

```
{  
    FILE* sfile;  
    int count = 0;  
  
    sfile = fopen("file1", "r");  
  
    if( sfile == NULL )  
    {  
        return -1;  
    }  
  
    while (1)  
    {  
        char c;  
        c = fgetc(sfile);  
        if(c == EOF)  
        {  
            break;  
        }  
        else  
        {  
            count++;  
        }  
    }  
  
    return count;  
}
```

68K/ColdFire v10.0

C Compiler/Assembler Reference Manual

A publication of
Altium BV
Documentation Department
Copyright © 1997–2003 Altium BV

All rights reserved. Reproduction in whole or part is prohibited
without the written consent of the copyright owner.

TASKING is a brand name of Altium Limited.

The following trademarks are acknowledged:

FLEXlm is a registered trademark of Globetrotter Software, Inc.
HP and HP-UX are trademarks of Hewlett-Packard Co.
Motorola is a trademark of Motorola, Inc.
MS-DOS and Windows are registered trademarks of Microsoft Corporation.
Solaris is a trademark of Sun Microsystems, Inc.
UNIX is a registered trademark of X/Open Company, Ltd.

All other trademarks are property of their respective owners.

Data subject to alteration without notice.

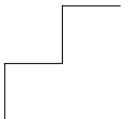
<http://www.tasking.com>
<http://www.altium.com>

The information in this document has been carefully reviewed and is believed to be accurate and reliable. However, Altium assumes no liabilities for inaccuracies in this document. Furthermore, the delivery of this information does not convey to the recipient any license to use or copy the software or documentation, except as provided in an executed license agreement covering the software and documentation.

Altium reserves the right to change specifications embodied in this document without prior notice.

CONTENTS

TABLE OF CONTENTS



CONTENTS

INTRODUCTION **1-1**

| | | |
|-----|---------------------|-----|
| 1.1 | Overview | 1-3 |
| 1.2 | Documentation | 1-3 |

RUN-TIME LIBRARY **2-1**

| | | |
|---------|---------------------------------------------------------------------------------------------|------|
| 2.1 | Introduction | 2-3 |
| 2.2 | System Initialization | 2-4 |
| 2.3 | I/O System | 2-6 |
| 2.4 | Time Functions | 2-8 |
| 2.4.1 | Time Conversion Routines | 2-8 |
| 2.4.2 | Low-level Time/Timer Routines | 2-8 |
| 2.5 | Storage Allocation | 2-9 |
| 2.6 | Support for the M68302ADS Development System | 2-9 |
| 2.7 | Support for the M68340BCC Development System | 2-10 |
| 2.8 | Support for the M68360QUADS Development System . | 2-11 |
| 2.9 | Modifying the Libraries | 2-11 |
| 2.9.1 | Integrating New Routines Into an Existing Library Without Using make on Unix Hosts | 2-12 |
| 2.10 | Library Object Modules | 2-13 |
| 2.11 | Summary of Library Routines | 2-14 |
| 2.11.1 | Standard Functions | 2-14 |
| 2.11.2 | Mathematical Functions | 2-16 |
| 2.11.3 | Standard I/O Functions | 2-17 |
| 2.11.4 | String Manipulation Functions | 2-19 |
| 2.11.5 | Non-local Goto Functions | 2-20 |
| 2.11.6 | Date and Time Routines | 2-21 |
| 2.11.7 | ASCII Character Set Macros and Functions | 2-21 |
| 2.11.8 | Global Definitions | 2-22 |
| 2.11.9 | Compile-time Assertions | 2-23 |
| 2.11.10 | Formatting of Numeric Values | 2-23 |
| 2.11.11 | Variable Length Argument List Access | 2-23 |
| 2.11.12 | Signal Handling | 2-24 |
| 2.11.13 | C Library Extensions | 2-24 |
| 2.12 | Run-Time Library Routines | 2-26 |

ASSEMBLY LANGUAGE REFERENCE **3-1**

| | | |
|-----|-------------------------------------|-----|
| 3.1 | Preface | 3-3 |
| 3.2 | Related Publications | 3-3 |
| 3.3 | Using Assembly Language | 3-4 |
| 3.4 | Elements of Assembly Language | 3-4 |
| 3.5 | Notation | 3-5 |

SOURCE PROGRAM CODING **4-1**

| | | |
|-------|------------------------------------------------------------------|------|
| 4.1 | Introduction | 4-3 |
| 4.2 | Comments | 4-3 |
| 4.3 | Source Line Format | 4-4 |
| 4.3.1 | Label Field | 4-4 |
| 4.3.2 | Operation Field | 4-5 |
| 4.3.3 | Operand Field | 4-7 |
| 4.3.4 | Comment Field | 4-7 |
| 4.4 | Symbols | 4-7 |
| 4.4.1 | Symbol Syntax | 4-8 |
| 4.4.2 | Symbol Definition Classes | 4-8 |
| 4.4.3 | User-Defined Labels | 4-9 |
| 4.4.4 | Location Counter Symbol "*)" | 4-9 |
| 4.5 | Constants | 4-9 |
| 4.5.1 | Integer Constants | 4-9 |
| 4.5.2 | Character Constants | 4-11 |
| 4.5.3 | Floating Point Constants (68881/68882/68040/68060 only) | 4-11 |
| 4.6 | Operators | 4-12 |
| 4.7 | Expressions | 4-14 |
| 4.8 | Addressing Modes | 4-16 |

ASSEMBLER DIRECTIVES **5-1**

| | | |
|-------|-------------------------------------------|-----|
| 5.1 | Assembly Control | 5-3 |
| 5.1.1 | COMMON – Enter Named Common Section | 5-4 |
| 5.1.2 | END – Program End | 5-5 |

| | | |
|--------|-----------------------------------------------------|------|
| 5.1.3 | INCLUDE – Include Secondary File | 5-6 |
| 5.1.4 | OFFSET – Define Offsets | 5-6 |
| 5.1.5 | ORG – Absolute Origin | 5-7 |
| 5.1.6 | RESERVE – Reserve storage | 5-8 |
| 5.1.7 | RESUME – Resume defined section | 5-9 |
| 5.1.8 | RORG – Relocatable ORG | 5-9 |
| 5.1.9 | SECTION – Relocatable Program Section | 5-10 |
| 5.2 | Symbol Definition | 5-11 |
| 5.2.1 | EQU – Equate Symbol Value | 5-12 |
| 5.2.2 | FEQU – Equate Floating Point Symbol Value | 5-12 |
| 5.2.3 | REG – Define Register List | 5-13 |
| 5.2.4 | SET – Set Symbol Value | 5-13 |
| 5.3 | Data Definition/Storage Allocation | 5-14 |
| 5.3.1 | COMLINE – Unimplemented | 5-14 |
| 5.3.2 | DC – Define Constant | 5-14 |
| 5.3.3 | DCB – Define Constant Block | 5-17 |
| 5.3.4 | DS – Define Storage | 5-17 |
| 5.4 | Listing Control and Output Options | 5-18 |
| 5.4.1 | FAIL – Programmer Generated Error | 5-19 |
| 5.4.2 | FORMAT/NOFORMAT – Unimplemented | 5-19 |
| 5.4.3 | LIST/NOLIST – Control Listing Generation | 5-19 |
| 5.4.4 | LLEN – Unimplemented | 5-19 |
| 5.4.5 | NOOBJ – Unimplemented | 5-19 |
| 5.4.6 | OPT – Assembler Options | 5-20 |
| 5.4.7 | PAGE/NOPAGE – Control Pagination | 5-23 |
| 5.4.8 | SPC – Space Between Source Lines | 5-23 |
| 5.4.9 | STTL – Set Subtitle | 5-23 |
| 5.4.10 | TTL – Set Title | 5-23 |
| 5.5 | External Symbol Controls | 5-24 |
| 5.5.1 | IDNT – Relocatable Identification Record | 5-24 |
| 5.5.2 | XDEF – External Symbol Definition | 5-24 |
| 5.5.3 | XREF – External Symbol Reference | 5-25 |
| 5.6 | Internal Assembly Controls | 5-25 |
| 5.6.1 | _BRINGIN Declare external symbol | 5-26 |
| 5.6.2 | _DEBSYM Put out debugging information | 5-26 |

5.6.3 _DGROUP Define data group 5-26

MACRO OPERATIONS AND CONDITIONAL ASSEMBLY **6-1**

6.1 Macro Operations 6-3

6.1.1 Macro Definition 6-4

6.1.2 Macro Invocation 6-4

6.1.3 Macro Parameter Definition and Use 6-5

6.1.4 Labels Within Macros 6-6

6.1.5 The MEXIT Directive 6-6

6.1.6 The NARG Symbol 6-7

6.1.7 Implementation of Macro Definition 6-7

6.1.8 Implementation of Macro Expansion 6-7

6.2 Conditional Assembly 6-8

6.2.1 Conditional Assembly Structure 6-9

6.2.2 Example of Macro and Conditional Assembly Usage .. 6-11

STRUCTURED CONTROL STATEMENTS **7-1**

7.1 Keyword Symbols 7-3

7.2 Syntax 7-3

7.2.1 IF Statement 7-5

7.2.2 Floating-Point Structured Assembler Syntax for
the IF Statement 7-6

7.2.3 FOR Statement 7-7

7.2.4 REPEAT Statement 7-8

7.2.5 WHILE Statement 7-8

7.3 Simple and Compound Expressions 7-9

7.3.1 Simple Expressions 7-9

7.3.2 Condition Code Expressions 7-9

7.3.3 Operand Comparison Expressions 7-10

7.3.4 Compound Expressions 7-12

7.4 Source Line Formatting 7-12

7.4.1 Class 1 Symbol Usage 7-12

| | | |
|-----------------------------------|---------------------------------------------------------------------------------|------------|
| 7.4.2 | Nesting of Structured Statements | 7-13 |
| 7.5 | Effects on the User's Environment | 7-14 |
| POSITION- INDEPENDENT CODE | | 8-1 |
| 8.1 | Forcing Position Independence | 8-3 |
| 8.2 | Base-Displacement Addressing | 8-4 |
| 8.3 | Base-Displacement in Conjunction with Forced Position Independence | 8-4 |
| CHARACTER SET | | A-1 |
| 1 | Characters Recognized | A-3 |
| 2 | ASCII Character Set | A-3 |
| <u>INDEX</u> | | |



CONTENTS

MANUAL PURPOSE AND STRUCTURE

PURPOSE

This manual is for users of the 68K/ColdFire C compiler/assembler.

MANUAL STRUCTURE

1. Introduction
Introduces the structure and conventions of the manuals
2. Run-Time Library
Covers installing and changing run-time libraries.
3. The Assembly Language Reference
Summarizes the structure of the assembly language and gives an assembly language overview.
4. Source Program Coding
Discusses source program coding including source line format, symbols, constants, registers, operators, expressions, addressing modes, instruction mnemonics, and other instruction types.
5. Assembler Directives
Describes and gives examples of the basic forms of the most frequently used assembler directives.
6. Macro Operations and Conditional Assembly
Describes the macro and the conditional assembly capabilities of the assembler.
7. Structured Control Statements
Describes how to use structured control statements with assembly language to improve readability of assembly language.
8. Position-independent Code
Describes Forcing Position Independence, Base-Displacement Addressing, and Base-Displacement in Conjunction with Forced Position Independence.

APPENDICES

- A. Character Set
 - Contains a list of the ASCII characters recognized by the assembler.

INDEX

CONVENTIONS USED IN THIS MANUAL

The notation used to describe the format of call lines is given below:

- { } Items shown inside curly braces enclose a list from which you must choose an item.
- [] Items shown inside square brackets enclose items that are optional.
- | The vertical bar separates items in a list. It can be read as OR.
- italics* Items shown in italic letters mean that you have to substitute the item. If italic items are inside square brackets, they are optional. For example:

filename

means: type the name of your file in place of the word *filename*.
- ... An ellipsis indicates that you can repeat the preceding item zero or more times.
- screen font Represents input examples and screen output examples.
- bold font** Represents a command name, an option or a complete command line which you can enter.

For example

command [*option*]... *filename*

This line could be written in plain English as: execute the command *command* with the optional options *option* and with the file *filename*.

Illustrations

The following illustrations are used in this manual:



This is a note. It gives you extra information.



This is a warning. Read the information carefully.



This illustration indicates actions you can perform with the mouse.



This illustration indicates keyboard input.



This illustration can be read as “See also”. It contains a reference to another command, option or section.



MANUAL STRUCTURE

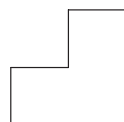
CHAPTER

1

INTRODUCTION



TASKING



1

CHAPTER

1.1 OVERVIEW

This *C Compiler/Assembler Reference Manual* contains run-time library and assembly language information. This chapter contains an overview of the 68K/ColdFire documentation. Please refer to the *Introduction* chapter in the *Getting Started Manual* for information concerning the 68K/ColdFire development system and for additional help.

1.2 DOCUMENTATION

Three manuals make up the 68K/ColdFire documentation: the *Getting Started Manual*, the *C Compiler/Assembler User's Manual* and the *C Compiler/Assembler Reference Manual*.

The *Getting Started Manual* contains an introduction to the development system, an installation guide, and a tutorial which contains sample code and exercises which lead you step-by-step through the powerful features of each software tool.

The *C Compiler/Assembler User's Manual* includes invocation, options, and usage summaries, along with examples for each of the tools and definitions of special terminology and functions. This manual also contains additional information in the appendices on run-time and naming conventions, C language extensions, and object module formats.

The *C Compiler/Assembler Reference Manual* provides information on the run-time libraries and the information necessary to write programs in assembly language. It contains sections on source program coding, assembler directives, macro operations, structured control statements, and position-independent code, as well as a summary of the character set.

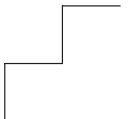


INTRODUCTION

CHAPTER

2

RUN-TIME LIBRARY



2

CHAPTER

This chapter contains the following sections:

- Introduction
- System Initialization
- I/O System
- Storage Allocation
- Support for the M68302ADS Development System
- Support for the M68340BCC Development System
- Support for the M68360QUADS Development System
- Modifying the Libraries
- Library Object Modules
- Summary of Library Routines
- Run-Time Library Routines

2.1 INTRODUCTION

This section identifies the parts of the run-time library which you may need to modify to integrate with your hardware and software. It also describes in some detail the nature of the required changes and how to install them in the library. Finally, this appendix describes each library routine. The overall organization of the run-time library is discussed in the *Compiler Library Organization* section in the *Linking Locator* chapter of the *User's Manual*.

The source code for most of the library routines, including the environment-dependent routines, is supplied. However, software floating-point emulation routines are not supplied.

The run-time library is primarily written in C, but the lowest level functions are written in assembly language.

The run-time library routines supplied can be used to interface to most target systems. In addition, there are routines written specifically for the M68302ADS, M68340BCC, and M68360QUADS Development Systems. These will be discussed later.

You can code your replacement routines in C or assembly language. Remember, any routines you code in assembly language must adhere to the conventions described in the *Linking C and Assembly* application note in the *User's Manual*.



As noted in the *Linking Locator* chapter in the *User's Manual*, there are multiple versions of object modules in the library. You must decide which library(ies) will be used before deciding what changes to make to the source modules.

You must supply replacement modules suitable for inclusion in your libraries. If you code a replacement module in C, you need to compile it with the appropriate options for each library you will use. However, if you code your module in assembly language, you must be sure the assembly language routine is correct for each library.

2.2 SYSTEM INITIALIZATION

Three features must be provided to establish an execution environment:

- The power-on condition on your target system must somehow transfer control to user code.
- The run-time environment must be appropriately initialized before compiled code is activated.
- An appropriate action should be taken when the top-level compiled routine exits.

These functions are provided by a run-time library routine. You can find the source for this routine in the run-time library source directory of the product.

| Target | Initialization File |
|--------------------------------------------------|---------------------|
| 68K targets with VME 105/107 | pmainr.68k |
| 68030 with VME143 | pmn030r.68k |
| 68040, hardware floating-point | pmn040fp.68k |
| 68060, hardware floating-point | pmainf.68k |
| 68060, no floating-point, with ROM monitor | pmn060r.68k |
| 68060, hardware floating-point, with ROM monitor | pmn06rf.68k |
| 68302 with M68302ADS | pmn302a.68k |
| 68332 (MC68330, MC68331, MC68332, MC68336) | pmn332.68k |
| 68332 with M68332EVS | pmn332r.68k |
| 68340 | pmn340.68k |
| 68340 with M68340BCC | pmn340b.68k |

| Target | Initialization File |
|--------------------------------------|---------------------|
| 68360 | pmn360.68k |
| 68360 with M68360QUADS | pmn360b.68k |
| All other 68K targets | pmain.68k |
| ColdFire targets without ROM Monitor | pmain.asm |
| ColdFire targets with ROM Monitor | pmainr.asm |

Table 2-1: Initialization files

When the 68000 hardware starts execution after a power on (cold start), it loads the SSP and PC registers from absolute locations 0 and 4. The 68020 behaves similarly, except that the ISP register replaces SSP. `pmain` defines eight bytes of data which are absolutely assembled at location 0. The initial PC value is the address of the `__main` routine. This address is also designated as the “start” address of the system. The initial SSP (ISP) value is 7FFC.

`pmain` performs the setup operations required by the compiled code, and then executes a long call to the external label `_main`. By default the compiler generates the global label `_main` at the start of your C main routine.

Of course, you can name the main routine whatever you like, but the initialization module `pmain` must be adjusted accordingly. The required setup operations are enumerated below:

1. Provide a stack area and initialize the USP and SSP (user and system stack pointer) registers accordingly. For the 68020, the USP, ISP, and MSP (user, interrupt, and master stack pointers) are set.
2. Initialize A6 (frame pointer) to zero.
3. Initialize A5 to point to the global data area.

The compiler assumes that the A5 register always contains the base address of the global data area. The global data area is a group named `data`, which consists of the `idata` and `udata` segments. In a real time system where re-entrancy is necessary, a program must dynamically allocate its stack and global data area. This ensures that multiple real time tasks running the same program will use different A5 and A7 register values.



If your system dynamically allocates the data area, remember that the base address (in A5) is the actual address of the data area if its size is less than or equal to 32K, otherwise it is the address of the data area *plus 32K*. The linking locator creates a global symbol named “ldata” whose value is the size of the data group. This symbol may be useful in coding the call to dynamically allocate memory.



It is possible to configure your system so that the A5 register is not used. You must rebuild the libraries and use the command line `-sd` option on all compilations. See the *Building Libraries That Do Not Use A5* application note in the *User's Manual* for more details.

4. In the hardware floating-point libraries, the 68881 floating-point coprocessor is initialized by setting both the floating-point status register, FPSR, and floating-point control register, FPCR, to zero.
5. You will have to define what happens when the user program returns (if that is possible). This decision is reflected in two places: in the `exit` routine, and just after the call to `_main` in the initialization code. Our sample routines contain an infinite loop. You may want your exit routine to deliver a return code and close any I/O channels.

Please refer to the initialization file appropriate to the target you are using for more information.

2.3 I/O SYSTEM

You will have to change the low level routines that “put” and “get” characters to interface with the character I/O on your target system. If you do not intend to support multiple files then this is all you need to do. If you do intend to support multiple files (our sample implementation does not) then you must modify the next higher level of I/O routines (`getc` and `putc`). You must also define what a “file control block” looks like, and what it means to “open” and “close” files.

The entire I/O system assumes some underlying structure that contains a file control block of some sort; however, only the routines mentioned below actually manipulate the contents of that structure. The only things ever passed to these routines or returned by them are pointers to the file control structure.

Here is a list of the routines you will have to provide:

- `FILE* fopen (char *filename, char *mode);`

The first argument can be something defined by your installation: maybe a port address, maybe a pointer to a string of characters. It **must** be the same size as a pointer. The second argument must definitely be a pointer to a character that specifies the mode to open the file.

If the mode character is `r`, it is open for reading, `w` means write, and `a` means append. `fopen` returns a pointer to a file control block.

- `void fclose (FILE *stream);`

The argument is the kind of pointer returned by `fopen`. This routine performs cleanup tasks, for example, flushing buffers.

- `int getc (FILE *stream);`

This function returns the next character (8-bit quantity) from the given I/O stream. The result is returned in an integer variable. It must return `-1` (all bits on) when it finds an end-of-file condition.

Our `getc` calls an external assembly language routine `_getc` which does the actual input.

- `int ungetc (char c, FILE *stream);`

Pushes the given character back into the stream. The character is returned in an integer variable. Only one “ungetc-ed” character at a time need be supported.

- `int putc (char c, FILE *stream);`

Writes the given character onto the given file stream. The character written is returned in an integer variable.

Our `putc` calls an external assembly language routine `_putc` which does the actual output.

The file `stdio.c` contains three global variables of type `FILE *`: `stdin`, `stdout`, `stderr`. These represent the default input, output, and error reporting I/O streams.

