For Rabbit Semiconductor Microprocessors

Integrated C Development System

User's Manual

010430 - M

SE and Premier Editions

Dynamic C User's Manual

Part Number 019-0071 • 010430-M

Copyright

© 1999 Z-World, Inc. • All rights reserved.

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

Trademarks

- Dynamic C® is a registered trademark of Z-World, Inc.
- Windows® is a registered trademark of Microsoft Corporation

Notice to Users

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

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

All Z-World products are 100 percent functionally tested. Additional testing may include visual quality control inspections or mechanical defects analyzer inspections. Specifications are based on characterization of tested sample units rather than testing over temperature and voltage of each unit. Rabbit Semiconductor may qualify components to operate within a range of parameters that is different from the recommended range of the manufacturer. This strategy is believed to be more economical and effective. Additional testing or burn-in of an individual unit is available by special arrangement.

Company Address

Z-World, Inc.

2900 Spafford Street
Davis, California 95616-6800
USA
Telephone: (530) 757-3737
Facsimile: (530) 753-5141
Web site: http://www.zworld.com

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
  - Speed 3
- 2.2 Dynamic C Enhancements and Differences 4
  - Dynamic C Enhancements 4
  - Dynamic C Differences 5
- 2.3 Dynamic C Differences Between Rabbit and Z180 5

3 Quick Tutorial 7
- 3.1 Run DEMO1.C 7
  - Single-Stepping 8
  - Watch Expression 9
  - Breakpoint 9
  - Editing the Program 9
- 3.2 Run DEMO2.C 10
  - Watching Variables Dynamically 10
- 3.3 Run DEMO3.C 10
  - Cooperative Multitasking 10
- 3.4 Summary of Features 12
  - Development Functions 12
  - Single-stepping 12
  - Setting breakpoints 12
  - Watch expressions 12
  - Costatements 12

4 Language 13
- 4.1 C Language Elements 13
- 4.2 Punctuation and Tokens 14
- 4.3 Data 14
- 4.4 Names 15
- 4.5 Macros 16
  - Restrictions 18
- 4.6 Numbers 19
- 4.7 Strings and Character Data 19
- 4.8 Statements 21
- 4.9 Declarations 21
- 4.10 Functions 22
- 4.11 Prototypes 22
- 4.12 Type Definitions 23
- 4.13 Aggregate Data Types 24
  - Array 24
  - Structure 24
  - Union 25
  - Composites 25
- 4.14 Storage Classes 25
- 4.15 Pointers 26
- 4.16 Pointers to Functions, Indirect Calls 27
- 4.17 Argument Passing 28
- 4.18 Program Flow 28
  - Loops 29
  - Continue and Break 29
  - Branching 31
- 4.19 Function Chaining 32
- 4.20 Global Initialization 33
- 4.21 Libraries 35
- 4.22 Support Files 36
- 4.23 Headers 36
- 4.24 Modules 37
  - The Key 37
  - The Header 37
  - The Body 38
  - Function Description Headers 39

5 Multitasking with Dynamic C 41
- 5.1 Cooperative Multitasking 41
- 5.2 A Real-time Problem 43
  - Solving the Real-time Problem With a State Machine 43
- 5.3 Costatements 44
  - Solving the Real-time Problem With Costatements 44
  - Costatement Syntax 45
  - Control Statements 45
- 5.4 Advanced Costatement Topics 46
  - The CoData Structure 46
  - CoData Fields 47
  - Pointer to CoData Structure 48
  - Library Extensions for Use With Named Costatements 48
  - Firsttime Functions 49
  - Shared Global Variables 49
- 5.5 Cofunctions 50
  - Syntax 50
  - Calling Restrictions 51
  - CoData Structure 51
  - Firsttime functions 51
  - Types of Cofunctions 52
  - Types of Cofunction Calls 53
  - Special Code Blocks 54
  - Solving the Real-time Problem With Cofunctions 55
- 5.6 Patterns of Cooperative Multitasking 55
- 5.7 Timing Considerations 56

waitfor Accuracy Limits................. 57
5.8 Overview of Preemptive Multitasking57
5.9 Slice Statements 57
Syntax 57
Usage 58
Restrictions 58
Slice Data Structure 59
Slice Internals 59
5.10 Summary 61
6 The Virtual Driver.................................. 63
6.1 Default Operation............................... 63
6.2 Calling _GLOBAL_INIT() 63
6.3 Global Timer Variables 64
6.4 Watchdog Timers 65
Hardware Watchdog 65
Virtual Watchdogs 65
6.5 Preemptive Multitasking Drivers 65
7 The Slave Port Driver 67
7.1 Slave Port Driver Protocol 67
Overview 67
Registers on the Slave 67
Polling and Interrupts 68
Communication Channels 69
7.2 Functions 69
SPinit 69
SPsetHandler 70
MyHandler...................................... 71
SPtick.. 72
SPclose 72
7.3 Examples 72
Example of a Simple Status Handler.. 72
Example of a Serial Port Handler... 73
cof_MSgetc..................................... 74
cof_MSputc 74
cof_MSread 75
cof_MSwrite 75
MSclose 76
MSgetc.. 76
MSgetError 77
MSinit 77
MSopen... 78
MSputc 78
MSrdFree 79
MSsendCommand 79
MSread 80
MSwrFree 80
MSwrite 81
Example of a Byte Stream Handler 83
cbuf_init.. 83
cof_SPSread 84
cof_SPSwrite 84
SPSinit 85
SPSread 85
SPSwrite 86
SPSwrFree 86
SPSrdFree 87
SPSwrUsed 87
SPSrdUsed 87
8 Efficiency.. 89
8.1 Nodebug Keyword 89
8.2 Static Variables 89
8.3 Function Entry and Exit 90
9 Run-Time Error Processing................. 91
9.1 User-defined error handlers................ 93
10 Memory Management 95
10.1 Memory Map..................................... 95
Memory Mapping Control............. 96
10.2 Extended Memory Functions 96
Code Placement in Memory 96
11 The Flash File System 99
11.1 General Usage 99
Wear Leveling 99
Low-level implementation 99
11.2 Application Requirements................ 100
11.3 Functions.. 100
Using File Names 101
11.4 Skeleton Program............................. 102
12 Using Assembly Language 103
12.1 Program Flow................................... 103
Embedded C in Assembly 104
12.2 Comments 104
12.3 Labels 104
12.4 Defining Constants........................... 105
12.5 Expressions 105
12.6 Multiline Macros.............................. 106
12.7 Special Symbols 106
12.8 C Variables 106
12.9 Stand-alone Assembly Code 107
12.10 Embedded Assembly Code 108
Not Using the IX Register, Function in Root Memory 109
Using the IX Register, Function in Root Memory 110
Not Using the IX Register, Function in Extended Memory..................... 112
12.11 C Functions Calling Assembly Code 113
12.12 Assembly Code Calling C Functions 114

12.13 Interrupt Routines in Assembly........115
12.14 Common Problems..........116

13 Keywords..........117
abort..........117
always_on..........117
anymem..........117
auto..........117
break..........118
case..........118
char..........118
const..........119
continue..........120
costate..........120
debug..........120
default..........121
do..........121
else..........121
extern..........121
firsttime..........122
float..........122
for..........123
goto..........123
if..........123
init_on..........124
int..........124
interrupt..........124
long..........124
main..........125
nodebug..........125
norst..........125
nouseix..........125
NULL..........125
protected..........126
return..........126
root..........127
segchain..........127
shared..........127
short..........128
size..........128
sizeof..........128
speed..........128
static..........129
struct..........129
switch..........130
typedef..........130
union..........131
unsigned..........131
useix..........131
waitfor..........131
waitfordone (wfd)..........132
while..........132
xdata..........132
xmem..........132
xstring..........133
yield..........133
13.1 Compiler Directives..........134
#asm options
#endasm..........134
#class options..........134
#debug
#nodebug..........134
#define name text
#define name(params...) text..........134
#fatal "…"..........134
#GLOBAL_INIT { variables }..........134
#error "…"..........135
#funcchain chainname name..........135
#if constant_expression
#elif constant_expression
#else
#endif..........135
#ifdef name
#ifndef name..........135
#interleave
#nointerleave..........135
#KILL name..........136
#makechain chainname..........136
#memmap options..........136
#undef name..........136
#use pathname..........136
#useix
#nouseix..........136
#warns "…"..........136
#warnt "…"..........136
#ximport *<filename>* *<symbol>*..........137

14 Operators..........139
14.1 Arithmetic Operators..........140
+..........140
–..........140
*..........141
/..........141
++..........141
—..........142
%..........142
14.2 Assignment Operators..........142
=..........142
+=..........142
-=..........143
*=..........143
/=..........143
%=..........143
<<=..........143
>>=..........143
&=..........143
^=..........144
|=..........144

14.3 Bitwise Operators 144
<< 144
>> 144
& 144
^ 145
| 145
~ 145
14.4 Relational Operators 145
< 145
<= 145
> 146
>= 146
14.5 Equality Operators 146
== 146
!= 146
14.6 Logical Operators 147
&& 147
|| 147
! 147
14.7 Postfix Expressions 147
( ) 147
[ ] 147
. (dot) 148
-> 148
14.8 Reference/Dereference Operators 148
& 148
* 149
14.9 Conditional Operators 149
? : 149
14.10 Other Operators 150
(type) 150
sizeof 150
, 151

15 Function Reference 153
15.1 Functional Groups 153
arithmetic 153
bit manipulation 153
character 153
extended memory 153
fast fourier transforms 153
file system 153
floating-point math 154
low-level flash access 154
I/O 154
interrupts 154
MicroC/OS-II 155
miscellaneous 155
multitasking 156
number-to-string conversion 156
real-time clock 156
serial communication 156
STDIO 157
string manipulation 157
string-to-number conversion 157
system 157
watchdog 158
15.2 Alphabetical Listing 159
abs 159
acos 159
acot 160
acsc 160
asec 161
asin 161
atan 162
atan2 163
atof 164
atoi 164
atol 165
bit 165
BIT 166
BitRdPortE 166
BitRdPortI 167
BitWrPortE 168
BitWrPortI 169
ceil 170
170
chkHardReset 170
chkSoftReset 171
chkWDTO 171
clockDoublerOn 172
clockDoublerOff 172
CoBegin 173
cof_serXgetc 173
cof_serXgets 174
cof_serXputc 175
cof_serXputs 176
cof_serXread 177
cof_serXwrite 178
CoPause 179
CoResume 180
cos 180
cosh 181
defineErrorHandler 181
deg 182
DelayMs 182
DelaySec 183
DelayTicks 183
Disable_HW_WDT 184
exit 184
exp 185
fabs 185
fclose 186
fcreate 186
fcreate_unused 187
fdelete 187
fftcplx 188

fftcplxinv 189
fftreal 190
fftrealinv 191
flash_erasechip 192
flash_erasesector 192
flash_gettype 193
flash_init 194
flash_read 195
flash_readsector 196
flash_sector2xwindow 197
flash_writesector 198
floor 199
fmod 199
fopen_rd 200
fopen_wr 200
forceSoftReset 201
fread 201
frexp 202
fs_format 203
fs_init 204
fs_reserve_blocks 205
fsck 205
fseek 206
ftell 207
fshift 207
fwrite 208
ftoa 208
getchar 209
getcrc 209
gets 210
GetVectExtern2000 210
GetVectIntern 211
hanncplx 212
hannreal 213
hitwd 214
htoa 214
IntervalMs 215
IntervalSec 215
IntervalTick 216
ipres 216
ipset 217
isalnum 217
isalpha 218
iscntrl 218
isCoDone 219
isCoRunning 219
isdigit 220
isgraph 220
islower 221
isspace 221
isprint 222
ispunct 223
isupper 224
isxdigit 224
itoa 225
kbhit 225
labs 226
ldexp 226
log 227
log10 227
longjmp 228
ltoa 228
ltoan 229
memchr 229
memcmp 230
memcpy 231
memmove 231
memset 232
mktime 232
mktm 233
modf 234
OSInit 234
OSMboxAccept 235
OSMboxCreate 235
OSMboxPend 236
OSMboxPost 237
OSMboxQuery 238
OSMemCreate 239
OSMemGet 240
OSMemPut 240
OSMemQuery 241
OSQAccept 241
OSQCreate 242
OSQFlush 243
OSQPend 244
OSQPost 245
OSQPostFront 246
OSQQuery 247
OSSchedLock 247
OSSchedUnlock 248
OSSemAccept 248
OSSemCreate 249
OSSemPend 249
OSSemPost 250
OSSemQuery 251
OSSetTickPerSec 252
OSStart 252
OSStatInit 253
OSTaskChangePrio 253
OSTaskCreate 254
OSTaskCreateExt 255
OSTaskCreateHook 256
OSTaskDel 257
OSTaskDelHook 258
OSTaskDelReq 259
OSTaskQuery 260
OSTaskResume 261
OSTaskStatHook 261

OSTaskStkChk ... 262
OSTaskSuspend ... 263
OSTaskSwHook ... 263
OSTimeDly ... 264
OSTimeDlyHMSM ... 265
OSTimeDlyResume ... 266
OSTimeDlySec ... 267
OSTimeGet ... 267
OSTimeSet ... 268
OSTimeTickHook ... 268
OSVersion ... 269
outchrs ... 269
outstr ... 270
paddr ... 270
poly ... 271
pow ... 272
pow10 ... 272
powerspectrum ... 273
premain ... 274
printf ... 274
putchar ... 275
puts ... 275
qsort ... 276
rad ... 277
rand ... 278
randb ... 278
randg ... 279
RdPortE ... 279
RdPortI ... 280
read_rtc ... 280
read_rtc_32kHz ... 281
res ... 281
RES ... 282
root2xmem ... 283
runwatch ... 283
serCheckParity ... 284
serXclose ... 284
serXdatabits ... 285
serXflowcontrolOff ... 285
serXflowcontrolOn ... 286
serXgetc ... 287
serXgetError ... 288
serXopen ... 289
serXparity ... 290
serXpeek ... 291
serXputc ... 291
serXputs ... 292
serXrdFlush ... 292
serXrdFree ... 293
serXrdUsed ... 293
serXread ... 294
serXwrFlush ... 295
serXwrFree ... 295
serXwrite ... 295
set ... 296
SET ... 297
setjmp ... 298
SetVectExtern2000 ... 299
SetVectIntern ... 300
sin ... 300
sinh ... 301
sprintf ... 302
sqrt ... 303
strcat ... 303
strchr ... 304
strcmp ... 305
strcmpi ... 306
strcpy ... 307
strcspn ... 307
strlen ... 308
strncat ... 308
strncmp ... 309
strncmpi ... 310
strncpy ... 311
strpbrk ... 312
strrchr ... 312
strspn ... 313
strstr ... 313
strtod ... 314
strtok ... 315
strtol ... 316
_sysIsSoftReset ... 316
sysResetChain ... 317
tan ... 317
tanh ... 318
tm_rd ... 319
tm_wr ... 320
tolower ... 321
toupper ... 321
updateTimers ... 322
use32HzOsc ... 322
useClockDivider ... 323
useMainOsc ... 323
utoa ... 324
VdGetFreeWd ... 325
VdHitWd ... 325
VdInit ... 326
VdReleaseWd ... 327
WriteFlash2 ... 328
write_rtc ... 329
WrPortE ... 330
WrPortI ... 330
xalloc ... 331
xmem2root ... 331
xmem2xmem ... 332

16 User Interface ... 333
16.1 Editing ... 333

16.2 Menus 334
New 335
Open 335
Save 335
Save As 335
Close 335
Print Preview 335
Print 335
Print Setup 336
Exit 336
16.3 Edit Menu 336
Undo 336
Redo 337
Cut 337
Copy 337
Paste 337
Find 337
Replace 337
Find Next 338
Goto 338
Previous Error 338
Next Error 338
Edit Mode 338
16.4 Compile Menu 339
Compile to Target 339
Compile to .bin file 339
Reset Target/Compile BIOS 340
Include Debug Code/RST 28 Instructions 340
16.5 Run Menu 341
Run 341
Run w/ No Polling 341
Stop 341
Reset Program 342
Trace Into 342
Step over 342
Toggle Breakpoint 342
Toggle Hard Breakpoint 342
Toggle Interrupt Flag 342
Toggle Polling 342
Reset Target 343
Close Serial Port 343
16.6 Inspect Menu 343
Add/Del Watch Expression 343
Clear Watch Window 344
Update Watch Window 344
Disassemble at Cursor 344
Disassemble at Address 344
Dump at Address 345
16.7 Options Menu 346
Editor 346
Compiler 347
Debugger 349
Display 350
Communications 351
Show Tool Bar 351
Save Environment 352
16.8 Window Menu 352
Cascade 352
Tile Horizontally 352
Tile Vertically 353
Arrange Icons 353
Message 353
Watch 353
STDIO 353
Assembly 354
Registers 355
Stack 355
Information 356
16.9 Help Menu 356
Online Documentation 356
Keywords 357
Operators 357
HTML Function Reference 357
Function Lookup/Insert 357
Keystrokes 359
Search for Help on 359
Contents 359
About 359
17 µC/OS-II 361
17.1 Changes 361
Ticks per Second 361
Task Creation 362
Restrictions 363
17.2 Tasking Aware Interrupt Service Routines (TA-ISR) 363
Interrupt Priority Levels 363
Possible ISR Scenarios 364
General Layout of a TA-ISR 365
17.3 Library Reentrancy 369
17.4 How to Get a µC/OS-II Application Running 370
17.5 Compatibility with TCP/IP 375
Software License Agreement 377
Index 381

Installing Dynamic C 1

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

```
<disk>:\SETUP
```

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

1.1 Requirements

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

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

1.2 Assumptions

Assumptions are made regarding your knowledge and experience in the following areas:

- Understanding of the basics of operating a software program and editing files under Windows on a PC.
- Knowledge of basic assembly language and architecture for controllers.

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

The C Programming Language by Kernighan and Ritchie (published by Prentice-Hall).

C: A Reference Manual by Harbison and Steel (published by Prentice-Hall).

Introduction to Dynamic C 2

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

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 text editor. 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, breakpoints and other advanced debugging features. Watch expressions can be used to compute C expressions involving the target's program variables or functions. Watch expressions can be evaluated while stopped at a breakpoint or while the target is running its program.

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

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

2.1.1 Speed

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

2.2 Dynamic C Enhancements and Differences

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

2.2.1 Dynamic C Enhancements

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

- Function chaining, a concept unique to Dynamic C, allows special segments of code to be embedded within one or more functions. When a named function chain executes, all the segments belonging to that chain execute. Function chains allow software to perform initialization, data recovery, or other kinds of tasks on request.
- Costatements allow concurrent parallel processes to be simulated in a single program.
- Cofunctions allow cooperative processes to be simulated in a single program.
- Slice statements allow preemptive processes in a single program.
- The interrupt keyword in Dynamic C allows the programmer to write interrupt service routines in C.
- Dynamic C supports embedded assembly code and stand-alone assembly code.
- Dynamic C has shared and protected keywords that help protect data shared between different contexts or stored in battery-backed memory.
- Dynamic C has a set of features that allow the programmer to make fullest use of extended memory. Dynamic C supports the 1M address space of the microprocessor. The address space is segmented by a memory management unit. Normally, Dynamic C takes care of memory management, but there are instances where the programmer will want to take control of it. Dynamic C has keywords and directives to help put code and data in the proper place. The keyword **`root`** selects root memory (addresses within the 64K physical address space). The keyword **`xmem`** selects extended memory, which means anywhere in the 1024K or 1M code space. **`root`** and **`xmem`** are semantically meaningful in function prototypes and more efficient code is generated when they are used. Their use must match between the prototype and the function definition. The directive **`#memmap`** allows further control. See “Memory Management” on page 95, for further details on memory.

2.2.2 Dynamic C Differences

The main differences in Dynamic C are summarized here and discussed in detail in chapters "Language" on page 13 and "Keywords" on page 117.

- If a variable is initialized in a declaration (e.g., `int x = 0;`), it is stored in Flash Memory (EEPROM) and cannot be changed by an assignment statement. Starting with Dynamic C 7.x such declaration will generate a warning which can be suppressed using the `const` keyword: `const int x = 0`; To initialize static variables in Static RAM (SRAM) use `#GLOBAL_INIT` sections.
- The default storage class is `static`, not `auto`. This avoids numerous bugs encountered in embedded systems due to the use of auto variables. Starting with Dynamic C 7.x, the default class can changed to auto by the compiler directive `#class auto`.
- The numerous include files found in typical C programs are not used because Dynamic C has a library system that automatically provides function prototypes and similar header information to the compiler before the user's program is compiled. This is done via the `#use` directive. This is an important topic for users who are writing their own libraries. Those users should refer to the Modules section of the language chapter.
- When declaring pointers to functions, arguments should not be used in the declaration. Arguments may be used when calling functions indirectly via pointer, but the compiler will not check the argument list in the call for correctness.
- Bit fields and enumerated types are not supported. Separate compilation of different parts of the program is not supported or needed. There are minor differences involving `extern` and `register` keywords.

2.3 Dynamic C Differences Between Rabbit and Z180

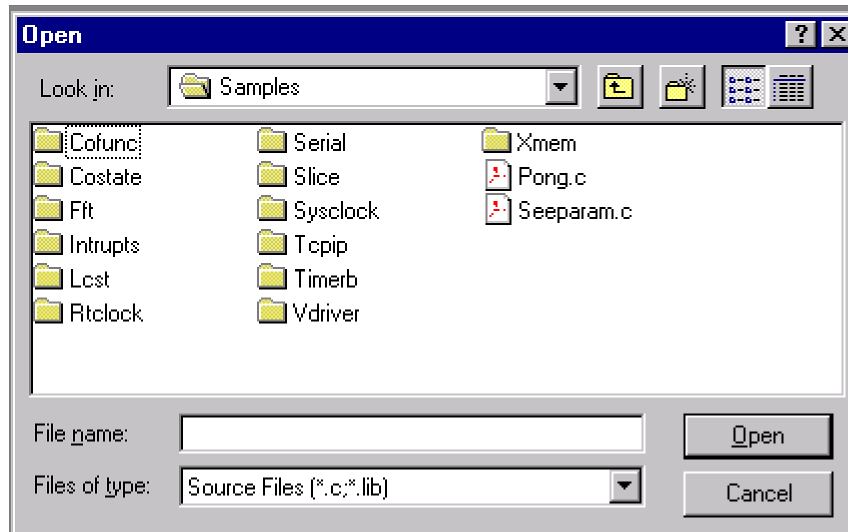
A major difference in the way Dynamic C interacts with a Rabbit-based board compared to a Z180 or 386EX board is that Dynamic C expects no BIOS kernel to be present on the target when it starts up. Dynamic C stores the BIOS kernel as a C source file. Dynamic C compiles and loads it to the Rabbit target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

- A socketed flash is no longer needed. BIOS updates can be made without a flash-EPROM burner since Dynamic C can communicate with a target that has a blank flash EPROM. Blank flash EPROM can be surface-mounted onto boards, reducing manufacturing costs for both Z-World and other board developers. BIOS updates can then be made available on the Web.
- Advanced users can see and modify the BIOS kernel directly.
- Board Developers can design Dynamic C compatible boards around the Rabbit CPU by simply following a few simple design guidelines and using a "skeleton" BIOS provided by Z-World.

- A major new feature introduced in Dynamic C 7.x is the ability to program and debug over the Internet or local Ethernet. This requires the use of a RabbitLink board, available alone or as an option with Rabbit-based development kits.

Quick Tutorial 3

Sample programs are provided in the Dynamic C **Samples** folder, shown below.



The subfolders contain sample programs that illustrate the use of the various Dynamic C libraries. The subfolder named Cofunc, for example, contains sample programs illustrating the use of **COFUNC.LIB**. The sample program **Pong.c** demonstrates output to the STDIO window. Each sample program has comments that describe its purpose and function.

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**. The program will appear in a window, as shown in Figure 1 below (minus some comments). Use the mouse to place the cursor on the function name **printf** in the program and press **<ctrl-H>**. This brings up a documentation box for the function **printf**. You can do this with all functions in the Dynamic C libraries, including libraries you write yourself. Close the documentation box.

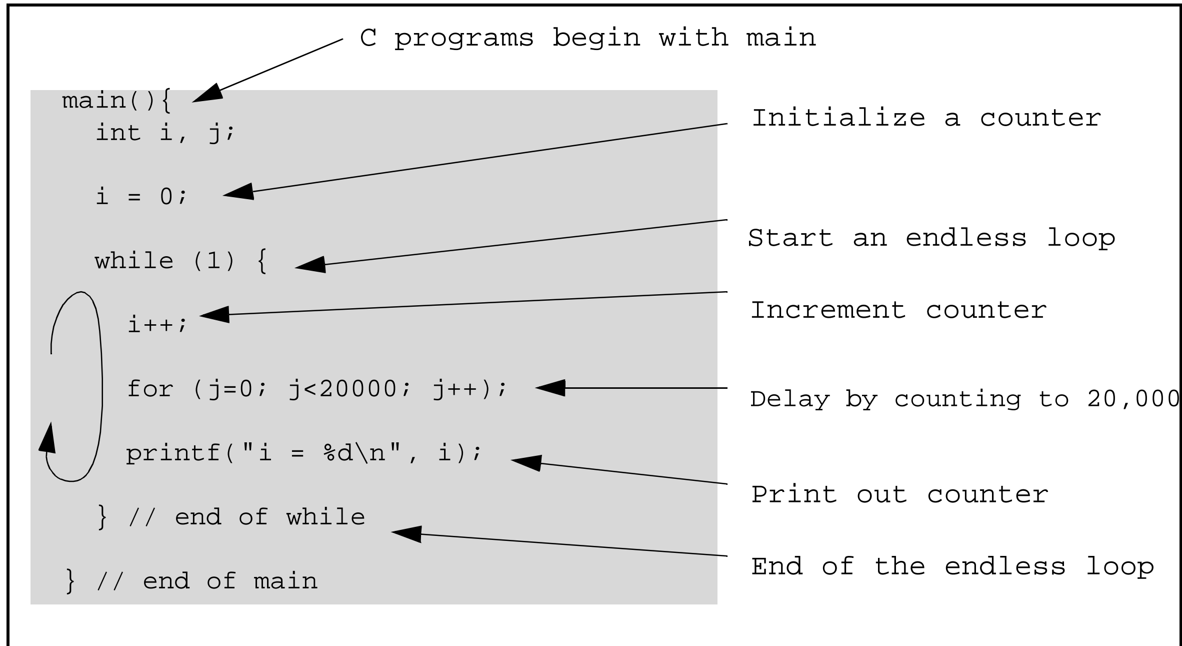


Figure 1. Sample Program DEMO1.C

To run the program **DEMO1.C**, open it with the **File** menu, compile it using the **Compile** menu, and then run it by selecting **Run** in the **Run** menu. The value of the counter should be printed repeatedly to the STDIO window if everything went well. If this doesn't work, review the following points:

- The target should be ready, indicated by the message "BIOS successfully compiled..." If you did not receive this message or you get a communication error, recompile the BIOS by typing **<ctrl-Y>** or select **Recompile BIOS** from the **Compile** menu.
- A message reports "No Rabbit Processor Detected" in cases where the wall transformer is either 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 Dynamic C **Options** menu must be the same as the one the programming cable is connected to.
- To check if you have the correct serial port, select **Compile**, then **Compile BIOS**, or press **<ctrl-Y>**. If the "BIOS successfully compiled …" message does not display, try a different serial port using the Dynamic C **Options** menu 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

Compile **DEMO1.C** by clicking the **Compile** button on the task bar. The program will compile and the screen will come up with a highlighted character (green) at the first executable statement of the program. Use the **F8** key to single-step. 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

Press **<ctrl-W>** or choose **Add/Del Watch Expression** in the **Inspect** menu. A box will come up. Type the lower case letter j and click on **Add to top**, then **Close**. Now continue single-stepping by pressing **F8**. Each time you step, the watch expression (`j`) will be evaluated and printed in the watch 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 **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** or select **Run** on the **Run** menu. 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** or select **Run** from the **Run** menu. Now the counter should be printing out regularly in the STDIO window.

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

3.1.4 Editing the Program

Click on the **Edit** box on the task bar. This will put Dynamic C into edit mode so that you can change the program. Use the **Save as** choice on the **File** menu to save the file with a new name so as not to change the 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 load the program **DEMO2.C** using the **File** menu **Open** command. 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. The statement

```
runwatch();
```

has been added as well. This is a debugging statement to view variables while the program is running. Use the **F9** key to compile and run **DEMO2.C**.

3.2.1 Watching Variables Dynamically

Press **<ctrl-W>** to open the watch window and add the watch expression **k** to the top of the list of watch expressions. Now press **<ctrl-U>**. Each time you press **<ctrl-U>**, you will see the current value of **k**.

As an experiment, 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 that comes with Dynamic C Premier.

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 out which key was entered.

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.

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

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

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 Summary of Features

This chapter provided a quick look at the intuitive interface of Dynamic C and some of the powerful options available for embedded systems programming.

3.4.1 Development Functions

When you load a program it appears in an edit 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 the **F9** key 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.

3.4.2 Single-stepping

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

3.4.3 Setting breakpoints

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

3.4.4 Watch expressions

A watch expression is a C expression that is evaluated on command in the watch window. An expression is basically any type of C formula that can include operators, variables and function calls, but not statements that require multiple lines such as `for` or `switch`. You can have a list of watch expressions in the watch window. If you are single-stepping, then they are all evaluated on each step. You can also command the watch expression to be evaluated by using the **<ctrl-U>** command. When a watch expression is evaluated at a breakpoint, it is evaluated as if the statement was at the beginning of the function where you are single-stepping. If your program is running you can also evaluate watch expressions with a **<ctrl-U>** if your program has a `runwatch()` command that is frequently executed. In this case, only expressions involving global variables can be evaluated, and the expression is evaluated as if it were in a separate function with no local variables.

3.4.5 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.

Language 4

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 (See “Function Reference” on page 153.) 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 C and Dynamic C.

4.1 C Language Elements

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

Table 1. C Language Elements

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 and Tokens

Punctuation marks serve as boundaries in C programs. The table below lists the punctuation marks and tokens.

Table 2. Punctuation Marks and Tokens

Symbol	Description
`:`	Terminates a statement label.
`;`	Terminates a simple statement or a `do` loop. C requires these!
`,`	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, or primitive, data types are shown below.

Table 3. Dynamic C Basic Data Types

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. (Z180 controllers do not have floating-point hardware.) Range: 1.18 x 10^{-38} to 3.40 x 10^{38}

The symbolic names for the hardcoded limits of the data types are defined in **`limits.h`** and are shown here.

```
#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. Prior to Dynamic C version 6.19, names were distinct up to 16 characters, but could 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
Cleanup_the_data_later  // not distinct!
```

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

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

Use the **#define** directive to create names for constants. These can be viewed as symbolic constants. See Section 4.5, “Macros.”

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

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

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

4.5 Macros

Macros can be defined in Dynamic C. A macro is a name replacement feature. Dynamic C has a text preprocessor that *expands* macros before the program text is compiled. The programmer assigns a name, up to 31 characters, to a fragment of text. Dynamic C then replaces the macro name with the text fragment wherever the name appears in the program. In this example,

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

the variable **i** gets the value **x * 72 + 12**. Macros can have parameters such as in the following example.

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

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

Dynamic C implements the **#** and **##** macro operators.

The **#** operator forces the compiler to interpret the parameter immediately following it as a string literal. For example, if a macro is defined

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

then the macro in

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

will expand to

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

and because C always concatenates adjacent strings, the final result of expansion will be

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

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

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

the macro in

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

will expand to

```
AASC6_FN();
```

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

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

The exceptions to this rule are

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

The following complex example illustrates this.

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

The code

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

will expand first to

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

then to

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

then to

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

and finally to

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

4.5.1 Restrictions

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

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

4.6 Numbers

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

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

```
10   -327   1000   0
```

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

```
0L   -32L   45000   32767L
```

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

```
0U   4294967294U   32767U   1700UL
```

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

```
0x7E   0xE000   0xFFFFFFFA
```

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

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

```
0177   020000   000000630
```

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

4.5 means 4.5
4f means 4.0
0.3125 means 0.3125

456e-31 means 456×10^{-31}
0.3141592e1 means 3.141592

4.7 Strings and Character Data

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

```
"Press any key when ready..."
```

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

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

The following example illustrates typical use of strings.

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

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

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

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

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

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

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

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

Examples

```
"He said \"Hello.\""    // embedded double quotes
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. Comments (the `/*...*/` kind) may occur almost anywhere, even in the middle of a statement, as long as they begin with `/*` and end with `*/`.

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

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

4.10 Functions

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

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

1. Functions that have not been compiled may be called.
2. Recursive functions may be written.
3. The compiler may perform type-checkingon 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 complex variable and ptr
```

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 end-of-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 args.
     float temp;                 // Local var.
     temp = a*a + b*b;           // Arithmetic.
     return( temp );             // Return value.
}
/* and here is the end of the program */
```

The program above calculates the sum of squares of two numbers, **g** and **h,** which are initialized to 10 and 12, respectively. The main function calls the **init** function to give values to the global

variables **g** and **h**. Then it uses the **sumSquare** function to perform the calculation and assign the result of the calculation to the variable **x**. It prints the result using the library function **printf**, which includes a formatting string as the first argument.

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

4.13 Aggregate Data Types

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

4.13.1 Array

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

```
int item[10];          // An array of 10 integers.
```

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

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

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

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

is referenced in a similar way.

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

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

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

4.13.2 Structure

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

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

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

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

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

4.13.3 Union

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

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

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

```
j = u.ival
```

4.13.4 Composites

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

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

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

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

4.14 Storage Classes

Variable storage can be **auto** or **static**. The default storage class is **static**, but can be changed by using **#class auto**. 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--meaning available anywhere--but there is no keyword in C to represent this fact. Global variables always have **static** storage

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

4.15 Pointers

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

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

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

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

Pointers may point to other pointers.

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

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

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

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

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

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

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

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

4.16 Pointers to Functions, Indirect Calls

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

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

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

for example:

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

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

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

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

```
typedef int (*fnptr)(); // create a pointer to int func.type
main(){
   int x,y;
   int (*fnc1)();  // declare a var. fnc1 as ptr to int func.
   fnptr  fp2;     // declare a var. fp2 as ptr to int func.
   fnc1 = intfunc; // initialize fnc1 to point to intfunc
   fp2 = intfunc;  // init. fp2 to point to the same func.

   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 *(exp_1)*, evaluates a terminating condition *(exp_2)*, and provides a stepping expression *(exp_3)* that is evaluated at the end of each iteration. Each of the three expressions is optional.

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

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

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

This loop initially sets **i** to 0, continues as long as **i** is less than **n** (stops when **i** equals **n**), and increments **i** at each pass.

Another use for the **for** loop is the infinite loop, which is useful in control systems.

```
for(;;){some statement(s)}
```

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

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

4.18.2 Continue and Break

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

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

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 “chain” of tests.

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

The program evaluates the first expression *($expr_1$)*. If that proves false, it tries the second expression *($expr_2$)*, and continues testing until it finds a true expression, an **else** clause, or the end of the if statement. An **else** clause is optional. Without an **else** clause, an **if/else if** statement that finds no true condition will execute none of the controlled statements.

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

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

First the **switch** *expression* is evaluated. It must have an integer value. If one of the **const**$_N$ values matches the **switch** *expression*, the sequence of statements identified by the **const**$_N$ expression is executed. If there is no match, the sequence of statements identified by the **default** label is executed. (The **default** part is optional.) Unless the **break** keyword is included at the end of the case’s statements, the program will “fall through” and execute the statements for any number of other cases. The **break** keyword causes the program to exit the **switch/case** statement.

The colons (**:**) after **break**, **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, or other kinds of tasks on request. There are two directives, **#makechain** and **#funcchain**, and one keyword, **segchain**.

#makechain *chain_name*

Creates a function chain. When a program executes the named function chain, all of the functions or chain segments belonging to that chain execute. (No particular order of execution can be guaranteed.)

#funcchain *chain_name name*

Adds a function, or another function chain, to a function chain.

segchain *chain_name { statements }*

Defines a program segment (enclosed in curly braces) and attaches it to the named function chain.

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

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

A program will call a function chain as it would an ordinary void function that has no parameters. 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 initialize variables and take initialization action with global initialization. This is done by adding segments to the **_GLOBAL_INIT** function chain, as shown in the example below.

The special directive **#GLOBAL_INIT{ }** tells the compiler to add the code in the block enclosed in braces to the **_GLOBAL_INIT** function chain. The **_GLOBAL_INIT** function chain is always called when your program starts up, so there is nothing special to do to invoke it. It may be called at anytime in an application program, but do this with caution. When it is called, all costatements and cofunctions will be initialized. See “Calling _GLOBAL_INIT()” on page 63 for more information.

Any number of **#GLOBAL_INIT** sections may be used in your code. The order in which the **#GLOBAL_INIT** sections are called is indeterminate since it depends on the order in which they were compiled.

```
long my_func( char j );
main(){
    my_func(100);
}
long my_func(char j){
    int i;
    long array[256];

    // The GLOBAL_INIT section is run
    // automatically once when 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
}
```

4.21 Libraries

Dynamic C is comprised of many libraries—files of useful functions. They are located in the **LIB** subdirectory where Dynamic C was installed. The default library file extension is **.LIB**.

Dynamic C will extract functions and data from library files and compile them with an application program that is then downloaded to a controller or saved to a **.bin** file.

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

```
main(){
}
```

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

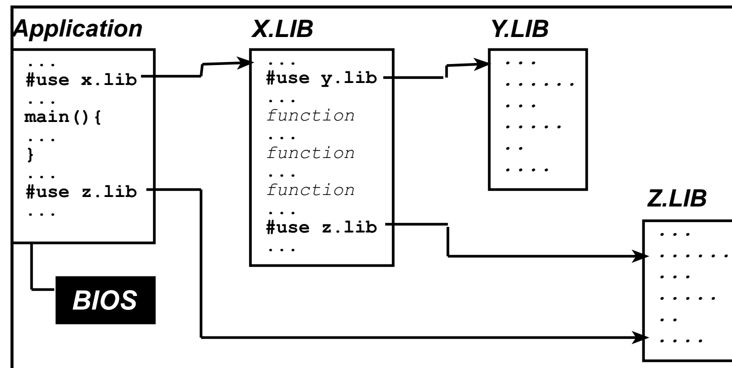


Figure 2. Nesting Files in Dynamic C

Most libraries needed by Dynamic C programs are **#use**’d in the file **lib\default.h**.

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

4.22 Support Files

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

.

Table 4. Dynamic C Support Files

File	Meaning
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.
LIB.DIR	Contains pathnames for all libraries that are to be known to Dynamic C. The programmer can add to, or remove libraries from this list. The factory default is for this file to contain all the libraries on the Dynamic C distribution disk. No library will be usable unless it is listed in this file.
DEFAULT.H	Contains a set of **#use** directives for each control product that Z-World ships. This file can be modified.

4.23 Headers

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

Table 5. Dynamic C Library Headers

Header	Description
Module headers	Makes 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.

4.24 Modules

This is a very important topic that must be understood by those writing their own libraries for Dynamic C. Modules provide Dynamic C with the ability to know which functions and global variables in a library to use.

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

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

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

4.24.1 The Key

The line (a specially-formatted comment)

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

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

If there are many names after **BeginHeader**, the list of names can continue on subsequent lines. All names must be separated by commas. A key can have no names in it and it's associated header will still be parsed by the precompiler and compiler.

4.24.2 The Header

Every line between the comments containing **BeginHeader** and **EndHeader** belongs to the *header* of the module. When an application **#use**s a library, Dynamic C compiles every header, and just the headers, in the library. The purpose of a header is to make certain names defined in a module known to the application. With proper function prototypes and variable declarations, a module header ensures proper type checking throughout the application program. Prototypes, variables, structures, typedefs and macros declared in a header section will always be parsed by the compiler if the library is used, and will have global scope. It is even permissible to put function bodies in header sections, but this is not recommended. Variables declared in a header section will be allocated memory space unless the declaration is preceded with **extern** .

4.24.3 The Body

Every line of code after the **EndHeader** comment belongs to the *body* of the module until (1) end-of-file or (2) the **BeginHeader** comment of another module. Dynamic C compiles the *entire* body of a module if *any* of the names in the key are referenced (used) anywhere in the application. For this reason, it is not wise to put many functions in one module regardless of whether they are actually going to be used by the program.

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

Example

```
/*** BeginHeader ticks */
    extern 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 further triggers compilation of the module body corresponding to **ticks** because the functions use the variable **ticks**.

4.24.4 Function Description Headers

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

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

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

RETURN VALUE:  None
KEY WORDS: parallel port

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

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

Multitasking with Dynamic C 5

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

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

5.1 Cooperative Multitasking

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

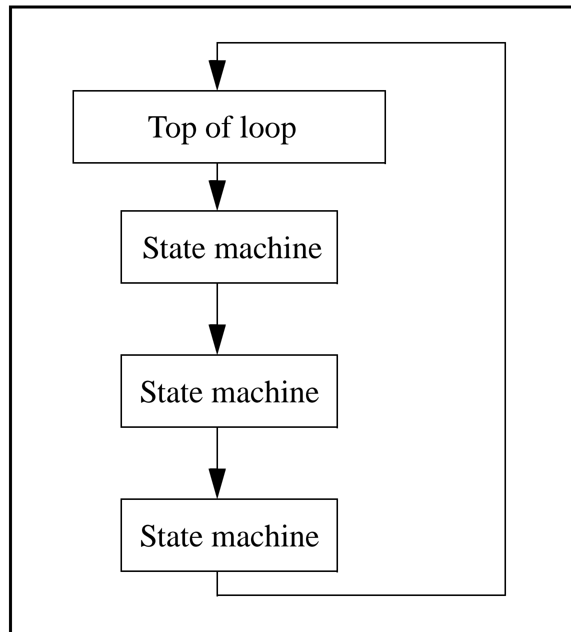


Figure 1. Big Loop

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

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

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

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

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

5.2 A Real-time Problem

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

Start:

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

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

5.2.1 Solving the Real-time Problem With a State Machine

Here is what a state machine solution might look like.

```
task1state = 1;                        // initialization:
while(1){
   switch(task1state){
      case 1:
         if( buttonpushed() ){
            task1state=2;   turndevice1();
            timer1 = time;  // time incremented every sec
         }
         break;
      case 2:
         if( (time-timer1) >= 60L){
            task1state=3;   turndevice2();
            timer2=time;
         }
         break;
      case 3:
         if( (time-timer2) >= 60L){
            task1state=1;   turnoffdevice1();
            turnoffdevice2();
         }
         break;
   }
   (other tasks or state machines)
}
```

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

5.3 Costatements

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

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

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

5.3.1 Solving the Real-time Problem With Costatements

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

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

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

    costate{ ... }                  // task n
}
```

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

5.3.2 Costatement Syntax

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

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

`name` can be one of the following:

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

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

`state` can be one of the following:

- **`always_on`**
  The costatement is always active. This option causes the costatement to be compiled in such a manner that it does not check for a paused condition. **`CoPause`** cannot be used.
- **`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 paused by **`CoPause.`**

If **`state`** is absent, a named costatement is initialized in a paused condition and will not execute until **`CoBegin`** or **`CoResume`** is executed. The costatement will then execute once and become inactive again.

