir()` function reads the next directory entry from the specified directory. A directory entry can be a file, directory or a label. A directory is treated just like a file.

```
fat_ReadDir(&dir, &dirent, mode);
```

The first parameter specifies the directory; `&dir` is an open file handle. A directory is opened by a call to `fat_OpenDir()` or by passing `FAT_DIR` in a call to `fat_Open()`. The second parameter, `&dirent`, is a pointer to a directory entry structure to fill in. The directory entry structure must be declared in your application, for example:

```
fat_dirent dirent;
```

Search Conditions

The last parameter, `mode`, determines which directory entry is being requested, a choice that is built from a combination of the macros described below. To understand the possible values for `mode`, the first thing to know is that a directory entry can be in one of three states: empty, active or deleted. This means you must choose one of the default flags described below, or one or more of the following macros:

- `FAT_INC_ACTIVE` - include active entries. This is the default setting if other `FAT_INC_*` macros are not specified; i.e., active files are included unless `FAT_INC_DELETED`, `FAT_INC_EMPTY`, or `FAT_INC_LNAME` is set.
- `FAT_INC_DELETED` - include deleted entries
- `FAT_INC_EMPTY` - include empty entries
- `FAT_INC_LNAME` - include long name entries (this is included for completeness, but is not used since long file names are not supported)

The above macros narrow the search to only those directory entries in the requested state. The search is then refined further by identifying particular attributes of the requested entry. This is done by choosing one or more of the following macros:

- `FATATTR_READ_ONLY` - include read-only entries
- `FATATTR_HIDDEN` - include hidden entries
- `FATATTR_SYSTEM` - include system entries
- `FATATTR_VOLUME_ID` - include label entries
- `FATATTR_DIRECTORY` - include directory entries
- `FATATTR_ARCHIVE` - include modified entries

Including a `FATATTR_*` macro means you do not care whether the corresponding attribute is turned on or off. Not including a `FATATTR_*` macro means you only want an entry with that particular attribute turned off. Note that the FAT system sets the archive bit on all new files as well as those written to, so including `FATATTR_ARCHIVE` in your mode setting is a good idea.

For example, if mode is (`FAT_INC_ACTIVE`) then the next directory entry that has all of its attributes turned off will be selected; i.e., an entry that is not read only, not hidden, not a system file, not a directory or a label, and not archived. In other words, the next writable file that is not hidden, system or already archived is selected.

But, if you want the next active file and do not care about the file's other attributes, mode should be (FAT_INC_ACTIVE | FATATTR_READ_ONLY | FATATTR_HIDDEN | FATATTR_SYSTEM | FATATTR_ARCHIVE). This search would only exclude directory and label entries.

Now suppose you want only the next active read-only file, leaving out hidden or system files. The next group of macros allows this search by filtering on whether the requested attribute is set. The filter macros are:

- FAT_FIL_RD_ONLY - filter on read-only attribute
- FAT_FIL_HIDDEN - filter on hidden attribute
- FAT_FIL_SYSTEM - filter on system attribute
- FAT_FIL_LABEL - filter on label attribute
- FAT_FIL_DIR - filter on directory attribute
- FAT_FIL_ARCHIVE - filter on modified attribute

If you set mode to (FAT_INC_ACTIVE | FATATTR_READ_ONLY | FAT_FIL_RD_ONLY | FATATTR_ARCHIVE), the result will be the next active file that has its read-only attribute set (and has the archive attribute in either state).

> **NOTE:** If you have FAT version 2.05 or earlier, you do not have access to the FAT_FIL_* macros.

Default Search Flags

To make things easier, there are two predefined mode flags. Each one may be used alone or in combination with the macros already described.

- FAT_INC_ALL - selects any directory entry of any type.
- FAT_INC_DEF - selects the next active file or directory entry, including read-only or archived files. No hidden, system, label, deleted, or empty directories or files will be selected. This is typically what you see when you do a directory listing on your PC.

Search Flag Examples

Here are some more examples of how the flags work.

1. If you want the next hidden file or directory:

   Start with the FAT_INC_DEF macro default flag. This flag does not allow hidden files, so we need FATATTR_HIDDEN. Then to narrow the search to consider only a hidden file or directory, we need the macro FAT_FIL_HIDDEN to filter on files or directories that have the hidden attribute set. That is, mode is set to:

   ```
   FAT_INC_DEF | FATATTR_HIDDEN | FAT_FIL_HIDDEN
   ```

2. If you want the next hidden directory:

   Start with the FAT_INC_DEF macro default flag. To narrow the search to directories only, we want entries with their directory attribute set; therefore, OR the macros FATATTR_DIRECTORY and FAT_FIL_DIR. Then OR the macros FATATTR_HIDDEN and FAT_FIL_HIDDEN to search only for directories with their hidden attribute set. Set mode to:

   ```
   FAT_INC_DEF | FATATTR_DIRECTORY | FAT_FIL_DIR | FATATTR_HIDDEN |
   FAT_FIL_HIDDEN
   ```

3. If you want the next hidden file (no directories):

   Start with the predefined flag, FAT_INC_DEF. This flag allows directories, which we do not want, so we do an AND NOT of the FATATTR_DIRECTORY macro.

   Next we want to narrow the search to only entries that have their hidden attribute set. The default flag does not allow hidden flags, so we need to OR the macros FATTR_HIDDEN and FAT_FIL_HIDDEN.

   That is, set mode to:

   ```
   FAT_INC_DEF & ~FATATTR_DIRECTORY | FATATTR_HIDDEN | FAT_FIL_HIDDEN
   ```

4. If you want the next non-hidden file (no directories):

   First, select the FAT_INC_DEF filter default flag. This flag allows directories, which we do not want, so we do an AND NOT of the FATATTR_DIRECTORY macro. The default flag already does not allow hidden files, so we are done. That is, set mode to:

   ```
   FAT_INC_DEF & ~FATATTR_DIRECTORY
   ```

5. Finally let's see how to get the next non-empty entry of any type.

   Start with the predefined flag, FAT_INC_ALL. This flag selects any directory entry of any type. Since we do not want empty entries, we have to remove that search condition from the flag, so we do an AND NOT for the FAT_INC_EMPTY macro to filter out the empty entries. That means mode is the bitwise combination of the macros:

   ```
   mode = FAT_INC_ALL & ~FAT_INC_EMPTY
   ```

10.2.4.2.6 Deleting Files and Directories

The fat_Delete() function is used to delete a file or directory. The second parameter sets whether a file or directory is being deleted. Only one file or directory may be deleted at any one time—this means that you must call fat_Delete() at least twice to delete a file and its associated directory (if the directory has no other files or subdirectories since a directory must be empty to be deleted).

```
fat_Delete(my_part, FAT_FILE, "DIR/FILE.TXT");
```

The first parameter, my_part, points to the static partition structure that was populated by fat_AutoMount(). The second parameter is the file type, FAT_FILE or FAT_DIR, depending on whether a file or a directory is to be deleted. The third parameter contains the file name, including the directory (if applicable) relative to the directory root. All paths in the FAT library are specified relative to the root directory.

10.2.4.3 Error Handling

Most routines in the FAT library return an int value error code indicating the status of the requested operation. Table 12 contains a list of error codes specific to the FAT file system. Most of these codes, along with some other error codes, are defined in /`Lib/../ERRNO.LIB`.

Table 12. FAT-Specific Error Codes

Code	Value	Description
EOF	231	End of File Encountered
EEOF	41	End-of-file marker reached
ETYPE	232	Incorrect Type
EPATHSTR	233	Invalid Path String
EROOTFULL	234	Root Directory is Full
EUNFORMAT	235	Unformatted Volume
EBADPART	236	Invalid Partition
ENOPART	237	Unpartitioned / Unformatted Media
ENOTEMPTY	238	Open Files in Partition / Directory to be Deleted
EPERM	1	Operation not permitted
ENOENT	2	No such file or directory
EIO	5	I/O error
EBUSY	16	Device or resource busy
EEXIST	17	File exists
ENODEV	19	No such device
ENOSPC	28	No space left on device
ENOTEMPTY	39	Directory is not empty
ENOMEDIUM	123	No medium found

10.2.5 More FAT Information

The FAT file system stores and organizes files on a storage device such as a hard drive or a memory device.

10.2.5.1 Clusters and Sectors

Every file is stored on one or more *clusters*. A cluster is made up of a contiguous number of bytes called *sectors* and is the smallest unit of allocation for files. The Dynamic C FAT implementation supports a sector size of 512 bytes. Cluster sizes depend on the media. The table below gives the cluster sizes used for some of our RabbitCore modules.

Table 13. Cluster Sizes on Flash Devices

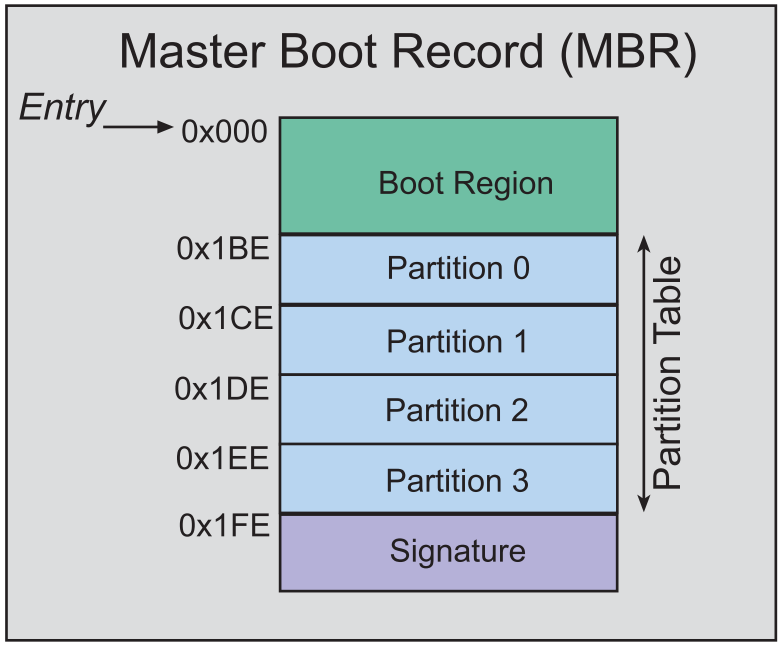
RabbitCore Model	Flash Device	Number of Sectors per Cluster
RCM 3700	1 MB Serial Flash	1
RCM 3300	4 and 8 MB Serial Flash	2
RCM3360/70	NAND Flash	32

The cluster size for a NAND device corresponds to its page size. Note that a file or directory takes at minimum one cluster. On a NAND device the page size is 16K bytes; therefore, while it is allowable to write very small files to the FAT file system on a NAND device, it is not space efficient. Even the smallest file takes at least 16,000 bytes of storage. Cluster sizes for SD cards vary with the size of the card inserted. To determine the number of sectors per cluster on an SD card, divide the size of the card by 32MB.

10.2.5.2 The Master Boot Record

The *master boot record* (MBR) is located on one or more sectors at the physical start of the device. Its basic structure is illustrated in Figure 3. The boot region of the MBR contains DOS boot loader code, which is written when the device is formatted (but is not otherwise used by the Dynamic C FAT file system). The partition table follows the boot region. It contains four 16-byte entries, which allows up to four partitions on the device. Partition table entries contain some critical information: the partition type (Dynamic C FAT recognizes partition types FAT12 and FAT16) and the partition's starting and ending sector numbers. There is also a field denoting the total number of sectors in the partition. If this number is zero, the corresponding partition is empty and available.

Figure 3. High-Level View of an MBR

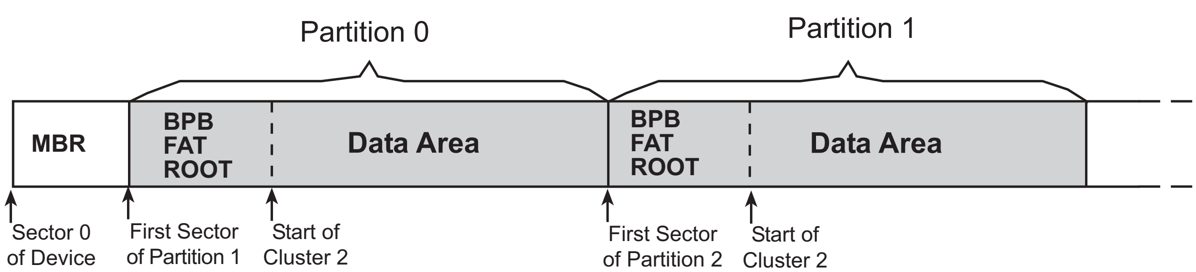


NOTE: Some devices are formatted without an MBR and, therefore, have no partition table. This configuration is not currently supported in the Dynamic C FAT file system.

10.2.5.3 FAT Partitions

The first sector of a valid FAT file system partition contains the *BIOS parameter block* (BPB); this is followed by the *file allocation table* (FAT), and then the *root directory*. The figure below shows a device with two FAT partitions.

Figure 4. Two FAT Partitions on a Device



10.2.5.3.1 BPB

The fields of the BPB contain information describing the partition:

- the number of bytes per sector
- the number of sectors per cluster (see Table 13)
- the total count of sectors on the partition
- the number of root directory entries
- plus additional information not mentioned here

The FAT type (FAT12 or FAT16) is determined by the count of clusters on the partition. The “12” and “16” refer to the number of bits used to hold the cluster number. The FAT type is calculated using information found in the BPB. Information from a BPB on a mounted partition is stored in the partition structure (of type `fat_part`) populated by `fat_AutoMount()`.

Partitions greater than or equal to 2 MB will be FAT16. Smaller partitions will be FAT12. To save code space, you can compile out support for either FAT type. Find the lines

```
#define FAT_FAT12          // comment out to disable FAT12 support
#define FAT_FAT16          // comment out to disable FAT16 support
```

in `LIB/../FAT.LIB`, make your change, and then recompile your application.

10.2.5.3.2 FAT

The file allocation table is the structure that gives the FAT file system its name. The FAT stores information about cluster assignments. A cluster is either assigned to a file, is available for use, or is marked as bad. A second copy of the FAT immediately follows the first.

10.2.5.3.3 Root Directory

The root directory has a predefined location and size. It has 512 entries of 32 bytes each. An entry in the root directory is either empty or contains a file or subdirectory name (in 8.3 format), file size, date and time of last revision and the starting cluster number for the file or subdirectory.

10.2.5.3.4 Data Area

The data area takes up most of the partition. It contains file data and subdirectories. Note that the data area of a partition must, by convention, start at cluster 2.

10.2.5.3.5 Creating Multiple FAT Partitions

FAT version 2.13 introduces `FAT_Write_MBR.c`, a utility that simplifies the creation of multiple partitions. It is distributed with FAT version 2.13. It is also compatible with FAT versions 2.01, 2.05 and 2.10. If you have one of these earlier versions of the FAT and would like a copy of `FAT_Write_MBR.c`, please contact Technical Support either by email to support@rabbitsemiconductor.com or by using the online form available on the Rabbit website: www.rabbitsemiconductor.com/support/questionSubmit.shtml. See Section 10.2.4.1.2 for information on running this utility.

Without the use of `FAT_Write_MBR.c`, creating multiple FAT partitions on the flash device requires a little more effort than the default partitioning. If the flash device does not contain an MBR, i.e., the device is not formatted, both `fat_Init()` and `fat_AutoMount()` return an error code (`-EUNFORMAT`) indicating this fact. So the next task is to write the MBR to the device. This is done with a call to `fat_FormatDevice()`. Since we want more than one partition on the flash device, `fat_FormatDevice()` must be called with a `mode` parameter of zero.

Before calling `fat_FormatDevice()`, partition specific information must be set in the `mbr_part` entries for each partition you are creating. The following code shows possible information for partition 0 where `MY_PARTITION_SIZE` is equal to the size of the desired partition in bytes, 512 is the flash sector size, and `dev` points to the `mbr_part` structure.

```
memset(dev->part, 0, sizeof(mbr_part));
dev->part[0].starthead = 0xFE;
dev->part[0].endhead = 0xFE;
dev->part[0].startsector = 1;
dev->part[0].partsecsize = (MY_PARTITION_SIZE / 512 ) + 1;
dev->part[0].parttype = (dev->part[0].partsecsize < SEC_2MB) ? 1 : 6;
```

The `memset()` function is used to initialize the entry to zero. The values for `starthead` and `endhead` should be 0xFE to indicate that the media uses LBA (Logical Block Addressing) instead of head and cylinder addressing. The FAT library uses LBA internally. The values for the `startsector`, `partsecsize` and `parttype` fields determine where the partition starts, how many sectors it contains and what partition type it is. The number of sectors in the partition is calculated by dividing the number of raw bytes in the partition by the sector size of the flash. The number of raw bytes in the partition includes not only bytes for file storage, but also the space needed by the BPB and the root directory. One is added to `dev->partsecsize` to ensure an extra sector is assigned if `MY_PARTITION_SIZE` is not evenly divisible by the size of a flash sector. The partition type (`.parttype`) is determined by the partition size: 1 indicates FAT12 and 6 indicates FAT16. Fill in an `mbr_part` structure for each partition you are creating. The remaining entries should be zeroed out.

When laying out partitions, there are three basic checks to make sure the partitions fit in the available device space and do not overlap.

1. No partition can start on a sector less than 1.
2. Each partition resides on sectors from `startsector` through `startsector+partsecsize-1`. No other partition can have a `startsector` value within that range.
3. No partition ending sector (`startsector+partsecsize-1`) can be greater than or equal to the total sectors on the device.

The partition boundaries are validated in the call to `fat_FormatDevice()` and the function will return an error if any of the partition boundaries are invalid. If `fat_FormatDevice()` returns success, then call `fat_AutoMount()` with flags of `FDDF_COND_PART_FORMAT` | `FDDF_MOUNT_DEV_#` | `FDDF_MOUNT_PART_ALL`; where # is the device number for the device being partitioned. This will format and mount the newly created partitions.

10.2.5.4 Directory and File Names

File and directory names are limited to 8 characters followed by an optional period (.) and an extension of up to 3 characters. The characters may be any combination of letters, digits, or characters with code point values greater than 127. The following special characters are also allowed:

```
$ % ' - _ @ ~ ` ! ( ) { } ^ # &
```

File names passed to the file system are always converted to upper case; the original case value is lost.

The maximum size of a directory is limited by the available space. It is recommended that no more than ten layers of directories be used with the Dynamic C FAT file system.

10.2.5.5 µC/OS-II and FAT Compatibility

Versions of the FAT file system prior to version 2.10 are compatible with µC/OS-II only if FAT API calls are confined to one µC/OS-II task. To make the FAT API reentrant from multiple tasks, you must do the following:

- Use FAT version 2.10
- #define `FAT_USE_UCOS_MUTEX` before #use'ing `FAT.LIB`
- Call the function `fat_InitUCOSMutex(priority)` after calling `OSInit()` and before calling FAT APIs or beginning multitasking; the parameter "priority" MUST be a higher priority than all tasks using FAT APIs
- Call only high-level fat APIs with names that begin with "fat_"

See the function description for `fat_InitUCOSMutex()` for more details, and the sample program `Samples/FileSystem/FAT_UCOS.C` for a demonstration of using FAT with µC/OS-II.

10.2.5.6 SF1000 and FAT Compatibility

There are two macros that need to be defined for the FAT to work with the SF1000 Serial Flash Expansion Board.

```
#define SF_SPI_DIVISOR 5
#define SF_SPI_INVERT_RX
```

10.2.5.7 Hot-Swapping an xD Card

Hot-swapping is currently supported on the RCM3365 and the RCM3375. FAT version 2.10 or later is required. Two sample programs are provided in `Samples/FileSystem` to demonstrate this feature: `FAT_HOT_SWAP.C` and `FAT_HOT_SWAP_3365_75.C`. The samples are mostly identical: they both test for a keyboard hit to determine if the user wants to hot-swap the xD card, but, in addition, the sample program `FAT_HOT_SWAP_3365_75.C` also checks for a switch press and indicates a ready-to-mount condition with an LED.

After unmounting the xD card call `_fat_config_init()`. This disconnects drive and device structures from internal tables to work around a potential problem swapping from smaller to larger removable devices.

As demonstrated in the sample programs, an xD card should only be removed after it has unmounted with `fat_UnmountDevice()` and no operations are happening on the device.

Only `fat_AutoMount()` should be used to remount xD cards. In addition, the function `nf_XD_Detect()` should be called to verify xD card presence before attempting to remount an xD card.

xD cards formatted with versions of the FAT prior to 2.10 did not have unique volume labels. If there is a chance that two such cards may be swapped, call `fat_autoMount()` with the `FDDF_NO_RECOVERY` flag set. This means that if there is a write cache entry to be written, it will not be written. The function `fat_UnmountDevice()` flushes the cache (i.e., writes all cache entries to the device) before unmounting, so this should not generally be a problem if the device was properly unmounted.

10.2.5.8 Hot-Swapping an SD Card

Hot-swapping is currently supported on the RCM3900 and the RCM3910. FAT version 2.14 or later is required. A sample program is provided in `Samples/FileSystem` to demonstrate this feature: `FAT_HOT_SWAP_SD.C`. The sample tests for a keyboard hit to determine if the user wants to hot-swap the SD card.

Hot-swapping an SD card requires that you unmount the device before removal, as the FAT filesystem employs a cache system that may not have written all information to the device unless unmounted.

As demonstrated in the sample program, the SD card should only be removed after it has unmounted with `fat_UnmountDevice()` and no operations are happening on the device. Only `fat_AutoMount()` should be used to remount SD cards. In addition, the function `sdspi_debounce()` should be called to verify SD card presence before attempting to remount an SD card.

10.2.5.9 Unsupported FAT Features

At this time, the Dynamic C FAT file system does not support the following.

- Single-volume drives (they do not have an MBR)
- FAT32 or long file or directory names
- Sector sizes other than 512 bytes
- Direct parsing of relative paths
- Direct support of a “working directory”
- Drive letters (the FAT file system is not DOS)

10.2.5.10 References

There are a number of good references regarding FAT file systems available on the Internet. Any reasonable search engine will bring up many hits if you type in relevant terms, such as “FAT,” “file system,” “file allocation table,” or something along those lines. At the time of this writing, the following links provided useful information.

1. This link is to Microsoft’s “FAT32 File System Specification,” which is also applicable to FAT12 and FAT16.

   www.microsoft.com/whdc/system/platform/firmware/fatgen.mspx

2. This article gives a brief history of FAT.

   http://en.wikipedia.org/wiki/File_Allocation_Table

3. These tutorials give lots of details plus links to more information.

   www.serverwatch.com/tutorials/article.php/2239651

   www.pcguide.com/ref/hdd/file/fat.htm

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 Microprocessor Instruction Reference Manual* available on the Rabbit website:

`www.rabbitsemiconductor.com/docs/`

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 the registers 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 software breakpoint support in 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`). Below is an example.

```
#asm debug
function::
...
ret
#endasm
```

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 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
.	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 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 is outside of that embbeded assembly segment).

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, Rabbit 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 11.4.1.2 for more information on the frame reference point.
`@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.

The name of a structure element represents the offset of the element from the beginning of the structure. In the following structure, for example, for the following structure

```
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{         // offset into s
   int x;         // 0
   struct a {     // 2 (i.e., sizeof(x))
      int b;      // 2, offset is 0 relative to a
      int c;      // 4, offset is 2 relative to a
   };
};
```

Just like in the first definition of structure “s”, the assembly expression s+x evaluates to 0; s+a evaluates to 2 and s+b evaluates to 2 (both expressions evaluate to the same value because both “a” and “b” are offset “0” from “a”); and finally, s+c evaluates to 4 because s+a evaluates to 2 and a+c evaluates to 2.