Sample code to perform these functions is provided in the following library modules:

| <u>Source</u> | | <u>Object</u> |
|----------------------|---|------------------------------------------------|
| <code>putc.c</code> | → | <code>putc.ln</code> , <code>putc.lln</code> |
| <code>getc.c</code> | → | <code>getc.ln</code> , <code>getc.lln</code> |
| <code>fopen.c</code> | → | <code>fopen.ln</code> , <code>fopen.lln</code> |
| <code>stdio.c</code> | → | <code>stdio.ln</code> , <code>stdio.lln</code> |
| <code>stdio.h</code> | | (include file) |

2.4 TIME FUNCTIONS

The current default libraries provide all of the time conversion and low-level time/timer routines described in Section 4.12 of the ANSI Standard. However, all low-level timer functions return ANSI values, stating that the timer function is not implemented.

If your application requires current low-level time/timer information, in addition to time conversion, you must modify the low-level time/timer routines to use the time hardware. These modifications are discussed below.

2.4.1 TIME CONVERSION ROUTINES

The `gmtime` routine is the only time conversion routine that returns the value of `(struct tm *) NULL`, which requires a low-level time function. You should modify this routine so that it returns the current UTC time, as prescribed in ANSI.

2.4.2 LOW-LEVEL TIME/TIMER ROUTINES

The `clock` in the `time.c` routine currently returns the value of `(clock_t)-1`. You should modify this routine so that it returns the elapsed clock count, as prescribed in ANSI.

The `time` in the `time.c` routine currently returns the value of `(time_t)-1`. You should modify this routine so that it returns the current calendar time, as prescribed in ANSI.

2.5 STORAGE ALLOCATION

The library storage allocation routines request “system” storage when they do not possess enough free storage to satisfy an allocation request. The routine which provides system storage is called `_alloc`.

```
char * _alloc (size_t request, size_t *given);
```

The first parameter is an integer: the number of words requested. The second parameter is a pointer to an integer. The routine returns the null pointer if it cannot provide at least as many words as were requested. Otherwise, it returns a pointer to a chunk of storage and sets the integer pointed to by the second parameter to the number of words actually allocated. This might be more than was actually requested.

A sample implementation is provided by the `xalloc` module for most targets. It implements a 4K heap.

```
xalloc.c → xalloc.ln, xalloc.lln
```

2.6 SUPPORT FOR THE M68302ADS DEVELOPMENT SYSTEM

There are two libraries that support the 68302 target with the M68302ADS Development System. `lib302ap` contains modules to support the M68302ADS with parallel I/O; `lib302at` contains modules to support the M68302ADS with trap-based I/O. The source files written specifically for this environment have the characters 302a in their names.

In addition to the extra source files, a locator command file, `ads302.cmd`, is supplied to specify the memory map of the M68302ADS card. This file results in the following:

- A `MEMORY` command defines the maximum size of memory on the M68302ADS as 512 kilobytes of RAM.
- A `RESERVE` command ensures that the first 0x4000 bytes are reserved for use by the monitor, `bug302`.

- The startup module, defined in `pmn302a.68k`, is in segment `init` and is located at address `0x4000`.
- The remaining segments are placed in memory starting at address `0x4080`.
- The segment `S_end_project` is used by the dynamic memory allocator to indicate the end of used memory.



`ads302.cmd` for more details.

2.7 SUPPORT FOR THE M68340BCC DEVELOPMENT SYSTEM

The `lib340b` library supports the 68340 target with the M68340BCC Development System. The source files written specifically for this environment have the characters `340b` in their names.

In addition to the extra source files, a locator command file, `bcc340.cmd`, is supplied to specify the memory map of the M68340BCC card. This file results in the following:

- A `MEMORY` command defines the maximum size of memory on the M68340BCC as 64 kilobytes of RAM.
- A `RESERVE` command ensures that the first `0x3000` bytes are reserved for use by the monitor program, `340bug`.
- The startup module, defined in `pmn340b.68k`, is in segment `init` and is located at address `0x3000`.
- The remaining segments are placed in memory starting at address `0x3080`.
- The segment `S_end_project` is used by the dynamic memory allocator to indicate the end of used memory.



`bcc340.cmd` for more details.

2.8 SUPPORT FOR THE M68360QUADS DEVELOPMENT SYSTEM

The `lib360b` library supports the 68360 target with the M68360QUADS Development System. The source files written specifically for this environment have the characters 360b in their names. The `lib360b` library also uses some M68340BCC sources files (which are M68360QUADS-compatible). These files have the characters 340b in their names.

In addition to the extra source files, a locator command file, `quads360.cmd`, is supplied to specify the memory map of the M68360QUADS card. This file results in the following:

- A `MEMORY` command defines the maximum size of memory on the M68360QUADS as 0x4E0000 of RAM.
- A `RESERVE` command ensures that locations between 0x0 – 0x20000 and 0x21800 – 0x400000 are reserved.
- The startup module, defined in `pmn360b.68k`, is in segment `init` and is located at address 0x400000.
- The segment `S_end_project` is used by the dynamic memory allocator to indicate the end of used memory.



`quads360.cmd` for more details.

2.9 MODIFYING THE LIBRARIES

Once you know what your low-level routines are going to look like, you can begin editing your replacement files. When you have completed your replacement routines, refer to the following procedure to integrate your new routines into the library.

The calling convention with which the run-time library you are modifying was built affects the way you assemble or compile any new library routines. All routines in a library must use the same calling convention.

Given the number of different run-time libraries, it is possible that you may have to go through this entire procedure several times. That is, once for the each target and once for both the hardware and software floating-point libraries, if you are using both. The integration procedure is identical in all cases. Of course, if you never intend to use a library you need not update it.

The only time the hardware/software library pairing affects the coding of your low-level routines is with the system initialization routine. You may need two slightly different `pmain` routines: one which contains instructions to initialize the 68881/68882 floating-point coprocessor and one which does not. Of course, the one which initializes the coprocessor goes in the hardware floating-point library.

From now on, we will describe the process of integrating with the library as if only the 68000 library exists.

2.9.1 INTEGRATING NEW ROUTINES INTO AN EXISTING LIBRARY WITHOUT USING MAKE ON UNIX HOSTS

1. Save the original versions of the sources and object modules for all the routines you will change.
2. Copy your new source files into the run-time library directory.
3. Assemble any new assembly language source modules.
4. Link each of the resulting `.o1` object module files with itself to produce the new `.1n` files. Use the `llink` utility and supply the `-lo`, `-o` and `-w` options.
5. 68K only: Compile all the new C source modules supplying the `-L` option.
6. 68K only: Link each of the resulting object modules with itself. Supply the `-lo`, `-o` and `-w` options to the `llink` utility to produce a `.1n` module for each new object module.
7. 68K only: Rename each linked C module from `- - .1n` to `- - .11n`.
8. Compile all the new C source modules again, but this time **without** `-L`.
9. Link each of the resulting object modules with itself. Supply the `-lo`, `-o` and `-w` options to the `llink` utility to produce a `.1n` module for each new object module.
10. Update the library index files using the `librarian` utility. This process is described in more detail below.

To update each library, you must use the `librarian` to:

- Delete the old object module, unless the new one has the same name.

- Add/replace the appropriate new module(s).

Example: Updating _alloc

Suppose you have coded a replacement for `xalloc.c` and your replacement routine is also called `xalloc.c`. You have compiled and linked your routine twice, once with the `-L` option and once without, producing `xalloc.lln` and `xalloc.ln`.

Enter the following librarian invocations:

```
libr xalloc.ln -v -L lib000
libr xalloc.ln -v -L lib000.nf
libr xalloc.lln -v -L lib000.l
libr xalloc.lln -v -L lib000.lnf
```



There is no need to delete the old library member, since the new member has the same name.

You can use the librarian utility to list all library entries.

2.10 LIBRARY OBJECT MODULES

Depending upon which run-time libraries have been installed on your system, your run-time library may include several directories containing library index files and library object modules. A single linked object module may be named in more than one library index file in its directory. For ease in building and modifying run-time libraries, a standard naming convention is used for the linked object modules.

When you modify a library source, you must rebuild all the corresponding object modules. This may be done by following the steps outlined above, in the section *Modifying the Libraries*.

Not all sources are compiled into all possible suffix combinations; you need only replace the ones which exist. For example, the `.in` suffix only applies to two modules: `xprintf.c` and `xscanf.c`. This special suffix is produced from compiling these modules with the `-P NO_FP_IO` option. This option defines a preprocessor variable which causes the compiler to exclude the code printing or scanning floating-point data. The resulting module can only be used in a “no-floats” library index file.



The following table summarizes the correct compiler options to supply when recompiling a library module and the meaning of the object module's utilities:

| Suffix | Meaning | Compiler Options |
|--------|----------------------|------------------|
| .ln | Default | |
| .in | No Floats | -P NO _FP _IO |
| .lln | Long Ints | -L |
| .iln | Long Ints, No Floats | -L -P NO _FP _IO |

Table 2-2: Options for recompiling a library module

2.11 SUMMARY OF LIBRARY ROUTINES

Run-time library routines can be accessed from C source code, via #include statements to resolve references such as simple math and I/O functions. The library routines follow the ANSI specification. In addition, TASKING has its own set of library routines described in the include file extended.h.

2.11.1 STANDARD FUNCTIONS

The standard functions are described in the ANSI C specification. Their external declarations are available in the library include file stdlib.h. A table summarizing the standard routines appears below:

| Name | Definition |
|---------|---------------------------------------------------------------|
| abort | terminate program |
| abs | absolute value |
| atexit | register functions to be called at normal program termination |
| atof | string to double conversion |
| atoi | string to integer conversion |
| atol | string to long integer conversion |
| bsearch | search an array of objects |
| calloc | allocate and zero dynamic storage |

| Name | Definition |
|----------|--------------------------------------------------------------------|
| div | compute integer quotient and remainder |
| exit | terminate a process |
| free | free previously allocated storage |
| getenv | get an environment variable |
| labs | long integer absolute value |
| ldiv | compute long quotient and remainder |
| malloc | allocate but do not zero dynamic storage |
| mblen | return multi-byte character length |
| mbstowcs | convert multi-byte string to wide-char string |
| mbtowc | convert multi-byte char to wide-char char |
| qsort | sort an array of elements |
| rand | return a random number between 0 and 32767 |
| realloc | change the size of an object |
| srand | reset the seed for <code>rand</code> , the random number generator |
| strtod | convert a string into a double |
| strtol | convert a string into a long integer |
| strtoul | convert a string into an unsigned long integer |
| system | pass a string to the host environment's command processor |
| wcstombs | convert wide-char string to multi-byte string |
| wctomb | convert wide-char char to multi-byte char |

Table 2-3: Standard functions



2.11.2 MATHEMATICAL FUNCTIONS

Mathematical functions compute and return a value based on the given argument(s). Their external declarations are available in the library include file `math.h`. A table summarizing the mathematical routines appears below:

| Name | Definition |
|--------------------|----------------------------------------------|
| <code>acos</code> | arccosine |
| <code>asin</code> | arcsine |
| <code>atan</code> | arctangent in range $-\pi/2$ to $\pi/2$ |
| <code>atan2</code> | arctangent of x/y in range $-\pi$ to π |
| <code>ceil</code> | round to more positive integer |
| <code>cos</code> | cosine |
| <code>cosh</code> | hyperbolic cosine |
| <code>exp</code> | exponential |
| <code>fabs</code> | floating-point absolute value |
| <code>floor</code> | round to more negative integer |
| <code>fmod</code> | floating-point modulus |
| <code>frexp</code> | extract fraction from exponent |
| <code>ldexp</code> | scale double exponent |
| <code>log</code> | natural logarithm |
| <code>log10</code> | common (base 10) logarithm |
| <code>modf</code> | extract fraction and integer from double |
| <code>pow</code> | raise x to the y power |
| <code>sin</code> | sine |
| <code>sinh</code> | hyperbolic sine |
| <code>sqrt</code> | real square root |
| <code>tan</code> | tangent |
| <code>tanh</code> | hyperbolic tangent |

Table 2-4: Mathematical functions

2.11.3 STANDARD I/O FUNCTIONS

The I/O system assumes some underlying structure that contains a file control block of some sort. Some files actually manipulate the contents of that structure. The include file `stdio.h` contains external declarations for these functions. A summary of standard I/O functions appears below:

| Name | Definition |
|-----------------------|-------------------------------------------------------------------|
| <code>clearerr</code> | clear the end-of-file and error indicators |
| <code>fclose</code> | close the specified file |
| <code>feof</code> | test the end-of-file indicator for a file |
| <code>ferror</code> | test the error indicator for a file |
| <code>fflush</code> | flush output buffer |
| <code>fgetc</code> | read a character from the specified file |
| <code>fgetpos</code> | store the current value of the file position indicator |
| <code>fgets</code> | read a string from the specified file |
| <code>fopen</code> | open a file |
| <code>fprintf</code> | write formatted output to the specified file |
| <code>fputc</code> | write a character to the specified file |
| <code>fputs</code> | write a string to the specified file |
| <code>fread</code> | block read from file |
| <code>freopen</code> | close, then open the specified file |
| <code>fscanf</code> | read formatted input from specified file |
| <code>fseek</code> | set the file position indicator for a file |
| <code>fsetpos</code> | set the file position indicator for a file to a specific value |
| <code>ftell</code> | return the current file position indicator for a specific stream |
| <code>fwrite</code> | block write to file |
| <code>getc</code> | same as <code>fgetc</code> |
| <code>getchar</code> | read a character from standard input |
| <code>gets</code> | read a line from standard input |
| <code>perror</code> | map the error number in an integer expression to an error message |



| Name | Definition |
|----------|-------------------------------------------------------------------------|
| printf | write formatted output to standard output |
| putc | same as <code>fputc</code> |
| putchar | write a character to standard output |
| puts | write a string to standard output |
| remove | delete a file |
| rename | rename a file |
| rewind | rewind a file |
| scanf | read formatted input from standard input |
| setbuf | set I/O buffer |
| setvbuf | set the buffering mode |
| sprintf | write formatted output to the specified string |
| sscanf | read formatted input from the specified string |
| tmpfile | create a temporary file |
| tmpnam | return a valid, unused filename |
| ungetc | push a character back into the specified file |
| vfprintf | write formatted output to specified file using variable arguments |
| vprintf | write formatted output to standard output using variable arguments |
| vsprintf | write formatted output to the specified string using variable arguments |

Table 2-5: Standard I/O functions

2.11.4 STRING MANIPULATION FUNCTIONS

String manipulation functions copy and test character strings in memory. The include file `string.h` contains external declarations for these functions. A summary of string manipulation functions appears in the table below:

| Name | Definition |
|-----------------------|-----------------------------------------------------------|
| <code>memchr</code> | search for a character in a buffer |
| <code>memcmp</code> | compare two buffers for lexical order |
| <code>memcpy</code> | copy one buffer to another |
| <code>memmove</code> | copy characters |
| <code>memset</code> | propagate a fill character throughout a buffer |
| <code>strcat</code> | concatenate two strings |
| <code>strchr</code> | scan a string for the first occurrence of a character |
| <code>strcmp</code> | compare two strings for lexical order |
| <code>strcoll</code> | compare two strings according to the current locale |
| <code>strcpy</code> | copy one string to another |
| <code>strcspn</code> | find the end of a span of characters in a set |
| <code>strerror</code> | map the error number to an error message |
| <code>strlen</code> | find the length of a string |
| <code>strncat</code> | concatenate two strings; append up to <i>n</i> characters |
| <code>strncmp</code> | compare two strings, up to <i>n</i> characters |
| <code>strncpy</code> | copy <i>n</i> length string |
| <code>strpbrk</code> | find occurrence in string of character in set |
| <code>strrchr</code> | scan string for the last occurrence of a character |
| <code>strspn</code> | find the end of a span of characters not in a set |



| Name | Definition |
|---------|--------------------------------------------------|
| strstr | find the first instance of a string |
| strtok | break string into tokens |
| strxfrm | transform string according to the current locale |

Table 2-6: String manipulation functions

2.11.5 NON-LOCAL GOTO FUNCTIONS

Non-local goto functions are used to define and restore an “environment.” In this implementation, an “environment” consists of a set of values in the non-volatile machine registers. These functions are useful for dealing with errors and interrupts encountered in a low-level subroutine of a program. The include file `setjmp.h` contains external declarations for non-local goto functions. A summary of the functions appears below:

| Name | Definition |
|---------|-------------------------------------------------------------------|
| longjmp | returns an environment established earlier by <code>setjmp</code> |
| setjmp | establishes an environment for later use by <code>longjmp</code> |

Table 2-7: Non-local Goto functions

2.11.6 DATE AND TIME ROUTINES

A summary of date and time functions appears below. The include file `time.h` contains external declarations for date and time routines. The following table summarizes date and time functions:

| Name | Definition |
|------------------------|-----------------------------------------------------|
| <code>asctime</code> | convert time to a string |
| <code>clock</code> | return the processor time used |
| <code>ctime</code> | convert calendar time to local time |
| <code>difftime</code> | compute the difference between two times |
| <code>gmtime</code> | convert calendar time to Coordinated Universal Time |
| <code>localtime</code> | convert calendar time to local time |
| <code>mktime</code> | convert broken-down time into calendar time |
| <code>strftime</code> | put characters into an array |
| <code>time</code> | return the current calendar time |

Table 2-8: Date and time routines

2.11.7 ASCII CHARACTER SET MACROS AND FUNCTIONS

ASCII character set macros are of two general types: “is” and “to.”

Macros with the prefix `is` take a character type parameter and evaluate to 0 or 1, acting as predicates. These macros can be used in `if`, `while` and `for` constructs. Macros with the prefix `to` convert between upper and lower-case.

The library include file `ctype.h` contains the ASCII character set macro and function definitions. A summary appears below:

| Name | Definition |
|----------|----------------------------------------------|
| isalnum | test for alphanumeric character |
| isalpha | test for alphabetic character |
| isctrl | test for control character |
| isdigit | test for digit |
| isgraph | test for graphic character |
| islower | test for lowercase character |
| isprint | test for printing character |
| ispunct | test for punctuation character |
| isspace | test for whitespace character |
| isupper | test for uppercase character |
| isxdigit | test for hexadecimal digit |
| tolower | convert character to lowercase, if necessary |
| toupper | convert character to uppercase, if necessary |

Table 2-9: ASCII character set macros and functions

2.11.8 GLOBAL DEFINITIONS

The include file `stddef.h` contains global definitions for use in C programs. `stddef.h` defines one macro whose summary appears below:

| Name | Definition |
|----------|--------------------------------------|
| offsetof | return offset of member in structure |

Table 2-10: Global definitions

2.11.9 COMPILE-TIME ASSERTIONS

The include file `assert.h` defines the macro `assert` which allows compile-time testing of expressions. If the expression is false, the `assert` macro writes information about the call that failed on the standard error file and then aborts the program. A summary of the `assert` macro appears below:

| Name | Definition |
|---------------------|----------------------------|
| <code>assert</code> | check run-time expressions |

Table 2-11: Compile-time assertions

2.11.10 FORMATTING OF NUMERIC VALUES

Two functions, `setlocale` and `localeconv`, allow the setting of defaults for the formatting of numeric values that depend on country and/or language. Their external declarations are available in the library include file `locale.h`. A table summarizing these functions appears below:

| Name | Definition |
|-------------------------|------------------------------------------|
| <code>localeconv</code> | set numeric formatting values for locale |
| <code>setlocale</code> | select or query the current locale |

Table 2-12: Formatting of numeric values

2.11.11 VARIABLE LENGTH ARGUMENT LIST ACCESS

The argument list macros facilitate the access of items on a variable length argument list. The macros are defined in the library include file `stdarg.h` and are summarized in the table below:

| Name | Definition |
|-----------------------|------------------------------------------------------------|
| <code>va_arg</code> | retrieve the next item on a variable length argument list. |
| <code>va_end</code> | finish access of variable length argument list |
| <code>va_start</code> | prepare to access a variable length argument list |

Table 2-13: Variable length argument list access

2.11.12 SIGNAL HANDLING

The library include file `signal.h` contains external declarations for dealing with internal and external events, or signals. A table summarizing these functions appears below:

| Name | Definition |
|---------------------|------------------------------------------|
| <code>raise</code> | generate a signal |
| <code>signal</code> | designate a function as a signal handler |

Table 2-14: Signal handling

2.11.13 C LIBRARY EXTENSIONS

Several extensions to the ANSI C library are available for use. These functions and macros are distributed with TASKING compilers for use in cases where their functionality may be helpful for embedded systems programming. Note however, that these functions are **not** part of the ANSI C standard, and their use may cause portability problems. The pre-processor macro `_EXTENSIONS` must be defined to make these external function declarations and macro definitions available. These functions are found in `extended.h`. The functions and macros are summarized in the following table.

| Name | Definition |
|------------------------|---------------------------------------------------------------------------------------------------------|
| <code>__tolower</code> | a macro to convert character to lowercase. The character must be a valid uppercase ASCII letter. |
| <code>__toupper</code> | a macro to convert character to uppercase. The character must be a valid lowercase ASCII letter. |
| <code>atanh</code> | inverse hyperbolic tangent |
| <code>log2</code> | base 2 logarithm |
| <code>getl</code> | direct long integer input |
| <code>getw</code> | direct integer input |
| <code>putl</code> | direct long integer output |
| <code>putw</code> | direct integer output |
| <code>memccpy</code> | copy memory up to marker character |

Table 2-15: C library extensions

Standard UNIX I/O Functions

Other extensions to the ANSI C library are the include files `unistd.h` and `fcntl.h`. This functions contain external declarations of the standard UNIX I/O functions. They are implemented using the file system simulation feature of CrossView Pro. A summary of these functions appears below:

| Name | Definition |
|---------------------|-------------------------------------------------|
| <code>access</code> | check the permissions of a file on the host |
| <code>chdir</code> | change the current directory on the host |
| <code>close</code> | close a file on the host |
| <code>getcwd</code> | retrieve the current directory on the host |
| <code>lseek</code> | seek in a file on the host |
| <code>open</code> | open a file on the host |
| <code>read</code> | read a sequence of characters from a file |
| <code>rename</code> | rename a file on the host |
| <code>stat</code> | <code>stat()</code> a file on the host platform |
| <code>unlink</code> | remove a file from the host |
| <code>write</code> | write a sequence of characters to a file |

Table 2-16: Standard UNIX I/O functions

Other include files

Three additional include files are distributed in ANSI C compliant TASKING compiler releases. These library include files contain definitions only, they do not contain any external function declarations. The files are: `errno.h` which contains variables used to process errors in the C language, `limits.h` which contains various defined CPU specific limitations, and `float.h` which defines various floating-point hardware limits and values.

