



**For Rabbit Semiconductor Microprocessors**

**Integrated C Development System**

**User's Manual**

010823 - N

**SE and Premier Editions**

---

# Dynamic C User's Manual

Part Number 019-0071 • 010823-N

## 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	4.20 Global Initialization.....	33	
1.1	Requirements.....	1	4.21 Libraries.....	35	
1.2	Assumptions.....	1	4.22 Support Files.....	36	
2	Introduction to Dynamic C.....	3	4.23 Headers.....	36	
2.1	The Nature of Dynamic C.....	3	4.24 Modules.....	37	
	Speed.....	3	The Key.....	37	
2.2	Dynamic C Enhancements and		The Header.....	37	
	Differences.....	4	The Body.....	38	
2.3	Dynamic C Differences Between Rabbit		Function Description Headers.....	39	
	and Z180.....	5			
3	Quick Tutorial.....	7	5	Multitasking with Dynamic C.....	41
3.1	Run DEMO1.C.....	7	5.1	Cooperative Multitasking.....	41
	Single-Stepping.....	8	5.2	A Real-time Problem.....	43
	Watch Expression.....	9		Solving the Real-time Problem	
	Breakpoint.....	9		With a State Machine.....	43
	Editing the Program.....	9	5.3	Costatements.....	44
3.2	Run DEMO2.C.....	10		Solving the Real-time Problem	
	Watching Variables Dynamically.....	10		With Costatements.....	44
3.3	Run DEMO3.C.....	10		Costatement Syntax.....	45
	Cooperative Multitasking.....	10		Control Statements.....	45
3.4	Summary of Features.....	12	5.4	Advanced Costatement Topics.....	46
4	Language.....	13		The CoData Structure.....	46
4.1	C Language Elements.....	13		CoData Fields.....	47
4.2	Punctuation and Tokens.....	14		Pointer to CoData Structure.....	48
4.3	Data.....	14		Functions for Use With Named	
	Data Type Limits.....	15		Costatements.....	48
4.4	Names.....	15		Firsttime Functions.....	49
4.5	Macros.....	16		Shared Global Variables.....	49
	Restrictions.....	18	5.5	Cofunctions.....	50
4.6	Numbers.....	19		Syntax.....	50
4.7	Strings and Character Data.....	19		Calling Restrictions.....	51
4.8	Statements.....	21		CoData Structure.....	51
4.9	Declarations.....	21		Firsttime functions.....	51
4.10	Functions.....	22		Types of Cofunctions.....	52
4.11	Prototypes.....	22		Types of Cofunction Calls.....	53
4.12	Type Definitions.....	23		Special Code Blocks.....	54
4.13	Aggregate Data Types.....	24		Solving the Real-time Problem	
	Array.....	24		With Cofunctions.....	55
	Structure.....	24	5.6	Patterns of Cooperative Multitasking.....	55
	Union.....	25	5.7	Timing Considerations.....	56
	Composites.....	25		waitfor Accuracy Limits.....	57
4.14	Storage Classes.....	25	5.8	Overview of Preemptive Multitasking.....	57
4.15	Pointers.....	26	5.9	Slice Statements.....	57
4.16	Pointers to Functions, Indirect Calls.....	27		Syntax.....	57
4.17	Argument Passing.....	28		Usage.....	58
4.18	Program Flow.....	28		Restrictions.....	58
	Loops.....	29		Slice Data Structure.....	59
	Continue and Break.....	30		Slice Internals.....	59
	Branching.....	31	5.10	Summary.....	61
4.19	Function Chaining.....	32	6	The Virtual Driver.....	63
			6.1	Default Operation.....	63
			6.2	Calling _GLOBAL_INIT().....	63

6.3	Global Timer Variables .....	64	Initial Formatting.....	104
6.4	Watchdog Timers .....	65	Logical Extents.....	105
	Hardware Watchdog .....	65	Logical Sector Size.....	106
	Virtual Watchdogs .....	65	11.5 File Identifiers .....	107
6.5	Preemptive Multitasking Drivers .....	65	File Numbers .....	107
7	The Slave Port Driver .....	67	File Names.....	107
7.1	Slave Port Driver Protocol .....	67	11.6 Use of First Flash in FS1 .....	108
	Overview .....	67	11.7 Use of First Flash in FS2.....	109
	Registers on the Slave .....	67	11.8 Skeleton Program Using FS1 .....	110
	Polling and Interrupts .....	68	11.9 Skeleton Program Using FS2 .....	111
	Communication Channels .....	69	12 Using Assembly Language .....	113
7.2	Functions.....	69	12.1 Mixing Assembly and C.....	113
7.3	Examples .....	72	Embedded Assembly Syntax.....	113
	Example of a Status Handler .....	72	Embedded C Syntax .....	114
	Example of a Serial Port Handler ...	73	12.2 The Assembler and the Preprocessor ...	114
	Example of a Byte Stream Handler	83	Comments.....	114
8	Efficiency.....	89	Defining Constants .....	115
8.1	Nodebug Keyword .....	89	Multiline Macros .....	115
8.2	Static Variables.....	89	Labels .....	116
8.3	Function Entry and Exit .....	90	Special Symbols .....	116
9	Run-Time Errors.....	91	C Variables .....	116
9.1	Run-Time Error Handling .....	91	12.3 Stand-alone Assembly Code .....	117
	Error Code Ranges .....	91	Example of Stand-Alone Assembly	
	Fatal Error Codes.....	92	Code .....	118
9.2	User-Defined Error Handler.....	93	12.4 Embedded Assembly Code .....	118
	Replacing the Default Handler .....	93	The Stack Frame.....	118
9.3	Run-Time Error Logging .....	94	Example of Embedded Assembly	
	Error Log Buffer.....	94	Code .....	120
	Initialization and Defaults .....	95	Local Variable Access .....	122
	Configuration Macros.....	95	12.5 C Functions Calling Assembly Code ...	123
	Error Logging Functions .....	96	Passing Parameters .....	123
	Examples of Error Log Use.....	96	Location of Return Results.....	123
10	Memory Management .....	97	12.6 Assembly Code Calling C Functions ...	125
10.1	Memory Map.....	97	Interrupt Routines in Assembly .....	126
	Memory Mapping Control.....	98	12.8 Common Problems.....	127
10.2	Extended Memory Functions .....	98	13 Keywords.....	129
	Code Placement in Memory .....	98	abandon.....	129
11	The Flash File System .....	99	abort .....	129
11.1	General Usage .....	99	always_on .....	129
	Using SRAM .....	100	anymem .....	129
	Wear Leveling.....	100	auto .....	130
	Low-level Implementation .....	100	break .....	130
	Multitasking and the File System .	100	case .....	130
11.2	Application Requirements .....	100	char .....	130
	FS1 Requirements .....	100	const.....	131
	FS2 Requirements .....	101	continue .....	132
11.3	Functions .....	102	costate .....	132
	FS1 API.....	102	debug .....	132
	FS2 API.....	103	default .....	133
11.4	Setting up and Partitioning the File		do .....	133
	System.....	104		



.....	159	AESencrypt .....	176
! .....	159	AESencryptStream .....	176
14.7 Postfix Expressions .....	159	AESexpandKey .....	177
() .....	159	AESinitStream .....	178
[] .....	159	asec .....	179
. (dot) .....	160	asin .....	179
-> .....	160	atan .....	180
14.8 Reference/Dereference Operators ...	160	atan2 .....	181
& .....	160	atof .....	182
* .....	161	atoi .....	182
14.9 Conditional Operators .....	161	atol .....	183
?: .....	161	bit .....	183
14.10 Other Operators .....	162	BIT .....	184
(type) .....	162	BitRdPortE .....	184
sizeof .....	162	BitRdPortI .....	185
, .....	163	BitWrPortE .....	186
BitWrPortI .....	187	BitWrPortI .....	187
15 Function Reference .....	165	ceil .....	188
15.1 Functional Groups .....	165	chkHardReset .....	188
Arithmetic .....	165	chkSoftReset .....	189
Bit Manipulation .....	165	chkWDTO .....	189
Character .....	165	clockDoublerOn .....	190
Data Encryption .....	165	clockDoublerOff .....	190
Error Handling .....	165	CoBegin .....	191
Extended Memory .....	165	cof_pktXreceive .....	191
Fast Fourier Transforms .....	166	cof_pktXsend .....	192
File System .....	166	cof_serXgetc .....	193
Floating-point Math .....	167	cof_serXgets .....	194
Low-level Flash Access .....	167	cof_serXputc .....	195
GPS .....	167	cof_serXputs .....	196
I <sup>2</sup> C Bus .....	167	cof_serXread .....	197
I/O .....	168	cof_serXwrite .....	198
Interrupts .....	168	CoPause .....	199
MicroC/OS-II .....	169	CoReset .....	199
Miscellaneous .....	169	CoResume .....	200
Multitasking .....	170	cos .....	200
Number-to-string Conversion .....	170	cosh .....	201
Real-time Clock .....	170	defineErrorHandler .....	202
Serial Communication .....	170	deg .....	203
Serial Communication .....	171	DelayMs .....	203
SPI .....	171	DelaySec .....	204
STDIO .....	171	DelayTicks .....	204
String Manipulation .....	171	Disable_HW_WDT .....	205
String-to-number Conversion .....	171	errlogGetHeaderInfo .....	206
System .....	172	errlogGetNthEntry .....	207
User Block .....	172	errlogFormatEntry .....	207
Watchdog .....	172	errlogFormatRegDump .....	208
15.2 Alphabetical Listing .....	173	errlogFormatStackDump .....	208
abs .....	173	errlogGetMessage .....	209
acos .....	173	errlogReadHeader .....	209
acot .....	174	exception .....	210
acsc .....	174	exit .....	210
AESdecrypt .....	175	exp .....	211
AESdecryptStream .....	175	fabs .....	211

fclose	212	gets	258
fcreate (FS1)	212	GetVectExtern2000	258
fcreate (FS2)	213	GetVectIntern	259
fcreate_unused (FS1)	214	gps_get_position	259
fcreate_unused (FS2)	215	gps_get_utc	260
fdelete (FS1)	215	gps_ground_distance	260
fdelete (FS2)	216	hanncplx	261
fflush (FS2)	217	hannreal	262
fftcpplx	218	hitwd	263
fftcpplxinv	219	htoa	263
fftreal	220	IntervalMs	264
fftrealinv	221	IntervalSec	264
flash_erasechip	222	IntervalTick	265
flash_erasector	222	ipres	265
flash_gettype	223	ipset	266
flash_init	224	isalnum	266
flash_read	225	isalpha	267
flash_readsector	226	iscntrl	267
flash_sector2xwindow	227	isCoDone	268
flash_writesector	228	isCoRunning	268
floor	229	isdigit	269
fmod	229	isgraph	269
fopen_rd (FS1)	230	islower	270
fopen_rd (FS2)	230	isspace	270
fopen_wr (FS1)	231	isprint	271
fopen_wr (FS2)	232	ispunct	272
forceSoftReset	233	isupper	273
fread (FS1)	233	isxdigit	273
fread (FS2)	234	itoa	274
frexp	235	i2c_check_ack	274
fs_format (FS1)	236	i2c_init	275
fs_format (FS2)	237	i2c_read_char	275
fs_init (FS1)	238	i2c_send_ack	276
fs_init (FS2)	239	i2c_send_nak	276
fs_reserve_blocks (FS1)	240	i2c_start_tx	277
fsck (FS1)	240	i2c_startw_tx	277
fseek (FS1)	241	i2c_stop_tx	278
fseek (FS2)	242	i2c_write_char	278
fs_get_flash_lx (FS2)	243	kbhit	279
fs_get_lx (FS2)	244	labs	279
fs_get_lx_size (FS2)	245	ldexp	280
fs_get_other_lx (FS2)	246	log	281
fs_get_ram_lx (FS2)	247	log10	281
fs_set_lx (FS2)	248	longjmp	282
fs_setup (FS2)	249	loophead	282
fs_sync (FS2)	251	loopinit	283
ftell (FS1)	252	ltoa	283
ftell (FS2)	253	ltoan	284
fshift	254	lx_format	285
fwrite (FS1)	254	memchr	286
fwrite (FS2)	255	memcmp	287
ftoa	256	memcpy	288
getchar	256	memmove	288
getcrc	257	memset	289

mktime .....	289	pktXclose.....	326
mktime .....	290	pktXgetErrors .....	326
modf.....	291	pktXinitBuffers.....	327
OSInit .....	291	pktXopen .....	328
OSMboxAccept .....	292	pktXreceive .....	329
OSMboxCreate.....	292	pktXsend.....	330
OSMboxPend .....	293	pktXsending.....	330
OSMboxPost.....	294	pktXsetParity .....	331
OSMboxQuery .....	295	poly .....	331
OSMemCreate .....	296	pow .....	332
OSMemGet.....	297	pow10 .....	332
OSMemPut .....	297	powerspectrum .....	333
OSMemQuery.....	298	premain .....	334
OSQAccept.....	298	printf .....	334
OSQCreate.....	299	putchar .....	335
OSQFlush .....	299	puts .....	335
OSQPend .....	300	qsort .....	336
OSQPost .....	301	rad .....	337
OSQPostFront.....	302	rand .....	338
OSQQuery .....	303	randb.....	338
OSSchedLock .....	303	randg.....	339
OSSchedUnlock .....	304	RdPortE .....	339
OSSemAccept.....	304	RdPortI .....	340
OSSemCreate .....	305	read_rtc.....	340
OSSemPend.....	305	read_rtc_32kHz .....	341
OSSemPost .....	306	readUserBlock .....	341
OSSemQuery .....	307	res .....	342
OSSetTickPerSec .....	308	RES.....	343
OSStart .....	308	ResetErrorLog .....	343
OSStatInit .....	309	root2xmem.....	344
OSTaskChangePrio .....	309	runwatch .....	344
OSTaskCreate.....	310	serCheckParity.....	345
OSTaskCreateExt .....	311	serXclose .....	345
OSTaskCreateHook.....	312	serXdatabits .....	346
OSTaskDel .....	313	serXflowcontrolOff .....	346
OSTaskDelHook.....	313	serXflowcontrolOn.....	347
OSTaskDelReq.....	314	serXgetc.....	348
OSTaskQuery .....	315	serXgetError .....	349
OSTaskResume .....	316	serXopen.....	350
OSTaskStatHook .....	316	serXparity .....	351
OSTaskStkChk .....	317	serXpeek .....	352
OSTaskSuspend.....	318	serXputc.....	352
OSTaskSwHook .....	318	serXputs.....	353
OSTimeDly.....	319	serXrdFlush .....	353
OSTimeDlyHMSM .....	320	serXrdFree .....	354
OSTimeDlyResume.....	321	serXrdUsed .....	354
OSTimeDlySec.....	322	serXread.....	355
OSTimeGet.....	322	serXwrFlush .....	356
OSTimeSet .....	323	serXwrFree .....	356
OSTimeTickHook .....	323	serXwrite .....	357
OSVersion .....	324	set.....	358
outchr .....	324	SET .....	358
outstr .....	325	setjmp .....	359
paddr .....	325	SetVectExtern2000.....	360

SetVectIntern .....	361
sin .....	361
sinh .....	362
SPIInit.....	363
SPIRead.....	363
SPIWrite.....	364
sprintf .....	365
sqrt.....	366
strcat .....	366
strchr.....	367
strcmp.....	368
strcmpi.....	369
strcpy .....	370
strcspn .....	370
strlen.....	371
strncat .....	371
strncmp.....	372
strncmpi.....	373
strncpy .....	374
strpbrk .....	375
strrchr .....	375
strspn .....	376
strstr.....	376
strtod.....	377
strtok.....	378
strtol .....	379
_sysIsSoftReset .....	379
sysResetChain .....	380
tan.....	380
tanh.....	381
tm_rd .....	382
tm_wr .....	383
tolower .....	384
toupper .....	384
updateTimers.....	385
use32HzOsc .....	385
useClockDivider.....	386
useMainOsc.....	386
utoa.....	387
VdGetFreeWd .....	387
VdHitWd.....	388
VdInit .....	388
VdReleaseWd.....	389
WriteFlash2.....	390
write_rtc .....	391
writeUserBlock .....	392
WrPortE .....	393
WrPortI.....	393
xalloc .....	394
xmem2root .....	395
xmem2xmem.....	396
16 Graphical User Interface.....	397
16.1 Editing.....	397
16.2 Menus.....	398
File Menu.....	398
New.....	399
Open.....	399
Save.....	399
Save As .....	399
Close .....	399
Print Preview.....	399
Print.....	399
Print Setup.....	400
Exit .....	400
Edit Menu.....	400
Undo.....	400
Redo .....	400
Cut .....	401
Copy.....	401
Paste .....	401
Find .....	401
Replace.....	401
Find Next .....	402
Goto .....	402
Previous Error.....	402
Next Error .....	402
Edit Mode .....	402
Compile Menu.....	402
Compile to Target .....	402
Compile to .bin file .....	403
Reset Target/Compile BIOS ..	404
Include Debug Code/RST	
28 Instructions .....	404
Run Menu.....	405
Run .....	405
Run w/ No Polling .....	405
Stop .....	405
Reset Program.....	406
Trace Into .....	406
Step over .....	406
Toggle Breakpoint.....	406
Toggle Hard Breakpoint.....	406
Toggle Interrupt Flag .....	406
Toggle Polling.....	406
Reset Target .....	406
Close Serial Port .....	406
Inspect Menu.....	407
Add/Del Watch Expression...407	
Clear Watch Window .....	408
Update Watch Window .....	408
Disassemble at Cursor .....	408
Disassemble at Address .....	408
Dump at Address .....	409
Options Menu.....	410
Editor.....	410
Compiler.....	411
Run-Time Checking.....	411
BIOS Memory Setting .....	411
User Defined BIOS File.....	412
User Defined Libraries File ..	412
Watch Expressions .....	412
Type Checking .....	413
Warning Reports .....	413
Optimize For.....	413
Max Shown .....	413
Defines .....	414
Debugger.....	414

Display.....	415	B Map File Generation .....	457
Communications .....	416	Grammar .....	457
TCP/IP Option .....	416	Software License Agreement .....	459
Serial Options .....	416	Index.....	463
Show Tool Bar.....	416		
Save Environment .....	417		
Window Menu .....	417		
Message.....	417		
Watch.....	417		
Stdio .....	418		
Assembly .....	418		
Registers .....	419		
Stack .....	419		
Information.....	420		
Help Menu .....	420		
Online Documentation .....	420		
Keywords .....	420		
Operators .....	420		
HTML Function Reference ...	420		
Function Lookup/Insert .....	421		
Instruction Set Reference .....	423		
Keystrokes.....	423		
Contents.....	423		
About.....	423		
17 Command Line Interface .....	425		
17.1 Default States .....	425		
17.2 User Input.....	425		
17.3 Saving Output to a File.....	425		
17.4 Command Line Switches .....	426		
Switches Without Parameters .....	426		
Switches Requiring a Parameter...	430		
17.5 Examples .....	435		
Example 1 .....	435		
Example 2.....	435		
18 $\mu$ C/OS-II .....	437		
18.1 Changes to $\mu$ C/OS-II.....	437		
Ticks per Second .....	437		
Task Creation.....	438		
Restrictions .....	439		
18.2 Tasking Aware Interrupt Service			
Routines (TA-ISR) .....	439		
Interrupt Priority Levels .....	439		
Possible ISR Scenarios.....	440		
General Layout of a TA-ISR .....	441		
18.3 Library Reentrancy.....	445		
18.4 How to Get a $\mu$ C/OS-II Application			
Running.....	446		
Default Configuration.....	446		
Custom Configuration .....	447		
Examples .....	448		
18.5 Compatibility with TCP/IP.....	451		
A Macros and Global Variables .....	453		
Compiler-Defined Macros .....	453		
Global Variables.....	454		
Exception Types.....	455		
Rabbit 2000 Internal registers.....	455		

# 1. Installing Dynamic C

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

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

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

## 1.1 Requirements

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

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

## 1.2 Assumptions

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

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

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

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

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



# 2. Introduction to Dynamic C

Dynamic C is an integrated development system for writing embedded software. It 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 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, *co-statements* 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 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 97, 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 129.

- 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 be 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. It is important to note that the `#use` directive is a replacement for the `#include` directive, and the `#include` directive is not supported.
- When declaring [pointers to functions](#), arguments should not be used in the declaration. Arguments may be used when calling functions indirectly via pointer, but the compiler will not check the argument list in the call for correctness.
- Bit fields 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

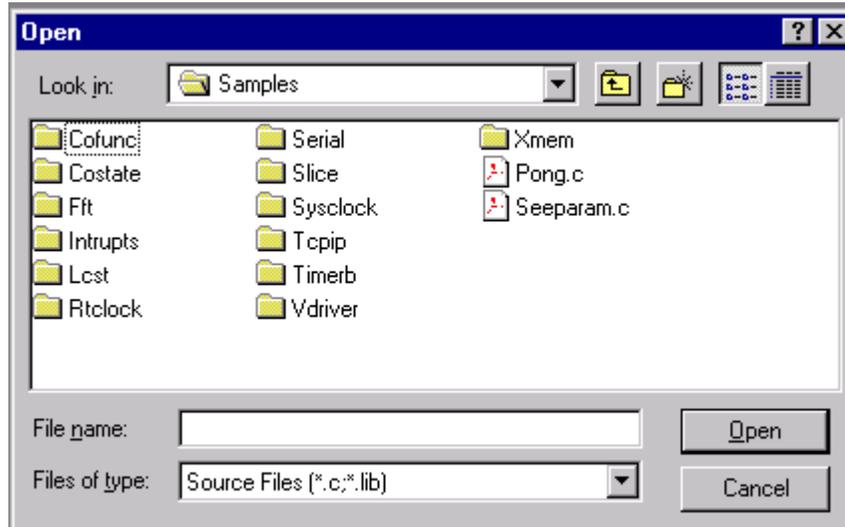
A major difference in the way Dynamic C interacts with a Rabbit-based board compared to a Z180 or 386EX board is that Dynamic C expects no BIOS kernel to be present on the target when it starts up. Dynamic C stores the BIOS kernel as a C source file. Dynamic C compiles and loads it to the Rabbit target 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.



# 3. Quick Tutorial

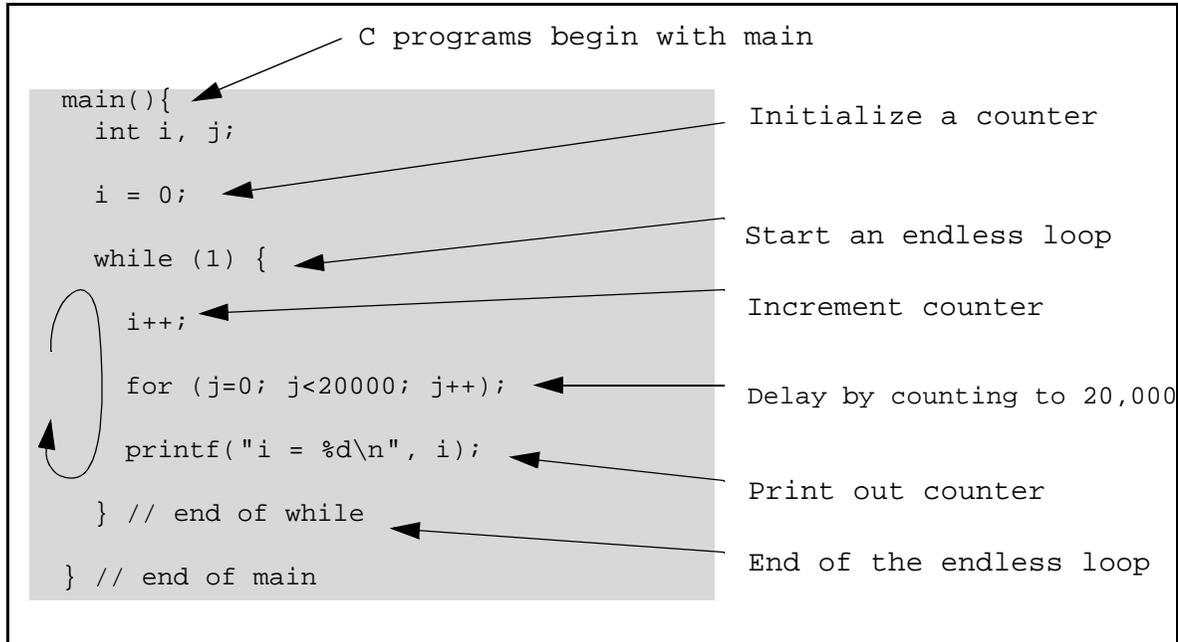
Sample programs are provided in the Dynamic C **samples** folder, shown below.



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.

```

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

    // First task will print the seconds elapsed.

(2)     costate {
        secs++;              // increment counter
(3)     waitfor( DelayMs(1000) ); // wait one second
        printf("%d seconds\n", secs); // print elapsed 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 numbers in the left margin are reference indicators and not part of the code. Load and run the program. The elapsed time is printed to the STDIO window once per second. Push several keys and note how they are reported.

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

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

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.

### 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.

### 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.

### Setting breakpoints

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

### Watch expressions

A watch expression is a C expression that is evaluated on command in the watch window. An expression is basically any type of C 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.

### Costatements

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

# 4. Language

Dynamic C is based on the C language. The programmer is expected to know programming methodologies and the basic principles of the C language. Dynamic C has its own set of libraries, which include user-callable functions. (See “Function Reference” on page 165.) Dynamic C libraries are in source code, allowing the creation of customized libraries.

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

## 4.1 C Language Elements

A Dynamic C program is a set of files, 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 <b>do</b> 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
<b>char</b>	8-bit unsigned integer. Range: 0 to 255 (0xFF)
<b>int</b>	16-bit signed integer. Range: -32,768 to +32,767
<b>unsigned int</b>	16-bit unsigned integer. Range: 0 to +65,535
<b>long</b>	32-bit signed integer. Range: -2,147,483,648 to +2,147,483,647
<b>unsigned long</b>	32-bit unsigned integer. Range 0 to $2^{32} - 1$
<b>float</b>	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: $1.18 \times 10^{-38}$ to $3.40 \times 10^{38}$

### 4.3.1 Data Type Limits

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 0xFFFFFFFF**

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

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

**0177 020000 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  $456 \times 10^{-31}$   
**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.

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

## 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-checking on the parameters to make sure that calls to the function receive arguments of the expected type. A function prototype describes how to call the function and is nearly identical to the function's initial code.

```
/* This is a function prototype.*/
long tick_count ( char clock_id );

/* This is the function's definition.*/
long tick_count ( char clock_id ){
    ...
}
```

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

```
/* This prototype is as good as the one above. */
long tick_count ( char );

/* This is a prototype that uses ellipsis. */
int startup ( device id, ... );
```

## 4.12 Type Definitions

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

```
typedef int MILES; // a basic type named MILES
typedef struct { // a structure type...
    float re; // ...
    float im; // ...
} COMPLEX; // ...named COMPLEX
MILES distance; // declare variable of type MILES
COMPLEX z, *zp; // declare 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[] = "ABCDEFGH";
```

### 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—available anywhere in the program—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 unallocated memory. However, if an uninitialized pointer happens to contain the address of a memory location that the compiler has already allocated, pointer checking will not catch this logic error. Because pointer checking is a run-time option, pointer checking adds instructions to code when pointer checking is used.

## 4.16 Pointers to Functions, Indirect Calls

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

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

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

for example:

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

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

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

You can pass arguments to functions that are called indirectly by 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<sub>1</sub>*), evaluates a terminating condition (*exp<sub>2</sub>*), and provides a stepping expression (*exp<sub>3</sub>*) 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**<sub>N</sub> values matches the **switch** *expression*, the sequence of statements identified by the **const**<sub>N</sub> 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, and other kinds of tasks on request. There are two directives, **#makechain** and **#funcchain**, and one keyword, **segchain**.that create and control function chains:

**#makechain** *chain\_name*

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

**#funcchain** *chain\_name name*

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

**segchain** *chain\_name { statements }*

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

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

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

A program will call a function chain as it would an ordinary void function that has no parameters. The following example shows how to call a function chain that is named **recover**.

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

## 4.20 Global Initialization

Various hardware devices in a system need to be initialized not only by setting variables and control registers, but often by complex initialization procedures. Dynamic C provides a specific function chain, **\_GLOBAL\_INIT**, for this purpose.

Your program can 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.

```
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
}
```

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.

## 4.21 Libraries

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

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

```
main(){  
}
```

Libraries (both user defined and Z-World defined) are “linked” with the application through the **#use** directive. The **#use** directive identifies a file from which functions and data may be extracted. Files identified by **#use** directives are nestable, as shown below. The **#use** directive is a replacement for the **#include** directive, which is not supported in Dynamic C. Any library that is to be used in a Dynamic C program must be listed in the file **LIB.DIR**, or another **\*.DIR** file specified by the user. (Starting with version Dynamic C 7.05, a different **\*.DIR** file may be specified by the user in the Compiler Options dialog to facilitate working on multiple projects.)

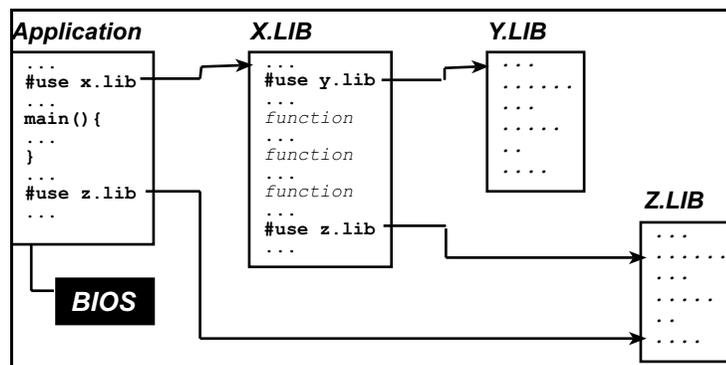


Figure 2. Nesting Files in Dynamic C

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 Name	Purpose of File
<b>DCW.CFG</b>	Contains configuration data for the target controller.
<b>DC.HH</b>	Contains prototypes, basic type definitions, <b>#define</b> , and default modes for Dynamic C. This file can be modified by the programmer.
<b>LIB.DIR</b>	Contains pathnames for all libraries that are to be known to Dynamic C. The programmer can add to, or remove libraries from this list. The factory default is for this file to contain all the libraries on the Dynamic C distribution disk. Any library that is to be used in a Dynamic C program must be listed in the file <b>LIB.DIR</b> , or another <b>*.DIR</b> file specified by the user. (Starting with version Dynamic C 7.05, a different <b>*.DIR</b> file may be specified by the user in the Compiler Options dialog to facilitate working on multiple projects.)
<b>DEFAULT.H</b>	Contains a set of <b>#use</b> 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 Name	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

To write a custom source library, modules must be understood because they provide Dynamic C with the ability to know which functions and global variables in a library to use. It is important to note that the **#use** directive is a replacement for the **#include** directive, and the **#include** directive is not supported.

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 **#uses** a library, Dynamic C compiles every header, and just the headers, in the library. The purpose of a header is to make certain names defined in a module known to the application. With proper function prototypes and variable declarations, a module header ensures proper type checking throughout the application program. 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** (or a user defined replacement for **LIB.DIR**) and a **#use** directive for the library must be placed somewhere in the code.

It should be noted that there is no way to define file scope variables other than having a file consist of a single module (which would mean that all data and functions in the file would be compiled whenever a function specified in the header is compiled).

#### 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.



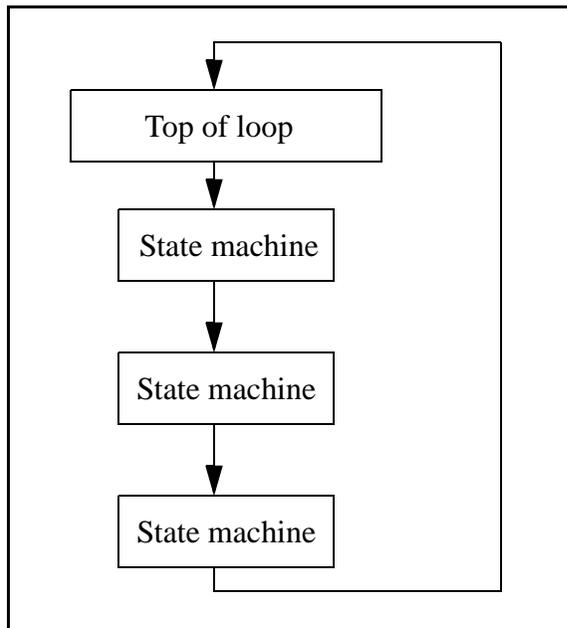
# 5. Multitasking with Dynamic C

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

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

## 5.1 Cooperative Multitasking

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



**Figure 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 be used waiting for the task, leaving no execution time for other tasks.

### 5.2.1 Solving the Real-time Problem With a State Machine

Here is what a state machine solution might look like.

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

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

## 5.3 Costatements

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

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

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

### 5.3.1 Solving the Real-time Problem With Costatements

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

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

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

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

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

### 5.3.2 Costatement Syntax

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

The keyword **costate** identifies the statements enclosed in 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 **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.

<b>STOPPED</b>	<b>INIT</b>	<b>State of Costatement</b>
yes	yes	Done, or has been initialized to run, but set to inactive. Set by <b>CoReset()</b> .
yes	no	Paused, waiting to resume. Set by <b>CoPause()</b> .
no	yes	Initialized to run. Set by <b>CoBegin()</b> .
no	no	Running. <b>CoResume()</b> will return the flags to this state.

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

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

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

### Last Location

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

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

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

### Check Sum

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

### First Time

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

## Content

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

## Check Sum 2

The `ChkSum2` field is currently unused.

### 5.4.3 Pointer to CoData Structure

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

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

### 5.4.4 Functions 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.

<b>DelayMs</b>	<b>IntervalMs</b>
<b>DelaySec</b>	<b>IntervalSec</b>
<b>DelayTicks</b>	<b>IntervalTick</b>

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

### 5.5.5.1 Simple Cofunction

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

### 5.5.5.2 Indexed Cofunction

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

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

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

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

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

#### 5.5.5.2.1 Indexed Cofunction Restrictions

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

### 5.5.5.3 Single User Cofunction

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

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

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

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

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

### loopinit()

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

### loophead()

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

### Example

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

## 5.5.6 Types of Cofunction Calls

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

### 5.5.6.1 First Time Call

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

### 5.5.6.2 Continuation Call

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

### 5.5.6.3 Terminal Call

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

#### 5.5.6.4 Lifetime of a Cofunction Instance

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

#### 5.5.7 Special Code Blocks

The following special code blocks can appear inside a cofunction.

##### **everytime** { *statements* }

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

##### **abandon** { *statements* }

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

#### Example

**SAMPLES/COFUNC/COFABAND.C** illustrates the use of **abandon**.

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

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

## 5.5.8 Solving the Real-time Problem With Cofunctions

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

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 satisfied. 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.

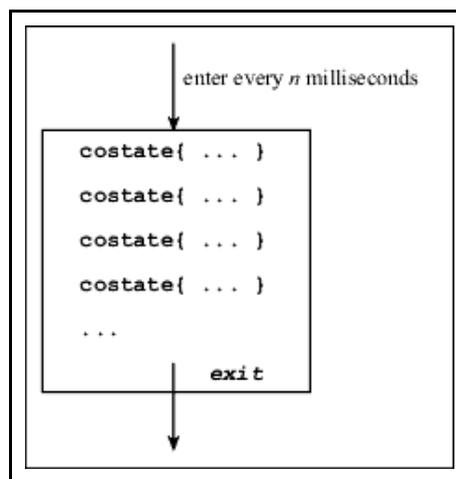
Some tasks may not have an end. They are endless loops. For example, a task to control a servo loop may run continuously to regulate the temperature in an oven. If there are a number of tasks that need to run continuously, then they can be called using a single `waitfordone` statement as shown below.