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, Rabbit 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.

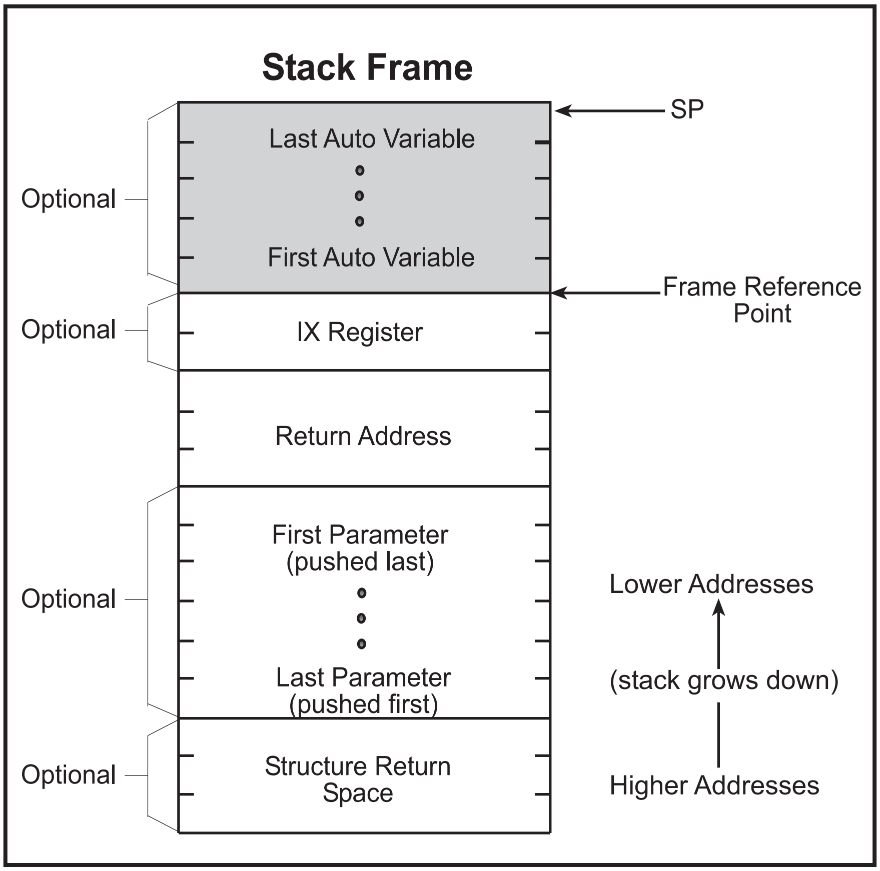
11.4.1 The Stack Frame

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.

11.4.1.1 Stack Frame Diagram

Figure 11.1 shows the general appearance of a stack frame.

Figure 11.1 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.

11.4.1.2 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

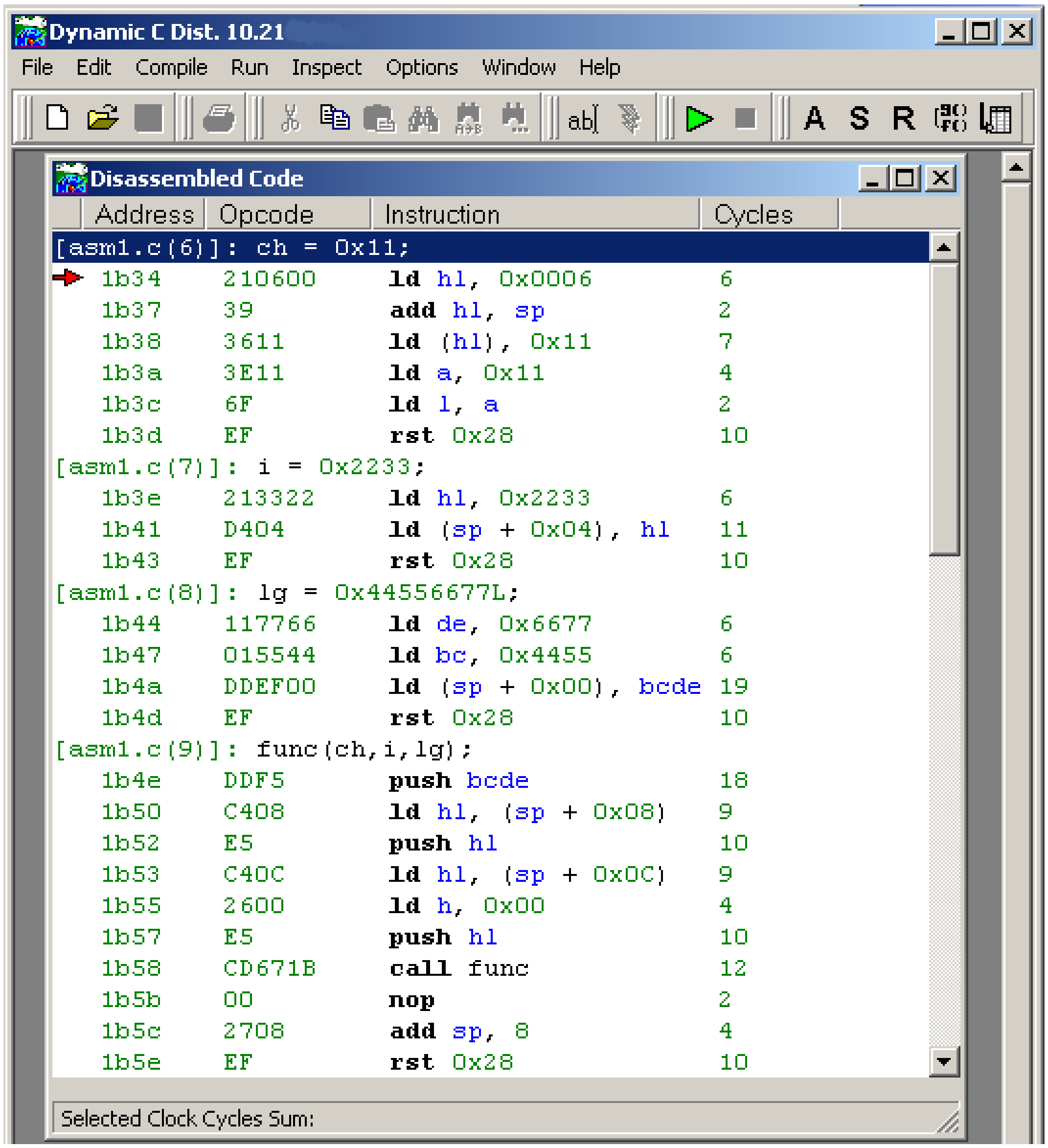
The purpose of the following sample program, `asm1.c`, is to show the different ways to access stack-based variables from assembly code.

```
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
}
```

11.4.3 The Disassembled Code Window

A program may be debugged at the assembly level by opening the Disassembled Code window (aka, the Assembly 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 Window



The Disassembled Code window shows the memory address on the far left, followed by the opcode bytes, 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 bottom of the window when the block is selected. The total assumes one execution per instruction, so the user must take looping and branching into consideration when evaluating execution times.

11.4.4 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
```

11.4.4.1 Using the IX Register as a Frame Pointer

Using IX as a frame pointer is a convenient way to access stack variables in assembly. Using SP requires extra bookkeeping when values are pushed on or popped off the stack.

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.

11.4.4.2 Using Index Registers as Pointers to Aggregate Types

The members of Dynamic C aggregate types (structures and unions) can be accessed from within an assembly block of code using any of the index registers: IX, IY, SP.

The library `pool.lib` has code that illustrates using an index register in assembly to access the member of a structure that was defined in Dynamic C. Refer to the function `palloc_fast()`.

Here is another example:

```
typedef struct{
   int x;
   int y;
   long time;
}TStruct;

void func(int x, int y, TStruct *s){
#asm
   ld ix,(sp+@SP+s)
   ld hl,(ix+[TStruct]+y)
   .
   .
#endasm
}
```

11.4.4.3 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.

11.5.3 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 f1 field of returned struct
   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. Push and pop the LXPC as a defensive programming strategy to avoid corrupting large memory support. For example, the LCALL instruction clears the LXPC so it is essential that this register is saved before issuing an LCALL and restored after the LRET.
3. 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.
4. Remove the cause of the interrupt.
5. If an interrupt has more than one possible cause, check for all the causes and remove all the causes at the same time.
6. 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.
7. 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.
8. Return. There are two types of interrupt returns: `ret` and `reti`.

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 designer's handbook for your Rabbit microprocessor (e.g., 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 `lib\..\bioslib\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
```

Table 11-5 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

If you have problems with your assembly code, consider the possibility of any of the following situations:

- **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.

- **Relocatable code.**

  Jump relative (JR) instructions allow easier code relocation because the jump is relative to the current program counter. For example, RAM functions are usually written in assembly and are relocated to RAM from flash. A jump (JP) instruction would not work in this case because the jump would be to a flash location and not the intended RAM location. Using JR instead of JP will jump to the intended RAM location.

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

IMPORTANT: bbram does not provide data integrity; instead, use the keyword protected to ensure integrity of data across power failures.

Identifies a variable to be placed into a second root data area with global extent/scope reserved for battery-backed RAM on boards with more than one RAM device. Generally, the battery-backed RAM is attached to CS1 due to the low-power requirements. Other than its assigned root data location, a bbram variable is identical to a normal root variable. 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. Note that bbram variables must have either static or global storage.

For boards that utilize fast SRAM in addition to a battery-backed SRAM, like the RCM3200, the size of the battery-backed root data space is specified by a BIOS macro called BBROOTDATASIZE. In version Dynamic C 9.50 and earlier, the default value for this is 4K. Note that this macro is defined to zero for boards with only a single SRAM.

See the *Rabbit 2000 Microprocessor Designer's Handbook* or 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 with the assumption that the RAM is battery-backed. Please refer to _xalloc for information on how to access battery-backed data in xmem.

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
```

cofunc

Indicates the beginning of a cofunction.

```
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.

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. The use of `const` is not currently permitted with return types, auto variables or parameters in a function prototype.

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;
```

NOTE: The default storage class is auto, so the above code would have to be outside of a function or would have to be explicitly set to static.

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
```

The `debug` keyword in combination with the `norst` keyword will give you run-time checking without debug. For example,

```
debug norst foo() {
}
```

will perform run-time checking if enabled, but will not have rst instructions.

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            // "statement" 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

The keyword `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>
```

Interrupt vector names and ISR names are found in Table 11-5 on page 183. 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.

If you are using Dynamic C 9.30 or later, the `_RK_FIXED_VECTORS` macro must be used to conditionally compile code containing the `interrupt_vector` keyword. For Rabbit 3000A and later CPUs, Dynamic C makes use of the new RAMSR capability to make in-RAM interrupt table access fast. The following code demonstrates the correct way to use `_RK_FIXED_VECTORS` so as to eliminate errors regarding undefined interrupt vectors.

As demonstrated in DC 9.52's standard samples that conditionally use the `interrupt_vector` keyword, the correct usage is as follows:

```
nodebug root interrupt void pwm_isr(){
   // example code does not do anything
}

nodebug root interrupt void ic_isr(){
   // example code does not do anything
}

main(){
#if __SEPARATE_INST_DATA__ && (_RK_FIXED_VECTORS)
   interrupt_vector inputcap_intvec ic_isr;
   interrupt_vector pwm_intvec pwm_isr;
#else
   SetVectIntern(0x1A, ic_isr); // set up ISR
   SetVectIntern(0x17, pwm_isr); // set up ISR
#endif

printf("ISR's setup correctly\n");
}
```

__lcall__

When used in a function definition, the __lcall__ function prefix forces long call and return (lcall and lret) instructions to be generated for that function, even if the function is in root. This allows root functions to be safely called from xmem. In addition to root functions, this prefix also works with function pointers. The __lcall__ prefix works safely with xmem functions, but has no effect on code generation. Its use with cofunctions is prohibited and will generate an error if attempted.

```
root __lcall__ int foo(void) {
   return 10;          // Generates an lret instruction, even though we are in root
}

main() {
   foo();              // This now generates an lcall instruction
}
```

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(){
    ...
}
```

The norst keyword in combination with the debug keyword will give you run-time checking without debug. For example,

```
debug norst foo() {
}
```

will perform runtime-checking if enabled, but will not have rst instructions.

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. This operation requires battery-backed RAM and the use of the main system clock. If you are using the 32 kHz clock you must switch back to the main system clock to use protected variables because the atomicity of the write cannot be ensured when using the 32 kHz clock.

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

register

The register keyword is not currently implemented in Dynamic C, but is reserved for possible future implementation. It is currently synonymous with the keyword `auto`.

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
```

scofunc

Indicates the beginning of a single-user cofunction. See cofunc on page 189.

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 reinitialize `vec[]`.

shared

Indicates that changes to a multi-byte variable (such as a `float`) are atomic. Interrupts are disabled when the variable is being changed. Local variables cannot be shared. Note that you must be running off the main system clock to use shared variables. This is because the atomicity of the write cannot be ensured when running off the 32 kHz clock.

```
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`”.

void

This keyword conforms to ANSI C. Thus, it can be used in three different ways.

1. Parameter List - used to identify an empty parameter list (a.k.a., argument list). An empty parameter list can also be identified by having nothing in it. The following two statements are functionally identical:

```
int functionName(void);
int functionName();
```

2. Pointer to Void - used to declare a pointer that points to something that has no type.

```
void *ptr_to_anything;
```

3. Return Type - used to state that no value is returned.

```
void functionName(param1, param2);
```

volatile

Reserved for future use.

waitfor

Used in a costatement or cofunction, 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();
}
```

`wfd` may return a value. In the example above, the variable `x` is set to 1 if `login()` completes execution normally and set to -1 if it aborts. This scheme is extended when there are multiple cofunctions inside the `wfd`: if no abort has taken place in any cofunction, `wfd` returns 1, 2, ..., n to indicate which cofunction inside the braces finished executing last. If an abort takes place, `wfd` returns -1, -2, ..., -n to indicate which cofunction caused the abort.

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 separate I&D space might cause an application to run out of xmem, depending on the size of the application and the size of the flash. If this occurs, the programmer should consider using only one of the #memmap xmem or separate I&D space options. If the application is extremely tight for xmem code memory but has root code memory to spare, the programmer may also consider explicitly tagging some xmem or anymem functions with the root keyword.

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. A function's local debug or nodebug keyword overrides the global #debug or #nodebug directive. In other words, if a function does *not* have a local debug or nodebug keyword, the #debug or #nodebug directive would apply.

#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 together, separately from the library functions.

`#nointerleave` forces the user-written functions to be compiled first.The `#nointerleave` directive, when placed at the top of application code, tells Dynamic C to compile all of the application code first and then to compile library code called by the application code afterward, and then to compile other library code called by the initial library code following that, and so on until finished.

Note that the `#nointerleave` directive can be placed anywhere in source code, with the effect of stopping interleaved compilation of functions from that point on. If `#nointerleave` is placed in library code, it will effectively cause the user-written functions to be compiled together starting at the statement following the library call that invoked `#nointerleave`.

#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.

#warnt

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.2.2 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 a pointer to an integer
int (*a)();               // pointer to a function returning an integer
```

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 while using either a RabbitLink board or a RabbitSys-enabled 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 directions.

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.

The Ctrl key works in conjunction with the arrow keys:

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.

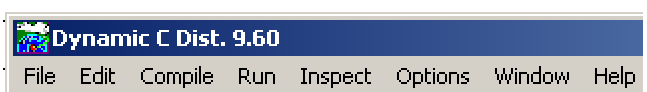
The Ctrl key also works in conjunction with “[” for delimiter matching. Place the cursor before the delimiter you are attempting to match and press “Ctrl+[”. The cursor will move to just before the matching delimiter.

Note that delimiters in comments are also matched. For example, in the following code, <Ctrl+[> counts commented-out braces in the matching, giving a false indication that the main function has balanced curly braces when in fact it does not.

```
main()
{
   {
   //}
/*
}
*/
```

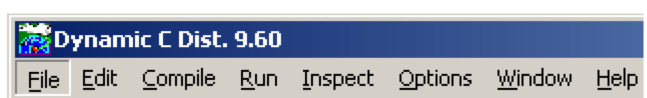
14.2 Menus

Dynamic C’s main menu has eight command menus, as well as the standard Windows system menus.



An available command can be executed from a menu by either clicking the menu and then clicking the command, or by pressing the Alt key to activate the menu bar, using the left and right arrow keys to select a menu, and then using the up or down arrow keys to select a command before pressing the Enter key.

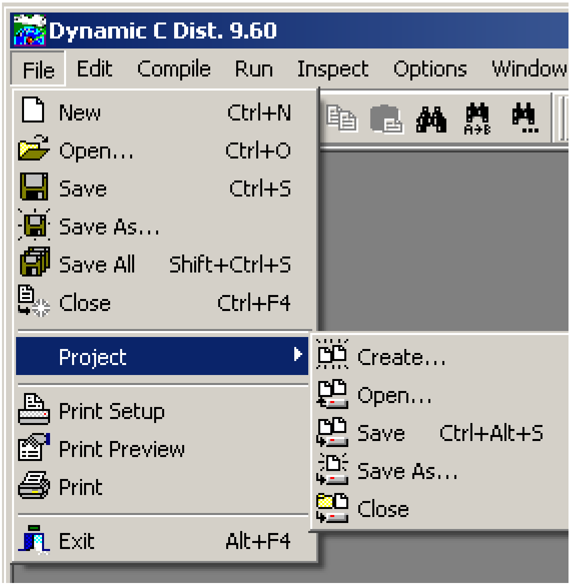
14.2.1 Using Keyboard Shortcuts



For some of us it is easier to type keyboard shortcuts than to use a mouse. A menu can be activated by pressing the Alt key while pressing the underlined letter of the menu name. This is the de facto standard, as it is used in numerous commercial software products. Pressing the Alt key allows you to see which character in the menu name is underlined, as shown in this second screenshot of Dynamic C’s main menu. All the keyboard shortcuts on the main menu use the first letter of the menu name in the shortcut. Some keyboard shortcuts have this obvious connection while others do not. See the Editor Tab screenshot in Section 14.2.7 for some examples of not so obvious keyboard shortcuts. A keyboard shortcut that is not menu specific is the Esc key, which will make any visible menu disappear.

14.2.2 File Menu

To select the File menu: click on its name in Dynamic C’s main menu or press <Alt+F>.



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. 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, “Project Files.” 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

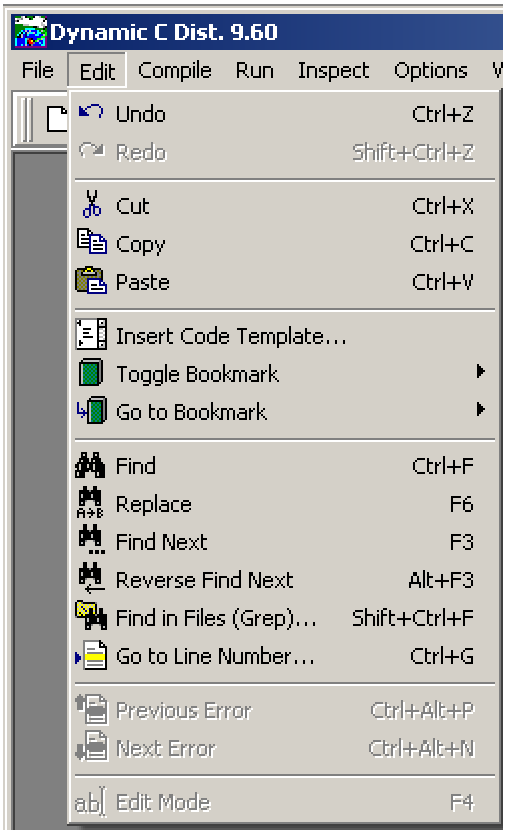
Brings up the Print dialog box, which allows you to choose a printer. Only text in an editor window can be printed. To print the contents of debug windows the text must be copied and pasted to an editor window. (The Stdio window is an exception; its contents may be automatically written to a file, which may then be printed.) 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.

14.2.3 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 ten bookmarks in the active edit window.

Go to Bookmark

Go to one of ten 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 the Persistent blocks option was checked in the Editor tab in 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. Morethan 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>

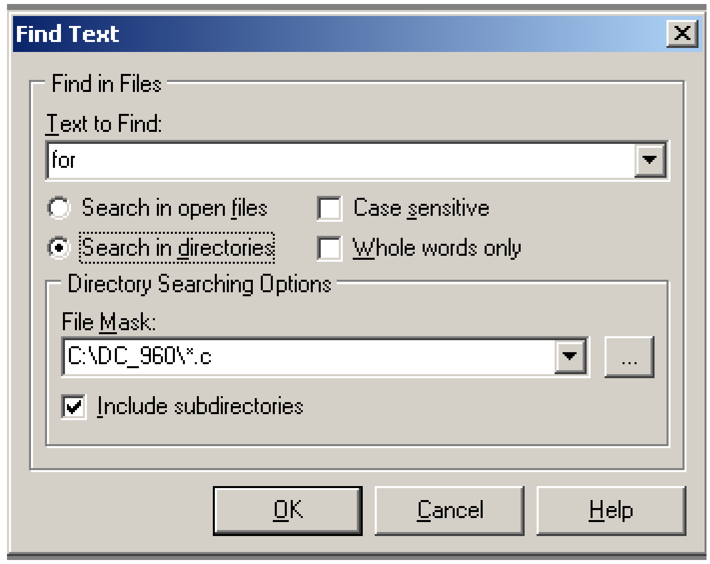
Once search text has been specified with the Find or Replace commands, the Find Next command will find the next occurrence of the same text, searching forward or in reverse, case sensitive or not, as specified with the previous Find or Replace command. If the previous command was Replace, the operation 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 file types can be separated by semicolons. For example, entering the following search criteria: `C:\mydirectory\*.lib;*.c` will search all `.lib` and `.c` files in `mydirectory`.

Starting with Dynamic C 9.60, the "Search Results" window has a right-click menu that allows you to view source files, as well as copy or delete selected entries.