In addition, there are several include files distributed that are used only in conjunction with building libraries. Under normal circumstances, these library include files will not be used. However, if the embedded application requires changes to the run-time library source, you **may** need to modify these files as required.

2.12 RUN-TIME LIBRARY ROUTINES

The following pages describe each of the run-time routines in reference format. The descriptions are in alphabetical order by function name.

__tolower

```
#define _EXTENSIONS
#include <stdio.h>
#define __tolower(c) ((c) + 'a' - 'A')
```

`__tolower` is a macro which converts an uppercase letter, `c`, to its lowercase equivalent. `__tolower` returns the corresponding lowercase character.

`__tolower` should only be used when `c` is known to be an uppercase letter (presumably checked via the `isupper` macro). Unlike the ANSI `tolower` function, `__tolower` is not guaranteed to return its argument if the argument is not an uppercase character.

`__tolower` is a TASKING extension.

__toupper

```
#define _EXTENSIONS
#include <stdio.h>
#define __toupper(c) ((c) + 'A' - 'a')
```

`__toupper` is a macro which converts a lowercase letter, `c`, to its uppercase equivalent. `__toupper` returns the corresponding uppercase character.

`__toupper` should only be used when `c` is known to be a lowercase letter (presumably checked via the `islower` macro). Unlike the ANSI `toupper` function, `__toupper` is not guaranteed to return its argument if its argument is not a lowercase letter.

`__toupper` is a TASKING extension.

abort

```
#include <stdlib.h>
void abort(void);
```

`abort` raises a SIGABRT condition.

abs

```
#include <stdlib.h>
int abs(int x);
```

`abs` calculates $|x|$, the absolute value of the integer argument, x .

`abs` returns its input if x is the most negative int value.

access

```
#include <unistd.h>
int access( const char * name, int mode );
```

Use the file system simulation feature of CrossView Pro to check the permissions of a file on the host. `mode` specifies the type of access and is a bit pattern constructed by a logical OR of the following values:

| | |
|------|-------------------------------------|
| R_OK | Checks read permission. |
| W_OK | Checks write permission. |
| X_OK | Checks execute (search) permission. |
| F_OK | Checks to see if the file exists. |

`access` returns zero if successful or -1 on error.

acos

```
#include <math.h>
double acos(double x);
```

`acos` computes the value whose cosine is x . The inverse cosine is in radians and in the range from zero to π .

If x is outside the range $[-1,1]$, `acos` sets `errno` to `EDOM`. The return value is undefined in this case.

asctime

```
#include <time.h>
char * asctime(const struct tm *timeblock);
```

The function `asctime` converts the time stored in *timeblock* into a 26 character string of the form:

```
Mon Jan 01 01:01:01 1999\n\0
```

`asctime` returns a pointer to the character string.

The string returned by `asctime` may be overwritten by subsequent calls to `asctime` or `ctime`.

asin

```
#include <math.h>
double asin(double x);
```

`asin` computes the value whose sine is x . The inverse sine is in radians and is in the range from $-\pi/2$ to $\pi/2$.

If x is outside the range $[-1,1]$, `asin` sets `errno` to `EDOM`. The return value is undefined in this case.

assert

```
#include <assert.h>
void assert(int expression);
```

The `assert` macro puts diagnostics into programs. When it is executed, if *expression* is false the `assert` macro writes information about the particular call that failed, including the text of the argument, the name of the source file, and the source line number. It then aborts the program by using `abort()`.

If the preprocessor variable `NDEBUG` is defined before including `assert.h`, the `assert` macro will have no effect.

atan

```
#include <math.h>
double atan(double x);
```

`atan` computes the value whose tangent is x . The inverse tangent is in radians, and is in the range from $-\pi/2$ to $\pi/2$.

atan2

```
#include <math.h>
double atan2(double x, double y);
```

`atan2` computes the principal value of the arctangent of x/y , using the signs of both arguments to determine the quadrant of the return value. If both x and y are zero, `errno` is set to `EDOM` and $\pi/2$ is returned.

The return value is in radians and is in the range $-\pi$ to π . Here is a chart showing how the sign of the arguments determines the range of the return value:

| <u>Sign of arguments</u> | <u>Return range</u> |
|--------------------------|---------------------|
| $x < 0, y < 0$ | $-\pi$ to $-\pi/2$ |
| $x < 0, y > 0$ | $-\pi/2$ to 0 |
| $x > 0, y > 0$ | 0 to $\pi/2$ |
| $x > 0, y < 0$ | $\pi/2$ to π |

atanh

```
#define _EXTENSIONS
#include <math.h>
double atanh(double x);
```

`atanh` computes the inverse hyperbolic tangent of x .

`atanh` is a TASKING extension.

atexit

```
#include <stdlib.h>
int atexit(void (*func)(void));
```

The `atexit` function causes the function pointed to by *func* to be called without arguments at normal program termination. Up to 32 functions can be remembered. `atexit` returns 0 if successful, and non-zero if it fails.

The functions are called in a last in, first out basis. Functions may be recorded more than once. Normal termination means `exit` was called.

atof

```
#include <stdlib.h>
double atof(const char *s);
```

`atof` converts the ASCII string *s* to a double and returns the new value. The ASCII string is examined with respect to the following pattern: any number of leading white-space characters (as specified by the `isspace` function), an optional plus or minus sign, any number of decimal digits, an optional decimal point followed by any number of decimal digits, an optional “e” or “E”, an optional plus or minus sign or blank, and any number of decimal digits. Input which matches this pattern is converted into the double return value. Input after the end of the pattern is ignored.

In case of error, `atof` is not required to set `errno`, and its return value is undefined. In this implementation, `atof` is implemented via `strtod`, as follows:

```
strtod(s, (char **) NULL)
```

atoi

```
#include <stdlib.h>
int atoi(const char *s);
```

`atoi` converts the ASCII string *s* into an integer. The ASCII string is examined with respect to the following pattern: any number of leading white-space characters (as specified by the `isspace` function), an optional plus or minus sign, and any number of decimal digits. Characters which match this pattern are converted to the integer return value. Characters after the end of the pattern are ignored. `atoi` returns the converted integer value. Except in the case of errors, `atoi` is equivalent to:

```
(int) strtol(s, (char **) NULL, 10)
```

atol

```
#include <stdlib.h>
long atol(const char *s);
```

`atol` converts the ASCII string *s* to a long integer. The ASCII string is examined with respect to the following pattern: any number of white-space characters (as defined by the `isspace` function), an optional plus or minus sign, and any number of decimal digits. Characters which match this pattern are converted to the long return value. Characters after the end of the pattern are ignored. `atol` returns the converted long integer.

Overflows are ignored. Except for the behavior on errors, `atol` is equivalent to:

```
strtol(s, (char **)NULL, 10)
```

bsearch

```
#include <stdlib.h>
void *bsearch(const void *key, const void *base,
              size_t nummembr, size_t size,
              int (*comp)(const void *, const void *));
```

`bsearch` searches an array of *nummembr* objects, the first element of which is pointed to by *base*, for an element that matches the object pointed to by *key*. The size of each array element is specified by *size*.

The function pointed to by *comp* is called with two arguments that point to the key object and to an array element, in that order. The *comp* function shall return an integer less than, equal to, or greater than zero if the *key* object is considered to be less than, equal to, or to be greater than the array element. The array should consist of all elements that compare less than, all elements that compare equal to, and all elements that compare greater than the *key* object, in that order. `bsearch` returns a pointer to the item if found, or `NULL` if not found.

calloc

```
#include <stdlib.h>
void *calloc(size_t nmemb, size_t size);
```

`calloc` allocates memory for the *nmemb* elements of size *size* and returns a pointer to the start of the allocated memory space. Allocated memory is initialized to zero.

If memory is exhausted, `calloc` returns a null pointer.

ceil

```
#include <math.h>
double ceil(double x);
```

`ceil` computes and returns the smallest integer value greater than or equal to *x*, expressed as a double value.

chdir

```
#include <unistd.h>
int chdir( const char *path );
```

Use the file system simulation feature of CrossView Pro to change the current directory on the host to the directory indicated by *path*.

chdir returns zero if successful or -1 on error.

clearerr

```
#include <stdio.h>
void clearerr(FILE *str);
```

This function clears the end-of-file and error indicators for the stream pointed to by *str*.

Users should implement this routine themselves.

clock

```
#include <time.h>
clock_t clock(void);
```

clock returns the processor time used. If the processor time is unavailable, the return value is (clock_t)-1.

Users should implement this routine themselves. The current return value is (clock_1)-1.

close

```
#include <unistd.h>
int close( int fd );
```

File close function. The given file descriptor should be properly closed. This function calls the low-level routine *_close*.

close returns zero if successful or -1 on error.

cos

```
#include <math.h>
double cos(double x);
```

`cos` computes the cosine of x , expressed in radians.

cosh

```
#include <math.h>
double cosh(double x);
```

`cosh` computes the hyperbolic cosine of x .

ctime

```
#include <time.h>
char *ctime(const time_t *time);
```

The `ctime` function converts the calendar time pointed to by *time* to local time in the form of a string.

This function is equivalent to:

```
asctime(localtime(time))
```

The value returned by `ctime` will be overwritten by the next call to `ctime` or `asctime`.

difftime

```
#include <time.h>
double difftime(time_t time1, time_t time2);
```

`difftime` computes the difference between two times: *time1* and *time2*. `difftime` returns the difference in seconds, expressed as a double.

div

```
#include <stdlib.h>
div_t div(int numerator, int denominator);
```

`div` computes the quotient and remainder of the numerator divided by the denominator. The type `div_t` is a structure that contains two components, `quot` and `rem`, both of type `int`.

When *denominator* equals zero `div` returns zeroes in *quot* and *rem*.

If the result cannot be represented, the behavior is undefined. Otherwise $quot * denominator + rem$ shall equal *numerator*.

exit

```
#include <stdlib.h>
void exit(int status);
```

`exit` caused normal program termination. `exit` never returns to its caller. First, any functions recorded by `atexit` are called, in reverse order of their presentation via `atexit`. Next, all open output streams are flushed, all open streams are closed, and all files created by the `tmpfile` function are removed. Finally, the return status is returned to the host environment.

`exit` calls `__exit` to return to the host environment. The `__exit` routine must be coded by the user.

exp

```
#include <math.h>
double exp(double x);
```

`exp` computes the exponential function of *x*.

On overflow `exp` returns an IEEE infinity.

fabs

```
#include <math.h>
double fabs(double x);
```

`fabs` computes $|x|$, the absolute value of the double *x*.

fclose

```
#include <stdio.h>
int fclose(FILE *stream);
```

`fclose` flushes any buffers for the named *stream* and causes the file to be closed. The return value indicates the presence of an I/O error.

`fclose` must be implemented by the user.

feof

```
#include <stdio.h>
int feof(FILE *str);
```

This function tests the end-of-file indicator for the stream pointed to by *str*. `feof` returns nonzero only if the end-of-file indicator is set for *str*.

`feof` must be implemented by the user.

ferror

```
#include <stdio.h>
int ferror(FILE *stream);
```

This routine tests the error indicator for the the stream pointed to by *stream*. `ferror` returns nonzero only if the error indicator is set for *stream*.

`ferror` must be implemented by the user.

fflush

```
#include <stdio.h>
int fflush(FILE *stream);
```

`fflush` forces out any buffered output on an I/O channel. `fflush` returns zero, if successful. If an I/O error was encountered, `fflush` sets the global integer `errno` to indicate the error code and returns negative one.

The release version of `fflush` is a stub routine which always returns zero. The standard I/O library does not buffer output.

fgetc

```
#include <stdio.h>
int fgetc(FILE *stream);
```

fgetc returns the next character from the named input stream pointed to by *stream*.

fgetpos

```
#include <stdio.h>
int fgetpos(FILE *stream, fpos_t *pos);
```

fgetpos stores the current value of the file position indicator for the stream pointed to by *stream* in the object pointed to by *pos*. The information stored can be used by *fsetpos* for resetting the stream to the time of the call to *fgetpos*. *fgetpos* returns zero on success; on failure *fgetpos* returns a nonzero number and sets *errno*.

fgetpos must be implemented by the user.

fgets

```
#include <stdio.h>
char *fgets(char *s, int n, FILE *stream);
```

fgets reads characters from the specified input *stream* into the string *s* until either *n*-1 characters are read, end-of-file is reached, or a newline character is read. If a newline character is read, it is retained. The stream is then terminated by a null. The associated file pointer is incremented by the number of bytes read.

If end-of-file is reached before any characters have been read, *s* remains unchanged, and *fgets* returns NULL.

floor

```
#include <math.h>
double floor(double x);
```

floor computes the largest integer value less than or equal to *x*, expressed as a double value.

fmod

```
#include <math.h>
double fmod(double x, double y);
```

`fmod` calculates the floating-point remainder of x/y . That is, `fmod` returns the value $x - i*y$, for some integer i such that the result has the same sign as x and magnitude less than the magnitude of y . If y is zero, `fmod` returns zero under software floating-point and IEEE NaN (Not a Number) under hardware floating-point.

fopen

```
#include <stdio.h>
FILE *fopen(const char *filename, const char *mode);
```

`fopen` opens the file *filename* and returns an associated input or output stream, depending on *mode*.

`fopen` must be implemented by the user.

fprintf

```
#include <stdio.h>
int fprintf(FILE *stream, const char *format, ...);
```

See `printf`.

`fprintf` is identical to `printf`, but directs its output to the specified output *stream*.

fputc

```
#include <stdio.h>
int fputc(int c, FILE *stream);
```

`fputc` writes a character specified by c (converted to an unsigned char), onto the named output *stream*. `fputc` returns the character written.

fputs

```
#include <stdio.h>
int fputs(const char *s, FILE *stream);
```

fputs copies the string *s* to the specified output *stream*. The terminating null byte is not copied. The *fputs* function returns EOF if a write error occurs; otherwise it returns a non-negative value.

fread

```
#include <stdio.h>
size_t fread(void *ptr, size_t size,
             size_t n, FILE *stream);
```

fread reads *n* items of size *size* from the specified input *stream* into a buffer at *ptr*. *fread* returns the number of items successfully read, which may be less than *n* if a read error or end-of-file is encountered. If *size* or *n* is zero, *fread* returns zero and the contents of the array and the state of the stream remain unchanged.

free

```
#include <stdlib.h>
void free(void *ptr);
```

free deallocates storage allocated by previous *malloc*, *calloc*, or *realloc* routines.

In ANSI C, *free* replaces the Unix *cfree* routine.

freopen

```
#include <stdio.h>
FILE *freopen(const char *filename,
              const char *mode, FILE *stream);
```

freopen closes the specified stream and then opens it in the same way that *fopen* does. Failure to close the file is ignored. The error and end-of-file indicators for the stream are cleared. *freopen* returns a null pointer if not successful. Upon success *freopen* returns *stream*.

`freopen` must be implemented by the user. The primary use of `freopen` is to change the file associated with the standard text streams `stdin`, `stdout`, and `stderr`.

frexp

```
#include <math.h>
double frexp(double x, int *eptr);
```

`frexp` breaks a floating-point number into a normalized fraction and an integral power of two. It stores the integer in the `int` object pointed to by `eptr` and returns the fraction.

After calling the function

```
y = frexp(x, &n)
```

the following identity is true:

$$x = y * 2^n.$$

fscanf

```
#include <stdio.h>
int fscanf(FILE *stream,
const char *format, ...);
```

See `scanf`.

`fscanf` is identical to `scanf`, except that it reads its input from the specified input *stream*.

fseek

```
#include <stdio.h>
int fseek(FILE *stream, long int offset, int mode);
```

`fseek` sets the file position indicator for the stream pointed to by *stream*. `fseek` returns zero on success and nonzero on failure.

For binary streams, `fseek` calculates the new position by adding the *offset* (in bytes) to the position specified by *mode*. The specified position is the beginning of the file if *mode* is `SEEK_SET`, the current file position if *mode* is `SEEK_CUR`, or end of file if *mode* is `SEEK_END`.

For text streams, *offset* is either zero or a value returned by a previous call to `ftell`, using the same stream, and *mode* is set to `SEEK_SET`.

`fseek` must be implemented by the user.

fsetpos

```
#include <stdio.h>
int fsetpos(FILE *stream, const fpos_t *pos);
```

`fsetpos` sets the file position indicator for the stream pointed to by *stream* to the value stored in *pos*. It also clears the end-of-file indicator for *stream*. `fsetpos` returns zero if successful, and returns a nonzero and sets `errno` to a positive value if unsuccessful.

`fsetpos` must be implemented by the user.

ftell

```
#include <stdio.h>
long int ftell(FILE *stream);
```

`ftell` returns the current file position indicator for the stream pointed to by *stream*. For binary streams this is the number of characters since the beginning of the file. For text streams this is some value that is meaningful to `fseek`, but does not necessarily have any relation to the number of characters. On failure `ftell` returns `-1L` and stores a positive value in `errno`.

`ftell` must be implemented by the user.

fwrite

```
#include <stdio.h>
size_t fwrite(const void *ptr, size_t size,
              size_t n, FILE *stream);
```

`fwrite` writes *n* items of size *size* from a buffer at address *ptr* to the specified output *stream*. `fwrite` returns the number of items written.

fgetc

```
#include <stdio.h>
int getc(FILE *stream);
```

`getc` reads a character from the specified input *stream* and increments the associated file pointer by one byte. `getc` returns the next character from the specified input *stream*.

If end-of-file is encountered or a read error occurs, `getc` returns EOF.

getchar

```
#include <stdio.h>
int getchar(void);
```

`getchar` is a macro defined as `getc(stdin)`. `getchar` returns the next character from standard input.

getcwd

```
#include <unistd.h>
char * getcwd( char * buf, size_t size );
```

Use the file system simulation feature of CrossView Pro to retrieve the current directory on the host.

`getcwd` returns the directory name if successful or NULL on error.

getenv

```
#include <stdlib.h>
char *getenv(const char *name);
```

The `getenv` function searches an environment list, provided by the host environment, for the environment variable *name*. It returns the variable definition, or NULL if no definition exists.

`getenv` must be implemented by the user.

getl

```
#define _EXTENSIONS
#include <stdio.h>
long getl (FILE * stream);
```

`getl` reads and returns a long from the specified file *stream*.

`getl` is a TASKING extension. The first byte read is the high-order byte of the result.

gets

```
#include <stdio.h>
char *gets(char *s);
```

`gets` reads a string from standard input (stdin) into *s*. The read terminates when a newline character is read or end-of-file is encountered. Any newline character is then discarded and the string is terminated with a null. `gets` returns *s*.

`fgets` does not discard the newline character. If end-of-file is encountered and no characters have been read, `gets` returns NULL.

getw

```
#define _EXTENSIONS
#include <stdio.h>
int getw(FILE *stream);
```

`getw` reads and returns an integer from the specified input *stream*.

`getw` is a TASKING extension. The first byte read is the high order byte of the result.

gmtime

```
#include <time.h>
struct tm *gmtime(const time_t *time);
```

`gmtime` converts a time value represented by a `time_t` into a time value represented as a `tm` structure, expressed as Coordinated Universal Time (UTC). If UTC is not available, then `gmtime` returns `NULL`.