```
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  $\mu$ C/OS-II, a real-time, preemptive kernel that runs on the Rabbit Microprocessor and is fully supported by Dynamic C. For more information see Chapter 18, “ $\mu$ C/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  $\mu$ C/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.



# 6. The Virtual Driver

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

## Initialization Services

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

## Periodic Interrupt Services

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

## 6.1 Default Operation

The user should be aware that by default, the Virtual Driver starts and runs in a Dynamic C program without the user doing anything. This happens because before `main()` is called, a function called `premain()` is called by the Rabbit kernel (BIOS) that actually calls `main()`. Before `premain()` calls `main()`, it calls a function named `VdInit()` that performs the initialization services, including starting 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 `SEC_TIMER`.

The periodic interrupt updates `SEC_TIMER` every second, `MS_TIMER` every millisecond, and `TICK_TIMER` 1024 times per second (the frequency of the periodic interrupt). These variables are used by the `DelaySec`, `DelayMS` and `DelayTicks` functions, but are also convenient for 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 time your system will be in an unknown state.

### 6.4.1 Hardware Watchdog

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

```
#asm
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.

### 6.4.2 Virtual Watchdogs

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

The function `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.*



# 7. The Slave Port Driver

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.

<b>SPD0R</b>	Command and response register
<b>SPD1R</b>	Address register
<b>SPD2R</b>	Optional data register
<b>SPSR</b>	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 SLAVE-ATTN 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 command to
2	Gets returned data from slave	Sets data byte to send to slave
3	Gets slave port status (see below)	Clears slave response bit (see below)

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

Pin assignments for 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 Microprocessor User's Manual*.

### 7.1.3 Polling and Interrupts

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

## 7.1.4 Communication Channels

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

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

<b>mode</b>	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

<b>address</b>	The 8-bit slave port address of the channel that corresponds to the handler function.
<b>handler</b>	Pointer to the handler function. This function must have a particular form, which is described by the function description for <b>MyHandler()</b> shown below. Setting this parameter to <b>NULL</b> unloads the current handler.
<b>handler_params</b>	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

<b>command</b>	This is the received command byte.
<b>data_in</b>	The optional data byte
<b>params</b>	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 Status Handler

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

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

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

## 7.3.2 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 logical OR of the two command responses.
4	Set baud factor, byte 1 (LSB). The actual baud rate is the baud factor multiplied by 300.
5	Set baud factor, byte 2 (MSB). The actual baud rate is the baud factor multiplied by 300.
6	Set port configuration bits
7	Open port
8	Close port
9	Get errors. Slave responds with 1 if the port is open and can return an error bitfield. The error bits are the same as for the function <code>serAgetErrors()</code> 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 read buffer. The two commands return the LSB and the MSB of the count respectively. The LSB(16) should be read first to latch the count.

### 7.3.2.2 Slave Side of Protocol

To set up the 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

<b>address</b>	Slave channel address of serial handler
<b>buffer</b>	Buffer to store received bytes
<b>length</b>	Size of buffer
<b>timeout</b>	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.

<b>address</b>	Slave channel address of serial handler
<b>data</b>	Array to be transmitted
<b>length</b>	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 received character;  
-1: No character available.

### **LIBRARY**

MASTER\_SERIAL.LIB

## MSgetError

```
int MSError(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

<b>address</b>	Slave channel address of serial handler.
<b>baud</b>	Baud rate for the serial port on the slave.

### RETURN VALUE

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

### LIBRARY

MASTER\_SERIAL.LIB

## MSputc

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

### DESCRIPTION

Transmits a single character through the serial port.

### PARAMETERS

<b>address</b>	Slave channel address of serial handler
<b>ch</b>	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 occurred 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

<b>address</b>	Slave channel address of serial handler.
<b>buffer</b>	Array to put received data into.
<b>size</b>	Size of array (max bytes to be read).
<b>timeout</b>	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

<b>address</b>	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

<b>address</b>	Slave channel address of serial handler.
<b>data</b>	Array of bytes to send.
<b>length</b>	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 auxiliary functions that supports the byte stream handler. This is also shown in the sample program.

##### 7.3.3.1.1 Functions

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

#### `cbuf_init`

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

#### DESCRIPTION

This function initializes a circular buffer.

#### PARAMETER

<code>circularBuffer</code>	The circular buffer to initialize.
<code>dataSize</code>	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

<b>stream</b>	Pointer to the stream state structure.
<b>data</b>	Structure to read from slave port buffer.
<b>length</b>	Number of bytes to read.
<b>tmout</b>	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

<b>stream</b>	Pointer to the stream state structure.
<b>data</b>	Structure to write to slave port buffer.
<b>length</b>	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

<b>stream</b>	Pointer to the stream state structure.
<b>data</b>	Buffer to read received data into.
<b>length</b>	Maximum number of bytes to read.
<b>tmout</b>	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

<b>stream</b>	Pointer to the stream state structure.
<b>data</b>	Bytes to write to stream.
<b>length</b>	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
 */
#include "slave_port.lib"
#include "sp_stream.lib"

#define STREAM_BUFFER_SIZE 31

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

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

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

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

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

# 8. Efficiency

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.
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.

# 9. Run-Time Errors

Compiled code generated by Dynamic C calls an exception handling routine for run-time errors. The exception handler supplied with Dynamic C prints internally defined error messages to a Windows message box when run-time errors are detected during a debugging session. When software runs stand-alone (disconnected from Dynamic C), such a run-time error will cause a watchdog timeout and reset. Starting with Dynamic C 7.05, run-time error logging is available for Rabbit-based target systems with battery-backed RAM.

## 9.1 Run-Time Error Handling

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

### 9.1.1 Error Code Ranges

The table below shows the range of error codes used by Dynamic C and the range available for a custom error handler to use. Please see section 9.2 on page 93 for more information on replacing the default error handler with a custom one.

*Table 6. Dynamic C Error Types Ranges*

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

## 9.1.2 Fatal Error Codes

This table lists the fatal errors generated by Dynamic C.

**Table 7. Dynamic C Fatal Errors**

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

## 9.2 User-Defined Error Handler

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

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

### 9.2.1 Replacing the Default Handler

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

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

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

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

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

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

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

Dynamic C expects the following values to be loaded:

**Table 9. Register contents loaded by error handler before passing the error to Dynamic C**

Register	Expected Value
H	XPC when <code>exception()</code> was called
L	Run-time error code
HL'	Address where <code>exception()</code> was called from

## 9.3 Run-Time Error Logging

Starting with Dynamic C 7.05, error logging is available as a BIOS enhancement for storing run-time exception history. It can be useful diagnosing problems in deployed Rabbit targets. To support error logging, the target must have battery-backed RAM.

### 9.3.1 Error Log Buffer

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

- The value of **SEC\_TIMER** at the time of the error. This variable contains the number of seconds since 00:00:00 on January 1st 1980 if the real-time clock has been set correctly. This variable is updated by the periodic timer which is enabled by default. Z-World sets the real-time clock in the factory. When the BIOS starts on boards with batteries, it initializes **SEC\_TIMER** to the value in the real-time clock.
- The address where the exception was called from. This can be traced to a particular function using the MAP file generated when a Dynamic C program is compiled.
- The exception type. Please see Table 7 on page 92 for a list of exception types.
- The value of all registers. This includes alternate registers, SP and XPC. This is a global option that is enabled by default.
- An 8 byte message. This is a global option that is disabled by default. The default error handler does nothing with this.
- A user-definable length of stack dump. This is a global option that is enabled by default.
- A one byte checksum of the entry

#### 9.3.1.1 Error Log Buffer Size

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

### 9.3.2 Initialization and Defaults

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

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

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

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

### 9.3.3 Configuration Macros

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

#### **ENABLE\_ERROR\_LOGGING**

Default: 0. Enables error logging. Changing this to one in the BIOS disables error logging.

#### **ERRLOG\_USE\_REG\_DUMP**

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

#### **ERRLOG\_STACKDUMP\_SIZE**

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

#### **ERRLOG\_NUM\_ENTRIES**

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

#### **ERRLOG\_USE\_MESSAGE**

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

### 9.3.4 Error Logging Functions

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

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

### 9.3.5 Examples of Error Log Use

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

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

and

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

# 10. Memory Management

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

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

## 10.1 Memory Map

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

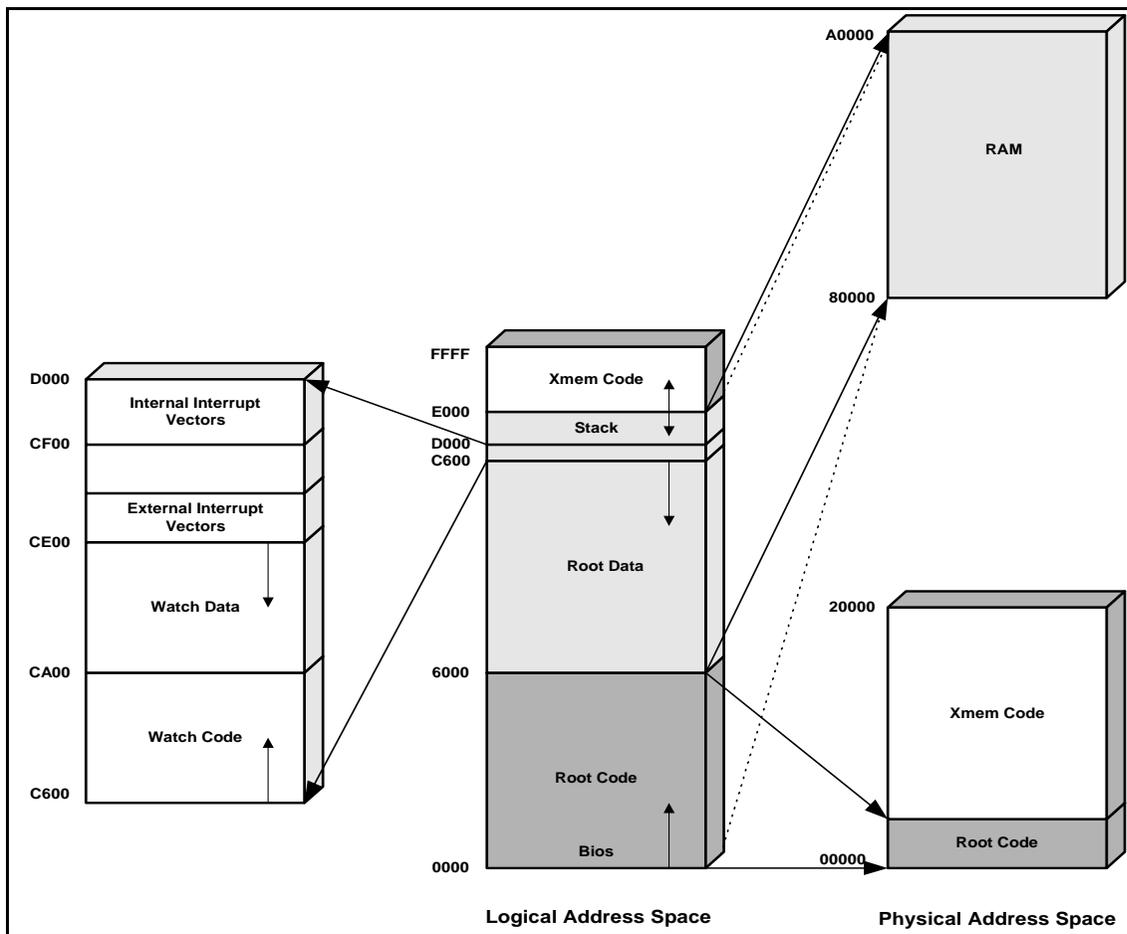


Figure 3. Dynamic C Memory Mapping

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 placed at a starting address 0x80000.

### 10.1.1 Memory Mapping Control

The advanced user of Dynamic C can control how Dynamic C allocates and maps memory. For details on memory mapping, refer to the *Rabbit 2000 Microprocessor User's Manual*.

## 10.2 Extended Memory Functions

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

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

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

### 10.2.1 Code Placement in Memory

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

#### Pure Assembly Routines

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

#### C Functions

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

#### Inline Assembly in C Functions

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

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

# 11. The Flash File System

Dynamic C 7.0 introduced a simple file system that can be used with a second flash memory or in SRAM. Dynamic C 7.05 introduces an improved file system with more features:

- The ability to overwrite parts of a file.
- The simultaneous use of multiple device types.
- The ability to partition devices.
- Efficient support for byte-writable devices.
- Better performance tuning.
- High degree of backwards compatibility with its predecessor.

This file system, known as the filesystem mk II or simply as FS2, uses the same API as the first file system, with some additional functions. Initialization is performed slightly differently, and the data format is not compatible. Z-World recommends that FS2 be used for all new applications. The first file system, which we will refer to as FS1, will be maintained but enhancements will only be implemented for FS2.

The Dynamic C file system supports a total of 255 files. Unlike FS1, it is not possible to “reserve” a range of file numbers for system use with FS2. Equivalent functionality is available via partitioning of devices.

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

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

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

### 11.1.1 Using SRAM

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

Using FS1, since only one device type is allowed at a time, the entire file system would have to be in SRAM. This is recommended for debugging purposes only. Using FS2 increases flexibility, with its capacity to use multiple device types simultaneously. Since RAM is usually a scarce resource, it can be used together with flash memory devices to obtain the best balance of speed, performance and capacity.

### 11.1.2 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.

### 11.1.3 Low-level Implementation

For information on the low-level implementation of the flash file system, refer to the beginning of the library files **FS2.LIB** and **FS\_DEV.LIB** if using FS2, or library file **FILESYSTEM.LIB**, if using FS1.

### 11.1.4 Multitasking and the File System

Neither FS1 nor FS2 are re-entrant. If using preemptive multitasking, ensure that only one thread performs calls to the file system, or implement locking around each call.

## 11.2 Application Requirements

The application requirements for FS1 and FS2 are slightly different.

### 11.2.1 FS1 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"
```

## 11.2.2 FS2 Requirements

The file system library must be compiled with the application:

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

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

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

These specify certain static array sizes that allow control over the amount of root data space taken by FS2. If you are using only one flash device (and possibly battery-backed RAM), and are not using partitions, then there is no need to set **FS\_MAX\_DEVICES** or **FS\_MAX\_LX**. See section 11.4 for more information on partitioning.

### 11.2.2.1 Configuration Macros

#### **FS2\_USE\_PROGRAM\_FLASH**

Default: 0, do not use first (program) flash. To use the first flash, **#define** this macro to be the number of KB to allocate to FS2. The minimum of **FS2\_USE\_PROGRAM\_FLASH** and **XMEM\_RESERVE\_SIZE** will be used. The first flash can also be used in FS1. See section 11.6 for details.

#### **FS\_MAX\_DEVICES**

This macro defines the maximum physical media. If it is not defined in the program code, it will default to 1, 2, or 3, depending on the values of **FS2\_USE\_PROGRAM\_FLASH**, **XMEM\_RESERVE\_SIZE** and **FS2\_RAM\_RESERVE**.

**XMEM\_RESERVE\_SIZE** is defined in the BIOS and determines the amount of space reserved in the first flash that will not be used for xmem code space. This is defined to 0x0000.

**FS2\_RAM\_RESERVE** is also defined in the BIOS and determines the amount of space used for FS2 in RAM. If some battery-backed RAM is to be used by FS2, then this macro must be modified to specify the amount of RAM to reserve. This should be a multiple of 1024 bytes. The memory is reserved near the top of RAM. Note that this RAM will be reserved whether or not the application actually uses FS2.

#### **FS\_MAX\_LX**

This macro defines the maximum logical extents. You must increase this value by 1 for each new partition you application creates. If this is not defined in the program code it will default to **FS\_MAX\_DEVICES**.

#### **FS\_MAX\_FILES**

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

## 11.3 Functions

For backwards compatibility FS2 uses the same function names as FS1. Some functions have enhanced semantics when using FS2. For example `fwrite()` will allow writing over existing parts of the file rather than just appending.

### 11.3.1 FS1 API

These functions are the file system API for FS1. They are defined in `FILESYSTEM.LIB`. For a complete description of these functions see “Function Reference” on page 165.

Command	Description
<code>fs_init (FS1)</code>	Initialize the internal data structures for the file system.
<code>fs_format (FS1)</code>	Initialize the flash memory and the internal data structures.
<code>fs_reserve_blocks (FS1)</code>	Reserves blocks for privileged files.
<code>fsck (FS1)</code>	Verifies data integrity of files.
<code>fcreate (FS1)</code>	Creates a file and open it for writing.
<code>fcreate_unused (FS1)</code>	Creates a file with an unused file number.
<code>fopen_rd (FS1)</code>	Opens a file for reading.
<code>fopen_wr (FS1)</code>	Opens a file for writing (also opens it for reading.)
<code>fshift</code>	Removes specified number of bytes from file.
<code>fwrite (FS1)</code>	Writes to the end of a file.
<code>fread (FS1)</code>	Reads from the current file pointer.
<code>fseek (FS1)</code>	Moves the read pointer.
<code>ftell (FS1)</code>	Returns the current offset of the file pointer.
<code>fclose</code>	Closes a file.
<code>fdelete (FS1)</code>	Deletes a file.

*Table 10. FS1 API*

#### 11.3.1.1 FS1 API Details

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.

### 11.3.2 FS2 API

The API for FS2 is defined in **FS2.LIB**. For more info see “Function Reference” on page 165.

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

**Table 11. FS2 API**

### 11.3.2.1 FS2 API Details

The functions `fs_init` and `fs_format` are used in a slightly different manner than in FS1. `fs_init()` does not use its two parameters (`reserveblocks` and `numblocks`) since it computes appropriate values internally. `fs_format()` should only be called after `fs_init()`, if necessary. This function's first parameter, `reserveblocks`, must be 0; anything else returns an error. This is one of the few cases of incompatibility between FS1 and FS2. The third parameter, `wearlevel`, should be 1 for a new flash memory; otherwise it should be 0 to use the current wear leveling.

The `fsck()` function is not available and is not needed in FS2; `fs_init()` always completely checks for internal consistency.

Refer to `SAMPLES\FILESYSTEM\FS2DEMO1.C` for more details.

### 11.3.2.2 FS2 API Error Codes

When an API function returns an error, it may also return an error code in the global variable `errno`. The error codes are defined in the library file `ERRNO.LIB`.

## 11.4 Setting up and Partitioning the File System

FS2 can be more complex to initialize than FS1. This is because multiple device types can be used in the same application. For example, if the target board contains both battery-backed SRAM and a second flash chip, then both types of storage may be used for their respective advantages. The SRAM might be used for a small application configuration file that changes frequently, and the flash used for a large log file.

FS2 automatically detects the second flash device (if any) and will also use any SRAM set aside for the file system (if `FS2_RESERVE_RAM` is set).

### 11.4.1 Initial Formatting

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

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

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

- A special program that is loaded and run once in the factory, before the application is loaded. The special program prepares the filesystem and formats it. The application never formats; it expects the filesystem to be in a proper state.
- The application can perform some sort of consistency check. If it determines an inconsistency, it calls `format`. The consistency check could include testing for a file that should exist, or by checking some sort of "signature" that would be unlikely to occur by chance.
- Have the application prompt the end-user, if some form of interaction is possible.
- A combination of one or more of the above.

- Rely on a flash device being erased. This would be OK for a production run, but not suitable if battery-backed SRAM was being used for part of the filesystem.

## 11.4.2 Logical Extents

In FS2, the presence of both “devices” causes an initial default configuration of two logical extents to be set up. A logical extent (LX) is analogous to disk partitions used in other operating systems. It represents a contiguous area of the device set aside for file system operations. A LX contains sectors that are all the same size, and all contiguously addressable within the one device. Thus a flash device with three different sector sizes would necessitate at least three logical extents, and more if the same-sized sectors were not adjacent.

FS1 does not allow mixing of devices; it supports only one LX as defined in this document.

Files stored by the file system are comprised of two parts: one part contains the actual application data, and the other is a fixed size area used to contain data controlled by the file system in order to track the file status. This second area, called metadata, is analogous to a “directory entry” of other operating systems. The data and metadata for a file are usually stored in the same LX, however they may be separated for performance reasons. Since the metadata needs to be updated for each write operation, it is often advantageous to store the metadata in battery-backed SRAM with the bulk of the data on a flash device.

### 11.4.2.1 Specifying Logical Extents

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

When creating a file, the associated logical extents can be changed from the default by calling `fs_set_lx()`. Thereafter, all created files are associated with the specified LXs until a new call to `fs_set_lx()` is made. Typically, there will be a call to `fs_set_lx()` before each file is created, in order to ensure that the new file gets created with the desired associations.

FS2 allows the initial default logical extents to be divided further. This must be done before calling `fs_init()`. The function to create sub-partitions is called `fs_setup()`. This function takes an existing LX number, divides that LX according to the given parameters, and returns a newly created LX number. The original partition still exists, but is smaller because of the division. For example, in a system with LX#1 as a flash device of 256K and LX#2 as 4K of RAM, an initial call to `fs_setup()` might be made to partition LX#1 into two equal sized extents of 128K each. LX#1 would then be 128K (the first half of the flash) and LX#3 would be 128K (the other half). LX#2 is untouched.

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

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

Note that in particular, files written by FS1 are not readable by FS2 since the two file systems are incompatible at the device level.

### 11.4.3 Logical Sector Size

**fs\_setup()** can also be used to specify non-default logical sector (LS) sizes and other parameters. FS1 uses fixed logical sectors (i.e. “blocks”) of 4096 bytes. FS2 allows any LS size between 64 and 8192 bytes, providing the LS size is an exact power of 2. Each LX, including sub-partitions, can have a different LS size. This allows some performance optimization. Small LSs are better for a RAM LX, since it minimizes wasted space without incurring a performance penalty. Larger LSs are better for bulk data such as logs. If the flash physical sector size (i.e. the actual hardware sector size) is large, it is better to use a correspondingly large LS size. This is especially the case for byte-writable devices. Large LSs should also be used for large LXs. This minimizes the amount of time needed to initialize the file system and access large files. As a rule of thumb, there should be no more than 1024 LSs in any LX. The ideal LS size for RAM (which is the default) is 128 bytes. 256 or 512 can also be reasonable values for some applications that have a lot of spare RAM.

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

Sample program **SAMPLES\FILESYSTEM\FS2DEMO2** illustrates use of **fs\_setup()**. This sample also allows you to experiment with various file system settings to obtain the best performance.

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

## 11.5 File Identifiers

There are two ways to identify a particular file in the file system.

### 11.5.1 File Numbers

The file number uniquely identifies a file within a logical extent. File numbers must be unique within the entire file system. FS2 accepts file numbers in word format rather than the byte format of FS1:

```
typedef word FileNumber
```

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

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

### 11.5.2 File Names

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

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

Command	Description
<b>sspec_addfsfile</b>	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.
<b>sspec_readfile</b>	Read a file represented by the return value of <b>sspec_addfsfile</b> into a buffer.
<b>sspec_getlength</b>	Get the length (number of bytes) of the file.
<b>sspec_getfileloc</b>	Get the file system file number (1- 255). Cast return value to <b>FILENUMBER</b> .
<b>sspec_findname</b>	Find the index into the array of structures associated with named files of the file that has the specified name.
<b>sspec_getfiletype</b>	Get file type. For flash file system files this value will be <b>SSPEC_FSFILE</b> .

Table 12. Flash File System Auxiliary Functions

Command	Description
<b>sspec_findnextfile</b>	Find the next named file in the flash file system, at or following the specified index, and return the index of the file.
<b>sspec_remove</b>	Remove the file name association.
<b>sspec_save</b>	Saves to the flash file system the array of structures that reference the named files in the flash file system.
<b>sspec_restore</b>	Restores the array of structures that reference the named files in the flash file system.

*Table 12. Flash File System Auxiliary Functions*

## 11.6 Use of First Flash in FS1

To use FS1 in the first flash, a low-level driver must be used:

```
#define FS_FLASH_SINGLE
```

Because this particular low-level driver must share the first flash with the program code, the file system must be carefully placed such that the two do not collide. Also, it should be noted that any time the first flash is written to during runtime, interrupts will be shut off for the duration of the write. This could have serious implications for real-time systems.

To reserve space in the first flash, such that Dynamic C will not clobber the file system, a minor BIOS modification is necessary. The macro **XMEM\_RESERVE\_SIZE** in the BIOS is currently set to 0x0000. Increasing this value will reserve that much space between the end of xmem-code that Dynamic C is building, and the SystemID Block at the end of memory. Unfortunately, the file system needs to start on a **FS\_BLOCK\_SIZE** boundary, which is normally 4096 bytes. Therefore, slightly more space that is needed should be allocated, to allow for the SystemID Block and that the end of xmem space might not lie on a 4096 byte boundary.

After this space has been allocated, the location of the file system's beginning can be found. The end of where Dynamic C will touch the flash is stored in the macro **END\_OF\_XMEMORY**, and the file system may start at the next 4096 byte boundary after that point. The following code computes what to pass to **fs\_format()**.

```
long fs_start; // where to start the file system
fs_start = END_OF_XMEMORY; // start at the end of xmem
fs_start = fs_start / FS_BLOCK_SIZE; // divide out the blocksize, to
// meet requirements for fs_format
if((fs_start * FS_BLOCK_SIZE) != END_OF_XMEMORY) {
// rounding error - move up one block so end of xmem not clobbered
fs_start++;
}
fs_format(fs_start, NUM_BLOCKS, 0);
```

After this point, the file system should act normally.

If the 4096 byte block size is too large, given the limited room in the first flash, that can be overwritten with the macro:

```
#define FS_BLOCK_SIZE 512
```

See the sample program, **1stflash.c**, for an example of using the first flash with FS1.

## 11.7 Use of First Flash in FS2

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

1. Define **XMEM\_RESERVE\_SIZE** (currently set to 0x0000 in the BIOS) to the number of bytes to allocate in the first flash for the file system.
2. Define **FS2\_USE\_PROGRAM\_FLASH** to the number of KB (1024 bytes) to allocate in the first flash for the file system. Do this in the application code before **#use "fs2.lib"**.
3. Obtain the LX number of the first flash: Call **fs\_get\_other\_lx()** when there are two flash memories; call **fs\_get\_flash\_lx()** when there is only one.
4. If desired, create additional logical extents by calling the function **fs\_setup (FS2)** to further partition the device. This function can also change the logical sector sizes of an extent. See page 249 for more information.

### Example Code

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

```
FSLXnum flash1;
File f;

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

For a one flash board, **fs\_get\_flash\_lx()** must be called instead of **fs\_get\_other\_lx()** to obtain the logical extent number.

## 11.8 Skeleton Program Using FS1

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
#define 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.

## 11.9 Skeleton Program Using FS2

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

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

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

    fs_init(0, 0);
    if (!fcreate(&file, TESTFILE) &&
        fopen_wr(&file,TESTFILE)) {
        printf("error opening TESTFILE %d\n", errno);
        return -1;
    }
    fseek(&file, 0, SEEK_END);
    fwrite(&file,"hello",6);
    fwrite(&file,"12345",6);
    fwrite(&file,"67890",6);

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

For a more robust program, more error checking should be included. See the sample programs **SAMPLES\FILESYSTEM\FS2DEMO?.C** for more complex examples which include error checking, formatting, partitioning and other new features.

FS2 returns more information in the case of errors than FS1. The library **ERRNO.LIB** contains a list of all possible error codes returnable by the FS2 API. These error codes mostly conform to POSIX standards. If the return value of an FS2 API indicates an error, then the `errno` variable may be examined to determine a more specific reason for the failure. The possible `errno` codes returned from each function are documented with the function.



# 12. Using Assembly Language

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

## 12.1 Mixing Assembly and C

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

### 12.1.1 Embedded Assembly Syntax

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

```
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**).

### 12.1.2 Embedded C Syntax

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, therefore, there is no need to place the **nodebug** directive on each block.

## 12.2 The Assembler and the Preprocessor

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

- ? : conditional
- [ ] array index
- . dot
- > points to
- \* dereference

### 12.2.1 Comments

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

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

## 12.2.2 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.

## 12.2.3 Multiline Macros

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

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

## 12.2.4 Labels

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

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

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

## 12.2.5 Special Symbols

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

*Table 13. Special Assembly-Language Symbols*

Symbol	Description
<code>@SP</code>	Indicates the amount of stack space (in bytes) used for stack-based variables. This does not include arguments.
<code>@RETVAl</code>	Evaluates the offset from the <i>frame reference point</i> to the stack space reserved for the <code>struct</code> function returns. See Section 12.4.1.1 on page 119 for more information.
<code>@LENGTH</code>	Determines the next reference address of a variable plus its size.

## 12.2.6 C Variables

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

The 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
    }
};
```

## 12.3 Stand-alone Assembly Code

A stand-alone assembly function is one that is defined outside the context of a C language function. Dynamic C always places a stand-alone assembly function in root memory.

A stand-alone assembly function has no **auto** variables and no formal parameters. It can, however, have arguments passed to it by the calling function. When a program calls a function from C, it puts the first argument into a *primary register*. If the first argument has one or two bytes (**int**, **unsigned int**, **char**, **pointer**), the primary register is HL (with register H containing the most significant byte). If the first argument has four bytes (**long**, **unsigned long**, **float**), the primary register is BC:DE (with register B containing the most significant byte). Assembly-language code can use the first argument very efficiently. *Only* the first argument is put into the primary register, while *all* arguments—including the first, pushed last—are pushed on the stack.

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

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

### 12.3.1 Example of Stand-Alone Assembly Code

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

```
int foo ( int );    // A function prototype can be declared for
                   // stand-alone assembly functions, which
                   // will cause the compiler to perform the
                   // appropriate type-checking.

main(){
    int i,j;
    i=1;
    j=foo(i);
}
#asm
foo::
    ...
    ld hl,2        // The return value expected by main()
ret                // is put in HL just before foo() returns
#endasm
```

The entire program can be written in assembly.

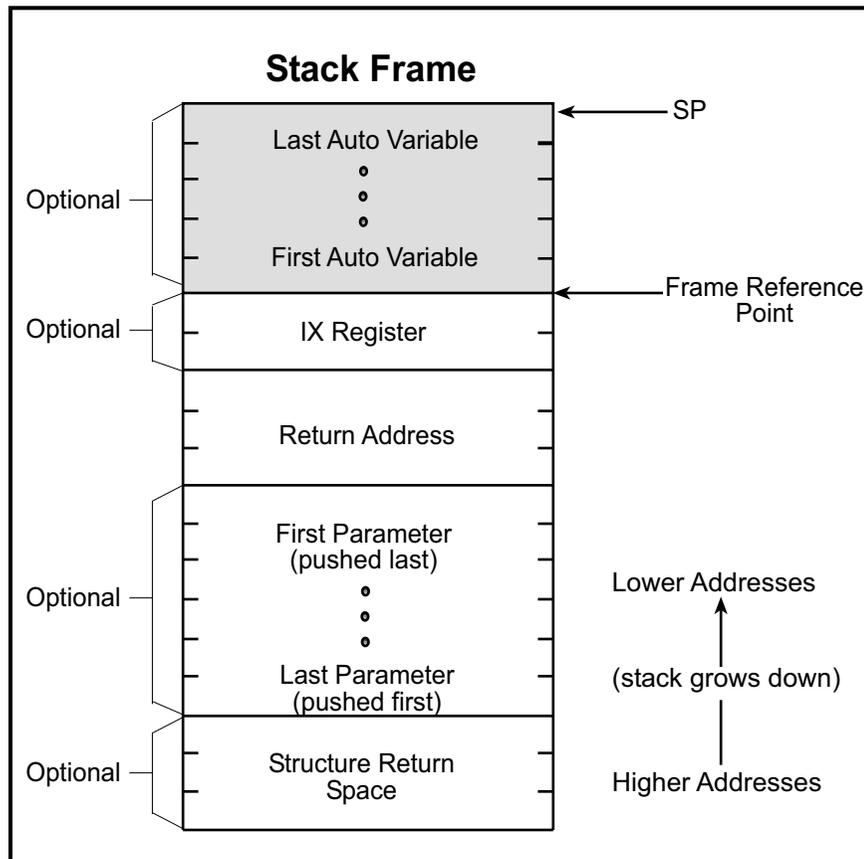
```
#asm
main::
    ...
ret
#endasm
```

## 12.4 Embedded Assembly Code

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

### 12.4.1 The Stack Frame

The purpose and structure of a *stack frame* should be understood before writing embedded assembly code. A stack frame is a run-time structure on the stack that provides the storage for all **auto** variables, function arguments and the return address for a particular function. If the IX register is used for a frame reference pointer, the previous value of IX is also kept in the stack frame. 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 depends on the function definition. The presence of arguments and structure return space depends on the function call. (The stack pointer may actually point lower than the indicated mark temporarily because of temporary information pushed on the stack.)

The shaded area in the stack frame is the stack storage allocated for **auto** variables. The assembler symbol **@SP** represents the size of this area.

### 12.4.1.1 The Frame Reference Point

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

## 12.4.2 Example of Embedded Assembly Code

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

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

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

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

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

    x = 0x8888;
    z = 0x9999;
#asm
    // @SP+i gives the offset of i from the stack frame on entry.
    // On the Z180, this is how HL is loaded with the value in i.
    // (The assembler combines i and @SP into one constant.)
    ld  hl,@SP+i
    add hl,sp
    ld  hl,(hl)

    // On the Rabbit, this code does the same:
    ld  hl,(sp+@SP+i)

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

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

### 12.4.2.1 The Disassembled Code Window

A program may be debugged at the assembly level by clicking the **Assemb** radio button on Dynamic C's toolbar to open the Disassembled Code window. Single-stepping and breakpoints are supported in this window. When the Disassembled Code window is open, single-stepping occurs instruction by instruction rather than statement by statement. The figure below shows the Registers, Stack and Disassembled Code windows for the example code, **asm1.c**, just before the function call.