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>

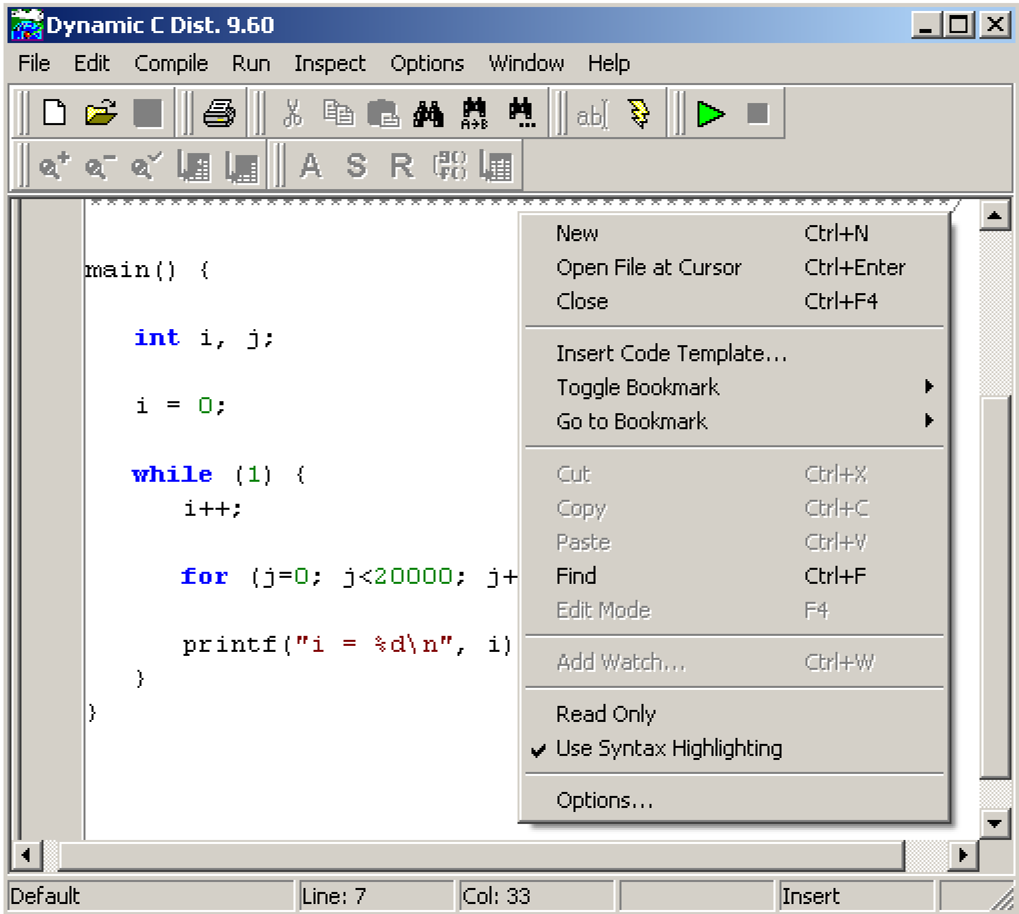
Locates the next compilation error in the source code. Any error messages will be displayed in a list in the Compiler Messages window after a program is compiled. Dynamic C selects the next error in the list and displays the offending line of code in the text window.

Edit Mode <F4>

Switches to edit mode from run, also known as debug, mode. After successful compilation or execution, no changes to the file are allowed unless in edit mode. If the compilation fails or a runtime error occurs, Dynamic C comes back already in edit mode.

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.

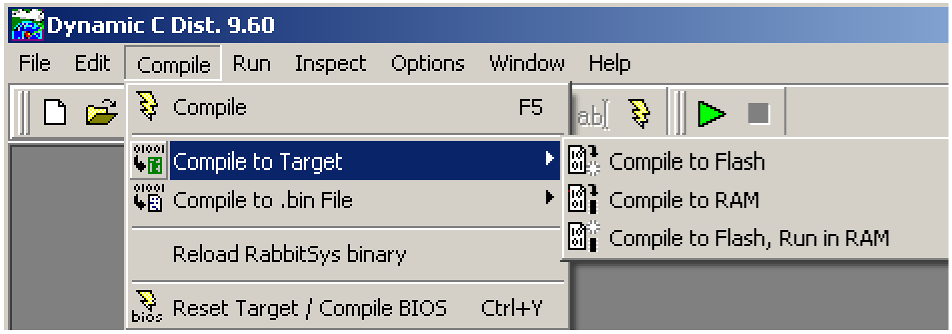


Open File at Cursor <Ctrl+Enter>

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 the "lib.dir" file 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.

14.2.4 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 Compiler tab of the Project Options dialog. 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.

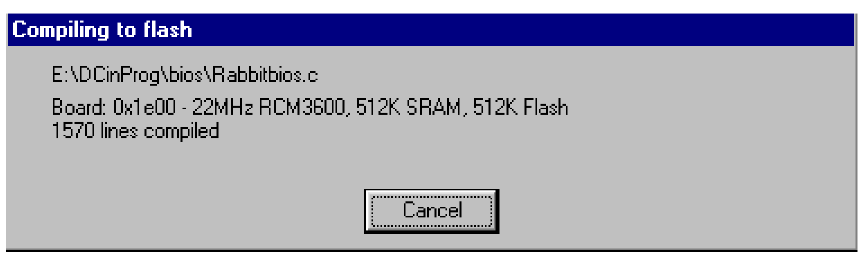
Any compilation errors are listed in the automatically activated Compiler Messages window. Press <F1> to obtain more information for any error message that is highlighted in this window.

Compile to Target

Expands to one of three choices. They override any BIOS Memory Setting choice made in the Compiler tab of the Project Options dialog.

- Compile to Flash
- Compile to RAM
- Compile to Flash, Run in RAM

Starting with Dynamic C 9, the compiler will show board type and other board specific information while doing a compile to target. The information shown will be identical to what the compiler already shows when compiling to a `.bin` file.



Compile to .bin File

Compiles a program and writes the image to a `.bin` file. There are two 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 the Compiler tab of the Project Options dialog. 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 “Targetless Tab” on page 283.

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 breakpoint 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, "Map File Generation."
- `DEMO1.rom` - ROM "output" file, containing redundant addresses (due to fixups); it's used to generate the BDL, BIN, HEX, and HDL files.

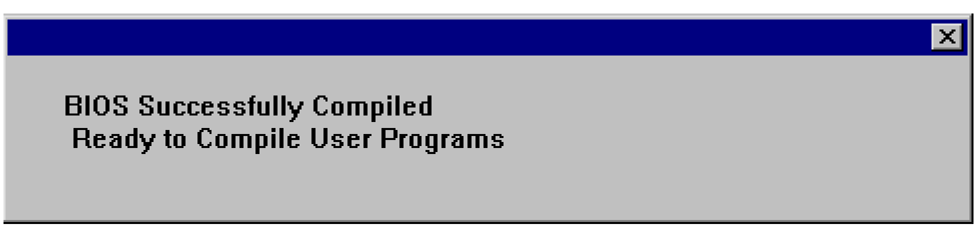
Reload RabbitSys binary

This option executes the command line RFU to reload the RabbitSys binary. You must have a target board with preloaded drivers to run RabbitSys.

Reset Target/Compile BIOS <Ctrl+Y>

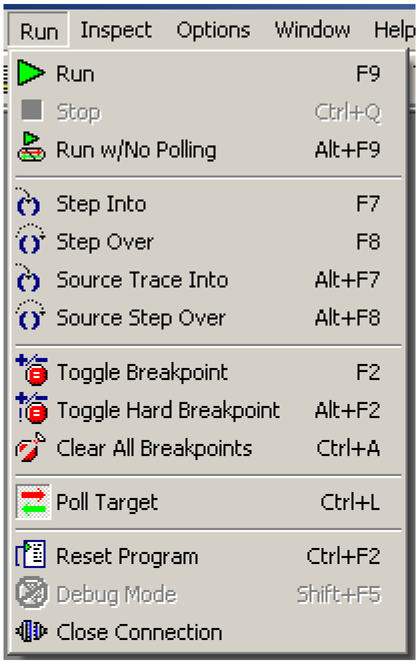
This option reloads the BIOS to RAM or Flash, depending on the choice made under BIOS Memory Setting in the Compiler dialog (viewable from Options | Project Options).

The following message will appear upon successful compilation and loading of BIOS code.



14.2.5 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 three 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.

Step 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 Step 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 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 and hardware breakpoints do.

Starting with Dynamic C 9, breakpoints can be toggled in edit mode as well as in debug mode. Breakpoint information is not only retained when going back and forth from edit mode to debug mode, it is stored when a file is closed and restored when the file is reopened.

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.

Starting with Dynamic C 9, breakpoints can be toggled in edit mode as well as in debug mode. Breakpoint information is not only retained when going back and forth from edit mode to debug mode, it is stored when a file is closed and restored when the file is reopened.

Clear All Breakpoints <Ctrl+A>

Clears all software breakpoints.

Poll Target <Ctrl+L>

A check mark means that Dynamic C will poll the target. The absence of a check mark means that Dynamic C will not poll the target. Prior to Dynamic C 7.30, this option was named “Toggle Polling;” however, now Dynamic C will not restart polling without the user explicitly requesting it.

If “Poll Target” is selected, Dynamic C sends a message to the target every three seconds and expects a response. If no response is received, Dynamic C ends the debugging session. Several things can be responsible for the target not replying to a polling message, such as loss of power, running in a loop with interrupts disabled, leaving interrupts disabled long enough to disrupt the serial port A ISR, or overwriting serial port A configuration, among other things. Polling does introduce overhead, but it is minimal since it only occurs every three seconds. Without polling turned on, Dynamic C will only notice an unresponsive target when the user attempts to do some other sort of debugging such as stopping the target, setting a breakpoint, single stepping, setting or evaluating a watch, etc.

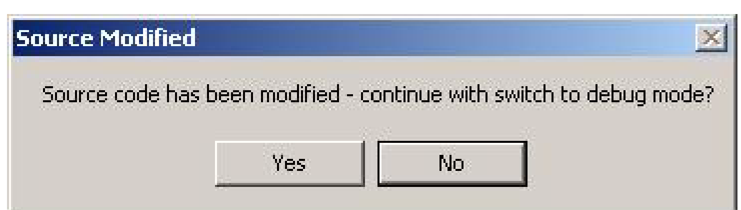
Reset Program <Ctrl+F2>

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.

Debug Mode <Shift+F5>

Dynamic C 9 introduces the ability to switch back to debug mode from edit mode without having to recompile the program. If the source file has been modified while in edit mode, a popup dialog lets you choose whether to run the non-modified code or to go ahead and recompile and download again.



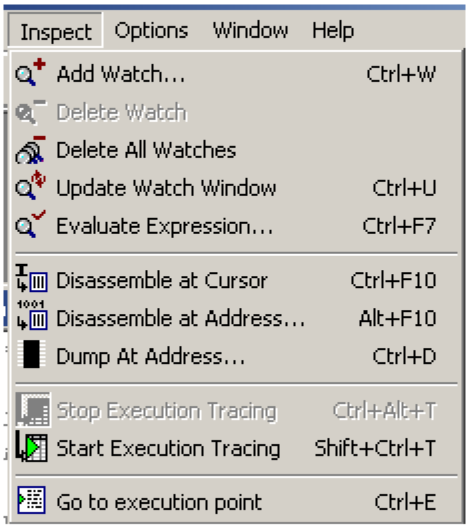
Close Connection

If using a serial connection, disconnects the programming serial port between PC and target so that the target serial port and the PC serial port are both accessible to other applications.

If using a TCP/IP connection, closes the socket between the PC and the RabbitLink or between the PC and the RabbitSys-enabled board.

14.2.6 Inspect Menu

Click the menu title or press <Alt+I> to open the Inspect menu.

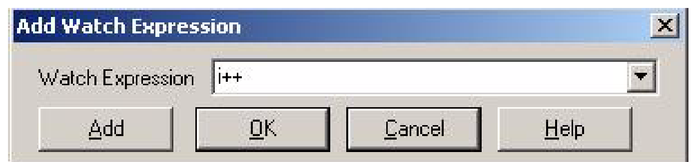


The Inspect menu provides commands to manipulate watch expressions, view disassembled code, and produce hexadecimal memory dumps. The Inspect menu commands and their functions are described here.

Add Watch <Ctrl+W>

This command displays the “Add Watch Expression” dialog. Enter watch expressions with this dialog box.

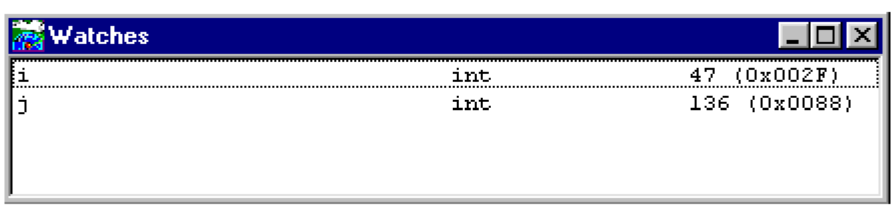
A watch expression may be any valid C expression, including assignments, function calls, and preprocessor macros. (Do not include a semicolon at the end of the expression.) If the watch expression is successfully compiled, it and its outcome will appear in the Watches window.



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.

Starting with Dynamic C 9, structure members are displayed whenever a watch expression is set on a struct. Prior to Dynamic C 9, separate watch expressions had to be added for each member. Introduced in Dynamic C 8.01, the Debug Windows tab of the Environment Options menu lets you set flyover hint evaluation of any expression that can be watched without having to explicitly set the watch expression. See “Watch” on page 286 and “Watch Window” on page 268 for more details.

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 Watches window to be updated while running in nodebug code. Normally the Watches 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).

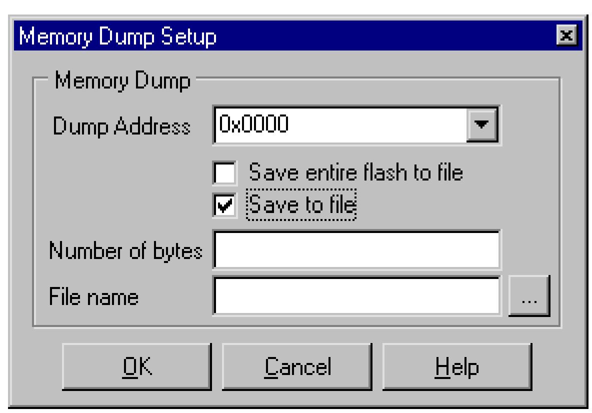
The Disassembled Code window displays the result. See “Assembly (F10)” on page 287 for details about this window.

Dump at Address <Ctrl+D>

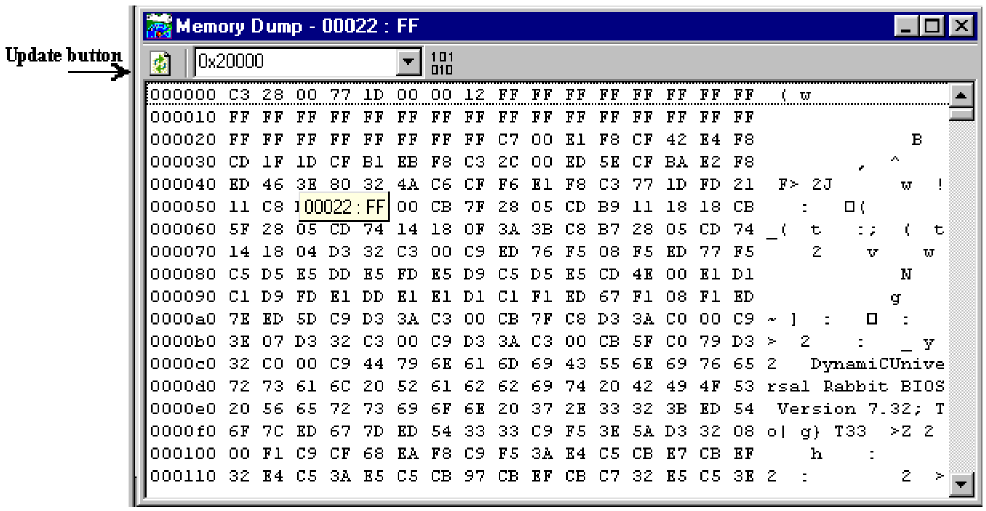
Allows blocks of raw values in any memory location to be displayed. Values are displayed on the screen or written to a file. If separate I&D space is enabled, you can choose which logical space to examine: instruction space or data space.

Dynamic C 9 introduced differences highlighting when displaying to the screen: each time you single step in C or assembly changed data is highlighted in reverse video in the Memory Dump window. (This is also true for the Stack and Register windows.)

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, a 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.



Memory Dump windows 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 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.

Note that adding a leading zero to a logical address makes it a physical address.

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.

Stop Execution Tracing <Ctrl+Alt+T>

This command causes the target to stop sending trace information to Dynamic C. You can also do this from within your program with the `_TRACEOFF` macro. The sample program `Samples/Demo4.c` describes and uses this trace macro.

Start Execution Tracing <Shift+Ctrl+T>

This command causes the target to send execution tracing information to Dynamic C based on the trace options you choose in the Debugger tab of the Project Options dialog. You can also do this from within your program with the `_TRACE` and `_TRACEON` macros. The sample program `Samples/Demo4.c` describes and uses these trace macros.

Trace entries received are displayed in the Trace window (see Stack Trace (Ctrl+T)). This menu command is only available if tracing is enabled in Project Options and Dynamic C is in run mode.

Note that turning on tracing causes a performance hit to your program because of the extra communication required between Dynamic C and the target. If your program requires precise timing, tracing may interfere.

Goto execution point <Ctrl+E>

When stopped in debug mode, this option places the cursor at the statement or instruction that will execute next.

14.2.7 Options Menu

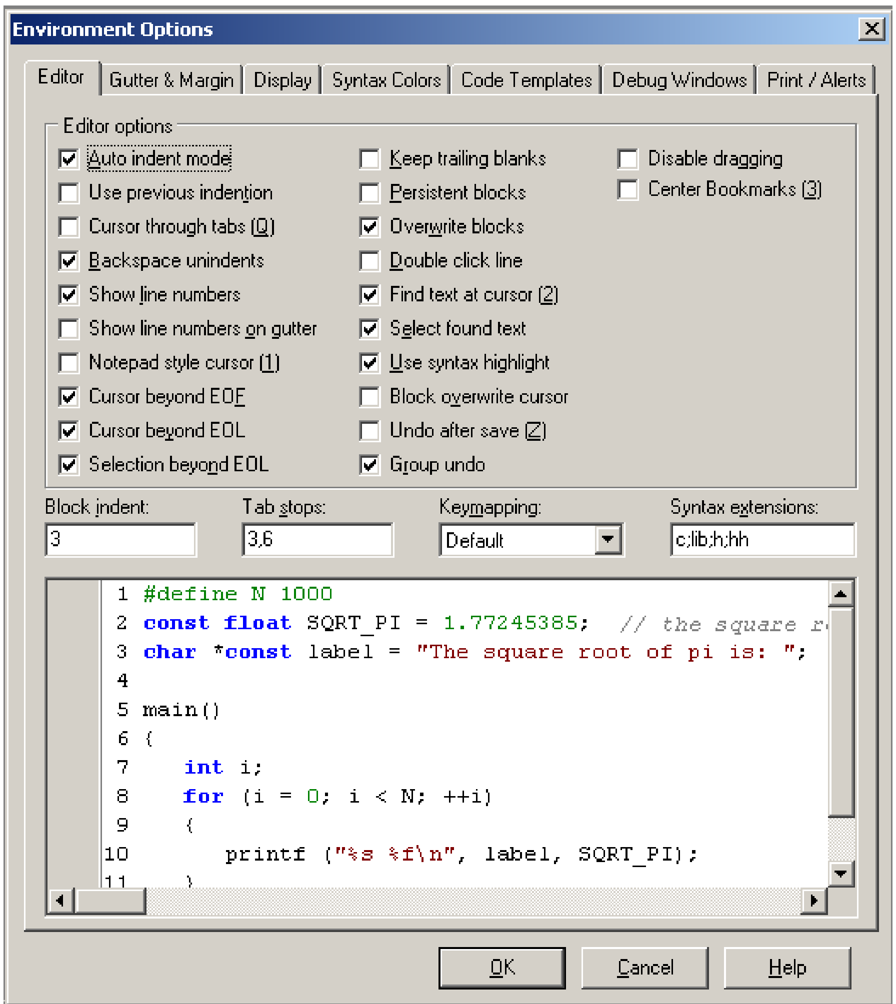
Click the Options menu title or press <Alt+O> to select the Options menu.

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. Note that keyboard shortcuts for some of the options have no character to underline, so the character is shown between brackets, thus, when the Editor menu options are visible, Alt+Q is the keyboard shortcut for toggling the option “Cursor through tabs”.

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. When this option is checked, the “Use Syntax Highlighting” in the edit window’s right-click menu allows you to toggle the syntax highlighting in the active file.

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

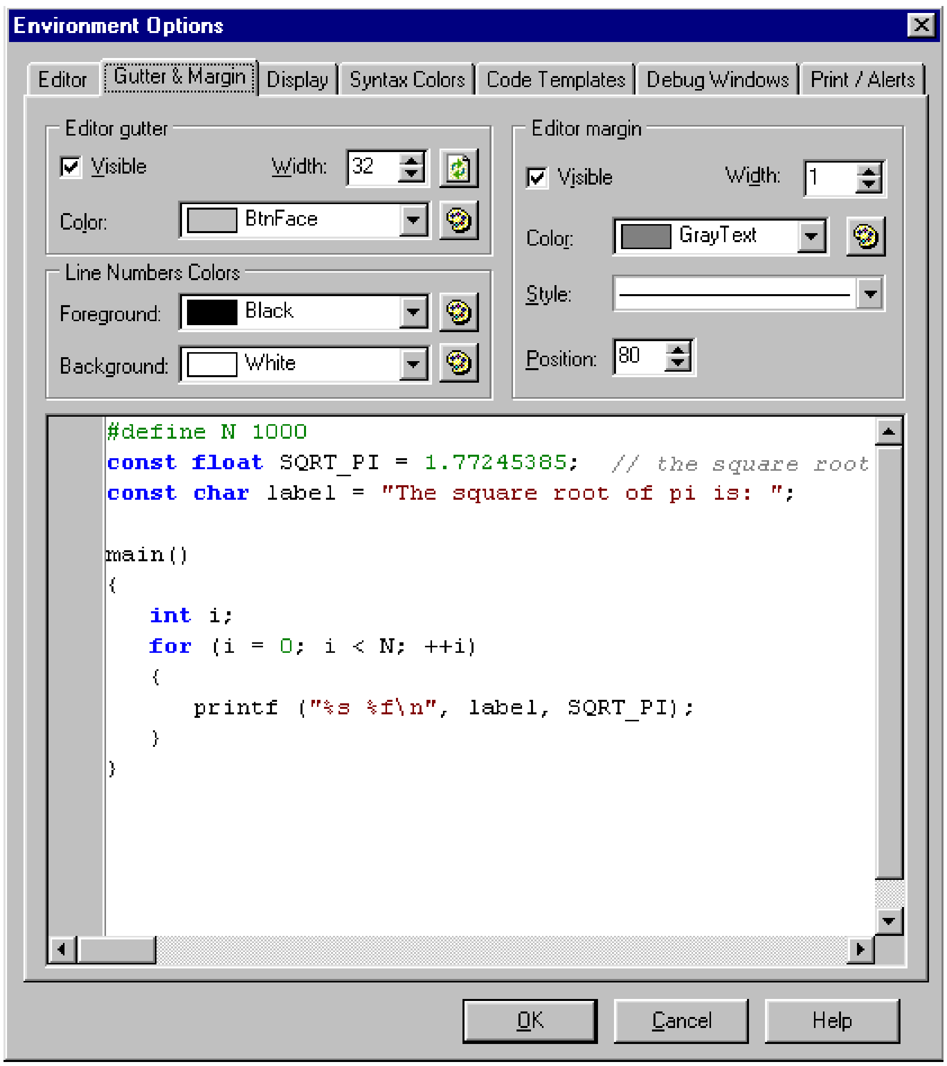
The keyboard has five different default key mappings: Default, Classic, Brief, Epsilon and Visual Studio. Change the keymapping with this pulldown menu.

Syntax extensions

Dynamic C will automatically syntax highlight the text in all files with the extensions listed here. Syntax highlighting can also be enabled by right-clicking on an open file and selecting the “Use Syntax Highlighting” menu item.