This function must be implemented by the user. The current return value is `NULL` pointer.

isalnum

```
#include <ctype.h>
int isalnum(int c);
```

`isalnum` tests whether `c` is an alphabetic character (either upper or lower case) or a decimal digit. If `c` is an alphanumeric character, `isalnum` returns `TRUE`. If `c` is not an alphanumeric character, `isalnum` returns `FALSE`.

If `c` is outside the range `[-1, 255]`, the result is undefined.

isalpha

```
#include <ctype.h>
int isalpha(int c);
```

`isalpha` tests whether `c` is an alphabetic character (either upper or lower case). If `c` is a letter, `isalpha` returns `TRUE`. If `c` is not a letter, `isalpha` returns `FALSE`.

If `c` is outside the range `[-1, 255]`, the result is undefined.

isctrl

```
#include <ctype.h>
int isctrl(int c);
```

`isctrl` tests whether `c` is a delete or control character. `isctrl` returns TRUE if `c` is a control character; FALSE if `c` is not a control character.

If `c` is outside the range `[-1, 255]`, the result is undefined. The control characters are those whose hex values are between 0 and 1F or equal to 7F.

isdigit

```
#include <ctype.h>
int isdigit(int c);
```

`isdigit` tests whether `c` is a decimal digit. `isdigit` returns TRUE if `c` is a digit; FALSE if `c` is not a digit.

If `c` is outside the range `[-1, 255]`, the result is undefined.

isgraph

```
#include <ctype.h>
int isgraph(int c);
```

`isgraph` tests whether `c` is a graphic character; that is, any printing character except a space. `isgraph` returns TRUE if `c` is a graphic character. `isgraph` returns FALSE if `c` is not a graphic character.

If `c` is outside the range `[-1, 255]`, the result is undefined. Graphic characters are those whose hex values are between 21 and 7E. The DEL character is not a graphic character.

islower

```
#include <ctype.h>
int islower(int c);
```

`islower` tests whether `c` is a lowercase alphabetic character. If `c` is a lowercase letter, `islower` returns TRUE. If `c` is not a lowercase letter, `islower` returns FALSE.

If `c` is outside the range `[-1, 255]`, the result is undefined.

isprint

```
#include <ctype.h>
int isprint(int c);
```

`isprint` tests whether `c` is any printable character, including space. `isprint` returns TRUE if `c` is a printable character; FALSE if `c` is not a printable character.

If `c` is outside the range `[-1, 255]`, the result is undefined. The printable characters are those whose hex values are between 20 and 7E.

ispunct

```
#include <ctype.h>
int ispunct(int c);
```

`ispunct` tests whether `c` is a punctuation character. Punctuation characters include any printable character except a space, a digit or a letter. If `c` is a punctuation character, `ispunct` returns TRUE. If `c` is not a punctuation character, `ispunct` returns FALSE.

If `c` is outside the range `[-1, 255]`, the result is undefined. The punctuation characters are those whose hex values are between 21 and 2f, 3A and 40, 5B and 60, or 7B to 7E.

isspace

```
#include <ctype.h>
int isspace(int c);
```

`isspace` tests whether *c* is a white-space character: a space, tab, carriage return, newline, vertical tab or formfeed. If *c* is a white-space character, `isspace` returns TRUE. If *c* is not a white-space character, `isspace` returns FALSE.

If *c* is outside the range $[-1, 255]$, the result is undefined. The white-space characters are those whose hex values are between 9 and D or equal to 20.

isupper

```
#include <ctype.h>
int isupper(int c);
```

`isupper` tests whether *c* is an upper-case alphabetic character. `isupper` returns TRUE if *c* is an upper-case letter. If *c* is not an upper-case letter, `isupper` returns FALSE.

If *c* is outside the range $[-1, 255]$, the result is undefined.

isxdigit

```
#include <ctype.h>
int isxdigit(int c);
```

`isxdigit` tests whether *c* is a hexadecimal digit, that is, in the set [0123456789abcdefABCDEF]. `isxdigit` returns TRUE if *c* is a hex digit; FALSE if *c* is not a hex digit.

If *c* is outside the range $[-1, 255]$, the result is undefined.

labs

```
#include <stdlib.h>
long labs(long x);
```

`labs` calculates $|x|$, the absolute value of the long integer *x*.

`labs` returns its input if x is the most negative long value.

ldexp

```
#include <math.h>
double ldexp(double x, int exp);
```

`ldexp` returns the product of x and 2 raised to the integer power exp . That is, `ldexp` returns the quantity $x \cdot (2^{exp})$.

`ldexp` returns IEEE infinity in case of overflow; zero in case of underflow.

ldiv

```
#include <stdlib.h>
ldiv_t ldiv(long int numerator, long int denominator);
```

`ldiv` computes the quotient and remainder of the division of the *numerator* by the *denominator*. The type *ldiv_t* is a structure that contains two components, `quot` and `rem`, both of type `long int`.

If the result cannot be represented the behavior is undefined. Otherwise `quot * denominator + rem` shall equal *numerator*.

localeconv

```
#include <locale.h>
struct lconv *localeconv(void);
```

`localeconv` returns a pointer to a filled-in structure that contains numeric formatting information for the current locale. The values in the structure cannot be changed by the program except by later calls to `localeconv` or on calls to `setlocale` that change the categories `LC_ALL`, `LC_MONETARY`, or `LC_NUMERIC`.

The locale is the ANSI C method of specifying culturally-dependent information. Currently the library supports only the default or “C” locale.

localtime

```
#include <time.h>
struct tm *localtime(const time_t *time);
```

`localtime` breaks down a time value expressed as a `time_t` into a time value expressed as a `tm` structure.

The `tm` structure pointed to by the return value may be overwritten by subsequent calls to `localtime` or `gmtime`.

log

```
#include <math.h>
double log(double x);
```

`log` computes the natural logarithm of x . If x is zero, then `log` returns IEEE negative infinity. If x is less than zero, `log` returns IEEE NaN (Not a Number) and sets `errno` to `EDOM`.

log2

```
#define _EXTENSIONS
#include <math.h>
double log2(double x);
```

`log2` computes the base 2 logarithm of the double x . If x is zero then `log2` returns IEEE negative infinity. If x is less than zero, `log2` returns IEEE NaN (Not a Number) and sets `errno` to `EDOM`.

`log2` is a TASKING extension.

log10

```
#include <math.h>
double log10(double x);
```

`log10` computes the base 10 logarithm of x . If x is zero, then `log10` returns IEEE negative infinity. If x is less than zero, `log10` returns IEEE NaN (Not a Number) and sets `errno` to `EDOM`.

longjmp

```
#include <setjmp.h>
void longjmp(jmp_buf x, int n);
```

`longjmp` returns to an environment established by `setjmp` using *n* as the return value, except that 1 is returned if *n* is zero. If `setjmp` was not invoked with this environment or if the function containing the invocation has terminated, then the results are undefined.

The values of non-volatile objects of automatic storage class (that is, non-volatile local variables) local to the function calling `setjmp` are indeterminate if they were modified between the `setjmp` and `longjmp` calls.

After `longjmp` is completed, program execution continues as if the corresponding invocation of `setjmp` had just returned *n* (or 1, if *n* is zero).

lseek

```
#include <unistd.h>
off_t lseek( int fd, off_t offset, int whence );
```

Moves read-write file offset. This function calls the low-level routine `_lseek`.

`lseek` returns the resulting pointer location if successful or -1 on error.

malloc

```
#include <stdlib.h>
void *malloc(size_t nwords );
```

`malloc` allocates space of the size *nwords* on the heap. Allocated memory is not initialized to zero.

If memory space is exhausted, `malloc` returns a null pointer.

mblen

```
#include <stdlib.h>
int mblen(const char *s, size_t n);
```

If *s* is a null pointer or points to a null character, then *mblen* returns 0. Otherwise, *mblen* returns the number of bytes comprising the multi-byte character *s*. If *s* points to an invalid multibyte character, then *mblen* returns -1.

mblen assumes the Shift JIS convention for Japanese character encoding. Values between 1 and 127 (hex 7F) are treated as one-byte ASCII codes. Values between 160 and 223 (hex A0 to DF) are treated as one-byte kana codes. Kanji characters are encoded as two-byte sequences where the first byte is between 129 and 159 (hex 81 to 9F) or 224 to 252 (hex E0 to FC) and the second byte is between 64 and 252 (hex 40 to FC).

Here is a summary:

| | |
|-------------------|---------------------------------------------------------------------------------------|
| ASCII (one byte) | 0 through 0x7F |
| Kana (one byte) | 0xA0 through 0xDF |
| Kanji (two bytes) | first byte: 0x81 through 0x9F and 0xE0 through 0xFC second byte: 0x40 through 0xFC |

mbstowcs

```
#include <stdlib.h>
size_t mbstowcs(wchar_t *pwcs,
const char *s, size_t n);
```

mbstowcs converts a sequence of multi-byte characters pointed to by *s* to wide characters, and stores no more than *n* of them in the array of wide characters pointed to by *pwcs*. It copies up to to *n* characters from *s* to *pwcs*, until it reaches a null character. If an invalid character is found, *mbstowcs* returns -1. Otherwise, the number of characters written is returned. The *mbstowcs* routine assumes the Shift JIS convention for Japanese characters. See *mblen* for more details.

mbstowc

```
#include <stdlib.h>
int mbtowc(wchar_t *pwc, const char *s, size_t n);
```

mbtowc converts the multi-byte character pointed to by *s* to a wide character, and stores the wide character in the location pointed to by *pwc*. It returns 0 if either *s* is null or points to the null character and returns -1 if the character is invalid. Otherwise, if *pwc* is not null, the character at *s* is stored at *pwc* and *mbtowc* returns 1 if the character is a one-byte ASCII or kana code and returns 2 if it is a two-byte Kanji code. The *mbstowcs* routine assumes the Shift JIS convention for Japanese characters. See *mblen* for more details.

memccpy

```
#include <extended.h>
void *memccpy(void *s1, void *s2, char c, size_t n);
```

memccpy copies characters from *s2* to *s1*, stopping after the first occurrence of character *c* has been copied, or after *n* characters have been copied, whichever comes first. *memccpy* returns a pointer to the character after the copy of *c* in *s1*, or NULL if *c* was not encountered.

memccpy is a TASKING extension.

memchr

```
#include <string.h>
void *memchr(const void *s, int c, size_t n);
```

memchr searches an *n* word memory area at address *s* for the character *c*. *memchr* returns a pointer to the first occurrence of *c*. If *c* is not found, *memchr* returns NULL.

memchr is equivalent to *strchr*, except it does not stop at nulls.

memcmp

```
#include <string.h>
int memcmp(const void *s1, const void *s2, size_t n);
```

`memcmp` compares n words of memory at addresses $s1$ and $s2$. `memcmp` returns zero if the memory areas are equal, an integer greater than zero if $s1$ is lexically larger than $s2$; else `memcmp` returns an integer less than zero.

`memcmp` is equivalent to the `strcmp` routine, except it does not stop at nulls.

memcpy

```
#include <string.h>
void *memcpy(void *s1, const void *s2, size_t n);
```

`memcpy` copies n words of memory from $s2$ to $s1$. `memcpy` returns its first argument, $s1$.

`memcpy` is equivalent to `strncpy` except that it does not stop at nulls. If the two memory areas overlap then the results are undefined. See also `memmove`.

memmove

```
#include <string.h>
void *memmove(void *s1, const void *s2, size_t n);
```

`memmove` copies n words of memory from $s2$ to $s1$. If the two memory areas overlap, then the copy is done as if the characters are first copied from $s2$ into a temporary area of size n , and then copied to $s1$. `memmove` returns its first argument, $s1$.

memset

```
#include <string.h>
void *memset(void *s, int c, size_t n);
```

`memset` fills n words of memory at s with a fill character c and returns its first argument, s .

mktime

```
#include <time.h>
time_t mktime(struct tm *time);
```

`mktime` converts a time value, expressed as a `tm` structure, into a time value expressed as a `time_t`. The original values of the `tm_wday` and `tm_yday` components of the `tm` structure are ignored, and the other values are not restricted to their usual ranges.

On successful completion the values of the `tm_wday` and `tm_yday` components of the `tm` structure are set appropriately, and the other values are normalized to be in their usual ranges.

Some time values which can be expressed as `tm` structures cannot be expressed as `time_t` values. In that case `(time_t) -1` is returned.

modf

```
#include <math.h>
double modf(double x, double *intptr);
```

`modf` returns the fractional part of x and stores the integral part indirectly through the pointer `intptr`. In effect, this breaks the double x into an integer and fractional part. The breakdown into integer and fractional part is defined by truncation (round towards zero).

For example, the integer part of -3.9 is -3.0 , and the fractional part is $-.9$.

offsetof

```
#include <stddef.h>
size_t offsetof(type, member);
```

The macro `offsetof` returns the offset in bytes from the beginning of the structure `type` to the structure member `member`.

The offset macro is defined as follows:

```
#define    offsetof(type, member)
           (size_t) & (((type *)0) -> member)
```

open

```
#include <fcntl.h>
int open( const char * name, int flags );
```

Opens a file a file for reading or writing. This function calls the low-level routine `_open`.

`open` returns the file descriptor if successful (a non-negative integer), or `-1` on error.

perror

```
#include <stdio.h>
void perror(const char *s);
```

`perror` maps the error number in the integer expression *errno* to an error message. It writes a sequence of characters to the standard error stream as follows. First, if *s* is not a null pointer and does not point to a null character, the string pointed to by *s* is printed, followed by a colon and a space. Next, the appropriate error string is printed, followed by a newline.



`strerror`.

pow

```
#include <math.h>
double pow(double x, double y);
```

`pow` returns *x* raised to the *y* power.

If *x* is zero and *y* is non-positive, then IEEE infinity is returned and `errno` is set to `EDOM`.

If *x* is negative and *y* is not an integer, then IEEE infinity is returned and `errno` is set to `EDOM`.

To avoid errors, *x* must be greater than or equal to zero, unless *y* is an integer.

printf

```
#include <stdio.h>
int printf(const char *format, ...);
```

`printf` converts and formats its arguments and prints them to `stdout`, following specifications of *format*. *format* may contain ordinary characters, which are simply copied to standard output, and conversion specifications, each of which causes conversion and printing of the next successive argument to `printf`.

The following description is taken from the ANSI C standard.

Each conversion specification is introduced by the character `%`. After the `%`, the following appear in sequence:

- Zero or more flags (in any order) that modify the meaning of the conversion specification.
- An optional minimum field width. If the converted value has fewer characters than the field width, it will be padded with spaces (by default) to the left (or right, if the left adjustment flag, described later, has been given) to the field width. The field width takes the form of an asterisk `*` (described later) or a decimal integer. Note that zero (0) is taken as a flag, not as the beginning of a field width.
- An optional precision that gives the minimum number of digits to appear for the `d`, `i`, `o`, `u`, `x` and `X` conversion, the number of digits to appear after the decimal-point character for `e`, `E` and `f` conversions, the maximum number of significant digits for the `g` and `G` conversions, or the maximum number of characters to be written from a string in `s` conversion. The precision takes the form of a period (.) followed either by an asterisk `*` (described later) or by an optional decimal integer; if only the period is specified, the precision is taken as zero. If a precision appears with any other conversion specifier, the behavior is undefined.

- An optional `h` specifying that a following `d`, `i`, `o`, `u`, `x` or `X` conversion specifier applies to a `short int` or `unsigned short int` argument (the argument will have been promoted according to integral promotions, and its value shall be converted to `short int` or `unsigned short int` before printing); an optional `h` specifying that a following `n` conversion specifier applies to a pointer to a `short int` argument; an optional `l` specifying that a following `d`, `i`, `o`, `u`, `x` or `X` conversion specifier applies to a `long int` or `unsigned long int` argument; an optional `l` specifying that a following `n` conversion specifier applies to a pointer to a `long int` argument; or an optional `L` specifying that a following `e`, `E`, `f`, `g` or `G` conversion specifier applies to a `long double` argument. If an `h`, `l` or `L` appears with any other conversion specifier, the behavior is undefined.
- A character that specifies the type of conversion to be applied.

As noted above, a field width, or precision, or both, may be indicated by an asterisk. In this case, an `int` argument supplies the field width or precision. The arguments specifying field width, or precision or both, shall appear (in that order) before the argument (if any) to be converted. A negative field width argument is taken as a `-` flag followed by a positive field width. A negative precision argument is taken as if the precision were omitted.

The flag characters and their meanings are:

- `-` The result of the conversion will be left-justified within the field. (It will be right-justified if this flag is not specified.)
- `+` The result of a signed conversion will always begin with a plus or minus sign. (It will begin with a sign only when a negative value is converted if this flag is not specified).
- space* If the first character of a signed conversion is not a sign, or if a signed conversion results in no characters, a space will be prefixed to the result. If the space and `+` flags both appear, the space flag will be ignored.

- # The result is to be converted to an “alternate form.” For `o` conversion, it increases the precision to force the first digit of the result to be a zero. For `x` (or `X`) conversion, a nonzero result will have `0x` (or `0X`) prefixed to it. For `e`, `E`, `f`, `g` and `G` conversions, the result will always contain a decimal-point character, even if no digits follow it. (Normally, a decimal-point character appears in the result of these conversions only if a digit follows it.) For `g` and `G` conversions, trailing zeros will *not* be removed from the result. For other conversions, the behavior is undefined.
- 0 For `d`, `i`, `o`, `u`, `x`, `X`, `e`, `E`, `f`, `g` and `G` conversions, leading zeros (following any indication of sign or base) are used to pad to the field width; no space padding is performed. If the `0` and `-` flags both appear, the `0` flag will be ignored. For `d`, `i`, `o`, `u`, `x` and `X` conversions, if a precision is specified, the `0` flag will be ignored. For other conversions, the behavior is undefined.

The conversion specifiers and their meanings are:

- `d`, `i` The `int` argument is converted to signed decimal in the style `[-]dddd`. The precision specifies the minimum number of digits to appear; if the value being converted can be represented in fewer digits, it will be expanded with leading zeros. The default precision is 1. The result of converting a zero value with a precision of zero is no characters.
- `o`, `u`, `x`, `X` The unsigned `int` argument is converted to unsigned octal (`o`), unsigned decimal (`u`) or unsigned hexadecimal notation (`x` or `X`) in the style `dddd`; the letters `abcdef` are used for `x` conversion and the letters `ABCDEF` for `X` conversion. The precision specifies the minimum number of digits to appear; if the value being converted can be represented in fewer digits, it will be expanded with leading zeros. The default precision is 1. The result of converting a zero value with a precision of zero is no characters.

- f** The double argument is converted to decimal notation in the style *[-]ddd.ddd*, where the number of digits after the decimal-point character is equal to the precision specification. If the precision is missing, it is taken as 6; if the precision is zero and the # flag is not specified, no decimal-point character appears. If a decimal-point character appears, at least one digit appears before it. The value is rounded to the appropriate number of digits.
- e, E** The double argument is converted in the style *[-]d.ddd \pm dd*, where there is one digit before the decimal-point character (which is nonzero if the argument is nonzero) and the number of digits after it is equal to the precision; if the precision is missing it is taken as 6; if the precision is zero and the # flag is not specified, no decimal point character appears. The value is rounded to the appropriate number of digits. The **E** conversion specifier will produce a number with **E** instead of **e** introducing the exponent. The exponent always contains at least two digits. If the value is zero, the exponent is zero.
- g, G** The double argument is converted in style **f** or **e** (or in style **E** in the case of a **G** conversion specifier) with the precision specifying the number of significant digits. If the precision is zero, it is taken as 1. The style used depends on the value converted; style **e** (or **E**) will be used only if the exponent resulting from such a conversion is less than -4 or greater than or equal to the precision. Trailing zeros are removed from the fractional portion of the result; a decimal-point character appears only if it is followed by a digit.
- c** The `int` argument is converted to an unsigned `char`, and the resulting character is written.
- s** The argument shall be a pointer to an array of character type. (No special provisions are made for multibyte characters.) Characters from the array are written up to (but not including) a terminating null character. If the precision is specified, no more than that many characters are written. If the precision is not specified or is greater than the size of the array, the array shall contain a null character.

| | |
|---|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| p | The argument shall be a pointer to <code>void</code> . The value of the pointer is converted to a sequence of hex characters, just like the <code>x</code> format. |
| n | The argument shall be a pointer to an integer into which is <i>written</i> the number of characters written to the output stream so far by this call to <code>printf</code> . No argument is converted. |
| % | A % is written. No argument is converted. The complete conversion specification shall be <code>%%</code> . |

If a conversion specification is invalid, the behavior is undefined.

If any argument is, or points to, a union or an aggregate (except for an array of character type using `%s` conversion, or a pointer using `%p` conversion), the behavior is undefined.

In no case does a nonexistent or small field width cause truncation of a field; if the result of a conversion is wider than the field width, the field is expanded to contain the conversion result.