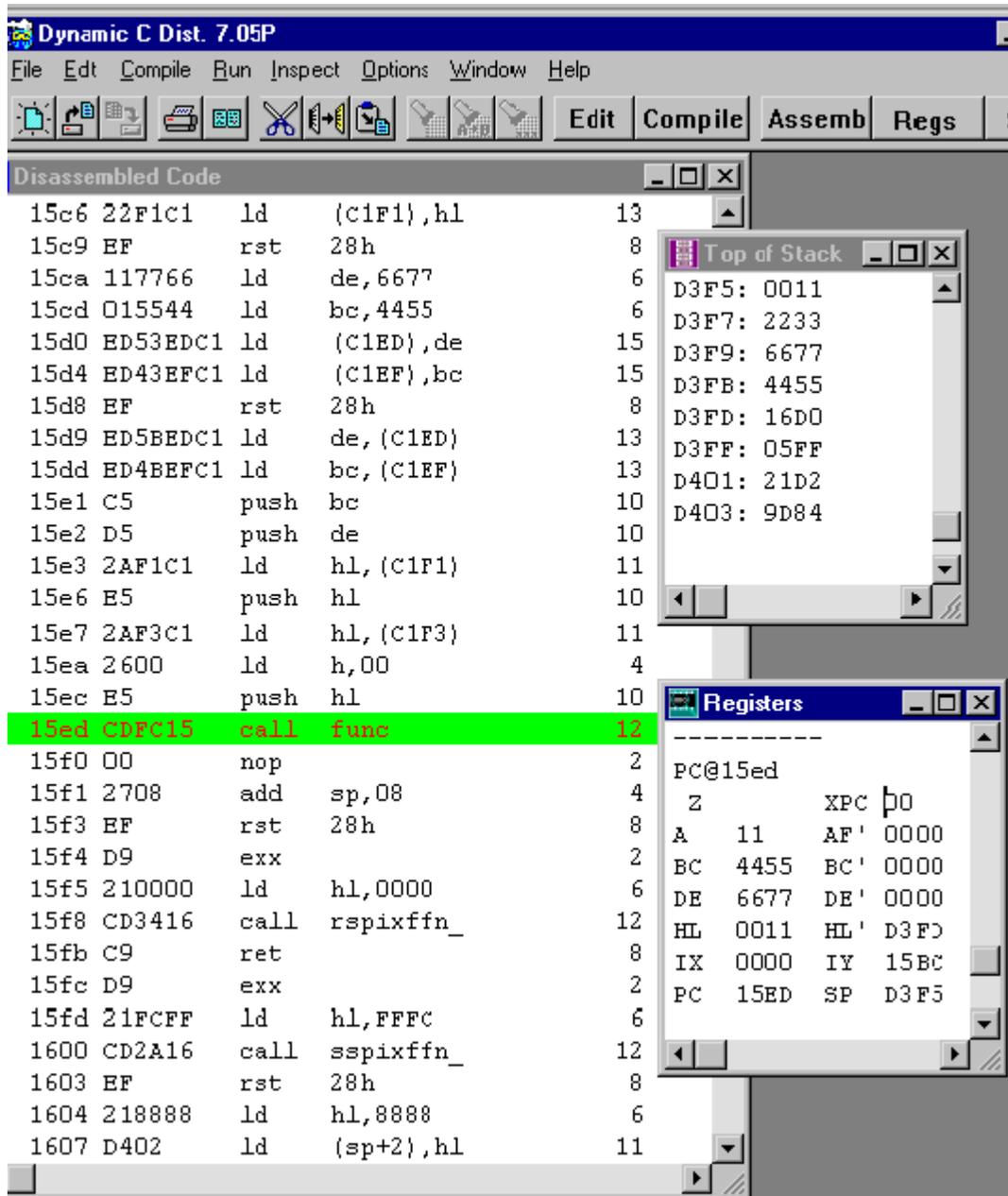


Figure 5. Registers, Stack and Disassembled Code Windows

### 12.4.2.2 Instruction Cycle Time

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

### 12.4.3 Local Variable Access

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

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

#### 12.4.3.1 Using the IX Register

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

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

```
ld a,(ix+ch) ; a ← ch
```

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

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

This takes a total of 24 cycles.

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

### 12.4.3.2 Functions in Extended Memory

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

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

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

## 12.5 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.

### 12.5.1 Passing Parameters

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

### 12.5.2 Location of Return Results

If a C-callable assembly function is expected to return a result (of primitive type), the function must pass the result in the “primary register.” If the result is an `int`, `unsigned int`, `char`, or a pointer, return the result in `HL` (register `H` contains the most significant byte). If the result is a `long`, `unsigned long`, or `float`, return the result in `BCDE` (register `B` contains the most significant byte). A C function containing embedded assembly code may, of course, use a C `return` statement to return a value. A stand-alone assembly routine, however, must load the primary register with the return value before the `ret` instruction.

### 12.5.2.1 Returning a Structure

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

Inline assembly code may access the stack area reserved for structure return values by the symbol `@RETVAl`, which is an offset from the frame reference point.

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

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

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

## 12.6 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.

## 12.7 Interrupt Routines in Assembly

Dynamic C allows Interrupt Service Routines (ISRs) 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 ISRs.

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 ISR is that there is no need to worry about saving and restoring registers or reenabling interrupts. The drawback is that the C interrupt function does save all registers, which takes some amount of time. A stand-alone assembly routine needs to save and restore only the registers it uses.

Interrupts are turned off by the CPU before the ISR is called. Generally, the ISR performs the following actions:

1. 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.
2. Determine the cause of the interrupt. Some devices map multiple causes to the same interrupt vector. An interrupt handler must determine what actually caused the interrupt.
3. Remove the cause of the interrupt.
4. If an interrupt has more than one possible cause, check for all the causes and remove all the causes at the same time.
5. When finished, restore registers saved on the stack. Naturally, this code must match the code that saved the registers. Interrupt routines written in C perform this automatically. Stand-alone assembly routines must pop the registers explicitly.
6. 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.
7. Return. There are three types of interrupt returns: **ret**, **reti**, and **retn**.

## 12.8 Common Problems

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

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

The following code illustrates the concept.

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

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

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



# 13. Keywords

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

## abandon

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

## abort

Jumps out of a costatement.

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

## 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 function'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 **firsttime** 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. This is the default case.

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

## NULL

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

## protected

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

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. Starting with Dynamic C 7.05, `sizeof()` can be used inside of assembly blocks.

```
int list[] = { 10, 99, 33, 2, -7, 63, 217 };
    ...
x = sizeof(list);           // x will be assigned 14
```

## speed

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

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

## static

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

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

The 20-bit physical address of the block is assigned to **name** by the compiler as an unsigned long variable. The amount of memory allocated depends on the data type. Each **char** is allocated one byte, and each **int** is allocated two bytes. If an integer fits into one byte, it is still allocated two bytes. Each **float** and **long** cause four bytes to be allocated.

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

```
xdata name1 { '\x46', '\x47', '\x48', '\x49', '\x4A', '\x20', '\x20' };  
xdata name2 { 'R', 'a', 'b', 'b', 'i', 't' };  
xdata name3 { " Rules! " };  
xdata name4 { 1.0, 2.0, (float)3, 40e-01, 5e00, .6e1 };
```

The data can be viewed directly in the dump window by doing a PHYSICAL memory dump using the 20-bit address of the xdata block. See `Samples\Xmem\xdata.c` for more information.

## xmem

Indicates that a function is to be placed in extended memory. This keyword is semantically meaningful in function prototypes. Its use must be consistent between the prototype and the function definition. This keyword is not valid with stand-alone assembly routines: they are always placed in root memory.

```
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 available options are:

**nobdebug** disables debug code during assembly.

**debug** enables 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  
#nobdebug
```

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* }**

**#GLOBAL\_INIT** sections are blocks of code that are run once before **main()** is called. They should appear in functions after variable declarations and before the first executable code.

If a local static variable must be initialized once only before the program runs, it should be done in a **#GLOBAL\_INIT** section, but other initialization may also be done. For example:

```
// This function outputs and returns the number of times
// it has been called.
int foo(){
    char count;

    #GLOBAL_INIT{
        // initialize count
        count = 1;

        // make port A output
        WrPortI(SPCR,SPCRShadow,0x84);
    }

    // output count
    WrPortI(PADR,NULL,count);

    // increment and return count
    return ++count;
}
```

## **#error "..."**

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

**root**: all functions not declared as **xmem** go to root memory.

**xmem**: all 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. **#nouseix** is the default.

## **#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.



# 14. Operators

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

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

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

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

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

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

For example, given the expression

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

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

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

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

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

uses parentheses to circumvent the normal order of evaluation.

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

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

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

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

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

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

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

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

The operators are summarized in the following pages.

## 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 = 0xFFFF0;  
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 0xFFFF
```

~

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

```
int switches;
switches = 0xFFFF;
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 ≤ 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  $\geq$  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  $\neq$  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 = 0xFFFF;
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 14. Operator Precedence**

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



# 15. Function Reference

## 15.1 Functional Groups

<b>Arithmetic</b>	<code>abs</code> <code>getcrc</code>
<b>Bit Manipulation</b>	<code>bit</code> <code>BIT</code> <code>res</code> <code>RES</code> <code>set</code> <code>SET</code>
<b>Character</b>	<code>isalnum</code> <code>isalpha</code> <code>iscntrl</code> <code>isdigit</code> <code>isgraph</code> <code>islower</code> <code>isprint</code> <code>ispunct</code> <code>isspace</code> <code>isupper</code> <code>isxdigit</code>
<b>Data Encryption</b>	<code>AESencrypt</code> <code>AESdecrypt</code> <code>AESencryptStream</code> <code>AESdecryptStream</code> <code>AEsexpandKey</code> <code>AESinitStream</code>
<b>Error Handling</b>	<code>errlogFormatEntry</code> <code>errlogFormatRegDump</code> <code>errlogFormatStackDump</code> <code>errlogGetHeaderInfo</code> <code>errlogGetMessage</code> <code>errlogGetNthEntry</code> <code>errlogReadHeader</code> <code>exception</code> <code>ResetErrorLog</code>
<b>Extended Memory</b>	<code>paddr</code> <code>root2xmem</code> <code>WriteFlash2</code> <code>xalloc</code> <code>xmem2root</code> <code>xmem2xmem</code>

<b>Fast Fourier Transforms</b>	fftcplx fftcplxinv fftrealm fftrealminv hanncplx hannreal powerspectrum
<b>File System</b>	fclose fcreate (FS1) fcreate (FS2) fcreate_unused (FS1) fcreate_unused (FS2) fdelete (FS1) fdelete (FS2) fflush (FS2) fopen_rd (FS1) fopen_rd (FS2) fopen_wr (FS1) fopen_wr (FS2) fread (FS1) fread (FS2) fsck (FS1) fseek (FS1) fseek (FS2) fs_format (FS1) fs_format (FS2) fshift fs_init (FS1) fs_init (FS2) fs_get_flash_lx (FS2) fs_get_lx (FS2) fs_get_lx_size (FS2) fs_get_other_lx (FS2) fs_get_ram_lx (FS2) fs_reserve_blocks (FS1) fs_set_lx (FS2) fs_setup (FS2) fs_sync (FS2) ftell (FS1) ftell (FS2) fwrite (FS1) fwrite (FS2)

<b>Floating-point Math</b>	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
<b>Low-level Flash Access</b>	flash_erasechip flash_erasesector flash_gettype flash_init flash_read flash_readsector flash_sector2xwindow flash_writesector
<b>GPS</b>	gps_get_position gps_get_utc gps_ground_distance
<b>I<sup>2</sup>C Bus</b>	i2c_check_ack i2c_init i2c_read_char i2c_send_ack i2c_send_nak i2c_start_tx i2c_startw_tx i2c_stop_tx i2c_write_char

<b>I/O</b>	BitRdPortE BitRdPortI BitWrPortE BitWrPortI RdPortE RdPortI WrPortE WrPortI
<b>Interrupts</b>	GetVectExtern2000 GetVectIntern SetVectExtern2000 SetVectIntern

<b>MicroC/OS-II</b>	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
<b>Miscellaneous</b>	long jmp qsort runwatch set jmp

<b>Multitasking</b>	CoBegin CoPause CoReset CoResume DelayMs DelaySec DelayTicks IntervalMs IntervalSec IntervalTick isCoDone isCoRunning loophead loopinit
<b>Number-to-string Conversion</b>	ftoa htoa itoa ltoa ltoan utoa
<b>Real-time Clock</b>	mktime mktm read_rtc read_rtc_32kHz tm_rd tm_wr write_rtc
<b>Serial Communication</b> (Interrupt-driven, Stream-based, 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

<b>Serial Communication</b> (Frame-Oriented Functions)	code <pre> cof_pktXreceive cof_pktXsend pktXclose pktXgetErrors pktXinitBuffers pktXopen pktXreceive pktXsend pktXsending pktXsetParity </pre>
<b>SPI</b>	code <pre> SPIinit SPIRead SPIWrite </pre>
<b>STDIO</b>	code <pre> getchar gets kbhit outchrs outstr printf putchar puts sprintf </pre>
<b>String Manipulation</b>	code <pre> memchr memcmp memcpy memmove memset strcat strchr strcmp strcmpi strcpy strcspn strlen strncat strncmp strncmpi strncpy strpbrk strrchr strspn strstr strtok tolower toupper </pre>
<b>String-to-number Conversion</b>	code <pre> atof atoi atol strtod strtol </pre>

<b>System</b>	chkHardReset chkSoftReset chkWDTO clockDoublerOff clockDoublerOn defineErrorHandler exit forceSoftReset ipres ipset premain _sysIsSoftReset sysResetChain updateTimers use32HzOsc useClockDivider useMainOsc
<b>User Block</b>	readUserBlock writeUserBlock
<b>Watchdog</b>	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

## AESdecrypt

```
void AESdecrypt(char *data, char *expandedkey, int nb, int nk );
```

### DESCRIPTION

Decrypts a block of data using an implementation of the Rijndael AES cipher. The encrypted block of data is overwritten by the decrypted block of data.

### PARAMETERS

<b>data</b>	A block of data to be decrypted.
<b>expandedkey</b>	A set of round keys (generated by <b>AESexpandKey()</b> ).
<b>nb</b>	The block size to use. Block is 4 * <b>nb</b> bytes long.
<b>nk</b>	The key size to use. Cipher key is 4 * <b>nk</b> bytes long.

### LIBRARY

AES\_CRYPT.LIB

## AESdecryptStream

```
void AESdecryptStream(AESstreamState *state, char *data, int count);
```

### DESCRIPTION

Decrypts an array of bytes using the Rabbit implementation of cipher feedback mode. See **Samples\Crypt\AES\_STREAMTEST.C** for a sample program and a detailed explanation of the encryption/decryption process.

### PARAMETERS

<b>state</b>	The <b>AESstreamState</b> structure. This memory must be allocated in the program code before calling <b>AESdecryptStream()</b> : <pre>static AESstreamState decrypt_state;</pre>
<b>data</b>	An array of bytes that will be decrypted in place.
<b>count</b>	Size of data array

### LIBRARY

AES\_CRYPT.LIB

## AESencrypt

```
void AESencrypt(char *data, char *expandedkey, int nb, int nk );
```

### DESCRIPTION

Encrypts a block of data using an implementation of the Rijndael AES cipher. The block of data is overwritten by the encrypted block of data.

### PARAMETERS

<b>data</b>	A block of data to be encrypted
<b>expandedkey</b>	A set of round keys (generated by <b>AESexpandKey()</b> )
<b>nb</b>	The block size to use. Block is 4 * <b>nb</b> bytes long
<b>nk</b>	The key size to use. Cipher key is 4 * <b>nk</b> bytes long

### RETURN VALUE

None

### LIBRARY

AES\_CRYPT.LIB

## AESencryptStream

```
void AESencryptStream(AESstreamState *state, char *data, int  
count);
```

### DESCRIPTION

Encrypts an array of bytes using the Rabbit implementation of cipher feedback mode. See **Samples\Crypt\AES\_STREAMTEST.C** for a sample program and a detailed explanation of the encryption/decryption process.

### PARAMETERS

<b>state</b>	The <b>AESstreamState</b> structure. This memory must be allocated in the program code before calling <b>AESencryptStream()</b> : <pre>static AESstreamState encrypt_state;</pre>
<b>data</b>	An array of bytes that will be encrypted in place.
<b>count</b>	Size of data array.

### LIBRARY

AES\_CRYPT.LIB

## AESEXPANDKEY

```
void AESEXPANDKEY(char *expanded, char *key, int nb, int nk, int
    rounds);
```

### DESCRIPTION

Prepares a key for use by expanding it into a set of round keys. A key is a “password” to decipher encoded data.

### PARAMETERS

<b>expanded</b>	A buffer for storing the expanded key. The size of the expanded key is $4 * \mathbf{nb} * (\mathbf{rounds} + 1)$ .
<b>key</b>	The cipher key, the size should be $4 * \mathbf{nk}$
<b>nb</b>	The block size will be $4 * \mathbf{nb}$ bytes long.
<b>nk</b>	The key size will be $4 * \mathbf{nk}$ bytes long.
<b>rounds</b>	The number of cipher rounds to use.

### RETURN VALUE

None

### LIBRARY

AES\_CRYPT.LIB

## AESinitStream

```
void AESinitStream(AESstreamState *state, char *key, char
    *init_vector);
```

### DESCRIPTION

Sets up **AESstreamState** to begin encrypting or decrypting a stream. Each **AESstreamState** structure can only be used for one direction. See **Samples\Crypt\AES\_STREAMTEST.C** for a sample program and a detailed explanation of the encryption/decryption process.

### PARAMETERS

<b>state</b>	An <b>AESstreamState</b> structure to be initialized. This memory must be allocated in the program code before calling <b>AESinitStream()</b> .
<b>key</b>	The 16-byte cipher key, using a <b>NULL</b> pointer will prevent an existing key from being recalculated.
<b>init_vector</b>	A 16-byte array representing the initial state of the feedback registers. Both ends of the stream must begin with the same initialization vector.

### RETURN VALUE

None

### LIBRARY

**AES\_CRYPT.LIB**

## 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

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

Returned Value	Parameter Values
<i>angle</i>	$x \neq 0, y \neq 0$
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, `_xtoxEr` is set to **1**. Otherwise `_xtoxEr` 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

## atoi

```
long atoi(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

<code>port</code>	Address of internal parallel port data register.
<code>bitnumber</code>	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 **bitcode** with **value** (0 or 1) and copies shadow to register.

WARNING! A shadow register is required for this function.

### PARAMETERS

<b>port</b>	Address of external parallel port data register.
<b>portshadow</b>	Reference pointer to a variable to shadow the current value of the register.
<b>value</b>	Value of 0 or 1 to be written to the bit position.
<b>bitcode</b>	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

<b>port</b>	Address of external parallel port data register.
<b>portshadow</b>	Reference pointer to a variable to shadow the current value of the register.
<b>value</b>	Value of 0 or 1 to be written to the bit position.
<b>bitcode</b>	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\_pktXreceive

```
int cof_pktXreceive(void *buffer, int buffer_size); X=A|B|C|D
```

### DESCRIPTION

Receives an incoming packet. This function will return after a complete packet has been read into the buffer.

### PARAMETERS

**buffer**                A buffer for the packet to be written into.

**buffer\_size**        Length of the buffer.

### RETURN VALUE

The number of bytes in the received packet: Success.  
0: No new packets have been received.  
-1: The packet is too large for the given buffer.  
-2: A needed **test\_packet** function is not defined

### LIBRARY

PACKET.LIB

## cof\_pktXsend

```
void cof_pktXsend(void *send_buffer int buffer_length, char
    delay); X=A|B|C|D
```

### DESCRIPTION

Initiates the sending of a packet of data. The function will exit when the packet is finished transmitting.

### PARAMETERS

<b>send_buffer</b>	The data to be sent.
<b>buffer_length</b>	Length of the data buffer to transmit.
<b>delay</b>	The number of byte times (0-255) to delay before sending data. This is used to implement protocol-specific delays between packets.

### LIBRARY

PACKET.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

<b>s</b>	Character array into which a <b>NULL</b> terminated string is read.
<b>max</b>	The maximum number of characters to read into s.
<b>tmout</b>	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** milliseconds 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

<b>data</b>	Data structure into which characters are read.
<b>length</b>	The number of characters to read into <b>data</b> .
<b>tmout</b>	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

<b>data</b>	Data structure to write.
<b>length</b>	Number of bytes in <b>data</b> 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**. This user-defined function must be in root memory. Specify **root** at the start of the function definition to ensure this. When a runtime error occurs, the following information is passed to the error handler on the stack:

Stack Position	Stack Contents
SP+0	return address for <b>exceptionRet</b>
SP+2	Error code
SP+4	0x0000 (can be used for additional information)
SP+6	XPC when <b>exception()</b> was called (upper byte)
SP+8	address where <b>exception()</b> was called

### PARAMETERS

**errfcn**            Pointer to user-defined run-time 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 delays);
```

### 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

**delays** 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

## errlogGetHeaderInfo

```
root char* errlogGetHeaderInfo();
```

### DESCRIPTION

Reads the error log header and formats the output.

When running stand alone (not talking to Dynamic C), this function reads the header directly from the log buffer. When in debug mode, this function reads the header from the copy in flash.

When a Dynamic C cold boot takes place, the header in RAM is zeroed out to initialize it, but first its contents are copied to an address in the BIOS code before the BIOS in RAM is copied to flash. This means that on the second cold boot, the data structure in flash will be zeroed out. The configuration of the log buffer may still be read, and the log buffer entries are not affected.

Because the exception mechanism resets the processor by causing a watchdog time-out, the number of watchdog time-outs reported by this functions is the number of actual WD-TOs plus the number of exceptions.

### RETURN VALUE

A **NULL**-terminated string containing the header information:

```
Status Byte: 0
#Exceptions: 5
Index last exception: 5
#SW Resets: 2
#HW Resets: 2
#WD Timeouts: 5
```

The string will contain "Header checksum invalid" if a checksum error occurs. The meaning of the status byte is as follows:

```
bit 0 - An error has occurred since deployment
bit 1 - The count of SW resets has rolled over.
bit 2 - The count of HW resets has rolled over.
bit 3 - The count of WDTOs has rolled over.
bit 4 - The count of exceptions has rolled over.
bit 5-7 - Not used
```

The index of the last exception is the index from the start of the error log entries. If this index does not equal the total exception count minus one, the error log entries have wrapped around the log buffer.

### LIBRARY

```
ERRORS.LIB
```

## errlogGetNthEntry

```
root int errlogGetNthEntry(int N);
```

### DESCRIPTION

Loads **errLogEntry** structure with Nth entry of the error buffer. This must be called before the functions below that format the output.

### PARAMETERS

**N**                      Index of entry to load into errLogEntry

### RETURN VALUE

0: Success, entry checksum OK  
-1: Failure, entry checksum not OK

### LIBRARY

ERRORS.LIB

## errlogFormatEntry

```
root char* errlogFormatEntry();
```

### DESCRIPTION

Returns a **NULL**-terminated string containing the basic information contained in **errLogEntry**:

```
Error type=240  
Address = 00:16aa  
Time: 06/11/2001 20:49:29
```

### RETURN VALUE

The **NULL**-terminated string described above.

### LIBRARY

ERRORS.LIB

## errlogFormatRegDump

```
root char* errlogFormatRegDump();
```

### DESCRIPTION

Returns a **NULL**-terminated string containing a register dump using the data in `errLogEntry`:

```
AF=0000,AF'=0000
HL=00f0,HL'=15e3
BC=16ce,BC'=1600
DE=0000,DE'=1731
IX=d3f1,IY =0560
SP=d3eb,XPC=0000
```

### RETURN VALUE

The **NULL**-terminated string described above.

### LIBRARY

ERRORS.LIB

## errlogFormatStackDump

```
root char* errlogFormatStackDump();
```

### DESCRIPTION

Returns a **NULL**-terminated string containing a stack dump using the data in `errLogEntry`.

```
Stack Dump:
0024,04f1,
d378,c146,
c400,a108,
2404,0000,
```

### RETURN VALUE

The **NULL**-terminated string describe above.

### LIBRARY

ERRORS.LIB

## errlogGetMessage

```
root char* errlogGetMessage();
```

### DESCRIPTION

Returns a **NULL**-terminated string containing the 8 byte message in **errLogEntry**.

### RETURN VALUE

A **NULL**-terminated string

### LIBRARY

ERRORS.LIB

## errlogReadHeader

```
root int errlogReadHeader();
```

### DESCRIPTION

Reads error log header into the structure **errlogInfo**.

### RETURN VALUE

0: Success, entry checksum OK  
-1: Failure, entry checksum not OK

### LIBRARY

ERRORS.LIB

## exception

```
int exception(int errCode);
```

### DESCRIPTION

This function is called by Rabbit libraries when a runtime error occurs. It puts information relevant to the runtime error on the stack and calls the default runtime error handler pointed to by the **ERROR\_EXIT** macro. To define your own error handler, see the **defineErrorHandler()** function.

When the error handler is called, the following information will be on the stack:

SP+0 return address for error handler call

SP+2 runtime 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 from

### RETURN VALUE

Runtime error code passed to it.

### LIBRARY

ERRORS.LIB

### SEE ALSO

defineErrorHandler

## 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 (FS1)**

```
int fcreate(File* f, FileNumber fnum);
```

### **DESCRIPTION**

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 (FS2)**

```
int fcreate(File* f, FileNumber name);
```

### **DESCRIPTION**

Create a new file with the given "file name" which is composed of two parts: the low byte is the actual file number (1 to 255 inclusive), and the high byte contains an extent number (1 to `_fs.num_lx`) on which to place the file metadata. The extent specified by `fs_set_lx()` is always used to determine the actual data extent. If the high byte contains 0, then the default metadata extent specified by `fs_set_lx()` is used. The file descriptor is filled in if successful. The file will be opened for writing, so a further call to `fopen_wr()` is not necessary.

The number of files which may be created is limited by the lower of `FS_MAX_FILES` and 255. This limit applies to the entire filesystem (all logical extents).

Once a file is created, its data and metadata extent numbers are fixed for the life of the file, i.e. until it is deleted.

When created, no space is allocated in the file system until the first write occurs for the file. Thus, if the system power is cycled after creation but before the first byte is written, the file will be effectively deleted.

Before calling this function, a variable of type `File` must be defined in the application program.

```
File file;  
fcreate (&file, 1);
```

### **PARAMETERS**

<code>f</code>	Pointer to the file descriptor to fill in.
<code>name</code>	File number including optional metadata extent number.

### **RETURN VALUE**

0: Success  
!0: Failure

### **ERRNO VALUES**

**EINVAL** - Zero file number requested, or invalid extent number.

**EEXIST** - File with given number already exists.

**ENFILE** - No space is available in the "existing fileable". If this error occurs, increase the definition of `FS_MAX_FILES`, which is a `#define` constant which should be declared before `#use "fs2.lib"`.

### **LIBRARY**

`fs2.LIB`

### **SEE ALSO**

`fcreate_unused (FS2)`, `fs_set_lx (FS2)`, `fdelete (FS2)`

## `fcreate_unused (FS1)`

```
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 (FS1)`

## **fcreate\_unused (FS2)**

```
FileNumber fcreate_unused(File* f);
```

### **DESCRIPTION**

Create a new file and return the "file name" which is a number between 1 and 255. The new file will be created on the current default extent(s) as specified by **fs\_set\_lx()**. Other behavior is the same as **fcreate()**.

### **PARAMETERS**

**f**                      Pointer to file descriptor to fill in.

### **RETURN VALUE**

>0: Success, the **FileNumber** (1-255) of the new file.  
0: Failure.

### **ERRNO VALUE**

**ENFILE** - No unused file number available.

### **LIBRARY**

fs2.LIB

### **SEE ALSO**

fcreate (FS2), fs\_set\_lx (FS2), fdelete (FS2)

## **fdelete (FS1)**

```
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

## **fdelete (FS2)**

```
int fdelete(FileNumber name);
```

### **DESCRIPTION**

Delete the file with the given number. The specified file must not be open. The file number (i.e. **name**) is composed of two parts: the low byte contains the actual file number, and the high byte (if not zero) contains the metadata extent number of the file.

### **PARAMETERS**

**name**                   File number (1 to 255 inclusive).

### **RETURN VALUE**

0: Success

!0: Failure

### **LIBRARY**

fs2.LIB

### **ERRNO VALUES**

**ENOENT** - File does not exist, or metadata extent number does not match an existing file.

**EBUSY** - File is open.

**EIO** - I/O error when releasing blocks occupied by this file.

### **SEE ALSO**

fcreate (FS2)

## **fflush (FS2)**

```
int fflush(File * f);
```

### **DESCRIPTION**

Flush any buffers, associated with the given file, retained in RAM to the underlying hardware device. This ensures that the file is completely written to the filesystem. The file system does not currently perform any buffering, however future revisions of this library may introduce buffering to improve performance.

### **PARAMETERS**

**f**                      Pointer to open file descriptor.

### **RETURN VALUE**

0: Success  
!0: Failure

### **ERRNO VALUES**

**EBADFD** - file invalid or not open.  
**EIO** - I/O error.

### **LIBRARY**

fs2.LIB

### **SEE ALSO**

fs\_sync (FS2)

## 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

<b>x</b>	Pointer to <b>N</b> -element array of complex fractions.
<b>N</b>	Number of complex elements in array <b>x</b> .
<b>blockexp</b>	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  $\mathbf{x}$  and returns the complex result in  $\mathbf{x}$ .  $N$  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  $\mathbf{x}$  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

<b>x</b>	Pointer to $N$ -element array of complex fractions.
<b>N</b>	Number of complex elements in array $\mathbf{x}$ .
<b>blockexp</b>	Pointer to integer block exponent.

### LIBRARY

FFT.LIB

### SEE ALSO

fftcplx, fftreal, fftrealinv, hanncplx, hannreal, powerspectrum

## fftrealm

```
void fftreal(int *x, int N, int *blockexp)
```

### DESCRIPTION

Computes the  $N$ -point, positive-frequency complex spectrum of the  $2N$ -point real sequence in array  $\mathbf{x}$ . The  $2N$ -point real sequence in array  $\mathbf{x}$  is replaced with its  $N$ -point positive-frequency complex spectrum. The value of `blockexp` is increased by 1 each time array  $\mathbf{x}$  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  $f_{max}$  term.

The  $2N$ -point real sequence is stored in natural order. The zeroth element of the sequence is stored in  $\mathbf{x}[0]$ , the first element in  $\mathbf{x}[1]$ , 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

<code>x</code>	Pointer to $2N$ -point sequence of real fractions.
<code>N</code>	Number of complex elements in output spectrum
<code>blockexp</code>	Pointer to integer block exponent.

### LIBRARY

FFT.LIB

### SEE ALSO

`fftcplx`, `fftcplxinv`, `fftrealm`, `hanncplx`, `hannreal`, `powerspectrum`

## fftrealignv

```
void fftrealinv(int *x, int N, int *blockexp)
```

### DESCRIPTION

Computes the  $2N$ -point real sequence corresponding to the  $N$ -point, positive-frequency complex spectrum in array  $\mathbf{x}$ . The  $N$ -point, positive-frequency spectrum contained in array  $\mathbf{x}$  is replaced with its corresponding  $2N$ -point real sequence. The value of `blockexp` is increased by 1 each time array  $\mathbf{x}$  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  $\mathbf{x}[0]$  term (stored  $\mathbf{x}[1]$ ).

The  $2N$ -point real sequence is stored in natural order. The zeroth element of the sequence is stored in  $\mathbf{x}[0]$ , the first element in  $\mathbf{x}[1]$ , and the  $k$ th element in  $\mathbf{x}[k]$ .

$N$  must be a power of 2 and lie between 4 and 1024. An invalid  $N$  causes a RANGE exception.

### PARAMETERS

$\mathbf{x}$	Pointer to $N$ -element array of complex fractions.
$N$	Number of complex elements in array $\mathbf{x}$ .
<code>blockexp</code>	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\_erasector, flash\_gettype, flash\_init, flash\_read,  
flash\_readsector, flash\_sector2xwindow, flash\_writesector

## flash\_erasector

```
int flash_erasector(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\_erasector, 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

<b>fd</b>	This is a pointer to an internal data structure that holds information about a Flash Memory chip.
<b>mb3cr</b>	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: Success
- 1: Invalid Flash Memory type
- 1: Attempt made to initialize primary Flash Memory

### LIBRARY

FLASH.LIB

### SEE ALSO

flash\_erasechip, flash\_erasector, 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

<b>fd</b>	The <b>FlashDescriptor</b> of the Flash Memory to read from.
<b>sector</b>	The sector of the Flash Memory to read from.
<b>offset</b>	The displacement, in bytes, from the beginning of the sector to start reading at.
<b>buffer</b>	The physical address of the destination buffer. TIP: A logical address can be changed to a physical with the function <b>paddr</b> .
<b>length</b>	The number of bytes to read.

### RETURN VALUE

0: Success

### LIBRARY

FLASH.LIB

### SEE ALSO

`flash_erasechip`, `flash_erasector`, `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

<b>fd</b>	The FlashDescriptor of the Flash Memory to read from.
<b>sector</b>	The source sector to read.
<b>buffer</b>	The physical address of the destination buffer. TIP: A logical address can be changed to a physical with the function <b>paddr</b> .

### RETURN VALUE

0: Success

### LIBRARY

FLASH.LIB

### SEE ALSO

flash\_erasechip, flash\_erasector, 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

<b>fd</b>	The FlashDescriptor of the Flash Memory.
<b>sector</b>	The sector to set the XPC window to.

### RETURN VALUE

The logical offset of the sector.

### LIBRARY

FLASH.LIB

### SEE ALSO

flash\_erasechip, flash\_erasector, 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

<b>fd</b>	The FlashDescriptor of the Flash Memory to write to.
<b>sector</b>	The destination sector.
<b>buffer</b>	The physical address of the source. TIP: A logical address can be changed to a physical address with the function <b>paddr</b>

### RETURN VALUE

0: 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 is:  $22.7 - 2 \times 10.3 = 2.1$ .

### LIBRARY

MATH.LIB

### SEE ALSO

ceil, floor

## **fopen\_rd (FS1)**

```
int fopen_rd(File* f, FileNumber fnum);
```

### **DESCRIPTION**

Opens a file for reading.

### **PARAMETERS**

<b>f</b>	A pointer to the file to read.
<b>fnum</b>	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

## **fopen\_rd (FS2)**

```
int fopen_rd(File* f, FileNumber name);
```

### **DESCRIPTION**

Open file for reading only. See **fopen\_wr ( )** for a more detailed description.

### **PARAMETERS**

<b>f</b>	Pointer to file descriptor (uninitialized).
<b>name</b>	File number (1 to 255 inclusive).

### **RETURN VALUE**

0: Success  
!0: Failure

### **ERRNO VALUES**

**ENOENT** - File does not exist, or metadata extent number does not match an existing file.

### **LIBRARY**

fs2.lib

### **SEE ALSO**

fclose, fopen\_wr (FS2)

## **fopen\_wr (FS1)**

```
int fopen_wr(File* f, FileNumber fnum);
```

### **DESCRIPTION**

Opens a file for writing.

### **PARAMETERS**

<b>f</b>	A pointer to the file to write.
<b>fnum</b>	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

## **fopen\_wr (FS2)**

```
int fopen_wr(File* f, FileNumber name);
```

### **DESCRIPTION**

Open file for read or write. The given file number is composed of two parts: the low byte contains the file number (1 to 255 inclusive) and the high byte, if not zero, contains the metadata extent number. If the extent number is zero, it defaults to the correct metadata extent - this is for the purpose of validating an expected extent number. Most applications should just pass the file number with zero high byte.

A file may be opened multiple times, with a different file descriptor pointer for each call, which allows the file to be read or written at more than one position at a time. A reference count for the actual file is maintained, so that the file can only be deleted when all file descriptors referring to this file are closed.

**fopen\_wr()** or **fopen\_rd()** must be called before any other function, which requires a File pointer, is called from this library. The "current position" is set to zero i.e. the start of the file.

When a file is created, it is automatically opened for writing thus a subsequent call to **fopen\_wr()** is redundant.

### **PARAMETERS**

<b>f</b>	Pointer to file descriptor (uninitialized).
<b>name</b>	File number (1 to 255 inclusive).

### **RETURN VALUE**

0: Success  
!0: Failure

### **ERRNO VALUES**

**ENOENT** - File does not exist, or metadata extent number does not match an existing file.

### **LIBRARY**

fs2.lib

### **SEE ALSO**

fclose, fopen\_rd (FS2)

## **forceSoftReset**

```
void forceSoftReset();
```

### **DESCRIPTION**

Forces the board into a software reset by jumping to the start of the BIOS.

### **LIBRARY**

SYS.LIB

## **fread (FS1)**

```
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**

<b>f</b>	A pointer to the file to read from
<b>buf</b>	A pointer to the destination buffer.
<b>len</b>	Number of bytes to copy.