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

5.3.3 Control Statements

`waitfor`(*expression*);

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

`yield`

The **`yield`** statement makes an unconditional exit from a costatement or a cofunction.

abort

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

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

5.4 Advanced Costatement Topics

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

5.4.1 The CoData Structure

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

5.4.2 CoData Fields

CSState

The **CSState** field contains two flags, **STOPPED** and **INIT,** summarized in the table below.

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

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

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

Check Sum

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

First Time

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

Content

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

Check Sum 2

The **ChkSum2** field is currently unused.

5.4.3 Pointer to CoData Structure

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

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

5.4.4 Library Extensions for Use With Named Costatements

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 will resume the next time it is called.

5.4.5 Firsttime Functions

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

The following **`firsttime`** functions are defined in **`COSTATE.LIB.`** For more information see Chapter 15, "Function Reference." These functions should be called inside a **`waitfor`** statement because they do not yield while waiting for the desired time to elapse, but instead return 0 to indicate that the desired time has not yet elapsed.

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

User-defined **`firsttime`** functions are allowed.

5.4.6 Shared Global Variables

These variables 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.

`SEC_TIMER`
`MS_TIMER`
`TICK_TIMER`

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

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.

5.5.3 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 49 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. Which one to use depends on the problem that is being solved.

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.

Costatements are not supported inside indexed cofunctions.

5.5.5.3 Single User Cofunction

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

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

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

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

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

loopinit()

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

loophead()

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

Example

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

5.5.6 Types of Cofunction Calls

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

5.5.6.1 First Time Call

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

5.5.6.2 Continuation Call

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

5.5.6.3 Terminal Call

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

5.5.6.4 Lifetime of a Cofunction Instance

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

5.5.7 Special Code Blocks

The following special code blocks can appear inside a cofunction.

everytime { *statements* }

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

abandon { ***statements*** }

This statement 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

The following code illustrates the use of **abandon**. This program, **COFABAND.C**, is in the **SAMPLES/COFUNC** folder in the directory where Dynamic C was installed.

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

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

5.5.8 Solving the Real-time Problem With Cofunctions

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

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

5.6 Patterns of Cooperative Multitasking

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

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

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

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

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

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

5.7 Timing Considerations

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

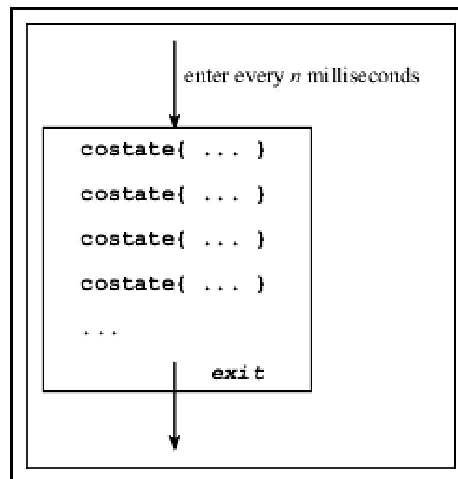


Figure 2. Costatement as Part of Real-Time Task

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

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

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

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

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

5.7.1 waitfor Accuracy Limits

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

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

5.8 Overview of Preemptive Multitasking

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

There are two ways to accomplish preemptive multitasking using Dynamic C. The first way is µC/OS-II, a real-time, preemptive kernel that runs on the Rabbit Microprocessor and is fully supported by Dynamic C. For more information see Chapter 17, “µC/OS-II.” The other way is to use **`slice`** statements.

5.9 Slice Statements

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

5.9.1 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 size for the **`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 statements, like costatements and cofunctions, can suspend its execution with an **`abort`**, **`yield`**, or **`waitfor`** as with costatements and cofunctions, or with an implicit **`yield`** determined by the **`time_slice`** parameter that was passed to it.

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

5.9.3 Restrictions

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

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

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

Slice statements cannot be used with µC/OS-II or **`DCRTCP.LIB`**.

5.9.4 Slice Data Structure

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

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

5.9.5 Slice Internals

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

5.9.5.1 Example 1

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

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

5.9.5.2 Example 2

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

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

5.9.5.3 Example 3

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

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

5.10 Summary

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

The Virtual Driver 6

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

Initialization Services

- Call **_GLOBAL_INIT()**
- Initialize the global timer variables
- Start the virtual driver periodic interrupt

Periodic Interrupt Services

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

6.1 Default Operation

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

6.2 Calling _GLOBAL_INIT()

VdInit calls **_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()** were not called, users could still use costatements and cofunctions if the call to **VdInit()** was replaced by a call to **_GLOBAL_INIT()**, but the **DelaySec()** and **DelayMs()** functions often used with costatements and cofunctions in **waitfor** statements would not work because those functions depend on timer variables which are maintained by the periodic interrupt.

6.3 Global Timer Variables

The following variables **SEC_TIMER, MS_TIMER** and **TICK_TIMER** are global variables defined as **shared unsigned long**. On initialization, **SEC_TIMER** is synchronized with the real time clock so that the date and time can be accessed more quickly than reading the real time clock simply by reading **MS_TIMER**.

The periodic interrupt updates **SEC_TIMER** every second, **MS_TIMER** every millisecond, and **TICK_TIMER** 2048 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 users to use for timing purposes. The following sample shows the use of **MS_TIMER** to measure the execution time in micro seconds of a Dynamic C integer add. The work is done in a "nodebug" function so that the 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 ms. to us.
   printf("time to execute test code = %f us\n",T2*1000.0);
}
```

6.4 Watchdog Timers

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

Hardware Watchdog

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

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

However, it is recommended that the watchdog not be disabled. This prevents the target from “locking up” by 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.

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 **`VdGetFreeW(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 **`VdGetFreeW`** is used as the argument when calling **`VdHitWd(ID)`** or **`VdReleaseWd(ID)`** to hit or deallocate a virtual watchdog

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

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

6.5 Preemptive Multitasking Drivers

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

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

The Slave Port Driver 7

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

7.1 Slave Port Driver Protocol

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

7.1.1 Overview

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

7.1.2 Registers on the Slave

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

`SPD0R`	Command and response register
`SPD1R`	Address register
`SPD2R`	Optional data register
`SPSR`	Slave port status register. In this protocol the only bits used in the status register are for checking the command/response register. Bit 3 is set if the slave has written a response to SPD0R. It is cleared when the master writes to SPSR, which also deasserts the SLAVEATTN line.

Reading and writing to the same address actually uses two different registers.

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

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

Pin assignments for a Rabbit processor acting as a slave are as follows:

Pin	Function
PE7	/CS chip select (active low to read/write slave port)
PB2	/SWR slave write (assert for write cycle)
PB3	/SRD slave read (assert for read cycle)
PB4	A0 low address bit for slave port registers
PB5	A1 high address bit for slave registers
PB7	/SLVATTN asserted by slave when it responds to a command. cleared by master write to status register
PA0-PA7	slave port data bus

For more details and read/write signal timing see the Rabbit 2000 Microproccessor Manual.

7.1.3 Polling and Interrupts

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

7.1.4 Communication Channels

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

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

7.2 Functions

`Slave_port.lib` provides the following functions:

SPinit

`int SPinit ( int mode );`

DESCRIPTION

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

PARAMETERS

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

RETURN VALUE

`1`: Success
`0`: Failure

LIBRARY

`Slave_port.lib`

SPsetHandler

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

DESCRIPTION

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

PARAMETERS

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

RETURN VALUE

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

LIBRARY

`Slave_port.lib`

MyHandler

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

DESCRIPTION

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

PARAMETERS

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

RETURN VALUE

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

LIBRARY

This is a developer-supplied function.

SPtick

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

DESCRIPTION

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

LIBRARY

`Slave_port.lib`

SPclose

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

DESCRIPTION

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

LIBRARY

`Slave_port.lib`

7.3 Examples

7.3.1 Example of a Simple Status Handler

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

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

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

7.3.2 Example of a Serial Port Handler

`Slave_port.lib` contains handlers for all four serial ports on the slave.

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

7.3.2.1 Commands to the Slave

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

a. The actual baud rate is the baud factor multiplied by 300.

7.3.2.2 Slave Side of Protocol

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

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

7.3.2.3 Master Side of Protocol

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

cof_MSgetc

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

DESCRIPTION

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

PARAMETERS

address	Slave channel address of the serial handler.

RETURN VALUE

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

LIBRARY

Master_serial.lib

cof_MSputc

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

DESCRIPTION

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

PARAMETER

address	Slave channel address of serial handler
ch	Character to send

RETURN VALUE

0: Character was sent
-1: Failure

LIBRARY

Master_serial.lib

cof_MSread

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

DESCRIPTION

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

PARAMETERS

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

RETURN VALUE

Bytes read, or
-1: Failure

LIBRARY

`Master_serial.lib`

cof_MSwrite

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

DESCRIPTION

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

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

RETURN VALUE

Number of bytes actually written,
-1 if error

LIBRARY

`Master_serial.lib`

MSclose

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

DESCRIPTION

Closes a serial port on the slave.

PARAMETERS

address	Slave channel address of serial handle.

RETURN VALUE

0: Success
-1: Failure

LIBRARY

Master_serial.lib

MSgetc

```
int MSgetc(char address);
```

DESCRIPTION

Receives a character from the serial port.

PARAMETERS

address	Slave channel address of serial handler.

RETURN VALUE

Value of recevied character;
-1: No character available.

LIBRARY

MASTER_SERIAL.LIB

MSgetError

```
int MSgetError(char address);
```

DESCRIPTION

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

SER_PARITY_ERROR 0x01
SER_OVERRUN_ERROR 0x02

PARAMETERS

address	Slave channel address of serial handler.

RETURN VALUE

Number of bytes free: Success
-1: Failure

LIBRARY

MASTER_SERIAL.LIB

MSinit

```
int MSinit(int io_bank);
```

DESCRIPTION

Sets up the connection to the slave.

PARAMETERS

io_bank	The IO bank and chip select pin number for the slave device (0-7).

RETURN VALUE

1: Success

LIBRARY

Master_serial.lib

MSopen

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

DESCRIPTION

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

PARAMETERS

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

RETURN VALUE

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

LIBRARY

MASTER_SERIAL.LIB

MSputc

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

DESCRIPTION

Transmits a single character through the serial port.

PARAMETERS

address	Slave channel address of serial handler
ch	Character to send

RETURN VALUE

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

LIBRARY

MASTER_SERIAL.LIB

MSrdFree

```
int MSrdFree(char address);
```

DESCRIPTION

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

PARAMETERS

address	Slave channel address of serial handler.

RETURN VALUE

Number of bytes free: Success
-1: Failure

LIBRARY

`Master_serial.lib`

MSsendCommand

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

DESCRIPTION

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

PARAMETERS

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

RETURN VALUE

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

LIBRARY

`MASTER_SERIAL.LIB`

MSread

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

DESCRIPTION

Receives bytes from the serial port on the slave.

PARAMETERS

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

RETURN VALUE

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

LIBRARY

`Master_serial.lib`

MSwrFree

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

DESCRIPTION

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

PARAMETERS

address	Slave channel address of serial handler

RETURN VALUE

Number of bytes free: Success
-1: Failure

LIBRARY

`Master_serial.lib`

MSwrite

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

DESCRIPTION

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

PARAMETERS

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

RETURN VALUE

Number of bytes actually sent.

LIBRARY

Master_serial.lib

7.3.2.4 Sample Program for Master

This sample program, **master_demo.c**, treats the slave like a serial port.

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

char* const test_string = "Hello There";

main(){
   char buffer[100];
   int read_length;

   MSinit(0);

   //comment this line out if talking to a stream handler
   printf("open returned:0x%x\n", MSopen(SP_CHANNEL, 9600));

   while(1)
   {
      costate
      {
         wfd{cof_MSwrite(SP_CHANNEL, test_string, strlen(test_string));}
         wfd{cof_MSwrite(SP_CHANNEL, test_string, strlen(test_string));}
      }
      costate
      {
         wfd{ read_length = cof_MSread(SP_CHANNEL, buffer, 99, 10); }
         if(read_length > 0)
         {
            buffer[read_length] = 0; //null terminator
            printf("Read:%s\n", buffer);
         }
         else if(read_length < 0)
         {
            printf("Got read error: %d\n", read_length);
         }
         printf("wrfree = %d\n", MSwrFree(SP_CHANNEL));
      }
   }
}
```

7.3.3 Example of a Byte Stream Handler

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

7.3.3.1 Slave Side of Stream Channel

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

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

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

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

7.3.3.1.1 Functions

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

cbuf_init

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

DESCRIPTION

This function initializes a circular buffer.

PARAMETER

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

LIBRARY

Rs232.lib

cof_SPSread

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

DESCRIPTION

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

PARAMETERS

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

RETURN VALUE

The number of bytes read from the buffer.

LIBRARY

SP_STREAM.LIB

cof_SPSwrite

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

DESCRIPTION

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

PARAMETERS

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

RETURN VALUE

The number of bytes successfully written to slave port.

LIBRARY

SP_STREAM.LIB

SPSinit

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

DESCRIPTION

Initializes the circular buffers used by the stream handler.

LIBRARY

SP_STREAM.LIB

SPSread

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

DESCRIPTION

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

PARAMETERS

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

RETURN VALUE

Number of bytes read into the data buffer

LIBRARY

SP_STREAM.LIB

SPSwrite

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

DESCRIPTION

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

PARAMETERS

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

RETURN VALUE

Number of bytes written into the data buffer

LIBRARY

SP_STREAM.LIB

SPSwrFree

```
int SPSwrFree();
```

DESCRIPTION

Returns number of free bytes in the stream write buffer.

RETURN VALUE

Space available in the stream write buffer.

LIBRARY

SP_STREAM.LIB

SPSrdFree

```
int SPSrdFree();
```

DESCRIPTION

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

RETURN VALUE

Space available in the stream read buffer.

LIBRARY

SP_STREAM.LIB

SPSwrUsed

```
int SPSwrUsed();
```

DESCRIPTION

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

RETURN VALUE

Number of bytes currently in the stream write buffer.

LIBRARY

SP_STREAM.LIB

SPSrdUsed

```
int SPSrdUsed();
```

DESCRIPTION

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

RETURN VALUE

Number of bytes currently in the stream read buffer.

LIBRARY

SP_STREAM.LIB

7.3.3.2 Byte Stream Sample Program

This program runs on a slave and implements a byte stream over the slave port.

```
/*
 * Slave_Port.c
/*
#use "slave_port.lib"
#use "sp_stream.lib"

#define STREAM_BUFFER_SIZE 31

main()
{
   char buffer[10];
   int bytes_read;

   SPStream stream;
   // Circular buffers need 9 bytes for bookkeeping.
   char stream_inbuf[STREAM_BUFFER_SIZE + 9];
   char stream_outbuf[STREAM_BUFFER_SIZE + 9];
   SPStream *stream_ptr;

   //setup buffers
   cbuf_init(stream_inbuf, STREAM_BUFFER_SIZE);
   stream.inbuf = stream_inbuf;
   cbuf_init(stream_outbuf, STREAM_BUFFER_SIZE);
   stream.outbuf = stream_outbuf;

   stream_ptr = &stream;
   SPinit(1);
   SPsetHandler(0x42, SPShandler, stream_ptr);

   while(1)
   {
      bytes_read = SPSread(stream_ptr, buffer, 10, 10);
      if(bytes_read)
      {
         SPSwrite(stream_ptr, buffer, bytes_read);
      }
   }
}
```

Efficiency 8

There are a number of methods that can be used to reduce the size of a program, or to increase its speed.

8.1 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 function that handles **`RST 28H`** instructions will be replaced by a simple **`ret`** instruction. 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 **`nodebug`** keyword with the **`#asm`** directive.

Use the directive **`#nodebug`** anywhere within the program to enable **`nodebug`** for all statements following the directive. The **`#debug`** directive has the opposite effect.

8.2 Static Variables

Using **`static`** variables with **`nodebug`** functions will increase the program speed greatly. Stack checking is disabled by default.

When there are more than 128 bytes of auto variables declared in a function, the first 128 bytes are more easily accessed than later declarations because of the limited 8-bit range of IX and SP register addressing. Performance is, therefore, slower for bytes above 128.

The **`shared`** and the **`protected`** keywords in data declarations cause slower fetches and stores, except for one-byte items and some two-byte items.

8.3 Function Entry and Exit

The following events occur when a program enters a function.

1. The function saves IX on the stack and makes IX the stack frame reference pointer (if the program is in the **`useix`** mode).
2. The function creates stack space for **`auto`** variables or to save **`register`** variables.
3. The function sets up stack corruption checks if stack checking is enabled (on).
4. The program notifies Dynamic C of the entry to the function so that single-stepping modes can be resolved (if in debug mode).

Items three and four consume significant execution time and are eliminated when stack checking is disabled or if the debug mode is off.

Run-Time Error Processing 9

Compiled code generated by Dynamic C calls an error-handling routine for abnormal situations. The error handler supplied with Dynamic C prints internally defined error messages to a Windows message box when runtime error messages are detected during a debugging session. When software runs stand-alone (disconnected from Dynamic C), such an error message will cause a watchdog timeout and reset.

The table below lists the ranges of Dynamic C error codes.

Table 6. Ranges of Dynamic C Error Codes

Code	Meaning
0–99	User, nonfatal.
100–127	System, nonfatal.
128–227	User, fatal, no return possible.
228–255	System, fatal, no return possible.

This table lists the fatal errors generated by Dynamic C.

Table 7. Dynamic C Fatal Errors

Code	Meaning
228	Pointer store out of bounds
229	Array index out of bounds
230	Stack corrupted
231	Stack overflow
232	Aux stack overflow
233	*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	*not used*
254	*not used*
255	*not used*

9.1 User-defined error handlers

It is possible that a user may want to develop their own runtime error handler. They may want to add their own runtime errors that would require special treatment, or simply add code that logs the runtime error data to memory.

Here is a particular 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 Dynamic C action is to halt with a runtime error if that state occurs. If continued execution was desired (the function in question would return a value of INF or whatever value is appropriate), then a simple error handler could be written by a user to pass execution back to the program when a domain or range error occurs, and pass any other runtime errors to Dynamic C.

A runtime error occurs by a call to **exception()**. The runtime error code is passed to the function; **exception()** pushes various parameters on the stack, and the installed error handler is called. 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 runtime error to the user.

To tell the BIOS to use a custom error handler, the following function should be called:

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

This function sets the BIOS function pointer for runtime errors to the one passed to it. The exception function provides data on the stack as described in Figure 8..

Table 8. Stack setup for runtime 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() called (upper byte)
SP+8	Address where exception() called

If the runtime error is to be passed to Dynamic C (i.e. it should halt or reset the system), then registers should be loaded appropriately and the **_xexit** function should be called. Dynamic C expects the following values to be loaded: H should contain the XPC when **exception()** was called, L should contain the runtime error code, and HL' should contain the address where **exception()** was called.

Memory Management 10

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

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

10.1 Memory Map

A typical Dynamic C memory mapping of logical and physical address space is shown in the figure below.

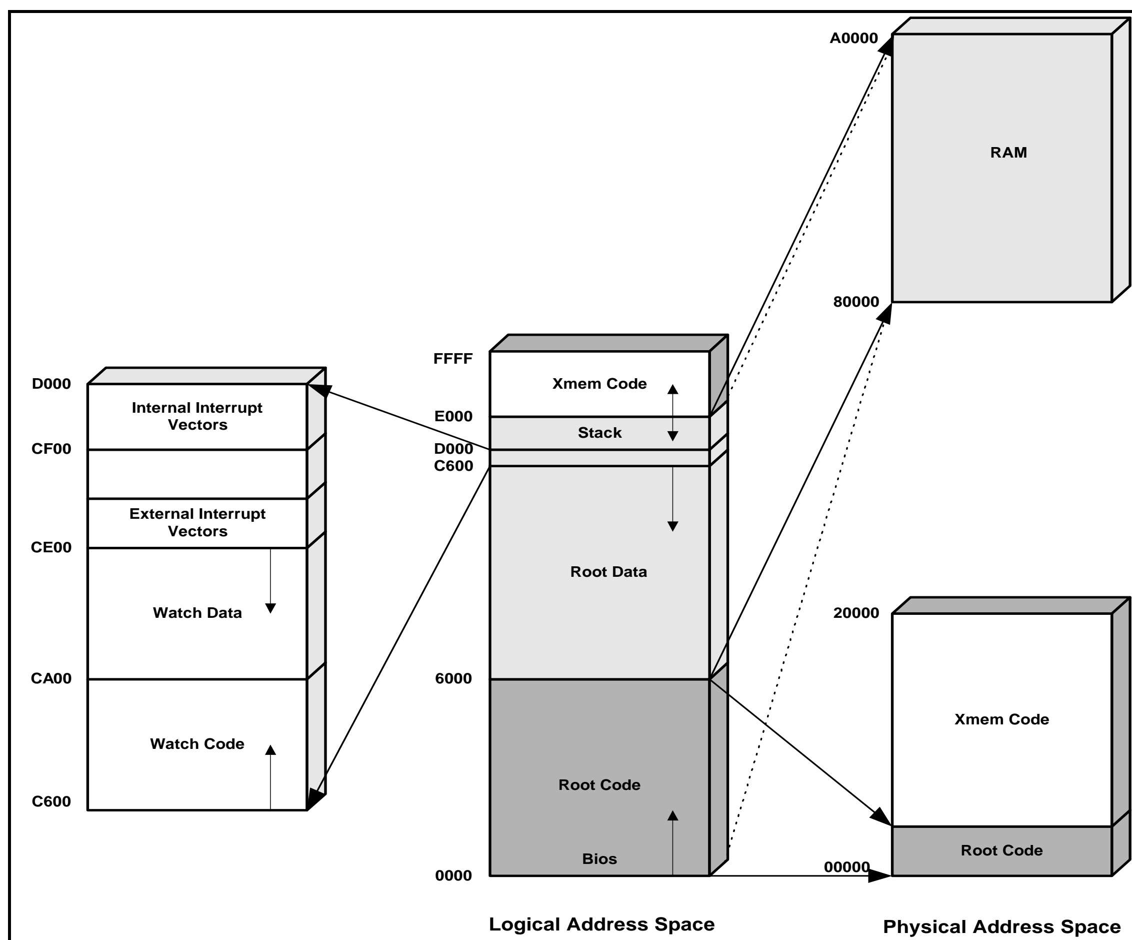


Figure 3. Dynamic C Memory Mapping

This figure 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 be place at a starting address 0x80000.

10.1.1 Memory Mapping Control

The advanced user of Dynamic C may control how Dynamic C allocates and maps memory.

For further details on memory mapping, refer to the ***Rabbit Microprocessor*** manual.

10.2 Extended Memory Functions

While any C function can call any other C function, no matter where it is located in memory, calling a function located in extended memory is less efficient than calling a function in root memory.

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

Extended memory addresses are 20-bit physical addresses (the lower 20 bits of a long integer). Pointers, on the other hand, are 16-bit machine addresses. They are not interchangeable. However, there are library functions to convert address formats.

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

10.2.1 Code Placement in Memory

Using the keywords **`xmem`** and **`root`**, there is some flexibility with regard to code placement in memory.

Pure Assembly Routines

Pure assembly functions (not inline assembly code) must reside in root memory. The keyword **`xmem`** does not apply to these pure assembly functions.

C Functions

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

Inline Assembly in C Functions

Inline assembly code may be written in any C function, regardless of whether it is compiled to extended memory or root memory.

However, because the stack frame of an extended memory function introduces four more bytes between the last pushed argument and the return address, the actual offset of arguments from the stack pointer depends on whether the code is compiled to extended memory or not. Therefore, it is important to use the symbolic names of stack-based variables instead of numeric offsets to access the variables. For example, if **`j`** is a stack variable, **`@sp+j`** is the actual offset of the variable from the stack pointer. Alternatively, if IX is the frame reference pointer, **`ix+j`** specifies the address of the stack-based variable.

Dynamic C issues a warning when it finds assembly code embedded in an extended memory function to discourage inline assembly segments that do not use symbolic offsets for stack-based variables.

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.

The Flash File System 11

Starting with Dynamic C 7.0, a simple file system has been added that should meet most people's needs. It can be used with a second Flash Memory or in SRAM (recommended for debugging purposes only).

The Dynamic C file system supports a total of 127 files. By default, blocks are allocated in 4096 byte chunks. A file, regardless of size, is comprised of at least one block. Files larger than the block size will be allocated multiple blocks which are not necessarily contiguous in memory.

The low-level Flash Memory access functions should not be used in the same area of the flash where the flash file system exists.

11.1 General Usage

Some care must be taken when using the file system. Since a Flash Memory is a finite resource, quickly writing data to the flash could result in using up the its write cycles. For a 256KB flash, we have 64 blocks. Each write to the flash uses up a single write. If you are using a flash with a maximum recommendation of 10,000 write cycles, we are limited in writing 640,000 times to the file system and 6,400,000 times for a 100,000 write cycle flash.

If you are performing one write to the flash per second, you will quickly use up the recommended lifetime of the flash within a week. You can increase the useful lifetime of the flash by buffering data before you write it to the flash. If you accumulate 1000 single byte writes into one, you can expand the life of the flash by an average of 750 times.

The main use of a flash file system should be for infrequently changing data or data rates that have writes on the order of tens of minutes instead of seconds.

Wear Leveling

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

Low-level implementation

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

11.2 Application Requirements

To use the file system, a macro that determines which low-level driver is loaded must be defined in the application program.

```
#define FS_FLASH    // use 2nd flash for file system
#define FS_RAM      // use SRAM (supported for debug purposes)
```

The file system library must be compiled with the application.

```
#use “FILESYSTEM.LIB”
```

11.3 Functions

These functions are the file system API. For a complete description see “Function Reference” on page 153.

Command	Description
fs_init	Initialize the internal data structures for the file system.
fs_format	Initialize the Flash Memory and the internal data structures.
fs_reserve_blocks	Reserve blocks for privileged files.
fsck	Verify data integrity of files.
fcreate	Create a file and open it for writing.
fcreate_unused	Create a file with an unused file number.
fopen_rd	Open a file for reading.
fopen_wr	Open a file for writing (also opens it for reading.)
fshift	Removes specified number of bytes from file.
fwrite	Write to the end of a file.
fread	Read from the current file pointer.
fseek	Move the read pointer.
ftell	Return the current offset of the file pointer.
fclose	Close a file.
fdelete	Delete a file.

Table 1. Flash File System API

The functions **fs_init** and **fs_format** are similar, in that they both start the file system. Use **fs_format** to erase all blocks in the file system. This function’s third parameter, **wearlevel**, should be **1** for a new Flash Memory; otherwise it should be **0** to use the current wear leveling.

Use **fs_init** to preserve blocks that are in use and to do an integrity check of them. In case of loss of power, **fs_init** will delete any blocks that may be partially written and will substitute

the last known good block for that file. This means that any changes to the file that occurred between the last write and the power outage would be lost.

Using File Names

To associate a descriptive name with a file, there are several functions in **ZSERVER.LIB** that will be useful for this purpose. The file must already 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, which is why 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 *Dynamic C's TCP/IP User's Manual*, or use <**CTRL-H**> in Dynamic C to use the Library Lookup feature.

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 (1-127). 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.

Table 2. Flash File System Auxiliary Functions

11.4 Skeleton Program

The following program uses many of the file system commands. It writes several strings into a file, reads the file back and prints the contents to the STDIO window. The macro **RESERVE** should be 0 when the file system is in SRAM. When the file system is in Flash Memory you can adjust where it starts by defining **RESERVE** to be 0 or a multiple of the block size.

```
#define FS_FLASH
#use "FILESYSTEM.LIB"
#define FORMAT
#define RESERVE  0L
#define BLOCKS  64
#define TESTFILE 1

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

#ifdef FORMAT
   fs_format(RESERVE,BLOCKS,1);
   if(fcreate(&file,TESTFILE)) {
      printf("error creating TESTFILE\n");
      return -1;
   }
#else
   fs_init(RESERVE,BLOCKS);
   if(fopen_wr(&file,TESTFILE) {
      printf("error opening TESTFILE\n");
      return -1;
   }
#endif
   fwrite(&file,"hello",6);
   fwrite(&file,"12345",6);
   fwrite(&file,"67890",6);

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

After running this program at least once, comment out "**#define FORMAT**". You will see that it runs in a similar fashion, but now the file is appended using **fopen_wr** instead of being erased by **fs_format** and then recreated with **fcreate**.

For a more robust program, more error checking should be included.

Using Assembly Language 12

Dynamic C permits programing in assembly language. Assembly-language statements may either be embedded in a C function or entire functions may be written in assembly language. C statements may also be embedded in assembly code and refer to C-language variables in the assembly code.

For further details on specific assembly instructions, refer to the ***Rabbit 2000 Microprocessor User's Manual***.

12.1 Program Flow

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.

```
void eightadd( char *ch1, char *ch2 ){
#asm
    ld   hl,(sp+ch2)              ; get source pointer
    ex   de,hl                    ; save in de
    ld   hl,(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 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**).

12.1.1 Embedded C in Assembly

A C statement may be placed within assembly code by placing a **C** in column 1. For example, initialize global variables.

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

The keyword **nodebug** can be placed on the same line as **#asm**. The main reason for the **nodebug** option is to prevent Dynamic C from running out of debugger table memory, and the option saves space and unnecessary calls to the debugger kernel. If **nodebug** is specified for an entire function, then all the blocks of assembly code within the function are assembled in **nodebug** mode. There is no need to place the **nodebug** directive on each block.

A program may be debugged at the assembly language level by opening the assembly window. Single-stepping and breakpoints are supported in the assembly window. When the assembly window is open, single-stepping occurs instruction by instruction rather than statement by statement.

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

12.2 Comments

C-style commenting is 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.

12.3 Labels

A label is a name followed by one or two colons. A label followed by a single colon is *local*, whereas one followed by two colons is *global*. A local label is not visible to the code out of the current embedded assembly segment (i.e., code before the **#asm** or after the **#endasm** directive).

Unless it is followed immediately by the assembly language keyword **equ**, the label identifies the current code segment address. If the label is followed by **equ**, the label "equates" to the value of the expression after the keyword **equ**.

Because C preprocessor macros are expanded in embedded assembly code, Z-World recommends that preprocessor macros be used instead of **equ** whenever possible.

12.4 Defining Constants

Constants may be created and defined in assembly code. The assembly language keyword **db** ("define byte") places bytes at the current code segment address. The keyword **db** should be followed immediately by numerical values and strings separated by commas as shown here.

Example

Each of the following defines a string **"ABC"** in code space.

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

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

The assembly language keyword **dw** defines 16-bit *words*, least significant byte first. The keyword **dw** should be followed immediately by numerical values, as shown in the following example.

Example

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

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

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

12.5 Expressions

The assembler parses most C language constant expressions. A C language constant expression is one whose value is known at compile time. All operators except the following are supported.

?:	conditional
[ ]	array index
.	dot
->	points to
*****	dereference
sizeof()	

12.6 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
```

12.7 Special Symbols

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

Table 3. 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.
@RETVAL	Evaluates the offset from the *frame reference point* to the stack space reserved for the **struct** function returns.
@LENGTH	Determines the next reference address of a variable plus it size.

12.8 C Variables

C variable names may be used in assembly language. What a variable name represents (the value associated with the name) depends on the variable. For a global, static local, or register local variable, the name represents the *address* of the variable in root memory. For an **auto** variable or formal argument, the variable name represents its own *offset* from the frame reference point.

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

```
struct s {
    int x;
    int y;
    int z;
};
```

the embedded assembly expression **s+x** evaluates to 0, **s+y** evaluates to 2, and **s+z** evaluates to 4, regardless of where structure **s** may be.

In nested structures, offsets can be composite, as shown here.

```
struct s {
    int x;              // s+x = 0
    struct a{           // s+a = 2
        int b;          // a+b = 0   s+a+b = 2
        int c;          // a+c = 2   s+a+c = 4
    }
};
```

12.9 Stand-alone Assembly Code

A stand-alone assembly function is one that is defined outside the context of a C language function. It can have no **auto** variables and no formal parameters. Dynamic C always places a stand-alone assembly function in root memory.

When a program calls a function from C, it puts the first argument into a *primary registe*r. 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 BCDE (with register B containing the most significant byte). Assembly-language code can use the first argument very efficiently. *Only* the first argument is put into the primary register, while *all* arguments—including the first, 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 BCDE.

Assembly language allows assumptions to be made about arguments passed on the stack, and **auto** variables can be defined by reserving locations on the stack for them. However, the offsets of such implicit arguments and variables must be kept track of. If a function expects arguments or needs to use stack-based variables, Z-World recommends using the embedded assembly techniques described in the next section.

12.10 Embedded Assembly Code

When embedded in a C function, assembly code can access arguments and local variables (either **`auto`** or **`static`**) by name. Furthermore, the assembly code does not need to manipulate the stack because the functions **`prolog`** and **`epilog`** already do so.

The concept and structure of a *stack frame* must be understood before correct embedded assembly code can be written. A stack frame is a run-time structure on the stack that provides the storage for all **`auto`** variables, function arguments and the return address. The following figure shows the general appearance of a stack frame.

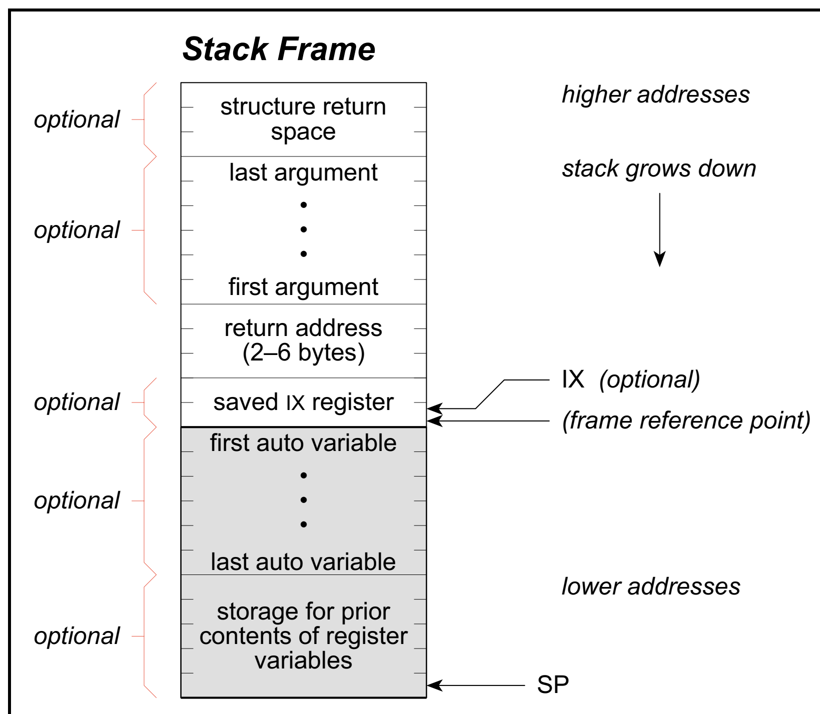


Figure 4. General Appearance of Assembly Code Stack Frame

The return address is always necessary. The presence of auto variables and register variables depends on the definition of the function. The presence of arguments and structure return space depends on the function call. (The stack pointer may actually point lower than the indicated mark temporarily because of temporary information pushed on the stack.)

The shaded area in the stack frame is the stack storage allocated for **`auto`** and **`register`** variables. The assembler symbol **`@SP`** represents the size of this area. The meaning of this symbol will become apparent later.

The following sections describe how to access local variables in various types of functions.

12.10.1 Not Using the IX Register, Function in Root Memory

Assume this simple function has been called.

```
int gi;                    // this is a global variable
root nouseix
void func( char ch, int i, long lg ){
    auto     int x;
    static   int y;
    register int z;
#asm
        some assembly code referencing gi, ch, i, lg, x, y,
and z
#endasm
}
```

The following figure shows how the stack frame will appear after the function call.

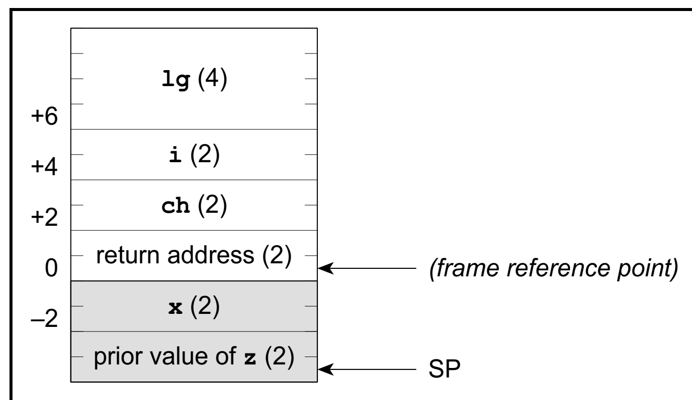


Figure 5. Assembly Language Stack Frame
No IX, Function in Root Memory

The symbols for **gi**, **ch**, **i**, **lg**, **x**, **y**, and **z** will have the following values when used in the assembly code

.

lg	offset = +6	**gi**	16-bit address (in root memory)
i	offset = +4	**x**	offset = -2
ch	offset = +2	**y, z**	16-bit address (in root memory)

There is a common method to access the stack-based variables **lg**, **i**, **ch**, and **x**. Consider, for example, the case of loading variable **x** into HL.

The following code (using the symbol **@SP**) is one way to do it.

```
ld hl,@SP+x  ; hl ← the offset from SP to the variable
add hl,sp    ; hl ← the address of the variable
ld a,(hl)    ; a ← the LSB of x
inc hl       ; hl now points to the MSB of x
ld h,(hl)    ; h ← the MSB of x
ld lg,a      ; lg ← the LSB of x
;; at this point, hl has the value of x
```

For static variables (**gi**, **y**, and **z**), the access is much simpler because the symbol evaluates to the address directly. The following code shows, for example, how to load variable **y** into HL.

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

12.10.2 Using the IX Register, Function in Root Memory

Access to stack-based local variables is fairly inefficient. The efficiency improves if there is a register for a frame pointer. Dynamic C can use the register IX as a frame pointer. The function in the previous section would then become the following.

```
int gi;                   // this is a global variable
root useix
void func( char ch, int i, long lg ){
    auto     int x;
    static   int y;
    register int z;
#asm
    some assembly code referencing gi, ch, i, lg, x, y, and z
#endasm
}
```

The keyword **useix** is the only change from the previous sample function. The following figure shows the stack frame for this function. IX points to the frame reference point.

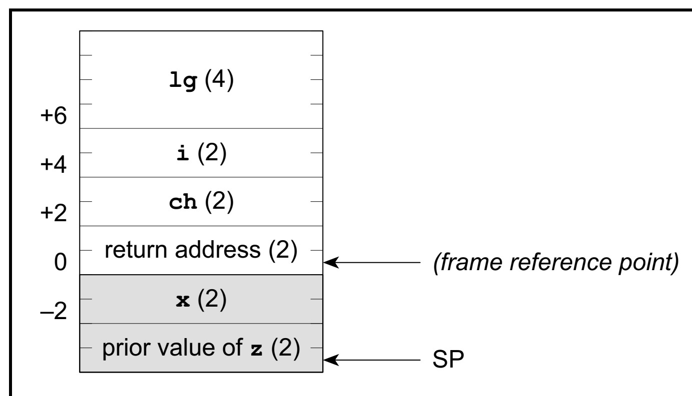


Figure 6. Assembly Language Stack Frame Using IX, Function in Root Memory

The arguments will have slightly different offsets because of the additional two bytes for the saved IX register value.

lg	offset = +8
i	offset = +6
ch	offset = +4

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 14 cycles and three bytes. If the program needs to load a four-byte variable such as **lg**, the IX+offset instructions are as follows.

```
ld e,(ix+lg)   ; load LSB of lg
ld d,(ix+lg+1) ;
ld c,(ix+lg+2) ;
ld b,(ix+lg+3) ; load MSB of lg
```

This takes a total of 56 cycles and 12 bytes. Even if IX is the frame reference pointer, the **@SP** symbol may still be used.

```
ld hl,@SP+lg ; hl ← the offset from SP to the variable
add hl,sp    ; hl ← the address of the variable
ld hl,(sp+@SP+lg); hl ← the address of the variable
ld e,(hl)    ; e ← the LSB of lg
inc hl       ;
ld d,(hl)    ;
inc hl       ;
ld c,(hl)    ;
inc hl       ;
ld b,(hl)    ; b ← the MSB of lg

; A faster way to do it with the Rabbit if
; the offset of lg < 127

ld hl,(sp+@SP+lg+2); load the LSW of lg
ld b,h
ld c,l
ld hl,(sp+@SP+lg)  ; load the LSW of lg
ex de,hl
```

This takes 52 cycles and 11 bytes. The two approaches are competitive. Nonetheless, the use of IX+offset is always beneficial when used to access single- or double-byte variables.

The offset from IX is a signed 8-bit integer. To use IX+offset, the variable must be within +127 or –128 bytes of the frame reference point. The **@SP** method is the only method for variables out of this range, even if IX is used as a frame reference pointer.

12.10.3 Not Using the IX Register, Function in Extended Memory

Functions that are (possibly) compiled to extended memory are not much different from functions compiled to root memory. Examine this extended memory function.

```
int gi;                    // this is a global variable
xmem
void func( char ch, int i, long lg ){
    auto     int x;
    static   int y;
    register int z;
#asm
    some assembly code referencing gi, ch, i, lg, x, y, and z
#endasm
}
```

If the **xmem** keyword is present, Dynamic C compiles the function to extended memory. Otherwise, Dynamic C determines where to compile the function. Note that funcitons compiled to extended memory have a 3-byte return address instead of a 2-byte return address. In this example, the IX register is not used. Figure 7 shows the stack frame of the function.

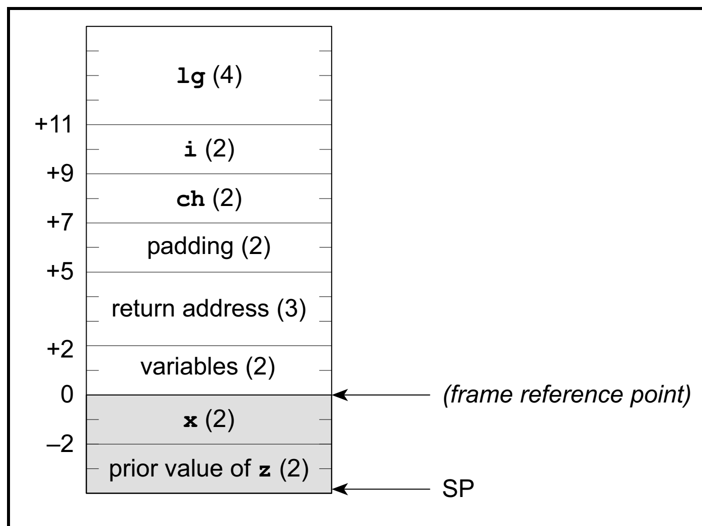


Figure 7. Assembly Language Stack Frame
No IX, Function in Extended Memory

Because of the additional 4 bytes for the return address, the arguments will have slightly different offsets.

lg	offset = +10
i	offset = +8
ch	offset = +6

Because the compiler maintains the offsets automatically, there is no need to worry about the change of offsets. The **@SP** approach discussed previously as a means of accessing stack-based variables works whether a function is compiled to extended memory or not, as long as the C-language names of local variables and arguments are used.

A function compiled to extended memory can use IX as a frame reference pointer as well. This adds an additional two bytes to argument offsets because of the saved IX value. Again, the IX+offset approach discussed previously.can be used because the compiler maintains the offsets automatically.

12.11 C Functions Calling Assembly Code

Dynamic C does not assume that registers are preserved in function calls. In other words, the function being called need not save and restore registers. 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.

In contrast, if a function returns a structure (of any size), the calling function reserves space on the stack for the return value before pushing the last argument (if any). A C function containing embedded assembly code may use a C **return** statement to return a value. A stand-alone assembly routine, however, must store the return value in the structure return space on the stack before returning.

An inline assembly code may access the stack area reserved for structure return values by the symbol **@RETVAL**, which is an offset from the frame reference point.

The following code shows how to clear field **f1** of a structure (as a returned value) of type **struct s**.