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

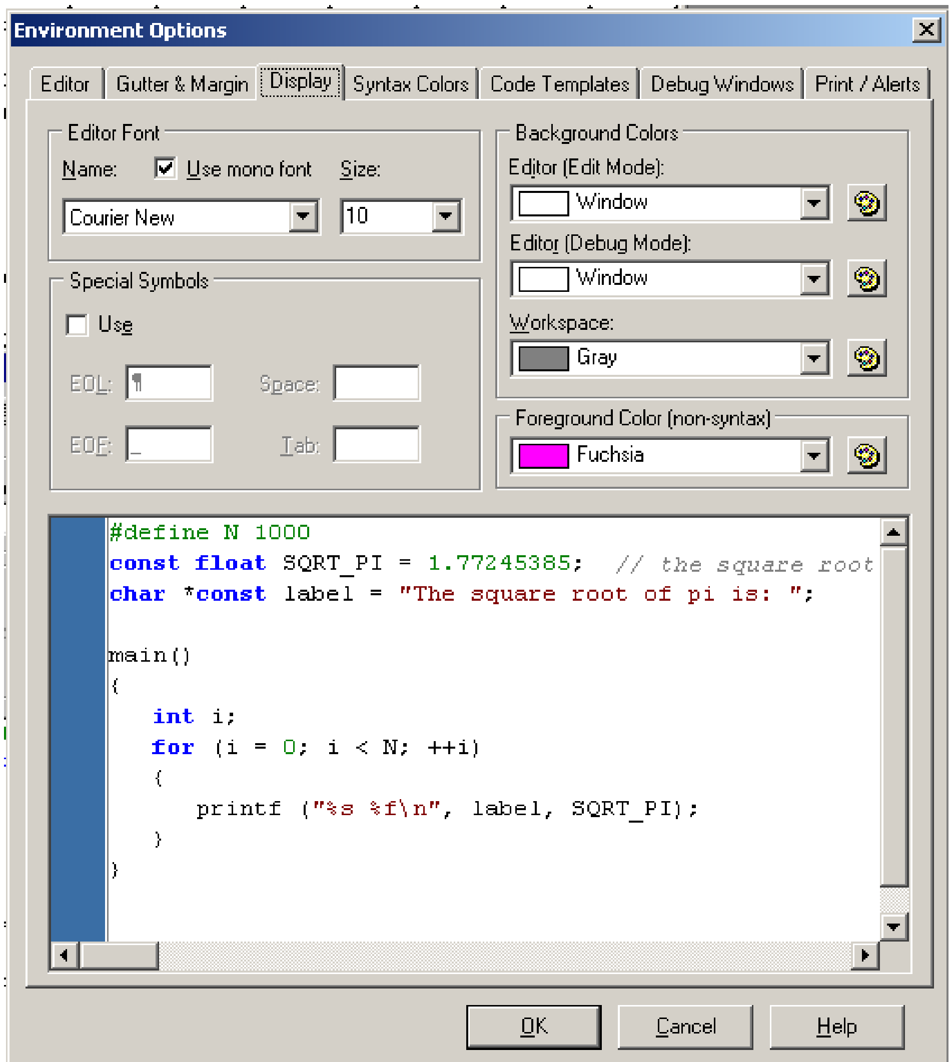
Check the Visible box to create a right-hand margin in the text window. Use the Width scroll bar and the Color pulldown menu to set the like-named attributes of the margin line. The Style pulldown menu displays the line choices available: a solid line and various dashed lines. The Position scroll box 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 notplaced 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 the box labeled “Use” to view end of line, end of file, space and/or tab symbols in the editor window.

Background Colors

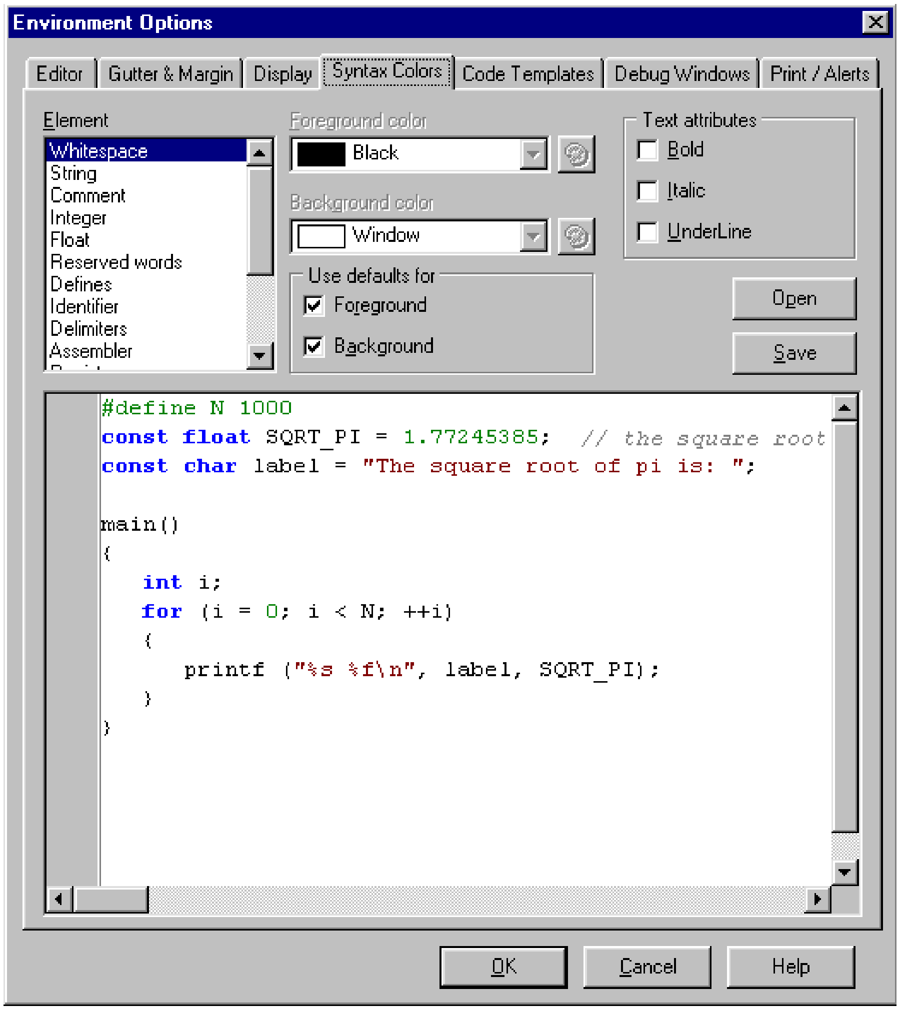
This area of the dialog box is for choosing background colors for editor windows and the main Dynamic C workspace. The editor window can have a different background color in edit mode than it does in run mode. Each pulldown menu has an icon to the right that brings up additional color choices.

Foreground Color (non-syntax)

If syntax highlighting is not used, the color selected here will be the foreground color used in the editor file.

Syntax Colors Tab

Click on the Syntax Colors tab to display the following dialog.



Element

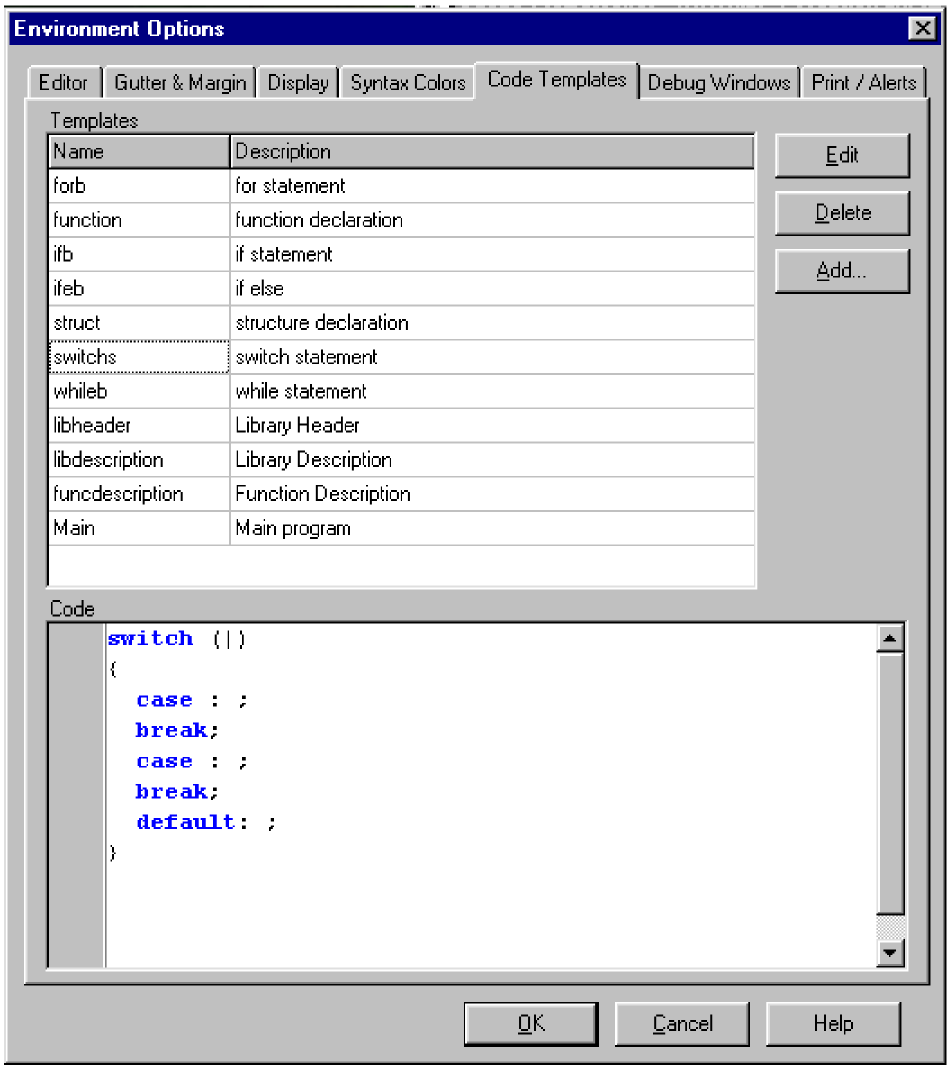
In this text box are the different elements that may be in a file (strings, comments, integers, etc.). For each one you may choose a foreground and a background color. You may also opt to use the default colors: black for foreground and white for background. In the Text attribues area of the dialog box, 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 Templates Tab

Click the Code Template tab to display the following dialog.

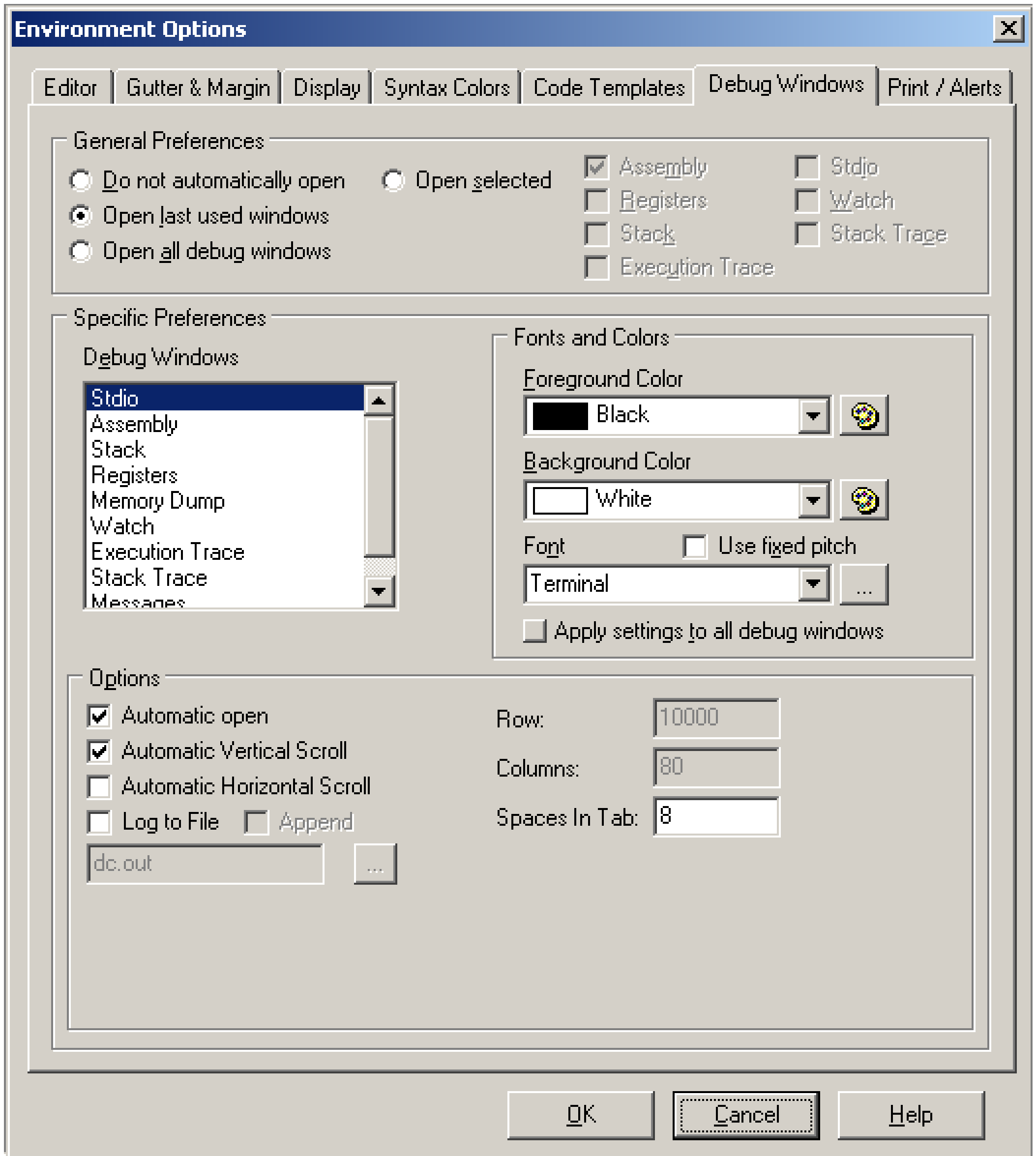


As you can see, there are several predefined templates. The Edit and Delete buttons allow the like-named operations on existing templates. The Add button gives the ability to create custom templates.

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

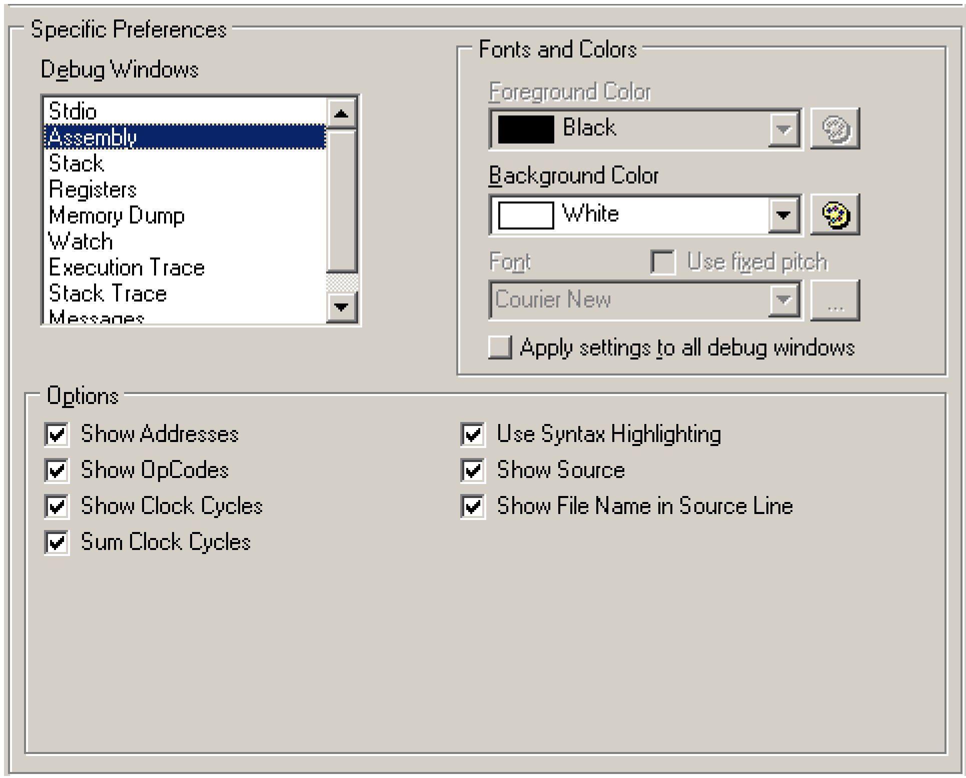
Tab stops display as the number of spaces specified here.

Starting with Dynamic C 9, the various Find commands available on the Edit menu can be used directly in the Stdio window.

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 shown below. For more information about this window see Section 11.4.3.



Show Addresses

Check this to show the logical address of the instruction in the far left column.

Show OpCodes

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.

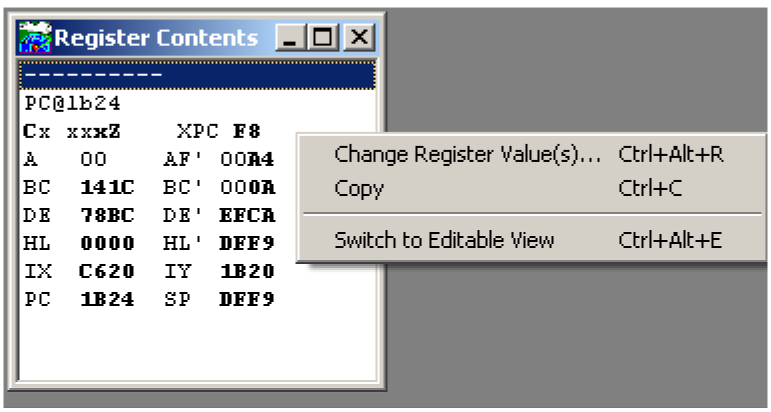
Register Window

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 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.

Starting with Dynamic C 9, each time you single step in C or assembly changed data is highlighted in the Register window. (This is also true for the Stack and Memory Dump windows.)

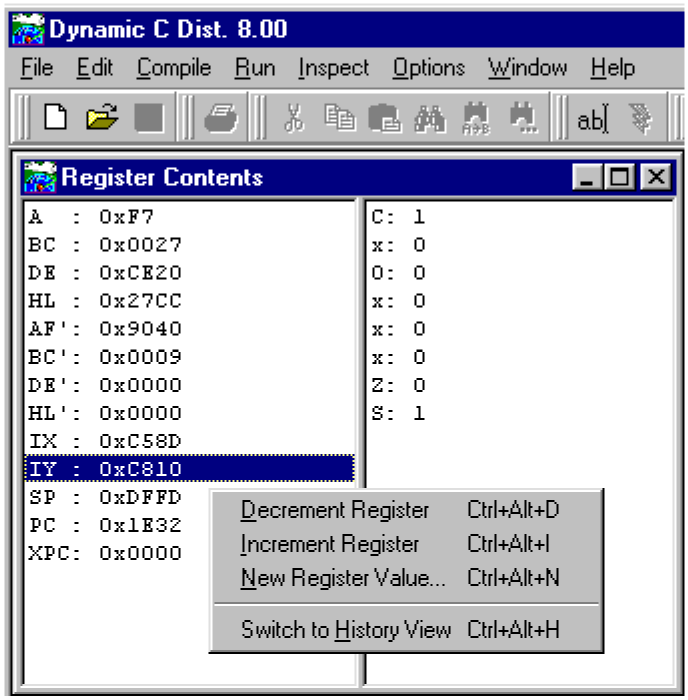


A click of the right mouse button brings up the menu pictured above. Choosing Change Register Value(s)... brings up a dialog where you can enter new values for any of the registers, except SP, PC and XPC.

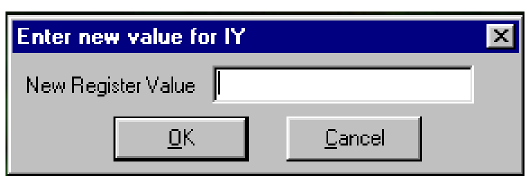
Show registers as editable

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.

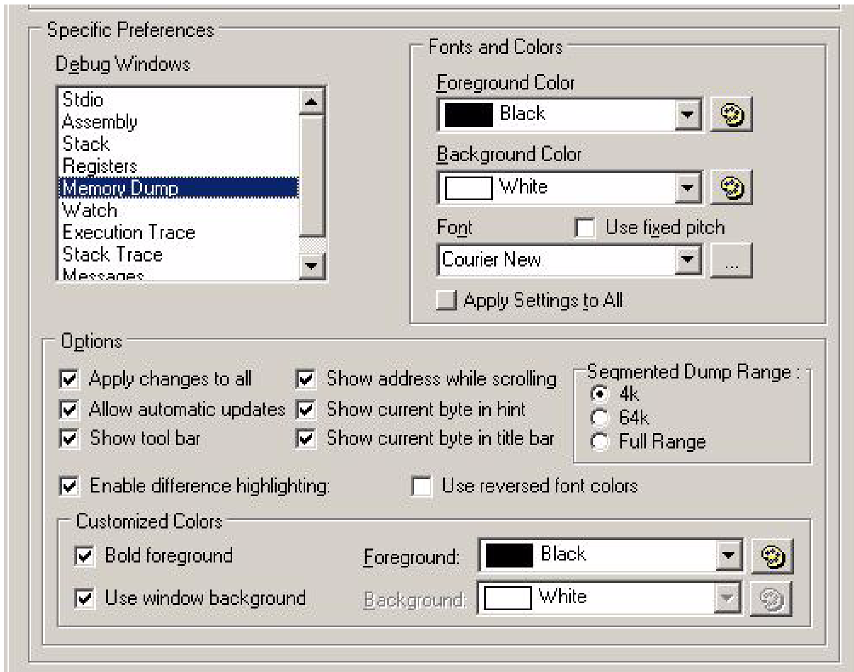


The option New Register Value will bring up a dialog to enter the new register value. Hex values must have “0x” prepended to the value. Values without a leading “0x” are treated as decimal.

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 Window

For more information on using the Memory Dump window go to the section titled, Dump at Address <Ctrl+D>.



The following are the options relevant to the Memory Dump window.

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.). Starting with Dynamic C 9, each time you single step changed data in the memory dump window is highlighted in reverse video.

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.

Watch Window

The Watches window configuration options, Enable watch expression evalution in flyover hint and Show watch expression evaluation errors in flyover hint, do not actually affect the Watches window. When checked, they allow you to use flyover hints in the source code window to see the value of watchable expressions.