### **RETURN VALUE**

Number of bytes read.

### **LIBRARY**

FILESYSTEM.LIB

## **fread (FS2)**

```
int fread(File* f, void* buf, int len);
```

### **DESCRIPTION**

Read data from the "current position" of the given file. When the file is opened, the current position is 0, i.e. at the start of file. Subsequent reads or writes advance the position by the number of bytes read/written **fseek()** can also be used to position the read point.

If the application permits, it is much more efficient to read multiple data bytes rather than reading one-by-one.

### **PARAMETERS**

<b>f</b>	Pointer to file descriptor (initialized by <b>fopen_rd()</b> , <b>fopen_wr()</b> or <b>fcreate()</b> ).
<b>buf</b>	Data buffer located in root data memory or stack. This must be dimensioned with at least <b>len</b> bytes.
<b>len</b>	Length of data to read (0 to 32767 inclusive).

### **RETURN VALUE**

**len**: Success  
<**len** - partial success. Returns amount successfully read. **errno** gives further details (probably 0 meaning that end-of-file was encountered).  
0: Failure, or **len** was zero.

### **LIBRARY**

fs2.LIB

### **ERRNO VALUES**

**EBADFD** - File descriptor not opened.  
**EINVAL** - **len** less than zero.  
0 - Success, but **len** was zero or EOF was reached prior to reading **len** bytes.  
**EIO** - I/O error.

### **SEE ALSO**

fseek (FS2), fwrite (FS2)

## frexp

```
float frexp(float x, int *n);
```

### DESCRIPTION

Splits **x** into a fraction and exponent,  $f \cdot (2^{*n})$

### PARAMETERS

<b>x</b>	Number to split
<b>n</b>	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 (FS1)

```
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

<b>reserveblocks</b>	Starting address of the flash file system. When <b>FS_FLASH</b> is defined this value should be 0 or a multiple of the block size. When <b>FS_RAM</b> is defined this parameter is ignored.
<b>num_blocks</b>	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.
<b>wearlevel</b>	This value should be 1 on a new Flash Memory, and some higher value on an unformatted used Flash Memory. If you are re-formatting a Flash Memory you can set <b>wearlevel</b> 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
#include "filesystem.lib"
#define RESERVE 0
#define BLOCKS 64
#define WEAR 1

main() {
    if(fs_format(RESERVE,BLOCKS,WEAR)) {
        printf("error formatting flash\n");
    } else {
        printf("flash successfully formatted\n");
    }
}
```

## fs\_format (FS2)

```
int fs_format(long reserveblocks, int num_blocks, unsigned
wearlevel)
```

### DESCRIPTION

Format all extents of the file system. This must be called after calling `fs_init()`. Only extents that are not defined as reserved are formatted. All files are deleted.

### PARAMETERS

<b>reserveblocks</b>	Must be zero. Retained for backward compatibility.
<b>num_blocks</b>	Ignored (backward compatibility).
<b>wearlevel</b>	Initial wearlevel value. This should be 1 if you have a new flash, and some larger number if the flash is used. If you are reformatting a flash, you can use 0 to use the old flash wear levels.

### RETURN VALUE

0: Success  
!0: Failure

### ERRNO VALUES

**EINVAL** - the **reserveblocks** parameter was non-zero.  
**EBUSY** - one or more files were open.  
**EIO** - I/O error during format. If this occurs, retry the format operation. If it fails again, there is probably a hardware error.

### SEE ALSO

`fs_init (FS2)`, `lx_format`

## **fs\_init (FS1)**

```
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**

<b>reserveblocks</b>	Starting address of the flash file system. When <b>FS_FLASH</b> is defined this value should be 0 or a multiple of the block size. When <b>FS_RAM</b> is defined this parameter is ignored.
<b>num_blocks</b>	The number of blocks that the file system contains. By default the block size is 4096 bytes.

### **RETURN VALUE**

0: Success  
1: Failure

### **LIBRARY**

FILESYSTEM.LIB

## **fs\_init (FS2)**

```
int fs_init(long reserveblocks, int num_blocks);
```

### **DESCRIPTION**

Initialize the filesystem. The static structure `_fs` contains information that defines the number and parameters associated with each extent or "partition". This function must be called before any of the other functions in this library, except for `fs_setup()`, `fs_get*_lx()` and `fs_get_lx_size()`.

Pre-main initialization will create up to 3 devices:

- The second flash device (if available on the board)
- Battery-backed SRAM (if `FS2_RAM_RESERVE` defined)
- The first (program) flash (if both `XMEM_RESERVE_SIZE` and `FS2_USE_PROGRAM_FLASH` defined).

The LX numbers of the default devices can be obtained using the `fs_get_flash_lx()`, `fs_get_ram_lx()` and `fs_get_other_lx()` calls.

If none of these devices can be set up successfully, `fs_init()` will return `ENOSPC` when called.

This function performs complete consistency checks and, if necessary, fixups for each LX. It may take up to several seconds to run. It should only be called once at application initialization time.

### **PARAMETERS**

<code>reserveblocks</code>	Must be zero. Retained for backward compatibility.
<code>num_blocks</code>	Ignored (backward compatibility).

### **RETURN VALUE**

0: Success  
!0: Failure

### **ERRNO VALUES**

**EINVAL** - the reserveblocks parameter was non-zero.  
**EIO** - I/O error. This indicates a hardware problem.  
**ENOMEM** - Insufficient memory for required buffers.  
**ENOSPC** - No valid extents obtained e.g. there is no recognized flash or RAM memory device available.

### **LIBRARY**

`fs2.lib`

### **SEE ALSO**

`fs_setup (FS2)`, `fs_get_flash_lx (FS2)`

## **fs\_reserve\_blocks (FS1)**

```
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: Success  
1: Failure

### **LIBRARY**

FILESYSTEM.LIB

## **fsck (FS1)**

```
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: Success;  
!0: Failure, this is a bitmask indicating which checks failed.

### **LIBRARY**

FILESYSTEM.LIB

## **fseek (FS1)**

```
int fseek(File* f, long to, char whence);
```

### **DESCRIPTION**

Places the read pointer at a desired location in the file.

### **PARAMETERS**

<b>f</b>	A pointer to the file to seek into.
<b>to</b>	The number of bytes to move the read pointer. This can be a positive or negative number.
<b>whence</b>	The location in the file to offset from. This is one of the following constants. <b>SEEK_SET</b> - Seek from the beginning of the file. <b>SEEK_CUR</b> - Seek from the current read position in the file. <b>SEEK_END</b> - 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**: Success  
**1**: Failure

### **LIBRARY**

FILESYSTEM.LIB

## **fseek (FS2)**

```
int fseek(File * f, long where, char whence)
```

### **DESCRIPTION**

Set the current read/write position of the file. Bytes in a file are sequentially numbered starting at zero. If the current position is zero, then the first byte of the file will be read or written. If the position equals the file length, then no data can be read, but any write will append data to the file.

**fseek()** allows the position to be set relative to the start or end of the file, or relative to its current position.

In the special case of **SEEK\_RAW**, an unspecified number of bytes beyond the "known" end-of-file may be readable. The actual amount depends on the amount of space left in the last internal block of the file. This mode only applies to reading, and is provided for the purpose of data recovery in the case that the application knows more about the file structure than the filesystem.

### **PARAMETERS**

<b>f</b>	Pointer to file descriptor (initialized by <b>fopen_rd()</b> , <b>fopen_wr()</b> or <b>fcreate()</b> ).
<b>where</b>	New position, or offset.
<b>whence</b>	One of the following values: <b>SEEK_SET</b> : 'where' (non-negative only) is relative to start of file. <b>SEEK_CUR</b> : 'where' (positive or negative) is relative to the current position. <b>SEEK_END</b> : 'where' (non-positive only) is relative to the end of the file. <b>SEEK_RAW</b> : Similar to <b>SEEK_END</b> , except the file descriptor is set in a special mode which allows reading beyond the end of the file.

### **RETURN VALUE**

**0**: Success.

**!0**: The computed position was outside of the current file contents, and has been adjusted to the nearest valid position.

### **ERRNO VALUES**

None.

### **LIBRARY**

fs2.lib

### **SEE ALSO**

ftell (FS2), fread (FS2), fwrite (FS2)

## **fs\_get\_flash\_lx (FS2)**

```
FSLXnum fs_get_flash_lx(void)
```

### **DESCRIPTION**

Returns the logical extent number of the preferred flash device. This is the second flash if one is available on your hardware, otherwise it is the reserved area in your program flash. In order for the program flash to be available for use by the file system, you must define two constants: the first constant is **XMEM\_RESERVE\_SIZE** near the top of **BIOS\RABBITBIOS.C**. This value is set to the amount of program flash to reserve (in bytes). This is required by the BIOS. The second constant is set in your code before **#use "fs2.lib"**. **FS2\_USE\_PROGRAM\_FLASH** must be defined to the number of KB (1024 bytes) that will actually be used by the file system. If this is set to a larger value than the actual amount of reserved space, then only the actual amount will be used.

The sample program **SAMPLES\FILESYSTEM\FS2INFO.C** demonstrates use of this function.

This function may be called before calling **fs\_init()**.

### **RETURN VALUE**

- 0: There is no flash file system available.
- !0: Logical extent number of the preferred flash.

### **LIBRARY**

FS2.lib

### **SEE ALSO**

fs\_get\_ram\_lx (FS2), fs\_get\_other\_lx (FS2)

## **fs\_get\_lx (FS2)**

```
FSLXnum fs_get_lx(int meta)
```

### **DESCRIPTION**

Return the current extent (LX) number for file creation. Each file has two parts: the main bulk of data, and the metadata which is a relatively small, fixed, amount of data used to journal changes to the file. Both data and metadata can reside on the same extent, or they may be separated.

### **PARAMETERS**

<b>meta</b>	<b>1:</b> return logical extent number for metadata. <b>0:</b> return logical extent number for data.
-------------	--

### **RETURN VALUE**

Logical extent number.

### **LIBRARY**

FS2.lib

### **SEE ALSO**

fcreate (FS2), fs\_set\_lx (FS2)

## **fs\_get\_lx\_size (FS2)**

```
long fs_get_lx_size(FSLXnum lxn, int all, word ls_shift)
```

### **DESCRIPTION**

Returns the size of the specified logical extent, in bytes. This information is useful when initially partitioning an LX, or when estimating the capacity of an LX for user data. **all** is a flag which indicates whether to return the total data capacity (as if all current files were deleted) or whether to return just the available data capacity. The return value accounts for the packing efficiency which will be less than 100% because of the bookkeeping overhead. It does not account for the free space required when any updates are performed; however this free space may be shared by all files on the LX. It also does not account for the space required for file metadata. You can account for this by adding one logical sector for each file to be created on this LX. You can also specify that the metadata be stored on a different LX by use of **fs\_set\_lx()**.

This function may be called either before or after **fs\_init()**. If called before, then the **ls\_shift** parameter must be set to the value to be used in **fs\_setup()**, since the LS size is not known at this point. **ls\_shift** can also be passed as zero, in which case the default size will be assumed. **all** must be non-zero if called before **fs\_init()**, since the number of files in use is not yet known.

### **PARAMETERS**

<b>lxn</b>	Logical extent number to query.
<b>all</b>	Boolean: 0 for current free capacity only, 1 for total. Must use 1 if calling before <b>fs_init()</b> .
<b>ls_shift</b>	Logical sector shift i.e. log base 2 of LS size (6 to 13); may be zero to use default.

### **RETURN VALUE**

- 0: The specified LX does not exist
- !0: Capacity of the LX in bytes

### **LIBRARY**

FS2.lib

## **fs\_get\_other\_lx (FS2)**

**FSLXnum fs\_get\_other\_lx(void)**

### **DESCRIPTION**

Returns the logical extent number of the non-preferred flash device. If it exists, this is usually the program flash. See the description under **fs\_get\_flash\_lx()** for details about setting up the program flash for use by the filesystem.

The sample program **SAMPLES\FILESYSTEM\FS2INFO.C** demonstrates use of this function.

This function may be called before calling **fs\_init()**.

### **RETURN VALUE**

- 0**: There is no other flash filesystem available
- !0**: Logical extent number of the non-preferred flash.

### **LIBRARY**

**FS2.LIB**

### **SEE ALSO**

**fs\_get\_ram\_lx (FS2), fs\_get\_flash\_lx (FS2)**

## **fs\_get\_ram\_lx (FS2)**

```
FSLXnum fs_get_ram_lx(void)
```

### **DESCRIPTION**

Return the logical extent number of the RAM file system device. This is only available if you have defined **FS2\_RAM\_RESERVE** to a non-zero number of bytes in the file **LIB\BIOSLIB\STACK.LIB**.

A RAM filesystem is only really useful if you have battery-backed SRAM on the board. You can still use a RAM file system on volatile RAM, but of course files will not persist over power cycles and you should explicitly format the RAM filesystem at power-up.

The sample program **SAMPLES\FILESYSTEM\FS2INFO.C** demonstrates use of this function.

This function may be called before calling **fs\_init()**.

### **RETURN VALUE**

- 0: There is no RAM filesystem available
- !0: Logical extent number of the RAM device.

### **LIBRARY**

FS2.LIB

### **SEE ALSO**

fs\_get\_flash\_lx (FS2), fs\_get\_other\_lx (FS2)

## **fs\_set\_lx (FS2)**

```
int fs_set_lx(FSLXnum meta, FSLXnum data)
```

### **DESCRIPTION**

Sets the default extent numbers for file creation. Each file has two parts: the main bulk of data, and the metadata which is a relatively small, fixed amount of data used to journal changes to the file. Both data and metadata can reside on the same extent, or they may be separated.

The file creation functions allow the metadata extent to be explicitly specified (in the high byte of the file number), however it is usually easier to call **fs\_set\_lx()** to set appropriate defaults. Calling **fs\_set\_lx()** is the only way to specify the data extent.

If **fs\_set\_lx()** is never called, both data and metadata will default to the first non-reserved extent number.

### **PARAMETERS**

**meta**                    Extent number for metadata.

**data**                    Extent number for data.

### **RETURN VALUE**

0: Success.

!0: Error, e.g. non-existent LX number.

### **ERRNO VALUES**

**ENODEV** - no such extent number, or extent is reserved.

### **LIBRARY**

FS2.LIB

### **SEE ALSO**

fcreate (FS2)

## fs\_setup (FS2)

```
FSLXnum fs_setup(FSLXnum lxn, word ls_shift, int reserve_it,  
                void * rfu, int partition_it, word part, word part_ls_shift,  
                int part_reserve, void * part_rfu);
```

### DESCRIPTION

To modify or add to the default extents, this function must be called before calling `fs_init()`. If called after `fs_init()`, the filesystem will be corrupted.

`fs_setup()` runs in one of two basic modes, determined by the `partition_it` parameter. If `partition_it` is non-zero, then the specified extent (`lxn`, which must exist), is split into two extents according to the given proportions. If `partition_it` is zero, then the specified extent must not exist; it is created. This use is beyond the scope of this note, since it involves filesystem internals. The partitioning usage is described here.

`partition_it` may be `FS_MODIFY_EXTENT` in which case the base extent, `lxn`, is modified to use the specified `ls_shift` and `reserve_it` parameters (the other parameters are ignored).

`partition_it` may be set to `FS_PARTITION_FRACTION` (other values reserved). This causes extent number `lxn` to be split. The first half is still referred to as extent `lxn`, and the other half is assigned a new extent number, which is returned.

The base extent number may itself have been previously partitioned, or it should be 1 for the 2nd flash device, or possibly 2 for the NVRAM device.

### PARAMETERS

<code>lxn</code>	Base extent number to partition or modify.
<code>ls_shift</code>	New logical sector size to assign to base partition, or zero to not alter it. This is expressed as the log base 2 of the desired size, and must be a number between 6 and 13 inclusive.
<code>reserve_it</code>	<b>TRUE</b> if base partition is to be marked reserved.
<code>rfu</code>	A pointer reserved for future use. Pass as <b>NULL</b> .
<code>partition_it</code>	Must be set to <code>FS_PARTITION_FRACTION</code> or <code>FS_MODIFY_EXTENT</code> . The following parameters are ignored if this parameter is not <code>FS_PARTITION_FRACTION</code> .

**part**                    The fraction of the existing base extent to assign to the new extent. This number is expressed as a fixed-point binary number with the binary point to the left of the MSB e.g. 0x3000 assigns 3/16 of the base extent to the new partition, updating the base extent to 13/16 of its original size. The nearest whole number of physical sectors is used for each extent.

**part\_ls\_shift**       Logical sector size to assign to the new extent, or zero to use the same LS size as the base extent. Expressed in same units as parameter 2.

**part\_reserve**        **TRUE** if the new extent is to be reserved.

**part\_rfu**             A pointer reserved for future use. Pass as **NULL**.

#### RETURN VALUE

0: Failure, extent could not be partitioned.  
!0: Success, number of the new extent, or same as **lxd** for existing extent modification.

#### ERRNO VALUES

**ENOSPC** - one or other half would contain an unusably small number of logical sectors, or the extent table is full. In the latter case, **#define FS\_MAX\_LX** to a larger value.

**EINVAL** - **partition\_it** set to an invalid value, or other parameter invalid.

**ENODEV** - specified base extent number not defined.

#### LIBRARY

FS2.LIB

#### SEE ALSO

fs\_init (FS2)

## **fs\_sync (FS2)**

```
int fs_sync(void);
```

### **DESCRIPTION**

Flush any buffers retained in RAM to the underlying hardware device. The file system does not currently perform any buffering, however future revisions of this library may introduce buffering to improve performance. This function is similar to **fflush()**, except that the entire file system is synchronized instead of the data for just one file. Use **fs\_sync()** in preference to **fflush()** if there is only one extent in the filesystem.

### **RETURN VALUE**

0: Success

!0: Failure

### **ERRNO VALUES**

**EIO** - I/O error.

### **LIBRARY**

FS2.LIB

### **SEE ALSO**

`fflush (FS2)`

## **ftell (FS1)**

```
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: Success  
-1: Failure

### **LIBRARY**

FILESYSTEM.LIB

## **ftell (FS2)**

```
long ftell(File * f);
```

### **DESCRIPTION**

Return the current read/write position of the file. Bytes in a file are sequentially numbered starting at zero. If the current position is zero, then the first byte of the file will be read or written. If the position equals the file length, then no data can be read, but any write will append data to the file.

Note that no checking is done to see if the file descriptor is valid. If the File is not actually open, the return value will be random.

### **PARAMETERS**

**f**                    Pointer to file descriptor (initialized by **fopen\_rd()**, **fopen\_wr()** or **fcreate()**).

### **RETURN VALUE**

Current read/write position (0 to length-of-file).

### **ERRNO VALUES**

None

### **LIBRARY**

fs2.lib

### **SEE ALSO**

fseek (FS2)

## **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 memory pointed to by **buffer**.

### **PARAMETERS**

<b>f</b>	A pointer to the file.
<b>count</b>	Number of bytes to shift out.
<b>*buffer</b>	Location to store shifted bytes. If this is <b>NULL</b> , the bytes will be discarded.

### **RETURN VALUE**

Number of bytes shifted out: Success  
0: Error

### **LIBRARY**

FILESYSTEM.LIB

## **fwrite (FS1)**

```
int fwrite(File* f, char* buf, int len);
```

### **DESCRIPTION**

Appends **len** bytes from the source buffer to the end of the file.

### **PARAMETERS**

<b>f</b>	A pointer to the file to write to.
<b>buf</b>	A pointer to the source buffer.
<b>len</b>	The number of bytes to write.

### **RETURN VALUE**

The number of bytes written: Success  
0: Failure

### **LIBRARY**

FILESYSTEM.LIB

## **fwrite (FS2)**

```
int fwrite(File* f, void* buf, int len);
```

### **DESCRIPTION**

Write data to file opened for writing. The data is written starting at the "current position". This is zero (start of file) when it is opened or created, but may be changed by `fread`, `fwrite`, `fshift` or `fseek` functions. After writing the data, the current position is advanced to the position just after the last byte written. Thus, sequential calls to `fwrite()` will add or append data contiguously.

Unlike the previous file system (**FILESYSTEM.LIB**), this library allows files to be overwritten not just appended. Internally, overwrite and append are different operations with differing performance, depending on the underlying hardware. Generally, appending is more efficient especially with byte-writable flash memory. If the application allows, it is preferable to use append/shift rather than overwrite. In order to ensure that data is appended, use `fseek(f, 0, SEEK_END)` before calling `fwrite()`.

The same "current position" pointer is used for both read and write. If interspersing read and write, then `fseek()` should be used to ensure the correct position for each operation. Alternatively, the same file can be opened twice, with one descriptor used for read and the other for write. This precludes use of `fshift()`, since it does not tolerate shared files.

### **PARAMETERS**

<b>f</b>	Pointer to file descriptor (initialized by <code>fopen_wr()</code> or <code>fcreate()</code> ).
<b>buf</b>	Data buffer located in root data memory or stack.
<b>len</b>	Length of data (0 to 32767 inclusive).

### **RETURN VALUE**

**len**: Success  
**<len**: Partial success. Returns amount successfully written. **errno** gives further details  
**0**: Failure, or **len** was zero.

### **ERRNO VALUES**

**EBADFD** - File descriptor not opened, or is read-only.  
**EINVAL** - **len** less than zero.  
**0** - Success, but **len** was zero.  
**EIO** - I/O error.  
**ENOSPC** - extent out of space.

### **LIBRARY**

`fs2.LIB`

### **SEE ALSO**

`fread (FS2)`

## 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 9 digits, no decimal points, and a minus sign if **f** is negative. The function returns the exponent (of 10) that should be used to compensate for the string: **ftoa(1.0, buf)** yields **buf="100000000"**, and returns **-8**.

### PARAMETERS

<b>f</b>	Float number to convert
<b>buf</b>	Converted string. The string is no longer than 10 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

<b>dataarray</b>	Data buffer
<b>count</b>	Number of bytes. Max is 255.
<b>accum</b>	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`

## gps\_get\_position

```
int gps_get_position(GPSPosition *newpos, char *sentence);
```

### DESCRIPTION

Parses a sentence to extract position data. This function is able to parse any of the following GPS sentence formats: GGA, GLL or RMC.

### PARAMETERS

**newpos**            A `GPSPosition` structure to fill.

**sentence**         A string containing a line of GPS data. in NMEA-0183 format.

### RETURN VALUE

0: Success  
-1: Parsing error  
-2: Sentence marked invalid

### LIBRARY

`gps.lib`

## gps\_get\_utc

```
int gps_get_utc(struct tm *newtime, char *sentence);
```

### DESCRIPTION

Parses an RMC sentence to extract time data.

### PARAMETERS

<b>newtime</b>	<b>tm</b> structure to fill with new UTC time.
<b>sentence</b>	A string containing a line of GPS data in NMEA-0183 format (RMC sentence).

### RETURN VALUE

0: Success  
-1: Parsing error  
-2: Sentence marked invalid

### LIBRARY

gps.lib

## gps\_ground\_distance

```
float gps_ground_distance(GPSPosition *a, GPSPosition *b);
```

### DESCRIPTION

Calculates ground distance (in km) between to geographical points. (Uses spherical earth model.)

### PARAMETERS

<b>a</b>	First point.
<b>b</b>	Second point.

### RETURN VALUE

Distance in kilometers

### LIBRARY

gps.lib

## hanncplx

```
void hanncplx(int *x, int N, int *blockexp)
```

### DESCRIPTION

Convolve 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

<b>x</b>	Pointer to <b>N</b> -element array of complex fractions.
<b>N</b>	Number of complex elements in array <b>x</b> .
<b>blockexp</b>	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

Convolve 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  $X[N-1]$  component (and vice versa), and should not arbitrarily be set to zero unless these components are unimportant.

### PARAMETERS

<b>x</b>	Pointer to <b>N</b> -element array of complex fractions.
<b>N</b>	Number of complex elements in array <b>x</b> .
<b>blockexp</b>	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 hexadecimal number and puts result into **buf**.

### PARAMETERS

<b>value</b>	16-bit number to convert
<b>buf</b>	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 characte,  
!0 otherwise

### LIBRARY

STRING.LIB

### SEE ALSO

isalnum, isdigit, ispunct

## isctrl

```
int isctrl(int c);
```

### DESCRIPTION

Tests for a control character:  $0 \leq c \leq 31$  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:  $33 \leq c \leq 126$

### PARAMETERS

**c** Character to test.

### RETURN VALUE

0: **c** is not a printing character  
!0: **c** is a printing character

### 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:  
 $9 \leq c \leq 13$  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:  $32 \leq c \leq 126$

### 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.

<u>Character</u>	<u>Decimal Code</u>
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 hexadecimal digit: 0 - 9, A - F, a - f

### **PARAMETERS**

**c**                    Character to test.

### **RETURN VALUE**

0 if not a hexadecimal digit, !0 otherwise.

### **LIBRARY**

STRING.LIB

### **SEE ALSO**

isdigit, isalpha, isalpha

## itoa

```
char *itoa(int value, char *buf);
```

### DESCRIPTION

Places up to a 5-digit character string, with a minus sign in the leftmost digit when appropriate, at **\*buf**. The string represents **value**, a signed number.

Leading zeros are suppressed in the character string, except for one zero digit when **value** = 0. The longest possible string is “-32768.”

### PARAMETERS

<b>value</b>	16-bit signed number to convert
<b>buf</b>	Character string of converted number in base 10

### RETURN VALUE

Pointer to the end (**NULL** terminator) of the string in **buf**.

### LIBRARY

STDIO.LIB

### SEE ALSO

atoi, utoa, ltoa

## i2c\_check\_ack

```
int i2c_check_ack();
```

### DESCRIPTION

Checks if slave pulls data low for ACK on clock pulse. Allows for clocks stretching on SCL going high.

### RETURN VALUE

0: ACK sent from slave  
1: NAK sent from slave  
-1: Timeout occurred

### LIBRARY

I2C.LIB

### SEE ALSO

Application Note 215, *Using the I<sup>2</sup>C Bus with a Rabbit Microprocessor*.

## `i2c_init`

```
void i2c_init();
```

### DESCRIPTION

Sets up the SCL and SDA port pins for open-drain output.

### LIBRARY

I2C.LIB

### SEE ALSO

Application Note 215, *Using the I<sup>2</sup>C Bus with a Rabbit Microprocessor*.

## `i2c_read_char`

```
int i2c_read_char(char *ch);
```

### DESCRIPTION

Reads 8 bits from the slave. Allows for clocks stretching on all SCL going high. This is not in the protocol for I<sup>2</sup>C, but allows I<sup>2</sup>C slaves to be implemented on slower devices.

### PARAMETERS

**ch**                    A one character return buffer.

### RETURN VALUE

0: Success;  
-1: Clock stretching timeout

### LIBRARY

I2C.LIB

### SEE ALSO

Application Note 215, *Using the I<sup>2</sup>C Bus with a Rabbit Microprocessor*.

## `i2c_send_ack`

```
int i2c_send_ack();
```

### DESCRIPTION

Sends ACK sequence to slave. ACK is usually sent after a successful transfer, where more bytes are going to be read.

### RETURN VALUE

0: Success;  
-1: Clock stretching timeout

### LIBRARY

I2C.LIB

### SEE ALSO

Application Note 215, *Using the I<sup>2</sup>C Bus with a Rabbit Microprocessor*.

## `i2c_send_nak`

```
int i2c_send_nak();
```

### DESCRIPTION

Sends NAK sequence to slave. NAK is often sent when the transfer is finished.

### RETURN VALUE

0: Success;  
-1: Clock stretching timeout

### LIBRARY

I2C.LIB

### SEE ALSO

Application Note 215, *Using the I<sup>2</sup>C Bus with a Rabbit Microprocessor*.

## `i2c_start_tx`

```
int i2c_start_tx();
```

### DESCRIPTION

Initiates I<sup>2</sup>C transmission by sending the start sequence, which is defined as bringing SDA low while SCL is high, i.e. data goes low while the clock is high. This function first waits for possible clock stretching, which is when a bus peripheral holds SCK low.

### RETURN VALUE

0: Success;  
-1: Clock stretching timeout.

### LIBRARY

I2C.LIB

### SEE ALSO

Application Note 215, *Using the I<sup>2</sup>C Bus with a Rabbit Microprocessor*.

## `i2c_startw_tx`

```
int i2c_startw_tx();
```

### DESCRIPTION

Initiates I<sup>2</sup>C transmission by sending the start sequence, which is defined as bringing SDA low while SCL is high, i.e. data goes low while the clock is high. This function first waits for possible clock stretching, which is when a bus peripheral holds SCL low. Inserts delay after S pulse.

### RETURN VALUE

0: Success;  
-1: Clock stretching timeout

### LIBRARY

I2C.LIB

### SEE ALSO

Application Note 215, *Using the I<sup>2</sup>C Bus with a Rabbit Microprocessor*.

## `i2c_stop_tx`

```
void i2c_stop_tx();
```

### DESCRIPTION

Sends the stop sequence to the slave, which is defined as bringing SDA high while SCL is high, i.e., the clock goes high, then data goes high.

### LIBRARY

I2C.LIB

### SEE ALSO

Application Note 215, *Using the I<sup>2</sup>C Bus with a Rabbit Microprocessor*.

## `i2c_write_char`

```
int i2c_write_char(char d)
```

### DESCRIPTION

Sends 8 bits to slave. Checks if slave pulls data low for ACK on clock pulse. Allows for clocks stretching on SCL going high.

### PARAMETERS

**d**                      Character to send

### RETURN VALUE

0: Success  
-1: Clock stretching timeout  
1: NAK sent from slave

### LIBRARY

I2C.LIB

### SEE ALSO

Application Note 215, *Using the I<sup>2</sup>C Bus with a Rabbit Microprocessor*.

## 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  $x \cdot (2^{**n})$

### PARAMETERS

<b>x</b>	The value between 0.5 inclusive, and 1.0
<b>n</b>	An integer

### RETURN VALUE

The result of  $x \cdot (2^n)$

### 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  $x \leq 0$ .

### 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  $x \leq 0$ .

### 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

<code>env</code>	Environment previously saved with <code>setjmp</code> .
<code>val</code>	Integer result of <code>setjmp</code> .

### LIBRARY

`SYS.LIB`

### SEE ALSO

`setjmp`

## loophead

```
void loophead();
```

### DESCRIPTION

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

When two costatements are requesting access to a single-user cofunction, 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. See `Samples\Cofunc\Cofaband.c` for sample code showing abandonment handling.

### PARAMETERS

None

### LIBRARY

`COFUNC.LIB`

## loopinit

```
void loopinit();
```

### DESCRIPTION

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()`.

### PARAMETERS

None

### LIBRARY

COFUNC.LIB

## ltoa

```
char *ltoa(long num, char *ibuf)
```

### DESCRIPTION

This function outputs a signed long number to the character array.

### PARAMETERS

<code>num</code>	Signed long number
<code>ibuf</code>	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

## lx\_format

```
int lx_format(FSLXnum lxn, long wearlevel)
```

### DESCRIPTION

Format a specified file system extent. This must not be called before calling `fs_init()`. All files which have either or both metadata and data on this extent are deleted. Formatting can be quite slow (depending on hardware) so it is best performed after power-up, if at all.

### PARAMETERS

<code>lxn</code>	Logical extent number (1..fs.num_lx inclusive).
<code>wearlevel</code>	Initial wearlevel value. This should be 1 if you have a new flash, and some larger number if the flash is used. If you are reformatting a flash, you can use 0 to use the old flash wear levels.

### RETURN VALUE

0: Success  
!0: Failure

### ERRNO VALUES

**ENODEV** - no such extent number, or extent is reserved.  
**EBUSY** - one or more files were open on this extent.  
**EIO** - I/O error during format. If this occurs, retry the format operation. If it fails again, there is probably a hardware error.

### LIBRARY

fs2.LIB

### SEE ALSO

fs\_init, fs\_format

## 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

<b>src</b>	Pointer to memory source.
<b>ch</b>	Character to search for.
<b>n</b>	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

strchr, 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

<b>s1</b>	Pointer to block 1.
<b>s2</b>	Pointer to block 2.
<b>n</b>	Maximum number of bytes to compare.