```
typedef struct ss {
    int  f0;                    // first field
    char f1;                    // second field
} xyz;
xyz my_struct;
    ...
my_struct = func();
    ...
xyz func(){
#asm
    ...
    xor a                         ; clear register A.
    ld  hl,@SP+@RETVAL+ss+f1      ; hl ← the offset from
                                  ; SP to the f1 field of
                                  ; the returned structure.
    add hl,sp                     ; hl now points to f1.
    ld (hl),a                     ; load a (now 0) to f1.
    ...
#endasm
}
```

It is crucial that **@SP** be added to **@RETVAL** because **@RETVAL** is an offset from the frame reference point, not from the current SP.

12.12 Assembly Code Calling C Functions

A program may call a C function from assembly code. To make this happen, set up part of the stack frame prior to the call and “unwind” the stack after the call. The procedure to set up the stack frame is described here.

1. Save all registers that the calling function wants to preserve. A called C function may change the value of any register. (Pushing registers values on the stack is a good way to save their values.)
2. If the function return is a **struct**, reserve space on the stack for the returned structure. Most functions do not return structures.
3. Compute and push the last argument, if any.
4. Compute and push the second to last argument, if any.
5. Continue to push arguments, if there are more.
6. Compute and push the first argument, if any. Also load the first argument into the primary register (HL for **int, 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.

12.13 Interrupt Routines in Assembly

Dynamic C allows interrupt service routines to be written in C (declared with the keyword **interrupt**). However, the efficiency of one interrupt routine affects the latency of other interrupt routines. Assembly routines can be more efficient than the equivalent C functions, and therefore more suitable for interrupt service routines.

Either stand-alone assembly code or embedded assembly code may be used for interrupt routines. The benefit of embedding assembly code in a C-language interrupt routine is that there is no need to worry about saving and restoring registers or reenabling interrupts. The drawback is that the C interrupt function does save all registers, which takes some amount of time. A stand-alone assembly routine needs to save and restore only the registers it uses.

In general, an interrupt routine performs the following actions.

1. Turn off interrupts upon entry.
2. Save all registers (that will be used) on the stack. Interrupt routines written in C save all registers on the stack automatically. Stand-alone assembly routines must push the registers explicitly.
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. Reenable interrupts. Interrupts are disabled for the entire duration of the interrupt routine (unless they are enabled explicitly). The interrupt handler must reenable the interrupt so that other interrupts can get the attention of the CPU. Interrupt routines written in C reenable interrupts automatically when the function returns. Stand-alone assembly interrupt routines, however, must reenable the interrupt (ipres) explicitly.
   The interrupts should be reenabled immediately before the return instructions **`ret`** or **`reti`**. If the interrupts are enabled earlier, the system can stack up the interrupts. This may or may not be acceptable because there is the potential to overflow the stack.
8. Return. There are three types of interrupt returns: **`ret, reti`**, and **`retn`**.

12.14 Common Problems

Unbalanced stack. Ensure the stack is "balanced" when a routine returns. In other words, the SP must be same on exit as it was on entry. From the caller's point of view, the SP register must be identical before and after the call instruction.

Using the `@SP` approach after pushing temporary information on the stack. The **`@SP`** approach for inline assembly code assumes that SP points to the low boundary of the stack frame. This might not be the case if the routine pushes temporary information onto the stack. The space taken by temporary information on the stack must be compensated for.

The following code illustrates the concept.

```
;SP still points to the low boundary of the call frame
    push hl         ; save HL
;SP now two bytes below the stack frame!
...
    ld hl,@SP+x+2   ; Add 2 to compensate for altered SP
    add hl,sp       ; compute as normal
    ld a,(hl)       ; get the content
...
    pop hl          ; restore HL
;SP again points to the low boundary of the call frame
```

Registers not preserved. In Dynamic C, the caller is responsible for saving and restoring all registers. An assembly routine that calls a C function must assume that all registers will be changed.

Unpreserved registers in interrupt routines cause unpredictable and unrepeatable problems. In contrast to normal functions, interrupt functions are responsible for saving and restoring all registers themselves.

Keywords 13

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

abort

Jumps out of a costatement.

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

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
```

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;
   ...
}
```

break

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

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

case

Identifies the next "case" in a **switch** statement.

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

char

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

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

`const`

This keyword announces that a variable will not have its value changed and that static and initialized global variable will be placed in flash memory. The keyword **`const`** is a type qualifier and may be used with any static or global type specifier (char, int, struct, etc.). The **`const`** qualifier appears before the type unless it is modifying a pointer. When modifying a pointer, the **`const`** keyword appears after the '**`*`**'.

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

Example 1:

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

Example 2:

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

Example 3:

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

Example 4:

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

Example 5:

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

continue

Skip to the next iteration of a loop.

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

costate

Indicates the beginning of a costatement.

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

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

debug

Indicates a function is to be compiled in debug mode.

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

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

default

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

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

do

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

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

The statement must have a semicolon at the end.

else

Indicates a false branch of an **if** statement

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

extern

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

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

firsttime

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

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

for

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

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

goto

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

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

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

```
int i, j, *k;               // 16-bit signed
unsigned int x;             // 16-bit unsigned
long int z;                 // 32-bit signed
unsigned long int w;        // 32-bit unsigned
int funct ( int arg ){
    ...
}
```

interrupt

Indicates that a function is an interrupt service routine. All registers, including alternates, are saved when an interrupt function is called and restored when the interrupt function returns. Writing ISRs in C is not recommended when timing is critical.

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

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

long

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

```
long i, j, *k;              // 32-bit signed
unsigned long int w;        // 32-bit unsigned
long funct ( long arg ){
    ...
}
```

main

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

nodebug

Indicates a function is not compiled in debug mode.

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

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

norst

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

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

nouseix

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

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

NULL

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

protected

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

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

Declaring a multibyte variable *shared* means that changes to the variable are atomic, i.e., interrupts are disabled while the variable is being changed.

Declaring a variable to be "protected" guards against system failure. This means that a copy of the variable is made before it is modified. If a transient effect such as power failure occurs when the variable is being changed, the system will restore the variable from the copy.

```
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 timeout, then any protected variables that were being written when the reset occurred are restored.

return

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

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

root

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

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

segchain

Identifies a function chain segment (within a function).

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

This example adds a segment to the function chain **`_GLOBAL_INIT`**. Using **`segchain`** is equivalent to using the **`#GLOBAL_INIT`** directive. When this function chain executes, this and perhaps other segments elsewhere execute. The effect in this example is to (re)initialize **`vec`**.

shared

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

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

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

short

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

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

size

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

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

sizeof

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

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

speed

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

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

static

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

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

struct

Indicates the beginning of a structure definition. Structure definitions can be nested.

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

switch

Indicates the start of a **switch** statement.

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

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

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

typedef

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

```
typedef struct {
     int x;
     int y;
} xyz;                     // defines a struct type...
xyz thing;                 // ...and a thing of type xyz
typedef uint node;         // meaningful type name
node master, slave1, slave2;
```

union

Identifies a variable that can contain objects of different types and sizes at different times. Items in a **union** have the same address. The size of a **union** is that of its largest member.

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

unsigned

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

```
unsigned i, j, *k;                  // 16-bit, unsigned
unsigned int x;                     // 16-bit, unsigned
unsigned long w;                    // 32-bit, unsigned
unsigned funct ( unsigned arg ){
     ...
}
```

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

useix

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

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

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

waitfor

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

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

waitfordone (wfd)

The **waitfordone** keyword can be abbreviated as **wfd**. It is part of Dynamic C's cooperative multitasking constructs. Used inside a costatement or a cofunction, it executes cofunctions and **firsttime** functions. When all the cofunctions and **firsttime** functions in the **wfd** statement are complete, or one of them aborts, execution proceeds to the statement following **wfd**. Otherwise a jump is made to the ending brace of the costatement or cofunction where the **wfd** statement appears; when the execution thread comes around again, control is given back to the **wfd** statement.

This keyword may return an argument.

while

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

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

xdata

Declares a block of data in extended memory.

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

The value list may include constant expressions of type **int**, **float**, **unsigned int**, **long**, **unsigned long**, **char**, and (quoted) strings.

The 20-bit physical address of the block is assigned to **name** by the compiler.

xmem

Indicates that a function is to be placed in extended memory. This keyword is semantically meaningful in function prototypes. Its use must be consistent between the prototype and the function definition.

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

xstring

Declares a table of strings in extended memory. 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;
      ...
   }
   ...
}
```

13.1 Compiler Directives

Directives are special keywords prefixed with the symbol **#**. They tell the compiler how to proceed. Only one directive per line is allowed, but a directive may span more than one line if a backslash (\) is placed at the end of the line(s).

#asm *options*
#endasm

Begins and ends blocks of assembly code. The following options are available.

nodebugdisables debug code during assembly

debugenables debug code during assembly

#class *options*

Controls the storage class for local variables. The available options are:

auto - local variables are placed on the stack.
static - local variables have permanent, fixed storage. This is the default storage class.

#debug
#nodebug

Enables or disables **debug** code compilation.

#define *name text*
#define *name***(** *params...* **)** *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.

#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 { *variables* **}**

Only way to initialize global variables in a function. For example:

```
#GLOBAL_INIT{ lk_ticks=0; lk_fc_block=0;}
```

`#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` *`chainname name`*

Adds a function, or another function chain, to a function chain.

`#if` *`constant_expression`*
`#elif` *`constant_expression`*
`#else`
`#endif`

These directives control conditional compilation. Combined, they 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 *constant_expressions* are false.

`#ifdef` *`name`*
`#ifndef` *`name`*

Similar to the **`#if`** above, these directives enable and disable code compilation based on whether or not *name* has been defined with a **`#define`** directive.

`#interleave`
`#nointerleave`

Controls whether Dynamic C will intersperse library functions with the program's functions during compilation. **`#nointerleave`** forces the user-written functions to be compiled first.

#KILL *name*

To redefine a symbol found in the BIOS of a controller, first **KILL** the prior *name*.

#makechain *chainname*

Creates a function chain. When a program executes the function chain named in this directive, all of the functions or segments belonging to that chain execute.

#memmap *options*

Controls the default memory area for functions. The following options are available.

anymem NNNNwhen code comes within NNNN bytes of the end of root code space, start putting it in xmem. Default memory usage is **#memmap anymem 0x2000**.

rootall functions not declared as **xmem** go to root memory

xmemall functions not declared as **root** go to extended memory

#undef *name*

Removes (undefines) a defined macro.

#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.

#warns "..."

Instructs the compiler to act as if a serious warning (**#warns**) was issued. The string in quotes following the directive is the message to be printed.

#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 <*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. The filename is assumed to be either relative to the Dynamic C installation directory or a fully qualified path. The symbol is a compiler 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.

Operators 14

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

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

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

```
-amount             // single operand with unary “minus”
```

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

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

For example, given the expression

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

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

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

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

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

uses parentheses to circumvent the normal order of evaluation.

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

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

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

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

```
float x;
*(char*)&x = 0x17;             // low byte of x gets value
```

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

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

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

The operators are summarized in the following pages.

14.1 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 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 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
```

/

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
```

14.2 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
```

14.3 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
```

14.4 Relational Operators

<

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

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

<=

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

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

>

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

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

>=

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

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

14.5 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 ≠ the right operand, and 0 otherwise.

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

14.6 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 ){
    ...
}
```

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

14.8 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 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
```

14.9 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.

14.10 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" };
                               // number of bytes in array
#define array_size sizeof(record)*100
a = sizeof(record);               // 7
b = array_size;                   // 700
c = sizeof(cc);                   // 20
d = sizeof(list);                 // 6
```

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

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

,

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 4. Operator Precedence

<table>
<tr><th>Operators</th><th>Associativity</th><th>Function</th></tr>
<tr><td><code>() [] -> .</code></td><td>left to right</td><td>member</td></tr>
<tr><td><code>! ~ ++ --</code>
<code>(</code>type<code>) * & sizeof</code></td><td>right to left</td><td>unary</td></tr>
<tr><td><code>* / %</code></td><td>left to right</td><td>multiplicative</td></tr>
<tr><td><code>+ -</code></td><td>left to right</td><td>additive</td></tr>
<tr><td><code><< >></code></td><td>left to right</td><td>bitwise</td></tr>
<tr><td><code>< <= > >=</code></td><td>left to right</td><td>relational</td></tr>
<tr><td><code>== !=</code></td><td>left to right</td><td>equality</td></tr>
<tr><td><code>&</code></td><td>left to right</td><td>bitwise</td></tr>
<tr><td><code>^</code></td><td>left to right</td><td>bitwise</td></tr>
<tr><td><code>|</code></td><td>left to right</td><td>bitwise</td></tr>
<tr><td><code>&&</code></td><td>left to right</td><td>logical</td></tr>
<tr><td><code>||</code></td><td>left to right</td><td>logical</td></tr>
<tr><td><code>? :</code></td><td>right to left</td><td>conditional</td></tr>
<tr><td><code>= *= /= %= += -=</code>
<code><<= >>= &= ^= |=</code></td><td>right to left</td><td>assignment</td></tr>
<tr><td><code>,</code> (comma)</td><td>left to right</td><td>series</td></tr>
</table>

Function Reference 15

15.1 Functional Groups

arithmetic	`abs` `getcrc`

bit manipulation	`bit`
	`BIT`
	`res`
	`RES`
	`set`
	`SET`

character	`isalnum`
	`isalpha`
	`iscntrl`
	`isdigit`
	`isgraph`
	`islower`
	`isprint`
	`ispunct`
	`isspace`
	`isupper`
	`isxdigit`

extended memory	`root2xmem`
	`WriteFlash2`
	`xalloc`
	`xmem2root`
	`xmem2xmem`

fast fourier transforms	`fftcplx`
	`fftcplxinv`
	`fftreal`
	`fftrealinv`
	`hanncplx`
	`hannreal`
	`powerspectrum`

file system	`fclose`
	`fcreate`
	`fcreate_unused`
	`fdelete`
	`fopen_rd`
	`fopen_wr`
	`fread`
	`fs_format`
	`fs_init`
	`fs_reserve_blocks`
	`fsck`
	`fseek`
	`fshift`
	`ftell`
	`fwrite`

floating-point math	`acos`
	`acot`
	`acsc`
	`asec`
	`asin`
	`atan`
	`atan2`
	`ceil`
	`cos`
	`cosh`
	`deg`
	`exp`
	`fabs`
	`floor`
	`fmod`
	`frexp`
	`labs`
	`ldexp`
	`log`
	`log10`
	`modf`
	`poly`
	`pow`
	`pow10`
	`rad`
	`rand`
	`randb`
	`randg`
	`sin`
	`sinh`
	`sqrt`
	`tan`
	`tanh`

low-level flash access	`flash_erasechip`
	`flash_erasesector`
	`flash_gettype`
	`flash_init`
	`flash_read`
	`flash_readsector`
	`flash_sector2xwindow`
	`flash_writesector`

I/O	`BitRdPortE`
	`BitRdPortI`
	`BitWrPortE`
	`BitWrPortI`
	`RdPortE`
	`RdPortI`
	`WrPortE`
	`WrPortI`

interrupts	`GetVectExtern2000`
	`GetVectIntern`
	`SetVectExtern2000`
	`SetVectIntern`

MicroC/OS-II	`OSInit`
	`OSMboxAccept`
	`OSMboxCreate`
	`OSMboxPend`
	`OSMboxPost`
	`OSMboxQuery`
	`OSMemCreate`
	`OSMemGet`
	`OSMemPut`
	`OSMemQuery`
	`OSQAccept`
	`OSQCreate`
	`OSQFlush`
	`OSQPend`
	`OSQPost`
	`OSQPostFront`
	`OSQQuery`
	`OSSchedLock`
	`OSSchedUnlock`
	`OSSemAccept`
	`OSSemCreate`
	`OSSemPend`
	`OSSemPost`
	`OSSemQuery`
	`OSSetTickPerSec`
	`OSStart`
	`OSStatInit`
	`OSTaskChangePrio`
	`OSTaskCreate`
	`OSTaskCreateExt`
	`OSTaskCreateHook`
	`OSTaskDel`
	`OSTaskDelHook`
	`OSTaskDelReq`
	`OSTaskQuery`
	`OSTaskResume`
	`OSTaskStatHook`
	`OSTaskStkChk`
	`OSTaskSuspend`
	`OSTaskSwHook`
	`OSTimeDly`
	`OSTimeDlyHMSM`
	`OSTimeDlyResume`
	`OSTimeDlySec`
	`OSTimeGet`
	`OSTimeSet`
	`OSTimeTickHook`
	`OSVersion`

miscellaneous	`longjmp`
	`qsort`
	`runwatch`
	`setjmp`

multitasking	`CoBegin`
	`CoPause`
	`CoReset`
	`CoResume`
	`DelayMs`
	`DelaySec`
	`DelayTicks`
	`IntervalMs`
	`IntervalSec`
	`IntervalTick`
	`isCoDone`
	`isCoRunning`

number-to-string conversion	`ftoa`
	`htoa`
	`itoa`
	`ltoa`
	`ltoan`
	`utoa`

real-time clock	`mktime`
	`mktm`
	`read_rtc`
	`read_rtc_32kHz`
	`tm_rd`
	`tm_wr`
	`write_rtc`

serial communication
(interrupt driven functions)
	`cof_serXgetc`
	`cof_serXgets`
	`cof_serXputc`
	`cof_serXputs`
	`cof_serXread`
	`cof_serXwrite`
	`serCheckParity`
	`serXclose`
	`serXdatabits`
	`serXflowcontrolOff`
	`serXflowcontrolOn`
	`serXgetc`
	`serXgetError`
	`serXopen`
	`serXparity`
	`serXpeek`
	`serXputc`
	`serXputs`
	`serXrdFlush`
	`serXrdFree`
	`serXrdUsed`
	`serXread`
	`serXwrFlush`
	`serXwrFree`
	`serXwrite`

STDIO	`getchar`
	`gets`
	`kbhit`
	`outchrs`
	`outstr`
	`printf`
	`putchar`
	`puts`
	`sprintf`

string manipulation	`memchr`
	`memcmp`
	`memcpy`
	`memmove`
	`memset`
	`strcat`
	`strchr`
	`strcmp`
	`strcmpi`
	`strcpy`
	`strcspn`
	`strlen`
	`strncat`
	`strncmp`
	`strncmpi`
	`strncpy`
	`strpbrk`
	`strrchr`
	`strspn`
	`strstr`
	`strtok`
	`tolower`
	`toupper`

string-to-number conversion	`atoi`
	`atol`
	`strtod`
	`strtol`

system	`chkHardReset`
	`chkSoftReset`
	`chkWDTO`
	`clockDoublerOff`
	`clockDoublerOn`
	`defineErrorHandler`
	`exit`
	`forceSoftReset`
	`ipres`
	`ipset`
	`premain`
	`_sysIsSoftReset`
	`sysResetChain`
	`updateTimers`
	`use32HzOsc`
	`useClockDivider`
	`useMainOsc`

watchdog	Disable_HW_WDT hitwd VdGetFreeWd VdHitWd VdInit VdReleaseWd

15.2 Alphabetical Listing

abs

```
int abs(int x);
```

DESCRIPTION

Computes the absolute value of an integer argument.

PARAMETERS

x	Integer argument

RETURN VALUE

Absolute value of the argument.

LIBRARY

`MATH.LIB`

SEE ALSO

`fabs`

acos

```
float acos(float x);
```

DESCRIPTION

Computes the arccosine of real **`float`** value **`x`**.

PARAMETERS

x	Assumed to be between -1 and 1.

RETURN VALUE

Arccosine of the argument
If **`x`** is out of bounds, the function returns **`0`** and signals a domain error.

LIBRARY

`MATH.LIB`

SEE ALSO

`cos, cosh, asin, atan`

acot

```
float acot(float x);
```

DESCRIPTION

Computes the arcotangent of real **float** value **x**.

PARAMETERS

x	Assumed to be between -INF and +INF.

RETURN VALUE

Arccotangent of the argument.

LIBRARY

MATH.LIB

SEE ALSO

tan, atan

acsc

```
float acsc(float x);
```

DESCRIPTION

Computes the arccosecant of real **float** value **x**.

PARAMETERS

x	Assumed to be between -INF and +INF.

RETURN VALUE

The arccosecant of the argument.

LIBRARY

MATH.LIB

SEE ALSO

sin, asin

asec

```
float asec(float x);
```

DESCRIPTION

Computes the arcsecant of real **float** value **x**.

PARAMETERS

x	Assumed to be between -INF and +INF.

RETURN VALUE

The arcsecant of the argument.

LIBRARY

MATH.LIB

SEE ALSO

cos, acos

asin

```
float asin(float x);
```

DESCRIPTION

Computes the arcsine of real **float** value **x**.

PARAMETERS

x	Assumed to be between -1 and +1.

RETURN VALUE

The arcsine of the argument.

LIBRARY

MATH.LIB

SEE ALSO

sin, acsc

atan

`float atan(float x);`

DESCRIPTION

Computes the arctangent of real **`float`** value **`x`**.

PARAMETERS

`x`	Assumed to be between -INF and +INF.

RETURN VALUE

The arctangent of the argument.

LIBRARY

MATH.LIB

SEE ALSO

tan, acot

atan2

```
float atan2(float y, float x);
```

DESCRIPTION

Computes the arctangent of real **`float`** value **`y/x`** to find the angle in radians between the x-axis and the ray through (0,0) and (x,y).

PARAMETERS

`y`	The point corresponding to the y-axis
`x`	The point corresponding to the x-axis

RETURN VALUE

Arctangent of **`y/x`**.

If both **`y`** and **`x`** are zero, the function returns **`0`** and signals a domain error. Otherwise the result is returned as follows:

angle	$x \neq 0$, $y \neq 0$
PI/2	$x = 0$, $y > 0$
–PI/2	$x = 0$, $y < 0$
0	$x > 0$, $y = 0$
PI	$x < 0$, $y = 0$

LIBRARY

`MATH.LIB`

SEE ALSO

`acos, asin, atan, cos, sin, tan`

atof

```
float atof(char *sptr);
```

DESCRIPTION

ANSI String to Float Conversion (UNIX compatible)

PARAMETERS

sptr	String to convert.

RETURN VALUE

The converted floating value.
If the conversion is invalid, **_xtoxErr** is set to **1**. Otherwise **_xtoxErr** is set to **0**.

LIBRARY

STRING.LIB

SEE ALSO

atoi, atol, strtod

atoi

```
int atoi(char *sptr);
```

DESCRIPTION

ANSI String to Integer Conversion (UNIX compatible).

PARAMETERS

sptr	String to convert.

RETURN VALUE

The converted integer value.

LIBRARY

STRING.LIB

SEE ALSO

atol, atof, strtod

atol

`long atol(char *sptr);`

DESCRIPTION

ANSI String to Long Conversion (UNIX compatible).

PARAMETERS

`sptr`	String to convert.

RETURN VALUE

The converted long integer value.

LIBRARY

`STRING.LIB`

SEE ALSO

`atoi, atof, strtod`

bit

`unsigned int bit(void *address, unsigned int bit);`

DESCRIPTION

Dynamic C may expand this call inline

Reads specified bit at memory address. **`bit`** may be from 0 to 31. This is equivalent to the following expression, but more efficient: **`(*(long *)address >> bit) & 1`**

PARAMETERS

`address`	Address of byte containing bits 7-0
`bit`	Bit location where 0 represents the least significant bit

RETURN VALUE

`1` if specified bit is set,
`0` if bit is clear.

LIBRARY

`UTIL.LIB`

SEE ALSO

`BIT`

BIT

`unsigned int BIT(void *address, unsigned int bit);`

DESCRIPTION

Dynamic C may expand this call inline

Reads specified bit at memory address. **`bit`** may be from 0 to 31. This is equivalent to the following expression, but more efficient: **`(*(long *)address>>bit) &1`**

PARAMETERS

`address`	Address of byte containing bits 7-0
`bit`	Bit location where 0 represents the least significant bit

RETURN VALUE

`1` if specified bit is set; **`0`** if bit is clear.

LIBRARY

`UTIL.LIB`

SEE ALSO

`bit`

BitRdPortE

`int BitRdportE(int port, int bitnumber);`

DESCRIPTION

Returns 1 or 0 matching the value of the bit read from the specified external I/O port.

PARAMETERS

`port`	Address of external parallel port data register.
`bitnumber`	Bit to read (0–7).

RETURN VALUE

Returns an integer equal to 1 or 0 matching the value of the bit read.

LIBRARY

`SYSIO.LIB`

SEE ALSO

`RdPortI, BitRdPortI, WrPortI, BitWrPortI, RdPortE, WrPortE, BitWrPortE`

BitRdPortI

`int BitRdI(int port, int bitnumber);`

DESCRIPTION

Returns 1 or 0 matching the value of the bit read from the specified internal I/O port.

PARAMETERS

`port`	Address of internal parallel port data register.
`bitnumber`	Bit to read (0–7).

RETURN VALUE

Returns an integer equal to 1 or 0 matching the value of the bit read.

LIBRARY

`SYSIO.LIB`

SEE ALSO

`RdPortI, WrPortI, BitWrPortI, BitRdPortE, RdPortE, WrPortE, BitWrPortE`

BitWrPortE

```
void BitWrPortE( int port, char *portshadow, int value, int
   bitcode);
```

DESCRIPTION

Updates shadow register at bit with value (0 or 1) and copies shadow to register.

WARNING! A shadow register is required for this function.

PARAMETERS

port	Address of external parallel port data register.
portshadow	Reference pointer to a variable to shadow the current value of the register.
value	Value of 0 or 1 to be written to the bit position.
bitcode	Bit position 0–7.

LIBRARY

SYSIO.LIB

SEE ALSO

RdPortI, BitRdPortI, WrPortI, BitWrPortI, BitRdPortE, RdPortE, WrPortE

BitWrPortI

```
void BitWrPortI( int port, char *portshadow, int value, int
   bitcode);
```

DESCRIPTION

Updates shadow register at position **bitcode** with value (0 or 1); copies shadow to register.

WARNING! A shadow register is required for this function.

PARAMETERS

port	Address of external parallel port data register.
portshadow	Reference pointer to a variable to shadow the current value of the register.
value	Value of 0 or 1 to be written to the bit position.
bitcode	Bit position 0–7.

LIBRARY

SYSIO.LIB

SEE ALSO

RdPortI, BitRdPortI, WrPortI, BitRdPortE, RdPortE, WrPortE, BitWrPortE

ceil

```
float ceil(float x);
```

DESCRIPTION

Computes the smallest integer greater than or equal to the given number.

PARAMETERS

x	Number to round up.

RETURN VALUE

The rounded up number.

LIBRARY

`MATH.LIB`

SEE ALSO

`floor, fmod`

chkHardReset

```
int chkHardReset( void );
```

DESCRIPTION

This function determines whether this restart of the board is due to a hardware reset. Asserting the RESET line or recycling power are both considered hardware resets. A watchdog timeout is not a hardware reset.

RETURN VALUE

`1`: The processor was restarted due to a hardware reset,
`0`: If it was not.

LIBRARY

`Sys.lib`

chkSoftReset

```
int chkSoftReset( void );
```

DESCRIPTION

This function determines whether this restart of the board is due to a software reset from Dynamic C or a call to **`forceSoftReset()`**.

RETURN VALUE

`1`: The board was restarted due to a soft reset,
`0`: If it was not.

LIBRARY

`Sys.lib`

chkWDTO

```
int chkWDTO( void );
```

DESCRIPTION

This function determines whether this restart of the board is due to a watchdog timeout.

RETURN VALUE

`1`: If the board was restarted due to a watchdog timeout,
`0`: If it was not.

LIBRARY

`Sys.lib`

clockDoublerOn

```
void clockDoublerOn();
```

DESCRIPTION

Enables the Rabbit clock doubler. If the doubler is already enabled, there will be no effect. Also attempts to adjust the communication rate between Dynamic C and the board to compensate for the frequency change. User serial port rates need to be adjusted accordingly. Also note that single-stepping through this routine will cause Dynamic C to lose communication with the target.

LIBRARY

`SYS.LIB`

SEE ALSO

`clockDoublerOff`

clockDoublerOff

```
void clockDoublerOff();
```

DESCRIPTION

Disables the Rabbit clock doubler. If the doubler is already disabled, there will be no effect. Also attempts to adjust the communication rate between Dynamic C and the board to compensate for the frequency change. User serial port rates need to be adjusted accordingly. Also note that single-stepping through this routine will cause Dynamic C to lose communication with the target.

LIBRARY

`SYS.LIB`

SEE ALSO

`clockDoublerOn`

CoBegin

void CoBegin(CoData *p);

DESCRIPTION

Initialize a costatement structure so the costatement will be executed next time it is encountered.

PARAMETERS

p	Address of costatement

LIBRARY

COSTATE.LIB

cof_serXgetc

int cof_serXgetc(); /* where X = A|B|C|D */

DESCRIPTION

This single-user cofunction yields to other tasks until a character is read from port X. This function only returns when a character is successfully written. It is non-reentrant.

RETURN VALUE

An integer with the character read into the low byte

LIBRARY

RS232.LIB

EXAMPLE

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

cof_serXgets

```
int cof_serXgets(char *s, int max, unsigned long tmout);
   /* where X = A|B|C|D */
```

DESCRIPTION

This single-user cofunction reads characters from port X until a **NULL** terminator, line-feed, or carriage return character is read, **max** characters are read, or until **tmout** milliseconds transpires between characters read. A timeout will never occur if no characters have been received. This function is non-reentrant.

It yields to other tasks for as long as the input buffer is locked or whenever the buffer becomes empty as characters are read. **s** will always be **NULL** terminated upon return.

PARAMETERS

s	Character array into which a **NULL** terminated string is read.
max	The maximum number of characters to read into s.
tmout	Millisecond wait period to allow between characters before timing out.

RETURN VALUE

`1` if CR or **max** bytes read into **s**
`0` if function times out before reading CR or **max** bytes

LIBRARY

`RS232.LIB`

EXAMPLE

```
// echoes NULL terminated character strings
main() {
    int getOk;
    char s[16];
    serAopen(19200);
    loopinit();
    while (1) {
        loophead();
        costate {
            wfd getOk = cof_serAgets (s, 15, 20);
            if (getOk) {
                wfd cof_serAputs(s);
            }
            else {          // timed out: s null terminated,
            }               // but incomplete
        }
    }
    serAclose();
}
```

cof_serXputc

void cof_serXputc(int c); /* where X = A|B|C|D */

DESCRIPTION

This single-user cofunction writes a character to serial port X, yielding to other tasks when the input buffer is locked. This function is non-reentrant.

PARAMETERS

c	Character to write.

LIBRARY

RS232.LIB

EXAMPLE

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

cof_serXputs

void cof_serXputs(char *str); /* where X = A|B|C|D */

DESCRIPTION

This single-user cofunction writes a **NULL** terminated string to port X. It yields to other tasks for as long as the input buffer may be locked or whenever the buffer may become full as characters are written. This function is non-reentrant.

PARAMETERS

str	**NULL**-terminated character string to write.

LIBRARY

RS232.LIB

EXAMPLE

```
// writes a null-terminated character string, repeatedly
main() {
    const char s[] = "Hello Z-World";
    serAopen(19200);
    loopinit();
    while (1) {
        loophead();
        costate {
            wfd cof_serAputs(s);
        }
    }
    serAclose();
}
```

cof_serXread

int cof_serXread(void* data, int length, unsigned long tmout);
/* where X = A|B|C|D */

DESCRIPTION

This single-user cofunction reads **length** characters from port X or until **tmout** milli-seconds transpires between characters read. It yields to other tasks for as long as the input buffer is locked or whenever the buffer becomes empty as characters are read. A timeout will never occur if no characters have been read. This function is non-reentrant.

PARAMETERS

data	Data structure into which characters are read.
length	The number of characters to read into **data**.
tmout	Millisecond wait period to allow between characters before timing out.

RETURN VALUE

Number of characters read into **data**.

LIBRARY

RS232.LIB

EXAMPLE

```
// echoes a block of characters
main() {
     int n;
     char s[16];
     serAopen(19200);
     loopinit();
     while (1) {
          loophead();
          costate {
               wfd n = cof_serAread(s, 15, 20);
               wfd cof_serAwrite(s, n);
          }
     }
     serAclose();
}
```

cof_serXwrite

```
void cof_serXwrite(void *data, int length);
    /* where X = A|B|C|D */
```

DESCRIPTION

This single-user cofunction writes **length** bytes to port X. It yields to other tasks for as long as the input buffer is locked or whenever the buffer becomes full as characters are written. This function is non-reentrant.

PARAMETERS

data	Data structure to write.
length	Number of bytes in **data** to write.

LIBRARY

RS232.LIB

EXAMPLE

```
// writes a block of characters, repeatedly
main() {
    const char s[] = "Hello Z-World";
    serAopen(19200);
    loopinit();
    while (1) {
        loophead();
        costate {
            wfd cof_serAwrite(s, strlen(s));
        }
    }
    serAclose();
}
```

CoPause

```
void CoPause(CoData *p);
```

DESCRIPTION

Pause execution of a costatement so that it will not run the next time it is encountered unless and until **CoResume(p)** or **CoBegin(p)** are called.

PARAMETERS

p	Address of costatement

LIBRARY

COSTATE.LIB

CoReset

```
void CoReset(CoData *p);
```

DESCRIPTION

Initializes a costatement structure so the costatement will not be executed next time it is encountered (unless the costatement is declared to be **always_on**).

PARAMETERS

p	Address of costatement

LIBRARY

COSTATE.LIB

CoResume

```
void CoResume(CoData *p);
```

DESCRIPTION

Resume execution of a costatement that has been paused.

PARAMETERS

p	Address of costatement

LIBRARY

COSTATE.LIB

cos

```
float cos(float x);
```

DESCRIPTION

Computes the cosine of real float value **x** (radians).

PARAMETERS

x	Radian value to compute

RETURN VALUE

Cosine of the argument.

LIBRARY

MATH.LIB

SEE ALSO

acos, cosh, sin, tan

cosh

`float cosh(float x);`

DESCRIPTION

Computes the hyperbolic cosine of real FLOAT value **`x`**.

PARAMETERS

`x`	value to compute

RETURN VALUE

Hyperbolic cosine

If |**x**| > 89.8 (approx.), the function returns INF and signals a range error.

LIBRARY

`MATH.LIB`

SEE ALSO

`cos, acos, sin, sinh, tan, tanh`

defineErrorHandler

`void defineErrorHandler(void *errfcn)`

DESCRIPTION

Sets the BIOS function pointer for runtime errors to the function pointed to by **`errfcn`**. When a runtime error occurs, the following information is passed to the error handler on the stack:

SP+0 - `return address for` **`exceptionRet`**
SP+2 - `Error code`
SP+4 - 0x0000 `(can be used for additional information)`
SP+6 - XPC `when` exception() `was called (upper byte)`
SP+8 - `address where` exception() `was called`

The user-defined function should ALWAYS be in root memory. Specify **`root`** at the start of the function definition to ensure this.

PARAMETERS

`errfcn`	Pointer to user-defined runtime error handler.

LIBRARY

`SYS.LIB`

deg

```
float deg(float x);
```

DESCRIPTION

Changes **float** radians **x** to degrees

PARAMETERS

x	Radian value to convert

RETURN VALUE

Angle in degrees (a **float**).

LIBRARY

MATH.LIB

SEE ALSO

rad

DelayMs

```
int DelayMs(long delayms);
```

DESCRIPTION

Millisecond time mechanism for the costatement "waitfor" constructs. The initial call to this function starts the timing. The function returns zero and continues to return zero until the number of milliseconds specified has passed.

PARAMETERS

delayms	The number of milliseconds to wait.

RETURN VALUE

1 if the specified number of milliseconds have elapsed; else **0**.

LIBRARY

COSTATE.LIB

DelaySec

`int DelaySec(long delaysec);`

DESCRIPTION

Second time mechanism for the costatement "waitfor" constructs. The initial call to this function starts the timing. The function returns zero and continues to return zero until the number of seconds specified has passed.

PARAMETERS

`delaysec`	The number of seconds to wait.

RETURN VALUE

`1` if the specified number of seconds have elapsed; else **`0`**.

LIBRARY

`COSTATE.LIB`

DelayTicks

`int DelayTicks(unsigned ticks);`

DESCRIPTION

Tick time mechanism for the costatement "waitfor" constructs. The initial call to this function starts the timing. The function returns zero and continues to return zero until the number of ticks specified has passed.

1 tick = 1/1024 second.

PARAMETERS

`ticks`	The number of ticks to wait.

RETURN VALUE

`1` if the specified tick delay has elapsed; else **`0`**.

LIBRARY

`COSTATE.LIB`

Disable_HW_WDT

```
void Disable_HW_WDT();
```

DESCRIPTION

Disables the hardware watchdog timer on the Rabbit processor. Note that the watchdog will be enabled again just by hitting it. The watchdog is hit by the periodic interrupt, which is on by default. This function is useful for special situations such as low power "sleepy mode".

LIBRARY

SYS.LIB

exit

```
void exit(int exitcode);
```

DESCRIPTION

Stops the program and returns **exitcode** to Dynamic C. Dynamic C uses values above 128 for run-time errors. When not debugging, **exit** will run an infinite loop, causing a watchdog timeout if the watchdog is enabled.

PARAMETERS

exitcode	Error code passed by Dynamic C

LIBRARY

SYS.LIB

exp

`float exp(float x);`

DESCRIPTION

Computes the exponential of real **`float`** value **`x`**.

PARAMETERS

`x`	Value to compute

RETURN VALUE

Returns the value of e^x.

If **`x`** > 89.8 (approx.), the function returns INF and signals a range error. If **`x`** < –89.8 (approx.), the function returns 0 and signals a range error.

LIBRARY

`MATH.LIB`

SEE ALSO

`log, log10, frexp, ldexp, pow, pow10, sqrt`

fabs

`float fabs(float x);`

DESCRIPTION

Computes the float absolute value of **`float x`**.

PARAMETERS

`x`	Value to compute

RETURN VALUE

`x`, if **`x`** >= 0,
else **`-x`**.

LIBRARY

`MATH.LIB`

SEE ALSO

`abs`

fclose

```
void fclose(File* f);
```

DESCRIPTION

Closes a file.

PARAMETERS

f	The pointer to the file to close.

LIBRARY

FILESYSTEM.LIB

fcreate

```
int fcreate(File* f, FileNumber fnum);
```

DESCRIPTION

This function creates a file. Before calling it, a variable of type **File** must be defined in the application program.

```
File file;
fcreate (&file, 1);
```

PARAMETERS

f	The pointer to the created file.
fnum	This is a number from 1 through 127. Each file in the flash file system is assigned a unique number in this range that is chosen by the user.

RETURN VALUE

0 - success
1 - failure

LIBRARY

FILESYSTEM.LIB

fcreate_unused

FileNumber fcreate_unused(File* f);

DESCRIPTION

Searches for the first unused file number in the range 1 through 127, and creates a file with that number.

PARAMETERS

f	The pointer to the created file.

RETURN VALUE

The **FileNumber** (1-127) of the new file if success.

LIBRARY

FILESYSTEM.LIB

SEE ALSO

fcreate

fdelete

int fdelete(FileNumber fnum);

DESCRIPTION

Deletes a file.

PARAMETERS

fnum	A number in the range 1 through 127 that identifies the file in the flash file system.

RETURN VALUE

0 - success
1 - failure

LIBRARY

FILESYSTEM.LIB

fftcplx

void fftcplx(int *x, int N, int *blockexp)

DESCRIPTION

Computes the complex DFT of the **N**-point complex sequence contained in the array **x** and returns the complex result in *x*. **N** must be a power of 2 and lie between 4 and 1024. An invalid **N** causes a RANGE exception. The **N**-point complex sequence in array **x** is replaced with its **N**-point complex spectrum. The value of **blockexp** is increased by 1 each time array **x** has to be scaled to avoid arithmetic overflow.

PARAMETERS

x	Pointer to **N**-element array of complex fractions.
N	Number of complex elements in array **x**.
blockexp	Pointer to integer block exponent.

LIBRARY

FFT.LIB

SEE ALSO

fftcplxinv, fftreal, fftrealinv, hanncplx, hannreal, powerspectrum

fftcplxinv

```
void fftcplxinv(int *x, int N, int *blockexp)
```

DESCRIPTION

Computes the inverse complex DFT of the **N**-point complex spectrum contained in the array **x** and returns the complex result in **x**. **N** must be a power of 2 and lie between 4 and 1024. An invalid **N** causes a RANGE exception. The value of **blockexp** is increased by 1 each time array **x** has to be scaled to avoid arithmetic overflow. The value of **blockexp** is also *decreased* by $\log_2$**N** to include the 1/**N** factor in the definition of the inverse DFT

PARAMETERS

x	Pointer to **N**-element array of complex fractions.
N	Number of complex elements in array **x**.
blockexp	Pointer to integer block exponent.

LIBRARY

FFT.LIB

SEE ALSO

fftcplx, fftreal, fftrealinv, hanncplx, hannreal, powerspectrum

fftreal

void fftreal(int *x, int N, int *blockexp)

DESCRIPTION

Computes the **N**-point, positive-frequency complex spectrum of the 2**N**-point real sequence in array **x**. The 2**N**-point real sequence in array **x** is replaced with its **N**-point positive-frequency complex spectrum. The value of **blockexp** is increased by 1 each time array **x** has to be scaled to avoid arithmetic overflow.

The imaginary part of the $X[0]$ term (stored in $x[1]$) is set to the real part of the *fmax* term.

The 2**N**-point real sequence is stored in natural order. The zeroth element of the sequence is stored in **x[0]**, the first element in **x[1]**, and the kth element in $x[k]$.

N must be a power of 2 and lie between 4 and 1024. An invalid **N** causes a RANGE exception.

PARAMETERS

x	Pointer to 2**N**-point sequence of real fractions.
N	Number of complex elements in output spectrum
blockexp	Pointer to integer block exponent.