Notes

- `printf` uses its first argument to decide how many arguments follow and what their types are. `printf` normally returns the number of characters printed. If there are not enough arguments, or arguments are of the wrong type, the results are undefined.
- `printf` returns the number of characters printed, or a negative number if an output error occurred. No more than 509 characters shall be produced for any single conversion.
- `fprintf` and `sprintf` are identical to `printf`, except that `fprintf` directs its output to the specified output *stream*; `sprintf` directs its output to the string *s*.

putc

```
#include <stdio.h>
int putc(int c, FILE *stream);
```

`putc` writes the character *c* to the specified output *stream*.

putchar

```
#include <stdio.h>
int putchar(int c);
```

`putchar` is equivalent to `putc` with the second argument of *stdout*. `putchar` returns the character written. If an error occurs, `putchar` returns EOF.

putl

```
#define _EXTENSIONS
#include <stdio.h>
long putl(long l, FILE *stream);
```

`putl` writes a long, *l*, to the specified output *stream* and returns its first argument.

`putl` is equivalent to `putw` if integers are 32 bits. The first character written is the high-order byte of *l*.

`putl` is a TASKING extension.

puts

```
#include <stdio.h>
int puts(const char *s);
```

`puts` copies the string *s* to standard output (*stdout*). `puts` returns 0, the success code.

putw

```
#define _EXTENSIONS
#include <stdio.h>
int putw(int w, FILE *stream);
```

`putw` writes an integer *w* to the specified output *stream* and returns its first argument.

The first character written is the high-order byte of *w*.

qsort

```
#include <stdlib.h>
void qsort(void *base, size_t
           nummembr, size_t size,
           int (*comp)(const void *,
                       const void *));
```

`qsort` sorts an array of *nummembr* elements, the first element of which is pointed to by *base*. The size of each object is specified by *size*. The contents of the array are sorted into ascending order according to the comparison function pointed to by *comp*, which takes two arguments that point to the objects being compared. The comparison function should return an integer less than, equal to, or greater than zero if the first argument is considered to be less than, equal to, or greater than the second.

raise

```
#include <signal.h>
int raise(int sig);
```

The `raise` function sends the signal *sig* to the executing program. `raise` returns zero if successful, and nonzero if unsuccessful.

See `signal` for more details.

rand

```
#include <stdlib.h>
int rand(void);
```

`rand` computes and returns pseudo-random integers in the range [0, 32767]. The pseudo-random number is computed via a simple multiplicative congruence algorithm based on a “seed” value (initially one). At any time the seed value may be reset using the `srand` routine.

The low bits of the numbers generated are not very random; use the middle bits. Specifically, the lowest bit alternates between 0 and 1.

rcopy

```
#include <rcopy.h>
void rcopy(struct hdr *addr)
```

`rcopy` is a ROM initialization utility, called at the start of a new C program. Its argument *addr* is the address of an initialization segment created by the `llink` utility.

`rcopy` should be called at the start of a new program or during system restart. The argument to `rcopy` is typically the address of a fictitious external variable. The name of this variable is carefully chosen to match that generated by `llink` for the initialization segment itself. The name of the initialization segment can be determined by `llink` options. Note that the C compiler prepends an underscore to the names of external variables, so an external variable named `x` would match with an initialization segment named `_x`.

`rcopy` should be called once for each initialization segment created by `llink`. Refer to the *Linking Locator* chapter in the *User's Manual* for more details.

`rcopy` is a TASKING extension.

realloc

```
#include <stdlib.h>
void *realloc(void *ptr, size_t size);
```

`realloc` changes the size of the object pointed to by *ptr* to the size specified by *size*. The contents of the object will remain unchanged up to the lesser of the new and old sizes. If *ptr* is a null pointer, `realloc` behaves like the `malloc` function for the specified size. If *size* is zero and *ptr* is not a null pointer, the object is freed.

read

```
#include <unistd.h>
size_t read( int fd, char * buffer, size_t count );
```

Reads a sequence of characters from a file. This function calls the low-level routine `_read`, which interfaces to CrossView Pro's file system simulation.

`read` returns the number of characters read.

remove

```
#include <stdio.h>
int remove(const char *filename);
```

`remove` deletes *filename* and returns zero upon successful completion.

`remove` must be implemented by the user.

rename

```
#include <stdio.h>
int rename(const char *old, const char *new);
```

This routine renames the file *old* to the filename *new* and returns zero if successful.

`rename` is implemented using CrossView Pro's file system simulation.

rewind

```
#include <stdio.h>
void rewind(FILE *stream);
```

The `rewind` function sets the file position indicator for the stream pointed to by *stream* to the beginning of the file. It is equivalent to:

```
(void) fseek(stream, 0L, SEEK_SET)
```

- except that the error indicator for the stream is also cleared.
- The `rewind` function returns no value.
- `rewind` must be implemented by the user.

scanf

```
#include <stdio.h>
int scanf(const char *format, ...);
```

scanf reads input from standard input under control of the string pointed to by *format*. The format string specifies admissible input sequences and how they are to be converted for assignment, using subsequent arguments as pointers to the objects to receive the converted values. If there are insufficient arguments or the types of the arguments do not match the converted values the behavior is undefined. *scanf* returns EOF if an input failure occurs before any conversion. Otherwise *scanf* returns the number of items assigned.

The following description of the format specification is taken from the ANSI C standard.

The format is composed of zero or more directives: one or more white-space characters; an ordinary multibyte character (neither % nor a white-space character); or a conversion specification. Each conversion specification is introduced by the character %. After the %, the following appear in sequence:

- An optional assignment-suppressing character *.
- An optional nonzero decimal integer that specifies the maximum field width.
- An optional h, l or L indicating the size of the receiving object. The conversion specifiers d, i, and n shall be preceded by h if the corresponding argument is a pointer to `short int` rather than a pointer to `int`, or by l if it is a pointer to `long int`. Similarly, the conversion specifiers o, u, and x shall be preceded by h if the corresponding argument is a pointer to `unsigned short int` rather than a pointer to `unsigned int`, or by l if it is a pointer to `unsigned long int`. Finally, the conversion specifiers e, f, and g shall be preceded by l if the corresponding argument is a pointer to `double` rather than a pointer to `float`, or by L if it is a pointer to `long double`. If an h, l, or L appears with any other conversion specifier, the behavior is undefined.
- A character that specifies the type of conversion to be applied. The valid conversion specifiers are described below.

The `scanf` function executes each directive of the format in turn. If a directive fails, as detailed below, the `scanf` function returns. Failures are described as input failures (due to the unavailability of input characters) or matching failures (due to inappropriate input).

A directive composed of white-space character(s) is executed by reading input up to the first non-white-space character (which remains unread), or until no more characters can be read.

A directive that is an ordinary multibyte character is executed by reading the next character of the stream. If one of the characters differs from one comprising the directive, the directive fails, and the differing and subsequent characters remain unread.

A directive that is a conversion specification defines a set of matching input sequences, as described below for each specifier. A conversion specification is executed in the following steps:

Input white-space characters (as specified by the `isspace` function) are skipped, unless the specification includes a `[`, `c`, or `n` specifier. (These white-space characters are not counted against a specified field width.)

An input item is read from the stream, unless the specification includes an `n` specifier. An input item is defined as the longest matching sequence of input characters, unless that exceeds a specified field width, in which case it is the initial subsequence of that length in the sequence. The first character, if any, after the input item remains unread. If the length of the input item is zero, the execution of the directive fails: this condition is a matching failure, unless an error prevented input from the stream, in which case it is an input failure.

Except in the case of a `%` specifier, the input item (or, in the case of the `%n` directive, the count of input characters) is converted to a type appropriate to the conversion specifier. If the input item is not a matching sequence, the execution of the directive fails: this condition is a matching failure. Unless assignment suppression was indicated by a `*`, the result of the conversion is placed in the object pointed to by the first argument following the *format* argument that has not already received a conversion result. If this object does not have an appropriate type, or if the result of the conversion cannot be represented in the space provided, the behavior is undefined.

The following conversion specifiers are valid:

- d Matches an optionally signed decimal integer, whose format is the same as expected for the subject sequence of the `strtol` function with the value 10 for the *base* argument. The corresponding argument shall be a pointer to integer.
- i Matches an optionally signed integer, whose format is the same as expected for the subject sequence of the `strtol` function with the value 0 for the *base* argument. The corresponding argument shall be a pointer to integer.
- o Matches an optionally signed octal integer, whose format is the same as expected for the subject sequence of the `strtoul` function with the value 8 for the *base* argument. The corresponding argument shall be a pointer to unsigned integer.
- u Matches an optionally signed decimal integer, whose format is the same as expected for the subject sequence of the `strtoul` function with the value 10 for the *base* argument. The corresponding argument shall be a pointer to unsigned integer.
- x Matches an optionally signed hexadecimal integer, whose format is the same as expected for the subject sequence of the `strtoul` function with the value 16 for the *base* argument. The corresponding argument shall be a pointer to unsigned integer.
- e, f, g Matches an optionally signed floating-point number, whose format is the same as expected for the subject string of the `strtod` function. The corresponding argument shall be a pointer to floating.
- s Matches a sequence of non-white-space characters. The corresponding argument shall be a pointer to the initial character of an array large enough to accept the sequence and a terminating null character, which will be added automatically.

- [Matches a nonempty sequence of characters from a set of expected characters (the *scanset*). The corresponding argument shall be a pointer to the initial character of an array large enough to accept the sequence and a terminating null character, which will be added automatically. The conversion specifier includes all subsequence characters in the *format* string, up to and including the matching right bracket (]). The characters between the brackets (the *scanlist*) comprise the scanset, unless the character after the left bracket is a circumflex (^), in which case the scanset contains all characters that do not appear in the scanlist between the circumflex and the right bracket. If the conversion specifier begins with [] or [^], the right bracket character is in the scanlist and the next right bracket character is the matching right bracket that ends the specification; otherwise the first right bracket character is the one that ends the specification.
- c Matches a sequence of characters of the number specified by the field width (1 if no field width is present in the directive). The corresponding argument shall be a pointer to the initial character of an array large enough to accept the sequence. No null character is added. No special provisions are made for multibyte characters.
- p Matches a sequence of hexadecimal characters whose format is the same as expected for the subject sequence of the `strtoul` function with the value 16 for the base argument. The corresponding argument shall be a pointer to a pointer to void. This matches the `%p` conversion by `printf`.
- n No input is consumed. The corresponding argument shall be a pointer to integer into which is to be written the number of characters read from the input stream so far by this call to the `scanf` function. Execution of a `%n` directive does not increment the assignment count returned at the completion of execution of the `scanf` function.
- % Matches a single %; no conversion or assignment occurs. The complete conversion specification shall be %%.

If a conversion specification is invalid, the behavior is undefined.

The conversion specifiers `E`, `G` and `X` are also valid and behave the same as, respectively, `e`, `g` and `x`.

If end-of-file is encountered during input, conversion is terminated. If end-of-file occurs before any characters matching the current directive have been read (other than leading white space, where permitted), execution of the current directive terminates with an input failure; otherwise, unless execution of the current directive is terminated with a matching failure, execution of the following directive (if any) is terminated with an input failure.

If conversion terminates on a conflicting input character, the offending input character is left unread in the input stream. Trailing white-space (including new-line characters) is left unread unless matched by a directive. The success of literal matches and suppressed assignments is not directly determinable other than via the `%n` directive.

Notes

- The `scanf` function returns the value of the macro `EOF` if an input failure occurs before any conversion. Otherwise, the `scanf` function returns the number of input items assigned, which can be fewer than provided for, or even zero, in the event of an early matching failure.
- `scanf` is identical to the `fscanf` and `sscanf` routines, except `fscanf` reads its input from the specified input *stream*; `sscanf` reads its input from the string *s*.

setbuf

```
#include <stdio.h>
void setbuf(FILE *stream, char *buf);
```

`setbuf` may be used only after the stream pointed to by *stream* has been opened but before any other operation is performed on the stream. If *buf* is `NULL`, then it causes the stream to be unbuffered. Otherwise *buf* must point to an array of size `BUFSIZE` and *buf* is used as the buffer in a fully buffered file.

`setbuf` must be implemented by the user.

setjmp

```
#include <setjmp.h>
int setjmp(jmp_buf x);
```

`setjmp` establishes an environment for later use by `longjmp`. The type of the argument, `jmp_buf`, is provided in the `setjmp.h` include file.

See `longjmp` for more details.

setlocale

```
#include <locale.h>
char *setlocale(int category, const char *locale);
```

`setlocale` returns a pointer to a string that describes the new *locale*, or a null pointer if *locale* cannot be changed. The value of *category* must match the value of one of the macros defined in the header file `locale.h`. These macros begin with `LC_`.

If *locale* is a null pointer, `setlocale` does not change the current *locale*. If *locale* points to the string “C”, the new *locale* is the “C” *locale* for the category specified. If *locale* points to the string “”, the *locale* is the native *locale* for the category specified. *locale* can point to strings returned from previous calls to `setlocale`.

At program startup, the target environment acts as if the call `setlocale("LC_ALL", "C")` was called before it called *main*.

The supplied `setlocale` only works for the “C” *locale*.

setvbuf

```
#include <stdio.h>
int setvbuf(FILE *stream, char *buf,
            int mode, size_t size);
```

`setvbuf` sets the buffering mode for the stream pointed to by *stream*, according to *buf*, *mode* and *size*. If *buf* is not a null pointer, then *buf* is the address of the first element of an array of chars of size *size* that can be used as a stream buffer. *mode* must be one of the following macros: `_IOFBF` (full buffering), `_IOLBF` (line buffering) `_IONBF` (no buffering).

`setvbuf` must be called immediately after a call to `fopen` to associate a file with that stream.

`setvbuf` must be implemented by the user.

signal

```
#include <signal.h>
void (*signal(int sig, void (*func)(int)))(int);
```

The `signal` function allows a program to specify what action shall be taken upon receipt of the signal *sig*, which may be generated externally, or may be explicitly generated with the `raise` function. If the value of *func* is `SIG_DFL`, default handling for that signal will occur. If the value of *func* is `SIG_IGN`, the signal will be ignored. Otherwise, the signal handler pointed to by *func* will be called upon receipt of signal *sig*. If no errors are detected, `signal` returns the value of *func* for the most recent call to `signal` for the specified flag *sig*. Otherwise, a value of `SIG_ERR` is returned and a positive value is stored in `errno`.

Since most embedded applications will not want to abort when errors are detected, the library as distributed does not raise signals under any circumstances.

sin

```
#include <math.h>
double sin(double x);
```

`sin` computes the sine of *x*, expressed in radians.

sinh

```
#include <math.h>
double sinh(double x);
```

`sinh` computes the hyperbolic sine of *x*.

sprintf

```
#include <stdio.h>
int sprintf(char *s, const char *format, ...);
```

 `printf`.

`sprintf` is identical to `printf`, but directs its output to the string `s`. See `printf` for more details.

sqrt

```
#include <math.h>
double sqrt(double x);
```

`sqrt` computes the square root of x . If x is less than zero, `errno` is set to `EDOM` and an IEEE NaN (Not a Number) is returned.

srand

```
#include <stdlib.h>
void srand(unsigned int seed)
```

`srand` resets the seed for the random number generator `rand` with the value `seed`.

sscanf

```
#include <stdio.h>
int sscanf(const char *s,
           const char *format, ...);
```

See `scanf`.

`sscanf` is identical to `fscanf` and `scanf`, but reads its input from the string `s`. See `scanf` for more details.

stat

```
#include <unistd.h>
int stat( const char * name, struct stat * buf );
```

Use the file system simulation feature of CrossView Pro to `stat()` a file on the host platform. Returns zero if successful or `-1` on error.

strcat

```
#include <string.h>
char *strcat(char *s1, const char *s2);
```

`strcat` appends a copy of the string pointed to by `s2` (including the terminating null character) to the end of the string pointed to by `s1`. The initial character of `s2` overwrites the null characters at the end of `s1`.

`s1` must contain enough room to hold the resulting string.

strchr

```
#include <string.h>
char *strchr(const char *s, int c);
```

`strchr` looks for the first occurrence of a specific character, `c`, in a null terminated target string, `s`. `strchr` returns a pointer to the first character that matches `c`. If no character matches `c`, `strchr` returns `NULL`.

In ANSI C, `strchr` replaces the Unix `index` routine.

strcmp

```
#include <string.h>
int strcmp(char *s1, const char *s2);
```

`strcmp` compares the strings `s1` and `s2`, character by character, for lexical order in the character collating sequence. `strcmp` returns zero if the strings are equal; an integer greater than zero if `s1` is lexically larger than `s2`. If `s1` is lexically smaller than `s2`, `strcmp` returns an integer less than zero.

strcoll

```
#include <string.h>
int strcoll(const char *s1, const char *s2);
```

`strcoll` compares the string *s1* to the string *s2*, and with both strings interpreted as appropriate to the `LC_COLLATE` setting of the current locale. The routine returns an integer greater than, equal to, or less than zero, if *s1* is greater than, equal to, or less than *s2*, interpreted according to the locale.

In the “C” locale, `strcoll` is equivalent to `strcmp`.

strcpy

```
#include <string.h>
char *strcpy(char *s1, const char *s2);
```

`strcpy` copies the string *s2* and its terminating null to *s1* and returns *s1*.

s1 must contain enough room to hold the result.

strcspn

```
#include <string.h>
size_t strcspn(char *s1, const char *s2);
```

`strcspn` scans the string starting at *s1* for the first occurrence of a character in the string starting at *s2*. `strcspn` returns the length of the initial segment of *s1*, which consists entirely of characters not in *s2*.

strerror

```
#include <string.h>
char *strerror(int errnum);
```

`strerror` maps the error number in *errnum* to an error message which `strerror` returns. The message buffer is static and is overwritten by subsequent calls to `strerror`.

strftime

```
#include <time.h>
size_t strftime(char *s, size_t maxsize,
                const char *format,
                const struct tm *timeptr);
```

`strftime` puts characters into the array, *s*, as specified by the format string, *format*. The format string consists of zero or more conversion specifiers and ordinary characters. The conversion specifiers are given below. Any ordinary characters including the terminating null character are copied unchanged into the array. No more than *maxsize* characters are placed into the array. The values used in the conversions are contained in the structure *timeptr*.

If the total number of resulting characters is not more than *maxsize*, then `strftime` returns the number of characters copied into the array, *s*, otherwise `strftime` puts *maxsize* characters in the array *s* and returns the value zero.

Format specifiers:

| | |
|----|-----------------------------------------------------------------------|
| %a | is replaced by the locale's abbreviated weekday name. |
| %A | is replaced by the locale's full weekday name. |
| %b | is replaced by the locale's abbreviated month name. |
| %B | is replaced by the locale's full month name. |
| %c | is replaced by the locale's appropriate date and time representation. |
| %d | is replaced by the day of the month (01-31). |
| %H | is replaced by the hour of the day, 24 hour clock (00-23). |
| %I | is replaced by the hour of the day, 12 hour clock (01-12). |
| %j | is replaced by the day of the year (001-366). |
| %m | is replaced by the month (01-12). |
| %M | is replaced by the minute (00-59). |

| | |
|----|-------------------------------------------------------------------------------------------------------------------|
| %P | is replaced by the locale's AM/PM designation associated with a 12 hour clock. |
| %S | is replaced by the seconds (00–59). |
| %U | is replaced by the week number of the year, with the first Sunday of the year as the first day of week 1 (00–53). |
| %w | is replaced by the weekday, Sunday = 0, Saturday = 6. |
| %W | is replaced by the week number of the year, with the first Monday of the year as the first day of week 1 (00–53). |
| %x | is replaced by the date representation. |
| %X | is replaced by the time representation. |
| %y | is replaced by the year without the century (00–99). |
| %Y | is replaced by the year with the century. |
| %Z | is replaced by no characters. |
| %% | is replaced by %. |

strlen

```
#include <string.h>
size_t strlen(const char *s);
```

`strlen` scans the text string starting at *s* and returns the number of characters it encounters before the first null character.

strncat

```
#include <string.h>
char *strncat(char *s1, const char *s2, size_t n);
```

`strncat` appends up to *n* characters from the string *s2* to the end of string *s1*, and then terminates the string with a null. `strncat` returns its first argument, *s1*.

`strncat` is identical to `strcat`, except it appends a limit of *n* characters, plus one for the null.

strncmp

```
#include <string.h>
char *strncmp(char *s1, const char *s2, size_t n);
```

`strncmp` compares two text strings, *s1* and *s2*, character by character, for lexical order in the character collating sequence. The first string starts at *s1*, the second at *s2*. *n* specifies the maximum number of characters to be compared, unless the terminating null in *s1* or *s2* is encountered first. The strings must match, including any terminating null characters that may be encountered, in order for them to be equal. `strncmp` returns an integer greater than zero if *s1* is lexically greater than *s2*, zero if *s1* is lexically equal to *s2*, and an integer less than zero if *s1* is lexically less than *s2*.