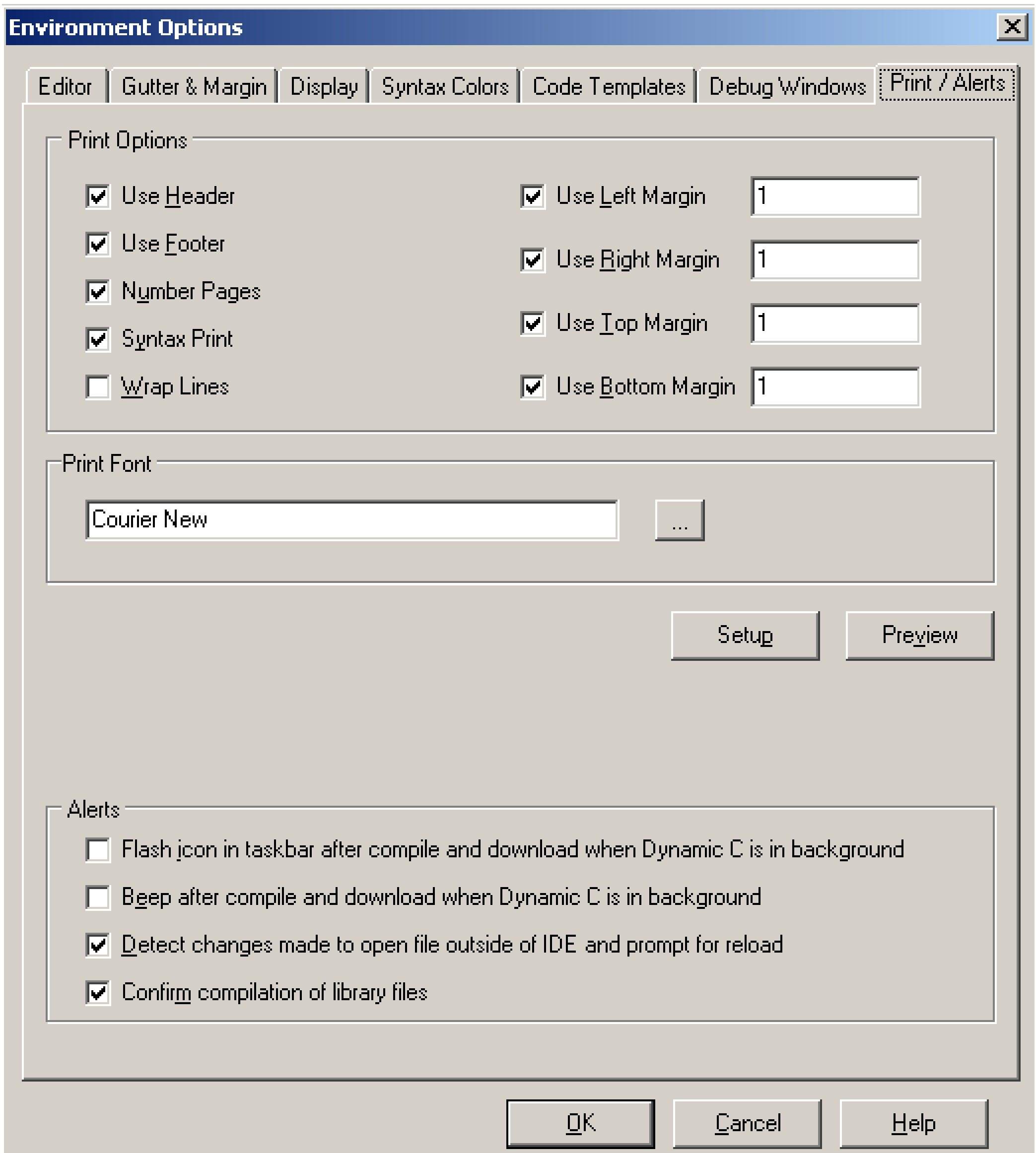
Move the cursor over a variable to see its current value and over a function to see its logical address or its return value. If you highlight the name of a function (e.g., `my_function`) you will see the location of the code in memory. If you highlight the function call (e.g., `my_function(`*my_parm*`)`) the function will be called and you will see its return value. If the cursor is over a structure member, the flyover hint will only contain information about the structure, not the individual member.

Stack Trace Window

There are no configuration options for the Stack Trace window.

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.

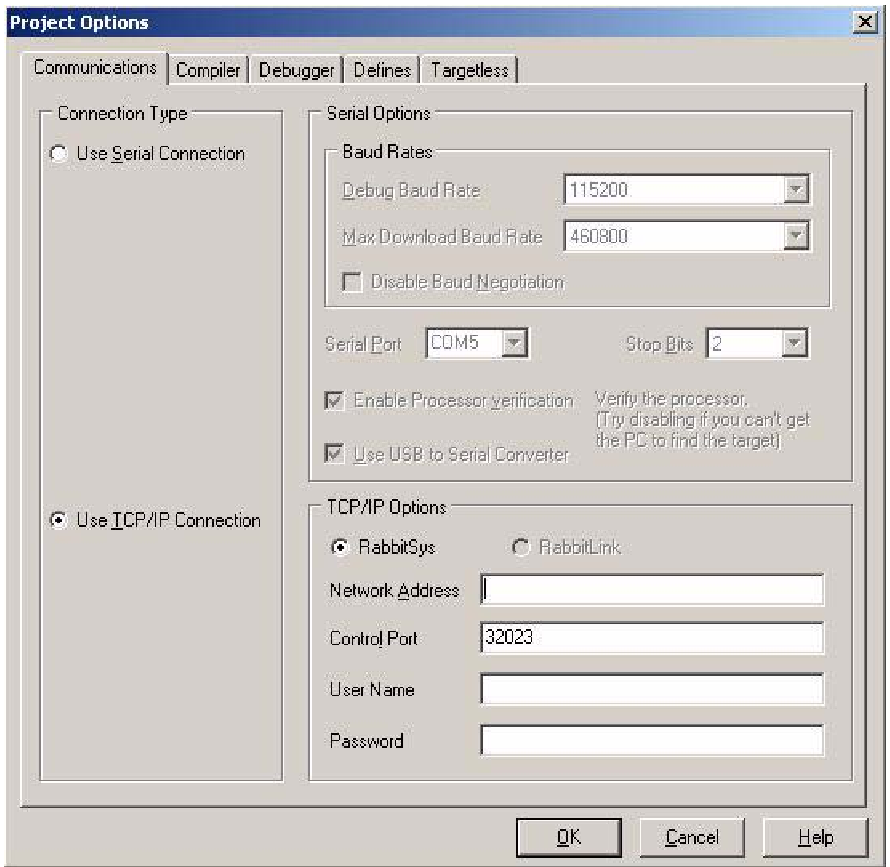
There are four checkboxes in the Alerts area of this dialog. The first two signal a successful compile and download, one with a visual signal, the other auditory. The third checkbox detects if a file that is currently open in Dynamic C has been modified by an external source, i.e., a third-party editor; 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 drop-down menu lists PC COM ports that may be connected to the Rabbit-based target. The default is COM1. Starting with version 9.60, Dynamic C identifies which ones are USB ports.

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 that 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.

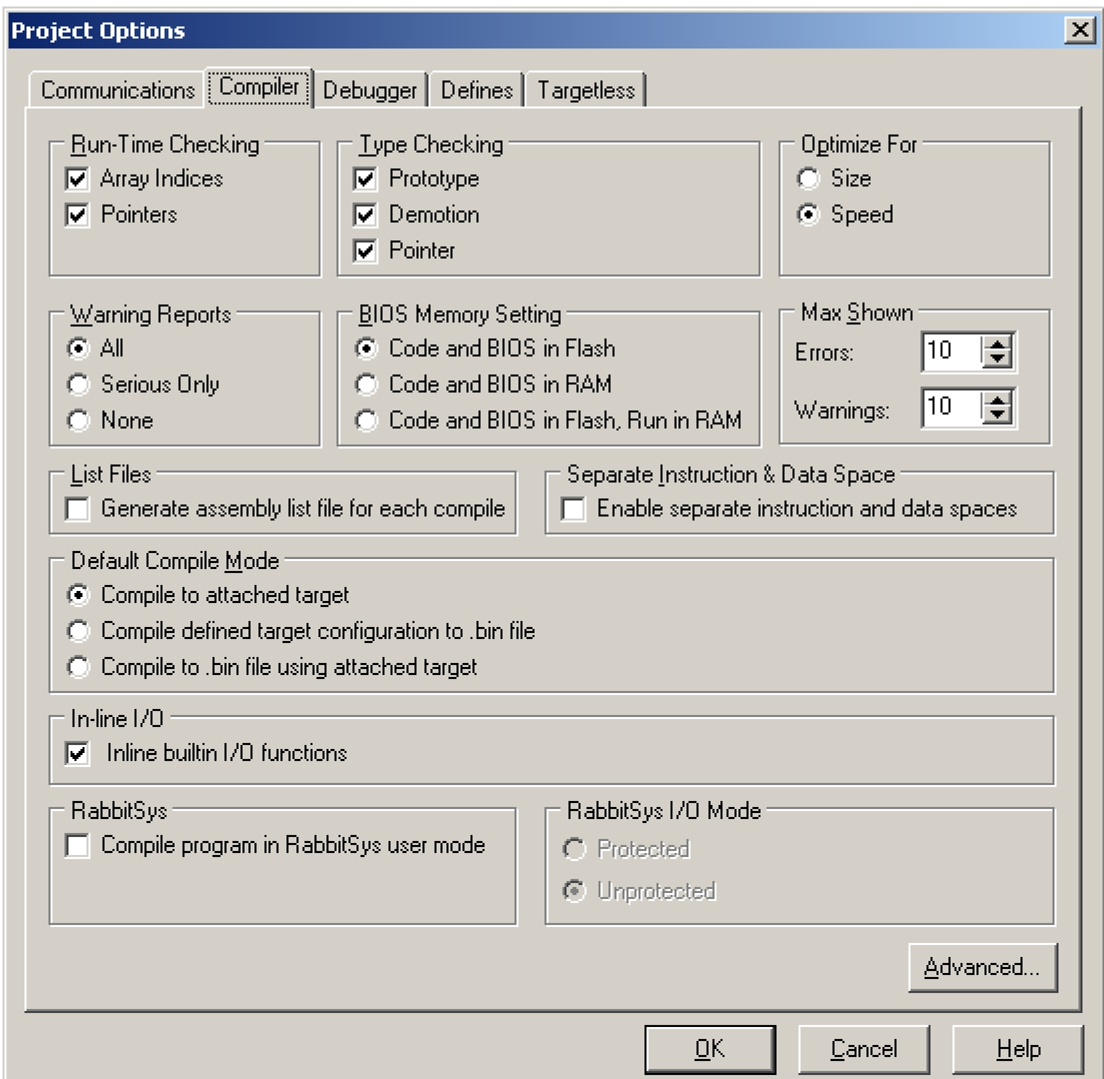
TCP/IP Options

To program and debug a controller across a TCP/IP connection, the Network Address field must have the IP address of either the RabbitLink board that is attached to the controller, or the IP address of a RabbitSys-enabled controller.

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 or RabbitSys-enabled controllers attached to the network. Any such boards 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.

Functions marked as nodebug disable the run-time checking options selected in the GUI.

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. Rabbit 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. Note that this option cannot work for boards with serial boot flashes. These boards should use Code and BIOS in Flash, Run in RAM.
- 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 coldloader.
- 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) and should be used for Rabbit core modules with serial boot flash.

  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, opcode, instruction and number of clock cycles. See page 287 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. The setup for targetless compile may differ for some board series. Please check your user manual for differences in setup.

- 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.

RabbitSys

This option was added in Dynamic C 9.30. Checking it allows you to compile a program to run on top of RabbitSys. The target board must be RabbitSys-enabled, which means that it has the necessary preloaded drivers and the RabbitSys firmware.

For more information about RabbitSys, see the *RabbitSys User's Manual*.

RabbitSys I/O Mode

The radio buttons labeled "Protected" and "Unprotected" choose between the available RabbitSys I/O protection modes.

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 (same as the command line compiler option "-lf")

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 the "lib.dir" file. Library files must be listed in the "lib.dir" file (or its replacement) to be available to a program.

If the function description headers are formatted correctly (See "Function Description Headers" on page 44.), the functions in the libraries listed in the user-defined replacement for the "lib.dir" file will be available with <Ctrl+H> just like the user-callable functions that come with Dynamic C.

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.

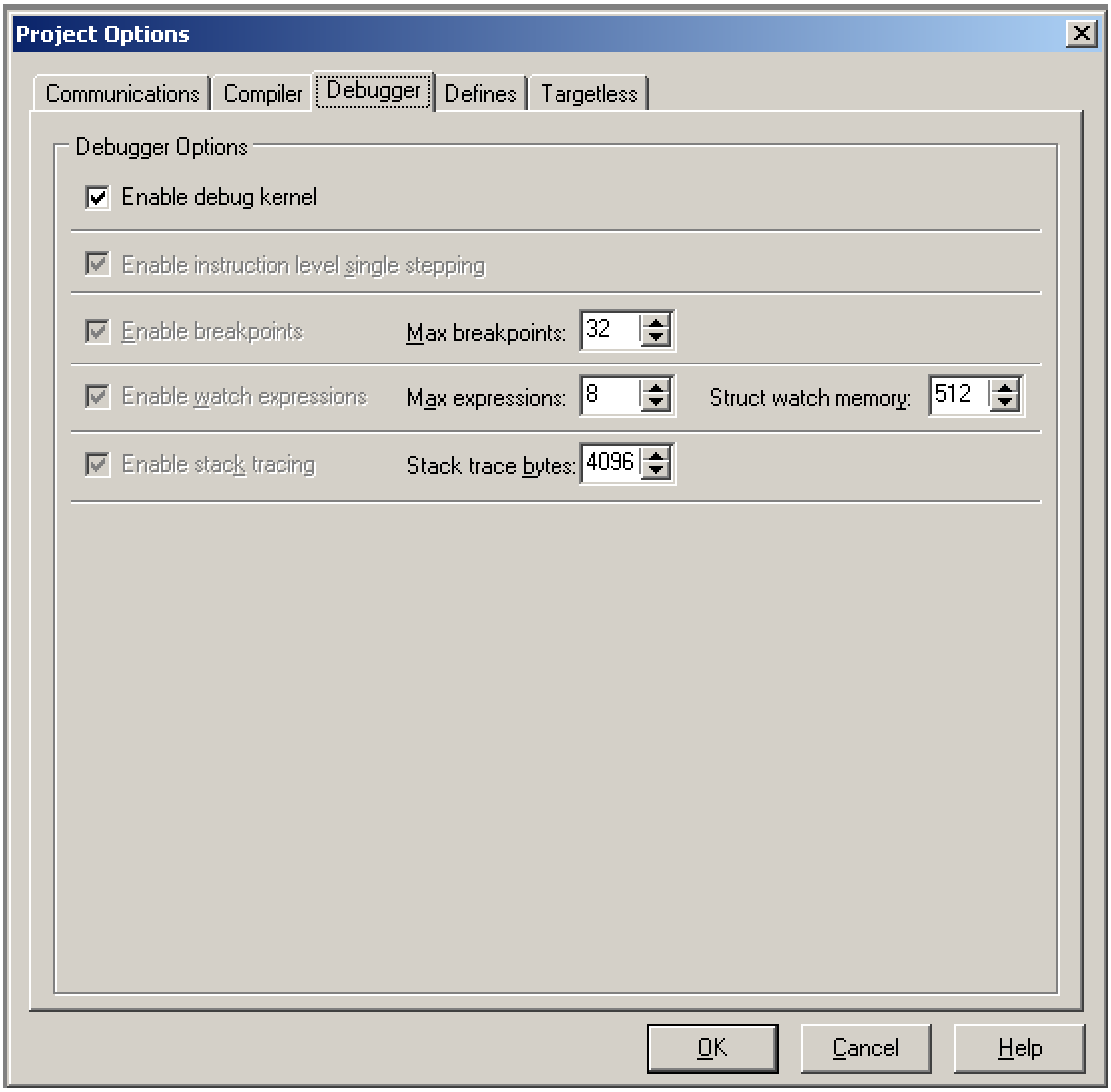
Include BIOS

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.

When you are compiling a program to the attached target controller, the BIOS is always included.

Debugger Tab

Click on the Debugger tab to display the following dialog. This is where you enable/disable debugging tools. Disabling parts of the debug kernel saves room to fit tight code space requirements.



Enable debug kernel

This option was added in Dynamic C 9.30. Leaving it unchecked allows you to compile your application without the debug kernel. You must check this option to set any of the other debug options.

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.

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 box 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 number of watches will slightly reduce the amount of root RAM used.

With the watch expression box 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.

With Dynamic C 9, watch expressions are enhanced to automatically include the addition of structure members when a watch expression is set on a struct. Some extended memory is reserved for handling watch expressions on structs. As shown in the above screen shot, 512 bytes of xmem is reserved by default. This can be changed to anything in the range 32 to 4096. Be aware that this watch memory is a tradeoff: not only does it dictate the number and complexity of watched structs, but also impacts the amount of memory available for `xalloc()` calls.

Enable stack tracing

Dynamic C 9 introduces stack tracing. If this box is checked the Stack Trace window is available to show the function call sequence leading to any point at which the program is stopped. The Stack Trace window shows a concise history of the execution path and values of local variables and function arguments that led to the current breakpoint, all for a very small cost in execution time and BIOS memory.

To the right of the checkbox is a spin/edit box for entering the maximum number of bytes of the current stack to transfer from the target at each breakpoint. The allowable range is 32 bytes to 4096 bytes inclusive. The default is 4096 bytes. If the stack depth is smaller than the number in this spin/edit box, only the depth number of bytes is transferred.

With the “Enable stack tracing” box unchecked, the debug kernel and the user program will be compiled without stack tracing support. Changing the status of the checkbox or the number of stack trace bytes forces a recompilation of the BIOS the next time the user program is compiled.

See “Stack Trace (Ctrl+T)” on page 290 for details on using this debug window.

Enable execution tracing

If this is checked, the target will send trace information back to Dynamic C when you turn on tracing by choosing Inspect | Start Execution Tracing or when your program does so by executing a `_TRACE` or `_TRACEON` macro. Unchecking this box will disable the menu command and macros.

Note that enabling tracing here will cause more code to be compiled into the BIOS, meaning there is less memory available on the target for your program, so if you get insufficient memory errors with your program, disabling tracing might help. Also, when you turn on tracing from the menu or a macro, your program will suffer a performance hit because of the extra communication required between Dynamic C and the target.

Trace Buffer (PC)

The trace buffer allows you to specify how much memory is allocated on your computer (the default is 64 megabytes) to hold trace entries received from the target. If you check the "Wrap" box, new trace entries overwrite existing ones when the buffer fills up, starting with the oldest. When "Wrap" is unchecked, any entries received after the buffer fills up are discarded.

The number of entries displayed is the maximum number of trace entries the buffer will hold given the size of the trace buffer you specify and the Trace window information fields you select.

Trace Level

Choose which events will be captured by the trace. Full tracing captures all debuggable statements plus function entries and exits. If you don't want to include all statements, you can choose to capture each function entry and exit only.

Dynamic C statements are debuggable by default, while assembly code is not. You can toggle this with the debug and nodebug keywords for Dynamic C functions, and with the debug and nodebug options of the #asm compiler directive for blocks of assembly code.

Trace Window Fields to Trace

You can select the trace information captured from the target and displayed in the Trace window. You can include the function name, file name, and line and column where each trace entry originated; the type of action being performed; the time stamp when the action was performed; and the contents of the registers. The more fields you select to be displayed in the Trace window, the larger each entry, and so the fewer entries the trace buffer can hold.

Saving Trace Window to a File

Checking the "Save on program termination" box will cause Dynamic C to write the contents of the trace buffer to a file when your program terminates. When this box is checked, you must specify the filename and location where you want to save.

Note that this feature saves the contents of the trace buffer at the time your program terminates, so if the buffer fills up while your program is running not all trace entries received will be written to the file. If you want to save trace entries before they are lost, you can do so at any time from the Trace window. See Execution Trace (Alt+ F12) for details.

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 supersede 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

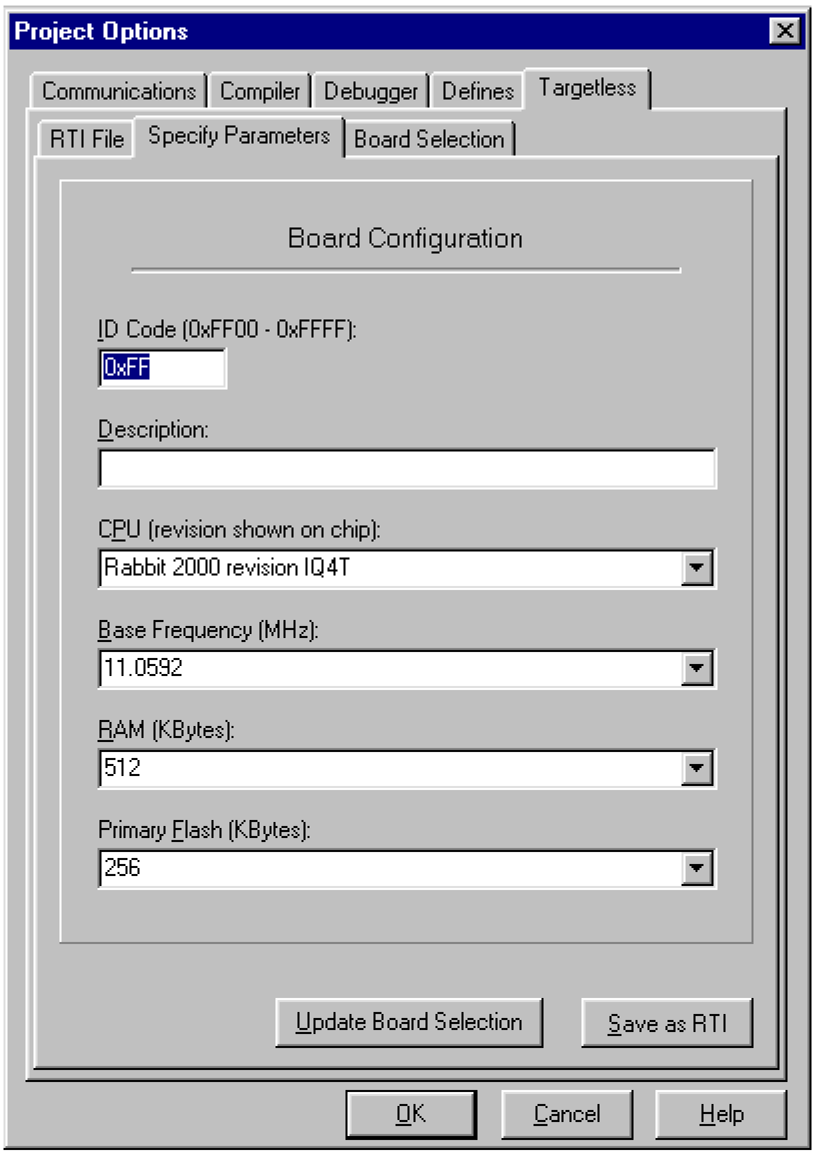
Click on the Targetless tab to reveal three additional tabs: RTI File, Specify Parameters and Board Selection. The setup for targetless compile may differ for some board series. Please check your user manual for differences in setup.

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 term "Primary Flash" refers to the Flash device connected to /CS0, not the total amount of Flash available on the board.

The result may be saved to a RTI file for later use, or the result may be saved to the list of board configurations.

Board Selection

The list of board configurations 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.