LIBRARY

FFT.LIB

SEE ALSO

fftcplx, fftcplxinv, fftrealinv, hanncplx, hannreal, powerspectrum

fftrealinv

void fftrealinv(int *x, int N, int *blockexp)

DESCRIPTION

Computes the 2**N**-point real sequence corresponding to the **N**-point, positive-frequency complex spectrum in array **x**. The **N**-point, positive-frequency spectrum contained in array **x** is replaced with its corresponding 2**N**-point real sequence. The value of blockexp is increased by 1 each time array **x** has to be scaled to avoid arithmetic overflow. The value of **blockexp** is also *decreased* by $\log_2$**N** to include the 1/**N** factor in the definition of the inverse DFT.

The function expects to find the real part of the *fmax* term in the imaginary part of the zero-frequency **X[0]** term (stored **x[1]**).

The 2**N**-point real sequence is stored in natural order. The zeroth element of the sequence is stored in **x[0]**, the first element in **x[1]**, and the *k*th element in **x[k]**.

N must be a power of 2 and lie between 4 and 1024. An invalid **N** causes a RANGE exception.

PARAMETERS

x	Pointer to **N**-element array of complex fractions.
N	Number of complex elements in array **x**.
blockexp	Pointer to integer block exponent.

LIBRARY

FFT.LIB

SEE ALSO

fftcplx, fftcplxinv, fftreal, hanncplx, hannreal, powerspectrum

flash_erasechip

`void flash_erasechip(FlashDescriptor* fd);`

DESCRIPTION

Erases an entire Flash Memory chip.

NOTE: **`fd`** must have already been initialized with **`flash_init`** before calling this function. See **`flash_init`** description for further restrictions.

PARAMETERS

`fd`	Pointer to flash descriptor of the chip to erase.

LIBRARY

`FLASH.LIB`

SEE ALSO

`flash_erasesector, flash_gettype, flash_init, flash_read, flash_readsector, flash_sector2xwindow, flash_writesector`

flash_erasesector

`int flash_erasesector(FlashDescriptor* fd, word which);`

DESCRIPTION

Erases a sector of a Flash Memory chip.

NOTE: **`fd`** must have already been initialized with **`flash_init`** before calling this function. See **`flash_init`** description for further restrictions.

PARAMETERS

`fd`	Pointer to flash descriptor of the chip to erase a sector of.
`which`	The sector to erase.

RETURN VALUE

`0` - success

LIBRARY

`FLASH.LIB`

SEE ALSO

`flash_erasechip, flash_gettype, flash_init, flash_read, flash_readsector, flash_sector2xwindow, flash_writesector`

flash_gettype

```
int flash_gettype(FlashDescriptor* fd);
```

DESCRIPTION

Returns the 16-bit Flash Memory type of the Flash Memory.

NOTE: **`fd`** must have already been initialized with **`flash_init`** before calling this function. See **`flash_init`** description for further restrictions.

PARAMETERS

`fd`	The **`FlashDescriptor`** of the memory to query.

RETURN VALUE

The integer representing the type of the Flash Memory.

LIBRARY

`FLASH.LIB`

SEE ALSO

`flash_erasechip, flash_erasesector, flash_init, flash_read, flash_readsector, flash_sector2xwindow, flash_writesector`

flash_init

```
int flash_init(FlashDescriptor* fd, int mb3cr);
```

DESCRIPTION

Initializes an internal data structure of type **FlashDescriptor** with information about the Flash Memory chip. The Memory Interface Unit bank register (MB3CR) will be assigned the value of **mb3cr** whenever a function accesses the Flash Memory referenced by **fd**. See the Rabbit 2000 Users Manual for the correct chip select and wait state settings.

NOTE: Improper use of this function can cause your program to be overwritten or operate incorrectly. This and the other Flash Memory access functions should not be used on the same Flash Memory that your program resides on, nor should they be used on the same region of a second Flash Memory where a file system resides.

Use **WriteFlash()** to write to the primary Flash Memory.

PARAMETERS

fd	This is a pointer to an internal data structure that holds information about a Flash Memory chip.
mb3cr	This is the value to set MB3CR to whenever the Flash Memory is accessed. 0xc2 (i.e., CS2, /OE0, /WE0, 0 WS) is a typical setting for the second Flash Memory on the TCP/IP Dev Kit, the Intellicom, the Advanced Ethernet Core, and the RabbitLink.

RETURN VALUE

0 on success
1 if invalid Flash Memory type
-1 for an attempt to initialize primary Flash Memory

LIBRARY

FLASH.LIB

SEE ALSO

flash_erasechip, flash_erasesector, flash_gettype, flash_read, flash_readsector, flash_sector2xwindow, flash_writesector

flash_read

```
int flash_read(FlashDescriptor* fd, word sector, word offset,
   unsigned long buffer, word length);
```

DESCRIPTION

Reads data from the Flash Memory and stores it in **buffer**.

NOTE: **fd** must have already been initialized with **flash_init** before calling this function. See the **flash_init** description for further restrictions.

PARAMETERS

fd	The **FlashDescriptor** of the Flash Memory to read from.
sector	The sector of the Flash Memory to read from.
offset	The displacement, in bytes, from the beginning of the sector to start reading at.
buffer	The physical address of the destination buffer. TIP: A logical address can be changed to a physical with the function **paddr**.
length	The number of bytes to read.

RETURN VALUE

0 on success

LIBRARY

FLASH.LIB

SEE ALSO

flash_erasechip, flash_erasesector, flash_gettype, flash_init, flash_readsector, flash_sector2xwindow, flash_writesector, paddr

flash_readsector

```
int flash_readsector(FlashDescriptor* fd, word sector, unsigned
   long buffer);
```

DESCRIPTION

Reads the contents of an entire sector of Flash Memory into a buffer.

NOTE: **fd** must have already been initialized with **flash_init** before calling this function. See **flash_init** description for further restrictions.

PARAMETERS

fd	The FlashDescriptor of the Flash Memory to read from.
sector	The source sector to read.
buffer	The physical address of the destination buffer. TIP: A logical address can be changed to a physical with the function **paddr**.

RETURN VALUE

0 on success

LIBRARY

FLASH.LIB

SEE ALSO

flash_erasechip, flash_erasesector, flash_gettype, flash_init, flash_read, flash_sector2xwindow, flash_writesector

flash_sector2xwindow

```
void* flash_sector2xwindow(FlashDescriptor* fd, word sector);
```

DESCRIPTION

This function sets the MB3CR and XPC value so the requested sector falls within the XPC window. The MB3CR is the Memory Interface Unit bank register. XPC is one of four Memory Management Unit registers. See **flash_init** description for restrictions.

PARAMETERS

fd	The FlashDescriptor of the Flash Memory.
sector	The sector to set the XPC window to.

RETURN VALUE

The logical offset of the sector.

LIBRARY

FLASH.LIB

SEE ALSO

flash_erasechip, flash_erasesector, flash_gettype, flash_init, flash_read, flash_readsector, flash_writesector

flash_writesector

```
int flash_writesector(FlashDescriptor* fd, word sector,
   unsigned long buffer);
```

DESCRIPTION

Writes the contents of **buffer** to **sector** on the Flash Memory referenced by **fd**. NOTE: **fd** must have already been initialized with **flash_init** before calling this function. See **flash_init** description for further restrictions.

PARAMETERS

fd	The FlashDescriptor of the Flash Memory to write to.
sector	The destination sector.
buffer	The physical address of the source. TIP: A logical address can be changed to a physical address with the function **paddr**

RETURN VALUE

0 on success

LIBRARY

FLASH.LIB

SEE ALSO

flash_erasechip, flash_erasesector, flash_gettype, flash_init, flash_read, flash_readsector, flash_sector2xwindow

floor

```
float floor(float x);
```

DESCRIPTION

Computes the largest integer less than or equal to the given number.

PARAMETERS

x	Value to round down

RETURN VALUE

Rounded down value

LIBRARY

MATH.LIB

SEE ALSO

ceil, fmod

fmod

```
float fmod(float x, float y);
```

DESCRIPTION

Calculates modulo math.

PARAMETERS

x	Dividend
y	Divisor

RETURN VALUE

Returns the remainder of x/y. The remaining part of **x** after all multiples of **y** have been removed. For example, if **x** is 22.7 and **y** is 10.3, the integral division result is 2. Then the remainder = $22.7 - 2 \times 10.3 = 2.1$.

LIBRARY

MATH.LIB

SEE ALSO

ceil, floor

fopen_rd

```
in fopen_rd(File* f, FileNumber fnum);
```

DESCRIPTION

Opens a file for reading.

PARAMETERS

f	A pointer to the file to read.
fnum	A number in the range 1 through 127 that identifies the file in the flash file system.

RETURN VALUE

`0` on success
`1` on failure

LIBRARY

`FILESYSTEM.LIB`

fopen_wr

```
in fopen_wr(File* f, FileNumber fnum);
```

DESCRIPTION

Opens a file for writing.

PARAMETERS

f	A pointer to the file to write.
fnum	A number in the range 1 through 127 that identifies the file in the flash file system.

RETURN VALUE

`0` on success
`1` on failure

LIBRARY

`FILESYSTEM.LIB`

forceSoftReset

```
void forceSoftReset();
```

DESCRIPTION

Forces the board into a software reset by jumping to the start of the BIOS.

LIBRARY

SYS.LIB

fread

```
int fread(File* f, char* buf, int len);
```

DESCRIPTION

Reads **len** bytes from a file pointed to by **f**, starting at the current offset into the file, into buffer. Data is read into buffer pointed to by **buf**.

PARAMETERS

f	A pointer to the file to read from
buf	A pointer to the destination buffer.
len	Number of bytes to copy.

RETURN VALUE

Number of bytes read.

LIBRARY

FILESYSTEM.LIB

frexp

```
float frexp(float x, int *n);
```

DESCRIPTION

Splits **x** into a fraction and exponent, f*(2**n)

PARAMETERS

x	Number to split
n	An integer

RETURN VALUE

The function returns the exponent in the integer ***n** and the fraction between 0.5, inclusive and 1.0.

LIBRARY

MATH.LIB

SEE ALSO

exp, ldexp

fs_format

int fs_format(long reserveblocks, int num_blocks, unsigned long wearlevel);

DESCRIPTION

Initializes the internal data structures and file system. All blocks in the file system are erased.

PARAMETERS

reserveblocks	Starting address of the flash file system. When **FS_FLASH** is defined this value should be 0 or a multiple of the block size. When **FS_RAM** is defined this parameter is ignored.
num_blocks	The number of blocks to allocate for the file system. With a default block size of 4096 bytes and a 256K Flash Memory, this value might be 64.
wearlevel	This value should be 1 on a new Flash Memory, and some higher value on an unformatted used Flash Memory. If you are reformatting a Flash Memory you can set **wearlevel** to 0 to keep the old wear leveling.

RETURN VALUE

0 on success; **1** on failure

LIBRARY

FILESYSTEM.LIB

EXAMPLE

This program can be found in **samples/filesystem/format.c**.

```
#define FS_FLASH
#use "filesystem.lib"
#define RESERVE    0
#define BLOCKS    64
#define WEAR       1

main() {
   if(fs_format(RESERVE,BLOCKS,WEAR)) {
      printf("error formating flash\n");
   } else {
      printf("flash successfully formatted\n");
   }
}
```

fs_init

```
int fs_init(long reserveblocks, int num_blocks);
```

DESCRIPTION

Initialize the internal data structures for an existing file system. Blocks that are used by a file are preserved and checked for data integrity.

PARAMETERS

reserveblocks Starting address of the flash file system. When **FS_FLASH** is defined this value should be 0 or a multiple of the block size. When **FS_RAM** is defined this parameter is ignored.

num_blocks The number of blocks that the file system contains. By default the block size is 4096 bytes.

RETURN VALUE

0 on success
1 on failure

LIBRARY

FILESYSTEM.LIB

fs_reserve_blocks

```
int fs_reserve_blocks(int blocks);
```

DESCRIPTION

Sets up a number of blocks that are guaranteed to be available for privileged files. A privileged file has an identifying number in the range 128 through 143. This function is not needed in most cases. If it is used, it should be called immediately after **fs_init** or **fs_format**.

PARAMETERS

blocks	Number of blocks to reserve.

RETURN VALUE

0 on success
1 on failure

LIBRARY

FILESYSTEM.LIB

fsck

```
int fsck(int flash);
```

DESCRIPTION

Check the filesystem for errors

PARAMETERS

flash	A bitmask indicating which checks to **NOT** perform. The following checks are available: **FSCK_HEADERS** - Block headers. **FSCK_CHECKSUMS** - Data checksums. **FSCK_VERSION** - Block versions, from a failed write.

RETURN VALUE

0 on success;
!0 on failure, this is a bitmask indicating which checks failed.

LIBRARY

FILESYSTEM.LIB

fseek

```
int fseek(File* f, long to, char whence);
```

DESCRIPTION

Places the read pointer at a desired location in the file.

PARAMETERS

f A pointer to the file to seek into.

to The number of bytes to move the read pointer. This can be a positive or negative number.

whence The location in the file to offset from. This is one of the following constants.

SEEK_SET - Seek from the beginning of the file.
SEEK_CUR - Seek from the current read position in the file.
SEEK_END - Seek from the end of the file.

EXAMPLE

To seek to 10 bytes from the end of the file **f**, use **fseek(f, -10, SEEK_END);**.
To rewind the file **f** by 5 bytes, use **fseek(f, -5, SEEK_CUR);**.

RETURN VALUE

0 on success
1 on failure

LIBRARY

FILESYSTEM.LIB

ftell

```
long ftell(File* f);
```

DESCRIPTION

Gets the offset from the beginning of a file that the read pointer is currently at.

TIP: **ftell()** can be used with **fseek()** to find the length of a file.

```
fseek(f, 0, SEEK_END); /* seek to the end of the file */
FileLength = ftell(f);    /* find the length of the file */
```

PARAMETERS

f	A pointer to the file to query.

RETURN VALUE

The offset in bytes of the read pointer from the beginning of the file.
-1 on failure.

LIBRARY

FILESYSTEM.LIB

fshift

```
int fshift(File *f, int count, char *buffer);
```

DESCRIPTION

Removes **count** number of bytes from the beginning of a file and copies them to **buffer**.

PARAMETERS

f	A pointer to the file.
count	Number of bytes to shift out.
buffer	Buffer to store shifted bytes. If this is **NULL**, the bytes will be discarded.

RETURN VALUE

Number of bytes shifted out;
0 on error.

LIBRARY

FILESYSTEM.LIB

fwrite

```
int fwrite(File* f, char* buf, int len);
```

DESCRIPTION

Appends **len** bytes from the source buffer to the end of the file.

PARAMETERS

f	A pointer to the file to write to.
buf	A pointer to the source buffer.
len	The number of bytes to write.

RETURN VALUE

The number of bytes written if successful;
0 on failure.

LIBRARY

FILESYSTEM.LIB

ftoa

```
int ftoa(float f, char *buf);
```

DESCRIPTION

Converts a float number to a character string.

The character string only displays the mantissa up to 12 digits, no decimal points. The function returns the exponent (of 10) that should be used to compensate for the string: **ftoa(1.0,buf)** yields **buf="1000000000"**, and returns **-10**.

PARAMETERS

f	Float number to convert
buf	Converted string. The string is no longer than 12 characters long.

RETURN VALUE

The exponent of the number.

LIBRARY

STDIO.LIB

SEE ALSO

utoa, itoa

getchar

```
char getchar(void);
```

DESCRIPTION

Busy waits for a character to be typed from the stdio window in Dynamic C. The user should make sure only one process calls this function at a time.

RETURN VALUE

A character typed in the stdio window in Dynamic C.

LIBRARY

STDIO.LIB

SEE ALSO

gets, putchar

getcrc

```
int getcrc(char *dataarray, char count, int accum);
```

DESCRIPTION

Computes the Cyclic Redundancy Check (CRC), or check sum, for **count** bytes (maximum 255) of data in buffer. Calls to **getcrc** can be “concatenated” using **accum** to compute the CRC for a large buffer.

PARAMETERS

dataarray	Data buffer
count	Number of bytes. Max is 255.
accum	Base CRC for the data array.

RETURN VALUE

CRC value.

LIBRARY

MATH.LIB

gets

```
char *gets(char *s);
```

DESCRIPTION

Waits for a string terminated by <CR> at the stdio window. The string returned is **NULL**-terminated without the return. The user should make sure only one process calls this function at a time.

PARAMETERS

s	The input string is put to the location pointed to by the argument **s**. The caller is responsible to make sure the location pointed to by **s** is big enough for the string.

RETURN VALUE

Same pointer passed in, but string is changed to a **NULL**-terminated.

LIBRARY

STDIO.LIB

SEE ALSO

puts, getchar

GetVectExtern2000

```
unsigned GetVectExtern2000();
```

DESCRIPTION

Reads the address of external interrupt table entry. This function really just returns what is present in the table. The return value is meaningless if the address of the external interrupt has not been written.

RETURN VALUE

Jump address in vector table.

LIBRARY

SYS.LIB

SEE ALSO

GetVectIntern, SetVectExtern2000, SetVectIntern

GetVectIntern

unsigned GetVectIntern(int vectNum);

DESCRIPTION

Reads the address of the internal interrupt table entry and returns whatever value is at the address **(internal vector table base) + (vectNum*16) + 1**.

PARAMETER

vectNum Interrupt number; should be 0–15.

RETURN VALUE

Jump address in vector table.

LIBRARY

SYS.LIB

SEE ALSO

GetVectExtern2000, SetVectExtern2000, SetVectIntern

hanncplx

```
void hanncplx(int *x, int N, int *blockexp)
```

DESCRIPTION

Convolves an **N**-point complex spectrum with the three-point Hann kernel. The filtered spectrum replaces the original spectrum.

The function produces the same results as would be obtained by multiplying the corresponding time sequence by the Hann raised-cosine window.

The zero–crossing width of the main lobe produced by the Hann window is 4 DFT bins. The adjacent sidelobes are 32 db below the main lobe. Sidelobes decay at an asymptotic rate of 18 db per octave.

N must be a power of 2 and lie between 4 and 1024. An invalid **N** causes a RANGE exception.

PARAMETERS

x	Pointer to **N**-element array of complex fractions.
N	Number of complex elements in array **x**.
blockexp	Pointer to integer block exponent.

LIBRARY

FFT.LIB

SEE ALSO

fftcplx, fftcplxinv, fftreal, fftrealinv, hanncplx, powerspectrum

hannreal

```
void hannreal(int *x, int N, int *blockexp)
```

DESCRIPTION

Convolves an **N**-point positive-frequency complex spectrum with the three-point Hann kernel. The function produces the same results as would be obtained by multiplying the corresponding time sequence by the Hann raised-cosine window.

The zero–crossing width of the main lobe produced by the Hann window is 4 DFT bins. The adjacent sidelobes are 32 db below the main lobe. Sidelobes decay at an asymptotic rate of 18 db per octave.

The imaginary part of the dc term (stored in **x[1]**) is considered to be the real part of the *fmax* term. The dc and *fmax* spectral components take part in the convolution along with the other spectral components. The real part of *fmax* component affects the real part of the $X[N\text{-}1]$ component (and vice versa), and should not arbitrarily be set to zero unless these components are unimportant.

PARAMETERS

x	Pointer to **N**-element array of complex fractions.
N	Number of complex elements in array **x**.
blockexp	Pointer to integer block exponent.

RETURN VALUE

None. The filtered spectrum replaces the original spectrum.

LIBRARY

FFT.LIB

SEE ALSO

fftcplx, fftcplxinv, fftreal, fftrealinv, hanncplx, powerspectrum

hitwd

```
void hitwd();
```

DESCRIPTION

Hits the watchdog timer, postponing a hardware reset for 2 seconds. Unless the watchdog timer is disabled, a program must call this function periodically, or the controller will automatically reset itself. If the virtual driver is enabled (which it is by default), it will call **hitwd** in the background. The virtual driver also makes additional "virtual" watchdog timers available.

LIBRARY

VDRIVER.LIB

htoa

```
char *htoa(int value, char *buf);
```

DESCRIPTION

Converts integer **value** to hexidecimal number and puts result into **buf**.

PARAMETERS

value	16-bit number to convert
buf	Character string of converted number

RETURN VALUE

Pointer to end (**NULL** terminator) of string in **buf**.

LIBRARY

STDIO.LIB

SEE ALSO

itoa, utoa, ltoa

IntervalMs

```
int IntervalMs( long ms );
```

DESCRIPTION

Similar to **DelayMs** but provides a periodic delay based on the time from the previous call. Intended for use with **waitfor**.

PARAMETERS

ms	The number of milliseconds to wait.

RETURN VALUE

0 if not finished, 1 if delay has expired.

LIBRARY

COSTATE.LIB

IntervalSec

```
int IntervalSec( long sec );
```

DESCRIPTION

Similar to **DelayMs** but provides a periodic delay based on the time from the previous call. Intended for use with **waitfor**.

PARAMETERS

sec	The number of seconds to delay.

RETURN VALUE

0 if not finished, 1 if delay has expired.

LIBRARY

COSTATE.LIB

IntervalTick

```
int IntervalTick( long tick );
```

DESCRIPTION

Provides a periodic delay based on the time from the previous call. Intended for use with **waitfor**. A tick is 1/1024 seconds.

PARAMETERS

tick	The number of ticks to delay

RETURN VALUE

0 if not finished, **1** if delay has expired.

LIBRARY

COSTATE.LIB

ipres

```
void ipres(void);
```

DESCRIPTION

Dynamic C expands this call inline. Restore previous interrupt priority by rotating the IP register.

LIBRARY

UTIL.LIB

SEE ALSO

ipset

ipset

`void ipset(int priority)`

DESCRIPTION

Dynamic C expands this call inline. Replaces current interrupt priority with another by rotating the new priority into the IP register.

PARAMETERS

`priority`	Interrupt priority range 0–3, lowest to highest priority.

LIBRARY

`UTIL.LIB`

SEE ALSO

`ipres`

isalnum

`int isalnum(int c);`

DESCRIPTION

Tests for an alphabetic or numeric character, (A to Z, a to z and 0 to 9).

PARAMETERS

`c`	Character to test.

RETURN VALUE

`0` if not an alphabetic or numeric character;
`!0` otherwise.

LIBRARY

`STRING.LIB`

SEE ALSO

`isalpha`, `isdigit`, `ispunct`

isalpha

```
int isalpha(int c);
```

DESCRIPTION

Tests for an alphabetic character, (A to Z, or a to z).

PARAMETERS

c	Character to test.

RETURN VALUE

0 if not a alphabetic character,
!0 otherwise.

LIBRARY

STRING.LIB

SEE ALSO

isalnum, isdigit, ispunct

iscntrl

```
int iscntrl(int c);
```

DESCRIPTION

Tests for a control character: 0 <= **c** <= 31 or **c** == 127.

PARAMETERS

c	Character to test.

RETURN VALUE

0 if not a control character;
!0 otherwise.

LIBRARY

STRING.LIB

SEE ALSO

isalpha, isalnum, isdigit, ispunct

isCoDone

```
int isCoDone(CoData *p);
```

DESCRIPTION

Determine if costatement is initialized and not running.

PARAMETERS

p	Address of costatement

RETURN VALUE

`1` if costatement is initialized and not running;
`0` otherwise.

LIBRARY

`COSTATE.LIB`

isCoRunning

```
int isCoRunning(CoData *p);
```

DESCRIPTION

Determine if costatement is stopped or running

PARAMETERS

p	Address of costatement

RETURN VALUE

`1` if costatement is running
`0` otherwise.

LIBRARY

`COSTATE.LIB`

isdigit

```
int isdigit(int c);
```

DESCRIPTION

Tests for a decimal digit: 0 - 9

PARAMETERS

c	Character to test.

RETURN VALUE

`0` if not a decimal digit;
`!0` otherwise.

LIBRARY

`STRING.LIB`

SEE ALSO

`isxdigit, isalpha, isalpha`

isgraph

```
int isgraph(int c);
```

DESCRIPTION

Tests for a printing character other than a space: 33 <= **c** <= 126

PARAMETERS

c	Character to test.

RETURN VALUE

`0` if not, `!0` otherwise.

LIBRARY

`STRING.LIB`

SEE ALSO

`isprint, isalpha, isalnum, isdigit, ispunct`

islower

```
int islower(int c);
```

DESCRIPTION

Tests for lower case character.

PARAMETERS

c	Character to test.

RETURN VALUE

0 if not a lower case character;
!0 otherwise.

LIBRARY

STRING.LIB

SEE ALSO

tolower, toupper, isupper

isspace

```
int isspace(int c);
```

DESCRIPTION

Tests for a white space, character, tab, return, newline, vertical tab, form feed, and space:
9 <= **c** <= 13 and **c** == 32 .

PARAMETERS

c	Character to test.

RETURN VALUE

0 if not, !0 otherwise.

LIBRARY

STRING.LIB

SEE ALSO

ispunct

isprint

```
int isprint(int c);
```

DESCRIPTION

Tests for printing character, including space: 32 <= **c** <= 126

PARAMETERS

c	Character to test.

RETURN VALUE

`0` if not a printing character, `!0` otherwise.

LIBRARY

`STRING.LIB`

SEE ALSO

`isdigit, isxdigit, isalpha, ispunct, isspace, isalnum, isgraph`

ispunct

```
int ispunct(int c);
```

DESCRIPTION

Tests for a punctuation character.

Character	Decimal Code
space	32
!"#$%&'()*+,-./	33 <= **c** <= 47
:;<=>?@	58 <= **c** <= 64
[\]^_`	91 <= **c** <= 96
{|}~	123 <= **c** <= 126

PARAMETERS

c	Character to test.

RETURN VALUE

0 if not a character,
!0 otherwise.

LIBRARY

STRING.LIB

SEE ALSO

isspace

isupper

`int isupper(int c);`

DESCRIPTION

Tests for upper case character.

PARAMETERS

`c`	Character to test.

RETURN VALUE

`0` if not, **`!0`** otherwise.

LIBRARY

`STRING.LIB`

SEE ALSO

`tolower, toupper, islower`

isxdigit

`int isxdigit(int c);`

DESCRIPTION

Tests for a hexidecimal digit: 0 - 9, A - F, a - f

PARAMETERS

`c`	Character to test.

RETURN VALUE

`0` if not a hexidecimal digit, **`!0`** otherwise.

LIBRARY

`STRING.LIB`

SEE ALSO

`isdigit, isalpha, isalpha`

itoa

`char *itoa(int value, char *buf);`

DESCRIPTION

Places up to 5 digit character string at **`*buf`**, representing value of signed number, with minus sign in first place, when appropriate.

Suppresses leading zeros, but leaves one zero digit for **`value`** = 0. Max = 65535. 73 program bytes.

PARAMETERS

`value`	16-bit number to convert
`buf`	Character string of converted number

RETURN VALUE

Pointer to the end (**`NULL`** terminator) of the string in **`buf`**.

LIBRARY

`STDIO.LIB`

SEE ALSO

`atoi, utoa, ltoa`

kbhit

`int kbhit();`

DESCRIPTION

Detects keystrokes in the Dynamic C STDIO window.

RETURN VALUE

`!0` if a key has been pressed, **`0`** otherwise

LIBRARY

`UTIL.LIB`

labs

`long labs(long x);`

DESCRIPTION

Computes the long integer absolute value of long integer **x**.

PARAMETERS

x	Number to compute.

RETURN VALUE

x, if **x** >= 0, else **-x**.

LIBRARY

MATH.LIB

SEE ALSO

abs, fabs

ldexp

`float ldexp(float x, int n);`

DESCRIPTION

Computes **x*(2**n)**

PARAMETERS

x	The value between 0.5, inclusive, and 1.0.
n	An integer

RETURN VALUE

The result of **x*(2^n)**

LIBRARY

MATH.LIB

SEE ALSO

frexp, exp

`log`

`float log(float x);`

DESCRIPTION

Computes the logarithm, base e, of real **`float`** value **`x`**.

PARAMETERS

`x`	Float value

RETURN VALUE

The function returns –INF and signals a domain error when **`x`** ≤ 0.

LIBRARY

`MATH.LIB`

SEE ALSO

`exp, log10`

`log10`

`float log10(float x);`

DESCRIPTION

Computes the base 10 logarithm of real **`float`** value **`x`**.

PARAMETERS

`x`	Value to compute

RETURN VALUE

The log base 10 of **`x`**.

The function returns –INF and signals a domain error when **`x`** ≤ 0.

LIBRARY

`MATH.LIB`

SEE ALSO

`log`, `exp`

longjmp

```
void longjmp(jmp_buf env, int val);
```

DESCRIPTION

Restores the stack environment saved in array **env[ ]**. See the description of **setjmp** for details of use.

PARAMETERS

env	Environment previously saved with **setjmp**.
val	Integer result of **setjmp**.

LIBRARY

SYS.LIB

SEE ALSO

setjmp

ltoa

```
char *ltoa(long num, char *ibuf)
```

DESCRIPTION

This function outputs a signed long number to the character array.

PARAMETERS

num	Signed long number
ibuf	Pointer to character array

RETURN VALUE

Pointer to the same array passed in to hold the result.

LIBRARY

STDIO.LIB

SEE ALSO

ltoa

ltoan

```
int ltoan(long num);
```

DESCRIPTION

This function returns the number of characters required to display a signed long number.

PARAMETERS

num	32-bit signed number

RETURN VALUE

The number of characters to display signed long number.

LIBRARY

STDIO.LIB

SEE ALSO

ltoa

memchr

```
void *memchr(void *src, int ch, unsigned int n);
```

DESCRIPTION

Searches up to **n** characters at memory pointed to by **src** for character **ch**.

PARAMETERS

src	Pointer to memory source.
ch	Character to search for.
n	Number of bytes to search.

RETURN VALUE

Pointer to first occurrence of **ch** if found within **n** characters. Otherwise returns **NULL**.

LIBRARY

STRING.LIB

SEE ALSO

strrchr, strstr

memcmp

```
int memcmp(void *s1, void *s2, size_t n);
```

DESCRIPTION

Performs unsigned character by character comparison of two memory blocks of length **n**.

PARAMETERS

s1	Pointer to block 1.
s2	Pointer to block 2.
n	Maximum number of bytes to compare.

RETURN VALUE

< 0 if **str1** is less than **str2**, meaning that a character in **str1** is less than the corresponding character in **str2**
0 if **str1** is identical to **str2**
> 0 if **str1** is greater than **str2,** meaning that a character in **str1** is greater than the corresponding character in **str2**

LIBRARY

STRING.LIB

SEE ALSO

strncmp

memcpy

```
void *memcpy(void *dst, void *src, unsigned int n);
```

DESCRIPTION

Copies a block of bytes from one destination to another. Overlap is handled correctly.

PARAMETERS

dst	Pointer to memory destination
src	Pointer to memory source
n	Number of characters to copy.

RETURN VALUE

Pointer to destination.

LIBRARY

`STRING.LIB`

SEE ALSO

`memmove, memset`

memmove

```
void *memmove(void *dst, void *src, unsigned int n);
```

DESCRIPTION

Copies a block of bytes from one destination to another. Overlap is handled correctly.

PARAMETERS

dst	Pointer to memory destination
src	Pointer to memory source
n	Number of characters to copy.

RETURN VALUE

Pointer to destination.

LIBRARY

`STRING.LIB`

SEE ALSO

`memcpy, memset`

memset

void *memset(void *dst, int chr, unsigned int n);

DESCRIPTION

Sets the first **n** bytes of a block of memory to byte destination.

PARAMETERS

dst	Block of memory to set.
chr	Byte destination
n	Amount of bytes to set.

LIBRARY

STRING.LIB

mktime

unsigned long mktime(struct tm *timeptr);

DESCRIPTION

Converts the contents of structure pointed to by **timeptr** into seconds.

```
struct tm {
   char tm_sec;        // seconds 0-59
   char tm_min;        // 0-59
   char tm_hour;       // 0-23
   char tm_mday;       // 1-31
   char tm_mon;        // 1-12
   char tm_year;       // 80-147 (1980-2047)
   char tm_wday;       // 0-6 0==sunday
};
```

PARAMETERS

timeptr	Pointer to **tm** structure:

RETURN VALUE

Time in seconds since January 1, 1980.

LIBRARY

RTCLOCK.LIB

SEE ALSO

mktm, tm_rd, tm_wr

mktm

`unsigned int mktm(struct tm *timeptr, unsigned long time);`

DESCRIPTION

Converts the seconds (**`time`**) to date and time and fills in the fields of the **`tm`** structure with the result.

```
struct tm {
    char tm_sec;        // seconds 0-59
    char tm_min;        // 0-59
    char tm_hour;       // 0-23
    char tm_mday;       // 1-31
    char tm_mon;        // 1-12
    char tm_year;       // 80-147 (1980-2047)
    char tm_wday;       // 0-6 0==sunday
};
```

PARAMETERS

`timeptr`	Address to store date and time into structure:
`time`	Seconds since January 1, 1980.

RETURN VALUE

`0`

LIBRARY

`RTCLOCK.LIB`

SEE ALSO

`mktime, tm_rd, tm_wr`

modf

```
float modf(float x, int *n);
```

DESCRIPTION

Splits **x** into a fraction and integer, **f + n**.

PARAMETERS

x	Floating-point integer
n	An integer

RETURN VALUE

The integer part in ***n** and the fractional part satisfies **|f| < 1.0**

LIBRARY

MATH.LIB

SEE ALSO

fmod, ldexp

OSInit

```
void OSInit(void);
```

DESCRIPTION

Initializes µC/OS-II data; must be called before any other µC/OS-II functions are called.

LIBRARY

UCOS2.LIB

SEE ALSO

OSTaskCreate, OSTaskCreateExt, OSStart

OSMboxAccept

```
void *OSMboxAccept (OS_EVENT *OSMboxAccept);
```

DESCRIPTION

Checks the mailbox to see if a message is available. Unlike **OSMboxPend()**, **OSMboxAccept()** does not suspend the calling task if a message is not available.

PARAMETERS

OSMboxAccept Pointer to the mailbox's event control block.

RETURN VALUE

Pointer to available message, or a **NULL** pointer if there is no available message or an error condition exists.

LIBRARY

UCOS2.LIB

SEE ALSO

OSMboxCreate, OSMboxPend, OSMboxPost, OSMboxQuery

OSMboxCreate

```
OS_EVENT *OSMboxCreate (void *msg);
```

DESCRIPTION

Creates a message mailbox if event control blocks are available.

PARAMETERS

msg Pointer to a message to put in the mailbox.

RETURN VALUE

Pointer to mailbox's event control block, or **NULL** pointer if no event control block was available.

LIBRARY

UCOS2.LIB

SEE ALSO

OSMboxAccept, OSMboxPend, OSMboxPost, OSMboxQuery

OSMboxPend

```
void *OSMboxPend(OS_EVENT *pevent, INT16U timeout, INT8U *err);
```

DESCRIPTION

Waits for a message to be sent to a mailbox.

PARAMETERS

pevent	Pointer to mailbox's event control block.
timeout	Allows task to resume execution if a message was not received by the number of clock ticks specified. Specifying 0 means the task is willing to wait forever.
err	Pointer to a variable for holding an error code.

RETURN VALUE

Pointer to a message or, if a timeout or error condition occurs, a **NULL** pointer.

LIBRARY

UCOS2.LIB

SEE ALSO

OSMboxAccept, OSMboxCreate, OSMboxPost, OSMboxQuery

OSMboxPost

INT8U OSMboxPost (OS_EVENT *pevent, void *msg);

DESCRIPTION

Sends a message to the specified mailbox

PARAMETERS

pevent	Pointer to mailbox's event control block.
msg	Pointer to message to be posted. A **NULL** pointer must not be sent.

RETURN VALUE

OS_NO_ERR	The call was successful and the message was sent.
OS_MBOX_FULL	The mailbox already contains a message. Only one message at a time can be sent and thus, the message MUST be consumed before another can be sent.
OS_ERR_EVENT_TYPE	Attempting to post to a non-mailbox.

LIBRARY

UCOS2.LIB

SEE ALSO

OSMboxAccept, OSMboxCreate, OSMboxPend, OSMboxQuery

OSMboxQuery

`INT8U OSMboxQuery (OS_EVENT *pevent, OS_MBOX_DATA *pdata);`

DESCRIPTION

Obtains information about a message mailbox.

PARAMETERS

pevent	Pointer to message mailbox's event control block.
pdata	Pointer to a data structure for information about the message mailbox

RETURN VALUE

OS_NO_ERR	The call was successful and the message was sent
OS_ERR_EVENT_TYPE	Attempting to obtain data from a non mailbox.

LIBRARY

`UCOS2.LIB`

SEE ALSO

`OSMboxAccept, OSMboxCreate, OSMboxPend, OSMboxPost`

OSMemCreate

```
OS_MEM *OSMemCreate (void *addr, INT32U nblks, INT32U blksize,
   INT8U *err);
```

DESCRIPTION

Creates a fixed-sized memory partition that will be managed by µC/OS-II.

PARAMETERS

addr	Pointer to starting address of the partition.
nblks	Number of memory blocks to create in the partition.
blksize	The size (in bytes) of the memory blocks.
err	Pointer to variable containing an error message.

RETURN VALUE

Pointer to the created memory partition control block if one is available, **NULL** pointer otherwise.

LIBRARY

UCOS2.LIB

SEE ALSO

OSMemGet, OSMemPut, OSMemQuery

OSMemGet

```
void *OSMemGet (OS_MEM *pmem, INT8U *err);
```

DESCRIPTION

Gets a memory block from the specified partition.

PARAMETERS

pmem	Pointer to partition's memory control block
err	Pointer to variable containing an error message

RETURN VALUE

Pointer to a memory block or a **NULL** pointer if an error condition is detected.

LIBRARY

UCOS2.LIB

SEE ALSO

OSMemCreate, OSMemPut, OSMemQuery

OSMemPut

```
INT8U OSMemPut(OS_MEM *pmem, void *pblk);
```

DESCRIPTION

Returns a memory block to a partition.

PARAMETERS

pmem	Pointer to the partition's memory control block.
pblk	Pointer to the memory block being released.

RETURN VALUE

OS_NO_ERR	The memory block was inserted into the partition.
OS_MEM_FULL	If returning a memory block to an already FULL memory partition (More blocks were freed than allocated!)

LIBRARY

UCOS2.LIB

SEE ALSO

OSMemCreate, OSMemGet, OSMemQuery

OSMemQuery

INT8U OSMemQuery (OS_MEM *pmem, OS_MEM_DATA *pdata);

DESCRIPTION

Determines the number of both free and used memory blocks in a memory partition.

PARAMETERS

pmem	Pointer to partition's memory control block.
pdata	Pointer to structure for holding information about the partition.

RETURN VALUE

OS_NO_ERR	This function always returns no error

LIBRARY

`UCOS2.LIB`

SEE ALSO

`OSMemCreate, OSMemGet, OSMemPut`

OSQAccept

void *OSQAccept (OS_EVENT *pevent);

DESCRIPTION

Checks the queue to see if a message is available. Unlike **OSQPend()**, with **OSQAccept()** the calling task is not suspended if a message is unavailable.

PARAMETERS

pevent	Pointer to the message queue's event control block.

RETURN VALUE

Pointer to message in the queue if one is available, **NULL** pointer otherwise.

LIBRARY

`UCOS2.LIB`

SEE ALSO

`OSQCreate, OSQFlush, OSQPend, OSQPost, OSQPostFront, OSQQuery`

OSQCreate

`OS_EVENT *OSQCreate (void **start, INT16U qsize);`

DESCRIPTION

Creates a message queue if event control blocks are available.

PARAMETERS

`start`	Pointer to the base address of the message queue storage area. The storage area MUST be declared an array of pointers to void: **`void *MessageStorage[qsize].`**
`qsize`	Number of elements in the storage area.

RETURN VALUE

Pointer to message queue's event control block or **`NULL`** pointer if no event control blocks were available.

LIBRARY

`UCOS2.LIB`

SEE ALSO

`OSQAccept, OSQFlush, OSQPend, OSQPost, OSQPostFront, OSQQuery`

OSQFlush

INT8U OSQFlush (OS_EVENT *pevent);

DESCRIPTION

Flushes the contents of the message queue.

PARAMETERS

pevent	Pointer to message queue's event control block.

RETURN VALUE

OS_NO_ERR	Upon success
OS_ERR_EVENT_TYPE	A pointer to a queue was not passed
OS_ERR_PEVENT_NULL	If 'pevent' is a **NULL** pointer

LIBRARY

UCOS2.LIB

SEE ALSO

OSQAccept, OSQCreate, OSQPend, OSQPost, OSQPostFront, OSQQuery

OSQPend

void *OSQPend (OS_EVENT *pevent, INT16U timeout, INT8U *err);

DESCRIPTION

Waits for a message to be sent to a queue.

PARAMETERS

pevent	Pointer to message queue's event control block.
timeout	Allow task to resume execution if a message was not received by the number of clock ticks specified. Specifying 0 means the task is willing to wait forever.
err	Pointer to a variable for holding an error code.

RETURN VALUE

Pointer to a message or, if a timeout occurs, a **NULL** pointer.

LIBRARY

UCOS2.LIB

SEE ALSO

OSQAccept, OSQCreate, OSQFlush, OSQPost, OSQPostFront, OSQQuery

OSQPost

```
INT8U OSQPost (OS_EVENT *pevent, void *msg);
```

DESCRIPTION

Sends a message to the specified queue.

PARAMETERS

pevent	Pointer to message queue's event control block.
msg	Pointer to the message to send. **NULL** pointer must not be sent.

RETURN VALUE

OS_NO_ERR	The call was successful and the message was sent.
OS_Q_FULL	The queue cannot accept any more messages because it is full.
OS_ERR_EVENT_TYPE	If a pointer to a queue not passed.
OS_ERR_PEVENT_NULL	If pevent is a **NULL** pointer.
OS_ERR_POST_NULL_PTR	If attempting to post to a **NULL** pointer.

LIBRARY

UCOS2.LIB

SEE ALSO

OSQAccept, OSQCreate, OSQFlush, OSQPend, OSQPostFront, OSQQuery

OSQPostFront

```
INT8U OSQPostFront (OS_EVENT *pevent, void *msg);
```

DESCRIPTION

Sends a message to the specified queue, but unlike **OSQPost()**, the message is posted at the front instead of the end of the queue. Using **OSQPostFront()** allows 'priority' messages to be sent.

PARAMETERS

pevent	Pointer to message queue's event control block.
msg	Pointer to the message to send. **NULL** pointer must not be sent.

RETURN VALUE

OS_NO_ERR	The call was successful and the message was sent.
OS_Q_FULL	The queue cannot accept any more messages because it is full.
OS_ERR_EVENT_TYPE	A pointer to a queue was not passed.
OS_ERR_PEVENT_NULL	If pevent is a NULL pointer.
OS_ERR_POST_NULL_PTR	Attempting to post to a non mailbox.

LIBRARY

UCOS2.LIB

SEE ALSO

OSQAccept, OSQCreate, OSQFlush, OSQPend, OSQPost, OSQQuery

OSQQuery

INT8U OSQQuery (OS_EVENT *pevent, OS_Q_DATA *pdata);

DESCRIPTION

Obtains information about a message queue.

PARAMETERS

pevent	Pointer to message queue's event control block.
pdata	Pointer to a data structure for message queue information.