`strncmp` is identical to `strcmp`, except it compares a maximum of *n* characters.

strncpy

```
#include <string.h>
char *strncpy(char *s1, const char *s2, size_t n);
```

`strncpy` copies characters from *s2* to *s1* until it reaches the end of *s2* or until *n* characters have been copied. `strncpy` pads with zeros, if necessary, to copy *n* characters total. `strncpy` returns its first argument, *s1*.

If the string *s2* is longer than *n* characters, *s1* may not end with a null character.

strpbrk

```
#include <string.h>
char *strpbrk(const char *s1, const char *s2);
```

`strpbrk` scans the string *s1* for the first occurrence of a character in the string *s2*. `strpbrk` returns a pointer to the first character in *s1* that is also in *s2*, or null if *s1* has no characters from *s2*.

strchr

```
#include <string.h>
char *strchr(const char *s, int c);
```

strchr looks for the last occurrence of a specific character, *c*, in a null terminated target string, *s*. *strchr* returns a pointer to the last character that matches *c*. If no character matches *c*, *strchr* returns NULL.

In ANSI C, *strchr* replaces the Unix *rindex* routine.

strspn

```
#include <string.h>
size_t strspn(char *s1, const char *s2);
```

strspn scans the string *s1* for the first occurrence of a character not in the string *s2*. *strspn* returns the length of the initial segment of *s1* which consists entirely of characters in *s2*.

strstr

```
#include <string.h>
char *strstr(const char *s1, const char *s2);
```

strstr finds the first instance of the string *s2* in the string *s1*. *strstr* returns a pointer to the occurrence of *s1* if found or a null pointer if the string was not found. If *s2* points to a string of zero length, *strstr* returns *s1*.

strtod

```
#include <stdlib.h>
double strtod(const char *s, char **endptr);
```

strtod converts a string *s*, into a double floating-point type. *strtod* is identical to *atof*, except that it stores a pointer to the remainder of the string in the object pointed to by *endptr*, providing that *endptr* is not NULL. Leading white-space characters (as defined by the *isspace* function) are allowed.

If `strtod` detects a format error in the string, it returns zero. If the correct value is outside the range of representable values, plus or minus `HUGE_VAL` is returned and the value of the macro `ERANGE` is stored in `errno`.

strtok

```
#include <string.h>
char *strtok(char *s1, const char *s2);
```

`strtok` breaks a string into tokens. Consider *s1* as a sequence of zero or more tokens separated by spans of one or more characters from the “separator” string, *s2*. The first call to `strtok` returns a pointer to the first token in *s1*, and will have a null written at the end of the token. The function keeps track of its position in *s1*, and subsequent calls work through *s1* after the last token returned. When no tokens remain, `strtok` returns NULL. The *s2* string may be different from call to call.

This routine will not operate correctly if *s1* points to ROM (read-only memory).

strtol

```
#include <stdlib.h>
long strtol(const char *s, char **endptr, int base);
```

`strtol` converts a string, *s*, into a long integer. The string that will be converted is the longest string which matches the following pattern:

```
optional white-space
optional +/- sign,
optional 0x or 0X,
0-9, a-z, A-Z
```

Here *a-z* represent 10 to 35 for bases greater than 10. The 0x or 0X are only allowed when the base is zero or 16. `strtol` only understands letters that are less than the base. If base equals 0, and there is a leading 0x or 0X, the base is assumed to be 16. If there is a leading 0, the base is octal (8). Any other initial patterns are considered to be decimal (base 10) numbers.

**endptr* is set to the remainder of the string that was not converted to a long.

If *s* does not contain a valid pattern, then **endptr* is set equal to *s*, and the `strtol` returns zero. If the value is too large to be represented by a long, `strtol` sets `errno` to `ERANGE`, returns `LONG_MAX` for positive numbers, and returns `LONG_MIN` for negative numbers.

strtoul

```
#include <stdlib.h>
unsigned long strtoul(const char *s,
                     char **endptr, int base);
```

`strtoul` converts a string, *s*, into an integer of type unsigned long integer. `strtoul` is identical to `strtol`, except that it converts the string to an unsigned long int.

**endptr* is set to the remainder of the string that was not converted to a long.

If *s* does not contain a valid pattern, **endptr* is set equal to *s*, and the `strtoul` returns zero. If the value is too large to be represented by an unsigned long, `strtoul` sets `errno` to `ERANGE`, and returns `ULONG_MAX`.

strxfrm

```
#include <string.h>
size_t strxfrm(char *s1, const char *s2, size_t n);
```

`strxfrm` transforms the input string pointed to by *s2* and places the result in *s1*. No more than *n* characters are stored in *s1*, including the terminating null character. The transformation is such that `strcmp` on the transformed strings yields the same value as `strcmp` on the untransformed strings.

If *n* is zero, then *s1* may be null. `strxfrm` returns the length of the transformed string, not including the null character. If the value returned is *n* or more, then no characters are copied.

swab

```
#include <extended.h>
void swab(char *from, char *to, int nbytes);
```

swab copies *nbytes* bytes pointed to by *from* to the position pointed to by *to*, exchanging adjacent even and odd bytes.

nbytes should be an even number.

If *from* and *to* data areas overlap, the results are undefined.

swab is a TASKING extension.

system

```
#include <stdlib.h>
int system(const char *str);
```

system passes the string *str* to the host environment's command processor. If *str* is a null pointer, *system* can be used to inquire whether a command processor exists, in which case *system* returns nonzero only if a command processor exists.

system must be implemented by the user.

tan

```
#include <math.h>
double tan(double x);
```

tan computes the tangent of *x*, expressed in radians.

tanh

```
#include <math.h>
double tanh(double x);
```

tanh computes the hyperbolic tangent of *x*.

time

```
#include <time.h>
time_t time(time_t *timer);
```

`time` returns the current calendar time, expressed as the number of seconds that have elapsed since Jan 1, 1970 12:00 A.M.

`time` must be implemented by the user. The current return value is `(time_t)-1`.

tmpfile

```
#include <stdio.h>
FILE *tmpfile(void);
```

`tmpfile` creates a file that will be automatically be removed when the file is closed or the program exits.

`tmpfile` must be implemented by the user.

tmpnam

```
#include <stdio.h>
char *tmpnam(char *s);
```

`tmpnam` returns a valid file name that will not conflict with any existing file names.

tolower

```
#include <ctype.h>
int tolower(int c);
```

`tolower` converts an upper-case letter `c` to its lower-case equivalent, leaving all other characters unmodified. `tolower` returns the corresponding lower-case character or the unchanged character.

`tolower` is a function, not a macro. If `c` is known to be an upper-case character, then `__tolower` is faster.

toupper

```
#include <ctype.h> int toupper(int c);
```

`toupper` converts a lower-case character *c* to its upper-case equivalent, leaving all other characters unmodified. `toupper` returns the corresponding upper-case letter or the unchanged letter.

`toupper` is a function, not a macro. If *c* is known to be a lower-case character, then `__toupper` is faster.

ungetc

```
#include <stdio.h>
int ungetc(int c, FILE *stream);
```

`ungetc` “puts back” the character *c* into the specified input *stream*.

The ANSI C standard only guarantees that one character can be “ungotten” without an intervening read.

unlink

```
#include <unistd.h>
int unlink( const char * name );
```

Removes the named file, so that a subsequent attempt to open it fails. This function calls the low-level routine `_unlink`.

`unlink` returns zero if file is successfully removed, or a non-zero value, if the attempt fails.

va_arg

```
#include <stdarg.h>
type va_arg(va_list ap, type);
```

The `va_arg` macro expands to an expression that has the type *type* and value of the next varying argument in the call. The parameter *ap* must be the same as that returned by `va_start`. Each invocation of `va_arg` modifies *ap* so that the values of successive arguments are returned.

va_end

```
#include <stdarg.h>
void va_end(va_list ap);
```

The `va_end` macro is used after all parameters on a variable length parameter list have been accessed with `va_arg`. It must be referenced before return from the function that contains the variable length argument list.

`va_end` generates no code; it is only used to guarantee portability to other compiler systems.

va_start

```
#include <stdarg.h>
void va_start(va_list ap, parameter);
```

The macro `va_start` is used to initialize the reading of variable length arguments. It initializes *ap* for subsequent use by `va_arg` and `va_end`. *parameter* is the rightmost identifier in the variable parameter list in the function definition (the one just before the ellipsis).

vfprintf

```
#include <stdio.h>
int vfprintf(FILE *stream, const char *format,
             va_list arg);
```

`vfprintf` is equivalent to `fprintf`, with the variable argument list replaced by *arg*. *arg* must be initialized by the `va_start` macro (and possibly subsequent `va_arg` calls). `vfprintf` returns the number of characters transmitted or a negative number if an output error occurred.

`vfprintf` does not invoke the `va_end` macro.

vprintf

```
#include <stdio.h>
int vprintf(const char *format, va_list arg);
```

`vprintf` is equivalent to `printf`, with the variable argument list replaced by *arg*. *arg* must be initialized by the `va_start` macro (and possibly subsequent `va_arg` calls). `vprintf` returns the number of characters transmitted or a negative number if an output error occurred.

`__yvprintf` and `vprintf` are identical.

`vprintf` does not invoke the `va_end` macro.

vsprintf

```
#include <stdio.h>
int vsprintf(char *s, const char *format,
             va_list arg);
```

`vsprintf` is equivalent to `sprintf`, with the variable argument list replaced by *arg*. *arg* must be initialized by the `va_start` macro (and possibly subsequent `va_arg` calls). `vsprintf` returns the number of characters written to the array, not including the terminating null character.

`vsprintf` does not invoke the `va_end` macro.

wcstombs

```
#include <stdlib.h>
size_t wcstombs(char *s, const wchar_t * pwcs,
                size_t n);
```

`wcstombs` copies *n* wide characters from *pwcs* to the multi-byte character string *s*. `wcstombs` returns 0 if *pwcs* is null, and returns -1 if *pwcs* contains an invalid character. Otherwise, up to *n* characters are copied from *pwcs* to *s* and the number of characters copied is returned. The `wcstombs` routine assumes the Shift JIS convention for Japanese characters. See `mblen` for more details.

wctomb

```
#include <stdlib.h>
int wctomb(char *s, wchar_t wc);
```

`wctomb` copies the wide character *wc* to *s*. It returns 0 if *s* is a null pointer and returns -1 if *wc* is an invalid character. Otherwise *wc* is written at *s* and 1 is returned if the character is a one-byte ASCII or kana code and 2 is returned if it is a two-byte Kanji code. `wctomb` assumes the shift JIS convention for Japanese characters. See `mblen` for more details.

write

```
#include <unistd.h>
size_t write( int fd, char * buffer, size_t count );
```

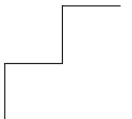
Write a sequence of characters to a file. This function calls the low-level routine `_write`, which interfaces to CrossView Pro's file system simulation.

`write` returns the number of characters correctly written.

CHAPTER

3

ASSEMBLY LANGUAGE REFERENCE



3

CHAPTER

The assembly language implemented by the 68K/ColdFire family assembler was designed by Motorola, Inc. It has the features commonly found in modern macro assembly languages. These features include absolute/relocatable code generation, complex relocatable expressions, macros, conditional assembly, and structured syntax. This chapter summarizes the basic structure of the assembly language, and gives an overview of the assembly language features.

3.1 PREFACE

The 68K/ColdFire family assembler translates assembly language source programs into object modules. It is part of the TASKING 68K/ColdFire toolkit, an integrated set of cross-compilers, assemblers, source level debugger and other utilities. The other parts of the development system are described in the *User's Manual*, which gives information on how to invoke the assembler on your host system.

The TASKING 68K assembler was designed to be compatible with the Motorola 68000 assembler. Most programs developed for the Motorola assembler should be readily portable and source-compatible with the TASKING 68K assembler.

This document provides information necessary to use the assembler to develop assembly language programs for the Motorola 68K/ColdFire family of microprocessors. It is not a comprehensive guide to the instruction set and architecture of the 68K/ColdFire family of microprocessors.

Portions of this document are copyrighted by and used with the permission of Motorola, Inc.

3.2 RELATED PUBLICATIONS

The following Motorola Inc. publications provide a comprehensive treatment of microprocessor architecture and the instruction set. They may be ordered from Motorola.

- M68000 Family Programmers Reference Manual (Motorola, Inc.)
- CPU32 Reference Manual (Motorola, Inc.)
- MC68xxx User's Manuals (Motorola, Inc.)
- ColdFire Family Programmers Reference Manual (Motorola, Inc.)
- MCF5xxx User's Manuals (Motorola, Inc.)

See the Motorola Semiconductor website (<http://e-www.motorola.com>) for the complete documentation list for your derivative.

3.3 USING ASSEMBLY LANGUAGE

High-level languages (e.g., C and Pascal) can decrease the development time needed for a program or system. All things being equal, a high-level language program is likely to be more reliable, easier to understand, and easier to maintain. However, high-level languages do not permit the programmer to directly access all the microprocessor's features, such as registers, processor flags, and special instructions.

As a result, an assembly language program, while sometimes taking longer to code and debug, can run much faster and occupy less memory than the equivalent program written in a high-level language. Since program development time is expensive, the trade-off between development time and program performance should be analyzed for each application. The optimal solution is usually found in writing most routines in a high-level language and in writing the time-critical, space-critical, and special routines (e.g., I/O routines) in assembly language.

3.4 ELEMENTS OF ASSEMBLY LANGUAGE

The lines in an assembly language source file can be classified in four general categories: instruction statements, data allocation statements, assembler directives, and assembler controls.

An **instruction statement** uses an easily remembered name, a **mnemonic**, and possibly one or two operands+ to specify a machine instruction to be generated.

A **data allocation statement** reserves, and optionally initializes, memory space for program data.

An **assembler directive** is a statement that gives special instructions to the assembler. Although directives may produce something in the object file, they are unlike the instruction and data allocation statements in that they do not specify the actual contents of memory.

An **assembler control** is also a statement that gives special instructions to the assembler. Assembler controls are used to control the assembly process rather than to define the meaning of the program being assembled.

Here are some examples of the different kinds of statements:

| Statement Type | Examples |
|---------------------|----------------------------------|
| Instruction | MOVE D2,D4 JSR SORT_PROCEDURE |
| Data Allocation | DS.W 0 DC.B 'H' |
| Assembler Directive | SECTION MYSEC COUNT EQU 5 |
| Assembler Control | NOPAGE INCLUDE source.inc |

Table 3-1: Statements

In addition, the language is composed of the following symbolic elements:

- Symbolic names or labels, which are instructions, directives, register mnemonics, user-defined memory labels, and macros.
- Numbers, which may be represented in binary, octal or decimal.
- Arithmetic and logical operators, which are used in complex expressions.
- Special purpose characters, which are used for macro functions, source line fields, and numeric bases.

3.5 NOTATION

A small amount of specialized notation is used in this document to specify the general format for instructions and directives. It is based on fairly standard additions to the Backus-Naur Form (BNF) formalism. The four “metasymbols” described below are used throughout this manual to indicate that the user must replace the metasymbols and the characters they enclose with some legal text. The actual text that can be substituted will be different in each case, and depends on what type of assembly language statement is being described.

When symbols or metasympols represent the actual commands or text to be supplied, they will be in **boldface**. When symbols or metasympols must be replaced by commands or text, they will be in *italics*. In some cases, the characters used as metasympols are required characters in a statement rather than metasympols. In those cases they will appear in boldface rather than italics. The four metasympols and their meanings are:

- < > Angular brackets enclosing a name indicate that one element of the general category specified by the name is to be selected.
- | The pipe (vertical bar) indicates that a choice is to be made. One of the symbols separated by the pipe character(s) should be selected.
- [] Square brackets indicate that the enclosed sequence is optional. The enclosed sequence may occur one time or not at all.
- []... Square brackets followed by periods enclose a symbol that is optional but repetitive. The symbol may appear zero or more times.

For example,

MOVE[.<size>] <source>,<destination>

where:

<size> = **B** | **W** | **L**

BFEXTS <ea>{<offset>:<width>},**Dn**

where:

<offset> = #<expr> | **Dn**

<width> = #<expr> | **Dn**

In the first example, the [.<size>] notation implies the user may supply **MOVE.L**, **MOVE.W**, **MOVE.B**, or **MOVE** as legal mnemonics.

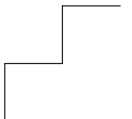
In the second example, the user must supply either an immediate value or a data register for the “offset” and “width” symbols. The curly braces must appear in the instruction. Thus, a legal instruction would be:

BFEXTS **LAB**{0:8},**D1**.

CHAPTER

4

SOURCE PROGRAM CODING



4

CHAPTER

A source program is a sequence of source statements arranged in a logical way to perform a predetermined task. The assembler interprets and processes each source line, generating object code or performing a specific assembly time process. This chapter discusses some facets of source program coding including source line format, symbols, constants, operators and expressions. For other facets such as registers, addressing modes, instruction mnemonics, and other instruction types refer to the *Microprocessor Manual* for your particular processor.

4.1 INTRODUCTION

A source program is a sequence of source statements arranged in a logical way to perform a predetermined task. The assembler interprets and processes each source line, generating object code or performing a specific assembly time process. Each source statement occupies a line of printable text, where each line may be one of the following:

- Comment
- Executable instruction
- Assembler directive
- Macro invocation



The TASKING 68K/ColdFire cross-assemblers are case insensitive to source input except as noted under the INCLUDE directive or for ASCII strings.

4.2 COMMENTS

Comments are strings of characters that are inserted only to identify or clarify the program. Comments are included in the assembly listing but are otherwise ignored by the assembler.

A comment may be inserted in one of two ways:

- As a line, starting in column one, where an asterisk (*) is the first character in the line. The entire line is a comment, and an instruction or directive in this line will not be recognized.
- Following the operation and operand fields of an assembler instruction or directive, where it is preceded by at least one blank or tab character.

Example

```
* THIS ENTIRE LINE IS A COMMENT.  
BRA LAB2      THIS COMMENT FOLLOWS AN  INSTRUCTION
```

4.3 SOURCE LINE FORMAT

Each source statement has an overall format that is some combination of the following fields:

- Label
- Operation
- Operand
- Comment

The statement lines in the source file must not be numbered. However, in the listing file the assembler prefixes each line with a sequential number, up to four decimal digits.

The format of each line of source code is described in the following paragraphs.

4.3.1 LABEL FIELD

The label field is the first field in the source line. A label which begins in the first column of the line may be terminated by either a blank, a tab, or a colon. A label may be preceded by one or more blanks or tabs, provided it is then terminated by a colon. In either case, the colon is not considered to be part of the label.

Labels are allowed on all instructions and on all assembler directives which define data structures. For such operations, the label is defined with a value equal to the location counter for the beginning of the instruction or directive, including a designation for the program section (if there is one) in which the definition appears.

Labels are required on the assembler directives which define symbol values (SET, EQU, REG). For these directives, the label is defined to be the value of the expression in the operand field. This value consists of a constant and, if the expression is relocatable, a section designation.

Labels are required on macro definitions and serve as the mnemonic by which that macro is subsequently invoked. No memory address is associated with such labels. A label is also required on the IDNT directive. This label is passed on to the relocatable object module *but it has no associated internal value*. The IDNT statement, therefore, cannot be used to define program entry points.

Labels which are the only field in the source line are defined equal to the current location counter value plus, if the section is relocatable, the program section.

4.3.2 OPERATION FIELD

The operation field follows the label field and is separated from it by at least one blank or tab. If the label field is not being used, the operation field must be at least one blank or tab from the left margin. The operation field specifies the action to be performed by the statement. Entries in this field fall under one of the following categories:

- *Instruction mnemonics:* The M68000-family processor instruction set.
- *Directive mnemonics.:* These control the assembly process.
- *Macro calls.:* These are invocations of previously defined macros.

The size of the data field affected by an instruction is determined by the data size code. Some instructions and directives can operate on more than one data size. For these operations, the data size code must be specified or a default size is assumed. The size code need not be specified if only one data size is permitted by the operation. The data size code is specified by appending a period (.) to the operation field, followed by B, W, L, S, D, X, or P, where the appended character is interpreted as described below:

| | |
|----------|-----------------------------------------------------------------------------------------------------------------------------------|
| B | Byte (8-bit data) |
| W | Word (16-bit data) |
| L | Longword (32-bit data) |
| S | Byte (8-bit signed offset for certain branch instructions) |
| | Single precision binary real (IEEE Standard, 32-bit: 1-bit sign, 8-bit exponent, 23-bit mantissa) (68881/68882/68040/68060 only) |
| D | Double word (64-bit data) (68030, 68040, 68060, and 68851 only) |
| | Double precision binary real (IEEE Standard, 64-bit: 1-bit sign, 11-bit exponent, 52-bit mantissa) (68881/68882/68040/68060 only) |

- X** Extended precision binary real (96-bit: 15-bit exponent, 1-bit sign, 64-bit mantissa), (16 bits are reserved)
(68881/68882/68040/68060 only)
- P** Packed Binary Coded Decimal (BCD) real string (96-bit: 3-decimal digit exponent and 17-decimal digit mantissa)
(68881/68882/68040/68060 only)

The data size code is not permitted, however, when the instruction or directive does not have a data size attribute.