### RETURN VALUE

<0: A character in **str1** is less than the corresponding character in **str2**  
0: **str1** is identical to **str2**  
>0: 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

<b>dst</b>	Pointer to memory destination
<b>src</b>	Pointer to memory source
<b>n</b>	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

<b>dst</b>	Pointer to memory destination
<b>src</b>	Pointer to memory source
<b>n</b>	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

<b>dst</b>	Block of memory to set
<b>chr</b>	Byte destination
<b>n</b>	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

<b>timeptr</b>	Pointer to <b>tm</b> 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**

<b>timeptr</b>	Address to store date and time into structure:
<b>time</b>	Seconds since January 1, 1980.

### **RETURN VALUE**

0

### **LIBRARY**

RTCLOCK.LIB

### **SEE ALSO**

mkttime, tm\_rd, tm\_wr

## modf

```
float modf(float x, int *n);
```

### DESCRIPTION

Splits **x** into a fraction and integer, **f** + **n**.

### PARAMETERS

<b>x</b>	Floating-point integer
<b>n</b>	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  $\mu$ C/OS-II data; must be called before any other  $\mu$ C/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

<b>pevent</b>	Pointer to mailbox's event control block.
<b>timeout</b>	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.
<b>err</b>	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

<b>pevent</b>	Pointer to mailbox's event control block.
<b>msg</b>	Pointer to message to be posted. A <b>NULL</b> pointer must not be sent.

### RETURN VALUE

<b>OS_NO_ERR</b>	The call was successful and the message was sent.
<b>OS_MBOX_FULL</b>	The mailbox already contains a message. Only one message at a time can be sent and thus, the message <b>MUST</b> be consumed before another can be sent.
<b>OS_ERR_EVENT_TYPE</b>	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

<b>pevent</b>	Pointer to message mailbox's event control block.
<b>pdata</b>	Pointer to a data structure for information about the message mailbox

### RETURN VALUE

<b>OS_NO_ERR</b>	The call was successful and the message was sent
<b>OS_ERR_EVENT_TYPE</b>	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  $\mu$ C/OS-II.

### PARAMETERS

<b>addr</b>	Pointer to starting address of the partition.
<b>nblks</b>	Number of memory blocks to create in the partition.
<b>blksize</b>	The size (in bytes) of the memory blocks.
<b>err</b>	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

<b>pmem</b>	Pointer to partition's memory control block
<b>err</b>	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

<b>pmem</b>	Pointer to the partition's memory control block.
<b>pblk</b>	Pointer to the memory block being released.

### RETURN VALUE

<b>OS_NO_ERR</b>	The memory block was inserted into the partition.
<b>OS_MEM_FULL</b>	If returning a memory block to an already <b>FULL</b> 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

<b>pmem</b>	Pointer to partition's memory control block.
<b>pdata</b>	Pointer to structure for holding information about the partition.

### RETURN VALUE

<b>OS_NO_ERR</b>	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

<b>pevent</b>	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

<b>start</b>	Pointer to the base address of the message queue storage area. The storage area <b>MUST</b> be declared an array of pointers to void: <b>void *MessageStorage[qsize]</b> .
<b>qsize</b>	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

<b>pevent</b>	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

<b>pevent</b>	Pointer to message queue's event control block.
<b>timeout</b>	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.
<b>err</b>	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

<b>pevent</b>	Pointer to message queue's event control block.
<b>msg</b>	Pointer to the message to send. <b>NULL</b> 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

<code>pevent</code>	Pointer to message queue's event control block.
<code>msg</code>	Pointer to the message to send. <b>NULL</b> 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

<b>pevent</b>	Pointer to message queue's event control block.
<b>pdata</b>	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 signaled 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

<b>pevent</b>	Pointer to the desired semaphore's event control block
<b>pdata</b>	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 OSetTickPerSec(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 than 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  $\mu$ C/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 `OSTaskSwHook` 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

<b>oldprio</b>	The priority level to change from.
<b>newprio</b>	The priority level to change to.

### RETURN VALUE

**OS\_NO\_ERR** - The call was successful.  
**OS\_PRIO\_INVALID** - The priority specified is higher than the maximum allowed (i.e.  $\geq$  **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  $\mu$ C/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

<b>task</b>	Pointer to the task's starting address.
<b>pdata</b>	Pointer to a task's initial parameters.
<b>stk_size</b>	Number of bytes of the stack.
<b>prior</b>	The task's unique priority number.

### RETURN VALUE

**OS\_NO\_ERR** - The call was successful.

**OS\_PRIO\_EXIT** - 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.  $\geq$  **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  $\mu$ C/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

<b>task</b>	Pointer to task's code.
<b>pdata</b>	Pointer to optional data area; used to pass parameters to the task at start of execution.
<b>prio</b>	The task's unique priority number; the lower the number the higher the priority.
<b>id</b>	The task's identification number (0..65535).
<b>stk_size</b>	Size of the stack in number of elements. If <code>OS_STK</code> is set to <code>INT8U</code> , <code>stk_size</code> corresponds to the number of bytes available. If <code>OS_STK</code> is set to <code>INT16U</code> , <code>stk_size</code> contains the number of 16-bit entries available. Finally, if <code>OS_STK</code> is set to <code>INT32U</code> , <code>stk_size</code> contains the number of 32-bit entries available on the stack.
<b>pext</b>	Pointer to a user-supplied Task Control Block (TCB) extension.
<b>opt</b>	The lower 8 bits are reserved by $\mu$ C/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` - 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.  $\geq$  `OS_LOWEST_PRIO`).

### LIBRARY

UCOS2.LIB

### SEE ALSO

OSTaskCreate

## OSTaskCreateHook

```
void OSTaskCreateHook(OS_TCB *ptcb);
```

### DESCRIPTION

Called by  $\mu$ C/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.  $\geq$  `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  $\mu$ C/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.  $\geq$  `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

<code>prio</code>	Priority number of the task.
<code>pdata</code>	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.  $\geq$  `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.  $\geq$  **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  $\mu$ C/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

<code>prio</code>	The task's priority.
<code>pdata</code>	Pointer to a data structure of type <code>OS_STK_DATA</code> .

### 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 maximum allowed (i.e.  $\geq$  **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

<b>hours</b>	Number of hours that the task will be delayed (max. is 255)
<b>minutes</b>	Number of minutes (max. 59)
<b>seconds</b>	Number of seconds (max. 59)
<b>milli</b>	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.  $\geq$  **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 **OSTimeTickHook** 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  $\mu$ C/OS-II. The returned value corresponds to  $\mu$ C/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

## outchr

```
char outchr(char c, int n, int (*putc) ( ) );
```

### DESCRIPTION

Use `putc` to output `n` times the character `c`.

### PARAMETERS

<code>c</code>	Character to output
<code>n</code>	Number of times to output
<code>putc</code>	Routine to output one character. The function pointed to by <code>putc</code> 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

<b>string</b>	String to output
<b>putc</b>	Routine to output one character. The function pointed to by <b>putc</b> 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

<b>pointer</b>	The pointer to convert.
----------------	-------------------------

### RETURN VALUE

The physical address of the pointer.

### LIBRARY

XMEM.LIB

## **pktXclose**

```
void pktXclose();
```

### **DESCRIPTION**

Disables serial port X, where X is A|B|C|D.

### **LIBRARY**

PACKET.LIB

## **pktXgetErrors**

```
char pktXgetErrors();
```

### **DESCRIPTION**

Gets a bit field with flags set for any errors that occurred on port X, where X is A|B|C|D. These flags are then cleared, so that a particular error will only cause the flag to be set once.

### **RETURN VALUE**

A bit field with flags for various errors. The errors along with their bit masks are as follows:

```
PKT_BUFFEROVERFLOW 0x01  
PKT_RXOVERRUN 0x02  
PKT_PARITYERROR 0x04  
PKT_NOBUFFER 0x08
```

### **LIBRARY**

PACKET.LIB

## pktXinitBuffers

```
int pktXinitBuffers(int buf_count, int buf_size); X = A|B|C|D
```

### DESCRIPTION

Allocates extended memory for channel X receive buffers. This function should not be called more than once in a program. The total memory allocated is `buf_count*(buf_size + 2)` bytes.

### PARAMETERS

<b>buf_count</b>	The number of buffers to allocate. Each buffer can store one received packet. Increasing this number allows for more pending packets and a larger latency time before packets must be processed by the user's program.
<b>buf_size</b>	The number of bytes each buffer can accommodate. This should be set to the size of the largest possible packet that can be expected.

### RETURN VALUE

1: Success, extended memory was allocated.  
0: Failure, no memory allocated, the packet channel cannot be used.

### LIBRARY

PACKET.LIB

## pktXopen

```
int pktXopen(long baud, int mode, char options, int
(*test_packet)());
```

### DESCRIPTION

Opens serial port X, where X is A|B|C|D.

### PARAMETERS

<b>baud</b>	Bits per second of data transfer: minimum is 2400.
<b>mode</b>	Type of packet scheme used, the options are: <ul style="list-style-type: none"><li>• <b>PKT_GAPMODE</b></li><li>• <b>PKT_9BITMODE</b></li><li>• <b>PKT_CHARSMODE</b></li></ul>
<b>options</b>	Further specification for the packet scheme. The value of this depends on the mode used: <ul style="list-style-type: none"><li>• gap mode - minimum gap size(in byte times)</li><li>• 9-bit mode - type of 9-bit protocol<ul style="list-style-type: none"><li><b>PKT_RABBITSTARTBYTE</b></li><li><b>PKT_LOWSTARTBYTE</b></li><li><b>PKT_HIGHSTARTBYTE</b></li></ul></li><li>• char mode - character marking start of packet</li></ul>
<b>test_packet</b>	Pointer to a function that tests for completeness of a packet. The function should return 1 if the packet is complete, or 0 if more data should be read in. For gap mode the test function is not used and should be set to <b>NULL</b> .

### RETURN VALUE

- 1: The baud set on the rabbit is the same as the input baud.
- 0: The baud set on the rabbit does not match the input baud.

### LIBRARY

PACKET.LIB

## pktXreceive

```
int pktXreceive(void *buffer, int buffer_size);
```

### DESCRIPTION

Gets a received packet, if there is one, from serial port X, where X is A|B|C|D.

### PARAMETERS

**buffer**            A buffer for the packet to be written into.

**buffer\_size**    Length of the data buffer.

### RETURN VALUE

Number of bytes in the received packet: Successfully received packet.

0: No new packet has been received.

-1: The packet is too large for the given buffer

-2: A needed **test\_packet** function is not defined

### LIBRARY

PACKET.LIB

## pktXsend

```
int pktXsend(void *send_buffer, int buffer_length, char delay);
```

### DESCRIPTION

Initiates the sending of a packet of data using serial port X, where X is A|B|C|D. This function will always return immediately. If there is already a packet being transmitted, this call will return 0 and the packet will not be transmitted, otherwise it will return 1.

**pktXsending()** checks if the packet is done transmitting. The system will be using the buffer until then.

### PARAMETERS

<b>send_buffer</b>	The data to be sent
<b>buffer_length</b>	Length of the data buffer to transmit
<b>delay</b>	The number of byte times to delay before sending the data (0-255) This is used to implement protocol-specific delays between packets

### RETURN VALUE

- 1: The packet is going to be transmitted.
- 0: There is already a packet transmitting, and the new packet was refused.

### LIBRARY

PACKET.LIB

## pktXsending

```
int pktXsending();
```

### DESCRIPTION

Tests if a packet is currently being sent on serial port X, where X=A|B|C|D. If **pktXsending()** returns true, the transmitter is busy and cannot accept another packet.

### RETURN VALUE

- 1: A packet is being transmitted.
- 0: Port X is idle, ready for a new packet.

### LIBRARY

PACKET.LIB

## pktXsetParity

```
void pktXsetParity(char mode);
```

### DESCRIPTION

Configures parity generation and checking. Can also configure for 2 stop bits.

### PARAMETERS

<b>mode</b>	Code for mode of parity bit: <ul style="list-style-type: none"><li>• <b>PKT_NOPARITY</b> - no parity bit (8N1 format, default)</li><li>• <b>PKT_OPARITY</b> - odd parity (8O1 format)</li><li>• <b>PKT_EPARITY</b> - even parity (8E1 format)</li><li>• <b>PKT_TWOSTOP</b> - an extra stop bit (8N2 format)</li></ul>
-------------	---

### LIBRARY

PACKET.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

<b>x</b>	Variable of the polynomial.
<b>n</b>	The order of the polynomial
<b>c</b>	Array containing the coefficients of each power of <b>x</b> .

### RETURN VALUE

The polynomial value.

### LIBRARY

MATH.LIB

## pow

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

### DESCRIPTION

Raises **x** to the **y**th power.

### PARAMETERS

<b>x</b>	Value to be raised
<b>y</b>	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

<b>x</b>	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

$$Power[k] = (\text{Re } X[k])^2 + (\text{Im } X[k])^2$$

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 **fftre-  
al()**, 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

<b>x</b>	pointer to <b>N</b> -element array of complex fractions.
<b>N</b>	number of complex elements in array <b>x</b> .
<b>blockexp</b>	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**

<b>format</b>	String to be formatted.
<b>...</b>	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: Success.

### 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

<b>base</b>	Base address of the raw string data
<b>n</b>	Number of blocks to sort
<b>s</b>	Number of bytes in each block
<b>cmp</b>	User-supplied compare routine for two block pointers, <b>p</b> and <b>q</b> , that returns an int with the same rules used by Unix <b>strcmp(p,q)</b> : = 0   Blocks <b>p</b> and <b>q</b> are equal < 0 <b>p</b> < <b>q</b> > 0 <b>p</b> > <b>q</b>  Beware of using ordinary <b>strcmp()</b> —it requires a <b>NULL</b> 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 ( $2\pi$  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:  $0.0 \leq v < 1.0$ .

### 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:  $-1.0 \leq v < 1.0$ .

### 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:  $-16.0 \leq v < 16.0$ .

### 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 Real-time Clock (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 32 kHz crystal, use **read\_rtc\_32kHz()** 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 32 kHz oscillator. See `read_rtc` for more details.

### RETURN VALUE

Time in seconds since January 1, 1980 (if RTC set correctly).

### LIBRARY

RTCLOCK.LIB

## readUserBlock

```
int readUserBlock(void *dest, int addr, int numbytes)
```

### DESCRIPTION

Reads a number of bytes from the user block on the primary flash to a buffer in root memory. NOTE: portions of the user block may be used by the BIOS for your board to store values such as calibration constants! See the manual for your particular board for more information before overwriting any part of the user block. Also, see the *Rabbit 2000 Microprocessor Designer's Handbook* for more information on the user block.

### PARAMETERS

<b>dest</b>	Pointer to destination to copy data to.
<b>addr</b>	Address offset in user block to read from.
<b>numbytes</b>	Number of bytes to copy.

### RETURN VALUE

0: Success  
-1: Invalid address or range

### LIBRARY

IDBLOCK.LIB

### SEE ALSO

`writeUserBlock`

## 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

<b>address</b>	Address of byte containing bits 7-0
<b>bit</b>	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

<b>address</b>	Address of byte containing bits 7-0
<b>bit</b>	Bit location where 0 represents the least significant bit

### LIBRARY

UTIL.LIB

### SEE ALSO

res

## ResetErrorLog

```
void ResetErrorLog()
```

### DESCRIPTION

This function resets the exception and restart type counts in the error log buffer header. This function is not called by default from anywhere. It should be used to initialize the error log when a board is programmed by means other than Dynamic C, cloning, the Rabbit Field Utility (RFU), or a service processor. For example, if boards are mass produced with pre-programmed flash chips, then the test program that runs on the boards should call this function.

### LIBRARY

ERRORS.LIB

## root2xmem

```
int root2xmem(unsigned long dest, void *src, unsigned len);
```

### DESCRIPTION

Stores `len` characters from logical address `src` to physical address `dest`.

### PARAMETERS

<code>dest</code>	Physical address
<code>src</code>	Logical address
<code>len</code>	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

<b>rx_byte</b>	The 8 bit character being tested for parity.
<b>parity</b>	The character 'O' for odd parity, or the character 'E' for even parity.

### RETURN VALUE

1: Parity of the byte being tested matches the parity supplied as an argument.  
0: 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

<b>state</b>	An integer indicating what bit mode to use. It is best to use one of the macros provided for this:
<b>PARAM_7BIT</b>	Configures serial port to use seven bit data.
<b>PARAM_8BIT</b>	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.

<b>SERX_RTS_PORT</b>	Data register for the parallel port that the RTS line is on. e.g. PCDR
<b>SERA_RTS_SHADOW</b>	Shadow register for the RTS line's parallel port. e.g. PCDRShadow
<b>SERA_RTS_BIT</b>	The bit number for the RTS line
<b>SERA_CTS_PORT</b>	Data register for the parallel port that the CTS line is on
<b>SERA_CTS_BIT</b>	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  $\div$  8192.

### RETURN VALUE

- 1: The baud rate achieved on the Rabbit is the same as the input baud rate.
- 0: 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 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

<b>data</b>	Data structure to read from serial port X
<b>length</b>	Number of bytes to read
<b>tmout</b>	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

<b>data</b>	Data structure to write to serial port X.
<b>length</b>	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

<b>address</b>	Address of byte containing bits 7-0
<b>bit</b>	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

<b>address</b>	Address of byte containing bits 7-0
<b>bit</b>	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

<b>priority</b>	Priority the ISR should run at. Valid values are 1–3.
<b>isr</b>	ISR handler address. Must be a root address.

### RETURN VALUE

Address of vector table entry, or zero if **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

<b>vectNum</b>	Interrupt number: 0–15 are the only valid values.
<b>isr</b>	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

<b>x</b>	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

## SPIinit

```
void SPIinit ();
```

### DESCRIPTION

Initialize the SPI port parameters for a serial interface only. This function does nothing for a parallel interface. A description of the values that the user may define before the `#use SPI.LIB` statement is found at the top of the library `Lib\Spi\Spi.lib`.

### LIBRARY

`SPI.LIB`

### SEE ALSO

`SPIRead`, `SPIWrite`

## SPIRead

```
void SPIRead ( void *DestAddr, int ByteCount );
```

### DESCRIPTION

Reads a block of bytes from the SPI port. Note: the device Chip Select must already be active. The variable `SPIxor` needs to be set to either `0x00` or `0xFF` depending on whether or not the received signal needs to be inverted. Most applications will not need inversion. `SPIinit()` sets the value of `SPIxor` to `0x00`.

### PARAMETERS

<code>DestAddr</code>	Address to store the data
<code>ByteCount</code>	Number of bytes to read

### RETURN VALUE

None

### LIBRARY

`SPI.LIB`

### SEE ALSO

`SPIinit`, `SPIWrite`

## SPIWrite

```
int SPIWrite ( void *SrcAddr, int ByteCount );
```

### DESCRIPTION

Write a block of bytes to the SPI port. Note: the device Chip Select must already be active.

### PARAMETERS

<b>SrcAddr</b>	Address of data to write
<b>ByteCount</b>	Number of bytes to write

### RETURN VALUE

None

### LIBRARY

SPI.LIB

### SEE ALSO

SPIinit, SPIRead

## 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** hexadecimal 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

<b>buffer</b>	Result string of the formatted string.
<b>format</b>	String to be formatted.
<b>...</b>	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

<b>src</b>	String to be scanned.
<b>ch</b>	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

<b>str1</b>	Pointer to string 1.
<b>str2</b>	Pointer to string 2.

### RETURN VALUE

<0	if <b>str1</b> is less than <b>str2</b> char in <b>str1</b> is less than corresponding char in <b>str2</b> <b>str1</b> is shorter than but otherwise identical to <b>str2</b>
=0	<b>str1</b> is identical to <b>str2</b>
>0	if <b>str1</b> is greater than <b>str2</b> char in <b>str2</b> is greater than corresponding char in <b>str1</b> <b>str2</b> is shorter than but otherwise identical to <b>str1</b>

### 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

<b>str1</b>	Pointer to string 1.
<b>str2</b>	Pointer to string 2.

### RETURN VALUE

<0	if <b>str1</b> is less than <b>str2</b> char in <b>str1</b> is less than corresponding char in <b>str2</b> <b>str1</b> is shorter than but otherwise identical to <b>str2</b>
=0	<b>str1</b> is identical to <b>str2</b>
>0	if <b>str1</b> is greater than <b>str2</b> char in <b>str2</b> is greater than corresponding char in <b>str2</b> <b>str2</b> is shorter than but otherwise identical to <b>str1</b>

### 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

<b>dst</b>	Pointer to location to receive string.
<b>src</b>	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

<b>s1</b>	String to be scanned.
<b>s2</b>	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

<b>str1</b>	Pointer to string 1.
<b>str2</b>	Pointer to string 2.
<b>n</b>	Maximum number of bytes to compare. If zero, both strings are considered equal.

### RETURN VALUE

<0	if <b>str1</b> is less than <b>str2</b> char in <b>str1</b> is less than corresponding char in <b>str2</b>
=0	if <b>str1</b> is identical to <b>str2</b>
>0	if <b>str1</b> is greater than <b>str2</b> char in <b>str2</b> is greater than corresponding char in <b>str2</b>

### 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

<b>str1</b>	Pointer to string 1.
<b>str2</b>	Pointer to string 2.
<b>n</b>	Maximum number of bytes to compare, if zero then strings are considered equal

### RETURN VALUE

<0	if <b>str1</b> is less than <b>str2</b> char in <b>str1</b> is less than corresponding char in <b>str2</b>
=0	if <b>str1</b> is identical to <b>str2</b>
>0	if <b>str1</b> is greater than <b>str2</b> char in <b>str2</b> is greater than corresponding char in <b>str2</b>

### 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

<b>dst</b>	Pointer to location to receive string.
<b>src</b>	Pointer to location to supply string.
<b>n</b>	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

## strupbrk

```
char *strupbrk(char *s1, char *s2);
```

### DESCRIPTION

Scans a string for the first occurrence of any character from another string.

### PARAMETERS

<b>s1</b>	String to be scanned.
<b>s2</b>	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

<b>s</b>	String to be searched
<b>c</b>	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

<b>src</b>	String to be scanned
<b>brk</b>	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

<b>s1</b>	String to be scanned
<b>s2</b>	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

<b>s</b>	String to convert
<b>tailptr</b>	Pointer to a pointer of character. The next conversion may resume at the location specified by <b>*tailptr</b> .

### 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., delimiters found), replace the first delimiter in **src** with a **NULL** terminator so that **src** points to a proper **NULL**-terminated token.

### PARAMETERS

<b>src</b>	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.
<b>brk</b>	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, strtok

## strtol

```
long strtol(char *sptr, char **tailptr, int base);
```

### DESCRIPTION

ANSI string to long conversion.

### PARAMETERS

<b>sptr</b>	String to convert
<b>tailptr</b>	Assigned the last position of the conversion. The next conversion may resume at the location specified by <b>*tailptr</b> .
<b>base</b>	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 may 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

<b>x</b>	Value to compute
----------	------------------

### RETURN VALUE

Returns the tangent of **x**, where  $-8 \times \text{PI} \leq \mathbf{x} \leq +8 \times \text{PI}$ . 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  $90^\circ$  ( $\text{PI}/2$ ) 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 from **SEC\_TIMER** into the structure **t**. WARNING: The variable **SEC\_TIMER** is initialized when a program is 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: Success  
-1: Failure

### 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 32 kHz oscillator. Since the periodic interrupt is disabled when running at 32 kHz, these values will not be 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

<b>value</b>	16-bit number to convert
<b>buf</b>	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

<b>count</b>	$1 < \text{count} \leq 255$
--------------	-----------------------------

### 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

<b>ndog</b>	Id of virtual watchdog returned by <b>VdGetFreeWd()</b>
-------------	---

### LIBRARY

VDRIVER.LIB

## VdInit

```
void VdInit(void);
```

### DESCRIPTION

Initializes the Virtual Driver for all Rabbit boards. Supports **DelayMs()**, **DelaySec()**, **DelayTick()**. **VdInit()** is called by the BIOS unless it has been 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 **flashDst** address must be in the range 0x00040000-0x0007FFFF. 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.

### PARAMETERS

<b>flashDst</b>	Physical address of the flash destination
<b>rootSrc</b>	Pointer to the root source
<b>len</b>	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: Time out 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

<code>time</code>	32-bit value representing the number of seconds since January 1, 1980.
-------------------	--

### LIBRARY

`RTCLOCK.C`

### SEE ALSO

`read_rtc`

## writeUserBlock

```
int writeUserBlock(int addr, void *source, int numbytes)
```

### DESCRIPTION

Z-World boards are released with a System ID Blocks located on the primary flash. (See the *Rabbit 2000 Microprocessor Designer's Handbook*.) Version 2 and later of this ID block has a pointer to a User ID Block which can be used for storing calibration constants, passwords, and other non-volatile data. This block is protected from normal write to the flash device and can only be accessed through this function. This function writes a number of bytes from root memory to the user block.

NOTE: portions of the user block may be used by the BIOS for your board to store values such as calibration constants! See the manual for your particular board for more information before overwriting any part of the user block.

#### Backwards Compatibility:

If the version of the System ID block doesn't support the User ID block, or no System ID block is present, then 8K bytes starting 16K bytes from the top of the primary flash are designated the User ID block area. However, to prevent errors arising from incompatible large sector configurations, this will only work if the flash type is small sector. Z-World manufactured boards with large sector flash will have valid System and User ID blocks, so this should not be problem on Z-World boards.

If users create boards with large sector flash, they must install System ID blocks version 2 or greater to use or modify this function.

### PARAMETERS

<b>addr</b>	Address offset in user block to write to.
<b>source</b>	Pointer to source to copy data from.
<b>numbytes</b>	Number of bytes to copy.

### RETURN VALUE

0: Successful  
-1: Invalid address or range

### LIBRARY

IDBLOCK.LIB

### SEE ALSO

readUserBlock

## 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

<b>port</b>	Address of external data register.
<b>portshadow</b>	Reference pointer to a variable shadowing the register data. Substitute with <b>NULL</b> pointer (or 0) if shadowing is not required.
<b>data_value</b>	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

<b>port</b>	Address of data register.
<b>portshadow</b>	Reference pointer to a variable shadowing the register data. Substitute with <b>NULL</b> pointer (or 0) if shadowing is not required.
<b>data_value</b>	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: Success;  
0: Failure.

Note: This return value cannot be used with pointer arithmetic.

### LIBRARY

STACK.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

<b>dest</b>	Logical address
<b>src</b>	Physical address
<b>len</b>	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

<b>dest</b>	Physical address of destination
<b>src</b>	Physical address of source data
<b>len</b>	Length of source data in bytes

### RETURN VALUE

0: Success  
-1: Attempt to write Flash Memory area, nothing written

### LIBRARY

XMEM.LIB

# 16. Graphical User Interface

Dynamic C can be used to edit source files, compile and run programs, and choose options for these activities using pull-down menus or keyboard shortcuts. There are two modes: *edit mode* and *run mode*, which is also known as *debug mode*. Various debugging windows can be viewed in run mode. Programs can compile directly to a target controller for debugging in RAM or flash. Programs can also be compiled to a **.bin** file, with or without a controller connected to the PC. In order to run a program, a controller must be connected to the PC.

Multiple instances of Dynamic C can be run simultaneously. This means multiple debugging sessions are possible over different serial ports. This is useful for debugging boards that are communicating among themselves.

## 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.

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

### 16.1.0.2 Home

Moves the cursor backward in the text to the start of the line.

<b>Home</b>	Move to beginning of line
<b>CTRL-Home</b>	Move to beginning of file
<b>SHIFT-Home</b>	Select to beginning of line
<b>SHIFT-CTRL-Home</b>	Select to beginning of file

### 16.1.0.3 End

Moves the cursor *forward* in the text.

<b>End</b>	Move to end of line
<b>CTRL-End</b>	Move to end of file
<b>SHIFT-End</b>	Select to end of line
<b>SHIFT-CTRL-End</b>	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.

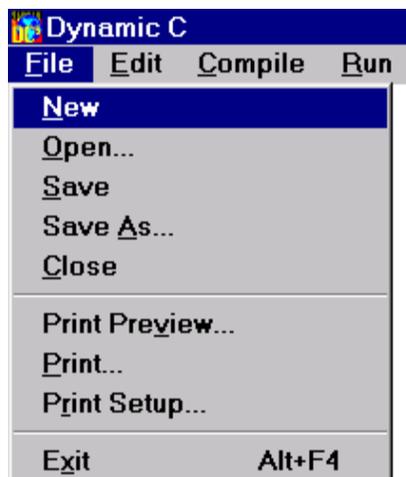
## 16.2 Menus



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.

### 16.2.1 File Menu

Click the menu title or press **<ALT-F>** to select the **FILE** menu. Prior to Dynamic C 8.x, there is a 10,000 line limit on the size of a single source file. If your source code is that big, split some of it up into libraries.



## New

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

## Open

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

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

## 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.

## Save As

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

## Close

Closes the active window. The active window may also be closed by pressing **<CTRL-F4>** or by double-clicking on its system menu. If there 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.

## 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.

## 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.

## 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.

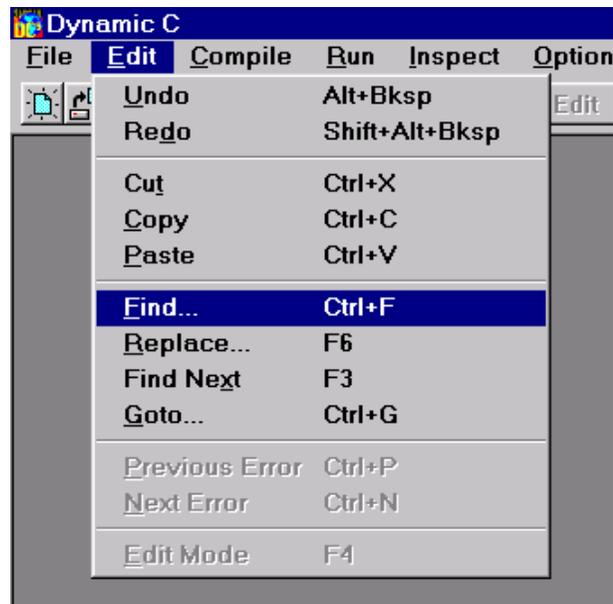
The **Options** button allows the print options dialog to be displayed for a specific printer. The **Network** button allows printers to be added or removed from the list of printers.

## 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>**.

### 16.2.2 Edit Menu

Click the menu title or press **<ALT-E>** to select the **EDIT** menu.



## 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>**.

## Redo

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

## Cut

Removes selected text from 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>**.

## 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>**.

## Paste

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

## Find

Finds specified text.

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 lower-case 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>**.

## 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>**.

## 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.

## Goto

Positions the insertion point at the start of the specified line.

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

## 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.

## 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.

## 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**.

## 16.2.3 Compile Menu

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



## Compile to Target

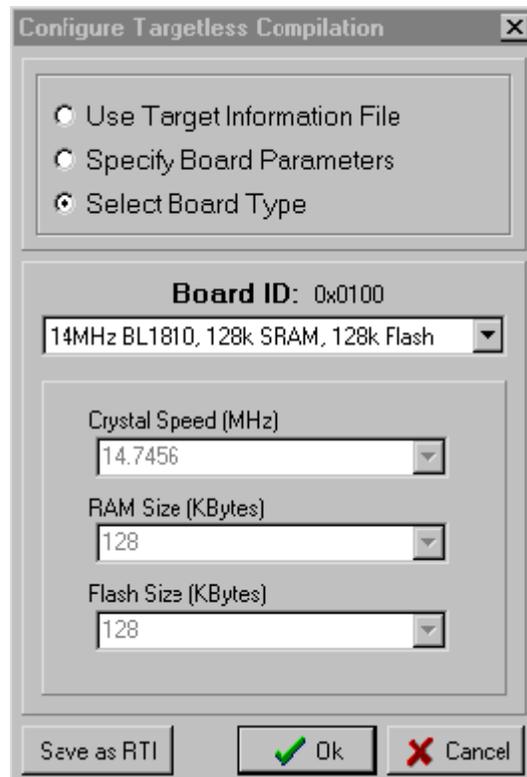
Compiles a program and loads it in the target controller's memory. The keyboard shortcut is **F5**.

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.

## 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. In most cases, the **Include BIOS** option is checked. This 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.

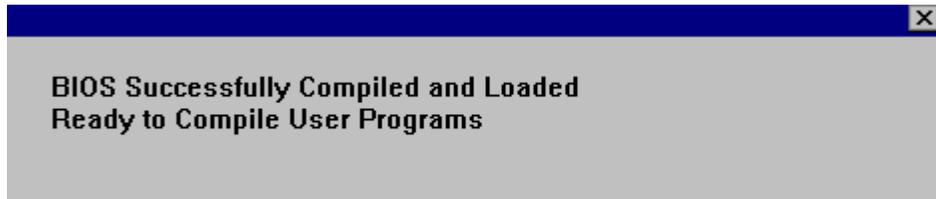
When compiling to a **.bin** file, choose **Use attached target** to use the parameters of the controller connected to your system. If there is no connected controller, or if there is but you want to define a different configuration, choose **Define target configuration**. The **Configure Targetless Compilation** dialog box will appear, as shown below. There are three options available in this dialog box for choosing the board parameters that will be used in the compile. **Select Board Type** is the default choice and activates the **Board ID** pull-down menu, a list of all known board configurations. **Specify Board Parameters**, when checked, brings up a dialog box to enter data for a new board configuration. The name specified in the dialog box for the new board configuration will be automatically included in the **Board ID** pull-down menu. **Use Target Information File**, when checked, will prompt for a Remote Target Information (RTI) file. Any target configuration can be saved as a **.rti** file by clicking the **Save as RTI** button at the bottom of the dialog box.



## 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.



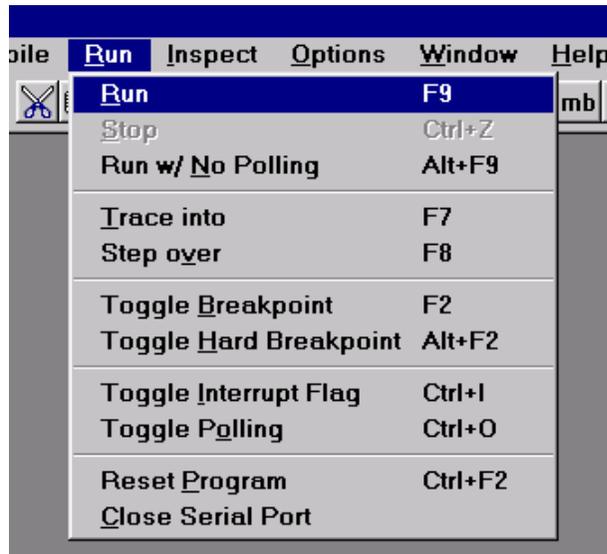
## Include Debug Code/RST 28 Instructions

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.

## 16.2.4 Run Menu

Click the menu title or press **<ALT-R>** to select the **RUN** menu.