RETURN VALUE

OS_NO_ERR	The call was successful and the message was sent.
OS_ERR_EVENT_TYPE	Attempting to obtain data from a non queue.
OS_ERR_PEVENT_NULL	If pevent is a **NULL** pointer.

LIBRARY

UCOS2.LIB

SEE ALSO

OSQAccept, OSQCreate, OSQFlush, OSQPend, OSQPost, OSQPostFront

OSSchedLock

void OSSchedLock(void);

DESCRIPTION

Prevents task rescheduling. This allows an application to prevent context switches until it is ready for them. There must be a matched call to **OSSchedUnlock()** for every call to **OSSchedLock()**.

LIBRARY

UCOS2.LIB

SEE ALSO

OSSchedUnlock

OSSchedUnlock

```
void OSSchedUnlock(void);
```

DESCRIPTION

Allow task rescheduling. There must be a matched call to **OSSchedUnlock()** for every call to **OSSchedLock()**.

LIBRARY

UCOS2.LIB

SEE ALSO

OSSchedLock

OSSemAccept

```
INT16U OSSemAccept (OS_EVENT *pevent);
```

DESCRIPTION

This function checks the semaphore to see if a resource is available or if an event occurred. Unlike **OSSemPend()**, **OSSemAccept()** does not suspend the calling task if the resource is not available or the event did not occur.

PARAMETERS

pevent	Pointer to the desired semaphore's event control block

RETURN VALUE

Semaphore value:
If `>0`, semaphore value is decremented; value is returned before the decrement.
If `0`, then either resource is unavailable, event did not occur, or **NULL** or invalid pointer was passed to the function.

LIBRARY

UCOS2.LIB

SEE ALSO

OSSemCreate, OSSemPend, OSSemPost, OSSemQuery

OSSemCreate

```
OS_EVENT *OSSemCreate (INT16U cnt);
```

DESCRIPTION

Creates a semaphore.

PARAMETERS

cnt	The initial value of the semaphore.

RETURN VALUE

Pointer to the event control block (**OS_EVENT**) associated with the created semaphore, or **NULL** if no event control block is available.

LIBRARY

UCOS2.LIB

SEE ALSO

OSSemAccept, OSSemPend, OSSemPost, OSSemQuery

OSSemPend

```
void OSSemPend (OS_EVENT *pevent, INT16U timeout, INT8U *err);
```

DESCRIPTION

Waits on a semaphore.

PARAMETERS

pevent	Pointer to the desired semaphore's event control block
timeout	Time in clock ticks to wait for the resource. If 0, the task will wait until the resource becomes available or the event occurs.
err	Pointer to error message.

LIBRARY

UCOS2.LIB

SEE ALSO

OSSemAccept, OSSemCreate, OSSemPost, OSSemQuery

OSSemPost

INT8U OSSemPost (OS_EVENT *pevent);

DESCRIPTION

This function signals a semaphore.

PARAMETERS

pevent	Pointer to the desired semaphore's event control block

RETURN VALUE

OS_NO_ERR	The call was successful and the semaphore was signaled.
OS_SEM_OVF	If the semaphore count exceeded its limit. In other words, you have signalled the semaphore more often than you waited on it with either **OSSemAccept()** or **OSSemPend()**.
OS_ERR_EVENT_TYPE	If a pointer to a semaphore not passed.
OS_ERR_PEVENT_NULL	If 'pevent' is a **NULL** pointer.

LIBRARY

UCOS2.LIB

SEE ALSO

OSSemAccept, OSSemCreate, OSSemPend, OSSemQuery

OSSemQuery

INT8U OSSemQuery (OS_EVENT *pevent, OS_SEM_DATA *pdata);

DESCRIPTION

Obtains information about a semaphore.

PARAMETERS

pevent	Pointer to the desired semaphore's event control block
pdata	Pointer to a data structure that will hold information about the semaphore.

RETURN VALUE

OS_NO_ERR	The call was successful and the message was sent.
OS_ERR_EVENT_TYPE	Attempting to obtain data from a non semaphore.
OS_ERR_PEVENT_NULL	If pevent is a **NULL** pointer.

LIBRARY

UCOS2.LIB

SEE ALSO

OSSemAccept, OSSemCreate, OSSemPend, OSSemPost

OSSetTickPerSec

`INT16U OSSetTickPerSec(INT16U TicksPerSec);`

DESCRIPTION

Sets the amount of ticks per second (from 1 - 2048). Ticks per second defaults to 64. If this function is used, the **`#define OS_TICKS_PER_SEC`** needs to be changed so that the time delay functions work correctly. Since this function uses integer division, the actual ticks per second may be slightly different that the desired ticks per second.

PARAMETERS

`TicksPerSec`	Unsigned 16-bit integer.

RETURN VALUE

The actual ticks per second set, as an unsigned 16-bit integer.

LIBRARY

`UCOS2.LIB`

SEE ALSO

`OSStart`

OSStart

`void OSStart(void);`

DESCRIPTION

Starts the multitasking process, allowing µC/OS-II to manage the tasks that have been created. Before **`OSStart()`** is called, **`OSInit()`** MUST have been called and at least one task MUST have been created. This function calls **`OSStartHighRdy`** which calls **`OSTaskSwHoo`**k and sets **`OSRunning`** to TRUE.

LIBRARY

`UCOS2.LIB`

SEE ALSO

`OSTaskCreate, OSTaskCreateExt`

OSStatInit

void OSStatInit(void);

DESCRIPTION

Determines CPU usage.

LIBRARY

UCOS2.LIB

OSTaskChangePrio

INT8U OSTaskChangePrio (INT8U oldprio, INT8U newprio);

DESCRIPTION

Allows a task's priority to be changed dynamically. Note that the new priority MUST be available.

PARAMETERS

oldprio	The priority level to change from.
newprio	The priority level to change to.

RETURN VALUE

OS_NO_ERR	The call was successful.
OS_PRIO_INVALID	The priority specified is higher that the maximum allowed (i.e. >= **OS_LOWEST_PRIO**).
OS_PRIO_EXIST	The new priority already exist.
OS_PRIO_ERR	There is no task with the specified OLD priority (i.e. the OLD task does not exist).

LIBRARY

UCOS2.LIB

OSTaskCreate

INT8U OSTaskCreate(void (*task)(), void *pdata, INT16U stk_size, INT8U prio);

DESCRIPTION

Creates a task to be managed by µC/OS-II. Tasks can either be created prior to the start of multitasking or by a running task. A task cannot be created by an ISR.

PARAMETERS

task	Pointer to the task's starting address.
pdata	Pointer to a task's initial parameters.
stk_size	Number of bytes of the stack.
prior	The task's unique priority number.

RETURN VALUE

OS_NO_ERR	The call was successful.
OS_PRIO_EXIT	The task priority already exists (each task MUST have a unique priority).
OS_PRIO_INVALID	The priority specified is higher than the maximum allowed (i.e. >= **OS_LOWEST_PRIO**).

LIBRARY

UCOS2.LIB

SEE ALSO

OSTaskCreateExt

OSTaskCreateExt

```
INT8U OSTaskCreateExt (void   (*task)(), void *pdata, INT8U
   prio, INT16U id, INT16U stk_size, void *pext, INT16U opt);
```

DESCRIPTION

Creates a task to be managed by µC/OS-II. Tasks can either be created prior to the start of multitasking or by a running task. A task cannot be created by an ISR. This function is similar to **`OSTaskCreate()`** except that it allows additional information about a task to be specified.

PARAMETERS

`task`	Pointer to task's code.
`pdata`	Pointer to optional data area; used to pass parameters to the task at start of execution.
`prio`	The task's unique priority number; the lower the number the higher the priority.
`id`	The task's identification number (0..65535).
`stk_size`	Size of the stack in number of elements. If **`OS_STK`** is set to **`INT8U`**, **`stk_size`** corresponds to the number of bytes available. If **`OS_STK`** is set to **`INT16U`**, **`stk_size`** contains the number of 16-bit entries available. Finally, if **`OS_STK`** is set to **`INT32U`**, **`stk_size`** contains the number of 32-bit entries available on the stack.
`pext`	Pointer to a user-supplied Task Control Block (TCB) extension.
`opt`	The lower 8 bits are reserved by µC/OS-II. The upper 8 bits control application-specific options. Select an option by setting the corresponding bit(s).

RETURN VALUE

`OS_NO_ERR`	The call was successful.
`OS_PRIO_EXIT`	The task priority already exists (each task MUST have a unique priority).
`OS_PRIO_INVALID`	The priority specified is higher than the maximum allowed (i.e. >= **`OS_LOWEST_PRIO`**).

LIBRARY

`UCOS2.LIB`

SEE ALSO

`OSTaskCreate`

OSTaskCreateHook

void OSTaskCreateHook(OS_TCB *ptcb);

DESCRIPTION

Called by µC/OS-II whenever a task is created. This call-back function resides in **UCOS2.LIB** and extends functionality during task creation by allowing additional information to be passed to the kernel, anything associated with a task. This function can also be used to trigger other hardware, such as an oscilloscope. Interrupts are disabled during this call, therefore, it is recommended that code be kept to a minimum.

PARAMETERS

ptcb	Pointer to the TCB of the task being created.

LIBRARY

UCOS2.LIB

SEE ALSO

OSTaskDelHook

OSTaskDel

INT8U OSTaskDel (INT8U prio);

DESCRIPTION

Deletes a task. The calling task can delete itself by passing either its own priority number or **OS_PRIO_SELF** if it doesn't know its priority number. The deleted task is returned to the dormant state and can be re-activated by creating the deleted task again.

PARAMETERS

prio	Task's priority number.

RETURN VALUE

OS_NO_ERR	The call was successful.
OS_TASK_DEL_IDLE	Attempting to delete uC/OS-II's idle task.
OS_PRIO_INVALID	The priority specified is higher than the maximum allowed (i.e. >= **OS_LOWEST_PRIO**) or, **OS_PRIO_SELF** not specified.
OS_TASK_DEL_ERR	The task to delete does not exist.
OS_TASK_DEL_ISR	Attempting to delete a task from an ISR.

LIBRARY

UCOS2.LIB

SEE ALSO

OSTaskDelReq

OSTaskDelHook

`void OSTaskDelHook(OS_TCB *ptcb);`

DESCRIPTION

Called by µC/OS-II whenever a task is deleted. This call-back function resides in **`UCOS2.LIB.`** Interrupts are disabled during this call, therefore, it is recommended that code be kept to a minimum.

PARAMETERS

`ptcb`	Pointer to TCB of task being deleted.

LIBRARY

`UCOS2.LIB`

SEE ALSO

`OSTaskCreateHook`

OSTaskDelReq

INT8U OSTaskDelReq (INT8U prio);

DESCRIPTION

Notifies a task to delete itself. A well-behaved task is deleted when it regains control of the CPU by calling **OSTaskDelReq (OSTaskDelReq)** and monitoring the return value.

PARAMETERS

prio	The priority of the task that is being asked to delete itself. **OS_PRIO_SELF** is used when asking whether another task wants the current task to be deleted.

RETURN VALUE

OS_NO_ERR	The task exists and the request has been registered.
OS_TASK_NOT_EXIST	The task has been deleted. This allows the caller to know whether the request has been executed.
OS_TASK_DEL_IDLE	If requesting to delete uC/OS-II's idletask.
OS_PRIO_INVALID	The priority specified is higher than the maximum allowed (i.e. >= **OS_LOWEST_PRIO**) or, **OS_PRIO_SELF** is not specified.
OS_TASK_DEL_REQ	A task (possibly another task) requested that the running task be deleted.

LIBRARY

UCOS2.LIB

SEE ALSO

OSTaskDel

OSTaskQuery

INT8U OSTaskQuery (INT8U prio, OS_TCB *pdata);

DESCRIPTION

Obtains a copy of the requested task's TCB.

PARAMETERS

prio	Priority number of the task.
pdata	Pointer to task's TCB.

RETURN VALUE

OS_NO_ERR	The requested task is suspended.
OS_PRIO_INVALID	The priority you specify is higher than the maximum allowed (i.e. >= **OS_LOWEST_PRIO**) or, **OS_PRIO_SELF** is not specified.
OS_PRIO_ERR	The desired task has not been created.

LIBRARY

UCOS2.LIB

OSTaskResume

`INT8U OSTaskResume (INT8U prio);`

DESCRIPTION

Resumes a suspended task. This is the only call that will remove an explicit task suspension.

PARAMETERS

prio	The priority of the task to resume.

RETURN VALUE

OS_NO_ERR	The requested task is resumed.
OS_PRIO_INVALID	The priority specified is higher than the maximum allowed (i.e. >= **OS_LOWEST_PRIO**).
OS_TASK_NOT_SUSPENDED	The task to resume has not been suspended.

LIBRARY

UCOS2.LIB

SEE ALSO

OSTaskSuspend

OSTaskStatHook

`void OSTaskStatHook();`

DESCRIPTION

Called every second by µC/OS-II's statistics task. This function resides in **UCOS2.LIB** and allows an application to add functionality to the statistics task.

LIBRARY

UCOS2.LIB

OSTaskStkChk

INT8U OSTaskStkChk (INT8U prio, OS_STK_DATA *pdata);

DESCRIPTION

Check the amount of free memory on the stack of the specified task.

PARAMETERS

prio	The task's priority.
pdata	Pointer to a data structure of type **OS_STK_DATA.**

RETURN VALUE

OS_NO_ERR	The call was successful.
OS_PRIO_INVALID	The priority you specify is higher than the maximum allowed (i.e. > **OS_LOWEST_PRIO**) or, **OS_PRIO_SELF** not specified.
OS_TASK_NOT_EXIST	The desired task has not been created.
OS_TASK_OPT_ERR	If **OS_TASK_OPT_STK_CHK** was NOT specified when the task was created.

LIBRARY

UCOS2.LIB

SEE ALSO

OSTaskCreateExt

OSTaskSuspend

```
INT8U OSTaskSuspend (INT8U prio);
```

DESCRIPTION

Suspends a task. The task can be the calling task if the priority passed to **OSTaskSuspend()** is the priority of the calling task or **OS_PRIO_SELF**. This function should be used with great care. If a task is suspended that is waiting for an event (i.e. a message, a semaphore, a queue ...) the task will be prevented from running when the event arrives.

PARAMETERS

prio	The priority of the task to suspend.

RETURN VALUE

OS_NO_ERR	The requested task is suspended.
OS_TASK_SUS_IDLE	Attempting to suspend the idle task (not allowed).
OS_PRIO_INVALID	The priority specified is higher than the max allowed (i.e. >= **OS_LOWEST_PRIO**) or, **OS_PRIO_SELF** is not specified .
OS_TASK_SUS_PRIO	The task to suspend does not exist.

LIBRARY

UCOS2.LIB

SEE ALSO

OSTaskResume

OSTaskSwHook

```
void OSTaskSwHook();
```

DESCRIPTION

Called whenever a context switch happens. The TCB for the task that is ready to run is accessed via the global variable **OSTCBHighRdy**, and the TCB for the task that is being switched out is accessed via the global variable **OSTCBCur**.

LIBRARY

UCOS2.LIB

OSTimeDly

void OSTimeDly (INT16U ticks);

DESCRIPTION

Delays execution of the task for the specified number of clock ticks. No delay will result if **ticks** is 0. If **ticks** is >0, then a context switch will result.

PARAMETERS

ticks	Number of clock ticks to delay the task.

LIBRARY

UCOS2.LIB

SEE ALSO

OSTimeDlyHMSM, OSTimeDlyResume, OSTimeDlySec

OSTimeDlyHMSM

```
INT8U OSTimeDlyHMSM (INT8U hours, INT8U minutes, INT8U seconds,
   INT16U milli);
```

DESCRIPTION

Delays execution of the task until specified amount of time expires. This call allows the delay to be specified in hours, minutes, seconds and milliseconds instead of ticks. The resolution on the milliseconds depends on the tick rate. For example, a 10 ms delay is not possible if the ticker interrupts every 100 ms. In this case, the delay would be set to 0. The actual delay is rounded to the nearest tick.

PARAMETERS

hours	Number of hours that the task will be delayed (max. is 255)
minutes	Number of minutes (max. 59)
seconds	Number of seconds (max. 59)
milli	Number of milliseconds (max. 999)

RETURN VALUE

OS_NO_ERR

OS_TIME_INVALID_MINUTES

OS_TIME_INVALID_SECONDS

OS_TIME_INVALID_MS

OS_TIME_ZERO_DLY

LIBRARY

UCOS2.LIB

SEE ALSO

OSTimeDly, OSTimeDlyResume, OSTimeDlySec

OSTimeDlyResume

`INT8U OSTimeDlyResume (INT8U prio);`

DESCRIPTION

Resumes a task that has been delayed through a call to either **`OSTimeDly()`** or **`OSTimeDlyHMSM()`**. Note that this function MUST NOT be called to resume a task that is waiting for an event with timeout. This situation would make the task look like a timeout occurred (unless this is the desired effect). Also, a task cannot be resumed that has called **`OSTimeDlyHMSM()`** with a combined time that exceeds 65535 clock ticks. In other words, if the clock tick runs at 100 Hz then, a delayed task will not be able to be resumed that called **`OSTimeDlyHMSM(0, 10, 55, 350)`** or higher.

PARAMETERS

`prio`	Priority of the task to resume.

RETURN VALUE

`OS_NO_ERR`	Task has been resumed.
`OS_PRIO_INVALID`	The priority you specify is higher than the maximum allowed (i.e. >= **`OS_LOWEST_PRIO`**).
`OS_TIME_NOT_DLY`	Task is not waiting for time to expire.
`OS_TASK_NOT_EXIST`	The desired task has not been created.

LIBRARY

`UCOS2.LIB`

SEE ALSO

`OSTimeDly, OSTimeDlyHMSM, OSTimeDlySec`

OSTimeDlySec

INT8U OSTimeDlySec (INT16U seconds);

DESCRIPTION

Delays execution of the task until **seconds** expires. This is a low-overhead version of **OSTimeDlyHMSM** for seconds only.

PARAMETERS

seconds	The number of seconds to delay.

RETURN VALUE

OS_NO_ERR	The call was successful.
OS_TIME_ZERO_DLY	A delay of zero seconds was requested.

LIBRARY

UCOS2.LIB

SEE ALSO

OSTimeDly, OSTimeDlyHMSM, OSTimeDlyResume

OSTimeGet

INT32U OSTimeGet (void);

DESCRIPTION

Obtain the current value of the 32-bit counter that keeps track of the number of clock ticks.

RETURN VALUE

The current value of **OSTime**

LIBRARY

UCOS2.LIB

SEE ALSO

OSTimeSet

OSTimeSet

void OSTimeSet (INT32U ticks);

DESCRIPTION

Sets the 32-bit counter that keeps track of the number of clock ticks.

PARAMETERS

ticks	The value to set **OSTime** to.

LIBRARY

UCOS2.LIB

SEE ALSO

OSTimeGet

OSTimeTickHook

void OSTimeTickHook();

DESCRIPTION

This function, as included with Dynamic C, is a stub that does nothing except return. It is called every clock tick. If the user chooses to rewrite this function, code should be kept to a minimum as it will directly affect interrupt latency. This function must preserve any registers it uses, other than the ones that are preserved prior to the call to **OSTimeTick-Hook** at the beginning of the periodic interrupt (**periodic_isr** in **VDRIVER.LIB**). Therefore, **OSTimeTickHook** should be written in assembly. The registers saved by **periodic_isr** are: AF,IP, HL,DE and IX.

LIBRARY

UCOS2.LIB

OSVersion

```
INT16U OSVersion (void)
```

DESCRIPTION

Returns the version number of µC/OS-II. The returned value corresponds to µC/OS-II's version number multiplied by 100; i.e., version 2.00 would be returned as 200.

RETURN VALUE

Version number multiplied by 100.

LIBRARY

UCOS2.LIB

outchrs

```
char outchrs(char c, int n, int (*putc) () );
```

DESCRIPTION

Use **putc** to output **n** times the character **c**.

PARAMETERS

c	Character to output
n	Number of times to output
putc	Routine to output one character. The function pointed to by **putc** should take a character argument.

RETURN VALUE

The character in parameter **c**.

LIBRARY

STDIO.LIB

SEE ALSO

outstr

outstr

```
char *outstr(char *string, int (*putc)() );
```

DESCRIPTION

Output the string pointed to by **string** via calls to **putc**. **putc** should take a one-character parameter.

PARAMETERS

string	String to output
putc	Routine to output one character. The function pointed to by **putc** should take a character argument.

RETURN VALUE

Pointer to **NULL** at end of string.

LIBRARY

STDIO.LIB

SEE ALSO

outchrs

paddr

```
unsigned long paddr(void* pointer)
```

DESCRIPTION

Converts a logical pointer into its physical address. Use caution when converting address in the E000-FFFF range. Returns the address based on the XPC on entry.

PARAMETERS

pointer	The pointer to convert.

RETURN VALUE

The physical address of the pointer.

LIBRARY

XMEM.LIB

poly

```
float poly(float x, int n, float c[]);
```

DESCRIPTION

Computes polynomial value by Horner's method. For example, for the fourth-order polynomial $10x^4 - 3x^2 + 4x + 6$, **n** would be 4 and the coefficients would be

```
c[4] = 10.0
c[3] = 0.0
c[2] = -3.0
c[1] = 4.0
c[0] = 6.0
```

PARAMETERS

x	Variable of the polynomial.
n	The order of the polynomial
c	Array containing the coefficients of each power of **x**.

RETURN VALUE

The polynomial value.

LIBRARY

MATH.LIB

pow

```
float pow(float x, float y);
```

DESCRIPTION

Raises **x** to the **y**th power.

PARAMETERS

x	Value to be raised
y	Exponent

RETURN VALUE

x to the **y**th power

LIBRARY

MATH.LIB

SEE ALSO

exp, pow10, sqrt

pow10

```
float pow10(float x);
```

DESCRIPTION

10 to the power of **x**.

PARAMETERS

x	Exponent

RETURN VALUE

10 raised to power **x**

LIBRARY

MATH.LIB

SEE ALSO

pow, exp, sqrt

powerspectrum

void powerspectrum(int *x, int N, *int blockexp)

DESCRIPTION

Computes the power spectrum from a complex spectrum according to

$$Power[k] = (\mathrm{Re}\,X[k])^2 + (\mathrm{Im}\,X[k])^2$$

The **N**-point power spectrum replaces the **N**-point complex spectrum. The power of each complex spectral component is computed as a 32-bit fraction. Its more significant 16-bits replace the imaginary part of the component; its less significant 16-bits replace the real part.

If the complex input spectrum is a positive-frequency spectrum computed by **fftreal(),** the imaginary part of the $X[0]$ term (stored **x[1]**) will contain the real part of the *fmax* term and will affect the calculation of the dc power. If the dc power or the *fmax* power is important, the *fmax* term should be retrieved from **x[1]** and **x[1]** set to zero before calling **powerspectrum()**.

The power of the *k*th term can be retrieved via
P[k]=*(long*)&x[2k]*2^blockexp.

The value of **blockexp** is first doubled to reflect the squaring operation applied to all elements in array **x**. Then it is further increased by 1 to reflect an inherent division-by-two that occurs during the squaring operation.

PARAMETERS

x	pointer to **N**-element array of complex fractions.
N	number of complex elements in array **x**.
blockexp	pointer to integer block exponent.

LIBRARY

FFT.LIB

SEE ALSO

fftcplx, fftcplxinv, fftreal, fftrealinv, hanncplx, hannreal

premain

```
void premain();
```

DESCRIPTION

Dynamic C calls **premain** to start initialization functions such as **VdInit**. The final thing **premain** does is call **main**. This function should never be called by an application program. It is included here for informational purposes only.

LIBRARY

PROGRAM.LIB

printf

```
void printf(char *fmt, ...);
```

DESCRIPTION

Outputs the formatted string to the Stdio window in Dynamic C. It will work only when the controller is in program mode and is connected to the PC running Dynamic C. Unlike **sprintf**, only one process should use this function at any time.

PARAMETERS

format	String to be formatted.
...	Format arguments.

LIBRARY

STDIO.LIB

SEE ALSO

sprintf

putchar

void putchar(int ch);

DESCRIPTION

Puts a single character to STDOUT. The user should make sure only one process calls this function at a time.

PARAMETERS

ch	Character to be displayed.

LIBRARY

`STDIO.LIB`

SEE ALSO

`puts, getchar`

puts

int puts(char *s);

DESCRIPTION

This function displays the string on the stdio window in Dynamic C. The STDIO window is responsible for interpreting any escape code sequences contained in the string. Only one process at a time should call this function.

PARAMETERS

s	Pointer to string argument to be displayed.

RETURN VALUE

1 if successful.

LIBRARY

`STDIO.LIB`

SEE ALSO

`putchar, gets`

qsort

```
int qsort(char *base, unsigned n, unsigned s, int (*cmp) ());
```

DESCRIPTION

Quick sort with center pivot, stack control, and easy-to-change comparison method. This version sorts fixed-length data items. It is ideal for integers, longs, floats and packed string data without delimiters.

Can sort raw integers, longs, floats or strings. However, the string sort is not efficient.

PARAMETERS

base	Base address of the raw string data
n	Number of blocks to sort
s	Number of bytes in each block
cmp	User-supplied compare routine for two block pointers, **p** and **q**, that returns an int with the same rules used by Unix **strcmp(p,q)**: = 0 Blocks **p** and **q** are equal < 0 **p** < **q** > 0 **p** > **q** Beware of using ordinary **strcmp()**—it requires a **NULL** at the end of each string.

RETURN VALUE

0 if the operation is successful.

LIBRARY

SYS.LIB

EXAMPLE

```
// Sort an array of integers.
int mycmp(p,q) int *p,*q; { return (*p - *q);}
const int q[10] = {12,1,3,-2,16,7,9,34,-90,10};
const int p[10] = {12,1,3,-2,16,7,9,34,-90,10};
main() {
     int i;
     qsort(p,10,2,mycmp);
     for(i=0;i<10;++i) printf("%d. %d, %d\n",i,p[i],q[i]);
}
```

```
Output from the above sample program:
0. -90,  12
1.  -2,   1
2.   1,   3
3.   3,  -2
4.   7,  16
5.   9,   7
6.  10,   9
7.  12,  34
8.  16, -90
9.  34,  10
```

rad

`float rad(float x);`

DESCRIPTION

Convert degrees (360 for one rotation) to radians (2π for a rotation).

PARAMETERS

x	Degree value to convert

RETURN VALUE

The radians equivalent of degree.

LIBRARY

`SYS.LIB`

SEE ALSO

`deg`

rand

```
float rand(void);
```

DESCRIPTION

Uses algorithm **rand = (5*rand)modulo 2^32**. The random seed is a global unsigned long, **ran_seed,** set by initialization (**GLOBAL_INIT**). It may be modified by the user. This function is not task reentrant.

RETURN VALUE

A uniformly distributed random number: 0.0 <= **v** < 1.0.

LIBRARY

MATH.LIB

SEE ALSO

randb, randg

randb

```
float randb(void);
```

DESCRIPTION

Uses algorithm **rand = (5*rand)modulo 2^32**. The random seed is a global unsigned long, **ran_seed,** set by initialization (**GLOBAL_INIT**). It may be modified by the user. This function is not task reentrant.

RETURN VALUE

Returns a uniformly distributed random number: -1.0 <= **v** < 1.0.

LIBRARY

MATH.LIB

SEE ALSO

rand, randg

randg

float randg(void);

DESCRIPTION

Distribution is made by adding 16 random numbers uniformly distributed as -1.0 < v < 1.0. Standard deviation is approximately 2.6, mean 0. Algorithm used is **rand = (5*rand)modulo 2^32**. The random seed is a global unsigned long, **ran_seed,** set by initialization (**GLOBAL_INIT**). It may be modified by the user. This function is not task reentrant.

RETURN VALUE

A gaussian distributed random number: -16.0 <= **v** <16.0.

LIBRARY

MATH.LIB

SEE ALSO

rand, randb

RdPortE

int RdPortE(int port);

DESCRIPTION

Reads an external I/O register specified by the argument.

PARAMETERS

port	Address of external parallel port data register.

RETURN VALUE

Returns an integer, the lower 8 bits of which contain the result of reading the port specified by the argument. Upper byte contains zero.

LIBRARY

SYSIO.LIB

SEE ALSO

RdPortI, BitRdPortI, WrPortI, BitWrPortI, BitRdPortE, WrPortE, BitWrPortE

RdPortI

int RdPortI(int port);

DESCRIPTION

Reads an internal I/O port specified by the argument.

PARAMETERS

port Address of internal parallel port data register.

RETURN VALUE

Returns an integer, the lower 8 bits of which contain the result of reading the port specified by the argument. Upper byte contains zero.

LIBRARY

SYSIO.LIB

SEE ALSO

RdPortE, BitRdPortI, WrPortI, BitWrPortI, BitRdPortE, WrPortE, BitWrPortE

read_rtc

unsigned long read_rtc(void);

DESCRIPTION

Reads the RTC directly - use with caution! In most cases use long variable **SEC_TIMER** which contains the same result, unless the RTC has been changed since the start of the program. If you are running the processor off the 32kHz crystal, use the **read_rtc_32kHz()** function instead.

RETURN VALUE

Time in seconds since January 1, 1980 (if RTC set correctly).

LIBRARY

RTCLOCK.LIB

SEE ALSO

write_rtc

read_rtc_32kHz

```
unsigned long read_rtc_32kHz(void);
```

DESCRIPTION

Reads the real-time clock directly when the Rabbit processor is running off the 32kHz oscillator. See **read_rtc** for more details.

RETURN VALUE

Time in seconds since January 1, 1980 (if RTC set correctly).

LIBRARY

RTCLOCK.LIB

res

```
void res(void *address, unsigned int bit);
```

DESCRIPTION

Dynamic C may expand this call inline

Clears specified bit at memory address to 0. bit may be from 0 to 31. This is equivalent to the following expression, but more efficient:

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

PARAMETERS

address	Address of byte containing bits 7-0
bit	Bit location where 0 represents the least significant bit

LIBRARY

UTIL.LIB

SEE ALSO

RES

RES

`void RES(void *address, unsigned int bit);`

DESCRIPTION

Dynamic C may expand this call inline.

Clears specified bit at memory address to 0. bit may be from 0 to 31. This is equivalent to the following expression, but more efficient:

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

PARAMETERS

`address`	Address of byte containing bits 7-0
`bit`	Bit location where 0 represents the least significant bit

LIBRARY

`UTIL.LIB`

SEE ALSO

`res`

root2xmem

```
int root2xmem(unsigned long dest, void *src, unsigned len);
```

DESCRIPTION

Stores **len** characters from logical address **src** to physical address **dest**.

PARAMETERS

dest	Physical address
src	Logical address
len	Numbers of bytes

RETURN VALUE

0—success
1—attempt to write Flash Memory area, nothing written
2—source not all in root

LIBRARY

`XMEM.LIB`

SEE ALSO

`xalloc, xmem2root`

runwatch

```
void runwatch();
```

DESCRIPTION

Runs and updates watch expressions if Dynamic C has requested it with a **Ctrl-U**. Should be called periodically in user program.

LIBRARY

`SYS.LIB`

serCheckParity

```
int serCheckParity(char rx_byte, char parity);
```

DESCRIPTION

This function is different from the other serial routines in that it does not specify a particular serial port. This function takes any 8-bit character and tests it for correct parity. It will return true if the parity of **rx_byte** matches the parity specified. This function is useful for checking individual characters when using a 7-bit data protocol.

PARAMETERS

rx_byte	The 8 bit character being tested for parity.
parity	The character 'O' for odd parity, or the character 'E' for even parity.

RETURN VALUE

1 - if the parity of the byte being tested matches the parity supplied as an argument.
0 - if the parity of the byte does not match.

LIBRARY

RS232.LIB

serXclose

```
void serXclose(); /* where X = A|B|C|D */
```

DESCRIPTION

Disables serial port X. This function is non-reentrant.

LIBRARY

RS232.LIB

serXdatabits

```
void serXdatabits(state); /* where X = A|B|C|D */
```

DESCRIPTION

Sets the number of data bits in the serial format for this channel. Currently seven or eight bit modes are supported. This function is non-reentrant.

PARAMETERS

state	An integer indicating what bit mode to use. It is best to use one of the macros provided for this:
PARAM_7BIT	Configures serial port to use seven bit data
PARAM_8BIT	Configures serial port to use eight bit data (default)

LIBRARY

RS232.LIB

serXflowcontrolOff

```
void serXflowcontrolOff(); /* where X = A|B|C|D */
```

DESCRIPTION

Turns off hardware flow control for serial port X. This function is non-reentrant.

LIBRARY

RS232.LIB

serXflowcontrolOn

```
void serXflowcontrolOn(); /* where X = A|B|C|D */
```

DESCRIPTION

Turns on hardware flow control for channel X. This enables two digital lines that handle flow control, CTS (clear to send) and RTS (ready to send). CTS is an input that will be pulled active low by the other system when it is ready to receive data. The RTS signal is an output that the system uses to indicate that it is ready to receive data; it is driven low when data can be received.

This function is non-reentrant.

If pins for the flow control lines are not explicitly defined, defaults will be used and compiler warnings will be issued. The locations of the flow control lines are specified using a set of 5 macros (X is A|B|C|D).

SERX_RTS_PORT	Data register for the parallel port that the RTS line is on. e.g. PCDR
SERA_RTS_SHADOW	Shadow register for the RTS line's parallel port. e.g. PCDRShadow
SERA_RTS_BIT	The bit number for the RTS line
SERA_CTS_PORT	Data register for the parallel port that the CTS line is on
SERA_CTS_BIT	The bit number for the CTS line

LIBRARY

RS232.LIB

serXgetc

`int serXgetc(); /* where X = A|B|C|D */`

DESCRIPTION

Get next available character from serial port X read buffer. This function is non-reentrant.

RETURN VALUE

Success: the next character in the low byte, **`0`** in the high byte
Failure: **`-1`**

LIBRARY

`RS232.LIB`

EXAMPLE

```
// echoes characters
main() {
     int c;
     serAopen(19200);
     while (1) {
          if ((c = serAgetc()) != -1) {
               serAputc(c);
          }
     }
     serAclose()
}
```

serXgetError

```
int serXgetError(); /* where X = A|B|C|D */
```

DESCRIPTION

Returns a byte of error flags, with bits set for any errors that occurred since the last time this function was called. Any bits set will be automatically cleared when this function is called, so a particular error will only be reported once. This function is non-reentrant.

The flags are checked with bitmasks to determine which errors occurred. Error bitmasks:

SER_PARITY_ERROR
SER_OVERRUN_ERROR

RETURN VALUE

The error flags byte.

LIBRARY

RS232.LIB

serXopen

`int serXopen(long baud); /* where X = A|B|C|D */`

DESCRIPTION

Opens serial port X. This function is non-reentrant.

Defining Buffer Sizes: **`XINBUFSIZE`** and **`XOUTBUFSIZE`**

The user must define the buffer sizes for each port being used to be a power of 2 minus 1 with a macro, e.g.

```
#define XINBUFSIZE   63
#define XOUTBUFSIZE  127
```

Defining the buffer sizes to 2^n - 1 makes the circular buffer operations very efficient. If a value not equal to 2^n- 1 is defined, a default of 31 is used and a compiler warning is given.

PARAMETERS

`baud`	Bits per second of data transfer. Note that the baud rate must be greater than or equal to the peripheral clock frequency ÷ 8192.

RETURN VALUE

`1`, if the baud rate achieved on the Rabbit is the same as the input baud rate.
`0`, if the baud rate achieved on the Rabbit does not match the input baud rate.

LIBRARY

`RS232.LIB`

SEE ALSO

`serXgetc, serXpeek, serXputs, serXwrite, cof_serXgetc, cof_serXgets, cof_serXread, cof_serXputc, cof_serXputs, cof_serXwrite, serXclose`

serXparity

```
void serXparity(int parity_mode); /* where X = A|B|C|D */
```

DESCRIPTION

Sets parity mode for channel X. This function is non-reentrant.

Parity generation for 8 bit data can be unusually slow due to the current method for generating high 9th bits. Whenever, a 9th high bit is needed, the UART is disabled for approximately 5 baud times to create a long stop bit that should be recognized by the receiver as a 9th high bit. The long delay is needed if we are using the serial port itself to handle timing for the delay. Creating a shorter delay would the require use of some other timer resource. Additionally, transmitting these long stops interferes with the receiver, since the baud rate is temporarily increased. Thus, 9th bit formats can only be used in half-duplex mode.

PARAMETERS

`parity_mode`	An integer indicating what parity mode to use. It is best to use one of the macros provided:
`PARAM_NOPARITY`	Disables parity handling (default)
`PARAM_OPARITY`	Configures serial port to check/generate for odd parity
`PARAM_EPARITY`	Configures serial port to check/generate for even parity
`PARAM_2STOP`	Configures serial port to generate 2 stop bits

LIBRARY

`RS232.LIB`

serXpeek

`int serXpeek(); /* where X = [A|B|C|D] */`

DESCRIPTION

Returns 1st character in input buffer X, without removing it from the buffer. This function is non-reentrant.

RETURN VALUE

An integer with 1st character in buffer in the low byte
`-1` if the buffer is empty

LIBRARY

`RS232.LIB`

serXputc

`int serXputc(char c); /* where X = A|B|C|D */`

DESCRIPTION

Writes a character to serial port X write buffer. This function is non-reentrant.

PARAMETERS

`c`	Character to write to serial port X write buffer.

RETURN VALUE

`0` if buffer locked or full, `1` if character sent.

LIBRARY

`RS232.LIB`

EXAMPLE

```
main() {     // echoes characters
    int c;
    serAopen(19200);
    while (1) {
        if ((c = serAgetc()) != -1) {
            serAputc(c);
        }
    }
    serAclose();
}
```

serXputs

int serXputs(char* s); /* where X = A|B|C|D */

DESCRIPTION

Calls **serXwrite(s, strlen(s))**. This function is non-reentrant.

PARAMETERS

s	**NULL**-terminated character string to write

RETURN VALUE

The number of characters actually sent from serial port X.

LIBRARY

RS232.LIB

EXAMPLE

```
// writes a null-terminated string of characters, repeatedly
main() {
    const char s[] = "Hello Z-World";
    serAopen(19200);
    while (1) {
        serAputs(s);
    }
    serAclose();
}
```

serXrdFlush

void serXrdFlush(); /* where X = A|B|C|D */

DESCRIPTION

Flushes serial port X input buffer. This function is non-reentrant.

LIBRARY

RS232.LIB

serXrdFree

```
int serXrdFree(); /* where X = A|B|C|D */
```

DESCRIPTION

Calculates the number of characters of unused data space. This function is non-reentrant.

RETURN VALUE

The number of chars it would take to fill input buffer X.

LIBRARY

`RS232.LIB`

serXrdUsed

```
int serXrdUsed(); /* where X = A|B|C|D */
```

DESCRIPTION

Calculates the number of characters ready to read from the serial port receive buffer. This function is non-reentrant.

RETURN VALUE

The number of characters currently in serial port X receive buffer.

LIBRARY

`RS232.LIB`

serXread

`int serXread(void *data, int length, unsigned long tmout);`
`/* where X = A|B|C|D */`

DESCRIPTION

Reads **`length`** bytes from serial port X or until **`tmout`** milliseconds transpires between bytes. The countdown of **`tmout`** does not begin until a byte has been received. A timeout occurs immediately if there are no characters to read. This function is non-reentrant.

PARAMETERS

`data`	Data structure to read from serial port X
`length`	Number of bytes to read
`tmout`	Maximum wait in milliseconds for any byte from previous one

RETURN VALUE

The number of bytes read from serial port X.

LIBRARY

`RS232.LIB`

EXAMPLE

```
// echoes a blocks of characters
main() {
    int n;
    char s[16];
    serAopen(19200);
    while (1) {
        if ((n = serAread(s, 15, 20)) > 0) {
            serAwrite(s, n);
        }
    }
    serAclose();
}
```

serXwrFlush

```
void serXwrFlush(); /* where X = A|B|C|D */
```

DESCRIPTION

Flushes serial port X transmit buffer. This function is non-reentrant.

LIBRARY

RS232.LIB

serXwrFree

```
int serXwrfree(); /* where X = A|B|C|D */
```

DESCRIPTION

Calculates the free space in the serial port transmit buffer. This function is non-reentrant.

RETURN VALUE

The number of characters the serial port transmit buffer can accept before becoming full.

LIBRARY

RS232.LIB

serXwrite

```
int serXwrite(void *data, int length); /* where X = A|B|C|D */
```

DESCRIPTION

Transmits **length** bytes to serial port X. This function is non-reentrant.

PARAMETERS

data	Data structure to write to serial port X.
length	Number of bytes to write

RETURN VALUE

The number of bytes successfully written to the serial port.

LIBRARY

RS232.LIB

EXAMPLE

```
// writes a block of characters, repeatedly
main() {
    const char s[] = "Hello Z-World";
    serAopen(19200);
    while (1) {
        serAwrite(s, strlen(s));
    }
serAclose();
}
```

set

`void set(void *address, unsigned int bit);`

DESCRIPTION

Dynamic C may expand this call inline

Sets specified bit at memory address to 1. bit may be from 0 to 31. This is equivalent to the following expression, but more efficient:

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

PARAMETERS

address	Address of byte containing bits 7-0
bit	Bit location where 0 represents the least significant bit

LIBRARY

UTIL.LIB

SEE ALSO

SET

SET

`void SET(void *address, unsigned int bit);`

DESCRIPTION

Dynamic C may expand this call inline

Sets specified bit at memory address to 1. bit may be from 0 to 31. This is equivalent to the following expression, but more efficient:

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

PARAMETERS

`address`	Address of byte containing bits 7-0
`bit`	Bit location where 0 represents the least significant bit

LIBRARY

`UTIL.LIB`

SEE ALSO

`set`

setjmp

`int setjmp(jmp_buf env);`

DESCRIPTION

Store the PC (program counter), SP (stack pointer) and other information about the current state into **env**. The saved information can be restored by executing longjmp.

Typical usage:

```
switch (setjmp(e)) {
   case 0:          // first time
      f();          // try to execute f(), may call longjmp
      break;        // if we get here, f() was successful
   case 1:          // to get here, f() called longjmp
      do exception handling
      break;
   case 2:          // like above, different exception code
      ...
}
f() {
   g()
   ...
}
g() {
   ...
   longjmp(e,2);    // exception code 2, jump to setjmp state
                    // ment, but causes setjmp to return 2,
                    // so execute case 2 in the switch
                    // statement
}
```

PARAMETERS

env Information about the current state

RETURN VALUE

Returns zero if it is executed. After longjmp is executed, the program counter, stack pointer and etc. are restored to the state when **setjmp** was executed the first time. However, this time **setjmp** returns whatever value is specified by the **longjmp** statement.

LIBRARY

SYS.LIB

SEE ALSO

longjmp

SetVectExtern2000

```
unsigned SetVectExtern2000(int priority, void *isr);
```

DESCRIPTION

Sets up the external interrupt table vectors for external interrupts 0 and 1. This function is presently used for Rabbit 2000 microprocessors because of the way they handle interrupts. Once this function is called, both interrupts 0 and 1 should be enabled with priority 3; the actual priority used by the interrupt service routine is passed to this function.