Examples (legal)

```

      LEA 2(A0),A1      Longword size is assumed.
*                      (.B, .W not allowed) This
*                      instruction loads the
*                      effective address pointed
*                      to by A0+2 into A1.
*
      ADD.B ADDR,D0     This adds the byte whose
*                      address is ADDR to
*                      the low order byte in D0.
*
      ADD D1,D2         This adds the low order
*                      word of D1 to the low
*                      order word of D2. (W is
*                      the default size code.)
*
      ADD.L A3,D3       This adds the 32-bit
*                      contents of A3 to D3.
*
```

Example (illegal)

```

SUBA.B #5,A1          Illegal size specification
*                      (.B not allowed on SUBA)
*                      This instruction attempts
*                      to subtract the value 5
*                      from the low order byte
*                      of A1; but byte operations
*                      on address registers are
*                      not allowed.
*
```

4.3.3 OPERAND FIELD

If present, the operand field follows the operation field and is separated from the operation field by at least one blank or tab. When two or more operand subfields appear within a statement, they must be separated by a comma but may not contain embedded blanks or tabs; e.g., “ADD D1, D2” is illegal.

For most two operand instructions, the general format “*opcode source,destination*” applies. For example, in an instruction like “ADD D1,D2”, the contents of D1 are added to the contents of D2 and the result is saved in register D2. In the instruction “MOVE D1,D2”, the first operand (D1) is the sending operand and the second operand (D2) is the receiving operand.

4.3.4 COMMENT FIELD

The last field of a source statement is an optional comment field. This field is ignored by the assembler, but is included in the listing. The comment field is separated from the operand field (or the operation field, if there is no operand) by at least one blank or tab, or by a semicolon (;), and may consist of any ASCII characters.

4.4 SYMBOLS

Symbols can correspond to either a specific numerical value by using an EQU or SET directive, or the address of a memory location. The memory location can represent the destination of a branch instruction or the start of a data area. This use of symbolic references to memory allows statements to be written without specifying actual memory locations. An entry in the label field is required for all statements that are the destination of jump and branch instructions and in statements using the EQU or SET directives.

4.4.1 SYMBOL SYNTAX

Symbols recognized by the assembler consist of one or more valid characters (refer to the *Character Set* appendix), all of which are significant. The first character must be an upper case or lower case letter (A-Z or a-z), a period (.) or an underscore (_). Each remaining character may be an upper case or lower case letter (A-Z, a-z), a period (.), an underscore (_), a digit (0-9), a dollar sign (\$), or a question mark (?).

4.4.2 SYMBOL DEFINITION CLASSES

Symbols may be differentiated by usage into two classes. Class 1 symbols are used in the operation field of an instruction. Class 2 symbols occur in the label and operand fields of the instruction. Assembler directives (including those for structured assembly), instruction mnemonics, and macro names comprise Class 1 symbols; Class 2 symbols consist of user defined labels and register mnemonics.

A Class 1 symbol may be redefined and used independently as a Class 2 symbol, and vice versa. As long as each symbol is used correctly, no conflict will result from the existence of two symbols of different classes with the same name. For example, the following is a legal instruction sequence:

```

      ADD    D1,ADD
      .
      .
      .
ADD    DS    2

```

By its usage as a Class 1 symbol, the first ADD is recognized as an instruction mnemonic; the second ADD is recognized as a Class 2 symbol identifying a reserved storage area. The assembler differentiates a Class 1 symbol from a Class 2 symbol with the same name, thereby allowing two symbol table entries with the same name but different class.

Macro labels are a special case because the same symbol will appear as the label (Class 2) in the MACRO definition and, subsequently, as an operation code mnemonic (Class 1) in invocation of that same macro. Macro labels are defined to be Class 1 symbols; their presence in the label field of a MACRO directive is ignored as a Class 2 symbol. Therefore, macro names may be redefined as Class 2 symbols without conflict.

Except for the SET directive, which allows multiple redefinition of a Class 2 symbol, a symbol may not be redefined within the same class. For example, SUB (reserved Class 1 symbol) may not be redefined as a macro label (also Class 1), nor may A5 (reserved Class 2 symbol) be redefined as a statement or storage location label (also Class 2).

4.4.3 USER-DEFINED LABELS

Labels are symbols that are defined by the user to identify memory locations in program or data areas of the assembly module.

Labels may have an absolute or relocatable value, depending upon the section in which the labeled memory location is found. If the memory location is within a relocatable section, the label has a relocatable value, i.e., it depends on where the section is placed. If the memory location is in an absolute section, the label has an absolute value.

Labels may be defined in the label field of an executable instruction or a data definition directive source line. It is also possible to define a label with the SET or EQU directive to an arbitrary value.

4.4.4 LOCATION COUNTER SYMBOL "*"

The special symbol "*" may be used to refer to the current location counter value.

4.5 CONSTANTS

4.5.1 INTEGER CONSTANTS

Numeric constants recognized by the assembler may be expressed in decimal, hexadecimal, octal, and binary form. They must have integral values and must be expressible in 32 bits.

Decimal Constants

Decimal is the default base used for evaluating numeric values and consists of a string of numeric digits.

Example:

```
12345      Valid
12.3       Invalid: can consist only of digits
```

Hexadecimal Constants

A hexadecimal constant consists of characters from the set of decimal digits (0–9) and the alphabetic characters (A–F, a–f) and is preceded by a dollar sign (\$) or followed by an **H**.

If the suffix form is used, the first character must be a digit to distinguish the hexadecimal constant from a symbol name.

Example:

```
      $12      Valid
      01CFH    Valid
      $01CF    Valid
      ABCDH    Invalid: no preceding $
*           This would be interpreted as
*           symbol ABCDH
```

Octal Constants

An octal constant consists of characters from the set of digits (0–7), preceded by a commercial at sign (@) or followed by a **Q**.

Example:

```
      @17634   Valid
      275Q     Valid
      @27832   Invalid: character 8 not
*                allowed
```

Binary Constants

A binary constant consists of 1's and 0's, preceded by a percent sign (%) or suffixed by a **B**.

Example:

```
      %10100   Valid
      10111B   Valid
      %21001   Invalid: character 2 not
*                allowed
```

4.5.2 CHARACTER CONSTANTS

One or more printable ASCII characters enclosed by apostrophes (') constitute a character constant. Character constants longer than four characters may only be stored in memory, e.g., using the DC (define constant) directive. Shorter character constants may also be used as immediate operands, in which case they are treated as an integer according to the rules described below.

Character constants are left justified and zero filled, if necessary, whether stored or used as immediate operands. Constants with four or three characters are aligned to a longword. Constants with two characters are aligned to a word. Single character constants are word aligned if the operand size is larger than a byte, and byte aligned if the operation size is a byte.

In order to specify an apostrophe within a character constant, two successive apostrophes must appear where the single apostrophe is intended to appear.

Examples

```
tB_DIG_1    EQU.B    '1'           Equates B_DIG_1 with
*                                     hex 31
W_DIG_1     EQU      '1'           Equates W_DIG_1 with
*                                     hex 3100
*                                     DC.L    '79'       Stores hex 37390000
*                                     in memory
*                                     MOVE.L   #'1',D0     Moves hex 00003100
*                                     into D0
*                                     MOVE.B   #'1',D0     Moves hex 31 into
*                                     low byte of D0
*                                     MOVE.L   #'123',D0    Moves hex 31323300
*                                     into D0
```

4.5.3 FLOATING POINT CONSTANTS
(68881/68882/68040/68060 ONLY)

IEEE standard floating-point numbers can be specified by an optionally signed fraction string of up to 17 decimal digits (0-9) containing a required decimal point, the constant "E" an optional sign, and an exponent up to 3 decimal digits. The exponent section "E<sign>yyy" is optional; underscores can occur for readability.

Floating Point Constant Notation

[<sign> | x.xxxxxxxxxxxxxxxxx[E[<sign>]yy]
(maximum size)

where:

<sign> is + | -

x and y are decimal digits

Example:

```
DC.X      1234.56E-33
DC.X      -12345.67
```

Floating point numbers can also be specified explicitly as a series of hexadecimal digits preceded by a colon (:). This floating-point hex format can be used to exactly represent the mantissa, exponent, and sign bit for a given floating-point number.

Floating Point Hex Constant Notation

:bbbbbb[b]...

where:

b is a hex digit

Up to 8 digits are allowed for .S precision, up to 16 for .D, and up to 24 for .X or .P.

Example:

```
DC.S :124F67B  STORES HEX  124F67B0
*              IN MEMORY
```

4.6 OPERATORS

Operators recognized by the assembler include, in order of operator precedence (from highest to lowest):

1. Parenthetical Expression (innermost first)
2. Unary Minus, Plus, SEGSIZE, SEGBASE
3. Shift Right, Shift Left
4. And, Or

- 5. Multiplication, Division, Remainder
- 6. Addition, Subtraction

Expressions involving operators with the same precedence are evaluated from left to right. All expressions are evaluated using 32-bit values and all intermediate results are stored internally as 32-bit integers.

All examples are shown using constants. Absolute labels, however, may be used in any expression where a constant is shown below. Absolute labels include labels of data or code located in absolute sections, or labels on EQU or SET directives which have an expression that evaluates to a constant value. In addition, labels on DS directives in the table associated with an OFFSET directive are absolute. Relocatable labels and SEGSIZE or SEGBASE values may only be used in expressions involving addition and subtraction.

| Arithmetic Operators: | | |
|-----------------------------|-----------------|------------------------------------------------------------------------------------------------------------------|
| addition | (+) | |
| subtraction | (-) | |
| multiplication | (*) | |
| division | (/) | Produces a truncated integer result. |
| remainder | (%) | |
| unary minus | (-) | |
| segment size | SEGSIZE (seg) | |
| segment base | SEGBASE (seg) | |
| Shift Operators (binary): | | |
| shift right | (>>) | The left operand is shifted to the right (and zero filled) by the number of bits specified by the right operand. |
| shift left | (<<) | Analogous to >. |
| Logical Operators (binary): | | |
| and | (&) | |
| or | (!) | |

Table 4-1: Operators



The percent (remainder) operator is only valid if the “select TASKING extensions” option is in effect. See the *Assembler* chapter in the *User’s Manual* for more information about this option. The same is true of the `SEGSIZE` and `SEGBASE` operators.

SEGSIZE — Get Segment Size

Syntax

SEGSIZE(<name>)

The `SEGSIZE` operator returns the size in bytes of the named section. It is necessary to `XREF` the section name if it is not defined in the current module.

SEGBASE — Get Base Address of Segment

Syntax

SEGBASE(<name>)

The `SEGBASE` operator returns the base address of the named section. It is necessary to `XREF` the section name if it is not defined in the current module.

Example

```
XREF udata

SECTION foo,, "code"

MOVEA.L #SEGBASE(udata),A0    load base address of udata
MOVE.L #SEGSIZE(udata)-1,D0    load size of udata - 1
                                (for dbf)

loop CLR.B    (A0)+
      DBF     D0,loop          zero out udata segment
```

4.7 EXPRESSIONS

Expressions are composed of one or more symbols and/or constants that may be combined with unary or binary operations. Expressions are evaluated according to precedence rules left to right.

Subexpressions which involve relocatable symbols may use only the “+” and “-” operators. It is possible for a subexpression involving the difference between two relocatable symbols to evaluate to an absolute value. For example, let R1 represent a memory location at OFFSET1 bytes beyond the start of section S1, and let R2 represent a memory location at OFFSET2 bytes beyond the start of section S2. That is:

```
R1 = OFFSET1 + <start of S1>
R2 = OFFSET2 + <start of S2>
```

The difference between R1 and R2 may then be:

```
R1-R2 = OFFSET1-OFFSET2
+ <start of S1> - <start of S2>
```

If sections S1 and S2 are the same, then:

```
R1-R2 = OFFSET1-OFFSET2
```

which is an absolute (non-relocatable) value. Of course, if sections S1 and S2 are separate and distinct, the expression remains a complex relocatable expression.

When an expression has been fully evaluated by the assembler, it may be categorized as one of three types of expressions:

- *Absolute Expression:*
The expression has reduced to an absolute value that is independent of the start address of any relocatable section.
- *Simple Relocatable Expression:*
The expression has reduced to an absolute offset from the start of a single relocatable section.
- *Complex Relocatable Expression:*
The expression has reduced to a constant, absolute offset in conjunction with either of the following relocatable terms:

A single, negated start address of a relocatable section.

or

An expression of two or more relocatable symbols,
or containing a SEGSIZE or SEGBASE value.



Only absolute expressions with no forward references are legal in ORG, OFFSET, DCB, and DS directives. SET and EQU can take any expression.

By themselves, all user-defined labels on memory locations are either absolute or simple relocatable expressions. This includes XREF labels, which are assumed to be absolute symbols unless their program section is specified. Complex relocatable expressions may arise only from the addition or subtraction of two relocatable expressions.

The following are examples of each type of expression:

| | | | |
|---------|---------|-------------|---------------------|
| | ORG | \$1000 | absolute section |
| ARRAY | DS | \$20 | ARRAY is absolute |
| ENDARRY | EQU | *-2 | ENDARRY |
| * | | | is absolute |
| | SECTION | 2 | relocatable |
| * | | | table section |
| L1 | CLR.L | D2 | L1 is simple |
| * | | | relocatable |
| L2 | MOVE | D3,(A0) | L2 is simple |
| * | | | relocatable |
| | MOVE.L | ARRAY+10,D7 | absolute source |
| * | | | operand |
| | MOVE.L | L1+10,D7 | simple relocatable |
| * | | | source operand |
| | MOVE.L | L2-L1,D7 | absolute source |
| * | | | operand |
| | MOVE.L | L1+L2,D7 | complex relocatable |
| * | | | source operand |

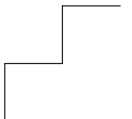
4.8 ADDRESSING MODES

Effective address modes, combined with operation codes, define the particular function to be performed by a given instruction. Effective addresses and data organization are described in detail in the *Data Organization and Addressing Capabilities* section of the *Microprocessor Manual* for the appropriate processor.

CHAPTER

5

ASSEMBLER DIRECTIVES



5

CHAPTER

All assembler directives (pseudo-ops), with the exception of DC and DCB, are instructions to the assembler rather than instructions to be translated into object code. This section contains descriptions and examples of the basic forms of the most frequently used assembler directives. See also the *Macro and Conditional Assembly* and *Structured Control Statements* sections about directives. The most commonly used directives supported by the assembler are Assembly Control, Symbol Definition, Data Definition/Storage Allocation, Listing Control and Output Options, External Symbol Control, and Internal Assembly Control.

5.1 ASSEMBLY CONTROL

The TASKING 68K/ColdFire assembler contains the following assembly control directives:

| | |
|---------|-------------------------------|
| COMMON | Named Common |
| END | Program End |
| INCLUDE | Include Secondary File |
| OFFSET | Define Offsets |
| ORG | Absolute Origin |
| RESERVE | Reserve room in section |
| RESUME | Resume section |
| RORG | Relocatable Org |
| SECTION | Relocatable Program Section |
| BRINGIN | Declare external symbol |
| DEBSYM | Put out debugging information |
| DGROUP | Define data group |

5.1.1 COMMON - ENTER NAMED COMMON SECTION

Syntax

[*label*]**COMMON**[.S]<*name*> [, [**ABSOLUTE**[:<*location*>]], "<*class*>"]

Description

The COMMON directive performs the same functions as the SECTION directive which is described later in this chapter, except that the generated section is marked as "common" in the object module. When the linker encounters the same common section name in two object modules, it combines them by overlay; the length of the resulting section is the maximum of the lengths of the input sections.

In contrast, when the linker encounters the same non-common section in two object modules, it combines them by *concatenation*; the length of the resulting section is then the sum of the lengths of the input sections.



This directive can only be used if the "select TASKING extensions" option is in effect.

If the ABSOLUTE specification is selected, the segment defined by the COMMON statement will be located at the indicated absolute address (or zero, if <*location*> is omitted). If the same absolute common section is defined in more than one assembly, then the same <*location*> must be specified at every definition.

If present, the .S indicates the section will be placed completely within low address memory, i.e., between addresses 0 and hex 7FFF or between hex FFFF8000 and FFFFFFFF. This allows the assembler to generate more efficient addressing to items within the section. In particular, it can implement direct addressing through the "absolute short" addressing mode. Unlike the ABSOLUTE attribute described above, this information is *not* passed on in the object module; the user must take responsibility for placing the section in low memory with the TASKING locator.

`<class>` is an arbitrary string that will be associated with the section in the object module. The locator can locate all the sections with a given class name with a single command, so a consistent use of class names may make it easier to locate your program in memory. Please see the *Linking Locator* chapter in the *User's Manual* for more details.

Example

```
clab1      COMMON      sect1,,"cclass"
clab2      COMMON      sect2,ABSOLUTE:3000
clab3      COMMON      sect3,ABSOLUTE:100,"myclass"
```

5.1.2 END - PROGRAM END

Syntax

END [`<start address>`]

The END directive indicates to the assembler that the source is finished. Subsequent source statements are ignored. The END directive encountered at the end of the first pass through the source program causes the assembler to start the second pass. The start address should be specified unless it is external to the module. If no start address is specified, it is still possible to include a comment field, provided the comment field is set off by the exclamation point (!).

This syntax indicates to the assembler that the operand field is null and that a comment field follows.

The END statement is optional.

Example

```
in file1.68k:
    SECTION FOO
    NOP
    END          !      end of program, no
*                starting point defined

in file2.68k:
    SECTION BAR
    __BEGIN
    NOP
    END          __BEGIN
* end of program, program starts at __begin
```

5.1.3 INCLUDE - INCLUDE SECONDARY FILE

Syntax

INCLUDE *<file>* Include file from system directory.

INCLUDE *file* Include file from user directory.

The INCLUDE directive is inserted in the source program at any point where a secondary file is to be included in the source input stream. The first column on the line with the INCLUDE directive **must** be white space.

The search algorithm for finding the *<file>* depends on whether the file is to be found in the system directory list (specified in a command line option) or the standard include directory list (specified in a command line option).

Directories Searched:

<file> System directories only.

file User and System directories.

* Current directory.

Example

```
*      The first column in the include line must be
*      "white space"
*
*      INCLUDE          local_file.inc
*
*      Now include a system level include file
*      INCLUDE          <system_file.inc>
```

5.1.4 OFFSET - DEFINE OFFSETS

Syntax

OFFSET *<expression>*

The OFFSET directive is used to define a table of offsets via the Define Storage (DS) directive without actually declaring storage for the table, in effect creating a dummy section. Symbols defined in an OFFSET table are kept internally, but will not appear in the output object module. No code producing instructions or directives may appear. SET, EQU, REG, XDEF directives are allowed.

<expression> is the value at which the offset table is to begin. The expression must be absolute and may not contain forward, undefined, or external references.

OFFSET must be terminated by an ORG or SECTION directive before further code producing instructions are generated. If not, the assembler produces an error message.

Example

```

                                OFFSET    $7FFF
OFF1    DS.W    1                *      OFF1 is defined to
                                *      be 8000 hex
OFF2    DS.W    1                *      OFF2 is defined to
                                *      be 8002 hex

                                OFFSET    $0
OFF3    DS.W    1                *      OFF3 is defined to
                                *      be 0
OFF4    DS.W    1                *      OFF4 is defined to
                                *      be 2

                                SECTION    MORECODE
                                LEA        OFF1,A1    *      LEA 8000,A1
                                LEA        OFF2,A1    *      LEA 8002,A1
                                LEA        OFF3,A1    *      LEA 0,A1
                                NOP
                                END
```

5.1.5 ORG - ABSOLUTE ORIGIN

Syntax

```
ORG[.<qualifier>]<expression>[<comment>]
```

where:
<qualifier> is S | L

The ORG directive changes the program counter to the value specified by the expression in its operand field. Subsequent statements are assigned absolute memory locations starting with the new program counter value. <expression> must be absolute and may not contain any forward, undefined, or external references.

ORG.S is interpreted as both ORG and OPT FRS (Forward Reference Short Option). ORG.L is interpreted as both ORG and OPT FRL (Forward Reference Long Option). Regardless of the forward reference option, references to previously defined absolute symbols will always generate the appropriate short or long addressing form, based upon the size of a symbol's absolute address.

5.1.6 RESERVE - RESERVE STORAGE

Syntax

[label] RESERVE <name>, <length> [, "<class>"]

The RESERVE directive is similar to the OFFSET directive. It defines a segment and gives it the specified length, but leaves the user in the previous segment. Subsequent RESERVE directives with the same <name> add to the segment length.



This directive can only be used if the “select TASKING extensions” option is in effect. See the *Assembler* chapter in the *User’s Manual* for more information on the TASKING extensions.

<class> is an arbitrary string that will be associated with the section in the object module. The TASKING locator can locate all the sections with a given class name with a single command, so a consistent use of class names may make it easier to locate your program in memory. See the *Linking Locator* chapter in the *User’s Manual* for more details.