### Run

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

### 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>**.

### 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>**.

## 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.

## 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**.

## Step over

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

## Toggle Breakpoint

Toggles a regular (“soft”) 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**.

## 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>**.

## Toggle Interrupt Flag

Toggles interrupt state. The keyboard shortcut is **<CTRL-I>**.

## Toggle Polling

Toggles polling mode. When running in polling mode (**F9**), the development PC polls or interrupts the target system every 100 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>**.

## Reset Target

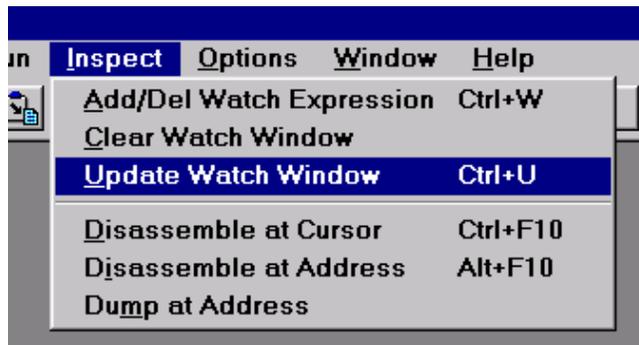
Tells the target system to perform a software reset including system initializations. Resetting a target *always* brings Dynamic C back to edit mode. The keyboard shortcut is **<CTRL-Y>**.

## Close Serial Port

Disconnects the programming serial port between PC and target so that the target serial port is accessible to other applications.

## 16.2.5 Inspect Menu

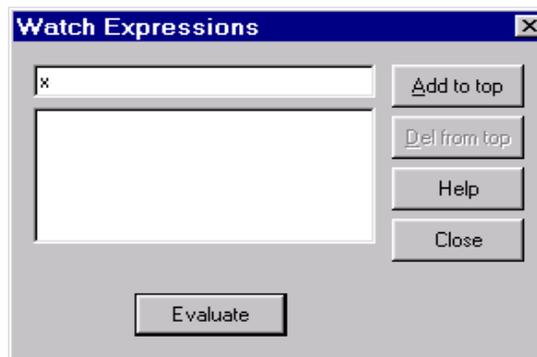
Click the menu title or press <ALT-I> to select the **INSPECT** menu.



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

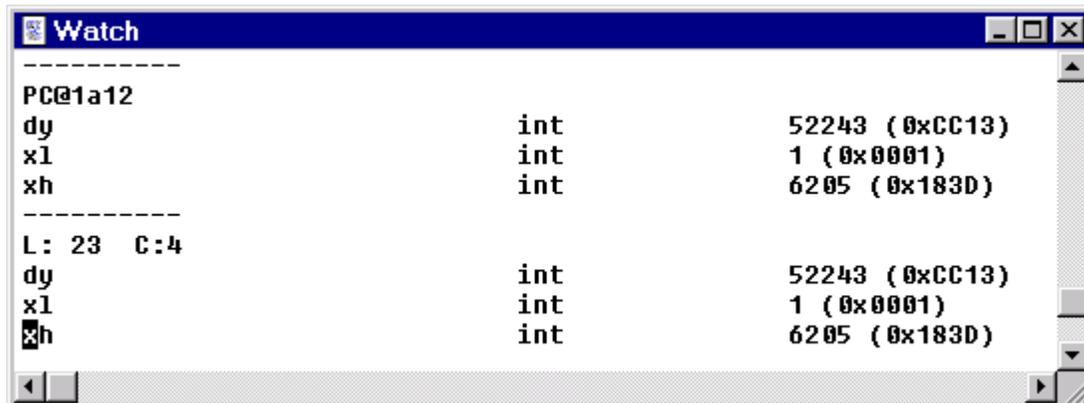
### Add/Del Watch Expression

This command provokes Dynamic C to display the following dialog.



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

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



### Clear Watch Window

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

### Update Watch Window

Forces expressions in the Watch Expression list to be evaluated and displayed in the Watch window 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>**.

### Disassemble at Cursor

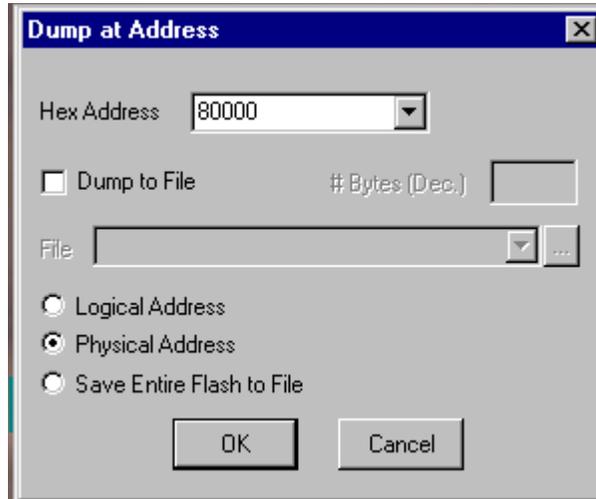
Loads, disassembles and displays the code at the current editor cursor. This command does not work in user application code declared as `nodebug`. Also, this command does not stop the execution on the target. The keyboard shortcut is **<CTRL-F10>**.

### 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>**.

## Dump at Address

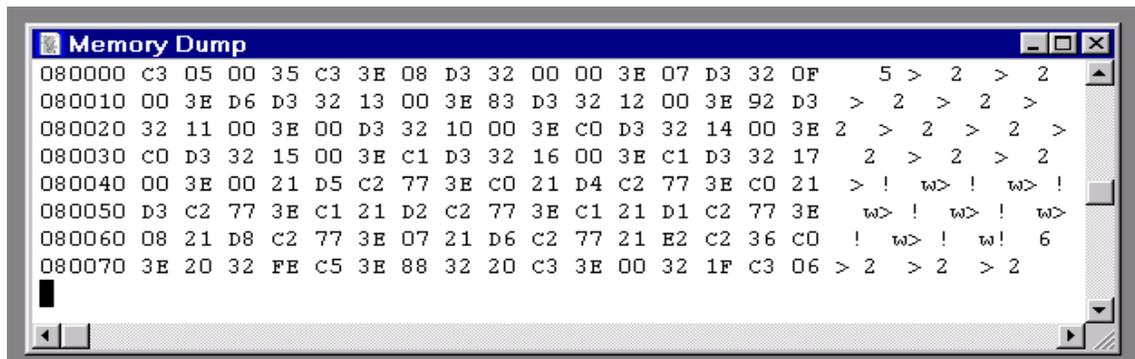
Allows blocks of raw values in any memory location (except the BIOS 0–2000H) to be looked at. Values can be displayed on the screen or written to a file.



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.

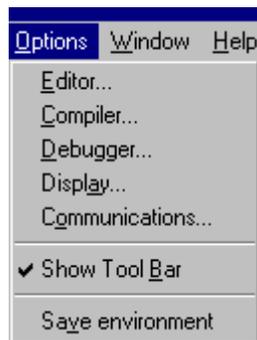
A typical screen display appears below.



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.

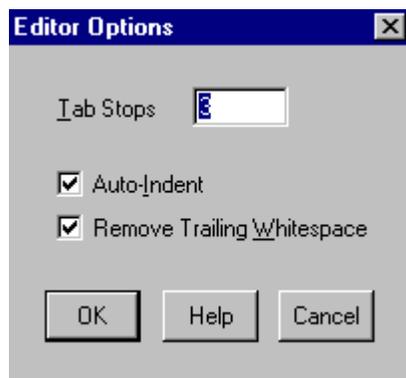
## 16.2.6 Options Menu

Click the menu title or press <ALT-O> to select the **OPTIONS** menu.



### 16.2.6.1 Editor

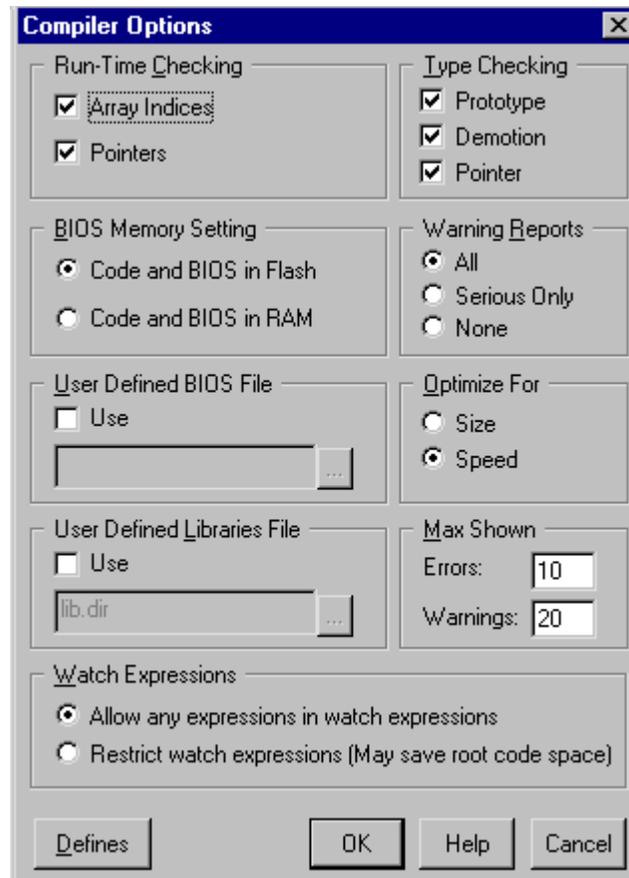
The **Editor** command gets Dynamic C to display the following dialog.



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.

### 16.2.6.2 Compiler

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



#### Run-Time Checking

These options, if checked, can allow a fatal error at run-time. They also increase the amount of code and cause slower execution, but they can be valuable debugging tools.

- **Array Indices**—Check array bounds. This feature adds code for every array reference.
- **Pointers**—Check for invalid pointer assignments. A pointer assignment is invalid if the code attempts to write to a location marked as not writable. Locations marked not writable include the *entire* root code segment. This feature adds code for every pointer reference.

#### 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.

### User Defined BIOS File

Use this option to change from the default BIOS to a user-specified file. Enter or select the file using the browse button/text box underneath this option. The check box labeled **use** must be selected or else the default file BIOS defined in the system registry will be used. Note that a single BIOS file can be made for compiling both to RAM and flash by using the preprocessor macros `_FLASH_` or `_RAM_`. These two macros are defined by the compiler based on the currently selected radio button in the **BIOS Memory Setting** group box.

### User Defined Libraries File

The Library Lookup information retrieved with Ctrl-H is parsed from the libraries found in the `lib.dir` file, which is part of the Dynamic C installation. Checking the **Use** box for **User Defined Libraries File**, allows the parsing of a user-defined replacement for `lib.dir` when Dynamic C starts. Library files must be listed in `lib.dir` (or its replacement) to be **#use'd** by a program.

If the function description headers are formatted correctly ( See “Function Description Headers” on page 39.), the functions in the libraries listed in the user-defined replacement for `lib.dir` will be available with Ctrl-H just like the user-callable functions that come with Dynamic C.

This is the same as the command line compiler `-LF` option.

### Watch Expressions

**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.

## Type Checking

This menu item allows the following choices:

- **Prototypes**—Performs strict type checking of arguments of function calls against the function prototype. The number of arguments passed must match the number of parameters in the prototype. In addition, the types of arguments must match those defined in the prototype. Z-World recommends prototype checking because it identifies likely run-time problems. To use this feature fully, all functions should have prototypes (including functions implemented in assembly).
- **Demotion**—Detects demotion. A demotion automatically converts the value of a larger or more complex type to the value of a smaller or less complex type. The increasing order of complexity of scalar types is:

```
char
unsigned int
int
unsigned long
long
float
```

A demotion deserves a warning because information may be lost in the conversion. For example, when a **long** variable whose value is 0x10000 is converted to an **int** value, the resulting value is 0. The high-order 16 bits are lost. An explicit type casting can eliminate demotion warnings. All demotion warnings are considered non-serious as far as warning reports are concerned.

- **Pointer**—Generates warnings if pointers to different types are intermixed without type casting. While type casting has no effect in straightforward pointer assignments of different types, type casting does affect pointer arithmetic and pointer dereferences. All pointer warnings are considered non-serious as far as warning reports are concerned.

## Warning Reports

This tells the compiler whether to report all warnings, no warnings or serious warnings only. It is advisable to let the compiler report all warnings because each warning is a potential run-time bug. Demotions (such as converting a **long** to an **int**) are considered non-serious with regard to warning reports.

## Optimize For

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

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

## Max Shown

This limits the number of error and warning messages displayed after compilation.

## Defines

The Defines button brings up another dialog box with a window for entering (or modifying) a list of defines that are global to any source file programs that are compiled and run. The syntax expected is a semi-colon separated list of defined constants with optional values given with an equal sign. This is the same as the command line compiler -d option, except that the CLC expects a single defined expression to follow each -d:

```
dccl_cmp mysourcefile.c -d DEF1 -d MAXN=10 -d DEF2
```

while the GUI window expects a semi-colon separated list

```
DEF1;MAXN=10;DEF2
```

The end result is the same as if every file compiled and run were prepended with:

```
#define DEF1
#define MAXN 10
#define DEF2
```

### 16.2.6.3 Debugger

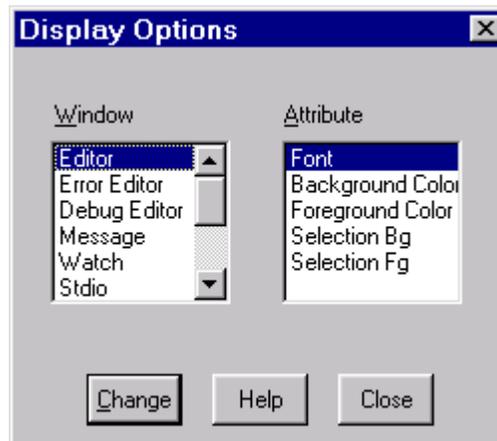
The **Debugger** command gets Dynamic C to display the following dialog.



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**.

#### 16.2.6.4 Display

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

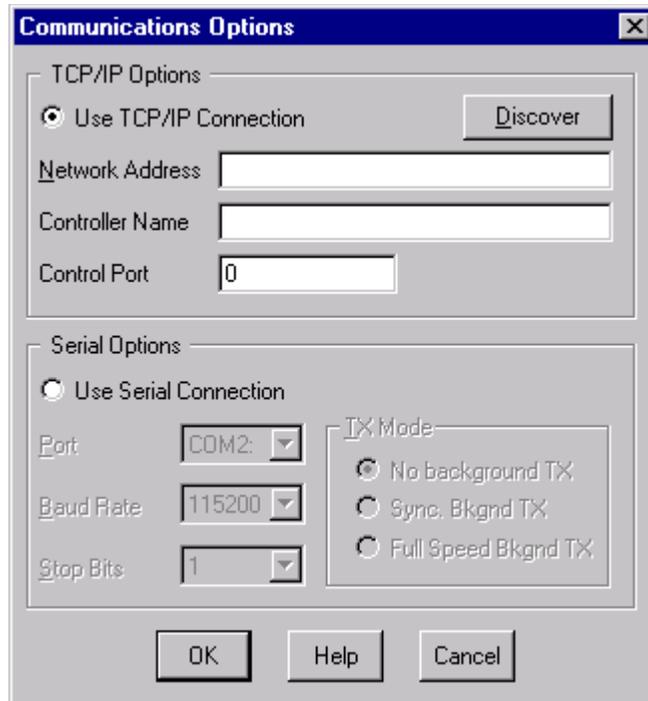


Use the **Display Options** dialog box to change the appearance of Dynamic C windows. First choose the window from the window list. Then select an attribute from the attribute list and click the change button. Another dialog box will appear to make the changes. Note that Dynamic C allows only fixed-pitch 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.

### 16.2.6.5 Communications

The **Communications** command displays the following dialog box. Use it to tell Dynamic C how to communicate with the target controller.



#### 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.

#### 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.

#### Show Tool Bar

The **Show Tool Bar** command toggles the display of the tool bar. Dynamic C remembers the tool-bar setting on exit.

## 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.

### 16.2.7 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.

### 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.

### 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.

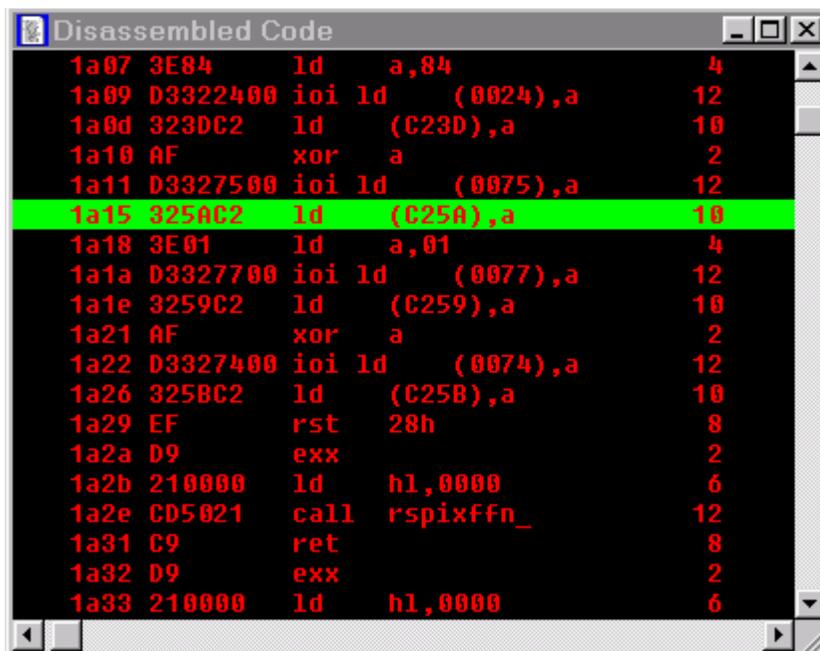
## Stdio

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.)

## 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.



```
Disassembled Code
1a07 3E84 ld a,84 4
1a09 D3322400 ioi ld (0024),a 12
1a0d 323DC2 ld (C23D),a 10
1a10 AF xor a 2
1a11 D3327500 ioi ld (0075),a 12
1a15 325AC2 ld (C25A),a 10
1a18 3E01 ld a,01 4
1a1a D3327700 ioi ld (0077),a 12
1a1e 3259C2 ld (C259),a 10
1a21 AF xor a 2
1a22 D3327400 ioi ld (0074),a 12
1a26 325BC2 ld (C25B),a 10
1a29 EF rst 28h 8
1a2a D9 exx 2
1a2b 210000 ld hl,0000 6
1a2e CD5021 call rspixffn_ 12
1a31 C9 ret 8
1a32 D9 exx 2
1a33 210000 ld hl,0000 6
```

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.

## 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.



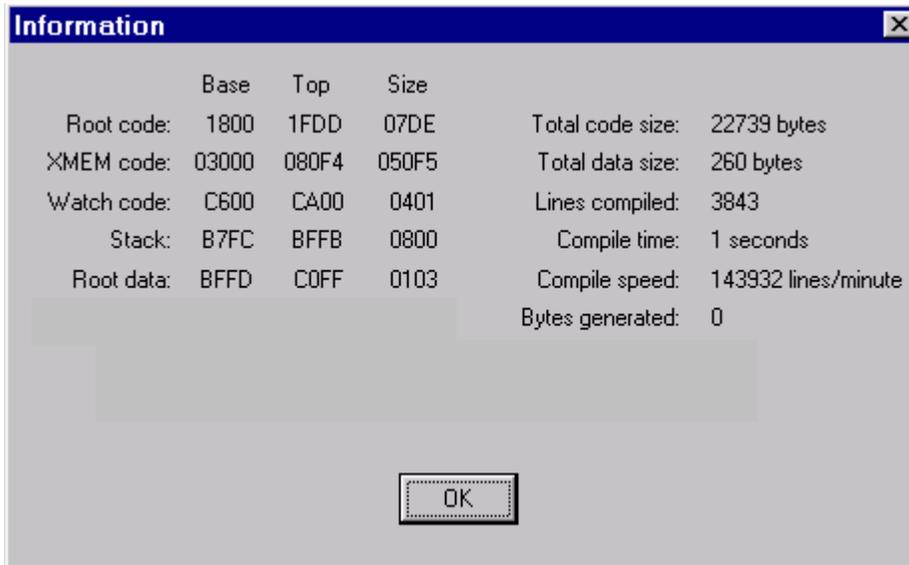
## Stack

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.



## 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.

### 16.2.8 Help Menu

Click the menu title or press **<ALT-H>** to select the **HELP** menu. The choices are given below:

#### Online Documentation

Opens a browser page and displays a file with links to other manuals. When installing Dynamic C from CD, this menu item points to the hard disk; after a Web upgrade of Dynamic C, this menu item optionally points to the Web.

#### Keywords

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

#### Operators

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

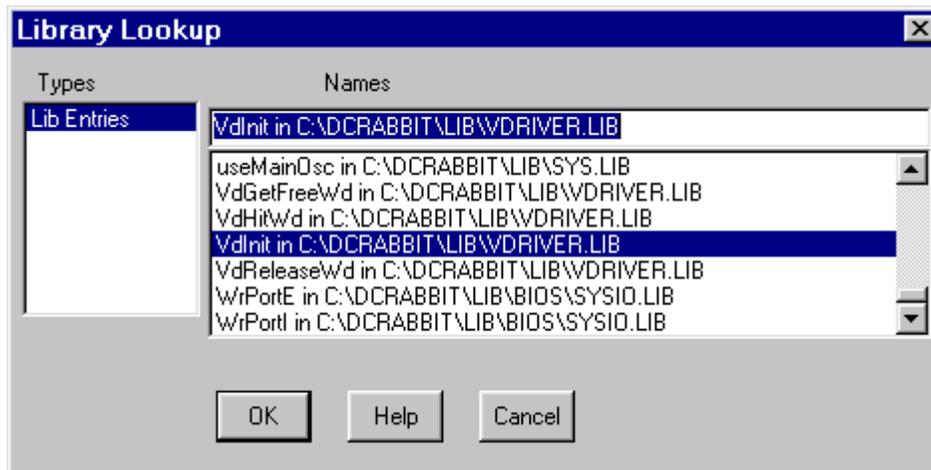
#### HTML Function Reference

Opens a browser page and displays an HTML file that has two links, one to Dynamic C functions listed alphabetically, the other to the functions listed by functional group. Each function listed is linked to its description in this manual.

## Function Lookup/Insert

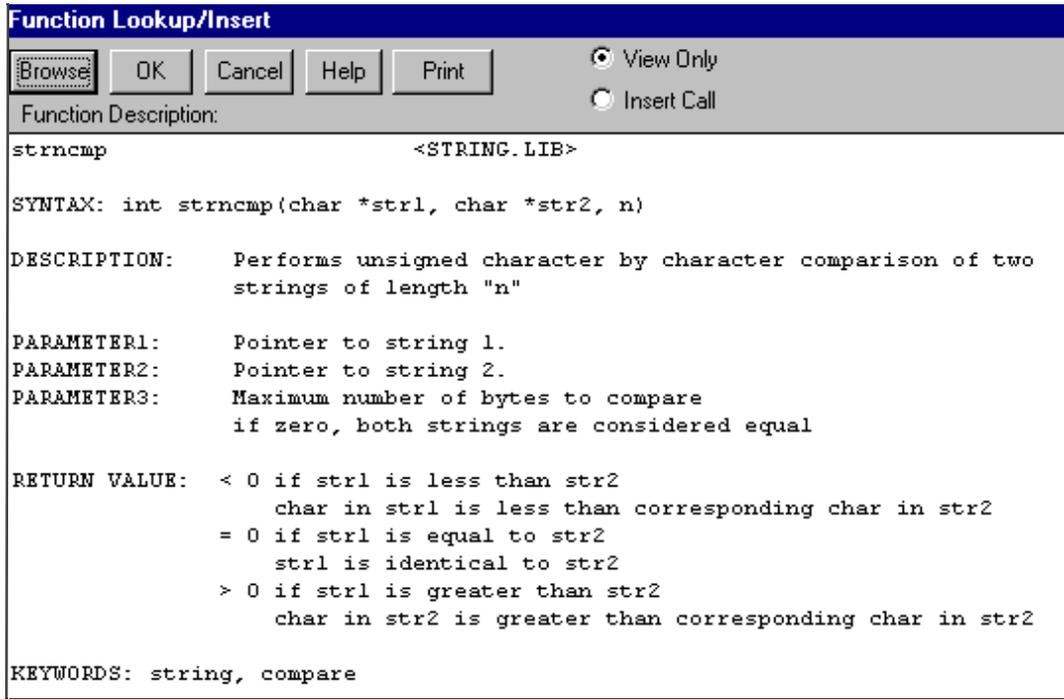
Obtains help information for library functions. When a function name is clicked (or the function name is selected) in source code and then the help command is issued, Dynamic C displays help information for that function. The keyboard shortcut is **<CTRL-H>**.

If Dynamic C cannot find a unique description for the function, it will display the following dialog box.

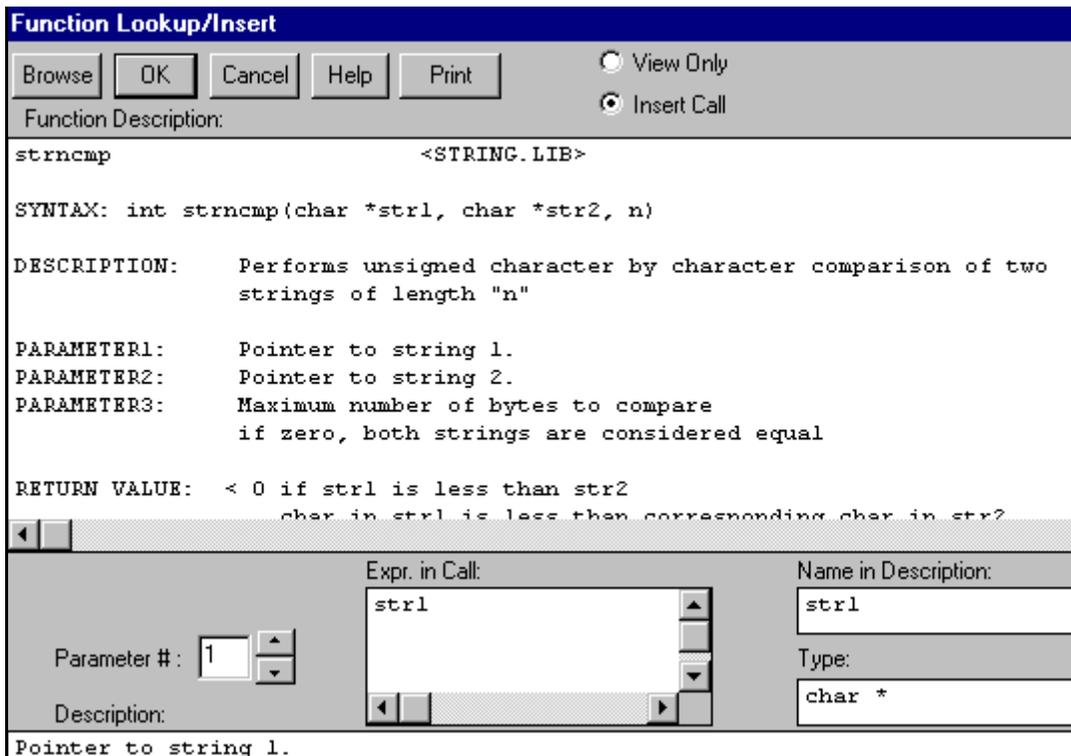


Click **Lib Entries** to display a list of the library functions currently available to the program. (These are the files named in the file **LIB.DIR**.) Then select a function name from the list to receive information about that function.

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.

## Instruction Set Reference

Invokes an on-line help system and displays the alphabetical list of instructions for the Rabbit 2000 Microprocessor.

## Keystrokes

Invokes an on-line help system and displays the keystrokes page. Although a mouse or other pointing device may be convenient, Dynamic C also supports operation entirely from the keyboard.

## Contents

Invokes an on-line help system and displays the contents page. From here view explanations of various features of Dynamic C.

## About

The **About** command displays the Dynamic C version number and the copyright notice.



# 17. Command Line Interface

The Dynamic C command line compiler, `dccl_cmp.exe`, performs the same compilation and program execution as its Graphical User Interface counterpart, but is invoked as a console application from a DOS window. It is called with a single source file program pathname as the first parameter, followed by optional, case insensitive switches that alter the default conditions under which the program is run. The results of the compilation and execution, all errors, warnings and program output, are directed to the console window and are optionally written or appended to a text file.

## 17.1 Default States

The default states of Dynamic C environment variables are used each time `dccl_cmp` is called. If a sequence of calls is written into a batch file, variations from the defaults must be repeated for each call. For instance, if a change is made to the serial parameters

```
dccl_cmp myProgram.c -s 2:115200:1:0
```

the next call will revert to the default settings of 1:115200:1:0 unless the switch is used with that next call as well.

## 17.2 User Input

Applications requiring user input must be called with the `-i` option:

```
dccl_cmp myProgram.c -i myProgramInputs.txt
```

where `myProgramInputs.txt` is a text file containing the inputs as separate lines, in the order in which `myProgram.c` expects them.

## 17.3 Saving Output to a File

The output consists of all program `printf`'s as well as all error and warning messages.

Output to a file can be accomplished with the `-o` option

```
dccl_cmp myProgram.c -i myProgramInputs.txt -o myOutputs.txt
```

where `myOutputs.txt` is overwritten if it exists or is created if it does not exist.

If the `-oa` option is used, `myOutputs.txt` is appended if it exists or is created if it does not.

## 17.4 Command Line Switches

Each switch must be separated from the others on the command line with at least one space or tab. Extra spaces or tabs are ignored. The parameter(s) required by some switches must be added as separate text immediately following the switch. Any of the parameters requiring a pathname, including the source file pathname, can have imbedded spaces by enclosing the pathname in quotes.

### 17.4.1 Switches Without Parameters

#### -h

Description: Print program header information.

Default: No header information will be printed.

GUI Equivalent: None.

Example: `dccl_cmp samples\demo1.c -h -o myoutputs.txt`

Header text preceding output of program:

```
*****
```

```
4/5/01 2:47:16 PM
```

```
dccl_cmp.exe, Version 7.05P - English
```

```
samples\demo1.c
```

```
Options: -h -o myoutputs.txt
```

```
Program outputs:
```

Note: Version information refers to `dcwd.exe` with the same compiler core.

#### -ri

Description: Disable runtime checking of array indices.

Default: Runtime checking of array indices is performed.

GUI Equivalent: Uncheck the **Options | Compiler | Array Indices** menu option.

#### -rp

Description: Disable runtime checking of pointers.

Default: Runtime checking of pointers is performed.

GUI Equivalent: Uncheck the **Options | Compiler | Pointers** menu option.

## **-wa**

Description: Report all warnings.

Default: All warnings reported.

GUI Equivalent: Select the **Options | Compiler | All** menu dialog box option.

## **-ws**

Description: Report only serious warnings.

Default: All warnings reported.

GUI Equivalent: Select the **Options | Compiler | Serious** menu dialog box option.

## **-wn**

Description: Report no warnings.

Default: All warnings reported.

GUI Equivalent: Select the **Options | Compiler | None** menu dialog box option.

## **-mr**

Description: Memory BIOS setting: RAM.

Default: Memory BIOS setting: Flash.

GUI Equivalent: Select the **Options | Compiler | Code and BIOS in RAM** menu dialog box option.

## **-mf**

Description: Memory BIOS setting: Flash.

Default: Memory BIOS setting: Flash.

GUI Equivalent: Select the **Options | Compiler | Code and BIOS in Flash** menu dialog box option.

## **-tt**

Description: Disable type checking of prototypes.

Default: Type checking of prototypes is performed.

GUI Equivalent: Uncheck the **Options | Compiler | Prototype** menu dialog box option.

## **-td**

Description: Disable type demotion checking.

Default: Type demotion checking is performed.

GUI Equivalent: Uncheck the **Options | Compiler | Demotion** menu dialog box option.

## **-tp**

Description: Disable type checking of pointers.

Default: Type checking of pointers is performed.

GUI Equivalent: Uncheck the **Options | Compiler | Pointer** menu dialog box option.

## **-sz**

Description: Optimize code generation for size.

Default: Optimize for speed.

GUI Equivalent: Select the **Options | Compiler | Size** menu dialog box option.

## **-sp**

Description: Optimize code generation for speed.

Default: Optimize for speed.

GUI Equivalent: Select the **Options | Compiler | Speed** menu dialog box option.

## **-rw**

- Description: Restrict watch expressions (May save root code space).
- Default: Allow any expressions in watch expressions.
- GUI Equivalent: Select the **Options | Compiler | Restrict watch expressions** menu dialog box option.

## **-b**

- Description: Compile to .bin file using attached target. The resulting file is created or overwritten with the same pathname as the source file, but with a .bin extension.
- Default: Compilation is written only to the target and not to a file.
- GUI Equivalent: Select the **Compile | Compile to .bin file | Use attached target** menu option.

## **-rd**

- Description: Do not include debug (RST 28) when compiling to a file. This option is ignored if not compiling to a file.
- Default: RST 28 is included if Compile to file is selected.
- GUI Equivalent: Uncheck the **Compile | Compile to .bin file | Include debug code/RST 28 instructions** menu option.

## **-rb**

- Description: Do not include BIOS when compiling to a file. This option is ignored if not compiling to a file.
- Default: BIOS is included if Compile to file is selected.
- GUI Equivalent: Uncheck the **Compile | Compile to .bin file | Include BIOS** menu option.

## 17.4.2 Switches Requiring a Parameter

### -ne maxNumberOfErrors

- Description: Change the maximum number of errors reported.
- Default: A maximum of 10 errors are reported.
- GUI Equivalent: Enter the maximum errors reported in the **Options | Compiler | Errors** menu dialog box option.
- Example: Allows up to 25 errors to be reported:  
`dccl_cmp myProgram.c -ne 25`

### -nw maxNumberOfWarnings

- Description: Change the maximum number of warnings reported.
- Default: A maximum of 10 warnings are reported.
- GUI Equivalent: Enter the maximum warnings reported in the **Options | Compiler | Warnings** menu dialog box option.
- Example: Allows up to 50 warnings to be reported:  
`dccl_cmp myProgram.c -nw 50`

### -s Port:Baud:Stopbits:BackgroundTx

- Description: Use serial transmission with parameters defined in a colon separated format of Port:Baud:Stopbits:BackgroundTx.
- Port: 1, 2, 3, 4, 5, 6, 7, 8
- Baud: 110, 150, 300, 600, 1200, 2400, 4800, 9600, 12800, 14400, 19200, 28800, 38400, 57600, 115200, 128000, 230400, 256000
- Stopbits: 1, 2
- BackgroundTx: 0: None, 1: Sync, 2: Full Speed
- Include all serial parameters in the prescribed format even if only one is being changed.
- Default: 1:115200:1:0
- GUI Equivalent: Select the **Options | Communications** Serial dialog box options.
- Example: Changing port from default of 1 to 2:  
`dccl_cmp myProgram.c -s 2:115200:1:0`

## **-t NetAddress:TcpName:TcpPort**

Description: Use TCP with parameters defined in a contiguous colon separated format of NetAddress:TcpName:TcpPort. Include all parameters even if only one is being changed.

netAddress: n.n.n.n  
tcpName: Text name of TCP port  
tcpPort: decimal number of TCP port

Default: None.