PARAMETERS

priority	Priority the ISR should run at. Valid values are 1–3.
isr	ISR handler address. Must be a root address.

RETURN VALUE

Address of vector table entry, or zero if the priority is not valid.

LIBRARY

SYS.LIB

SEE ALSO

GetVectExtern2000, SetVectIntern, GetVectIntern

SetVectIntern

```
unsigned SetVectIntern(int vectNum, void *isr);
```

DESCRIPTION

Sets an internal interrupt table entry. All Rabbit interrupts use jump vectors. This function writes a **jp** instruction (0xC3) followed by the 16 bit ISR address. It is perfectly permissible to have ISRs in xmem and do long jumps to them from the vector table. It is even possible to place the entire body of the ISR in the vector table if it is 16 bytes long or less, but this function only sets up jumps to 16 bit addresses.

PARAMETERS

vectNum	Interrupt number: 0–15 are the only valid values.
isr	ISR handler address. Must be a root address.

RETURN VALUE

Address of vector table entry, or zero if **vectnum** is not valid.

LIBRARY

SYS.LIB

SEE ALSO

GetVectExtern2000, SetVectExtern2000, GetVectIntern

sin

```
float sin(float x);
```

DESCRIPTION

Computes the sine of **x**.

PARAMETERS

x	Value to compute

RETURN VALUE

Sine of **x**.

LIBRARY

MATH.LIB

SEE ALSO

sinh, asin, cos, tan

sinh

`float sinh(float x);`

DESCRIPTION

Computes the hyperbolic sine of **`x`**.

PARAMETERS

`x`	Value to compute

RETURN VALUE

The hyperbolic sine of **`x`**.

If **`x`** > 89.8 (approx.), the function returns INF and signals a range error. If **`x`** < – 89.8 (approx.), the function returns –INF and signals a range error.

LIBRARY

`MATH.LIB`

SEE ALSO

`sin, asin, cosh, tanh`

sprintf

```
void sprintf(char *buffer, char *format, ...);
```

DESCRIPTION

This function takes a **format** string (pointed to by **format**), arguments of the format, and output the formatted string to **buffer** (pointed to by **buffer**). The user should make sure that:

- there are enough arguments after **format** to fill in the format parameters in the format string.
- the types of arguments after **format** match the format fields in **format**.
- the buffer is large enough to hold the longest possible formatted string.

The following is a short list of possible format parameters in the format string. For more details, refer to any C language book.

%d decimal integer (expects type int)
%u decimal unsigned integer (expects type unsigned int)
%x hexidecimal integer (expects type signed int or unsigned int)
%s a string (not interpreted, expects type (char *))
%f a float (expects type float)

For example, **sprintf(buffer,"%s=%x","variable x",256);** should put the string **variable x=100** into **buffer**.

This function can be called by processes of different priorities.

PARAMETERS

buffer	Result string of the formatted string.
format	String to be formatted.
...	Format arguments.

LIBRARY

STDIO.LIB

SEE ALSO

printf

sqrt

```
float sqrt(float x);
```

DESCRIPTION

Calculate the square root of **x**.

PARAMETERS

x	Value to compute

RETURN VALUE

The square root of **x**.

LIBRARY

MATH.LIB

SEE ALSO

exp, pow, pow10

strcat

```
char *strcat(char *dst, char *src);
```

DESCRIPTION

Appends one string to another

PARAMETERS

dst	Pointer to location to destination string.
src	Pointer to location to source string.

RETURN VALUE

Pointer to destination string.

LIBRARY

STRING.LIB

SEE ALSO

strncat

strchr

```
char *strchr(char *src, char ch);
```

DESCRIPTION

Scans a string for the first occurrence of a given character.

PARAMETERS

src	String to be scanned.
ch	Character to search

RETURN VALUE

Pointer to the first occurrence of **ch** in **src**.
NULL if **ch** is not found.

LIBRARY

STRING.LIB

SEE ALSO

strrchr, strtok

strcmp

```
int strcmp(char *str1, char *str2)
```

DESCRIPTION

Performs unsigned character by character comparison of two **NULL**-terminated strings.

PARAMETERS

str1	Pointer to string 1.
str2	Pointer to string 2.

RETURN VALUE

< 0	if **str1** is less than **str2** char in **str1** is less than corresponding char in **str2** **str1** is shorter than but otherwise identical to **str2**
= 0	**str1** is identical to **str2**
> 0	if **str1** is greater than **str2** char in **str2** is greater than corresponding char in **str2** **str2** is shorter than but otherwise identical to **str1**

LIBRARY

STRING.LIB

SEE ALSO

strncmp, strcmpi, strncmpi

strcmpi

```
int *strcmpi(char *str1, char *str2);
```

DESCRIPTION

Performs case-insensitive unsigned character by character comparison of two null terminated strings.

PARAMETERS

str1	Pointer to string 1.
str2	Pointer to string 2.

RETURN VALUE

< 0	if **str1** is less than **str2** char in **str1** is less than corresponding char in **str2** **str1** is shorter than but otherwise identical to **str2**
= 0	**str1** is identical to **str2**
> 0	if **str1** is greater than **str2** char in **str2** is greater than corresponding char in **str2** **str2** is shorter than but otherwise identical to **str1**

LIBRARY

STRING.LIB

SEE ALSO

strncmpi, strncmp, strcmp

strcpy

```
char *strcpy(char *dst, char *src);
```

DESCRIPTION

Copies one string into another string including the **NULL** terminator.

PARAMETERS

dst	Pointer to location to receive string.
src	Pointer to location to supply string.

RETURN VALUE

Pointer to destination string.

LIBRARY

STRING.LIB

SEE ALSO

strncpy

strcspn

```
unsigned int strcspn(char *s1, char *s2);
```

DESCRIPTION

Scans a string for the occurrence of any of the characters in another string.

PARAMETERS

s1	String to be scanned.
s2	Character occurrence string.

RETURN VALUE

Returns the position (less one) of the first occurrence of a character in **s1** that matches any character in **s2**.

LIBRARY

STRING.LIB

SEE ALSO

strchr, strrchr, strtok

strlen

```
int strlen(char *s);
```

DESCRIPTION

Calculate the length of a string.

PARAMETERS

s	Character string

RETURN VALUE

Number of bytes in a string.

LIBRARY

STRING.LIB

strncat

```
char *strncat(char *dst, char *src, unsigned int n);
```

DESCRIPTION

Appends one string to another up to and including the **NULL** terminator or until **n** characters are transferred, followed by a **NULL** terminator.

PARAMETERS

dst	Pointer to location to receive string.
src	Pointer to location to supply string.
n	Maximum number of bytes to copy. If equal to zero, this function has no effect.

RETURN VALUE

Pointer to destination string.

LIBRARY

STRING.LIB

SEE ALSO

strcat

strncmp

int strncmp(char *str1, char *str2, n)

DESCRIPTION

Performs unsigned character by character comparison of two strings of length **n**.

PARAMETERS

str1	Pointer to string 1.
str2	Pointer to string 2.
n	Maximum number of bytes to compare. If zero, both strings are considered equal.

RETURN VALUE

< 0	if **str1** is less than **str2** char in **str1** is less than corresponding char in **str2**
= 0	if **str1** is identical to **str2**
> 0	if **str1** is greater than **str2** char in **str2** is greater than corresponding char in **str2**

LIBRARY

STRING.LIB

SEE ALSO

strcmp, strcmpi, strncmpi

strncmpi

`int strncmpi(char *str1, char *str2, unsigned n)`

DESCRIPTION

Performs case-insensitive unsigned character by character comparison of two strings of length **`n`**.

PARAMETERS

`str1`	Pointer to string 1.
`str2`	Pointer to string 2.
`n`	Maximum number of bytes to compare, if zero then strings are considered equal

RETURN VALUE

`< 0`	if **`str1`** is less than **`str2`** char in **`str1`** is less than corresponding char in **`str2`**
`= 0`	if **`str1`** is identical to **`str2`**
`> 0`	if **`str1`** is greater than **`str2`** char in **`str2`** is greater than corresponding char in **`str2`**

LIBRARY

`STRING.LIB`

SEE ALSO

`strcmpi, strcmp, strncmp`

strncpy

```
char *strncpy(char *dst, char *src, unsigned int n);
```

DESCRIPTION

Copies a given number of characters from one string to another and padding with **NULL** characters or truncating as necessary.

PARAMETERS

dst	Pointer to location to receive string.
src	Pointer to location to supply string.
n	Maximum number of bytes to copy. If equal to zero, this function has no effect.

RETURN VALUE

Pointer to destination string.

LIBRARY

STRING.LIB

SEE ALSO

strcpy

strpbrk

```
char *strpbrk(char *s1, char *s2);
```

DESCRIPTION

Scans a string for the first occurrence of any character from another string.

PARAMETERS

s1	String to be scanned.
s2	Character occurrence string.

RETURN VALUE

Pointer pointing to the first occurrence of a character contained in **s2** in **s1**. Returns **NULL** if not found.

LIBRARY

STRING.LIB

SEE ALSO

strchr, strrchr, strtok

strrchr

```
char *strrchr(char *s, int c);
```

DESCRIPTION

Similar to **strchr**, except this function searches backward from the end of **s** to the beginning.

PARAMETERS

s	String to be searched
c	Search character

RETURN VALUE

Pointer to last occurrence of **c** in **s**. If **c** is not found in **s**, return **NULL**.

LIBRARY

STRING.LIB

SEE ALSO

strchr, strcspn, strtok

strspn

```
size_t strspn(char *src, char *brk);
```

DESCRIPTION

Scans a string for the first segment in **src** containing only characters specified in **brk**.

PARAMETERS

src	String to be scanned
brk	Set of characters

RETURN VALUE

Returns the length of the segment.

LIBRARY

STRING.LIB

strstr

```
char *strstr(char *s1, char *s2);
```

DESCRIPTION

Finds a substring specified by **s2** in string **s1**.

PARAMETERS

s1	String to be scanned
s2	Substring

RETURN VALUE

Pointer pointing to the first occurrence of substring **s2** in **s1**. Returns **NULL** if **s2** is not found in **s1**.

LIBRARY

STRING.LIB

SEE ALSO

strcspn, strrchr, strtok

strtod

```
float strtod(char *s, char **tailptr);
```

DESCRIPTION

ANSI String to Float Conversion.

PARAMETERS

s	String to convert
tailptr	Pointer to a pointer of character. The next conversion may resume at the location specified by ***tailptr**.

RETURN VALUE

The float number.

LIBRARY

STRING.LIB

SEE ALSO

atof

strtok

`char *strtok(char *src, char *brk);`

DESCRIPTION

Scans **`src`** for tokens separated by delimiter characters specified in **`brk`**.

First call with non-**`NULL`** for **`src`**. Subsequent calls with **`NULL`** for **`src`** continue to search tokens in the string. If a token is found (i.e., delineators found), replace the first delimiter in **`src`** with a **`NULL`** terminator so that **`src`** points to a proper **`NULL`**-terminated token.

PARAMETERS

`src`	String to be scanned, must be in SRAM, cannot be a constant. In contrast, strings initialized when they are declared are stored in Flash Memory, and are treated as constants.
`brk`	Character delimiter

RETURN VALUE

Pointer to a token. If no delimiter (therefore no token) is found, returns **`NULL`**.

LIBRARY

`STRING.LIB`

SEE ALSO

`strchr, strrchr, strstr, strcspn`

strtol

long strtol(char *sptr, char **tailptr, int base);

DESCRIPTION

ANSI String to Long Conversion.

PARAMETERS

sptr	String to convert
tailptr	Assigned the last position of the conversion. The next conversion may resume at the location specified by ***tailptr**.
base	Indicates the radix of conversion.

RETURN VALUE

The long integer.

LIBRARY

STRING.LIB

SEE ALSO

atoi, atol

_sysIsSoftReset

void _sysIsSoftReset();

DESCRIPTION

This function determines whether this restart of the board is due to a software reset from Dynamic C or a call to **forceReset()**. If it was a soft reset, this function then does the following:

Calls **_prot_init()** to initialize the protected variable mechanisms. It is up to the user to initialize protected variables.

Calls **sysResetChain()**. The user my attach functions to this chain to perform additional startup actions (for example, initializing protected variables). If a soft reset did not take place, this function calls **_prot_recover()** to recover any protected variables.

LIBRARY

SYS.LIB

sysResetChain

```
void sysResetChain ( void );
```

DESCRIPTION

This is a function chain that should be used to initialize protected variables. By default, it's empty.

LIBRARY

SYS.LIB

tan

```
float tan(float x);
```

DESCRIPTION

Compute the tangent of the argument.

PARAMETERS

x	Value to compute

RETURN VALUE

Returns the tangent of **x**, where $-8 \times PI \leq$ **x** $\leq +8 \times PI$. If **x** is out of bounds, the function returns 0 and signals a domain error. If the value of **x** is too close to a multiple of 90° (PI/2) the function returns INF and signals a range error.

LIBRARY

MATH.LIB

SEE ALSO

atan, cos, sin, tanh

tanh

```
float tanh(float x);
```

DESCRIPTION

Computes the hyperbolic tangent of argument.

PARAMETERS

x	Value to compute

RETURN VALUE

Returns the hyperbolic tangent of **x**. If **x** > 49.9 (approx.), the function returns INF and signals a range error. If **x** < –49.9 (approx.), the function returns –INF and signals a range error.

LIBRARY

MATH.LIB

SEE ALSO

atan, cosh, sinh, tan

tm_rd

```
int tm_rd(struct tm *t);
```

DESCRIPTION

Reads the current system time into the structure **t**. WARNING: The variable **SEC_TIMER** is initialized when a program is first started. If you change the Real Time Clock (RTC), this variable will not be updated until you restart a program, and the **tm_rd** function will not return the time that the RTC has been reset to. The **read_rtc** function will read the actual RTC and can be used if necessary.

PARAMETERS

t Address of structure to store time data

```
struct tm {
    char tm_sec;        // seconds 0-59
    char tm_min;        // 0-59
    char tm_hour;       // 0-23
    char tm_mday;       // 1-31
    char tm_mon;        // 1-12
    char tm_year;       // 80-147 (1980-2047)
    char tm_wday;       // 0-6 0==Sunday
};
```

RETURN VALUE

0 if successful,
-1 if clock read failed.

LIBRARY

RTCLOCK.LIB

SEE ALSO

mktm, mktime, tm_wr

tm_wr

```
int tm_wr(struct tm *t);
```

DESCRIPTION

Sets the system time from a **tm** struct. It is important to note that although **tm_rd()** reads the **SEC_TIMER** variable, not the RTC, **tm_wr()** writes to the RTC directly, and **SEC_TIMER** is not changed until the program is restarted. The reason for this is so that the **DelaySec()** function continues to work correctly after setting the system time. To make **tm_rd()** match the new time written to the RTC without restarting the program, the following should be done:

```
tm_wr(tm);
SEC_TIMER = mktime(tm);
```

But this could cause problems if a **waitfor(DelaySec(n))** is pending completion in a cooperative multitasking program or if the **SEC_TIMER** variable is being used in another way the user, so user beware.

PARAMETERS

t Pointer to structure to read date and time from.

```
struct tm {
    char tm_sec;        // seconds 0-59
    char tm_min;        // 0-59
    char tm_hour;       // 0-23
    char tm_mday;       // 1-31
    char tm_mon;        // 1-12
    char tm_year;       // 80-147 (1980-2047)
    char tm_wday;       // 0-6 0==Sunday
};
```

RETURN VALUE

0 if successful,
-1 if clock write failed.

LIBRARY

RTCLOCK.LIB

SEE ALSO

mktm, mktime, tm_rd

tolower

```
int tolower(int c);
```

DESCRIPTION

Convert alphabetic character to lower case.

PARAMETERS

c	Character to convert

RETURN VALUE

Lower case alphabetic character.

LIBRARY

STRING.LIB

SEE ALSO

toupper, isupper, islower

toupper

```
int toupper(int c);
```

DESCRIPTION

Convert alphabetic character to uppercase.

PARAMETERS

c	Character to convert

RETURN VALUE

Upper case alphabetic character.

LIBRARY

STRING.LIB

SEE ALSO

tolower, isupper, islower

updateTimers

void updateTimers();

DESCRIPTION

Updates the values of **TICK_TIMER**, **MS_TIMER**, and **SEC_TIMER** while running off the 32kHz oscillator. Since the periodic interrupt is disabled when running at 32kHz, these values will not updated unless this function is called.

LIBRARY

`SYS.LIB`

SEE ALSO

`useMainOsc, use32HzOsc`

use32HzOsc

void use32kHzOsc();

DESCRIPTION

Sets the Rabbit processor to use the 32kHz real time clock oscillator for both the CPU and peripheral clock, and shuts off the main oscillator. If this is already set, there is no effect. This mode should provide greatly reduced power consumption. Serial communications will be lost since typical baud rates cannot be made from a 32kHz clock. Also note that this function disables the periodic interrupt, so waitfor and related statements will not work properly (although costatements in general will still work). In addition, the values in **TICK_TIMER**, **MS_TIMER**, and **SEC_TIMER** will not be updated unless you call the function **updateTimers()** frequently in your code. In addition, you will need to call **hitwd()** periodically to hit the hardware watchdog timer since the periodic interrupt normally handles that, or disable the watchdog timer before calling this `function.` The watchdog can be disabled with **Disable_HW_WDT()**.

use32kHzOsc() is not task reentrant.

LIBRARY

`SYS.LIB`

SEE ALSO

`useMainOsc, useClockDivider, updateTimers`

useClockDivider

```
void useClockDivider();
```

DESCRIPTION

Sets the Rabbit processor to use the main oscillator divided by 8 for the CPU (but not the peripheral clock). If this is already set, there is no effect. Because the peripheral clock is not affected, serial communications should still work. This function also enables the periodic interrupt in case it was disabled by a call to **`user32kHzOsc()`**. This function is not task reentrant.

LIBRARY

`SYS.LIB`

SEE ALSO

`useMainOsc, use32HzOsc`

useMainOsc

```
void useMainOsc();
```

DESCRIPTION

Sets the Rabbit processor to use the main oscillator for both the CPU and peripheral clock. If this is already set, there is no effect. This function also enables the periodic interrupt in case it was disabled by a call to **`user32kHzOsc()`**, and updates the **`TICK_TIMER`**, **`MS_TIMER`**, and **`SEC_TIMER`** variables from the real-time clock. This function is not task reentrant.

LIBRARY

`sys.lib`

SEE ALSO

`use32HzOsc, useClockDivider`

utoa

```
char *utoa(unsigned value, char *buf);
```

DESCRIPTION

Places up to 5 digit character string at **`*buf`** representing value of unsigned number. Suppresses leading zeros, but leaves one zero digit for value = 0. Max = 65535. 73 program bytes.

PARAMETERS

`value`	16-bit number to convert
`buf`	Character string of converted number

RETURN VALUE

Pointer to **`NULL`** at end of string.

LIBRARY

`STDIO.LIB`

SEE ALSO

`itoa, htoa, ltoa`

VdGetFreeWd

int VdGetFreeWd(char count);

DESCRIPTION

Returns a free virtual watchdog and initializes that watchdog so that the virtual driver begins counting it down from **count**. The number of virtual watchdogs available is determined by **N_WATCHDOG**, which is 5 by default, but can be defined by the user: **#define N_WATCHDOG 10**. The virtual driver is called every 0.00048828125 sec. On every 128th call to it (62.5 ms), the virtual watchdogs are counted down. If any virtual watchdog reaches 0, this is a fatal error. Once a virtual watchdog is active, it should reset periodically with a call to **VdHitWd** to prevent this. The count is decremented, tested and, if 0, a fatal error occurs.

PARAMETERS

count	1 < **count** <= 255

RETURN VALUE

Integer id number of an unused virtual watchdog timer.

LIBRARY

VDRIVER.LIB

VdHitWd

int VdHitWd(int ndog);

DESCRIPTION

Resets virtual watchdog counter to N counts where N is the argument to the call to **VdGetFreeWd()** that obtained the virtual watchdog **ndog**. The virtual driver counts down watchdogs every 62.5 ms. If a virtual watchdog reaches 0, this is a fatal error. Once a virtual watchdog is active it should reset periodically with a call to **VdHitWd** to prevent this. If count = 2 the **VdHitWd** will need to be called again for virtual watchdog **ndog** within 62.5 ms. If count = 255, **VdHitWd** will need to be called again for virtual watchdog **ndog** within 15.9375 seconds.

PARAMETERS

ndog	Id of virtual watchdog returned by **VdGetFreeWd()**

LIBRARY

VDRIVER.LIB

VdInit

```
void VdInit(void);
```

DESCRIPTION

Initializes virtual driver for all Rabbit boards. Supports **DelayMs**, **DelaySec**, **DelayTick**. **VdInit** is called by the BIOS unless disabled.

LIBRARY

VDRIVER.LIB

VdReleaseWd

`int VdReleaseWd(int ndog);`

DESCRIPTION

Deactivates a virtual watchdog and makes it available for **`VdGetFreeWd()`**.

PARAMETERS

`ndog`	Handle returned by **`VdGetFreeWd`**

RETURN VALUE

`0` - **`ndog`** out of range
`1` - success

LIBRARY

VDRIVER.LIB

EXAMPLE

```
// VdReleaseWd virtual watchdog example
main() {
   int wd;                // handle for a virtual watchdog
   unsigned long tm;
   tm = SEC_TIMER;
   wd = VdGetFreeWd(255);// wd activated, 9 virtual watchdogs now
available

                          // wd must be hit at least every 15.875
seconds

    while(SEC_TIMER - tm < 60) {     // let it run for a minute
        VdHitWd(wd);// decrements counter corresponding to wd
reset to 12
    }
   VdReleaseWd(wd);           // now there are 10 virtual
                              //  watchdogs available
}
```

WriteFlash2

int WriteFlash2(unsigned long flashDst, void* rootSrc, int len);

DESCRIPTION

Write len bytes to physical address flashDst on the 2nd

flash device from rootSrc. The source must be in root. The flashDstaddress must be in the range 0x00040000-0x0007FFFF, since the topmost memory quadrant will be mapped to the 2nd flash (256kb is the maximum size visible on the second flash by this function). This function is not reentrant.

NOTE: this function should NOT be used if you are using the second flash device for a flash file system , e.g. if you are writing a TCP/IP-based application!

PARAMETERS

flashDst	Physical address of the flash destination
rootSrc	Pointer to the root source
len	Number of bytes to write

RETURN VALUE

```
 0: Success
-1: Attempt to write non-2nd flash area, nothing written
-2: Rootsrc not in root
-3: Timeout while writing flash
```

LIBRARY

```
XMEM.LIB
```

write_rtc

void write_rtc(unsigned long int time);

DESCRIPTION

Writes a 32 bit seconds value to the RTC, zeros other bits. This function does not stop or delay periodic interrupt. It does not affect the **SEC_TIMER** or **MS_TIMER** variables.

PARAMETERS

time	32-bit value representing the number of seconds since January 1, 1980.

LIBRARY

RTCLOCK.C

SEE ALSO

read_rtc

WrPortE

void WrPortE(int port, char *portshadow, int data_value);

DESCRIPTION

Writes an external I/O register with 8 bits and updates shadow for that register. The variable names must be of the form **port** and **portshadow** for the most efficient operation. A **NULL** pointer may be substituted if shadow support is not desired or needed.

PARAMETERS

port	Address of external data register.
portshadow	Reference pointer to a variable shadowing the register data. Substitute with **NULL** pointer (or 0) if shadowing is not required.
data_value	Value to be written to the data register

LIBRARY

SYSIO.LIB

SEE ALSO

RdPortI, BitRdPortI, WrPortI, BitWrPortI, RdPortE, BitRdPortE, BitWrPortE

WrPortI

void WrPortI(int port, char *portshadow, int data_value);

DESCRIPTION

Writes an internal I/O register with 8 bits and updates shadow for that register.

PARAMETERS

port	Address of data register.
portshadow	Reference pointer to a variable shadowing the register data. Substitute with **NULL** pointer (or 0) if shadowing is not required.
data_value	Value to be written to the data register

LIBRARY

SYSIO.LIB

SEE ALSO

RdPortI, BitRdPortI, BitRdPortE, BitWrPortI, RdPortE, WrPortE, BitWrPortE

xalloc

long xalloc(long sz)

DESCRIPTION

Allocates the specified number of bytes in extended memory.

PARAMETERS

sz	Number of bytes to allocate.

RETURN VALUE

The 20-bit physical address of the allocated data on success;
0 on failure.

LIBRARY

SYS.LIB

SEE ALSO

root2xmem, xmem2root

xmem2root

int xmem2root(void *dest, unsigned long int src, unsigned int len);

DESCRIPTION

Stores **len** characters from physical address **src** to logical address **dest**.

PARAMETERS

dest	Logical address
src	Physical address
len	Numbers of bytes

RETURN VALUE

0 - success
1 - attempt to write Flash Memory area, nothing written
2 - destination not all in root

LIBRARY

XMEM.LIB

SEE ALSO

root2xmem, xalloc

xmem2xmem

```
int xmem2xmem(unsigned long dest, unsigned long src, unsigned
   len);
```

DESCRIPTION

Stores **len** characters from physical address **src** to physical address **dest**.

PARAMETERS

dest	Physical address of destination
src	Physical address of source data
len	Length of source data in bytes

RETURN VALUE

0 - success
1 - attempt to write Flash Memory area, nothing written

LIBRARY

XMEM.LIB

User Interface 16

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

Multiple instances of Dynamic C may be run simultaneously. This means multiple debugging sessions are possible over different serial ports. This is useful for debugging multiple boards that are communicating among themselves

Programs can compile directly to a target controller for debugging in RAM or flash. Programs can also be compiled to a `.bin` file.

In order to compile or run a program, a controller must be connected to the PC. Dynamic C includes editing options and compiler options. Most of the options are in the **OPTIONS** menu.

16.1 Editing

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

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

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

16.1.0.1 Arrows

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

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

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

16.1.0.2 Home

Moves the cursor *backward* in the text to the start of the line.

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

16.1.0.3 End

Moves the cursor *forward* in the text.

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

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

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

16.2 Menus

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

Click the menu title or press **<ALT-F>** to select the **FILE** menu.

16.2.1 New

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

16.2.2 Open

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

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

16.2.3 Save

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

Use the **Save** command often while editing to protect against loss during power failures or system crashes.

16.2.4 Save As

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

16.2.5 Close

Closes the active window. The active window may also be closed by pressing **<CTRL-F4>** or by double-clicking on its system menu. If there is an attempt to close a file before it has been saved, Dynamic C will present a dialog similar to one of these two dialogs.

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

16.2.6 Print Preview

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

16.2.7 Print

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

At present, printing all pages is the only option.

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.

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

To choose a printer, click the **Setup** button in the Print dialog, or choose the **Print Setup...** command from the **FILE** menu.

16.2.8 Print Setup

Allows choice of which printers to use and to set them up to print text.

There is a choice between using the computer system's default printer or selecting a specific printer. Depending on the printer selected, it may be possible to specify paper orientation (portrait or tall, vs. landscape or wide), and paper size. Most printers have these options. A specific printer may or may not have more than one paper source.

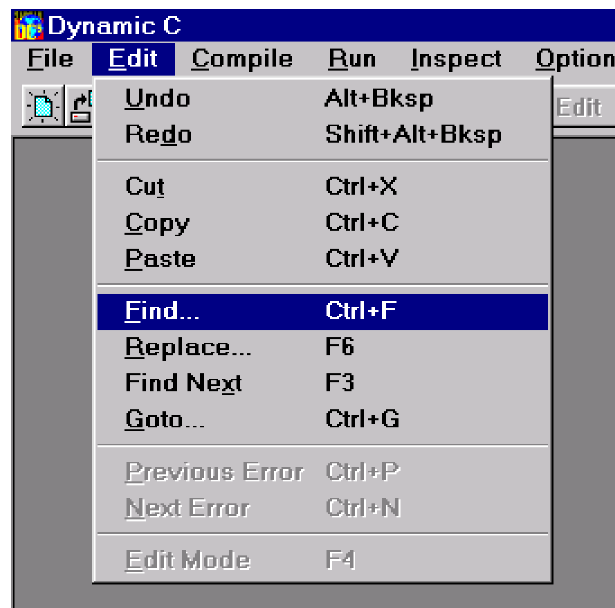
The **Options** button allows the print options dialog to be displayed for a specific printer. The **Network** button allows printers to be added or removed from the list of printers.

16.2.9 Exit

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

16.3 Edit Menu

Click the menu title or press **<ALT-E>** to select the **EDIT** menu.



16.3.1 Undo

This option undoes recent changes in the active edit window. The command may be repeated several times to undo multiple changes. The amount of editing that may be undone will vary with the type of operations performed, but should suffice for a few large cut and paste operations or many

lines of typing. Dynamic C discards all undo information for an edit window when the file is saved. The keyboard shortcut is **<ALT-backspace>**.

16.3.2 Redo

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

16.3.3 Cut

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

16.3.4 Copy

Makes a copy of selected text in a file or in one of the debugging windows. The copy of the text is saved on the "clipboard." The contents of the clipboard may be pasted virtually anywhere. The keyboard shortcut is **<CTRL-C>**.

16.3.5 Paste

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

16.3.6 Find

Finds specified text.

Type the text to be found in the **Find** box. The **Find** command (and the **Find Next** command, too) will find occurrences of the word "switch." If **case sensitive** is clicked, the search will find occurrences that match exactly. Otherwise, the search will find matches having upper- and lowercase letters. For example, "switch," "Switch," and "SWITCH" would all match. If **reverse** is clicked the search will occur in reverse, that is, the search will proceed toward the beginning of the file, rather than toward the end of the file. Use the **From cursor** checkbox to choose whether to search the entire file or to begin at the cursor location. The keyboard shortcut is **<CTRL F>**.

16.3.7 Replace

Replaces specified text.

Type the text to be found in the **Find** text box (there is a pulldown list of previously entered strings). Then type the text to substitute in the **Change to** text box. If **Case sensitive** is selected, the search will find an occurrence that matches exactly. Otherwise, the search will find a match having upper- and lower-case letters. For example, "reg7," "REG7," and "Reg7" all match.

If **Reverse** is clicked, the search will occur in reverse, that is, the search will proceed toward the beginning of the file, rather than toward the end of the file. The entire file may be searched from

the current cursor location by clicking the **From cursor** box, or the search may begin at the current cursor location.

The **Selection only** box allows the substitution to be performed only within the currently selected text. Use this in conjunction with the **Change All** button. This box is disabled if no text is selected.

Normally, Dynamic C will find the search text, then prompts for whether to make the change. This is an important safeguard, particularly if the **Change All** button is clicked. If **No prompt** is clicked, Dynamic C will make the change (or changes) without prompting.

The keyboard shortcut for **Replace** is **<F6>**.

16.3.8 Find Next

Once search text has been specified with the **Find** or **Replace** commands, the **Find Next** command (**F3** for short) will find the next occurrence of the same text, searching forward or in reverse, case sensitive or not, as specified with the previous **Find** or **Replace** command. If the previous command was **Replace**, the operation will be a replace.

16.3.9 Goto

Positions the insertion point at the start of the specified line.

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

16.3.10 Previous Error

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

16.3.11 Next Error

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

16.3.12 Edit Mode

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

16.4 Compile Menu

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

16.4.1 Compile to Target

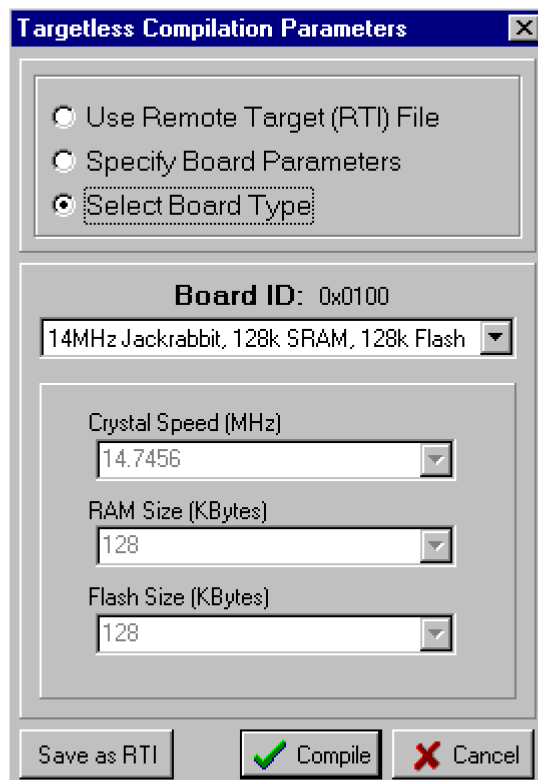
Compiles a program and loads it in the target controller's memory. The keyboard shortcut is **F5**.

Dynamic C determines whether to compile to RAM or flash based on the current compiler options (set with the Options menu). Any compilation errors are listed in the automatically activated message window. Hit **<F1>** to obtain a more descriptive message for any error message that is highlighted in this window.

16.4.2 Compile to .bin file

Compiles a program and writes the image to a **`.bin`** file. The **`.bin`** file can then be used with a device programmer to program multiple chips; or the Rabbit Field Utility can load the **`.bin`** files to the target. The **Include BIOS** option should normally be checked. It just causes the BIOS as well as the user program to be included in the BIN file. If you are creating special program such as a cold loader that starts at address 0x0000, then this option should be unchecked. This type of use is for advanced users.

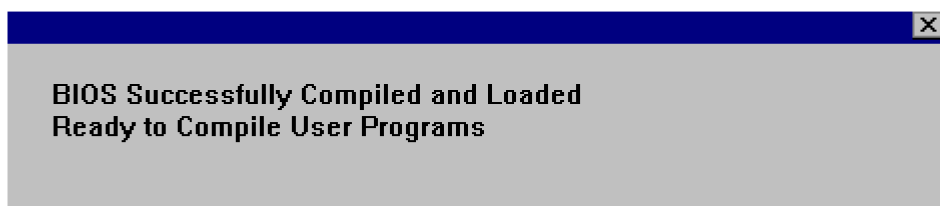
When compiling to a **`.bin`** file, choose **Use attached target** to use the parameters of the controller board connected to your system. If there is no controller board connected to your system or if there is but you want to define a different configuration, choose **Define target configuration**. The Targetless Compilation Parameters menu will appear, as shown below. You can specify board type and parameters and save the information in a Remote Target Information (RTI) file.



16.4.3 Reset Target/Compile BIOS

This option reloads the BIOS to RAM or flash, depending on the BIOS memory setting chosen in **Options->Compiler Options**. The default option is flash.

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



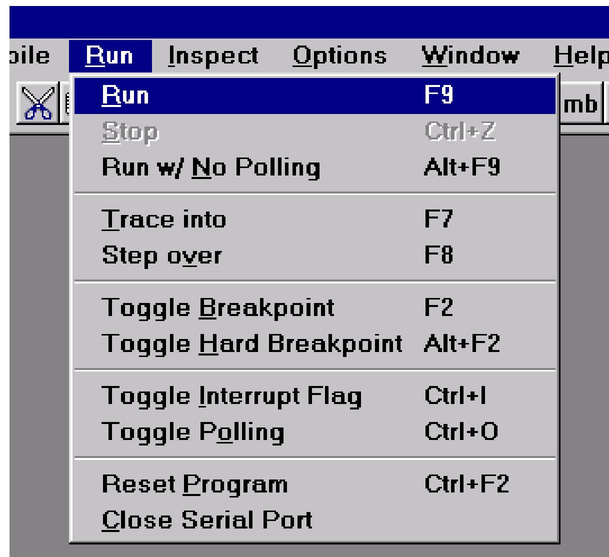
16.4.4 Include Debug Code/RST 28 Instructions

If this is checked, debug code will be included in the program even if **#nodebug** precedes the main function in the program. Debug code consists mainly of **RST 28h** instructions inserted after every C statement. At an **RST 28h** instruction, program execution is transferred to the debug kernel where communication between Dynamic C and the target is tended to before returning to the user program. ***There are certain loop optimizations that are not generated when code is compiled as debug***. 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 **1**, otherwise it is **0**.

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 effects both code compiled to `.bin` files and code compiled to the target . In order to run the program after compiling to the target with this option, disconnect the target from the programming port and reset the target CPU.

16.5 Run Menu

Click the menu title or press **<ALT-R>** to select the **RUN** menu.



16.5.1 Run

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

16.5.2 Run w/ No Polling

This command is identical to the **Run** command, with an important exception. When running in polling mode (**F9**), the development PC polls or interrupts the target system every 100 ms to obtain or send information about target breakpoints, watch lines, keyboard-entered target input, and target output from `printf` statements. Polling creates interrupt overhead in the target, which can be undesirable in programs with tight loops. The **Run w/ No Polling** command allows the program to run without polling and its overhead. (Any `printf` calls in the program will cause execution to pause until polling is resumed. Running without polling also prevents debugging until polling is resumed.) The keyboard shortcut for this command is **<ALT-F9>**.

16.5.3 Stop

The **Stop** command places a hard breakpoint at the point of current program execution. Usually, the compiler cannot stop within ROM code or in `nodebug` code. On the other hand, the target can

be stopped at the `rst 028h` instruction if `rst 028h` assembly code is inserted as inline assembly code in `nodebug` code. However, the debugger will never be able to find and place the execution cursor in `nodebug` code. The keyboard shortcut is **<CTRL-Z>**.

16.5.4 Reset Program

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

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

16.5.5 Trace Into

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

16.5.6 Step over

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

16.5.7 Toggle Breakpoint

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

16.5.8 Toggle Hard Breakpoint

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

16.5.9 Toggle Interrupt Flag

Toggles interrupt state. The keyboard shortcut is **<CTRL-I>**.

16.5.10 Toggle Polling

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

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

16.5.11 Reset Target

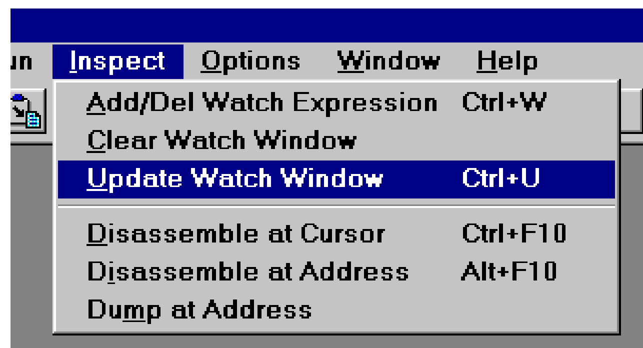
Tells the target system to perform a software reset including system initializations. Resetting a target *always* brings Dynamic C back to edit mode. The keyboard shortcut is **<CTRL-Y>.**

16.5.12 Close Serial Port

Disconnects the programming serial port between PC and target so that the target serial port is accessible to other applications.

16.6 Inspect Menu

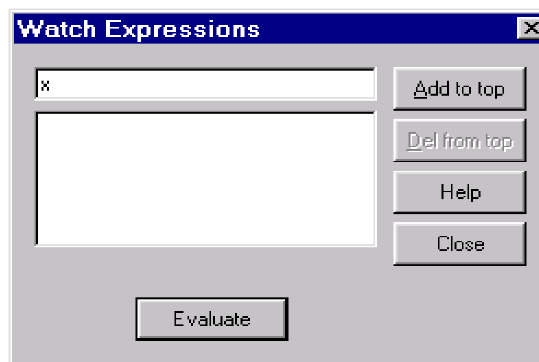
Click the menu title or press **<ALT-I>** to select the **INSPECT** menu.



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

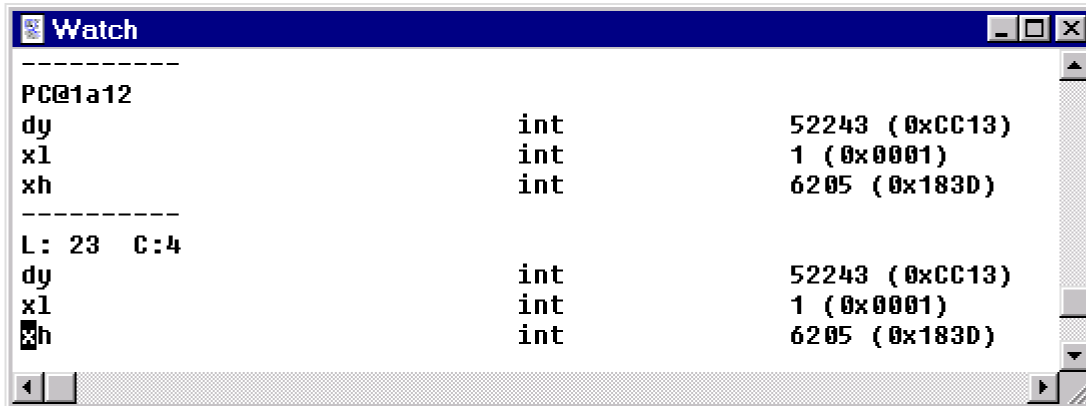
16.6.1 Add/Del Watch Expression

This command provokes Dynamic C to display the following dialog.



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

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



16.6.2 Clear Watch Window

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

16.6.3 Update Watch Window

Forces expressions in the Watch Expression list to be evaluated and displayed in the Watch window only when the function **`runwatch()`** is called from the application program. **`runwatch()`** monitors for watch update requests and should be called periodically if watch expressions are used. Normally the Watch window is updated every time the execution cursor is changed, that is when a single step, a breakpoint, or a stop occurs in the program. The keyboard shortcut is **<CTRL-U>**.

16.6.4 Disassemble at Cursor

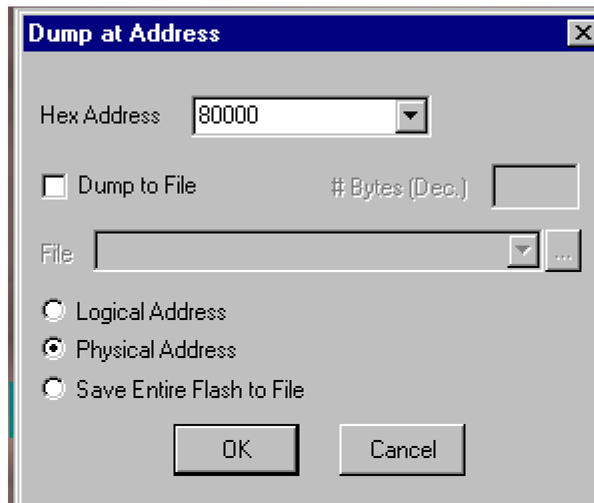
Loads, disassembles and displays the code at the current editor cursor. This command does not work in user application code declared as **`nodebug`**. Also, this command does not stop the execution on the target. The keyboard shortcut is **<CTRL-F10>**.

16.6.5 Disassemble at Address

Loads, disassembles and displays the code at the specified address. This command produces a dialog box that asks for the address at which disassembling should begin. Addresses may be entered in two formats: a 4-digit hexadecimal number that specifies any location in the root space, or a 2-digit page number followed by a colon followed by a 4-digit logical address, from 00 to FF. The keyboard shortcut is **<ALT-F10>**.