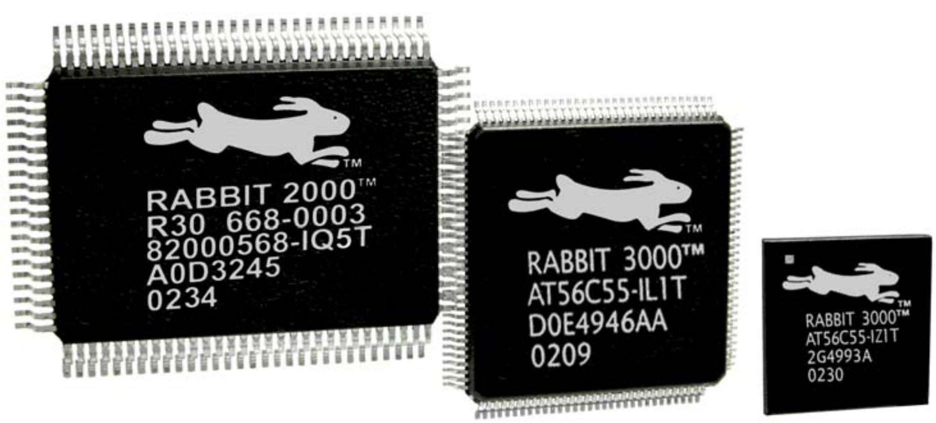
If you save to the list of board configurations by clicking on the button Update Board Selection, then you must fill in all fields of the dialog. The baud rate, calculated from the value in the Base Frequency (MHz) field, only applies to debugging. The fastest baud rate for downloading is negotiated 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 third line from the top on the Rabbit 2000 and the second line from the top on the Rabbit 3000. The table below lists the possible values.

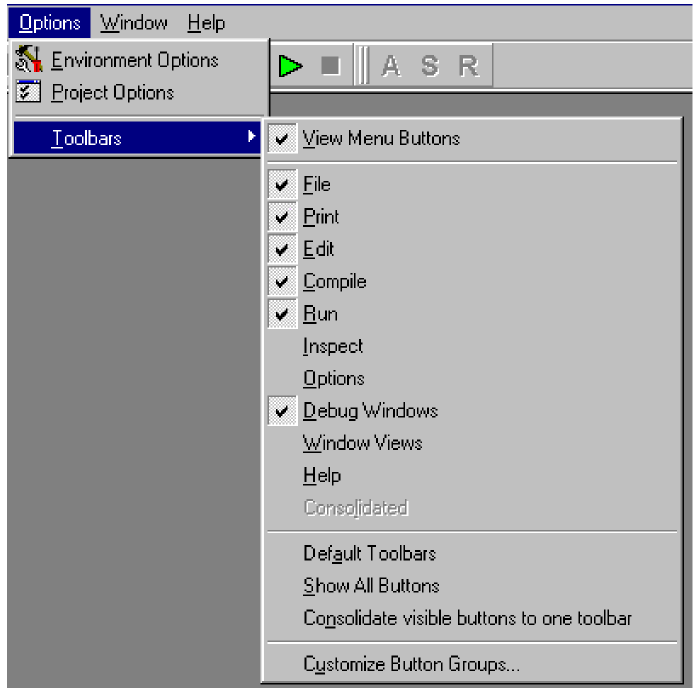
Rabbit Microprocessor	non-RoHS	RoHS
Rabbit 2000	IQ#T	UQ#T
Rabbit 3000	IL#T or IZ#T	UL#T

Where "#" is the revision number and the letters are associated information.



Toolbars

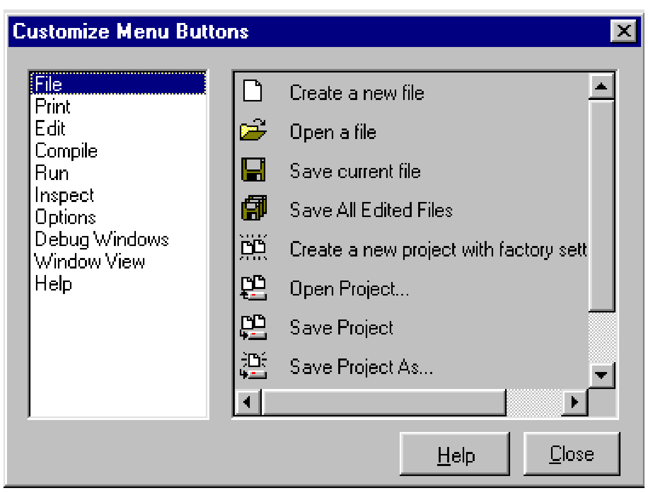
Selecting this menu item reveals alist of all menu button groups, i.e., the groups of icons that appear in toolbars beneath the title bar and the main menu items (File, Edit, …). This area is called the control bar. Uncheck View Menu Buttons to remove the control bar from the Dynamic C window. Any undocked 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 to make its toolbar visible on the control bar.

To quickly return to showing only 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 that, enables the option Consolidated, which toggles the visibility of the consolidated toolbar, even when it is undocked from the control bar.

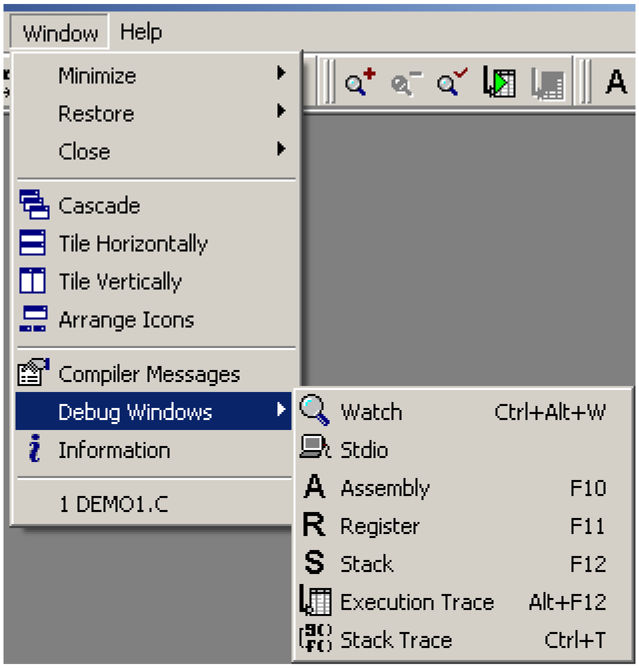


Select "Customize Button Groups" to bring up the Customize Menu Buttons window. This window allows you to change which buttons are associated with which button group on the toolbar. Choose a button group on the left side of the window; this causes the icons for the buttons in that group to display on 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.

14.2.8 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.

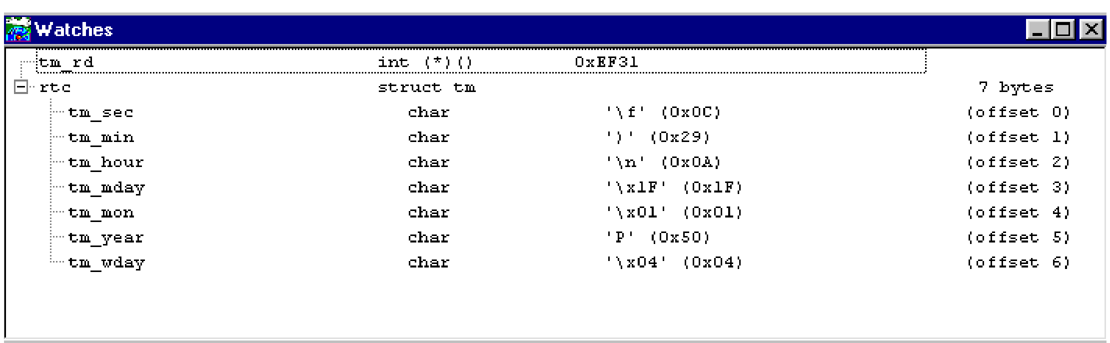
The Compiler Messages option is a toggle for displaying that window. This is only available if an error or warning occurred during compilation.

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, i.e., make it the active window.

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 watch expressions whenever Dynamic C evaluates watch expressions. Starting with Dynamic C 9, a watch expression for a structure will automatically include all members of the structure. Previous versions of Dynamic C required each struct member to be added as a separate watch expression.



Keep in mind that when single stepping in assembly, the value of the watch expression may not be valid for variables located on the stack (all auto variables). This is because the debug kernel does not keep track of the pushes and pops that occur on the stack, and since watches of stack variables only make sense in the context of the pushes and pops that have happened, they will not always be accurate when assembly code is being single stepped.

Stdio

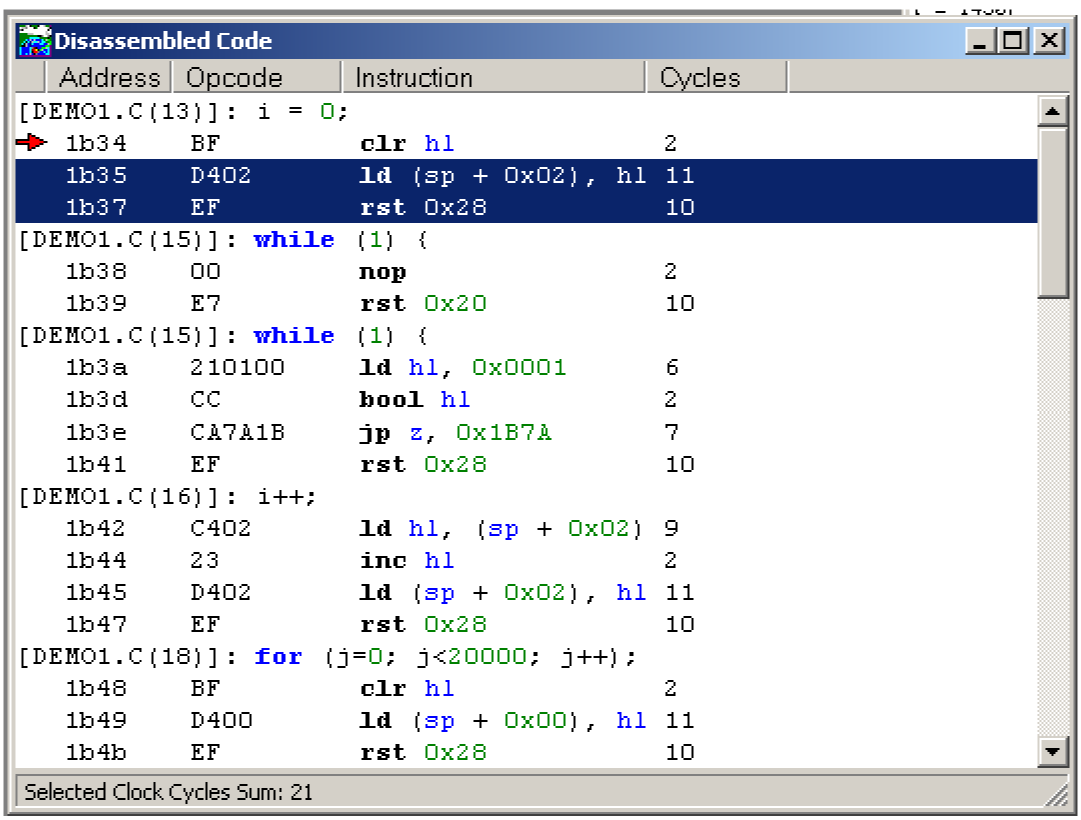
Select this option to activate or deactivate the Stdio window. The Stdio window displays output from calls to `printf()`. If the program calls `printf()`, Dynamic C will activate the Stdio window automatically if it is not already open, unless “Automatic open” is unchecked in the Debug Windows dialog in Options | Environment Options.

Starting with Dynamic C 9, the various Find commands available on the Edit menu can be used directly in the Stdio window.

Assembly (F10)

Select this option to activate or deactivate 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.

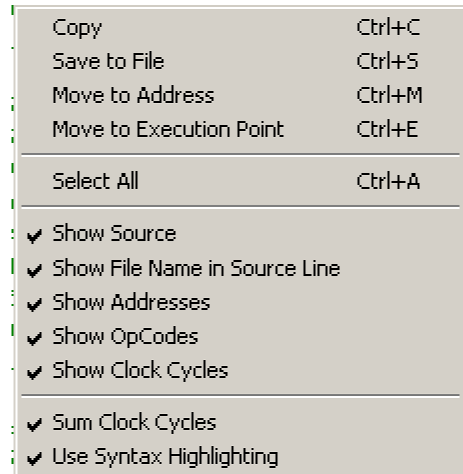
The Disassemble at Cursor or Disassemble at Address commands from the Inspect menu also activate the Disassembled Code window.



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 opcode bytes, 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.

Select ALL

Selects all text 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.

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 “Opcode” 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.

Register (F11)

Select this option 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 (also known as the flags register). The window also shows the source-code line and column at which the snapshot of the register was taken.

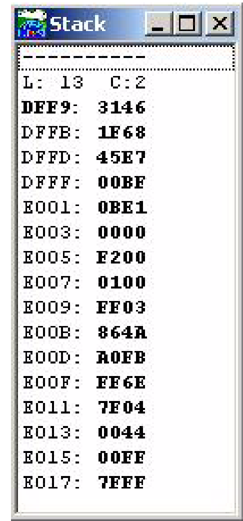
It is possible to scroll back to see the progression of successive register snapshots. Register values may be changed when program execution is stopped Registers PC, XPC, and SP may not be edited as this can adversely affect program flow and debugging.

See “Register Window” on page 265 for more details on this window.

Stack (F12)

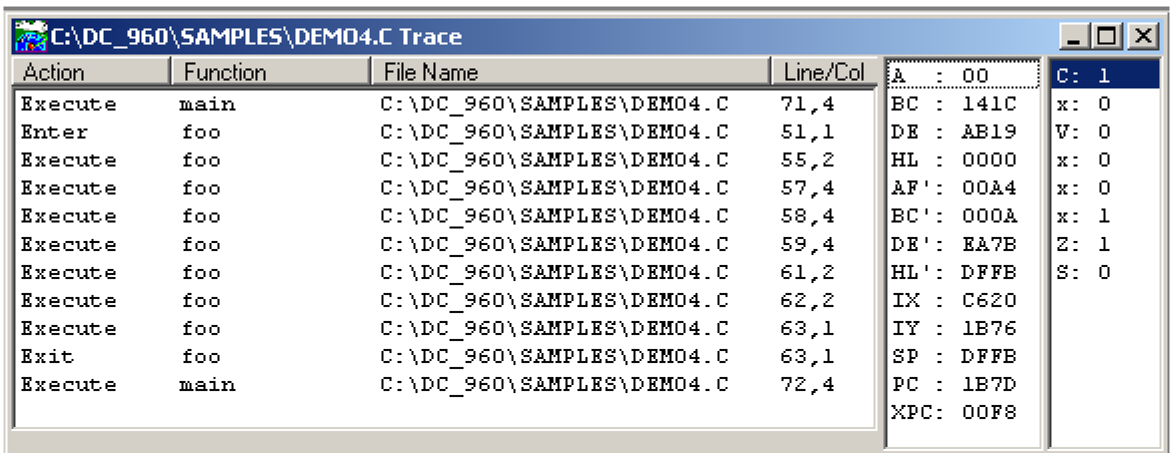
Select this option to activate or deactivate the Stack window. The Stack window displays the top 32 bytes of the run-time stack. It also shows the line and column at which the stack "snapshot" was taken. It is possible to scroll back to see the progression of successive stack snapshots.

Dynamic C 9 introduced differences highlighting: each time you single step in C or assembly, changed data can be highlighted in the Stack window. (This is also true for the Memory Dump and Register windows.)



Execution Trace (Alt+ F12)

Select Execution Trace to activate or deactivate the Execution Trace window. The fields displayed in this window were specified in the Debugger dialog box that is accessed via the Options | Project Options menu (see Enable execution tracing).



The Trace window has a right-click pop-up menu. An option on this menu controls the display of an additional column in the Trace window. If Group repeated statements is selected, the Show Repeat Count may also be selected and will display in the rightmost column of the Trace window that comes before the register contents column. A value displayed under Show Repeat Count is the number of times the corresponding statement has been executed and, therefore, traced. The Timestamp column is not updated for subsequent traces of a repeated statement.

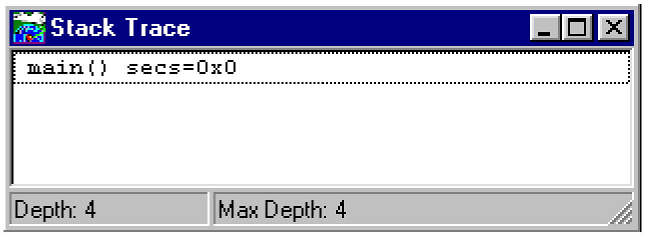
The Group repeated statements option is useful when tracing statements inside a loop.

The rest of the pop-up menu options are more or less self-explanatory. You can choose to open the source code for any function in the Trace window by selecting the function and choosing Open Source. In the above screenshots, note that a trace statement for `kbhit()` is selected in the Trace window. Choosing Open Source in this situation would open a window for `STDIO.LIB`, the library file that contains the function `kbhit()`.

You can also toggle auto scroll, as well as decide whether to display the complete path in the File Name column. The last three menu options are for saving Trace window contents to another file. You can select trace statements in the window and then using Copy selected traces or Copy with header you can paste the selected traces anywhere you can perform a paste operation. You can also choose to copy the entire contents of the current Trace window to a named file. This is similar to the option in the Debugger tab of the Project Options dialog, which allows saving the Trace window to a file upon program termination.

Stack Trace (Ctrl+T)

The Stack Trace window displays the call sequence and the values of function arguments and local variables of the currently running program. The screenshot shown here is the Stack Trace window when `Samples/Demo3.c` is running. The window contents tell us that the function `main()` has been called and that it has one local variable named `secs`, which currently has a value of 0.



The Depth value along the bottom of the Stack Trace window is the current number of bytes on the stack. The Max Depth value is the maximum number of bytes pushed on the stack at any one time for the current run of the program or since the Max Depth value was reset. The Max Depth value can be reset by a right click in the Stack Trace window to bring up some menu options. Along with resetting the Max Depth value back to zero (think of it like a car trip odometer) you can use the right click menu to copy text from the Stack Trace window or to cause the source code file to become the active window. The source code file could be a library file if a library function is executing at the time the menu option is requested.

Information

Select this option to activate the Information window, which displays how the memory is partitioned and how well the compilation went.

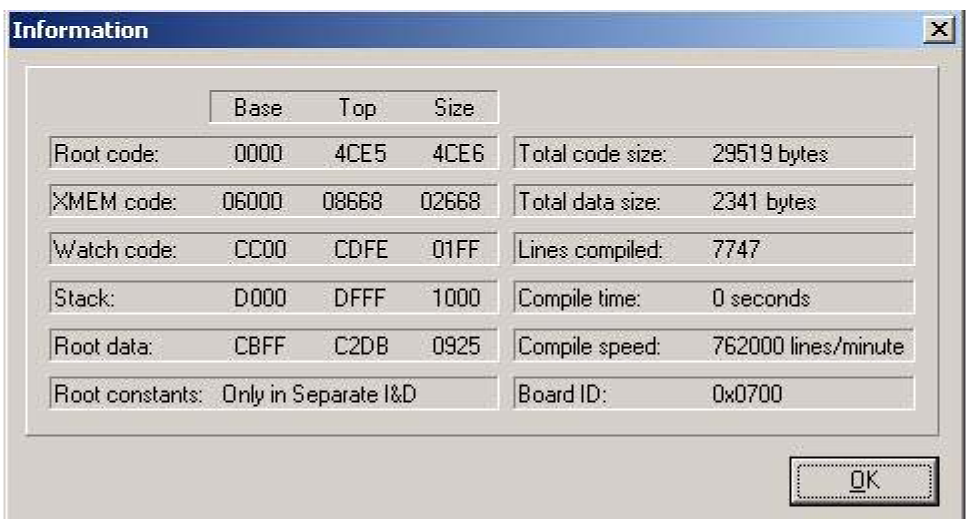


Table 14-1. Information Window

Name of Field	Description of Field
Root code	The begin (base), end (top) and size of the root code area, expressed in logical address format (16-bit).

Table 14-1. Information Window

Name of Field	Description of Field
XMEM code	The begin, end and size of the XMEM code area, expressed in physical address format (20-bit).
Watch code	The begin, end and size of the watch code area, expressed in logical address format (16-bit).
Stack	The begin, end and size of the run-time stack, expressed in logical address format (16-bit).
Root data	The begin, end and size of the root data area, expressed in logical address format (16-bit).
Root constants	The begin, end and size of the root constant area, expressed in physical address format (20-bit).
Total code size	The number of code bytes (including both root and XMEM code areas.
Total data size	The number of data bytes (including both root and XMEM data areas
Lines compiled	The number of lines compiled, including lines from library files.
Compile time	The number of seconds taken to compile the program.
Compile speed	Average speed of compilation measured in lines compiled per minute.
Board ID	A number identifying the board type. A list of board types is at `\Lib\default.h`.

Note that some of the memory areas described here may be non-contiguous (e.g., 2 Flash compiles and the XMEM code area with separate I&D). If an application is large enough to span into the non-contiguous part of an area, the values presented in the Information window for that area are not accurate.

14.2.9 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

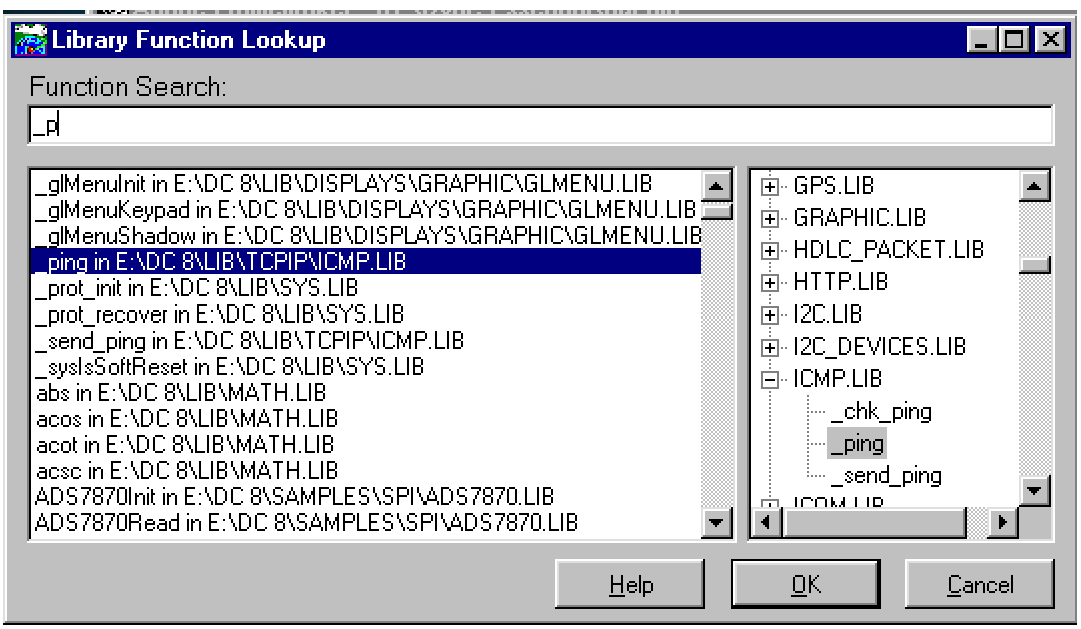
Opens a browser page and displays an HTML file of Dynamic C operators, with links to their descriptions in this manual.

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*.