GUI Equivalent: Select the **Options | Communications** TCP dialog box options.

Example: `dccl_cmp myProgram.c -t 10.10.6.138:TCPName:4244`

## **-pw TCPPassPhrase**

Description: Enter the passphrase required for your TCP/IP connection. If no passphrase is required this option need not be used.

Default: No passphrase.

GUI Equivalent: Enter the passphrase required at the dialog prompt when compiling over a TCP/IP connection

Example: `dccl_cmp myProgram.c -pw "My passphrase"`

## **-rf RTIFilePathname**

Description: Compile to a .bin file using targetless compilation parameters found in RTI-FilePathname. The resulting compiled file will have the same pathname as the source (.c) file being compiled, but with a .bin extension.

Default: None.

GUI Equivalent: Select the **Compile | Compile to .bin file | Define target information | Use Target Information File** menu option.

Example: `dccl_cmp myProgram.c -rf MyTCparameters.rti`  
`dccl_cmp myProgram.c -rf "My Long Pathname\MyTCpa-rameters.rti"`

## **-rti BoardID:CpuID:CrystalSpeed:RAMSize:FlashSize**

Description: Compile to a .bin file using parameters defined in a colon separated format of BoardID:CpuID:CrystalSpeed:RAMSize:FlashSize. The resulting compiled file will have the same pathname as the source (.c) file being compiled, but with a .bin extension.

BoardID: Hex integer

CpuID: Decimal integer

CrystalSpeed: Decimal floating point, in MHz

RAMSize: Decimal, in KBytes

FlashSize: Decimal, in KBytes.

Default: None.

GUI Equivalent: Select the **Compile | Compile to .bin file | Define target information | Specify Board Parameters** menu option.

Example: `dccl_cmp myProgram.c -rti  
0x0101:2000:29.4912:128:256`

## **-bf BIOSFilePathname**

Description: Compile using a BIOS file found in BIOSFilePathname.

Default: Lib\BIOSLib\Biosfsm.lib.

GUI Equivalent: Select the **Options | Compiler | User Defined BIOS File | Use | ...** menu dialog box option.

Example: `dccl_cmp myProgram.c -bf MyPath\MyBIOS.lib`

## **-lf LibrariesFilePathname**

Description: Compile using a file found in LibrariesFilePathname which lists all libraries to be made available to your programs.

Default: Lib.dir.

GUI Equivalent: Select **Options | Compiler | User Defined Libraries File | Use | ...** from the menu dialog box.

Example `dccl_cmp myProgram.c -lf MyPath\MyLibs.txt`

## **-i InputsFilePathname**

Description: Execute a program that requires user input by supplying the input in a text file. Each input required should be entered into the text file exactly as it would be when entered into the Stdio Window in `dcwd.exe`. Extra input is ignored and missing input causes `dccl_cmp` to wait for keyboard input at the command line.

Default: None.

GUI Equivalent: Using `-i` is like entering inputs into the Stdio Window in `dcwd.exe`.

Example `dccl_cmp myProgram.c -i MyInputs.txt`

## **-o OutputFilePathname**

Description: Write header information (if specified with `-h`) and all program errors, warnings and outputs to a text file. If the text file does not exist it will be created, otherwise it will be overwritten.

Default: None.

GUI Equivalent: Select **Options | Debugger | Log STDOUT | Log file** menu dialog box option.

Example `dccl_cmp myProgram.c -o MyOutput.txt`  
`dccl_cmp myProgram.c -o MyOutput.txt -h`  
`dccl_cmp myProgram.c -h -o MyOutput.txt`

## **-oa OutputFilePathname**

Description: Append header information (if specified with `-h`) and all program errors, warnings and outputs to a text file. If the text file does not exist it will be created, otherwise it will be appended.

Default: None.

GUI Equivalent: Select the **Options | Debugger | Log STDOUT | Log file, Append Log** menu dialog box option.

Example `dccl_cmp myProgram.c -oa MyOutput.txt`

## -d MacroDefinition

Description: Defines a macro, optionally equating to a value. An optional value is given as the text following an equals sign (=). Only one parameter can be given after **-d** but the command line will interpret anything within quotes as a single parameter as in the last 2 examples. In the last example the parameter contains embedded quotes ("). Note that the command line processing only escapes the embedded quote and the newline character '\n' is kept as is. Also, the defined item and its optional value are trimmed of any leading and trailing spaces or tabs used in the command line.

Default: None.

GUI Equivalent: Uncheck the **Options | Compiler | Errors** menu option.

Example

```
dccl_cmp myProgram.c -d USEDEFAULTS
dccl_cmp myProgram.c -d MAXNUM=100
dccl_cmp myProgram.c -d " SUM = (A + B) "
dccl_cmp myProgram.c -d "FORMATSTR = \"Temp = %6.2f
degrees C\n\""
```

The above examples will be implemented as though the following were entered at the top of the source file:

```
#define USEDEFAULTS
#define MAXNUM 100
#define SUM (A + B)
#define FORMATSTR "Temp = %6.2f degrees C\n"
```

## 17.5 Examples

The following examples illustrate using multiple command line switches at the same time. If the switches on the command line are contradictory, such as `-mr` and `-mf`, the last switch (read left to right) will be used.

### 17.5.1 Example 1

In this example, `timer_b.c` is compiled to `timer_b.bin`. Any warnings or errors are written to `myoutputs.txt`.

```
dccl_smp samples\timerb\timer_b.c -o myoutputs.txt -b
```

### 17.5.2 Example 2

These examples will compile and run `myProgram.c` using different defines, displaying up to 50 warnings and capture all output to one file with a header for each run.

```
dccl_smp myProgram.c -d MAXCOUNT=99 -nw 50 -h -o myOutput.txt
dccl_smp myProgram.c -d MAXCOUNT=15 -nw 50 -h -oa myOutput.txt
dccl_smp myProgram.c -d MAXCOUNT=15 -d DEF1 -nw 50 -h -oa
myOutput.txt
```

The first run could have used the `-oa` option if `myOutput.txt` were known to not initially exist. `myProgram.c` presumably uses a constant `MAXCOUNT` and contains one or more compiler directives which react to whether `DEF1` is defined.



# 18. $\mu$ C/OS-II

## Not available with SE versions of Dynamic C.

$\mu$ C/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.  $\mu$ C/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  $\mu$ C/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  $\mu$ C/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](http://www.zworld.com/store/home.html), or at <http://www.ucos-ii.com/>.

## 18.1 Changes to $\mu$ C/OS-II

To take full advantage of services provided by Dynamic C, minor changes have been made to  $\mu$ C/OS-II.

### 18.1.1 Ticks per Second

In most implementations of  $\mu$ C/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  $\mu$ C/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  $\mu$ C/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();
```

## 18.1.2 Task Creation

In a  $\mu$ C/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  $\mu$ C/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  $\mu$ C/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.  $\mu$ C/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  $\mu$ C/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  $\mu$ C/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);
```

### 18.1.3 Restrictions

At the time of this writing,  $\mu$ C/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.

## 18.2 Tasking Aware Interrupt Service Routines (TA-ISR)

Special care must be taken when writing an interrupt service routine (ISR) that will be used in conjunction with  $\mu$ C/OS-II so that  $\mu$ C/OS-II scheduling will be performed at the proper time.

### 18.2.1 Interrupt Priority Levels

$\mu$ C/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  $\mu$ C/OS-II application should only disable interrupts by setting the interrupt priority level to 1, and should never raise the interrupt priority level above 1.

### 18.2.2 Possible ISR Scenarios

There are several different scenarios that must be considered when writing an ISR for use with  $\mu\text{C}/\text{OS-II}$ . Depending on the use of the ISR, it may or may not have to be 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.

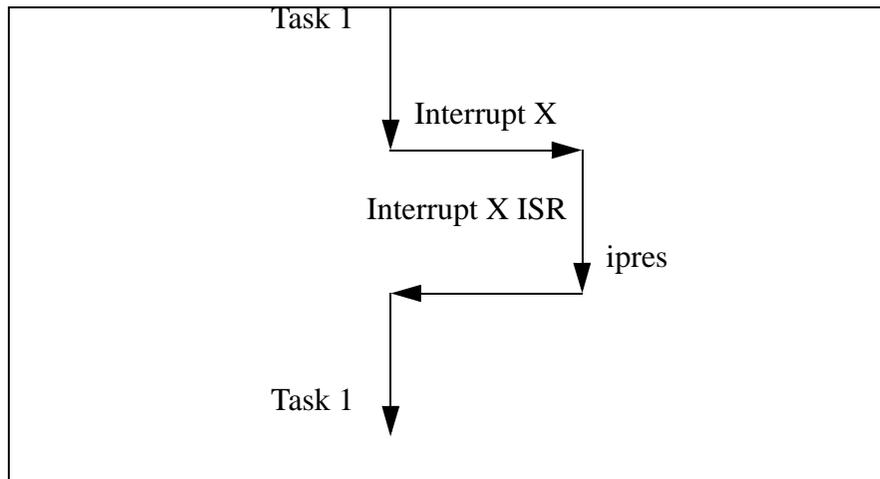


Figure 6. Type 1 ISR

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.

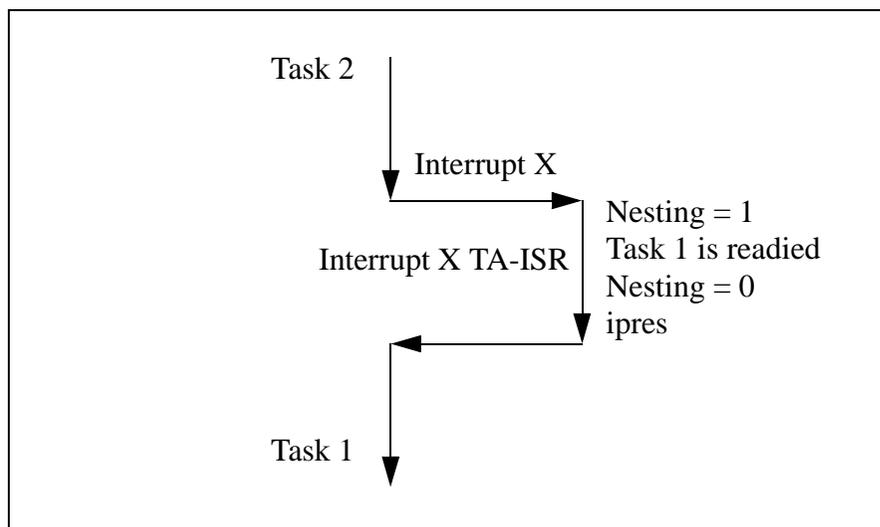
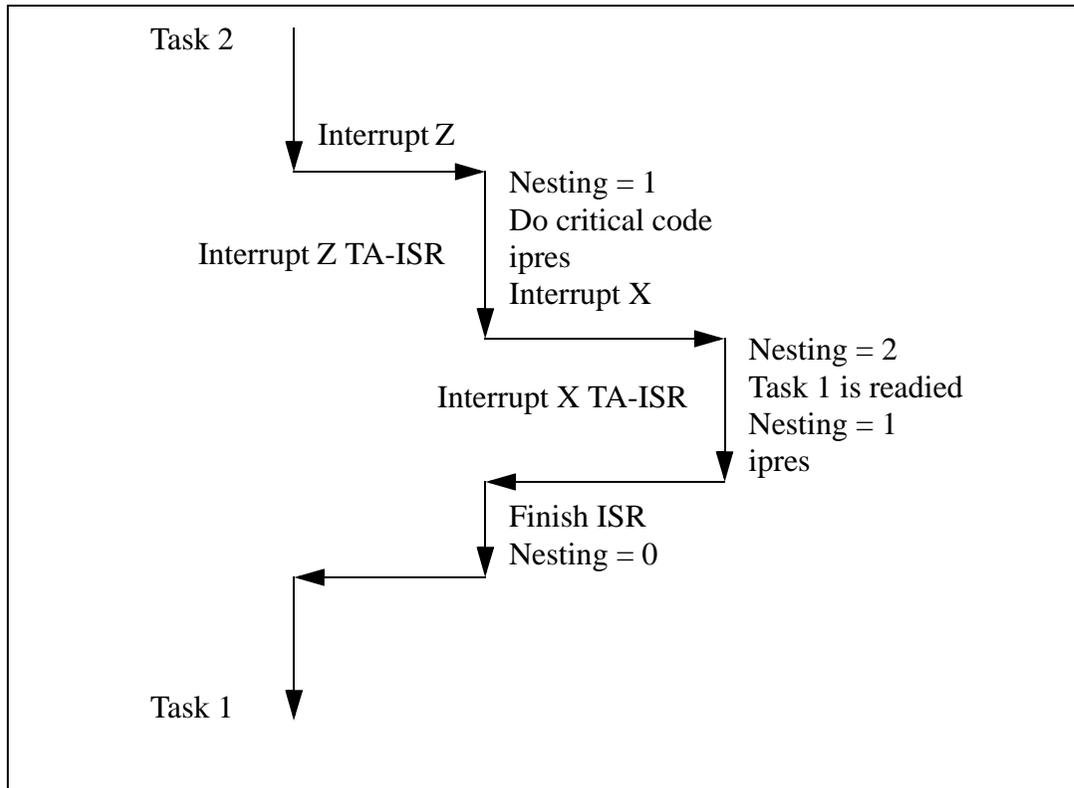


Figure 7. Type 2 ISR

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 8. Type 2 ISR Nested Inside Type 3 ISR**

As can be seen here, although the ISR for interrupt Z does not signal any tasks by posting to a semaphore, mailbox, or queue, it must increment and decrement the interrupt nesting count since it re-enables interrupts (`ipres`) prior to finishing all of its work.

### 18.2.3 General Layout of a TA-ISR

A TA-ISR is just like a standard ISR except that it does some extra checking and house-keeping. The following table summarizes when to use a TA-ISR.

**Table 15: Use of TA-ISR**

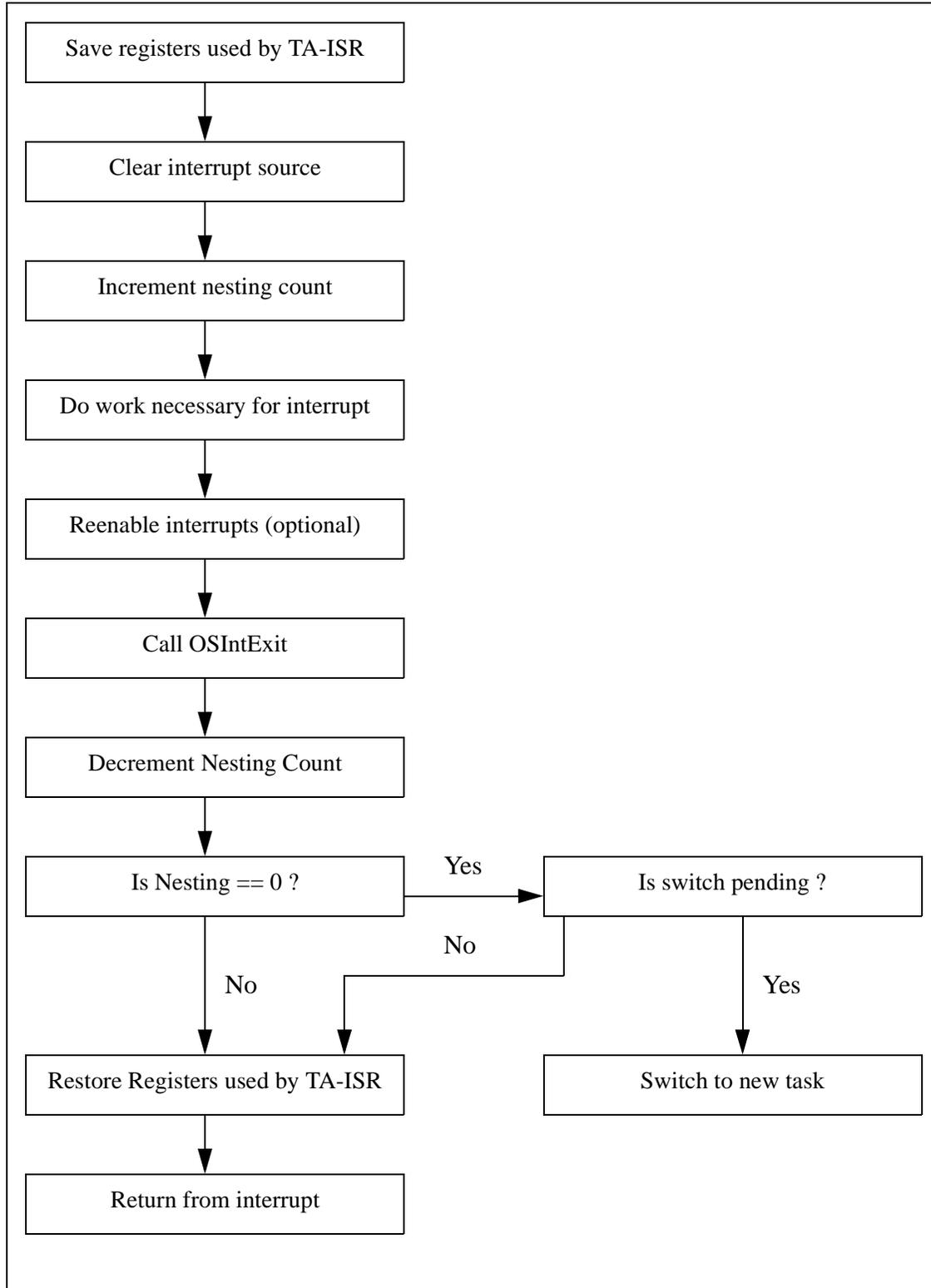
	μC/OS-II Application		
	Type 1*	Type 2†	Type 3‡
TA-ISR Required?	No	Yes	Yes

\*. 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 9. Logical Flow of a TA-ISR**

### 18.2.3.1 Sample Code for a TA-ISR

Fortunately, the Rabbit BIOS and libraries provide all of the necessary flags to make TA-ISRs work. With the code found in Listing 1, minimal work is needed to make a TA-ISR function correctly with  $\mu$ C/OS-II. TA-ISRs allow  $\mu$ C/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  $\mu$ C/OS-II has been fully initialized and started multi-tasking, a check should be made (5) to insure that  $\mu$ C/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  $\mu$ C/OS-II function determines the highest priority task ready to run, sets it as the currently running task, and sets the global flag `bios_swpnd` if a context switch needs to take place. Interrupts are disabled since a context switch is treated as a critical section (8). If the TA-ISR decrements the nesting counter and the count does not go to zero, then the nesting level is saved in `bios_intnesting` (9), the registers used by the TA-ISR are restored, interrupts are re-enabled (if not already done in (4)), and the TA-ISR returns (12). However, if decrementing the nesting counter in (9) causes the counter to become zero, then `bios_swpnd` must be checked to see if a context switch needs to occur (10). If a context switch is not pending, then the nesting level is set (9) and the TA-ISR exits (12). If a context switch is pending, then the remaining context of the previous task is saved and a long call, which insures that the `xpc` is saved and restored properly, is made to `bios_intexit` (11). `bios_intexit` is responsible for switching to the stack of the task that is now ready to run and executing a long call to switch to the new task. The remainder of (11) is executed when a previously preempted task is allowed to run again.

Listing 1

```
#asm
taskaware_isr::
    push    af                ;push regs needed by isr    (1)
    push    hl                ;clear interrupt source    (2)
    ld      hl,bios_intnesting ;increase the nesting count (3)
    inc     (hl)
    ; ipres (optional)                (4)
    ; do processing necessary for interrupt
    ld      a,(OSRunning)        ;MCOS multitasking yet?    (5)
    or      a
    jr      z,taisr_decnesting
    ; possibly signal task to become ready    (6)
    call    OSIntExit            ;sets bios_swpnd if higher
                                ; prio ready    (7)
```

```

taisr_decnesting:
    push    ip                                (8)
    ipset  1
    ld     hl,bios_intnesting                ;nesting counter == 1?
    dec   (hl)                               (9)
    jr    nz,taisr_noswitch

    ld     a,(bios_swpnd)                    ;switch pending?
    or    a                                    (10)
    jr    z,taisr_noswitch

    push  de                                (11)
    push  bc
    ex   af,af'
    push  af
    exx
    push  hl
    push  de
    push  bc
    push  iy
    lcall bios_intexit
    pop   iy
    pop   bc
    pop   de
    pop   hl
    exx
    pop   af
    ex   af,af'
    pop   bc
    pop   de

taisr_noswitch:
    pop   ip

taisr_done:
    pop   hl                                (12)
    pop   af
    ipres
    ret
#endasm

```

## 18.3 Library Reentrancy

When writing a  $\mu$ C/OS-II application, it is important to know which Dynamic C library functions are non-reentrant. If a function is non-reentrant, then only one task may access the function at a time, and access to the function should be controlled with a  $\mu$ C/OS-II semaphore. The following is a list of Dynamic C functions that are non-reentrant.

Library	Non-reentrant Functions
<b>MATH.LIB</b>	randg, randb, rand
<b>RS232.LIB</b>	All
<b>RTCLOCK.LIB</b>	write_rtc, tm_wr
<b>STDIO.LIB</b>	kbhit, getchar, gets, getswf, selectkey
<b>STRING.LIB</b>	atof*, atoi*, strtok
<b>SYS.LIB</b>	clockDoublerOn, clockDoublerOff, useMainOsc, useClockDivider, use32kHzOsc
<b>VDRIVER.LIB</b>	VdGetFreeWd, VdReleaseWd
<b>XMEM.LIB</b>	root2xmem, xmem2root, WriteFlash
<b>JRIO.LIB</b>	digOut, digOn, digOff, jrioInit, anaIn, anaOut, cof_anaIn
<b>JR485.LIB</b>	All

\*reentrant but sets the global `_xtoxErr` flag

The serial port functions (**RS232.LIB** functions) should be used in a restricted manner with  $\mu$ C/OS-II. Two tasks can use the same port as long as both are not reading, or both are not writing; i.e., one task can read from serial port X and another task can write to serial port X at the same time without conflict.

## 18.4 How to Get a $\mu$ C/OS-II Application Running

$\mu$ C/OS-II is a highly configurable, 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  $\mu$ C/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  $\mu$ C/OS-II or has a  $\mu$ C/OS-II reference ([MicroC/OS-II, The Real Time Kernel](#) by Jean J. Labrosse is highly recommended).

### 18.4.1 Default Configuration

$\mu$ C/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`.  $\mu$ C/OS-II for the Rabbit uses a more intelligent stack allocation scheme than other  $\mu$ C/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
#define OS_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
```

```

///  

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 stacks (task stacks + initial program stack)  

#define STACK_CNT_512 OS_MAX_TASKS+1

```

If a particular portion of  $\mu$ C/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  $\mu$ C/OS-II based on the needs of each application.

## 18.4.2 Custom Configuration

In order to customize  $\mu$ C/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  $\mu$ C/OS-II and stack configuration constants with a **#use "ucos2.lib"** statement. This ensures that the definitions supplied outside of the library are used, rather than the defaults in the library.

This configuration uses 20 tasks, two semaphores, up to 15 memory partitions that the memory manager will control, and makes use of the statistics task. Note that the configuration constants for task creation, task deletion, and semaphores are not defined, as the library defaults will suffice. Also note that 10 of the application tasks will each have a 1024 byte stack, 10 will each have a 2048 byte stack, and an extra stack is declared for the statistics task.

### 18.4.3 Examples

The following sample programs demonstrate the use of the default configuration supplied in `UCOS2.LIB` and a custom configuration which overrides the defaults.

#### Example 1

In this application, ten tasks are created and one semaphore is created. Each task pends on the semaphore, gets a random number, posts to the semaphore, displays its random number, and finally delays itself for three seconds.

Looking at the code for this short application, there are several things to note. First, since  $\mu\text{C}/\text{OS-II}$  and slice statements are mutually exclusive (both rely on the periodic interrupt for a “heart-beat”), `#use "ucos2.lib"` must be included in every  $\mu\text{C}/\text{OS-II}$  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  $\mu\text{C}/\text{OS-II}$  function to ensure that the operating system is properly initialized (3). Before  $\mu\text{C}/\text{OS-II}$  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  $\mu\text{C}/\text{OS-II}$  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,  $\mu\text{C}/\text{OS-II}$  will run indefinitely.

```

// 1. Explicitly use uC/OS-II library
#include "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  $\mu$ C/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
#include "ucos2.lib"

void RandomNumberTask(void *pdata);

// Declare semaphore global so all tasks have access
OS_EVENT* RandomSem;

void main(){
    int i;

    // Initialize OS internals
    OSInit();

    for(i = 0; i < OS_MAX_TASKS; i++){
        // Create each of the system tasks
        OSTaskCreate(RandomNumberTask, NULL, 1024, i);
    }
    // semaphore to control access to random number generator
    RandomSem = OSSemCreate(1);

    // 4. Set number of system ticks per second
    OSSetTicksPerSec(OS_TICKS_PER_SEC);

    // Begin multi-tasking
    OSStart();
}
```

```

void RandomNumberTask(void *pdata)
{
    // Declare as auto to ensure reentrancy.
    auto OS_TCB data;
    auto INT8U err;
    auto INT16U RNum;

    OSTaskQuery(OS_PRIO_SELF, &data);
    while(1)
    {
        // Rand is not reentrant, so access must be controlled
        // via a semaphore.
        OSSemPend(RandomSem, 0, &err);
        RNum = (int)(rand() * 100);
        OSSemPost(RandomSem);

        printf("Task%02d's random #: %d\n",data.OSTCBPrio,RNum);
        // Wait 3 seconds in order to view output from each task.
        OSTimeDlySec(3);
    }
}

```

## 18.5 Compatibility with TCP/IP

The TCP/IP stack is reentrant and may be used with the  $\mu$ C/OS real-time kernel. The line

```
#use ucos2.lib
```

must appear before the line

```
#use dcrtcp.lib.
```



# Appendix A. Macros and Global Variables

This is not a comprehensive list of the macros and global variables available in Dynamic C. This appendix contains many macros and global variables the user may want to know about.

## A.1 Compiler-Defined Macros

Macro Name	Definition and Default
<code>_BOARD_TYPE_</code>	This is read from the System ID block or defaulted to 0x100 (the BL1810 JackRabbit board) if no System ID block is present. This can be used for conditional compilation based on board type. Board types are listed in <code>default.h</code> .
<code>CC_VER</code>	Gives the Dynamic C version in hex, ie. version 7.05 is 0x0705.
<code>DC_CRC_PTR</code>	Reserved.
<code>DEBUG_RST</code>	In the <b>Compile</b> pull-down menu, check “Include Debug Code/RST 28 Instructions” to set <code>DEBUG_RST</code> to <b>1</b> . Debug code will be included even if <code>#nodebug</code> precedes the main function in the program.
<code>_FLASH_</code>	These are used for conditional compilation of the BIOS to distinguish between compiling to RAM and compiling to flash. These are set in the <b>Options   Compiler</b> menu.
<code>_RAM_</code>	
<code>_FLASH_SIZE_</code>	These are used to set the MMU registers and code and data sizes available to the compiler. The values given by these macros represent the number of 0x1000 blocks of memory available.
<code>_RAM_SIZE_</code>	
<code>NO_BIOS</code>	Boolean value. Tells the compiler whether or not to include the BIOS when compiling to a .bin file. This is set in the <b>Compile</b> menu
<code>_SECTOR_SIZE_</code>	Prior to Dynamic C 7.02, this macro (near the top of <code>LIB\BIOSLIB\FIRMWARE.LIB</code> ) needs to be hard-coded to the sector size of the first flash in bytes.
<code>_TARGETLESS_COMPILE_</code>	Boolean value. This is set in the <b>Compile</b> menu. It defaults to 0.
<code>_USE115KBAUD_</code>	Boolean value. Tells BIOS to use 115k baud if value is 1. The baud rate can be changed in the <b>Options   Communications</b> menu.

Macro Name	Definition and Default
<b>USE_2NDFLASH_CODE</b>	Uncomment this macro at the top of the BIOS to allow compilation of a program into two flash chips. The macro is commented out by default.

## A.2 Global Variables

### **dc\_timestamp**

The number of seconds that have passed since 00:00:00 January 1, 1980, Greenwich Mean Time (GMT) adjusted by the current time zone and daylight savings of the PC on which the program was compiled. The recorded time indicates when the program finished compiling.

```
printf("The date and time: %lx\n", dc_timestamp);
```

### **OPMODE**

This is a char. It can have the following values:

- 0x88 = debug mode
- 0x80 = run mode

### **SEC\_TIMER**

This unsigned long variable is initialized to the value of the real-time clock (RTC). If the RTC is set correctly, this is the number of seconds that have elapsed since the reference date of January 1, 1980. The periodic interrupt updates **SEC\_TIMER** every second. This variable is initialized by the Virtual Driver when a program starts.

### **MS\_TIMER**

This unsigned long variable is initialized to zero. The periodic interrupt updates **MS\_TIMER** every millisecond. This variable is initialized by the Virtual Driver when a program starts.

### **TICK\_TIMER**

This unsigned long variable is initialized to zero. The periodic interrupt updates **TICK\_TIMER** 1024 times per second. This variable is initialized by the Virtual Driver when a program starts.

## A.3 Exception Types

These macros are defined in `errors.lib`:

```
#define ERR_BADPOINTER          228
#define ERR_BADARRAYINDEX      229
#define ERR_DOMAIN              234
#define ERR_RANGE               235
#define ERR_FLOATOVERFLOW      236
#define ERR_LONGDIVBYZERO      237
#define ERR_LONGZEROMODULUS    238
#define ERR_BADPARAMETER       239
#define ERR_INTDIVBYZERO       240
#define ERR_UNEXPECTEDINTRPT   241
#define ERR_CORRUPTEDCODATA    243
#define ERR_VIRTWDOGTIMEOUT    244
#define ERR_BADXALLOC          245
#define ERR_BADSTACKALLOC      246
#define ERR_BADSTACKDEALLOC    247
#define ERR_BADXALLOCINIT      249
#define ERR_NOVIRTWDOGAVAIL    250
#define ERR_INVALIDMACADDR     251
#define ERR_INVALIDCOFUNC      252
```

## A.4 Rabbit 2000 Internal registers

Macros are defined for all of the Rabbit's I/O registers. A listing of these register macros can be found in the *Rabbit 2000 Microprocessor User's Manual*.

### A.4.1 Shadow Registers

Shadow registers exist for many of the I/O registers. They are character variables defined in the BIOS. The naming convention for shadow registers is to append the word Shadow to the name of the register. For example, the global control status register, GCSR, has a corresponding shadow register named GCSRShadow.



# Appendix B. Map File Generation

Starting with Dynamic C 7.05, all symbol information is put into a single file. The map file has three sections: a memory map section, a function section, and a globals section.

The map file format is designed to be easy to read, but with parsing in mind for use in program down-loaders and in other possible future utilities (for example, an independent debugger). Also, the memory map, as defined by the **#org** statements, will be saved into the map file.

Map files are generated in the same directory as the file that is compiled. If compilation is not successful, the contents of the map file cannot be considered reliable.

## B.1 Grammar

<mapfile>: <memmap section> <function section> <global section>

<memmap section>: <memmapreg>+

<memmapreg>: <register var> = <8-bit const>

<register var>: **XPC|SEGSIZE|DATASEG**

<function section>: <function description>+

<function description>: <identifier> <address> <size>

<address>: <logical address> | <physical address>

<logical address>: <16-bit constant>

<physical address>: <8-bit constant>:<16-bit constant>

<size>: <20-bit constant>

<global section>: <global description>+

<global description>: <scoped name> <address>

<scoped name>: <global>| <local static>

<global>: <identifier>

<local static>: <identifier>:<identifier>

Comments are C++ style (*//* only).



## 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-World's intellectual property rights than the terms and conditions of this License. (iv) use and distribute with Qualified Applications and Qualified Systems the program files distributed with Dynamic C named **RFU.EXE**, **PILOT.BIN**, and **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-WORLD'S TOTAL LIABILITY TO YOU FOR ALL DAMAGES, LOSSES, AND CAUSES OF ACTION (WHETHER IN CONTRACT, TORT, INCLUDING NEGLIGENCE, OR OTHERWISE) EXCEED THE AMOUNT PAID BY YOU FOR THE SOFTWARE.

8. Termination. This License is effective for the duration of the copyright in the Software unless terminated. You may terminate this License at any time by destroying all copies of the Software and its documentation. This License will terminate immediately without notice from 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, 113, 114, 146  
#class .....146  
#debug .....89, 137, 146  
#define .....16, 17, 18, 146  
#elif .....148  
#else .....148  
#endasm .....113, 116, 146  
#endif .....148  
#error .....147  
#fatal .....146  
#funcchain .....32, 147  
#if .....148  
#ifdef .....148  
#ifndef .....148  
#include  
    absence of .....35  
#interleave .....148  
#KILL .....148  
#makechain .....32, 148  
#mmap .....4, 149  
#nodebug .....89, 137, 146, 404  
#nointerleave .....148  
#noseix .....149  
#undef .....18, 149  
#use .....35, 38, 149  
#useix .....149  
#warns .....149  
#warnt .....149  
#ximport .....149  
& (address operator) .....26  
\* (indirection operator) .....26  
@RETVAl .....124  
@SP .....119, 122, 123, 124, 127  
\_GLOBAL\_INIT .....139  
{ } curly braces .....21

## A

abort .....129  
About Dynamic C .....423  
abstract data types .....23  
adc (add-with-carry) .....113  
Add to Top button .....407  
Add/Del Items <CTRL-W> 408, 417  
Add/Del Watch Expression  
    <CTRL-W> .....407  
adding watch window items .....  
    407, 408

address operator (&) .....26  
address space .....4  
addresses .....97  
addresses in assembly language  
    116  
aggregate data types .....24  
ALT key .....398  
ALT-Backspace .....400  
ALT-C .....402  
ALT-CTRL-F3 .....402  
ALT-F10 .....408  
ALT-F2 .....405, 406  
ALT-F4 .....400  
ALT-F9 .....405  
ALT-H .....420  
ALT-O .....410  
ALT-R .....405  
ALT-SHIFT-backspace .....400  
ALT-W .....417  
always\_on .....129  
anymem .....129  
argument passing ..28, 118, 119,  
    124, 125  
    modifying value .....28  
arrange icons  
    command .....417  
arrays .....24, 25, 28  
    characters .....19  
    subscripts .....24  
arrow keys .....397, 398  
Assembly Language .....113  
assembly language .....3, 38, 89,  
    114, 115, 116, 121, 123,  
    124, 125, 126, 127, 406  
    embedding C statements ..114  
assembly window ...3, 121, 417,  
    418  
assignment operators .....155  
associativity .....151  
auto 90, 116, 117, 118, 119, 130  
Auto Open STDIO Window 414

## B

backslash  
    continuation in directives .146  
backslash (\)  
    character literals .....16, 20  
basic unit of a C program .....22  
baud rate .....416  
BCDE .....117, 123, 125  
BeginHeader .....37, 38  
binary operators .....151  
BIOS .....133  
body

    module .....37, 38, 39  
branching .....31, 32  
break .....30, 32, 130, 142  
    example .....30  
break points ...89, 121, 137, 406,  
    408  
    hard .....405, 406  
    interrupt status .....405, 406  
    soft .....405, 406  
breaking out of a loop .....30  
breaking out of a switch state-  
    ment .....30  
buttons, toolbar .....416

## C

C functions calling assembly  
    code .....123  
C language 3, 4, 5, 13, 19, 23, 28,  
    32, 114, 117  
C statements embedded in as-  
    sembly code .....114  
C variables in assembly language  
    116  
cascaded windows .....417  
case .....32, 130, 133  
case-sensitive searching .....401,  
    402  
char .....23, 130, 144  
characters  
    arrays .....19  
    embedded quotes .....20  
    nonprinting values .....20  
    special values .....20  
checking  
    pointers .....27  
    stack .....89, 90  
    type .....22  
Clear Watch Window .....408  
clipboard .....401  
Close <CTRL-F4> .....399  
closing a file .....399  
CoData Structure .....46  
    pointer to .....48  
Cofunctions .....50  
    abandon .....54  
    calling restrictions .....51  
    everytime .....54  
    indexed .....52  
    single user .....52  
    syntax .....50  
COM port .....416  
communication  
    serial .....416  
compilation .....402, 417, 420

direct to controller .....	3	const .....	131	storage class .....	5
errors .....	402	Contents		Del from Top button .....	407
speed .....	3	Help .....	423	deleting watch window items ...	407, 408
targetless .....	402	continue .....	30, 132, 142	demotion .....	413
Compile		example .....	30	direct	
to flash .....	402	copying text <CTRL-C> ...	400, 401	compilation .....	3
to RAM .....	402	costate .....	132	directives .....	4
to Target .....	402	Costatements .....	44	#asm .....	89, 113, 114
COMPILE menu .....	402	syntax .....	45	#debug .....	89, 137, 146
Compile to File <CTRL-F3> ...	402	costatements 129, 132, 143, 145		#define .....	16, 17, 18
402		Create *.RTI File for Targetless		#endasm .....	113, 116
Compile to File with *.RTI File		Compile .....	402	#funcchain .....	32
<ALT-CTRL-F3> .....	402	CTRL key .....	397	#makechain .....	32
Compile to Target <F3> .....	402	CTRL-F10 .....	408	#nodebug .....	89, 137, 404
compiler directives .....	146	CTRL-F2 .....	406	#undef .....	18
#asm .....	146	CTRL-F3 .....	402	#use .....	35, 38
#class .....	146	CTRL-G .....	402	Disassemble at Address <ALT-	
#debug .....	146	CTRL-H .....	421, 422, 423	F10> .....	408, 418
#define .....	146	CTRL-I .....	405, 406	Disassemble at Cursor <CTRL-	
#elif .....	148	CTRL-N .....	402	F10> .....	408, 418
#else .....	148	CTRL-O .....	405, 406	disassembled code .....	407
#endasm .....	146	CTRL-P .....	402	display	
#endif .....	148	CTRL-U .....	408	options .....	410, 415
#error .....	147	CTRL-V .....	401	do loop .....	29
#fatal .....	146	CTRL-W .....	408	dot operator .....	15, 25
#funcchain .....	147	CTRL-X .....	401	dump window .....	409
#GLOBAL_INIT .....	147	CTRL-Y .....	405, 406	dw .....	115
#if .....	148	CTRL-Z .....	405	dynamic	
#ifdef .....	148	curly braces { } .....	21	storage allocation .....	25
#endif .....	148	cursor		Dynamic C .....	3
#interleave .....	148	execution .....	406	differences .....	4, 5, 32
#KILL .....	148	positioning .....	402	exit .....	400, 417
#makechain .....	148	text .....	423	installation .....	5, 127
#mmap .....	149	cutting text <CTRL-X> .....	401	support files .....	36
#nodebug .....	146				
#nointerleave .....	148	<b>D</b>		<b>E</b>	
#nouseix .....	149	data types .....	24	EDIT menu .....	400, 401, 402
#undef .....	149	aggregate .....	24	Edit Menu .....	400
#use .....	149	primitive .....	14	edit mode .....	397, 402, 406
#useix .....	149	DATASEG .....	97	editing .....	3
#warns .....	149	db .....	115	editor .....	3
#warn .....	149	DCW.CFG .....	417	options .....	410
#ximport .....	149	DCW.INI .....	417	else .....	133
line continuation .....	146	debug .....	132, 146	embedded assembly code 3, 118,	
Compiler options .. 27, 410, 411,		editor .....	415	123, 124, 125, 126, 127	
413		mode .....	90, 402, 405	embedded quotes .....	20
compiling .....	3	debugger .....	3	End key .....	397
to file .....	397, 402	options .....	410, 414	EndHeader .....	37, 38
to RAM .....	402	debugging . 3, 89, 146, 405, 406,		EPROM .....	4, 5
to ROM .....	402	408, 409		equ .....	116
to target .....	397, 402	assembly-level view .....	3	errors	
compound		declarations .....	21, 37	editor .....	415
names .....	15	default .....	32, 133	error code ranges .....	91
statements .....	21				

locating .....	402		
run-time .....	91		
ESC key			
to close menu .....	398		
Evaluate button .....	407		
examples			
break .....	30		
continue .....	30		
for loop .....	29		
modules .....	38		
of array .....	24		
union .....	25		
execution .....	405, 408		
cursor .....	406		
Exit <ALT-F4> .....	400		
Expr. in Call .....	423		
extended memory .....	4, 123, 145		
extern .....	38, 133		
<b>F</b>			
F (status register) .....	419		
F10 .....	417		
F2 .....	405, 406		
F3 .....	402		
F4 .....	402		
F6 .....	401		
F7 .....	405, 406		
F8 .....	405, 406		
F9 .....	405		
file commands			
close file .....	399		
create file .....	399		
open file .....	399		
save file .....	399		
FILE menu .....	399, 400		
file size .....	398		
Filesystem			
metadata .....	105		
Find next <SHIFT-F5> .....	402		
firsttime .....	134		
float .....	23, 134, 144		
values .....	19		
for .....	21, 135		
character literals .....	20		
loop .....	29		
example .....	29		
frame			
reference point .....	124		
reference pointer .....	90, 122, 123, 137		
free space .....	420		
Full Speed Bkgnd TX .....	416		
function calls .....	22, 28, 90, 118, 119, 123, 124, 125, 127,		
130, 423			
function chains .....	32, 33, 139, 148		
function groups			
arithmetic			
abs .....	173		
getcrc .....	257		
bit manipulation			
BIT .....	184		
bit .....	183		
RES .....	343		
res .....	342		
SET .....	358		
set .....	358		
character			
isalnum .....	266		
isalpha .....	267		
iscntrl .....	267		
isdigit .....	269		
isgraph .....	269		
islower .....	270		
isprint .....	271		
ispunct .....	272		
isspace .....	270		
isupper .....	273		
isxdigit .....	273		
data encryption			
AESdecrypt .....	175		
AESdecryptStream .....	175		
AESencrypt .....	176		
AESencryptStream .....	176		
AESexpandKey .....	177		
AESinitStream .....	178		
error handling			
errlogFormatEntry .....	207		
errlogFormatRegDump .....	208		
errlogFormatStackDump .....	208		
errlogGetHeaderInfo .....	206		
errlogGetMessage .....	209		
errlogGetNthEntry .....	207		
errlogReadHeader .....	209		
exception .....	210		
ResetErrorLog .....	343		
extended memory			
paddr .....	325		
root2xmem .....	344		
WriteFlash2 .....	390		
xalloc .....	394		
xmem2root .....	395		
xmem2xmem .....	396		
fast fourier transforms			
fftcplx .....	218		
fftcplxinv .....	219		
fftrealm .....	220		
fftrealmv .....	221		
hanncplx .....	261		
hannreal .....	262		
powerspectrum .....	333		
file system			
fclose .....	212		
fcreate .....	212		
fcreate_unused .....	214		
fdelete .....	215		
fopen_rd .....	230		
fopen_wr .....	231		
fread .....	233		
fs_format .....	236		
fs_init .....	238		
fs_reserve_blocks .....	240		
fsck .....	240		
fseek .....	241		
fshift .....	254		
ftell .....	252		
fwrite .....	254		
floating-point math			
acos .....	173		
acot .....	174		
acsc .....	174		
asec .....	179		
asin .....	179		
atan .....	180		
atan2 .....	181		
ceil .....	188		
cos .....	200		
cosh .....	201		
deg .....	203		
exp .....	211		
fabs .....	211		
floor .....	229		
fmod .....	229		
frexp .....	235		
labs .....	279		
ldexp .....	280		
log .....	281		
log10 .....	281		
modf .....	291		
poly .....	331		
pow .....	332		
pow10 .....	332		
rad .....	337		
rand .....	338		
randb .....	338		
randg .....	339		
sin .....	361		
sinh .....	362		
sqrt .....	366		
tan .....	380		
tanh .....	381		

GPS		
gps_get_position .....	259	
gps_get_utc .....	260	
gps_ground_distance ...	260	
I/O		
BitRdPortE .....	184	
BitRdPortI .....	185	
BitWrPortE .....	186	
BitWrPortI .....	187	
RdPortE .....	339	
RdPortI .....	340	
WrPortE .....	393	
WrPortI .....	393	
I2C protocol		
i2c_check_ack .....	274	
i2c_init .....	275	
i2c_read_char .....	275	
i2c_send_ack .....	276	
i2c_send_nak .....	276	
i2c_start_tx .....	277	
i2c_startw_tx .....	277	
i2c_stop_tx .....	278	
i2c_write_char .....	278	
interrupts		
GetVectExtern2000 .....	258	
GetVectIntern .....	259	
SetVectExtern2000 .....	360	
SetVectIntern .....	361	
low-level flash access		
flash_erasechip .....	222	
flash_erasesector .....	222	
flash_gettype .....	223	
flash_init .....	224	
flash_read .....	225	
flash_readsector .....	226	
flash_sector2xwindow ..	227	
flash_writesector .....	228	
MicroC/OS-II		
OSInit .....	291	
OSMboxAccept .....	292	
OSMboxCreate .....	292	
OSMboxPend .....	293	
OSMboxPost .....	294	
OSMboxQuery .....	295	
OSMemCreate .....	296	
OSMemGet .....	297	
OSMemPut .....	297	
OSMemQuery .....	298	
OSQAccept .....	298	
OSQCreate .....	299	
OSQFlush .....	299	
OSQPend .....	300	
OSQPost .....	301	
OSQPostFront .....	302	
OSQQuery .....	303	
OSSchedLock .....	303	
OSSchedUnlock .....	304	
OSSemAccept .....	304	
OSSemCreate .....	305	
OSSemPend .....	305	
OSSemPost .....	306	
OSSemQuery .....	307	
OSSetTickPerSec .....	308	
OSStart .....	308	
OSStatInit .....	309	
OSTaskChangePrio .....	309	
OSTaskCreate .....	310	
OSTaskCreateExt .....	311	
OSTaskCreateHook .....	312	
OSTaskDel .....	313	
OSTaskDelHook .....	313	
OSTaskDelReq .....	314	
OSTaskQuery .....	315	
OSTaskResume .....	316	
OSTaskStatHook .....	316	
OSTaskStkChk .....	317	
OSTaskSuspend .....	318	
OSTaskSwHook .....	318	
OSTimeDly .....	319	
OSTimeDlyHMSM .....	320	
OSTimeDlyResume .....	321	
OSTimeDlySec .....	322	
OSTimeGet .....	322	
OSTimeSet .....	323	
OSTimeTickHook .....	323	
OSVersion .....	324	
miscellaneous		
longjmp .....	282	
qsort .....	336	
runwatch .....	344	
setjmp .....	359	
multitasking		
CoBegin .....	191	
CoPause .....	199	
CoReset .....	199	
CoResume .....	200	
DelayMs .....	203	
DelaySec .....	204	
DelayTicks .....	204	
IntervalMs .....	264	
IntervalSec .....	264	
IntervalTick .....	265	
isCoDone .....	268	
isCoRunning .....	268	
number-to-string conversion		
ftoa .....	256	
htoa .....	263	
itoa .....	274	
ltoa .....	283	
ltoan .....	284	
utoa .....	387	
real-time clock		
mktime .....	289	
mktm .....	290	
read_rtc .....	340	
read_rtc_32kHz .....	341	
tm_rd .....	382	
tm_wr .....	383	
write_rtc .....	391	
serial communication		
cof_serAgetc .....	193	
cof_serAgets .....	194	
cof_serAputc .....	195	
cof_serAputs .....	196	
cof_serAread .....	197	
cof_serAwrite .....	198	
cof_serBgetc .....	193	
cof_serBgets .....	194	
cof_serBputc .....	195	
cof_serBputs .....	196	
cof_serBread .....	197	
cof_serBwrite .....	198	
cof_serCgetc .....	193	
cof_serCgets .....	194	
cof_serCputc .....	195	
cof_serCputs .....	196	
cof_serCread .....	197	
cof_serCwrite .....	198	
cof_serDgetc .....	193	
cof_serDgets .....	194	
cof_serDputc .....	195	
cof_serDputs .....	196	
cof_serDread .....	197	
cof_serDwrite .....	198	
serAclose .....	345	
serAdatabits .....	346	
serAflowcontrolOff .....	346	
serAflowcontrolOn .....	347	
serAgetc .....	348	
serAgetError .....	349	
serAopen .....	350	
serAparity .....	351	
serApeek .....	352	
serAputc .....	352	
serAputs .....	353	
serArdFlush .....	353	
serArdFree .....	354	
serArdUsed .....	354	
serAread .....	355	
serAwrFlush .....	356	
serAwrFree .....	356	
serAwrite .....	357	

serBclose .....	345
serBdatabits .....	346
serBflowcontrolOff .....	346
serBflowcontrolOn .....	347
serBgetc .....	348
serBgetError .....	349
serBopen .....	350
serBparity .....	351
serBpeek .....	352
serBputc .....	352
serBputs .....	353
serBrdFlush .....	353
serBrdFree .....	354
serBrdUsed .....	354
serBread .....	355
serBwrFlush .....	356
serBwrFree .....	356
serBwrite .....	357
serCclose .....	345
serCdatabits .....	346
serCflowcontrolOff .....	346
serCflowcontrolOn .....	347
serCgetc .....	348
serCgetError .....	349
serCheckParity .....	345
serCopen .....	350
serCparity .....	351
serCpeek .....	352
serCputc .....	352
serCputs .....	353
serCrdFlush .....	353
serCrdFree .....	354
serCrdUsed .....	354
serCread .....	355
serCwrFlush .....	356
serCwrFree .....	356
serCwrite .....	357
serDclose .....	345
serDdatabits .....	346
serDflowcontrolOff .....	346
serDflowcontrolOn .....	347
serDgetc .....	348
serDgetError .....	349
serDopen .....	350
serDparity .....	351
serDpeek .....	352
serDputc .....	352
serDputs .....	353
serDrdFlush .....	353
serDrdFree .....	354
serDrdUsed .....	354
serDread .....	355
serDwrFlush .....	356
serDwrFree .....	356
serDwrite .....	357
serial packet driver	
cof_pktXreceive .....	191
cof_pktXsend .....	192
pktXgetErrors .....	326
pktXreceive .....	329
pktXsend .....	330
pktXsending .....	330
pktXsetParity .....	331
STDIO	
getchar .....	256
gets .....	258
kbhit .....	279
outchr .....	324
outstr .....	325
printf .....	334
putchar .....	335
puts .....	335
sprintf .....	365
string manipulation	
memchr .....	286
memcmp .....	287
memcpy .....	288
memmove .....	288
memset .....	289
strcat .....	366
strchr .....	367
strcmp .....	368
strncmpi .....	369
strcpy .....	370
strcspn .....	370
strlen .....	371
strncat .....	371
strncmp .....	372
strncmpi .....	373
strncpy .....	374
strpbrk .....	375
strchr .....	375
strspn .....	376
strstr .....	376
strtok .....	378
tolower .....	384
toupper .....	384
string-to-number conversion	
atof .....	182
atoi .....	182
atol .....	183
strtod .....	377
strtoul .....	379
system	
_sysIsSoftReset .....	379
chkHardReset .....	188
chkSoftReset .....	189
chkWDTO .....	189
clockDoublerOff .....	190
clockDoublerOn .....	190
defineErrorHandler .....	202
exit .....	210
forceSoftReset .....	233
GetVectExtern2000 .....	258
ipres .....	265
ipset .....	266
premain .....	334
sysResetChain .....	380
updateTimers .....	385
use32HzOsc .....	385
useClockDivider .....	386
useMainOsc .....	386
watchdog	
Disable_HW_WDT .....	205
hitwd .....	263
VdGetFreeWd .....	387
VdInit .....	388
VdReleaseWd .....	389
function headers .....	39
function help .....	39
function libraries .....	3, 35, 37
function lookup <CTRL-H> .....	421, 422, 423
function returns .....	90, 123, 124, 125
functions .....	22
entry and exit .....	90
prototypes .....	22, 24, 37
<b>G</b>	
Global Initialization .....	33
global variables .....	25
goto .....	30, 31, 135
Goto <CTRL-G> .....	402
<b>H</b>	
hard break points .....	405, 406
header	
function .....	39
module .....	37, 38
heap storage .....	420
HELP menu .....	420, 421, 422, 423
HL .....	117, 122, 123, 125
Home key .....	397
horizontal tiling .....	417
<b>I</b>	
IBM PC .....	3, 405, 416
icons	
arranged .....	417
IEEE floating point .....	134

if ..... 133  
 multichoice ..... 31  
 simple ..... 31  
 with else ..... 31  
 indirection operator (\*) ..... 26  
 information window .... 417, 420  
 init\_on ..... 136  
 insertion point ..... 401, 402  
 INSPECT menu .. 407, 408, 409,  
 417  
 installation  
 Dynamic C ..... 5, 127  
 Instruction Set Reference .... 423  
 int ..... 23, 136, 144  
 integers ..... 19  
 hexadecimal ..... 19  
 long ..... 19  
 octal ..... 19  
 unsigned ..... 19  
 interrupt ..... 136  
 interrupt service routines 3, 126,  
 127, 136  
 interrupt status  
 and break points ..... 405, 406  
 interrupts ..... 126, 127  
 flag ..... 406  
 latency ..... 126  
 IX (index register) .. 89, 90, 122,  
 123, 137, 143

## K

kernel  
 real-time ..... 90  
 key module ..... 37  
 keystrokes  
 <ALT R> select RUN menu .  
 405  
 <ALT-Backspace> undoing  
 changes ..... 400  
 <ALT-C> select COMPILE  
 menu ..... 402  
 <ALT-F> select FILE menu ..  
 398  
 <ALT-F10> Disassemble at  
 Address ..... 408  
 <ALT-F2> Toggle hard break  
 point ..... 405, 406  
 <ALT-F4> Exit ..... 400  
 <ALT-F4> Quitting Dynamic  
 C ..... 398  
 <ALT-F9> Run w/ No Polling  
 405  
 <ALT-H> select HELP menu  
 420

<ALT-O> select OPTIONS  
 menu ..... 410  
 <ALT-SHIFT-backspace> re-  
 doing changes ..... 400  
 <ALT-W> select WINDOW  
 menu ..... 417  
 <CTRL-F> Compile to File ..  
 402  
 <CTRL-F10> Disassemble at  
 Cursor ..... 408  
 <CTRL-F2> Reset Program ..  
 405, 406  
 <CTRL-F3> Compile to File  
 with \*.RTI File ..... 402  
 <CTRL-G> Goto ..... 402  
 <CTRL-H> Library Help  
 lookup . 398, 421, 422, 423  
 <CTRL-I> Toggle interrupt ..  
 405, 406  
 <CTRL-N> next error ..... 402  
 <CTRL-O> Toggle polling ...  
 405, 406  
 <CTRL-P> previous error 402  
 <CTRL-U> Update Watch  
 window ..... 408  
 <CTRL-V> pasting text .. 401  
 <CTRL-W> Add/Del Items ..  
 408  
 <CTRL-X> cutting text ... 401  
 <CTRL-Y> Reset target . 405,  
 406  
 <CTRL-Z> Stop ..... 405  
 <F10> Assembly window 417  
 <F2> Toggle break point 405,  
 406  
 <F3> Compile to Target .. 402  
 <F7> Trace into ..... 405, 406  
 <F8> Step over ..... 405, 406  
 <F9> Run ..... 405  
 <SHIFT-F5> Find next ... 402  
 keywords .... 4, 32, 89, 123, 129,  
 133, 137, 139, 143  
 abort ..... 129  
 always\_on ..... 129  
 anymem ..... 129  
 auto ..... 130  
 break ..... 130  
 case ..... 130  
 char ..... 130  
 continue ..... 132  
 costate ..... 132  
 debug ..... 132  
 default ..... 133  
 do ..... 133

else ..... 133  
 extern ..... 133  
 firsttime ..... 134  
 float ..... 134  
 for ..... 135  
 goto ..... 135  
 if ..... 135  
 init\_on ..... 136  
 int ..... 136  
 interrupt ..... 136  
 long ..... 136  
 nodebug ..... 137  
 norst ..... 137  
 nouseix ..... 137  
 NULL ..... 137  
 protected ..... 138  
 return ..... 138  
 root ..... 139  
 segchain ..... 139  
 shared ..... 139  
 short ..... 140  
 size ..... 140  
 sizeof ..... 140  
 speed ..... 140  
 static ..... 141  
 struct ..... 141  
 switch ..... 142  
 typedef ..... 142  
 union ..... 143  
 unsigned ..... 143  
 useix ..... 143  
 waitfor ..... 143  
 while ..... 144  
 xdata ..... 144  
 xmem ..... 145  
 xstring ..... 145  
 yield ..... 145

## L

language elements .... 13, 15, 19,  
 129  
 operators ..... 151  
 latency interrupts ..... 126  
 Lib Entries ..... 421  
 LIB.DIR ..... 38, 149, 421  
 Libraries ..... 35  
 libraries ..... 3, 35  
 modules ..... 37  
 real-time programming ..... 3  
 writing your own ..... 37  
 library functions ..... 421  
 Library Help lookup ..... 39  
 Library Help lookup <CTRL-H>  
 421, 422, 423

linking .....	3	headers .....	133	shift right (>>=) .....	155
locating errors .....	402	modules .....	35, 37, 38, 39	subtract assign (=) .....	155
long .....	136, 144	body .....	37, 38, 39	XOR assign (^=) .....	156
lookup function <CTRL-H> .....	421, 422, 423	custom libraries .....	37	associativity .....	151
loops .....	29	example .....	38	binary .....	151
breaking out of .....	30	header .....	37, 38	bitwise operators	
do .....	133	key .....	37	address (&) .....	156
for .....	135	library .....	37	bitwise AND (&) .....	156
skipping to next pass .....	30	mouse .....	397	bitwise exclusive OR (^) .....	157
<b>M</b>		Multitasking		bitwise inclusive OR ( ) .....	157
macros 16, 17, 18, 115, 116, 146		cooperative .....	41	complement (~) .....	157
restrictions .....	18	preemptive .....	57	pointers .....	156
with parameters .....	16	<b>N</b>		shift left (<<) .....	156
main function .....	22, 35, 89, 137	names .....	15	shift right (>>) .....	156
memory		#define .....	16	comma .....	163
dump .....	407	Next error <CTRL-N> .....	402	conditional operators (? :) .....	161
dump at address .....	409	No Background TX .....	416	equality operators .....	158
dump Flash .....	409	nodebug 89, 114, 137, 146, 405,		equal (==) .....	158
dump to file .....	409	406, 408, 413		not equal (!=) .....	158
extended .....	4, 97, 123, 145	norst .....	137	in assembly language .....	114
logical .....	97	nouseix .....	137	logical operators .....	159
management .....	129, 139	NULL .....	137	logical AND (&&) .....	159
physical .....	97	<b>O</b>		logical NOT (!) .....	159
random access .....	4, 5	offsets in assembly language .....		logical OR ( ) .....	159
read-only .....	4, 5	116, 122, 123		operator precedence .....	163
root .....	4, 97, 98, 116, 117, 122,	online help .....	39	postfix expressions .....	159
139		operators .....	151	( ) parentheses .....	159
memory management unit		# (macros) .....	16, 17, 18	[ ] array indices .....	159
(MMU) .....	4, 97	## (macros) .....	16, 17, 18	array subscripts or dimen-	
Memory options .....	410	arithmetic operators .....	152	sion [ ] .....	159
menus		decrement (--)	154	dot (.) .....	160
COMPILE .....	398, 402	division (/)	153	parentheses ( ) .....	159
EDIT .....	398, 400, 401, 402	increment (++)	153	right arrow (->) .....	160
FILE .....	398, 399, 400	indirection (*)	153	precedence .....	151
HELP 398, 420, 421, 422, 423		minus (-)	152	reference/dereference opera-	
INSPECT 398, 407, 408, 409,		modulus (%) .....	154	tors .....	160
417		multiplication (*) .....	153	address (&) .....	160
OPTIONS .....	27, 398, 410, 411,	plus (+) .....	152	bitwise AND (&) .....	160
413, 414, 415, 416		pointers .....	153	indirection (*) .....	161
RUN .....	398, 405, 406	post-decrement (--)	154	multiplication (*) .....	161
system .....	398	post-increment (++)	153	relational operators .....	157
WINDOW 398, 417, 418, 419,		pre-decrement (--)	154	greater than (>) .....	158
420		pre-increment (++)	153	greater than or equal (>=) ..	158
message window .....	402, 417	assignment operators .....	154	less than (<) .....	157
MMU (memory management		add assign (+=) .....	154	less than or equal (<=) ..	157
unit) .....	4	AND assign (&=) .....	155	sizeof .....	162
modes		assign (=) .....	154	unary .....	151
debug .....	90, 402, 405	divide assign (/=) .....	155	Optimize For (size or speed) 413	
edit .....	402, 406	modulo assign (%=) .....	155	options	
preview .....	399	multiply assign (*=) .....	155	compiler .....	410, 411, 413
run .....	402, 405	OR assign ( =) .....	156	debugger .....	410, 414
module		shift left (<<=) .....	155	display .....	410, 415

editor ..... 410  
 memory ..... 410  
 serial ..... 410, 416  
 OPTIONS menu ... 27, 410, 411,  
 413, 414, 415, 416

## P

PageDown key ..... 397  
 PageUp key ..... 397  
 passing arguments 28, 118, 119,  
 123, 124, 125  
 Paste ..... 401  
 pasting text <CTRL-V> ..... 401  
 PC ..... 3, 405, 416  
 pointer checking ..... 27  
 pointers ..... 19, 26, 28  
   uninitialized ..... 26  
 polling ..... 405, 406  
 ports  
   serial ..... 416  
 positioning text ..... 402  
 power failure ..... 138  
 preserving registers .... 125, 126,  
 127  
 preview mode ..... 399  
 Previous error <CTRL-P> ... 402  
 primary register ... 117, 123, 125  
 primitive data types ..... 14  
 Print ..... 399  
 Print Preview ..... 399  
 Print Setup ..... 399, 400  
 printf 20, 24, 405, 406, 414, 418  
 program  
   example ..... 23  
 program flow . 28, 29, 30, 31, 32  
 programmable ROM ..... 4, 5  
 programming  
   real-time ..... 3  
 promotion ..... 152  
 protected ..... 138  
 protected variables .... 3, 89, 138  
 prototypes  
   function ..... 22, 24, 37  
   in headers ..... 37  
 punctuation ..... 14

## Q

quitting Dynamic C <ALT-F4>  
 400

## R

Rabbit reset  
 \_sysIsSoftReset ..... 379

chkHardReset ..... 188  
 chkSoftReset ..... 189  
 chkWDTO ..... 189  
 Rabbit restart  
   protected variables ..... 138  
 sysResetChain ..... 380

## RAM

static ..... 4, 5  
 read-only memory ..... 4, 5  
 real-time  
   kernel (RTK) ..... 3, 90  
   programming ..... 3  
 redoing changes

<ALT-SHIFT-backspace> ...  
 400

## registers

snapshots ..... 419  
 variables ..... 26  
 window ..... 3, 417, 419

## remote target information (RTI)

file ..... 402

Replace <F6> ..... 398, 401

replacing text ..... 401, 402

## reset

software ..... 406

Reset program <CTRL-F2> 405,  
 406

Reset target <CTRL-Y> ..... 405,  
 406

resetting program ..... 406

## restarting

program ..... 406

target controller ..... 406

ret ..... 123, 126

reti ..... 126

retn ..... 126

return ..... 123, 124, 138, 142

return address ..... 119

reverse searching ..... 401, 402

ROM ..... 405, 406

  programmable ..... 4, 5

root ..... 98, 139

  memory 4, 98, 116, 117, 122,  
 139

rst 028h ..... 405

RST 28H ..... 89

RTI (remote target information)

  file ..... 402

RTK (real-time kernel) ..... 3, 90

Run <F9> ..... 405

RUN menu ..... 405, 406

run mode ..... 402, 405

Run w/ No Polling <ALT-F9> ..

405

## running

  a program ..... 405

  in polling mode ..... 405

  with no polling ..... 405

## run-time

  checking ..... 411

## S

### sample programs

  basic C constructs ..... 23

Save ..... 399

Save as ..... 399

Save Environment ..... 417

saving a file ..... 399

scrolling ..... 419

searching for text ..... 401, 402

searching in reverse .... 401, 402

segchain ..... 32, 139

SEGSIZE ..... 97

### selecting

  COMPILE menu <ALT-C> ..  
 402

  HELP menu <ALT-H> ... 420

  OPTIONS menu <ALT-O> ..  
 410

  RUN menu <ALT-R> ..... 405

  WINDOW menu <ALT-W> .  
 417

serial communication ..... 416

serial options ..... 410, 416

serial port ..... 416

shared ..... 139

shared variables ..... 3, 89, 138

SHIFT-F5 ..... 402

short ..... 140

Show Tool Bar ..... 416

single stepping ..... 90, 121, 408

  in assembly language ..... 89

  with descent <F7> ..... 406

  without descent <F8> ..... 406

size ..... 140

sizeof ..... 140

skipping to next loop pass .... 30

Slice Statements ..... 57

soft break points ..... 405, 406

### software

  libraries ..... 35, 37

  reset ..... 406

  source window ..... 417

  SP (stack pointer) 119, 124, 125,  
 127, 149

  special characters ..... 20

### special symbols

  in assembly language ..... 116

speed .....140  
 stack 28, 90, 118, 119, 122, 124, 125, 126, 127, 130, 137  
     checking .....89, 90  
     frame 118, 119, 124, 125, 127  
     frame reference point .....124  
     frame reference pointer .....90, 122, 123, 137  
     pointer (SP) .....119, 124, 125, 127, 149  
     snapshots .....419  
     window .....3, 417, 419  
 STACKSEG .....97  
 standalone  
     assembly code .....117  
 state machine  
     example .....43  
 statements .....21  
 static .....141  
     RAM .....4, 5  
     variables .....5, 116, 118  
 status register (F) .....419  
 STDIO window 3, 414, 417, 418  
 Step over <F8> .....405, 406  
 Stop <CTRL-Z> .....405  
 stop bits .....416  
 stopping a running program .405  
 storage class .....21  
     auto .....25  
     default .....5  
     register .....25, 26  
     static .....25  
 strcpy .....423  
 strings .....19, 20, 144, 146, 147  
     functions .....19  
     terminating null byte .....19  
 struct ...21, 24, 25, 28, 116, 119, 124, 125, 141  
 structures .24, 25, 116, 119, 124, 125  
     return space .....119, 124, 125  
 subscripts  
     array .....24  
 support files .....36  
 switch .....32, 133, 142  
     breaking out of .....30  
     case .....142  
 switching to edit mode .....402  
 symbolic constant .....146  
 Sync. Bkgnd TX .....416

**T**

targetless compilation .....402  
 text cursor .....423

tiling windows .....417  
 Toggle break point <F2> ....405, 406  
 Toggle hard break point <ALT-F2> .....405, 406  
 Toggle interrupt <CTRL-I> 405, 406  
 Toggle polling <CTRL-O> .405, 406  
 toolbar .....416  
 Trace into <F7> .....405, 406  
 type  
     checking .....22, 413  
     conversion .....152  
     definitions .....23  
 type casting .....152  
 typedef .....23, 142  
 types  
     function .....22

**U**

unary operators .....151  
 unbalanced stack .....127  
 undoing changes <ALT-Back-space> .....400  
 uninitialized  
     pointers .....26  
 union .....21, 25, 143  
 unreserved registers ..125, 126, 127  
 unsigned .....143  
 untitled files .....399  
 Update Watch window <CTRL-U> .....408  
 useix .....90, 122, 143

**V**

variables  
     global .....25  
 vertical tiling .....417

**W**

waitfor .....143  
 waitfordone .....144  
 warning reports .....411  
 watch  
     dialog .....407, 408  
     expressions .....408, 417  
     list .....408  
     window .....3, 407, 408, 417  
         adding items .....407, 408  
         clearing .....408  
         deleting items .....407, 408  
         updating .....408  
 wfd .....144  
 while .....21, 29, 144  
 WINDOW menu .417, 418, 419, 420  
 windows .....417  
     assembly .....3, 121, 417, 418  
     cascaded .....417  
     information .....417, 420  
     message .....417  
     register .....3, 417, 419  
     stack .....3, 417, 419  
     STDIO .....3, 414, 417, 418  
     tiled horizontally .....417  
     tiled vertically .....417  
     watch .....3, 407, 408, 417

**X**

xdata .....144  
 xmem .....123, 145  
 XPC .....97  
 xstring .....145

**Y**

yield .....145

**Z**

Z180 .....97, 419