16.6.6 Dump at Address

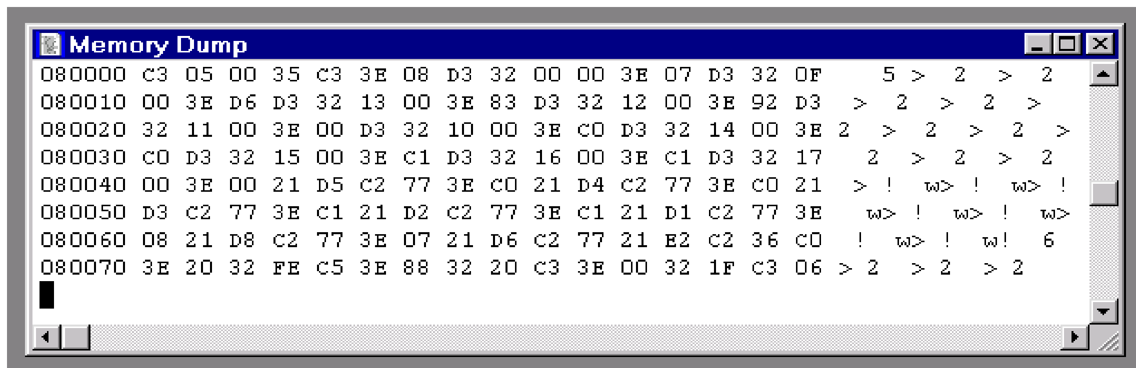
Allows blocks of raw values in any memory location (except the BIOS 0–2000H) to be looked at. Values can be displayed on the screen or written to a file.



The option **Dump 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 0.

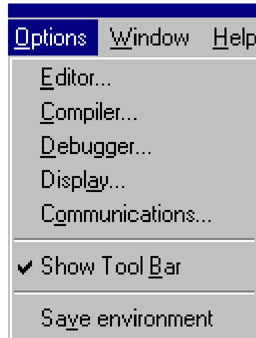
A typical screen display appears below.



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

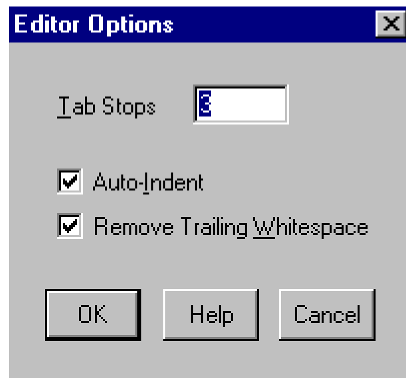
16.7 Options Menu

Click the menu title or press **<ALT-O>** to select the **OPTIONS** menu.



16.7.1 Editor

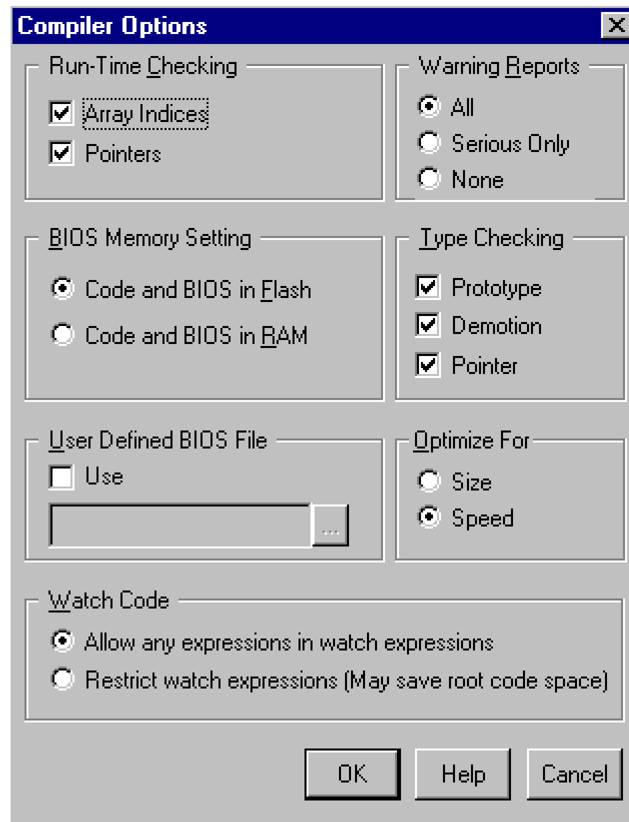
The **Editor** command gets Dynamic C to display the following dialog.



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

16.7.2 Compiler

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



Warning Reports tell 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.

The **Run-Time Checking** options, if checked, cause a fatal error message at run-time. These options increase the amount of code and cause slower execution, but they can be valuable debugging tools. The options are described in below.

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.

16.7.2.1 Optimize For

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

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

16.7.2.2 Type Checking

Prototypes—Performs strict type checking of arguments of function calls against the function prototype. The number of arguments passed must match the number of parameters in the prototype. In addition, the types of arguments must match those defined in the prototype. Z-World recommends prototype checking because it identifies likely run-time problems. To use this feature fully, all functions should have prototypes (including functions implemented in assembly).

Demotion—Detects demotion. A demotion automatically converts the value of a larger or more complex type to the value of a smaller or less complex type. The increasing order of complexity of scalar types is:

```
char
unsigned int
int
unsigned long
long
float
```

A demotion deserves a warning because information may be lost in the conversion. For example, when a **long** variable whose value is 0x10000 is converted to an **int** value, the resulting value is 0. The high-order 16 bits are lost. An explicit type casting can eliminate demotion warnings. All demotion warnings are considered non-serious as far as warning reports are concerned.

Pointer—Generates warnings if pointers to different types are intermixed without type casting. While type casting has no effect in straightforward pointer assignments of different types, type casting does affect pointer arithmetic and pointer dereferences. All pointer warnings are considered non-serious as far as warning reports are concerned.

16.7.2.3 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_** or **_RAM_**, depending on which of the following options is selected. This macro is used to determine the relevant sections of code to compile for the corresponding memory type.

Code and BIOS in Flash—If you select this option, the compiler will load the BIOS to flash when cold-booting, and will compile the user program to flash where it will normally reside.

Code and BIOS in RAM—If you select this option, the compiler will load the BIOS to RAM on cold-booting and compile the user program to RAM. This option is useful if you want to use breakpoints while you are debugging your application, but you don’t want interrupts disabled while the debugger writes a breakpoint to flash (this can take 10 ms to 20 ms or more, depending on the flash type used). Note that when you single step through code, the debugger is writing breakpoints at the next point in code you will step to. 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.

16.7.2.4 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.

16.7.2.5 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.

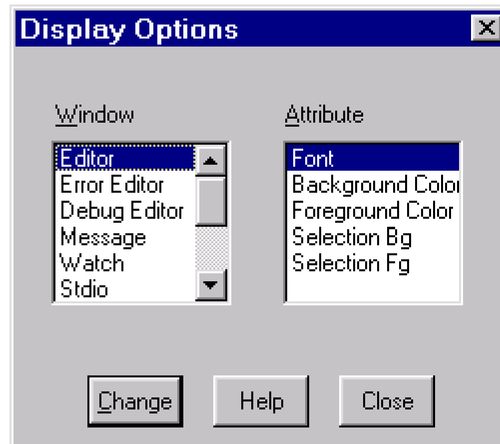
16.7.3 Debugger

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

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

16.7.4 Display

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

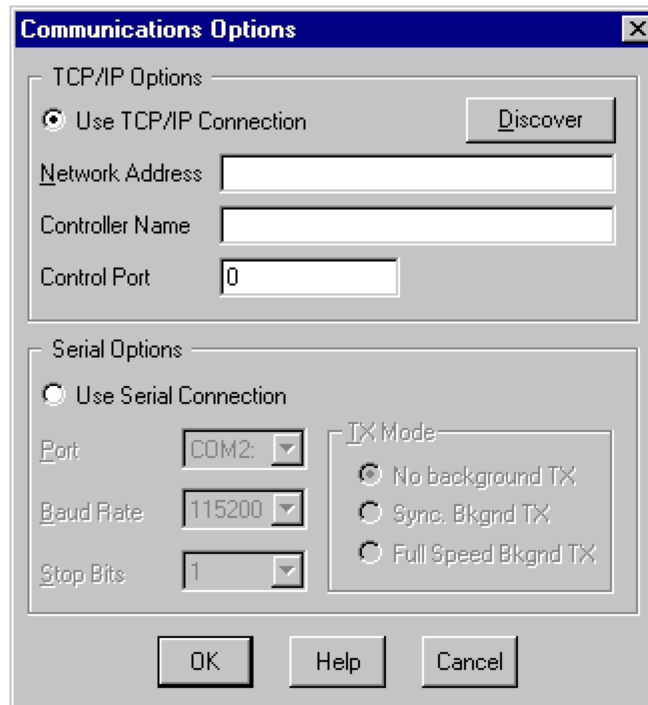


Use the **Display Options** dialog box to change the appearance of Dynamic C windows. First choose the window from the window list. Then select an attribute from the attribute list and click the change button. Another dialog box will appear to make the changes. Note that Dynamic C allows only fixed-pitch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

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

16.7.5 Communications

The **Communications** command displays the following dialog box. Use it to tell Dynamic C how to communicate with the target controller.



16.7.5.1 TCP/IP Option

In order to program and debug a controller across a TCP/IP connection, the **Network Address** field must have the IP Address of the Z-World RabbitLink that is attached to the controller. To accept control commands from Dynamic C, the **Control Port** field must be set to the port used by the RabbitLink. The Controller Name is for informational purposes only. The **Discover** button makes Dynamic C broadcast a query to any RabbitLinks attached to the network. Any RabbitLinks that respond to the broadcast can be selected and their information will be placed in the appropriate fields.

16.7.5.2 Serial Options

The COM port, baud rate, and number of stop bits may be selected. The transmission mode radio buttons also affect communication by controlling the overlap of compilation and downloading. With **No Background TX**, Dynamic C will not overlap compilation and downloading. This is the most reliable mode, but also the slowest—the total compile time is the sum of the processing time and the communication time. With **Full Speed Bkgnd TX**, Dynamic C will almost entirely overlap compilation and downloading. This mode is the fastest, but may result in communication failure. The **Sync. Bkgnd TX** mode provides partial overlap of compilation and downloading. This is the default mode used by Dynamic C.

16.7.6 Show Tool Bar

The **Show Tool Bar** command toggles the display of the tool bar:

Dynamic C remembers the toolbar setting on exit.

16.7.7 Save Environment

The **Save Environment** command gets Dynamic C to update the registry and **DCW.CFG** initialization files immediately with the current options settings. Dynamic C always updates these files on exit. Saving them while working provides an extra measure of security against Windows crashes.

16.8 Window Menu

Click the menu title or press **<ALT-W>** to select the **WINDOW** menu.

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

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

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

16.8.1 Cascade

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

16.8.2 Tile Horizontally

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

16.8.3 Tile Vertically

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

16.8.4 Arrange Icons

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

16.8.5 Message

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

16.8.6 Watch

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

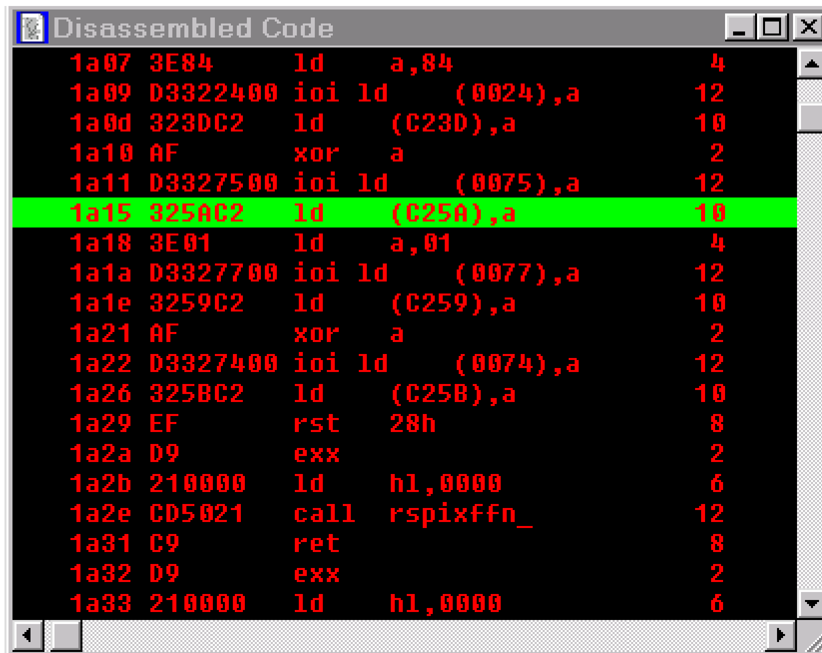
16.8.7 STDIO

Click the **STDIO** command 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, unless another request was made by the programmer. (See the **Debugger Options** under the **OPTIONS** menu.)

16.8.8 Assembly

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

The **Disassemble at Cursor** or **Disassemble at Address** commands also activate the Assembly window.

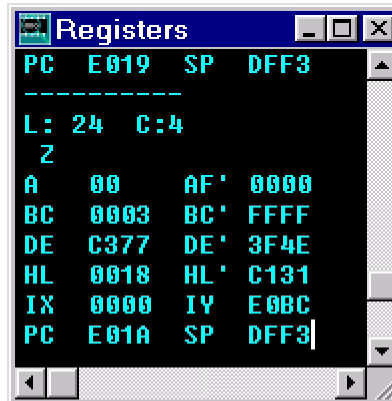


The Assembly window shows the memory address on the far left, followed by the code bytes for the instruction at the address, followed by the mnemonics for the instruction. The last column shows the number of cycles for the instruction, assuming no wait states. The total cycle time for a block of instructions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

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 such as `ldir` or `ret nz` with an indeterminate cycle time was selected.

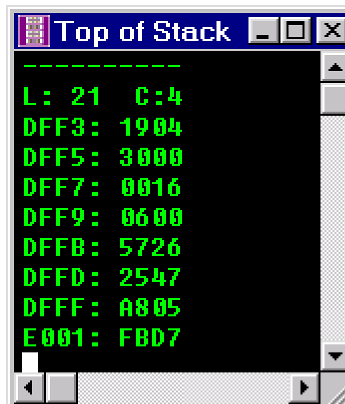
16.8.9 Registers

Click the **Registers** command to activate or deactivate the Register window. The Register window displays the processor register set, including the status register. Letter codes indicate the bits of the status register (F register). The window also shows the source-code line and column at which the register "snapshot" was taken. It is possible to scroll back to see the progression of successive register snapshots. Registers may be changed when program execution is stopped by clicking the right mouse button over the name or value of the register to be changed. Registers PC, XPC, and SP may not be edited as this can adversely effect program flow and debugging.



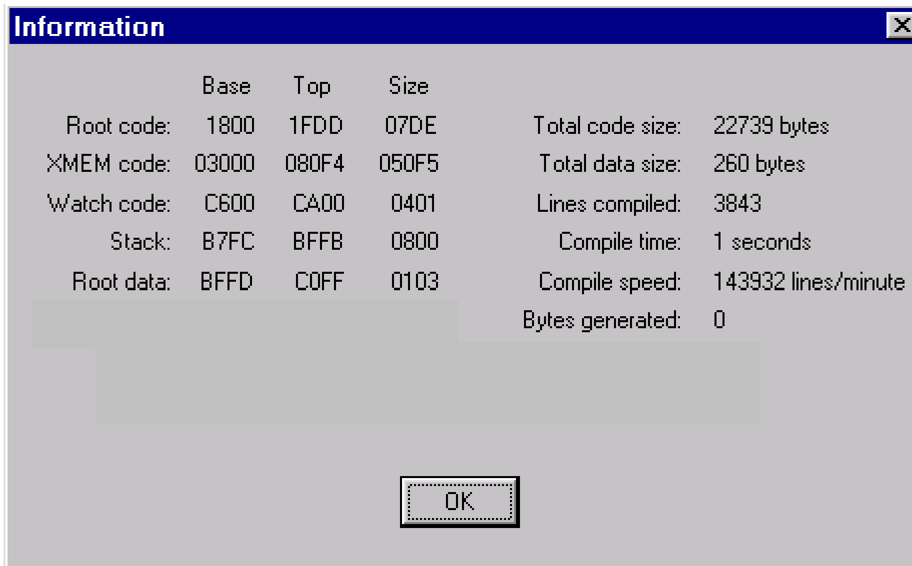
16.8.10 Stack

Click the **Stack** command to activate or deactivate the Stack window. The Stack window displays the top 8 bytes of the run-time stack. It also shows the line and column at which the stack "snapshot" was taken. It is possible to scroll back to see the progression of successive stack snapshots.



16.8.11 Information

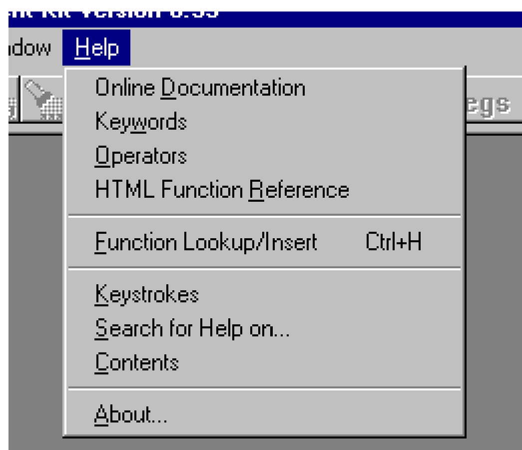
Click the **Information** command to activate the Information window.



The Information window displays how the memory is partitioned and how well the compilation went. In this example, no space has been allocated to the heap or free space.

16.9 Help Menu

Click the menu title or press **<ALT-H>** to select the **HELP** menu.



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

16.9.1 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 points to the Web.

16.9.2 Keywords

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

16.9.3 Operators

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

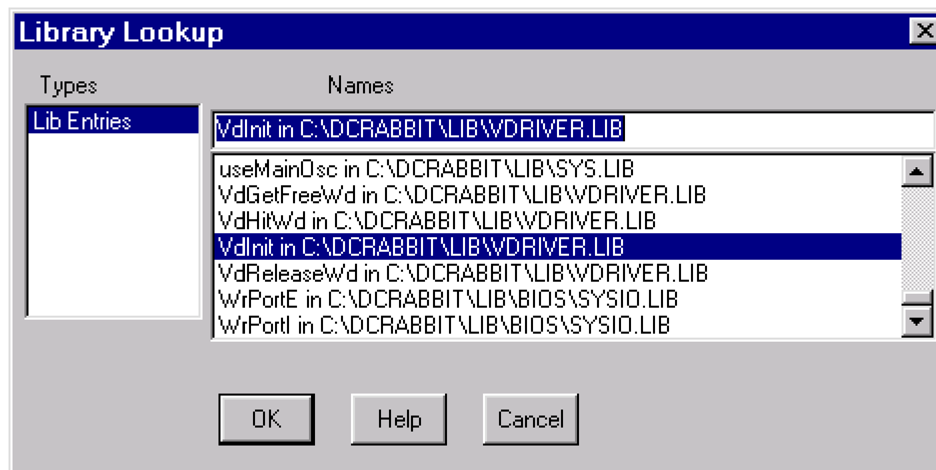
16.9.4 HTML Function Reference

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 this manual.

16.9.5 Function Lookup/Insert

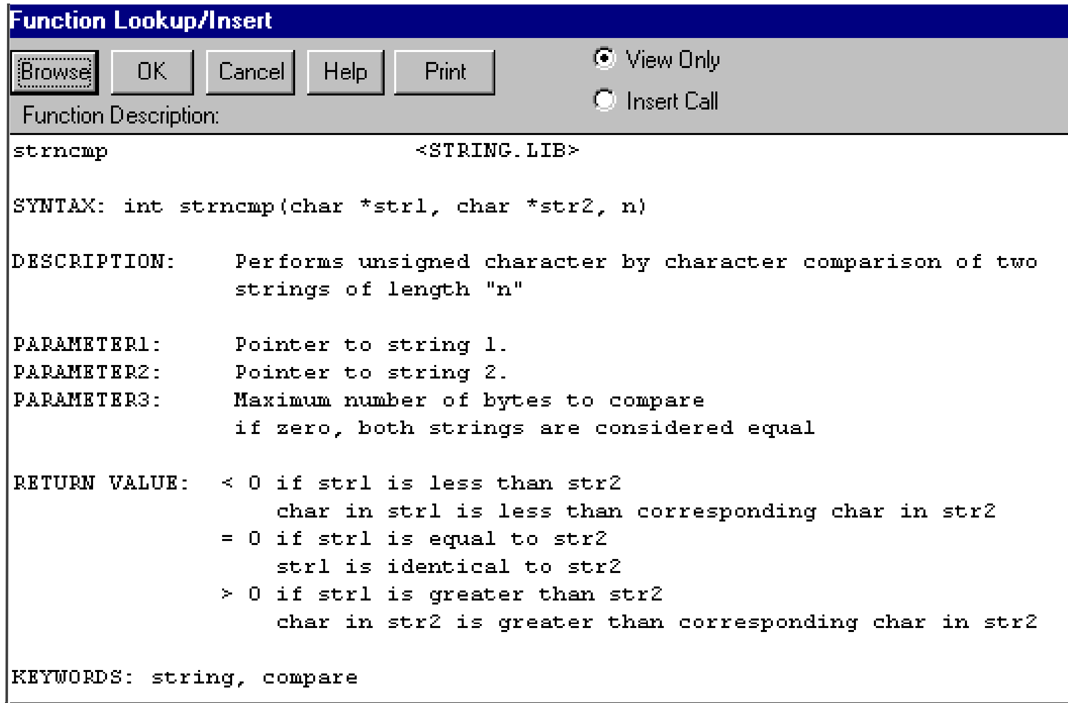
Obtains help information for library functions. When a function name is clicked (or the function name is selected) in source code and then the help command is issued, Dynamic C displays help information for that function. The keyboard shortcut is **<CTRL-H>**.

If Dynamic C cannot find a unique description for the function, it will display the following dialog box.

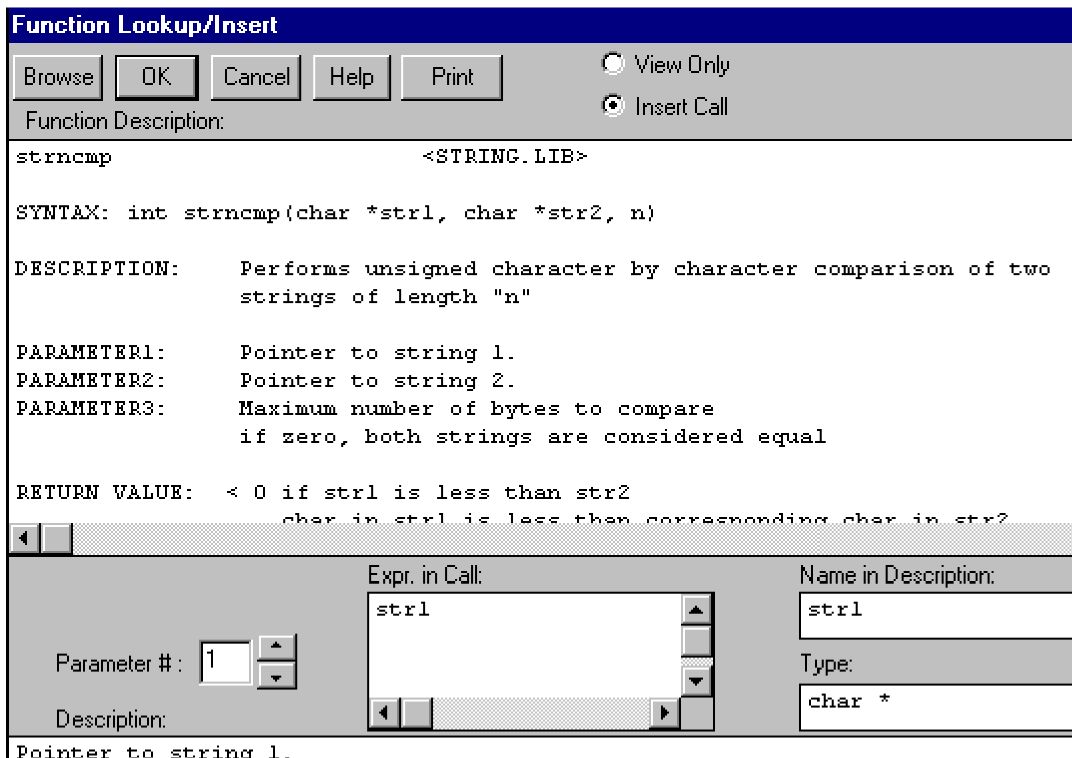


Click **Lib Entries** to display a list of the library functions currently available to the program. (These are the files named in the file **`LIB.DIR`**.) Then select a function name from the list to receive information about that function.

Dynamic C displays a dialog box like this one when a function is selected to display help information.



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



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

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

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

Consider, for example, the following code.

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

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

16.9.6 Keystrokes

Invokes the on-line help system and displays the keystrokes page.

16.9.7 Search for Help on

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

16.9.8 Contents

Invokes the on-line help system and displays the contents page.

16.9.9 About

The **About** command displays the Dynamic C version number and the copyright notice.

µC/OS-II 17

Not available with SE versions of Dynamic C.

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. 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 µC/OS-II function descriptions in Chapter 15 are fully explained in Labrosse's book. It can be purchased at the Z-World store, www.zworld.com/store/home.html, or at http://www.ucos-ii.com/.

17.1 Changes

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

17.1.1 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. To change the default tick rate to 32, do the following:

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

17.1.2 Task Creation

In a µC/OS-II application, stacks are declared as static arrays, and the address of either the top or bottom (depending on the CPU) of the stack is passed to **OSTaskCreate**. In a Rabbit-based system, the Dynamic C development environment provides a superior stack allocation mechanism that µC/OS-II incorporates. Rather than declaring stacks as static arrays, the number of stacks of particular sizes are declared, and when a task is created using either **OSTaskCreate** or **OSTaskCreateExt,** only the size of the stack is passed, not the memory address. This mechanism allows a large number of stacks to be defined without using up root RAM.

There are five macros located in ucos2.lib that define the number of stacks needed of five different sizes. In order to have three 256 byte stacks, one 512 byte stack, two 1024 byte stacks, one 2048 byte stack, and no 4096 byte stacks, the following macro definitions would be used:

```
#define STACK_CNT_256      3    // number of 256 byte stacks
#define STACK_CNT_512      1    // number of 512 byte stacks
#define STACK_CNT_1K       2    // number of 1K stacks
#define STACK_CNT_2K       1    // number of 2K stacks
#define STACK_CNT_4K       0    // number of 4K stacks
```

These macros can be placed into each µC/OS-II application so that the number of each size stack can be customized based on the needs of the application. Suppose that an application needs 5 tasks, and each task has a consecutively larger stack. The macros and calls to **OSTaskCreate** would look as follows

```
#define STACK_CNT_256     2    // number of 256 byte stacks
#define STACK_CNT_512     2    // number of 512 byte stacks
#define STACK_CNT_1K      1    // number of 1K stacks
#define STACK_CNT_2K      1    // number of 2K stacks
#define STACK_CNT_4K      1    // number of 4K stacks

OSTaskCreate(task1, NULL, 256, 0);
OSTaskCreate(task2, NULL, 512, 1);
OSTaskCreate(task3, NULL, 1024, 2);
OSTaskCreate(task4, NULL, 2048, 3);
OSTaskCreate(task5, NULL, 4096, 4);
```

Note that the macro **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_TASKOPTSTK_CHK |
                                         OS_TASKOPTSTK_CLR);
OSTaskCreateExt(task2,NULL,1,1,512,NULL, OS_TASKOPTSTK_CHK |
                                         OS_TASKOPTSTK_CLR);
OSTaskCreateExt(task3,NULL,2,2,1024,NULL, OS_TASKOPTSTK_CHK |
                                         OS_TASKOPTSTK_CLR);
OSTaskCreateExt(task4,NULL,3,3,2048,NULL, OS_TASKOPTSTK_CHK |
                                         OS_TASKOPTSTK_CLR);
OSTaskCreateExt(task5,NULL,4,4,4096,NULL,OS_TASKOPTSTK_CHK |
                                         OS_TASKOPTSTK_CLR);
```

17.1.3 Restrictions

At the time of this writing, uC/OS-II for Dynamic C is not compatible with the use of Dynamic C's 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.

17.2 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.

17.2.1 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 which 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.

17.2.2 Possible ISR Scenarios

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

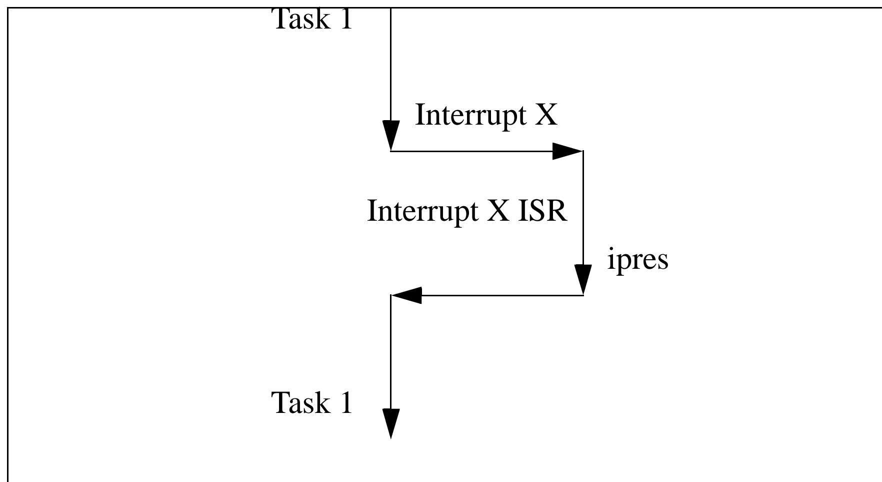


Figure 8. Type 1 ISR

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

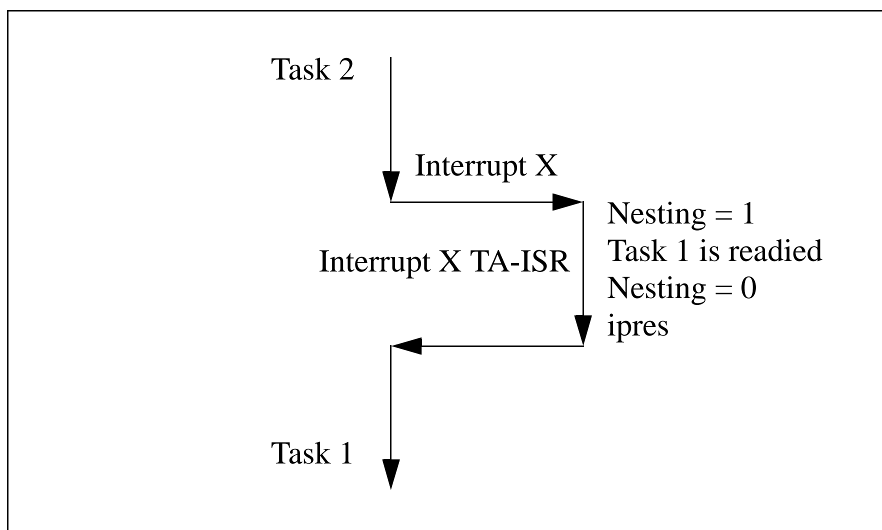


Figure 9. Type 2 ISR

It may seem as though the ISR in this Figure does not have to increment and decrement the nesting count. This is, however, very important. If the ISR for Interrupt X is called during an ISR that re-enables interrupts before completion, scheduling should not be performed when Interrupt X completes; scheduling should instead be deferred until the least nested ISR completes. The next Figure shows an example of this situation.

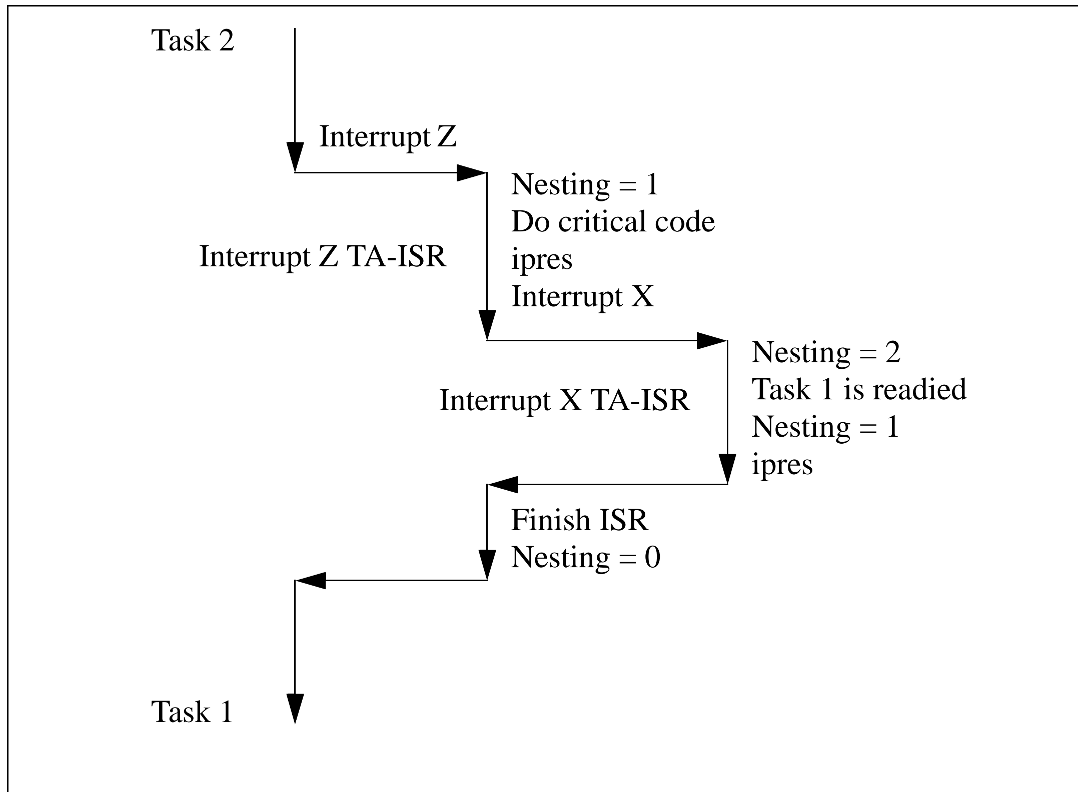


Figure 10. Type 2 ISR Nested Inside Type 3 ISR

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.

17.2.3 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: Use of TA-ISR

	µC/OS-II Application		
	Type 1*	**Type 2†**	**Type 3‡**
TA-ISR Required?	No	Yes	Yes

*. Type 1—Leaves interrupts disabled and does not signal task to ready state
†. Type 2—Leaves interrupts disabled and signals task to ready state
‡. Type 3—Reenables interrupts before completion

The following Figure shows the logical flow of a TA-ISR.

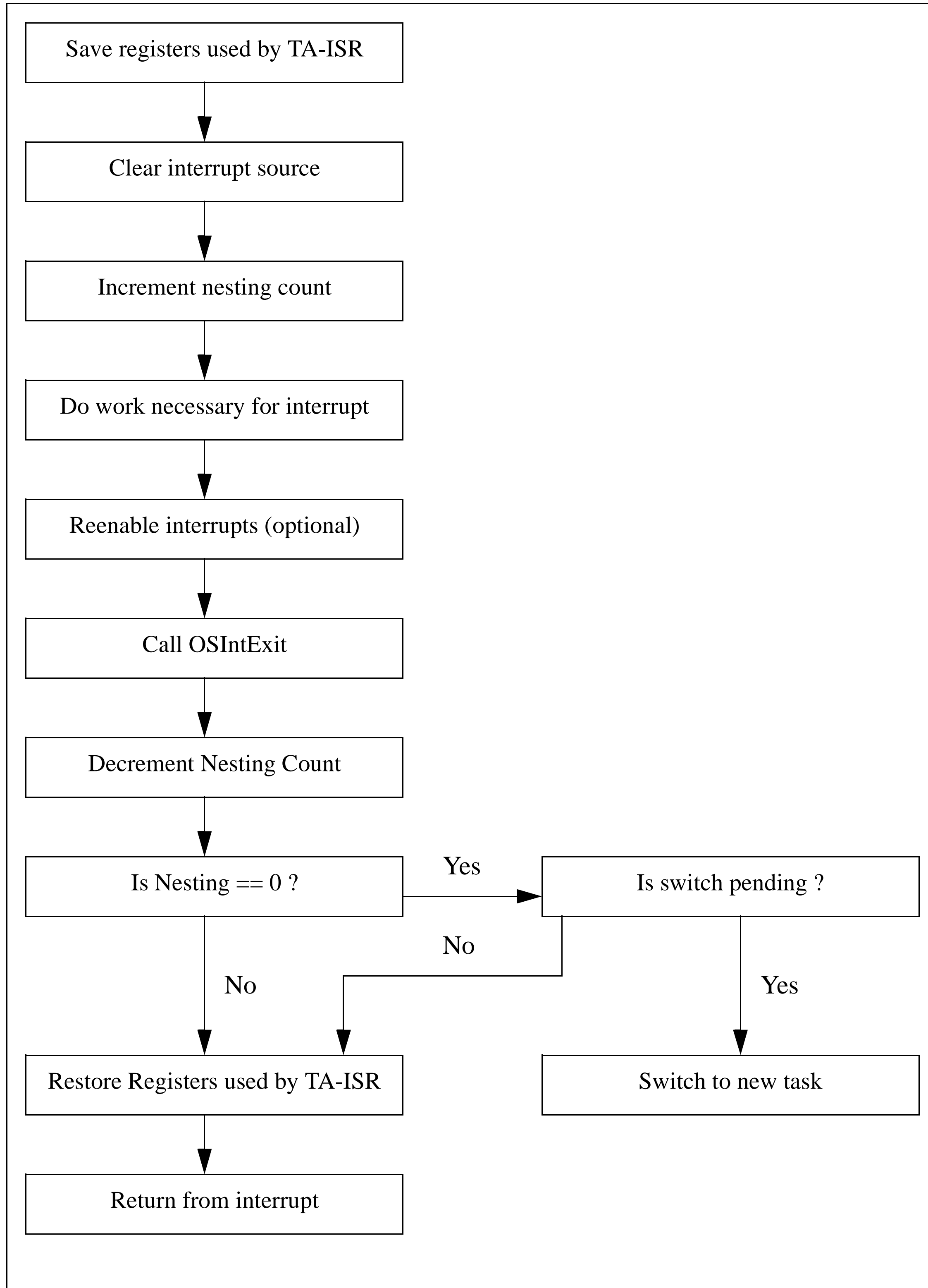


Figure 11. Logical Flow of a TA-ISR

17.2.3.1 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. If the TA-ISR decrements the nesting counter (8) 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 (8) 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 jump into 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 registers 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)     ;has MCOS started 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:
   ld    a,(bios_intnesting) ;nesting counter == 1?
(8)
   dec   a
   jr    z,taisr_intexit

taisr_setnesting:
   ld    (bios_intnesting),a
(9)
   jr    taisr_done
taisr_intexit:
   ld    a,(bios_swpend)     ;switch pend-
ing?                        (10)
   or    a
   jr    z,taisr_setnesting
   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_done:
   pop  hl
(12)
   pop  af
   ipres
   ret
#endasm
```

17.3 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 via a μC/OS-II semaphore. The following is a list of Dynamic C functions that are non-reentrant.

Library	Non-reentrant Functions
MATH.LIB	randg, randb, rand
RS232.LIB	All
RTCLOCK.LIB	write_rtc, tm_wr
STDIO.LIB	kbhit, getchar, gets, getswf, selectkey
STRING.LIB	atof*, atoi*, strtok
SYS.LIB	clockDoublerOn, clockDoublerOff, useMainOsc, useClockDivider, use32kHzOsc
VDRIVER.LIB	VdGetFreeWd, VdReleaseWd
XMEM.LIB	root2xmem, xmem2root, WriteFlash
JRIO.LIB	digOut, digOn, digOff, jrioInit, anaIn, anaOut, cof_anaIn
JR485.LIB	All

*reentrant but sets the global **_xtoxErr** flag

The serial port functions (**RS232.LIB** functions) should be used in a restricted manner with μC/OS-II. Two tasks can use the same port as long as both are not reading, or both are not writing; i.e., one task can read from serial port X and another task can write to serial port X at the same time without conflict.

17.4 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).

Default Configuration

µC/OS-II usually relies on the include file **os_cfg.h** to get values for the configuration constants. Since Dynamic C does not use this header file, these constants, along with their default values, are in **UCOS2.LIB**. A default stack configuration is also supplied in **UCOS2.LIB.** µC/OS-II for the Rabbit uses a more intelligent stack allocation scheme than other µC/OS-II implementations to take better advantage of unused memory.

The default configuration allows up to 10 normally created application tasks running at 64 ticks per second. Each task has a 512-byte stack. There are 2 queues specified, and 10 events. An event is a queue, mailbox or semaphore. You can define any combination of these three for a total of 10. If you want more than 2 queues, however, you must change the default value of **OS_MAX_QS**.

Some of the default configuration constants are:

```
// Maximum number of events (semaphores, queues, mailboxes)
#define OS_MAX_EVENTS 10

// Maximum number of tasks (less stat and idle tasks)
#define OS_MAX_TASKS 10

// Maximum number of queues in system
#define OS_MAX_QS 2

// Maximum number of memory partitions
#define OS_MAX_MEM_PART 0

// Enable normal task creation
#define OS_TASK_CREATE_EN 1

//Disable extended task creation
#defineOS_TASK_CREATE_EXT_EN 0

// Disable task deletion
#define OS_TASK_DEL_EN 0

// Disable statistics task creation
#define OS_TASK_STAT_EN 0

// Enable queue usage
#define OS_Q_EN 1

// Disable memory manager
#define OS_MEM_EN 0

// Enable mailboxes
#define OS_MBOX_EN 1
```

```
#// Enable semaphores
define OS_SEM_EN 1

// # of ticks in one second
#define OS_TICKS_PER_SEC 64

// # of 256 byte stacks (idle task stack)
#define STACK_CNT_256 1

//# of 512-byte stackstask stacks + initial program stack
#define STACK_CNT_512 OS_MAX_TASKS+1
```

If a particular portion of µC/OS-II is disabled, the code for that portion will not be compiled, making the overall size of the operating system smaller. Take advantage of this feature by customizing µC/OS-II based on the needs of each application.

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             0
#define OS_MAX_MEM_PART      15
#define OS_TASK_STAT_EN       1
#define OS_Q_EN               0
#define OS_MEM_EN             1
#define OS_MBOX_EN            0
#define OS_TICKS_PER_SEC     64
```

If a custom stack configuration is needed also, define the necessary macros for the counts of the different stack sizes needed by the application.