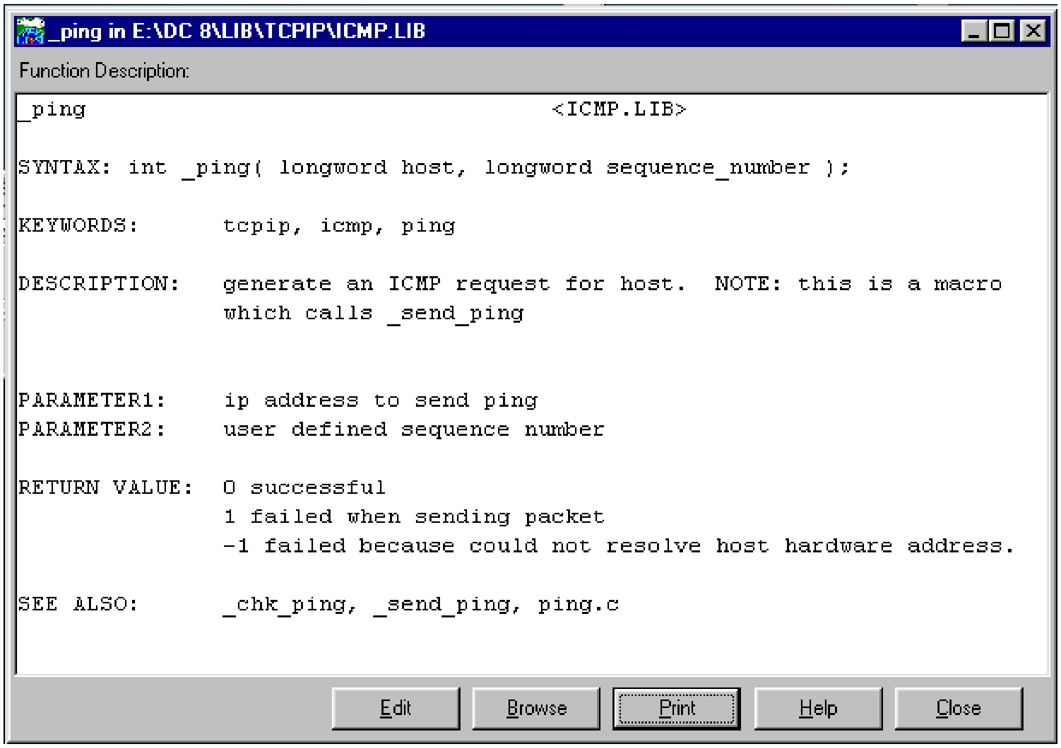
Function Lookup <Ctrl+H>

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 <Alt+F1>

Invokes an on-line help system and displays the alphabetical list of instructions for the Rabbit family of microprocessors.

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 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 registered serial number.

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.

Note that the command line compiler resides in the directory where you installed Dynamic C. In the console window, you need to "cd" into the directory where the command line compiler resides. From there you must type in the relative path of the sample you want to compile. Quotes are need if there are spaces in the path. For example,

```
> cd c:\DCRabbit_9.24
> dccl_cmp samples\memory_usage.c
> dccl_cmp "c:\My Documents\my program.c"
```

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 317.

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 builtin 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

The following switches require one or more parameters.

-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 MAX-
CHARS=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\MyTCparameters.rti”
 ters.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 284.

-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.

Example 1

In this example, all current settings of `default.dcp` are used for the compile.

```
dccl_cmp samples\timerb\timerb.c
```

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
```

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 myOut-
put.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.

15.6 Command Line RFU

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:

-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.bin`

-d

Description: Run Ethernet discovery to find RabbitLink or RabbitSys-enabled boards on a local area network (LAN). 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`

-fi Flash.ini PathName

Description: Select a new file that Dynamic C will use to externally define flash.

Default: `flash.ini`

RFU GUI Equivalent: From the "Choose File Locations..." dialog box, visible by selecting Setup | File Locations, type in a pathname or click on the ellipses radio button to browse for a file.

Example: `clRFU myProgram.bin -fi myflash.ini`

-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.bin`

-pw

Description: Passphrase for TCP/IP loader when using a RabbitLink.

Default: RabbitLink always prompts for a passphrase. Press "Enter" if no passphrase has been set.

RFU GUI Equivalent: None.

Example: `clRFU -pw mypassphrase`

-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.

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`

-vp+

Description: Verify the presence of the processor by using the DSR line of the PC serial connection.

Default: The processor is verified.

RFU GUI Equivalent: From the "Communications Options" dialog box, visible by selecting Setup | Communications, check the "Enable Processor Detection" option.

Example: `clRFU myProgram.bin -vp+`

-vp-

Description: Do not verify the presence of the processor.

Default: The processor is verified.

RFU GUI Equivalent: From the “Communications Options” dialog box, visible by selecting Setup | Communications, uncheck the “Enable Processor Detection” option.

Example: `clRFU myProgram.bin -vp-`

-usb+

Description: Enable use of USB to serial converter.

Default: The use of the USB to serial converter is disabled.

RFU GUI Equivalent: From the “Communications Options” dialog box, visible by selecting Setup | Communications, check the “Use USB to Serial Converter” option.

Example: `clRFU myProgram.bin -usb+`

-usb-

Description: Disable use of USB to serial converter.

Default: The use of the USB to serial converter is disabled.

RFU GUI Equivalent: From the “Communications Options” dialog box, visible by selecting Setup | Communications, uncheck the “Use USB to Serial Converter” option.

Example: `clRFU myProgram.bin -usb-`

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. Starting with Dynamic C 9.30, the File Open directory last used will be stored in the project file[i]. 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 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.

---

i. If DC is started with a cwd (current working directory) other than the exe directory, the cwd will be used instead of the one saved in the project file. This can happen if Dynamic C is started from a Windows shortcut with a specified “starts in” directory.

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. The command:

```
dccl_cmp
```

shows the current state of all `default.dcp` settings. The command:

```
dccl_cmp -pf myProject
```

shows the current state of all `myProject.dcp` settings. And the command:

```
dccl_cmp myProgram.c -ne 25 -pf myProject
```

reads `myProject.dcp`, then compiles and runs `myProgram.c`, showing a maximum of 25 errors.

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. Use an editor that 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 to not 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 A User-Defined BIOS

Before discussing a user-defined BIOS, we will review the history of the Rabbit BIOS. Dynamic C 9.30 introduced a reorganization of the BIOS. Prior to 9.30, RabbitBIOS.c contained all the BIOS code and a variety of configuration macros. Now, RabbitBIOS.c is a wrapper that permits a choice of which BIOS to compile. In addition, a more modular design has been implemented by moving many of the configuration macros to separate configuration libraries. The new BIOS file and configuration libraries are located in LIB\BIOSLIB. Table 17-1 lists the new files and gives a brief description of their content.

Table 17-1. BIOS File and Configuration Libraries

File Name	Description
STDBIOS.C	Most of the code from RabbitBIOS.c was moved here.
CLONECONFIG.LIB	Macros for configuring cloning.
DKCONFIG.LIB	Macros for configuring the debug kernel
ERRLOGCONFIG.LIB	Macros for configuring non-RabbitSys error logging. RabbitSys has its own error logging method.
MEMCONFIG.LIB	Macros for configuring memory organization.
SYSCONFIG.LIB	Macros for other system-level configuration options, such as the clock doubler and the specturm spreader.
TWOPROGRAMCONFIG.LIB	Macros for configuring split memory for the old-style DLM/DLP.
FATCONFIG.LIB	Macros for configuring the FAT file system.

To create a user-defined BIOS prior to Dynamic C 9.30, begin with a copy of RABBITBIOS.C. Starting with Dynamic C 9.30, begin with a copy of STDBIOS.C. Modify the BIOS file. It is prudent to save RABBITBIOS.C or STDBIOS.C as is and rename the modified file.

The Dynamic C GUI offers an option for hooking a user-defined BIOS into the system. See the description of the "Advanced... Button" in Section 14.2.7 for details on using this GUI option. Prior to Dynamic C 9.30, this GUI option was the easiest way to accomplish the goal. If you are using Dynamic C 9.30 or later and if you use the GUI option to hook in your BIOS, you will need to consider the configuration files and associated macros described in Table 17-1.

The suggested method to use with Dynamic C 9.30 or later involves editing the file RABBITBIOS.C to include the user-defined BIOS file. To do so, find the “#if __RABBITSYS == 0” statement and modify the code as follows:

```
#if MYBIOS == 1
   #use “mybios.c”
#elif __RABBITSYS == 0
   #use “STDBIOS.C”
#elif __RABBITSYS == 1
   #use “sysBIOS.C”
#else
   #use”rkBIOS.c”
#endif
```

To select the customized BIOS, define “MYBIOS = 1” in the Defines tab of the Options | Project Options dialog box.

17.2 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.

17.2.1 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.

17.2.2 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.

17.3 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. Factory-stored calibration constants live in the User block for boards with analog I/O. Space here is limited to as small as (`8K-sizeof(SysIDBlock)`) bytes, or less if there are calibration constants. For specific information about the User block on your board, open the sample programs `userblock_info.c` and/or `idblock_report.c`. The latter program will print, among other things, the location of the User block.
- **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 life of a battery on a Rabbit board is specified in the user's manual for that board; some boards have batteries that last several years, most board have batteries that come close to or surpass the shelf-life of the battery. If it is important that battery-backed data not be lost during a battery failure, know how long your battery will last and plan accordingly.

17.3.1 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 designer's handbook for more details about the User block.

17.3.2 Flash File System

For a complete discussion of the file system, please see Chapter 10, "File Systems."

17.3.3 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. For more information on flash compatibility issues, see technical notes TN216 "Is your Application Ready for Large Sector Flash?" and TN217 "Binary and Source Compatibility Issues for 4K Flash Sector Sizes" at Rabbit's website: `rabbit.com`..

17.3.4 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.

17.4 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 at: rabbit.com, or by choosing Online Documentation from within the Help menu of Dynamic C.

17.4.1 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**

  The macro `DATAORG` is the beginning logical address for the data segment.

  Root code space can be increased by increasing `DATAORG` in the BIOS (in `RabbitBios.c` prior to Dynamic C version 9.30 or in `StdBIOS.c` thereafter) in increments of 0x1000. The default is 0x3000 when separate I&D space is on, and 0x6000 otherwise. It can be as high as 0xB000.

  When separate I&D space is on, `DATAORG` defines the boundary between root variable data and root constant data. In this case, increasing `DATAORG` increases root constant space and decreases root variable space.

  When separate I&D space is off, `DATAORG` defines the boundary between root variable data and the combination of root code and root constant data. Note that root constants are in the base segment with root code. In this case, increasing `DATAORG` increases root code and root constant space and decreases root data space.

- **Compile out floating point support**

  Floating point support can be conditionally compiled out of `stdio.lib` by adding `#define STDIO_DISABLE_FLOATS` to either a user program or the Defines tab page in the Project Options dialog. This can save several thousand bytes of code 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 atcompile 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.

17.4.2 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**

  The macro `DATAORG` is the beginning logical address for the data segment.

  Root data space can be increased by decreasing `DATAORG` in the BIOS (in `RabbitBios.c` prior to Dynamic C version 9.30 or in `StdBIOS.c` thereafter) in increments of 0x1000. At the time of this writing, RAM compiles should be done with no less than the default value (0x6000) of `DATAORG` when separate I&D space is off. This restriction is to ensure that the pilot BIOS does not overwrite itself.

  When separate I&D space is on, `DATAORG` defines the boundary between root variable data and root constant data. In this case, decreasing `DATAORG` increases root variable space and descreases root constant space.

  When separate I&D space is off, `DATAORG` defines the boundary between root variable data and the combination of root code and root constant data. Note that root constants are in the base segment with root code. In this case, decreasing `DATAORG` increases root data space and decreases 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.

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 Macros Defined by the Compiler

The macros in the following table are defined internally. Default values are given where applicable, as well as directions for changing values.

Table A-1. 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, including its revision; e.g., `#if _CPU_ID_ >= R3000_R1` will identify a Rabbit 3000 rev. 1 or newer chip Look in `\Lib\..\BIOSLIB\sysiodefs.lib` for the constants and mask macros that are defined for use with `_CPU_ID_`.
`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.

Table A-1. Macros Defined by the Compiler

Macro Name	Definition and Default
`__FILE__`	The compiler substitutes this macro with the current source code file name as a character string literal.

`_FAST_RAM_`
`_FLASH_`
`_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_SIZE_` `_RAM_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.
`__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 Macros Defined in the BIOS or Configuration Libraries

This is not a comprehensive list of configuration macros, but rather, a short list of those found to be commonly used by Dynamic C programmers. Most default conditions can be overridden by defining the macro in the "Defines" tab of the "Project Options" dialog.

All the configuration macros listed here were defined in `RabbitBIOS.c` prior to Dynamic C 9.30. Since then many of them have been moved to configuration libraries while `RabbitBIOS.c` has become a wrapper file that permits a choice of which BIOS to compile. See Section 17.1 for more information on the reorganization of the BIOS that occured with Dynamic C 9.30.

CLOCK_DOUBLED

Determines whether or not to use the clock doubler. The default condition is to use the clock doubler, defined in `\BIOSLIB\sysconfig.lib`. Override the default condition by defining `CLOCK_DOUBLED` to "0" in an application or in the project.

DATAORG

Defines the beginning logical address for the data segment. Defaults are defined in the BIOS: 0x3000 if separate I&D space enabled, 0x6000 otherwise. Users can override the defaults in the Defines tab of Project Options dialog.

WATCHCODESIZE

Specifies the number of root RAM bytes for watch code. Defaults are defined in the BIOS: 0x200 bytes if watch expressions are enabled, zero bytes otherwise. The defaults cannot be overridden by an application.

USE_TIMERA_PRESCALE

Uncomment this macro in `\BIOSLIB\sysconfig.c` to run the peripheral clock at the same frequency as the CPU clock instead of the standard "CPU clock/2." This feature is not compatible with the Rabbit 2000.

USE_2NDFLASH_CODE

Uncomment this macro in `\BIOSLIB\sysconfig.c` only if you have a a board with two 256K flashes, and you want to use the second flash for extra code space. The file system (FS2) is not compatible with using the second flash for code.

A.3 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. The following program will use dc_timestamp to help calculate the date and time.

```
printf("The date and time: %lx\n", dc_timestamp);

main(){
   struct tm t;
   printf("dc_timestamp = %lx\n", dc_timestamp);
   mktm(&t, dc_timestamp);

   printf("%2d/%02d/%4d %02d:%02d:%02d\n",
      t.tm_mon,t.tm_mday,t.tm_year + 1900, t.tm_hour,t.tm_min,
            t.tm_sec);
}
```

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.4 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.5 Rabbit Registers

Macros are defined for all of the Rabbit registers that are accessible for application programming. A list of these register macros can be found in the user's manuals for the Rabbit microprocessor, as well as in the Rabbit Registers file accessible from the Dynamic C Help menu.

A.5.1 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. SECURITY SOFTWARE & UTILITY PROGRAMS

This appendix documents the security software and utility programs available for Rabbit-based systems. The security software is called the Rabbit Embedded Security Pack. It is summarized in Section C.1. There are several Dynamic C utilities, each one described in Section C.2.

C.1 Rabbit Embedded Security Pack

The Rabbit Embedded Security Pack is composed of AES and SSL functionality. It is available for purchase on the Rabibt website:

```
www.rabbit.com/products/dc/index.shtml
```

Documentation for the security pack is also available online:

```
www.rabbit.com/products/dc/DC9/docs.shtml
```

C.1.1 AES

Advanced Encryption Standard (AES) is an implementation of the Rijndael Advanced Encryption Standard cipher with 128 bit key. This is useful for encrypting sensitive data to be sent over unsecured network paths.

C.1.2 SSL

Secure Sockets Layer (SSL) is a security protocol that transforms a typical reliable transport protocol (such as TCP) into a secure communications channel for conducting sensitive transactions. The SSL protocol defines the methods by which a secure communications channel can be established—it does not indicate which cryptographic algorithms to use. SSL supports many different algorithms, and serves as a framework whereby cryptography can be used in a convenient and distributed manner.

C.2 Dynamic C Utilities

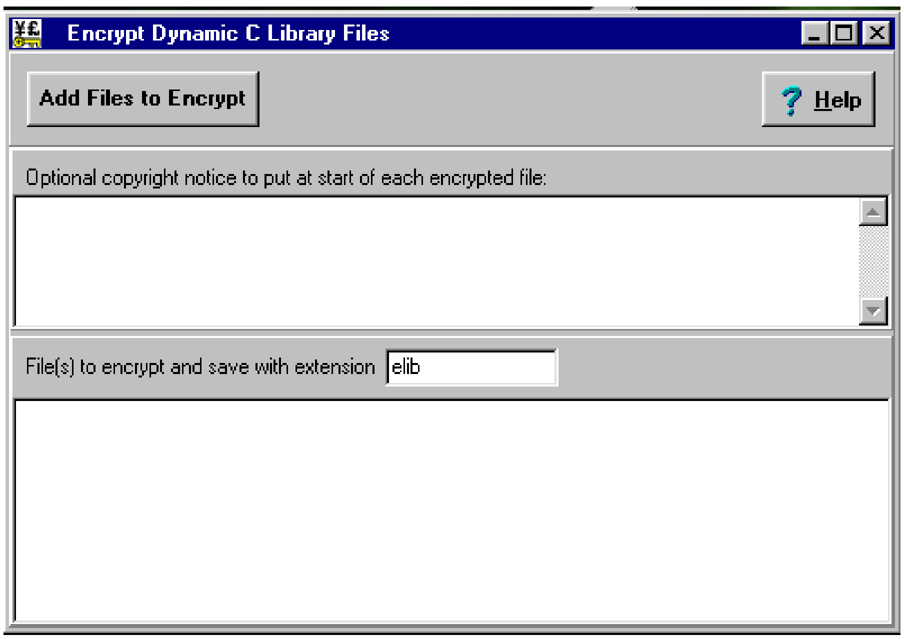
There are several utilities bundled with Dynamic C.

C.2.1 Library File Encryption

The Library File Encryption Utility, `Encrypt.exe`, allows distribution of sensitive runtime library files.

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.

To use this utility, double-click on the program name, `Encrypt.exe`. The following window will appear:



Complete instructions are available by clicking on the Help button in the upper righthand corner of the program window. Context-sensitive help is accessed by positioning the cursor over the desired subject and then pressing <F1>.

C.2.1.1 Add Files to Encrypt

There are two ways to select files to encrypt.

1. Type the path and filename in the lower window.
2. Click the Add Files to Encrypt button to bring up a file open dialog box and browse for the desired file.

The list of files to be encrypted may be edited if desired. Notice that if anything is entered in the lower window, a new button named “Encrypt” appears. Two entries in the window change it to “Encrypt All”. Clicking this button causes the utility to encrypt the file(s) listed in the lower window.

C.2.1.2 File Extension

Encrypted files will be saved with the same pathname but with the extension supplied. Dynamic C will use encrypted and non-encrypted files seamlessly, so the choice of extension is for one's own file management.

C.2.1.3 Optional Text Area

The upper window is a text window of up to 4k bytes in length. Any text entered will appear in all files in the list appearing in the lower window. If two files are to be given unique headers, they should be encrypted separately.

This area can be used for copyright information, instructions, disclaimers, warnings, or anything else relevant to viewers of the file.

C.2.2 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.rabbit.com.

C.2.2.1 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.2.2.2 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.2.2.3 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.2.3 Font and Bitmap Converter Utility

The Font and Bitmap Converter converts Windows fonts and monochrome bitmaps to a library file format compatible with Rabbit'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 Utilities folder where you installed Dynamic C. 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 `name_me.lib`: the name of your choice.

Add the file to applications with the statement:

```
#use name_me.lib        // remember to add this filename to "lib.dir" file
```

or by cut and pasting from `name_me.lib` directly into the application file.

C.2.4 Rabbit Field Utility

The Rabbit Field Utility (RFU) will load a binary file created with Dynamic C to a Rabbit-based board. The RFU can be used to load a binary file without Dynamic C present on the host computer, and without recompiling the program each time it is loaded to a controller.

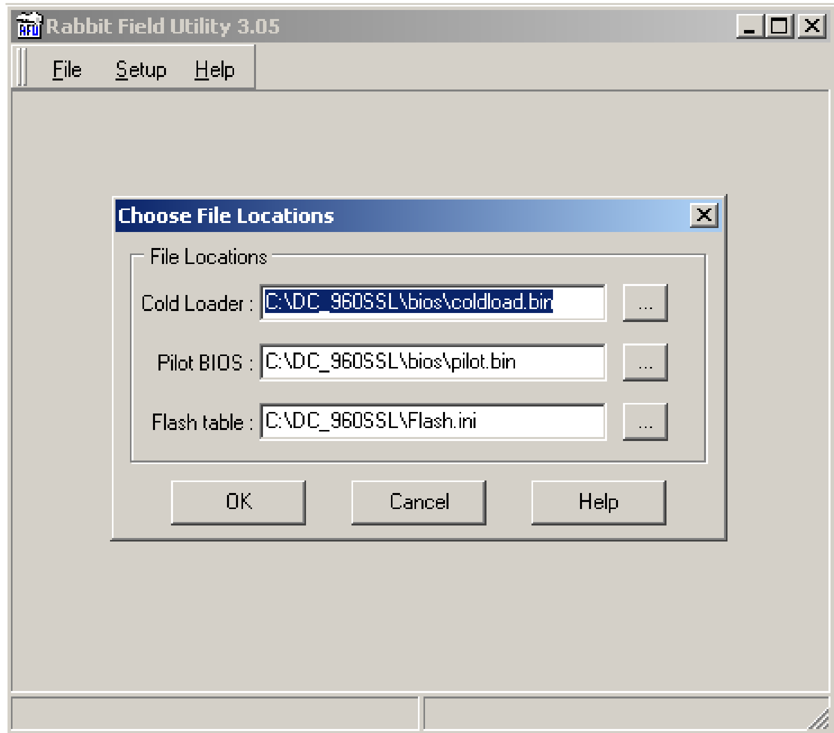
The Dynamic C installation created a desktop icon for the RFU. The executable file, `rfu.exe`, can be found in the subdirectory named "Utilities" where Dynamic C was installed. Complete instructions are available by clicking on the Help button within the utility. The Help document details setup information, the file menu options and BIOS requirements.

The RFU executable that comes with the Dynamic C distribution is branded as a product, as seen in the "About" screenshot shown here. You can brand the RFU or customize its functionality to suit your needs. Please contact technical support for the source file needed for customization:

http://www.rabbit.com/support/questionSubmit.shtml

The RFU enables those without Dynamic C to update their Rabbit-based board with a few files installed on the computer and the appropriate connection to the target board.

The necessary files are included with Dynamic C. They are: the executable (`Rfu.exe`), the cold loader, the pilot BIOS, and a file used to determine information about the memory device being used. The default files used for the cold loader, etc., can be seen by selecting "File Locations..." from the Setup menu.



`Rfu.exe` and its ancillary files are freely distributable.

The RFU communicates with the target using either a serial or a TCP/IP connection. The serial connection requires a programming cable. The TCP/IP connection requires either a RabbitLink board or a RabbitSys-enabled board.

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 “Choose File Locations...” dialog box, visible by selecting the menu option Setup | File Locations,, 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 “Choose File Locations...” dialog box, visible by selecting the menu option Setup | File Locations, type in a pathname or click on the ellipses radio button to browse for a file.

Example: `clRFU myProgram.bin -pb mySecondaryLoader.c`

-fi Flash.ini PathName

Description: Select a new file that Dynamic C will use to externally define flash.

Default: `flash.ini`

RFU GUI Equivalent: From the "Choose File Locations..." dialog box, visible by selecting the menu option Setup | File Locations, type in a pathname or click on the ellipses radio button to browse for a file.

Example: `clRFU myProgram.bin -fi myflash.ini`

-vp+

Description: Verify the presence of the processor by using the DSR line of the PC serial connection.

Default: The processor is verified.

RFU GUI Equivalent: From the "Communications Options" dialog box, visible by selecting Setup | Communications, check the "Enable Processor Detection" option.

Example: `clRFU myProgram.bin -vp+`

-vp-

Description: Do not verify the presence of the processor.

Default: The processor is verified.

RFU GUI Equivalent: From the "Communications Options" dialog box, visible by selecting Setup | Communications, uncheck the "Enable Processor Detection" option.

Example: `clRFU myProgram.bin -vp-`

-usb+

Description: Enable use of USB to serial converter.

Default: The use of the USB to serial converter is disabled.

RFU GUI Equivalent: From the "Communications Options" dialog box, visible by selecting Setup | Communications, check the "Use USB to Serial Converter" option.

Example: `clRFU myProgram.bin -usb+`

-usb-

Description: Disable use of USB to serial converter.

Default: The use of the USB to serial converter is disabled.

RFU GUI Equivalent: From the "Communications Options" dialog box, visible by selecting Setup | Communications, uncheck the "Use USB to Servile Converter" option.

Example: `clRFU myProgram.bin -usb-`

-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`

APPENDIX D. ADDITIONAL DOCUMENTATION

There is a suite of documentation available for the Dynamic C user. Numerous application notes, technical notes and white papers are available to help the reader learn more about different topics likely to be of interest to embedded systems engineers.

Dynamic C documentation is found in two places:

1. **Online at the Rabbit website**: www.rabbit.com. All manuals, application notes, technical notes and white papers are linked from http://www.rabbitsemiconductor.com/docs/. Documentation specific to Dynamic C is found here: http://www.rabbitsemiconductor.com/products/dc/docs.shtml#.
2. **On the software CD that comes with Rabbit-based hardware.** The documentation can be accessed by opening Dynamic C and clicking on the Help menu or by clicking on the desktop icon that was created during the Dynamic C installation.

Some technical notes of general interest are:

- TN202 "*Rabbit Memory Management in a Nutshell*"
  Found online here (pdf) and here (html).
- TN203 "*Porting a Program to Dynamic C*"
  Found online here (pdf) and here (html).
- TN213 "*Rabbit Serial Port Software*"
  Found online here (pdf) and here (html).
- TN261 "*The Slave Port Driver*"
  Found online here (pdf) and here (html).

RABBIT® SOFTWARE END USER LICENSE AGREEMENT

IMPORTANT-READ CAREFULLY: BY INSTALLING, COPYING OR OTHERWISE USING THE ENCLOSED RABBIT 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 RABBIT 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 Rabbit 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 and the Point-to-Point Protocol (PPP) library 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 Rabbit, or (ii) based on the Rabbit 2000 microprocessor, the Rabbit 3000 microprocessor, the Rabbit 4000 microprocessor, or any other Rabbit microprocessor. Qualified Systems may not be (a) designed or intended to be re-programmable by your customer using the Software, or (b) competitive with Rabbit products, except as otherwise stated in a written agreement between Rabbit and the system manufacturer. Such written agreement may require an end user to pay run time royalties to Rabbit.

2. **License**. Rabbit 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 Rabbit's 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 Rabbit; 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 Rabbit 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, Rabbit 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 Rabbit, 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 Rabbit, and will be governed solely by the terms agreed upon between You and the third party providing such services. RABBIT 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. RABBIT 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, RABBIT 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 RABBIT 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 RABBIT 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 RABBIT 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 RABBIT'S 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 Rabbit 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 Rabbit, 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 Rabbit'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 Rabbit. No amendment to or modification of this License will be binding unless in writing and signed by a duly authorized representative of Rabbit.

Copyright 2007 Rabbit®. All rights reserved.

Index

Symbols

_GLOBAL_INIT .. 204
{ } curly braces 24
@LENGTH 167
@PC 167
@RETVAL 167, 175
@SP 167, 171, 174, 175, 176, 184
. .. 39
\\ .. 148
\n . 138
\r 138
and ## (operators) 19
#asm 163, 214, 323
#debug 201, 215, 323
#define . 18, 19, 215
#elif 217
#else 217
#endasm 163, 167, 215
#endif 217
#error 216
#fatal 215
#funcchain 36, 216
#if 217
#ifdef 217
#ifndef 218
#include
  absence of 38
#interleave 218
#makechain 36, 218
#memmap 219, 325
#nodebug 201, 215, 323
#nointerleave 218
#nouseix 221
#undef 21
#use 38, 39, 220
#useix 221
#warns 221
#warnt 221
#ximport 221
#zimport 222

Numerics

2nd copy of FAT 158

A

abandon 185
abort 185
about Dynamic C 294
abstract data types 26
adc (add-with-carry) 163
add-on modules 337
address space 4, 117
Advanced button 275
AES encryption 337
aggregate data types 27
align 186
ALT key
  See keystrokes
always_on 186
anymem 186
application program 38
argument passing 31, 170, 171, 175, 177
  modifying value 31
arrange icons 286
arrays 27, 28, 31
  characters 22
  subscripts 27
arrow keys 237, 238
asm 187
assembly 3, 163–184, 247
  blocks in xmem 169
  embedding C statements 164
  stand-alone 169
  window 173, 287
assignment operators 227
associativity 223, 224
attributes of a file 141
auto 168, 169, 187
  storage of variables 170

B

back slash 148
backslash (\)
  character literals 19, 23
  continuation in directives 214
basic unit of a C program 25
baud rate 271
BCDE . 169, 175, 177
BeginHeader 40, 41
binary operators 223

BIOS 6
_xexit 111
calling premain() 105
command line compiler 296, 304
compilation environments 317
compile option 330
configuration macros 115
control blocks 122
macro definitions 281
memory location 118
memory settings 276
user-defined 276, 321
variable defined in 193
blocking 132
blocking a non-blocking function 143
blocking mode 135
board information 245, 283–284
BPB 157
branching 34, 35
break 188, 207
example 33
keyword 33
limitations 33
out of a loop 33
out of a switch statement 33
breakpoints
assembly window 173
enable 279
hard 248
interrupt status 247, 248
norst keyword 201
persistent 247
RST 28 322
single stepping 247
soft 247
Watches window 251
bringing up the FAT file system 134

C

C language 3, 4, 15, 22, 26, 36, 165, 169
calling assembly 175
embedded in assembly 164
cached write 139
call sequence 290
carriage return 138
cascaded windows 286
case 35, 188, 192
char 26, 189, 211
characters
arrays 22
embedded quotes 23
nonprinting values 23
special values 23
clipboard 241
closing a file 239
clusters
assignments 158
definition 156
CoData Structure 51
pointer to 53
cofunc 189
cofunctions 54–60
abandon 59
calling restrictions 55
everytime 59
firsttime 194
indexed 56
keyword 189
single user 57
suspend 209
syntax 54
cold loader 246
column resizing 288
command line interface 295–316
communication
TCP/IP 272
compatibility with µC/OS-II 160
compile
BIOS 246
command line 295–313
errors 243
menu 245
options 272
RAM 274, 327
speed 3
status 290
to .bin file 246
to file 237
to flash 245
to target 237, 245
compiler
line parsing limit 24
compiler directives 214
#asm 163, 214, 323
options 214
#class 214
options 214
#debug 201, 215, 323
#define 19, 215
#elif 217
#else 217
#endasm 163, 167, 215
#endif 217
#error 216
#fatal 215
#funcchain 36, 216
#GLOBAL_INIT 216
#if 217

#ifdef 217
#ifndef 218
#interleave 218
#makechain 36, 218
#memmap 219
options 219
#nodebug 201, 215, 323
#nointerleave 218
#nouseix 221
#pragma 219
#precompile 220
#undef 21, 220
#use 38, 39, 220
#useix 221
#warns 221
#warnt 221
#ximport 221
#zimport 222
line continuation 214
compound
names 18
statements 24
compression 339
concatenation of strings 22
configuration library 135
const 165, 190, 214
continue 33, 191, 207
example 33
copying text 241
costate 191
costatements 48–54, 143–145
abort 185
firsttime 194
keyword 191
suspend 209
syntax 49
yield 213
creating a file 138
curly braces { } 24
cursor
execution 247, 248
positioning 238, 243
custom device driver 135
cutting text 241

D

data area 158
data structure
composites 28
keyword 24
nesting 28
offset of element 168
pass by value 31
returned by function 175
union 28
data types 27
aggregate 27
primitive 17
DATAORG 325, 327
DATASEG 117
date and time 106
db 165
debug 322
dialog box 278
differences highlighting 251
disassemble at address 251
disassembled code 251
hints and tips 83–104
keyword 191
memory dump 251
mode 248
polling the target 247
step over 247
switching modes 243
trace into 247
trace macros 13
update watch expressions 251
watchdog timers 108
windows 262–268, 286–290
declarations 24, 40
default 35, 192
Default Compile Mode 275
delay loop 107
delimiter matching 238
demotion 273
device 132
differences highlighting 251
directory 132
create 151
default search 153
delete 154
entry structure 152
names 159
root 158
search conditions 152
disassemble
at address 251, 287
at cursor 251, 287
DLM and FAT 147
do loop 32
dot operator 18, 28
download manager 147
downloading 3
driver 132
DSR check 271
dump window 252
dw 166
Dynamic C

differences ... 4, 36
exit ... 240
support files ... 44
Dynamic C modules ... 337
dynamic storage allocation ... 29

E

Edit menu ... 241
edit mode ... 237, 243
editor ... 3
else ... 192
embedded assembly ... 3, 170, 175
embedded quotes ... 23
encryption ... 337
End key ... 237
EndHeader ... 40, 41
enum ... 193
equ ... 167
error codes ... 133, 135
errors
error code ranges ... 111
locating ... 243
run-time ... 111, 273
ESC key
to close menu ... 238
escape character ... 148
examples
break ... 33
continue ... 33
delay loop ... 107
for loop ... 32
modules ... 42
of array ... 27
timing loop ... 106
union ... 28
exit Dynamic C ... 240
extended memory ... 175, 212
asm blocks ... 169
extern ... 41, 42, 193

F

FAT and DLM ... 147
fat_AutoMount ... 136
fat_config.lib ... 135
fat_dirent ... 152
fat_Init ... 136
fat_part ... 136
fat_part_mounted ... 136, 151
FAT_USE_FORWARDSLASH ... 148
file
attributes ... 141
commands ... 239
compression ... 339
create ... 138, 151
delete ... 154
extensions ... 246
generated ... 246
names ... 159
open ... 138, 148
print ... 240
read ... 139, 149
seek ... 150
state ... 143
write ... 138, 149
file system
in primary flash ... 125
in RAM ... 122
max. # of files ... 122
max. file size ... 122
multitasking ... 123
files
additional source ... 38
Find Next <F3> ... 242
firsttime ... 194
flags register ... 288
flash
file system ... 122
initialized variables ... 5
USE_2NDFLASH_CODE ... 122
writing to ... 121
xmem access ... 117
flash types supported ... 133
float ... 26, 194, 211
values ... 21
for loop ... 32, 195
forward slash ... 148
frame
reference point ... 175
reference pointer ... 174, 175, 201, 322
function ... 25
auto variables ... 187
calls ... 25, 170, 171, 175
calls from assembly ... 177
chains ... 36, 204
create chains ... 218
entry and exit ... 322
execution time ... 322
headers ... 44
help ... 44
indirect call ... 30
prototypes ... 25, 27, 40
returns ... 175, 177
saving registers ... 184
stack space ... 322
transferring control ... 32
unbalanced stack ... 184
function lookup <CTRL-H> ... 292
function prefix

anymem ... 186
debug ... 191
firsttime ... 194
interrupt ... 197
nodebug ... 201
norst ... 201
nouseix ... 201
root ... 203
size ... 205
speed ... 205
useix ... 208
xmem ... 212

G

Global Initialization ... 37
global variables ... 29
goto ... 33, 34, 195, 243
grep ... 243

H

hard breakpoints ... 248
header
function ... 44
module ... 40, 42
Help menu ... 292
hexadecimal integer ... 21
HL ... 169, 174, 175, 177
Home key ... 237
horizontal tiling ... 286
hot-swapping
SD card ... 161
xD card ... 160

I

icons
arranged ... 286
IEEE floating point ... 194
if ... 192
multichoice ... 34
simple ... 34
with else ... 34
information window ... 286, 290
init_on ... 196
initialization ... 136
inline code ... 275
insertion point ... 241, 243
Inspect menu ... 249, 286
Instruction Set Reference ... 294
int ... 26, 197, 211
integers ... 21
interrupts ... 178
breakpoints ... 247
keyword for ISR ... 197
latency ... 178
unpreserved registers ... 184
vectors ... 179, 198
ISR ... 178, 325
IX (index register) ... 56, 174, 175, 201, 208

K

key ... 40
keystrokes
<ALT-Backspace>
undoing changes ... 241
<ALT-C>
select Compile menu ... 245
<ALT-F10>
Disassemble at Address ... 251
<ALT-F2>
Toggle Hard Breakpoint ... 248
<ALT-F4>
quitting Dynamic C ... 240
<ALT-F9>
Run w/ No Polling ... 247
<ALT-H>
select Help menu ... 292
<ALT-O>
select Options menu ... 254
<ALT-SHIFT-backspace>
redoing changes ... 241
<ALT-W>
select Window menu ... 286
<CTRL-F10>
Disassemble at Cursor ... 251
<CTRL-F2>
Reset Program ... 248
<CTRL-G>
Goto ... 243
<CTRL-H>
Library Help lookup ... 292
<CTRL-N>
next error ... 243
<CTRL-O>
Poll Target ... 248
<CTRL-P>
previous error ... 243
<CTRL-U>
Update Watch window ... 251
<CTRL-V>
pasting text ... 241
<CTRL-W>
Add/Del Items ... 249
<CTRL-X>
cutting text ... 241
<CTRL-Y>
Reset Target/Compile BIOS ... 246
<CTRL-Z>

Stop ... 247
<F10>
Assembly window 286
<F2>
Toggle Breakpoint 247
<F3>
Find Next 242
<F5>
Compile . 245
<F7>
Trace into 247
<F8>
Step over 247
<F9>
Run 247
keywords 175, 185, 201
abort 185
align 186
always_on 186
anymem 186
asm 187
auto 187
bbram 187
break 188
c 188
case 188
char 189
cofunc 189
const 165
continue 191
costate 191
debug 191
default 192
do 192
else 192
enum 193
extern 193
firsttime 194
float 194
for 195
goto 195
if 196
init_on 196
int 197
interrupt 197
interrupt_vector 198
long 200
nodebug 201
norst 201
nouseix 201
NULL 201
protected 202
register 202
return 203
root 203
scofunc 203
segchain 204
shared 204
short 205
size 205
sizeof 205
speed 205
static 206
struct 206
switch 207
typedef 207
union 208
unsigned 208
useix 208
void 209
volatile 209
waitfor 209
waitfordone 210
while 210
xdata 211
xmem 212
xstring 212
yield 213

L

language elements 15, 18, 22, 185
operators 223
lib.dir 38, 39, 43, 220
libraries 3, 38
linking 38
real-time programming 3
writing your own 39
Library Help lookup 44, 292
line feed 138
linking 3
list files 274
locating errors 243
long
integer 21
keyword 200
lookup function 292
loops 32, 33
breaking out of 33
delay with MS_TIMER 107
do 192
for 195
skipping to next pass 33
timing with MS_TIMER 106

M

macros 19, 167, 215
restrictions 21
with parameters 19

main function 25, 38, 200, 323
map file 335
max number of characters read 139
MBR 156
memory
  address space 117
  configuration macros 124
  DATAORG 325, 327
  dump 249
  dump at address 251
  dump flash 251
  dump to file 251
  extended 175, 212
  management 186, 203
  map 117, 335
  root 119, 168, 203, 325
  use_2ndflash_code 118
memory management unit 117
menus
  close all open 238
  Compile 245
  Edit 241
  Help 292
  Inspect 249, 286
  Options 254
  Run 247
message window 243, 286
metadata 128
Micro C/OS-II 160
MMU 117
mode
  changing 248
  debug (run) 243
  edit 243
  print preview 240
modules 39, 42, 337
  body 40, 41, 42
  example 42
  header 40, 42, 193
  key 40
mouse 237
MS_TIMER 106, 332
multitasking
  cooperative 45
  preemptive 62
multitasking compatibility 160

N

names 18, 159
  #define 18
  in assembly 168
Next error <CTRL-N> 243
nodebug 163, 201, 247, 251, 274, 322, 323
non-blocking 132, 142
norst 201
nouseix 201
NULL 201
num_fat_devices 136

O

octal integer 21
offsets in assembly 174, 175
online help 44, 294
opening a file 138
operators 223
  # and ## (macros) 19
  arithmetic operators 224
    decrement (--) 226
    division (/) 225
    increment (++) 226
    indirection (*) 225
    minus (-) 224
    modulus (%) 226
    multiplication (*) 225
    plus (+) 224
    pointers 225
    post-decrement (--) 226
    post-increment (++) 226
    pre-decrement (--) 226
    pre-increment (++) 226
  assignment operators 227
    add assign (+=) 227
    AND assign (&=) 228
    assign (=) 227
    divide assign (/=) 227
    modulo assign (%=) 227
    multiply assign (*=) 227
    OR assign (|=) 228
    shift left (<<=) 227
    shift right (>>=) 228
    subtract assign (-=) 227
    XOR assign (^=) 228
  associativity 223, 224
  binary 223
  bitwise operators
    address (&) 229
    bitwise AND (&) 229
    bitwise exclusive OR (^) 229
    bitwise inclusive OR (|) 229
    complement (~) 229
    pointers 229
    shift left (<<) 228
    shift right (>>) 228
  comma 236
  conditional operators (? :) 234
  equality operators 231
    equal (==) 231
    not equal (!=) 231

in assembly 165
logical operators 231
logical AND (&&) 231
logical NOT (!) 232
logical OR (| |) 231
operator precedence 236
postfix expressions 232
( ) parentheses 232
[ ] array indices 232
dot (.) 232
parentheses ( ) 232
right arrow (->) 233
precedence 223
reference/dereference operators 233
address (&) 233
bitwise AND (&) 233
indirection (*) 233
multiplication (*) 233
relational operators 230
greater than (>) 230
greater than or equal (>=) 230
less than (<) 230
less than or equal (<=) 230
sizeof 235
unary 223
optimize size or speed 274
options
compiler 272
menu 254
origins 214

P

PageDown key 237
PageUp key 237
partition 132
partition structure 136
partitioning 127, 146–147
passing arguments 31, 170, 171, 175, 177
pasting text 241
path separator 148
periodic interrupt 54, 62, 105, 332
pointer checking 30
pointers 22, 29, 30, 31
uninitialized 30
poll target 248
polling 247
positioning text 243
prealloc 135
precompile 40, 220
preserving registers 177, 184
Previous error <CTRL-P> 243
primary register 169, 175, 177
primitive data types 17
print
choosing a printer 240
print file 240
print preview 240
printf 23, 27, 263
program
example 26
flow 32
reset 248
spanning 2 flash 122, 323
project files 239, 317–319
promotion 224
protected
keyword 202
variables 3, 202
prototypes
checking 273
function 25, 27, 40
in module header 40
punctuation 16

Q

quitting Dynamic C 240

R

Rabbit restart
protected variables 202
RabbitSys 275
RAM compile 274, 327
RAM functions 184
reading a file 139
reading max number of characters 139
real-time
programming 3
redoing changes 241
registers
saving and restoring 178
shadow 333
snapshots 288
window 286, 288
relocatable code 184
removable device advice 139
reserving file space 135
reset
program 248
resizing columns 288
result code (rc) 135
ret 175, 178
reti 178
return 175, 203, 207
return address 170
root memory
file system usage 123
keyword 203
memory map 117

static variables ... 119
variable address ... 168
RST 28H ... 247, 322
run
menu ... 247
mode ... 243, 247
no polling ... 247
run-time errors ... 111

S

sample programs
basic C constructs ... 26
saving a file ... 239
scofunc ... 203
search text ... 242
SEC_TIMER ... 106, 332
sector ... 156
secure communications ... 337
segchain ... 36, 204
SEGSIZE ... 117
separate I&D space ... 165, 179, 251, 275
SF1000 ... 160
shadow registers ... 333
shared ... 204
shared variables ... 3, 202
shell program ... 140
short ... 205
single stepping
assembly window ... 173
options ... 247
watches window ... 251
size ... 205, 274
sizeof ... 205
skipping to next loop pass ... 33
slice statements ... 62
soft breakpoints ... 247
source files ... 38
SP (stack pointer) ... 171, 176, 177, 184, 221
special characters ... 23
special symbols
in assembly ... 167
speed ... 205, 274
SSL ... 337
stack
enable tracing ... 279
enter function ... 322
frame ... 170, 171, 175, 177, 184
frame reference point ... 175
frame reference pointer ... 174, 175, 201, 322
function arguments ... 31
function returning struct ... 175
ISR ... 178
local variables ... 174, 187
nouseix ... 201
pointer (SP) ... 171, 176, 177, 184, 221
snapshots ... 289
trace window ... 268, 290
unbalanced ... 184
window ... 289
STACKSEG ... 117
state machine
example ... 46
state of file ... 143
statements ... 24
static variables
initialization ... 5
keyword ... 206
root memory ... 119
status register ... 288
Stdio window ... 262, 286
STDIO_DEBUG_SERIAL ... 263
step over ... 247
stop program execution ... 247
storage class ... 24
auto ... 29
static ... 29
strings ... 22, 211
concatenation ... 22
functions ... 22
literal ... 19
terminating null byte ... 22
struct keyword ... 206
structure
composites ... 28
keyword ... 24
nesting ... 28
offset of element ... 168
pass by value ... 31
return space ... 171, 175, 177
returned by function ... 175
union ... 28
subdirectory ... 132
subscripts
array ... 27
support files ... 44
supported flash types ... 133
switch ... 35, 192, 207
breaking out of ... 33
case ... 207
switching to edit mode ... 243
symbol information ... 335
symbolic constant ... 215

T

target information ... 245, 283–284
TCP/IP ... 272
text editing ... 241
text search ... 242

TICK_TIMER 106, 332
tiling windows 286
timing loop 106
toggle
  breakpoint 247, 248
toolbar 285
trace into 247
trace macros 13
tracing
  example 12
type
  casting 224
  checking 25, 273
  definitions 26
typedef 26, 207

U

ucos2 160
udppages.c 140
unary operators 223
unbalanced stack 184
undoing changes 241
uninitialized
  pointers 30
union 24, 28, 208
unpreserved registers 177, 184
unsigned 208
unsigned integer 21
unsupported FAT features 161
untitled files 239
USB 271
USE_2NDFLASH_CODE 118, 122, 323
useix 174, 208, 322
User block 323, 324
using the FAT file system 137
Utility Programs
  File Compression/Decompression 339
  Font/ Bitmap Converter 341
  Rabbit Field Utility 341

V

variables
  auto 187
  global 29
  static 206
vertical tiling 286
virtual watchdogs 108
void 209
volatile 209

W

waitfor 209
waitfordone 210
warning reports 273
watch expressions
  add or delete 249
  enable 279
  single stepping in assembly 89
  watch menu option 286
  watch window 249, 286
watchdog timers 108
watchdogs, virtual 108
wfd 210
while 24, 32, 210
wildcard mask 39
windows
  assembly 173, 287
  cascaded 286
  information 286, 290
  message 286
  register 286, 288
  stack 286, 289
  Stdio 262, 286
  tiled horizontally 286
  tiled vertically 286
  watch 251, 286
write-back cache 139
writing a file 138

X

xdata 211
xmem 175, 212
  asm blocks 169
  definition 117
  root functions in 200
XPC 117, 325
xstring 212

Y

yield 213

Z

µC/OS-II compatibility 160