```
#define STACK_CNT_256 1  // idle task stack
#define STACK_CNT_512 2  // initial program + stat task stack
#define STACK_CNT_1K 10  // task stacks
#define STACK_CNT_2K 10  // number of 2K stacks
```

In the application code, follow the µC/OS-II and stack configuration constants with a **#use "ucos2.lib"** statement. This ensures that the definitions supplied outside of the library are used, rather than the defaults in the library.

This configuration uses 20 tasks, two semaphores, up to 15 memory partitions that the memory manager will control, and makes use of the statistics task. Note that the configuration constants for task creation, task deletion, and semaphores are not defined as the library defaults will suffice. Also, note that 10 of the application tasks will each have a 1024 byte stack, 10 will each have a 2048 byte stack, and an extra stack is declared for the statistics task.

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 "heart-beat"), **#use "ucos2.lib"** must be included in every µC/OS-II application (1). In order for each of the tasks to have access to the random number generator semaphore, it is declared as a global variable (2). In most cases, all mailboxes, queues, and semaphores will be declared with global scope. Next, **OSInit** must be called before any other µC/OS-II function to ensure that the operating system is properly initialized (3). Before µC/OS-II can begin running, at least one application task must be created. In this application, all tasks are created before the operating system begins running (4). It is perfectly acceptable for tasks to create other tasks. Next, the semaphore each task uses is created (5). Once all of the initialization is done, **OSStart** is called to start µC/OS-II running (6). In the code that each of the tasks run, it is important to note the variable declarations. The default storage class in Dynamic C is static, so to ensure that the task code is reentrant, all are declared auto (7). Each task runs as an infinite loop and once this application is started, µC/OS-II will run indefinitely.

```
// 1. Explicitly use uC/OS-II library
#use "ucos2.lib"
void RandomNumberTask(void *pdata);
// 2. Declare semaphore global so all tasks have access
OS_EVENT* RandomSem;
void main()
{
   int i;
   // 3. Initialize OS internals
   OSinit();
   for(i = 0; i < OS_MAX_TASKS; i++)
      // 4. Create each of the system tasks
      OSTaskCreate(RandomNumberTask, NULL, 512, i);

   // 5. semaphore to control access to random number generator
   RandomSem = OSSemCreate(1);
   // 6. Begin multitasking
   OSStart();
}
void RandomNumberTask(void *pdata)
{
   // 7. Declare as auto to ensure reentrancy.
   auto OS_TCB data;
   auto INT8U  err;
   auto INT16U RNum;

   OSTaskQuery(OS_PRIO_SELF, &data);
   while(1)
   {
      // Rand is not reentrant, so access must be controlled
      // via a semaphore.
      OSSemPend(RandomSem, 0, &err);
      RNum = (int)(rand() * 100);
      OSSemPost(RandomSem);
      printf("Task%d's random #: %d\n",data.OSTCBPrio,RNum);
      // Wait 3 seconds in order to view output from each task.
      OSTimeDlySec(3);
   }
}
```

Example 2

This application runs exactly the same code as Example 1, except that each of the tasks are created with 1024 byte stacks. The main difference between the two is the configuration of µC/OS-II.

First, each configuration constant that differs from the library default is defined. The configuration in this example differs from the default in that it allows only two events (the minimum needed when using only one semaphore), 20 tasks, no queues, no mailboxes, and the system tick rate is set to 32 ticks per second (1). Next, since this application uses tasks with 1024 byte stacks, it is necessary to define the configuration constants differently than the library default (2). Notice that one 512 byte stack is declared. Every Dynamic C program starts with an initial stack, and defining **`STACK_CNT_512`** is crucial to ensure that the application has a stack to use during initialization and before multi-tasking begins. Finally **`ucos2.lib`** is explicitly used (3). This ensures that the definitions in (1 and 2) are used rather than the library defaults. The last step in initialization is to set the number of ticks per second via **`OSSetTicksPerSec`** (4).

The rest of this application is identical to example 1 and is explained in the previous section.

```
// 1. Define necessary configuration constants for uC/OS-II
#define OS_MAX_EVENTS          2
#define OS_MAX_TASKS          20
#define OS_MAX_QS              0
#define OS_Q_EN                0
#define OS_MBOX_EN             0
#define OS_TICKS_PER_SEC      32

// 2. Define necessary stack configuration constants
#define STACK_CNT_512 1            // initial program stack
#define STACK_CNT_1K OS_MAX_TASKS  // task stacks

// 3. This ensures that the above definitions are used
#use "ucos2.lib"
void RandomNumberTask(void *pdata);
// Declare semaphore global so all tasks have access
OS_EVENT* RandomSem;
void main(){
   int i;
   // Initialize OS internals
   OSInit();
   for(i = 0; i < OS_MAX_TASKS; i++){
      // Create each of the system tasks
      OSTaskCreate(RandomNumberTask, NULL, 1024, i);
   }
   // semaphore to control access to random number generator
   RandomSem = OSSemCreate(1);
   // 4. Set number of system ticks per second
   OSSetTicksPerSec(OS_TICKS_PER_SEC);
   // Begin multi-tasking
   OSStart();
}
```

```
void RandomNumberTask(void *pdata)
{
   // Declare as auto to ensure reentrancy.
   auto OS_TCB data;
   auto INT8U  err;
   auto INT16U RNum;

   OSTaskQuery(OS_PRIO_SELF, &data);
   while(1)
   {
      // Rand is not reentrant, so access must be controlled
      // via a semaphore.
      OSSemPend(RandomSem, 0, &err);
      RNum = (int)(rand() * 100);
      OSSemPost(RandomSem);
      printf("Task%02d's random #: %d\n",data.OSTCBPrio,RNum);
      // Wait 3 seconds in order to view output from each task.
      OSTimeDlySec(3);
   }
}
```

17.5 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.
```

Software License Agreement

Z-WORLD SOFTWARE END USER LICENSE AGREEMENT

IMPORTANT-READ CAREFULLY: BY INSTALLING, COPYING OR OTHERWISE USING THE ENCLOSED Z-WORLD,INC. ("Z-WORLD") DYNAMIC C SOFTWARE, WHICH INCLUDES COMPUTER SOFTWARE ("SOFTWARE") AND MAY INCLUDE ASSOCIATED MEDIA, PRINTED MATERIALS, AND "ONLINE" OR ELECTRONIC DOCUMENTATION ("DOCUMENTATION"), YOU (ON BEHALF OF YOURSELF OR AS AN AUTHORIZED REPRESENTATIVE ON BEHALF OF AN ENTITY) AGREE TO ALL THE TERMS OF THIS END USER LICENSE AGREEMENT ("LICENSE") REGARDING YOUR USE OF THE SOFTWARE. IF YOU DO NOT AGREE WITH ALL OF THE TERMS OF THIS LICENSE, DO NOT INSTALL, COPY OR OTHERWISE USE THE SOFTWARE AND IMMEDIATELY CONTACT Z-WORLD FOR RETURN OF THE SOFTWARE AND A REFUND OF THE PURCHASE PRICE FOR THE SOFTWARE.

We are sorry about the formality of the language below, which our lawyers tell us we need to include to protect our legal rights. If You have any questions, write or call Z-World at (530) 757-4616, 2900 Spafford Street, Davis, California 95616.

1. Definitions. In addition to the definitions stated in the first paragraph of this document, capitalized words used in this License shall have the following meanings:

   “Qualified Applications” means an application program developed using the Software and that links with the development libraries of the Software.

   “Qualified Systems” means a microprocessor-based computer system which is either (i) manufactured by, for or under license from Z-WORLD, or (ii) based on the Rabbit 2000 microprocessor. Qualified Systems may not be (a) designed or intended to be re-programmable by your customer using the Software, or (b) competitive with Z-WORLD products, except as otherwise stated in a written agreement between Z-World and the system manufacturer. Such written agreement may require an end user to pay run time royalties to Z-World.

2. License. Z-WORLD grants to You a nonexclusive, nontransferable license to (i) use and reproduce the Software, solely for internal purposes and only for the number of users for which You have purchased licenses for (the "Users") and not for redistribution or resale; (ii) use and reproduce the Software solely to develop the Qualified Applications; and (iii) use, reproduce and distribute, the Qualified Applications, in object code only, to end users solely for use on Qualified Systems; provided, however, any agreement entered into between You and such end users with respect to a Qualified Application is no less protective of Z-Worldís 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 **COLDLOADER.BIN** in their unaltered forms.

3. Restrictions. Except as otherwise stated, You may not, nor permit anyone else to, decompile, reverse engineer, disassemble or otherwise attempt to reconstruct or discover the source code of the Software, alter, merge, modify, translate, adapt in any way, prepare any derivative work based upon the Software, rent, lease network, loan, distribute or otherwise transfer the Software or any copy thereof. You shall not make copies of the copyrighted Software and/or documentation without the prior written permission of Z-WORLD; provided that, You may make one (1) hard copy of such documentation for each User and a reasonable number of back-up copies for

Your own archival purposes. You may not use copies of the Software as part of a benchmark or comparison test against other similar products in order to produce results strictly for purposes of comparison. The Software contains copyrighted material, trade secrets and other proprietary material of Z-WORLD and/or its licensors and You must reproduce, on each copy of the Software, all copyright notices and any other proprietary legends that appear on or in the original copy of the Software. Except for the limited license granted above, Z-WORLD retains all right, title and interest in and to all intellectual property rights embodied in the Software, including but not limited to, patents, copyrights and trade secrets.

4. Export Law Assurances. You agree and certify that neither the Software nor any other technical data received from Z-WORLD, nor the direct product thereof, will be exported outside the United States or re-exported except as authorized and as permitted by the laws and regulations of the United States and/or the laws and regulations of the jurisdiction, (if other than the United States) in which You rightfully obtained the Software. The Software may not be exported to any of the following countries: Cuba, Iran, Iraq, Libya, North Korea, or Syria.
5. Government End Users. If You are acquiring the Software on behalf of any unit or agency of the United States Government, the following provisions apply. The Government agrees: (i)if the Software is supplied to the Department of Defense ("DOD"), the Software is classified as "Commercial Computer Software" and the Government is acquiring only "restricted rights" in the Software and its documentation as that term is defined in Clause 252.227-7013(c)(1) of the DFARS; and (ii) if the Software is supplied to any unit or agency of the United States Government other than DOD, the Government's rights in the Software and its documentation will be as defined in Clause 52.227-19(c)(2) of the FAR or, in the case of NASA, in Clause 18-52.227-86(d) of the NASA Supplement to the FAR.
6. Disclaimer of Warranty. You expressly acknowledge and agree that the use of the Software and its documentation is at Your sole risk. THE SOFTWARE, DOCUMENTATION, AND TECHNICAL SUPPORT ARE PROVIDED ON AN "AS IS" BASIS AND WITHOUT WARRANTY OF ANY KIND. Information regarding any third party services included in this package is provided as a convenience only, without any warranty by Z-WORLD, and will be governed solely by the terms agreed upon between You and the third party providing such services. Z-WORLD AND ITS LICENSORS EXPRESSLY DISCLAIM ALL WARRANTIES, EXPRESS, IMPLIED, STATUTORY OR OTHERWISE, INCLUDING BUT NOT LIMITED TO THE IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NON-INFRINGEMENT OF THIRD PARTY RIGHTS. Z-WORLD DOES NOT WARRANT THAT THE FUNCTIONS CONTAINED IN THE SOFTWARE WILL MEET YOUR REQUIREMENTS, OR THAT THE OPERATION OF THE SOFTWARE WILL BE UNINTERRUPTED OR ERROR-FREE, OR THAT DEFECTS IN THE SOFTWARE WILL BE CORRECTED. FURTHERMORE, Z-WORLD DOES NOT WARRANT OR MAKE ANY REPRESENTATIONS REGARDING THE USE OR THE RESULTS OF THE SOFTWARE IN TERMS OF ITS CORRECTNESS, ACCURACY, RELIABILITY OR OTHERWISE. NO ORAL OR WRITTEN INFORMATION OR ADVICE GIVEN BY Z-WORLD OR ITS AUTHORIZED REPRESENTATIVES SHALL CREATE A WARRANTY OR IN ANY WAY INCREASE THE SCOPE OF THIS WARRANTY. SOME JURISDICTIONS DO NOT ALLOW THE EXCLUSION OF IMPLIED WARRANTIES, SO THE ABOVE EXCLUSION MAY NOT APPLY TO YOU.
7. Limitation of Liability. YOU AGREE THAT UNDER NO CIRCUMSTANCES, INCLUDING NEGLIGENCE, SHALL Z-WORLD BE LIABLE FOR ANY INCIDENTAL, SPECIAL OR

CONSEQUENTIAL DAMAGES (INCLUDING DAMAGES FOR LOSS OF BUSINESS PROFITS, BUSINESS INTERRUPTION, LOSS OF BUSINESS INFORMATION AND THE LIKE) ARISING OUT OF THE USE AND/OR INABILITY TO USE THE SOFTWARE, EVEN IF Z-WORLD OR ITS AUTHORIZED REPRESENTATIVE HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES. SOME JURISDICTIONS DO NOT ALLOW THE LIMITATION OR EXCLUSION OF LIABILITY FOR INCIDENTAL OR CONSEQUENTIAL DAMAGES SO THE ABOVE LIMITATION OR EXCLUSION MAY NOT APPLY TO YOU. IN NO EVENT SHALL Z-WORLDíS TOTAL LIABILITY TO YOU FOR ALL DAMAGES, LOSSES, AND CAUSES OF ACTION (WHETHER IN CONTRACT, TORT, INCLUDING NEGLIGENCE, OR OTHERWISE) EXCEED THE AMOUNT PAID BY YOU FOR THE SOFTWARE.

8. Termination. This License is effective for the duration of the copyright in the Software unless terminated. You may terminate this License at any time by destroying all copies of the Software and its documentation. This License will terminate immediately without notice from Z-WORLD if You fail to comply with any provision of this License. Upon termination, You must destroy all copies of the Software and its documentation. Except for Section 2 ("License"), all Sections of this Agreement shall survive any expiration or termination of this License.
9. General Provisions. No delay or failure to take action under this License will constitute a waiver unless expressly waived in writing, signed by a duly authorized representative of Z-WORLD, and no single waiver will constitute a continuing or subsequent waiver. This License may not be assigned, sublicensed or otherwise transferred by You, by operation of law or otherwise, without Z-WORLD's prior written consent. This License shall be governed by and construed in accordance with the laws of the United States and the State of California, exclusive of the conflicts of laws principles. The United Nations Convention on Contracts for the International Sale of Goods shall not apply to this License. If for any reason a court of competent jurisdiction finds any provision of this License, or portion thereof, to be unenforceable, that provision of the License shall be enforced to the maximum extent permissible so as to affect the intent of the parties, and the remainder of this License shall continue in full force and effect. This License constitutes the entire agreement between the parties with respect to the use of the Software and its documentation, and supersedes all prior or contemporaneous understandings or agreements, written or oral, regarding such subject matter. There shall be no contract for purchase or sale of the Software except upon the terms and conditions specified herein. Any additional or different terms or conditions proposed by You or contained in any purchase order are hereby rejected and shall be of no force and effect unless expressly agreed to in writing by Z-WORLD. No amendment to or modification of this License will be binding unless in writing and signed by a duly authorized representative of Z-WORLD.

Copyright 2000 Z-World, Inc. All rights reserved.

Index

Symbols

operator ...16, 17, 18
operator ...16, 17, 18
#asm ...89, 103, 104, 134
#class ...134
#debug ...89, 125, 134
#define ...16, 17, 18, 134
#elif ...135
#else ...135
#endasm ...103, 104, 134
#endif ...135
#error ...135
#fatal ...134
#funcchain ...32, 135
#if ...135
#ifdef ...135
#ifndef ...135
#include
  absence of ...35
#interleave ...135
#KILL ...136
#makechain ...32, 136
#memmap ...4, 136
#nodebug ...89, 125, 134, 340
#nointerleave ...135
#nouseix ...136
#undef ...18, 136
#use ...35, 38, 136
#useix ...136
#warns ...136
#warnt ...136
#ximport ...137
& (address operator) ...26
* (indirection operator) ...26
@RETVAL ...113, 114
@SP ...108, 110, 111, 113, 114, 116
_GLOBAL_INIT ...127
{ } curly braces ...21

A

abort ...117
About Dynamic C ...359
abstract data types ...23
adc (add-with-carry) ...103
Add to Top button ...343
Add/Del Items <CTRL-W> 344, 353
Add/Del Watch Expression <CTRL-W> ...343
adding watch window items 343, 344
address operator (&) ...26
address space ...4
addresses ...95
addresses in assembly language 106, 109
aggregate data types ...24
ALT key ...334
ALT-Backspace ...336
ALT-C ...339
ALT-CTRL-F3 ...339
ALT-F10 ...344
ALT-F2 ...341, 342
ALT-F4 ...336
ALT-F9 ...341
ALT-H ...356
ALT-O ...346
ALT-R ...341
ALT-SHIFT-backspace ...337
ALT-W ...352
always_on ...117
anymem ...117
argument passing ..28, 108, 113, 114, 115
  modifying value ...28
arrange icons
  command ...352
arranged icons ...353
arrays ...24, 25, 28
  characters ...19
  subscripts ...24
arrow keys ...333, 334
Assembly Language ...103
assembly language ...3, 38, 89, 104, 105, 113, 114, 115, 116, 342
  embedding C statements ..104
assembly window ...3, 104, 352, 354
assignment operators ...143
associativity ...139
auto ...90, 106, 107, 108, 117
Auto Open STDIO Window 349

B

backslash
  continuation in directives .134
backslash (\)
  character literals ...16, 20
basic unit of a C program ...22
baud rate ...351
BCDE ...107, 113, 114, 115
BeginHeader ...37, 38
binary operators ...139
BIOS ...121
body
  module ...37, 38
branching ...31, 32
break ...29, 30, 32, 118, 130
  example ...30
break points ...89, 104, 125, 342, 344
  hard ...341, 342
  interrupt status ...341, 342
  soft ...341, 342
breaking out of a loop ...30
breaking out of a switch statement ...30
buttons, toolbar ...351

C

C functions calling assembly code ...113
C language 3, 4, 5, 13, 19, 23, 28, 32, 105, 107
C statements embedded in assembly code ...104
C variables in assembly language 106
cascaded windows ...352
case ...32, 118, 121
case-sensitive searching ...337, 338
char ...23, 118, 132
characters
  arrays ...19
  embedded quotes ...20
  nonprinting values ...20
  special values ...20
checking
  pointers ...27
  stack ...89, 90
  type ...22
Clear Watch Window ...344
clipboard ...337
Close <CTRL-F4> ...335
closing a file ...335
CoData Structure ...46
  pointer to ...48
Cofunctions ...50
  abandon ...54
  calling restrictions ...51
  everytime ...54
  indexed ...52
  single user ...52
  syntax ...50
COM port ...351
communication

serial 351
compilation .. 333, 339, 353, 356
direct to controller 3
errors 338
speed 3
targetless 339
Compile
to flash 339
to RAM 339
to Target 339
COMPILE menu 339
Compile to File <CTRL-F3> 339
Compile to File with *.RTI File <ALT-CTRL-F3> 339
Compile to Target <F3> 339
compiler directives 134
#asm 134
#class 134
#debug 134
#define 134
#elif 135
#else 135
#endasm 134
#endif 135
#error 135
#fatal 134
#funcchain 135
#GLOBAL_INIT 134
#if 135
#ifdef 135
#ifndef 135
#interleave 135
#KILL 136
#makechain 136
#memmap 136
#nodebug 134
#nointerleave 135
#nouseix 136
#undef 136
#use 136
#useix 136
#warns 136
#warnt 136
#ximport 137
line continuation 134
Compiler options .. 27, 346, 347, 348
compiling 3
to file 333, 339
to RAM 339
to ROM 339
to target 333, 339
compound
names 15
statements 21
const 119
Contents
Help 359
continue 29, 30, 120, 130
example 30
copying text <CTRL-C> 336, 337
costate 120
Costatements 44
syntax 45
costatements 117, 120, 131, 133
Create *.RTI File for Targetless Compile 339
CTRL key 333
CTRL-F10 344
CTRL-F2 342
CTRL-F3 339
CTRL-G 338
CTRL-H 357, 358, 359
CTRL-I 341, 342
CTRL-N 338
CTRL-O 341, 342
CTRL-P 338
CTRL-U 344
CTRL-V 337
CTRL-W 344
CTRL-X 337
CTRL-Y 341, 343
CTRL-Z 341
curly braces { } 21
cursor
execution 342
positioning 338
text 359
cutting text <CTRL-X> 337

D

data types 24
aggregate 24
primitive 14
DATASEG 95
db 105
DCW.CFG 352
DCW.INI 352
debug 120, 134
editor 350
mode 90, 338, 341
debugger 3
options 346, 349
debugging . 3, 89, 134, 341, 342, 344, 345
assembly-level view 3
declarations 21, 37
default 32, 121
storage class 5
Del from Top button 343
deleting watch window items 343, 344
demotion 347
direct
compilation 3
directives 4
#asm 89, 103, 104
#debug 89, 125, 134
#define 16, 17, 18
#endasm 103, 104
#funcchain 32
#makechain 32
#nodebug 89, 125, 340
#undef 18
#use 35, 38
Disassemble at Address <ALT-F10> 344, 354
Disassemble at Cursor <CTRL-F10> 344, 354
disassembled code 343
display
options 346, 350
do loop 29
dot operator 15, 25
dump window 345
dw 105
dynamic
storage allocation 25
Dynamic C 3
differences 4, 5, 32
exit 336, 352
installation 5, 116
support files 36

E

EDIT menu 336, 337, 338
edit mode 333, 338, 343
editing 3
editor 3
options 346
else 121
embedded assembly code 3, 108, 113, 114, 115, 116
embedded quotes 20
End key 333
EndHeader 37, 38
EPROM 4, 5
equ 104
errors
codes 91, 92

editor 350
fatal 92
locating 338
run-time 91
ESC key
to close menu 334
Evaluate button 343
examples
break 30
continue 30
for loop 29
modules 38
of array 24
union 25
execution 341, 344
cursor 342
Exit <ALT-F4> 336
Expr. in Call 359
extended memory 4, 112, 113, 132
extern 38, 121

F

F (status register) 355
F10 352
F2 ....341, 342
F3 339
F4 338
F6 ....337, 338
F7 ....341, 342
F8 ....341, 342
F9 341
file commands
close file 335
create file 335
open file 335
save file 335
FILE menu 335, 336
Find next <SHIFT-F5> 338
firsttime 122
float 23, 122, 132
values 19
for 21, 123
character literals 20
loop 29
example 29
frame
reference point 113, 114
reference pointer 90, 111, 113, 125
free space 356
Full Speed Bkgnd TX 351
function calls 22, 28, 90, 108, 113, 114, 115, 116, 117, 359
function chains .32, 33, 127, 136
function groups
arithmetic
abs 159
getcrc 209
bit manipulation
BIT 166
bit 165
RES 282
res 281
SET 297
set 296
character
isalnum 217
isalpha 218
iscntrl 218
isdigit 220
isgraph 220
islower 221
isprint 222
ispunct 223
isspace 221
isupper 224
isxdigit 224
extended memory
paddr 270
root2xmem 283
WriteFlash2 328
xalloc 331
xmem2root 331
xmem2xmem 332
fast fourier transforms
fftcplx 188
fftcplxinv 189
fftreal 190
fftrealinv 191
hanncplx 212
hannreal 213
powerspectrum 273
file system
fclose 186
fcreate 186
fcreate_unused 187
fdelete 187
fopen_rd 200
fopen_wr 200
fread 201
fs_format 203
fs_init 204
fs_reserve_blocks 205
fsck 205
fseek 206
fshift 207
ftell 207
fwrite 208
floating-point math
acos 159
acot 160
acsc 160
asec 161
asin 161
atan 162
atan2 163
ceil 170
cos 180
cosh 181
deg 182
exp 185
fabs 185
floor 199
fmod 199
frexp 202
labs 226
ldexp 226
log 227
log10 227
modf 234
poly 271
pow 272
pow10 272
rad 277
rand 278
randb 278
randg 279
sin 300
sinh 301
sqrt 303
tan 317
tanh 318
I/O
BitRdPortE 166
BitRdPortI 167
BitWrPortE 168
BitWrPortI 169
RdPortE 279
RdPortI 280
WrPortE 330
WrPortI 330
interrupts
GetVectExtern2000 210
GetVectIntern 211
SetVectExtern2000 299
SetVectIntern 300
low-level flash access
flash_erasechip 192
flash_erasesector 192
flash_gettype 193

flash_init 194
flash_read 195
flash_readsector 196
flash_sector2xwindow . 197
flash_writesector 198
MicroC/OS-II
OSInit 234
OSMboxAccept 235
OSMboxCreate 235
OSMboxPend 236
OSMboxPost 237
OSMboxQuery 238
OSMemCreate 239
OSMemGet 240
OSMemPut 240
OSMemQuery 241
OSQAccept 241
OSQCreate 242
OSQFlush 243
OSQPend 244
OSQPost 245
OSQPostFront 246
OSQQuery 247
OSSchedLock 247
OSSchedUnlock 248
OSSemAccept 248
OSSemCreate 249
OSSemPend 249
OSSemPost 250
OSSemQuery 251
OSSetTickPerSec 252
OSStart 252
OSStatInit 253
OSTaskChangePrio 253
OSTaskCreate 254
OSTaskCreateExt 255
OSTaskCreateHook 256
OSTaskDel 257
OSTaskDelHook 258
OSTaskDelReq 259
OSTaskQuery 260
OSTaskResume 261
OSTaskStatHook 261
OSTaskStkChk 262
OSTaskSuspend 263
OSTaskSwHook 263
OSTimeDly 264
OSTimeDlyHMSM 265
OSTimeDlyResume 266
OSTimeDlySec 267
OSTimeGet 267
OSTimeSet 268
OSTimeTickHook 268
OSVersion 269
miscellaneous
longjmp 228
qsort 276
runwatch 283
setjmp 298
multitasking
CoBegin 173
CoPause 179
CoReset 179
CoResume 180
DelayMs 182
DelaySec 183
DelayTicks 183
IntervalMs 215
IntervalSec 215
IntervalTick 216
isCoDone 219
isCoRunning 219
number-to-string conversion
ftoa 208
htoa 214
itoa 225
ltoa 228
ltoan 229
utoa 324
real-time clock
mktime 232
mktm 233
read_rtc 280
read_rtc_32kHz 281
tm_rd 319
tm_wr 320
write_rtc 329
serial communication
cof_serAgetc 173
cof_serAgets 174
cof_serAputc 175
cof_serAputs 176
cof_serAread 177
cof_serAwrite 178
cof_serBgetc 173
cof_serBgets 174
cof_serBputc 175
cof_serBputs 176
cof_serBread 177
cof_serBwrite 178
cof_serCgetc 173
cof_serCgets 174
cof_serCputc 175
cof_serCputs 176
cof_serCread 177
cof_serCwrite 178
cof_serDgetc 173
cof_serDgets 174
cof_serDputc 175
cof_serDputs 176
cof_serDread 177
cof_serDwrite 178
serAclose 284
serAdatabits 285
serAflowcontrolOff 285
serAflowcontrolOn 286
serAgetc 287
serAgetError 288
serAopen 289
serAparity 290
serApeek 291
serAputc 291
serAputs 292
serArdFlush 292
serArdFree 293
serArdUsed 293
serAread 294
serAwrFlush 295
serAwrFree 295
serAwrite 295
serBclose 284
serBdatabits 285
serBflowcontrolOff 285
serBflowcontrolOn 286
serBgetc 287
serBgetError 288
serBopen 289
serBparity 290
serBpeek 291
serBputc 291
serBputs 292
serBrdFlush 292
serBrdFree 293
serBrdUsed 293
serBread 294
serBwrFlush 295
serBwrFree 295
serBwrite 295
serCclose 284
serCdatabits 285
serCflowcontrolOff 285
serCflowcontrolOn 286
serCgetc 287
serCgetError 288
serCheckParity 284
serCopen 289
serCparity 290
serCpeek 291
serCputc 291
serCputs 292
serCrdFlush 292
serCrdFree 293

serCrdUsed 293
serCread 294
serCwrFlush 295
serCwrFree 295
serCwrite 295
serDclose 284
serDdatabits 285
serDflowcontrolOff 285
serDflowcontrolOn 286
serDgetc 287
serDgetError 288
serDopen 289
serDparity 290
serDpeek 291
serDputc 291
serDputs 292
serDrdFlush 292
serDrdFree 293
serDrdUsed 293
serDread 294
serDwrFlush 295
serDwrFree 295
serDwrite 295
STDIO
getchar 209
gets 210
kbhit 225
outchrs 269
outstr 270
printf 274
putchar 275
puts 275
sprintf 302
string manipulation
memchr 229
memcmp 230
memcpy 231
memmove 231
memset 232
strcat 303
strchr 304
strcmp 305
strcmpi 306
strcpy 307
strcspn 307
strlen 308
strncat 308
strncmp 309
strncmpi 310
strncpy 311
strpbrk 312
strrchr 312
strspn 313
strstr 313
strtok 315
tolower 321
toupper 321
string-to-number conversion
atof 164
atoi 164
atol 165
strtod 314
strtol 316
system
_sysIsSoftReset 316
chkHardReset 170
chkSoftReset 171
chkWDTO 171
clockDoublerOff 172
clockDoublerOn 172
defineErrorHandler 181
exit 184
forceSoftReset 201
GetVectExtern2000 210
ipres 216
ipset 217
premain 274
sysResetChain 317
updateTimers 322
use32HzOsc 322
useClockDivider 323
useMainOsc 323
watchdog
Disable_HW_WDT 184
hitwd 214
VdGetFreeWd 325
VdInit 326
VdReleaseWd 327
function headers 39
function help 39
function libraries 3, 35, 37
function lookup <CTRL-H> 357, 358, 359
function returns 90, 113, 114, 115
functions 22
entry and exit 90
prototypes 22, 24, 37

G

generated 92
Global Initialization 33
global variables 25
goto 30, 31, 123
Goto <CTRL-G> 338

H

hard break points 341, 342
header
function 39
module 37, 38
heap storage 356
Help
online 359
HELP menu .356, 357, 358, 359
HL 107, 109, 110, 113, 114, 115
Home key 333
horizontal tiling 352, 353

I

IBM PC 3, 341, 351
icons
arranged 352, 353
IEEE floating point 122
if 121
multichoice 31
simple 31
with else 31
indirection operator (*) 26
information window 352, 356
init_on 124
insertion point 337, 338
INSPECT menu ..343, 344, 345, 353
installation
Dynamic C 5, 116
int 23, 124, 132
integers 19
hexadecimal 19
long 19
octal 19
unsigned 19
interrupt 124
interrupt service routines 3, 115, 116, 124
interrupt status
and break points 341, 342
interrupts 115, 116
flag 342
latency 115
IX (index register) ..89, 90, 111, 112, 113, 125, 131

K

kernel
real-time 90
key module 37
keystrokes
<ALT R> select RUN menu .. 341
<ALT-Backspace> undoing changes 336

<ALT-C> select COMPILE menu ... 339
<ALT-F> select FILE menu .. 334
<ALT-F10> Disassemble at Address ... 344
<ALT-F2> Toggle hard break point ... 341, 342
<ALT-F4> Exit ... 336
<ALT-F4> Quitting Dynamic C ... 334
<ALT-F9> Run w/ No Polling 341
<ALT-H> select HELP menu 356
<ALT-O> select OPTIONS menu ... 346
<ALT-SHIFT-backspace> re-doing changes ... 337
<ALT-W> select WINDOW menu ... 352
<CTRL-F> Compile to File .. 339
<CTRL-F10> Disassemble at Cursor ... 344
<CTRL-F2> Reset Program .. 341, 342
<CTRL-F3> Compile to File with *.RTI File ... 339
<CTRL-G> Goto ... 338
<CTRL-H> Library Help lookup . 334, 357, 358, 359
<CTRL-I> Toggle interrupt .. 341, 342
<CTRL-N> next error ... 338
<CTRL-O> Toggle polling ... 341, 342
<CTRL-P> previous error 338
<CTRL-U> Update Watch window ... 344
<CTRL-V> pasting text .. 337
<CTRL-W> Add/Del Items .. 344
<CTRL-X> cutting text ... 337
<CTRL-Y> Reset target . 341, 343
<CTRL-Z> Stop ... 341
<F10> Assembly window 352
<F2> Toggle break point 341, 342
<F3> Compile to Target .. 339
<F7> Trace into ... 341, 342
<F8> Step over ... 341, 342
<F9> Run ... 341
<SHIFT-F5> Find next ... 338
keywords 4, 32, 89, 112, 117, 121, 125, 127, 131
- abort ... 117
- always_on ... 117
- anymem ... 117
- auto ... 117
- break ... 118
- case ... 118
- char ... 118
- continue ... 120
- costate ... 120
- debug ... 120
- default ... 121
- do ... 121
- else ... 121
- extern ... 121
- firsttime ... 122
- float ... 122
- for ... 123
- goto ... 123
- if ... 123
- init_on ... 124
- int ... 124
- interrupt ... 124
- long ... 124
- nodebug ... 125
- norst ... 125
- nouseix ... 125
- NULL ... 125
- protected ... 126
- return ... 126
- root ... 127
- segchain ... 127
- shared ... 127
- short ... 128
- size ... 128
- sizeof ... 128
- speed ... 128
- static ... 129
- struct ... 129
- switch ... 130
- typedef ... 130
- union ... 131
- unsigned ... 131
- useix ... 131
- waitfor ... 131
- while ... 132
- xdata ... 132
- xmem ... 132
- xstring ... 133
- yield ... 133

L

language elements 13, 15, 19, 117
- operators ... 139

latency interrupts ... 115
Lib Entries ... 357
LIB.DIR ... 38, 136, 357
Libraries ... 35
libraries ... 3, 35
- modules ... 37
- real-time programming ... 3

library functions ... 357
Library Help lookup ... 39
Library Help lookup <CTRL-H> 357, 358, 359
lick ... 336
linking ... 3
locating errors ... 338
long ... 124, 132
lookup function <CTRL-H> 357, 358, 359
loops ... 29
- breaking out of ... 30
- do ... 121
- for ... 123
- skipping to next pass ... 30

M

macros 16, 17, 18, 104, 106, 134
- restrictions ... 18
- with parameters ... 16

main function 22, 35, 89, 125
memory
- dump ... 343
- dump at address ... 345
- dump Flash ... 345
- dump to file ... 345
- extended 4, 95, 112, 113, 132
- logical ... 95
- management ... 117, 127
- physical ... 95
- random access ... 4, 5
- read-only ... 4, 5
- root . 4, 95, 97, 106, 107, 109, 110, 112, 127

memory management unit (MMU) ... 4, 95
Memory options ... 346
menus
- COMPILE ... 334, 339
- EDIT ... 334, 336, 337, 338
- FILE ... 334, 335, 336
- HELP 334, 356, 357, 358, 359

INSPECT 334, 343, 344, 345, 353
OPTIONS ..27, 334, 346, 347, 348, 349, 350, 351
RUN 334, 341, 342, 343
system 334
WINDOW 334, 352, 353, 354, 355, 356
message window ..338, 352, 353
minimized windows 353
MMU (memory management unit) 4
modes
debug 90, 338, 341
edit 338, 343
preview 335
run 338, 341
module
headers 121
modules 35, 37, 38
body 37, 38
example 38
header 37, 38
key 37
library 37
mouse 333
Multitasking
cooperative 41
preemptive 57

N

names 15
#define 16
Next error <CTRL-N> 338
No Background TX 351
nodebug .89, 104, 125, 134, 341, 342, 344, 348
norst 125
nouseix 109, 125
NULL 125

O

offsets in assembly language 106, 109, 111, 112, 113
online help 39, 359
operators 139
(macros) 16, 17, 18
(macros) 16, 17, 18
arithmetic operators 140
decrement (--) 142
division (/) 141
increment (++) 141
indirection (*) 141
minus (-) 140
modulus (%) 142
multiplication (*) 141
plus (+) 140
pointers 141
post-decrement (--) 142
post-increment (++) 141
pre-decrement (--) 142
pre-increment (++) 141
assignment operators 142
add assign (+=) 142
AND assign (&=) 143
assign (=) 142
divide assign (/=) 143
modulo assign (%=) 143
multiply assign (*=) 143
OR assign (|=) 144
shift left (<<=) 143
shift right (>>=) 143
subtract assign (-=) 143
XOR assign (^=) 144
associativity 139
binary 139
bitwise operators
address (&) 144
bitwise AND (&) 144
bitwise exclusive OR (^) 145
bitwise inclusive OR (|) 145
complement (~) 145
pointers 144
shift left (<<) 144
shift right (>>) 144
comma 151
conditional operators (? :) 149
equality operators 146
equal (==) 146
not equal (!=) 146
in assembly language 105
logical operators 147
logical AND (&&) 147
logical NOT (!) 147
logical OR (| |) 147
operator precedence 151
postfix expressions 147
( ) parentheses 147
[ ] array indices 147
array subscripts or dimension [ ] 147
dot (.) 148
parentheses ( ) 147
right arrow (->) 148
precedence 139
reference/dereference operators 148
address (&) 148
bitwise AND (&) 148
indirection (*) 149
multiplication (*) 149
relational operators 145
greater than (>) 146
greater than or equal (>=) .. 146
less than (<) 145
less than or equal (<=) .145
sizeof 150
unary 139
Optimize For (size or speed) 347
options
compiler 346, 347, 348
debugger 346, 349
display 346, 350
editor 346
memory 346
serial 346, 351
OPTIONS menu ...27, 346, 347, 348, 349, 350, 351

P

PageDown key 333
PageUp key 333
passing arguments .28, 108, 113, 114, 115
Paste 337
pasting text <CTRL-V> 337
PC 3, 341, 351
pointer checking 27
pointers 19, 26, 28
uninitialized 26
polling 341, 342
ports
serial 351
positioning text 338
power failure 126
preserving registers 114, 115, 116
preview mode 335
Previous error <CTRL-P> ...338
primary register ...107, 113, 114, 115
primitive data types 14
Print 335
Print Preview 335
Print Setup 336
printf 20, 24, 341, 342, 349, 353
program
example 23
program flow .28, 29, 30, 31, 32
programmable ROM 4, 5

programming
  real-time 3
promotion 140
protected 126
protected variables 3, 89, 126
prototypes
  function 22, 24, 37
  in headers 37
punctuation 14

Q

quitting Dynamic C <ALT-F4> 336

R

Rabbit reset
  _sysIsSoftReset 316
  chkHardReset 170
  chkSoftReset 171
  chkWDTO 171
Rabbit restart
  protected variables 126
  sysResetChain 317
RAM
  static 4, 5
read-only memory 4, 5
real-time
  kernel (RTK) 3, 90
  programming 3
redoing changes
  <ALT-SHIFT-backspace> 337
registers 90, 108
  snapshots 355
  variables 26
  window 3, 352, 355
remote target information (RTI) file 339
Replace <F6> 334, 337
replacing text 337, 338
reset
  software 343
Reset program <CTRL-F2> 341, 342
Reset target <CTRL-Y> 341, 343
resetting program 342
restarting
  program 342
  target controller 343
ret 113, 116
reti 116
retn 116
return 113, 126, 130
return address 108, 112
reverse searching 337, 338
ROM 341, 342
  programmable 4, 5
root 97, 127
  memory 4, 97, 106, 107, 109, 110, 112, 127
rst 028h 342
RST 28H 89
RTI (remote target information) file 339
RTK (real-time kernel) 3, 90
Run <F9> 341
RUN menu 341, 342, 343
run mode 338, 341
Run w/ No Polling <ALT-F9> .. 341
running
  a program 341
  in polling mode 341
  with no polling 341
run-time
  checking 347

S

sample programs
  basic C constructs 23
Save 335
Save as 335
Save Environment 352
saving a file 335
scrolling 355
Search for Help 359
searching for text 337, 338
searching in reverse 337, 338
segchain 32, 127
SEGSIZE 95
selecting
  COMPILE menu <ALT-C> .. 339
  HELP menu <ALT-H> ... 356
  OPTIONS menu <ALT-O> ... 346
  RUN menu <ALT-R> 341
  WINDOW menu <ALT-W> . 352
serial communication 351
serial options 346, 351
serial port 351
shared 127
shared variables 3, 89, 126
SHIFT-F5 338
short 128
Show Tool Bar 351
single stepping 90, 104, 344
  in assembly language 89
  with descent <F7> 342
  without descent <F8> 342
size 128
sizeof 128
skipping to next loop pass 30
Slice Statements 57
soft break points 341, 342
software
  libraries 35, 37
  reset 343
source window 352
SP (stack pointer) 108, 114, 115, 116, 136
special characters 20
special symbols
  in assembly language 106
speed 128
stack 28, 90, 108, 109, 110, 111, 112, 113, 114, 115, 116, 117, 125
  checking 89, 90
  frame 108, 110, 112, 113, 114, 115, 116
  frame reference point 113, 114
  frame reference pointer 90, 111, 113, 125
  pointer (SP) 108, 114, 115, 116, 136
  snapshots 355
  window 3, 352, 355
STACKSEG 95
standalone
  assembly code 107
state machine
  example 43
statements 21
static 129
  RAM 4, 5
  variables 5, 106, 108
status register (F) 355
STDIO window 3, 349, 352, 353
Step over <F8> 341, 342
Stop <CTRL-Z> 341
stop bits 351
stopping a running program 341
storage class 21, 108
  auto 25
  default 5
  register 25, 26
  static 25
strcpy 359

strings 19, 20, 132, 134, 135
functions 19
terminating null byte 19
struct ...21, 24, 25, 28, 107, 108, 113, 114, 115, 129
structures .24, 25, 107, 108, 113, 114, 115
return space 108, 113, 114, 115
subscripts
array 24
support files 36
switch 32, 121, 130
breaking out of 30
case 130
switching to edit mode 338
symbolic constant 134
Sync. Bkgnd TX 351

T

targetless compilation 339
text cursor 359
Tile Horizontally 352, 353
tiling windows 352, 353
Toggle break point <F2> 341, 342
Toggle hard break point <ALT-F2> 341, 342
Toggle interrupt <CTRL-I> 341, 342
Toggle polling <CTRL-O> .341, 342
toolbar 351
Trace into <F7> 341, 342
type
checking 22, 348
conversion 140
definitions 23
type casting 140
typedef 23, 130
types
function 22

U

unary operators 139
unbalanced stack 116
undoing changes <ALT-Backspace> 336
uninitialized
pointers 26
union 21, 25, 131
unpreserved registers ..114, 115, 116
unsigned 131
untitled files 335
Update Watch window <CTRL-U> 344
useix 90, 110, 131

V

variables
global 25
vertical tiling 352

W

waitfor 131
waitfordone 132
warning reports 347
watch
dialog 343, 344
expressions 344, 353
list 344
window .3, 343, 344, 352, 353
adding items 343, 344
clearing 344
deleting items 343, 344
updating 344
wfd 132
while 21, 29, 132
WINDOW menu .352, 353, 354, 355, 356
windows 352
assembly 3, 104, 352, 354
cascaded 352
information 352, 356
message 352, 353
minimized 353
register 3, 352, 355
stack 3, 352, 355
STDIO 3, 349, 352, 353
tiled horizontally 352, 353
tiled vertically 352
watch 3, 343, 344, 352, 353

X

xdata 132
xmem 112, 132
XPC 95
xstring 133

Y

yield 133

Z

Z180 ....89, 95, 355