Example

| | | |
|-------|---------|-----------|
| | SECTION | DSCT |
| TOP | RESERVE | RSECT , 2 |
| | DC . L | TOP |
| HERE | RESERVE | RSECT , 3 |
| | DC . L | HERE |
| THERE | RESERVE | RSECT , 1 |
| | DC . L | THERE |

This creates a segment, RSECT, of length six and defines labels at locations 0, 2, and 5 within it. The addresses of the labels are stored in DSCT.

5.1.7 RESUME - RESUME DEFINED SECTION

Syntax

RESUME <name>

The RESUME directive resumes the named section. If no argument is given, the PSCT section is resumed.



This directive can only be used if the “select TASKING extensions” option is in effect. See the *Assembler* chapter in the *User’s Manual* for more information on the TASKING extensions.

Example

```
SECTION          TOP
NOP
SECTION          BOTTOM
NOP
RESUME           TOP
NOP
```

5.1.8 RORG - RELOCATABLE ORG

Syntax

RORG <expression> [<comment>]

The RORG directive has an effect similar to ORG, but it is intended for use in relocatable sections. The location counter is set to the value of the expression but remains in the current section. This is different from ORG, which switches to absolute assembly.



This directive can only be used if the “select TASKING extensions” option is in effect. See the *Assembler* chapter in the *User’s Manual* for more information on the TASKING extensions.

Example

```

DIGINX      EQU.B      0
ISDIGIT     DCB.B      256,0 Define table of 256 zeros.
*           Reset location counter back to ISDIGIT
              RORG      ISDIGIT
*           Advance to index of first ASCII digit
              RORG      *+DIGINX
*           Redefine table contents to be 1's at index of
*           ASCII digits
              DCB.B      10,1
              RORG      ISDIGIT+256      Reset location
*                                           counter past
*                                           ISDIGIT
              ...
*           LEA          ISDIGIT,A1      Load address of
                                           table
              MOVEQ.L    #0,D0
              MOVE.B     NEXTCHAR,D0    Pick up a
*                                           character
              BNZ        (A1,D0.W),L1    Goto L1 if it's a
*                                           digit
              BRA        L2              Goto L2 otherwise

```

5.1.9 SECTION - RELOCATABLE PROGRAM SECTION**Syntax**

SECTION[.S] <name>[,[ABSOLUTE[:<location>]][, "<class>"]]

SECTION[.S] <number>

This directive causes the program counter to be restored to the address following the last location allocated in the indicated section, or to zero if used for the first time with this <name>.



<location> and “class” can only be supplied if the “select TASKING extensions” option is in effect. See the *Assembler* chapter in the *User's Manual* for more information on the TASKING extensions.

If the ABSOLUTE specification is selected, the segment defined by the SECTION statement will be located at the indicated absolute address (or zero, if <location> is omitted).



Absolute sections are “uncombinable”, so it will cause an error if the same section is defined in another assembly or compilation. See the *Linking Locator* chapter in the *User’s Manual* for a description of segment combinability.

The `.S` indicates the section will be placed completely within low address memory, i.e., between addresses 0 and 32767. This allows the assembler to generate more efficient addressing to items within the section. In particular, it can implement direct addressing through the absolute short addressing mode. Unlike the absolute attribute described above, this information is *not* passed on in the object module; the user must take responsibility for placing the section in low memory with the TASKING locator.

`<number>` causes a segment named `$$seg<number>` to be created.

`<class>` is an arbitrary string that will be associated with the section in the object module. The TASKING locator can locate all the sections with a given class name with a single command, so a consistent use of class names may make it easier to locate your program in memory. Please see the *Linking Locator* chapter in the *User’s Manual* for more details.

Example

```
SECTION    lost
SECTION    abs1,ABSOLUTE,"abclass"
SECTION    abs2,ABSOLUTE
SECTION    found2,,"foundclass"
SECTION    2
```

5.2 SYMBOL DEFINITION

Symbol definition directives EQU, FEQU, REG, and SET provide the only method by which a symbol appearing in the label field may be assigned a 'value' other than that corresponding to the current location counter. The following Symbol Definition Directives are described in this section:

| | |
|------|------------------------------------|
| EQU | Equate Symbol Value |
| FEQU | Equate Floating Point Symbol Value |
| REG | Define Register List |
| SET | Set Symbol Value |

5.2.1 EQU - EQUATE SYMBOL VALUE

Syntax

```
<label> EQU <expression> [<comment>]
```

The EQU directive assigns the value of the expression in the operand field to the symbol in the label field. The label and expression follow the rules given in the *Source Program Coding* chapter. The label and operand fields are both required, and the label cannot be defined anywhere else in the program.

Any valid expression is allowed in the operand field of an EQU, including forward and complex.

Example

```
STRT      EQU      *      This is the start location
```

5.2.2 FEQU - EQUATE FLOATING POINT SYMBOL VALUE

Syntax

```
<label> FEQU[.<size>] <value> [<comments>]
```

where:
<size> = S | D | X | P (S is default)

FEQU directive assigns the floating-point value in the operand field to the symbol in the label field. The label and value follow the rules given in the *Source Program Coding* chapter. The operand fields are both required, and the label cannot be defined anywhere else in the program. Note that <value> is stored as a string and only converted to its binary format when it is used in instructions. <value> may be a floating-point decimal string or a floating-point hexadecimal value as defined in the *Source Program Coding* chapter. A warning is generated whenever the number of bits required to represent the specified precision is exceeded.

Example

```
OP1      FEQU.X      2.3444
OP2      FEQU.S      :23444
```

5.2.3 REG - DEFINE REGISTER LIST

Syntax

<label> **REG** *<reg_list>* [*<comment>*]

The REG directive assigns a value to *<label>* that can be translated into the register list mask format used in the MOVEM instruction. The label cannot be redefined as a Class 2 symbol anywhere else in the program.

<reg_list> is of the form R1[-R2][/R3[-R4]]...

Example

```

SAVE      REG      A1-A5/D0/D2-D4/D7
*      Following two statements are then equivalent
          MOVEM.L   SAVE, -(A7)
          MOVEM.L   A1-A5/D0/D2-D4/D7, -(A7)

```

5.2.4 SET - SET SYMBOL VALUE

Syntax

<label> **SET** *<expression>* [*<comments>*]

SET directive assigns the value of the expression in the operand field to the symbol in the label field. Thus, the SET directive is similar to the EQU directive. However, the SET directive allows the symbol in the label field to be redefined by other SET directives in the program. The label and operand fields are both required.

As with EQU, any valid expression is allowed in the operand field of a SET, including forward and complex.

Example

```

THIRTY    SET      LAB_AT_30

```

5.3 DATA DEFINITION/STORAGE ALLOCATION

The directives in this section provide the only means by which object code may begin or end on odd byte boundaries. All instructions and all word or longword size data must begin and end on even byte boundaries. Odd byte alignment is allowed only for the DC.B, DS.B, and DCB.B directives. All other operations which generate object code are preceded by a zero fill byte if word boundary alignment is required.

The following directives are described in this section:

| | |
|---------|------------------------------|
| COMLINE | Command Line (unimplemented) |
| DC | Define Constant |
| DCB | Define Constant Block |
| DS | Define Storage |

5.3.1 COMLINE - UNIMPLEMENTED

The COMLINE directive is not implemented. It is read and ignored. In the Motorola assembler it allows the user to specify the command line.

5.3.2 DC - DEFINE CONSTANT

Syntax

[<label>] **DC**[,<fmt>] <operand>[,<operand>...]

where:

<fmt> = B | W | L | S | D | X | P (W is default)

<operand> = link-time constant expression

The DC directive defines a constant in memory. The DC directive may have one or more operands, which are separated by commas. The operand field may contain the actual value (decimal, hexadecimal, or ASCII). Alternatively, the operand may be a symbol or expression which can be evaluated either by the assembler or the linker. The constant is aligned on a word boundary if word (.W), longword (.L), single precision (.S), double precision (.D), extended precision floating-point (.X) or packed BCD (.P) is specified. Alignment is on a byte boundary if byte (.B) is specified. The type of the operand must be floating-point if and only if the format is S, D, X, or P.

The following rules apply to size specifications on DC directives with ASCII strings as operands:

- DC.B
One byte is allocated per ASCII character.
- DC.W
The string begins on a word boundary. If the string address contains an odd number of characters, a zero fill byte follows the last character.
- DC.L
The string begins on a word boundary. If the string length is not a multiple of four bytes, the last longword is zero filled.

Examples of ASCII Strings

| Directive | Result |
|------------------|---------------|
|------------------|---------------|

| | |
|------------------|-------------------------------------------------------------------------------------------------------------------|
| DC.B 'ABCDEFGHI' | Memory has nine contiguous bytes with the ASCII characters A through I. |
| DC.B 'E' | Memory has characters "EJ" (\$454A) in DC.B 'J' contiguous bytes. |
| DC.B 'E' | Memory has \$45004500 in contiguous bytes, DC.W 'E' the first zero byte being an odd byte fill as outlined above. |
| DC 'X' | Memory has \$5800 in contiguous bytes. |
| DC.L '12345' | Memory has \$3132333435000000 in contiguous bytes. |

Examples of Numeric Constants**Directive Result**

DC.B 10,5,7

Memory has three contiguous bytes with the decimal values 10, 5, and 7 in their respective bytes.

DC.W 10,5,7

Each operand is contained in a word. The value 10 is contained in the first word, right justified. The value 5 is in the second word, and the value 7 is in the third word.

DC.L 10,5,7

Each operand is contained in a longword. The value 10 is contained in the first longword (4bytes) right justified. The value 5 is in the second longword, and the value 7 is in the third longword.

DC LABEL+1

The generated value is the address of LABEL plus 1 in a word size operand.

DC \$FF,\$10,\$AE

Rules for hexadecimal are the same as decimal.

DC.S 3.1415

A single precision floating-point value is created.
(68881/68882/68040/68060 only)

DC.D 2.54

A double precision floating-point value is created.
(68881/68882/68040/68060 only)

DC.X 6.0224E23

An extended precision floating-point value is created.
(68881/68882/68040/68060 only)

DC.X :ABCD10

An extended precision floating-point hex value is created.
(68881/68882/68040/68060 only)

DC.P 3.00E9 A packed BCD value is created.



For DC.X, “E” can only be a hex digit, not an exponent.

If the resulting value in an operand expression exceeds the size of the operand, an error is generated. For example,

DC.B \$FFF This causes an error because \$FFF cannot be represented in 8 bits.

DC \$FFF6F This causes an error because \$FFF6F cannot be represented in 16 bits.

5.3.3 DCB - DEFINE CONSTANT BLOCK

Syntax

[<label>] DCB[.<size>] <length>,<value>

where:

<size> = B | W | L | S | D | X | P (W is default)

<value> = integer, character, or floating-point value

DCB directive causes the assembler to allocate a block of bytes, words, or long words, quad words (.D), or hex words (.X or .P) depending upon the <size> specified. If <size> is omitted, word (.W) is the default size. The block length is specified by the absolute expression <length>, which may not contain undefined, forward, or external references. The initial value of each storage unit allocated will be the sign-extended expression <value>. <value> may be relocatable and may contain forward references. <length> must be greater than zero.

Example

```
CLEAR      DCB  80,0 Clears one line to spaces
```

5.3.4 DS - DEFINE STORAGE

Syntax

[<label>] DS[.<fmt>] <objects>

where:

<fmt> = B | W | L | S | D | X | P (W is default)

<objects> = The number of objects to be reserved (a pass 1 constant).



The DS directive is used to reserve memory locations. The contents of the memory reserved are not initialized in any way. The *<fmt>* values of S, D, X, and P are only used for floating-point values, and so only apply when assembling for the 68881/68882/68040/68060.

Example

```

PT1      DS.B      10      Reserve 10 bytes
PT1      DS        $10     Reserve 16 words
PT2      DS.L      100     Reserve 100 long words
*        DS.D      10      Reserve 10 8-byte
                        double words
```

The label will reference the lowest address of the defined storage area. The storage area is aligned to a word boundary unless *<fmt>* is “B”.

```

      DS.B      1      Reserve one byte
      DS        0      Set location counter
*                      to even boundary
```

The operand must be absolute and may not contain forward, undefined, or external references.

5.4 LISTING CONTROL AND OUTPUT OPTIONS

The following Listing Control and Output Options are described in this section:

| | |
|-----------------|----------------------------|
| FAIL | Programmer Generated Error |
| FORMAT/NOFORMAT | Format Options |
| LIST/NOLIST | List Options |
| LLEN | Line Length |
| NOOBJ | No Object |
| OPT | General Option Selection |
| PAGE/NOPAGE | Pagination Options |
| SPC | Space Between Source Lines |
| STTL | Subtitle |
| TTL | Title |

5.4.1 FAIL - PROGRAMMER GENERATED ERROR

Syntax

FAIL <message>

The FAIL directive causes a warning message to be printed by the assembler. The FAIL directive is normally used in conjunction with conditional assembly directives for exceptional condition checking. The assembly proceeds normally after the warning has been printed. The argument is printed as the warning message.

5.4.2 FORMAT/NOFORMAT - UNIMPLEMENTED

FORMAT and NOFORMAT are not implemented. They are read and ignored. In the Motorola assembler it allows the user to control the formatting of the assembler's listing file.

5.4.3 LIST/NOLIST - CONTROL LISTING GENERATION

Syntax

LIST
NOLIST
NOL

Print or do not print the assembly listing on the output device. The LIST option is selected by default. The source text following the LIST directive is printed until an END or NOLIST directive is encountered.

5.4.4 LLEN - UNIMPLEMENTED

The LLEN directive is not implemented. It is read and ignored. In the Motorola assembler it allows the user to specify the length of a listing line.

5.4.5 NOOBJ - UNIMPLEMENTED

The NOOBJ directive is not implemented. It is read and ignored. In the Motorola assembler it allows the user to request that no object module be produced.

5.4.6 OPT - ASSEMBLER OPTIONS

Syntax

OPT <option>[,<option>...] [<comment>]

Follows the command format. The available options are:

| | |
|--------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| A | Absolute address. All non-indexed operands which reference either labels or the current assembler location counter (*) is resolved as absolute. |
| NOA | Disable A (default). |
| BRB BRW | Generates default branch size of 16 bits. |
| BRL | Forward branch long (default). Forward references in relative branch instructions (Bcc, BRA, BSR) will assume the longer form (16-bit displacement, yielding a 4-byte instruction). A 32-bit displacement is assumed unless the directive OPT OLD is in effect (68020-plus/CPU32 only). |
| BRS | Forward branch short. As with BRL, but using the shorter or form (8-bit displacement, yielding a 2-byte instruction). |
| CRE | Print cross-reference table at end of source listing. This option must precede first symbol in source program. If this option is not in effect, only the symbol table is printed. |
| EXT | Not implemented. |
| NOEXT | Not implemented. |
| FRL | Forward reference long (default). Forward references in the absolute format assumes absolute long mode (32-bit). |
| FRS | Forward reference short. Forward references in the absolute format assumes absolute short mode (16-bit). |
| L | Turn on source listing. |
| NOL | Turn off source listing. |
| MC | Print macro calls (default). |

| | |
|-----------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| NOMC | Opposite of MC. |
| MD | Print macro definitions (default). |
| NOMD | Opposite of MD. |
| MEX | Print macro expansions. |
| NOMEX | Opposite of MEX (default). |
| MEXG | List only those macro expansions that generate code. |
| NOMEXG | Turn off MEXG flag. |
| OLD | Interpret the branch size code .L as being a 16-bit branch. Also interpret future uses of OPT BRL as referring to forward 16-bit branches. |
| NOOLD | Change back to new branch size meanings for size .L (68020-plus/CPU32 only). |
| P | Turn on listing paging. |
| NOP | Turn off listing paging. |
| PCO | PC relative addressing within ORG. Employ relative addressing, when possible, on backward references occurring in an ORG section. |
| NOPCO | Disable PCO (default). |
| PCS | Force PC relative addressing within SECTION. Forces PC relative addressing (whenever such an addressing mode is legal) in an instruction which occurs within a relocatable SECTION and references an operand in a relocatable SECTION (need not be the same SECTION as the instruction). Failure to resolve such a reference into a 16-bit displacement from the PC results in an error. This option may be used to force position-independent code (refer to the <i>Position-independent Code</i> chapter); however, this option does not force PC relative addressing of absolute operands (defined in ORG section) or unknown forward references. |
| NOPCS | Disable PCS (default). |
| P=<type> | Not implemented. |

| | |
|--------------|------------------------------------------------------------------|
| S | Turn on symbol table listing. |
| NOS | Turn off symbol table listing. |
| PSA | List expanded instructions from structured assembler constructs. |
| NOPSA | Turn off expanded instruction listing. |
| TRM | Trim comments from listing. |
| NOTRM | Do not trim comments from listing. |
| U | Turn on listing of unassembled lines in conditional assembly. |
| NOU | Turn off listing of unassembled lines in conditional assembly. |

The following options are **not** implemented:

ASM
NOASM
CEX
NOCEX
CL
NOCL
D

To generate debug output, use the command-line debugging option. See the *Tutorial* chapter of the *Getting Started Manual* for more information.

EQU
NOEQU
FMT
NOFMT
G
NOG
LLE
O
NOO
REL

5.4.7 PAGE/NOPAGE - CONTROL PAGINATION

Syntax

PAGE [*size*]
NOPAGE

Advance the paper to the top of the next page. The PAGE directive does not appear on the program listing. No label is used, and no machine code results. The optional size argument is used as the number of lines per page. A negative value for <*size*> turns off paging.

NOPAGE turns off pagination. Output lines are printed continuously with no page headings or top and bottom margins.

5.4.8 SPC - SPACE BETWEEN SOURCE LINES

Syntax

SPC [*n*]

Output *n* blank lines on the assembly listing. This has the same effect as putting *n* blank lines in the assembly source. The default value for *n* is 1.

5.4.9 STTL - SET SUBTITLE

Syntax

STTL <*subtitle string*>

Print the <*subtitle*> string on the second line of each page. A subtitle consists of up to 60 characters. The same subtitle will appear on all successive pages until another STTL directive is encountered. In order to print a subtitle on the first listing page, the STTL directive must precede the first source line which will appear on the listing.

5.4.10 TTL - SET TITLE

Syntax

TTL <*title string*>



Print the *<title>* string at the top of each page. A title consists of up to 60 characters. The same title will appear at the top of all successive pages until another TTL directive is encountered. In order to print a title on the first listing page, the TTL directive must precede the first source line which will appear on the listing.

5.5 EXTERNAL SYMBOL CONTROLS

The following External Symbol Controls are described in this section:

| | |
|------|-----------------------------------|
| IDNT | Relocatable Identification Record |
| XDEF | External Symbol Definition |
| XREF | External Symbol Reference |

5.5.1 IDNT - RELOCATABLE IDENTIFICATION RECORD

Syntax

```
<name> IDNT <version_string>
```

The assembler takes the provided information and puts it in the object module as a .ID statement. This statement is ignored by subsequent processors, but is passed on for informational purposes only. *<version>* must be supplied as a quoted string.



<name> is NOT considered a label, and may not be used elsewhere in the assembly.

5.5.2 XDEF - EXTERNAL SYMBOL DEFINITION

Syntax

```
XDEF <symbol>[,<symbol>...] [<comment>]
```

The XDEF directive specifies symbols defined in the current module that are to be globally visible, and can therefore be referenced by other modules.

Example

```

XDEF var1,var2,var2
*      These names may now be referenced in other
*      modules

```

5.5.3 XREF - EXTERNAL SYMBOL REFERENCE**Syntax**

XREF[.S] [[<section>:]<symbol>[,<symbol>]...]

This directive specifies symbols referenced in the current module but defined in other modules. Each symbol is associated with the specified <section> number which it follows. Symbols may occur in any section, including an absolute ORG section, if no <section> designation is specified.

“S” indicates the XREF symbols should be directly addressed through absolute short mode. Remember, however, that the location of the symbols in low memory is the responsibility of the user.

Example

```

XREF      Simple_var
XREF      AA,2:E2,3:E3,B3,C3

```

The symbol AA can be in any section; E2 is in section 2; and E3, B3, and C3 are in section 3.

5.6 INTERNAL ASSEMBLY CONTROLS

The directives that are described in this section are put out by our 68K C compiler. Compiling with the “-ia” option, the compiler will produce real assembly language output. These directives are needed to provide the connection between the compiler produced assembly and the compiler libraries and to pass symbolic debug information through the assembler:

| |
|----------|
| _BRINGIN |
| _DEBSYM |
| _DGROUP |

5.6.1 _BRINGIN DECLARE EXTERNAL SYMBOL

Syntax

_BRINGIN <symbol>

This directive tells the assembler to emit an external reference for the named symbol into the object module. This causes the link editor to bring the object module that defines this symbol into the link.

5.6.2 _DEBSYM PUT OUT DEBUGGING INFORMATION

Syntax

_DEBSYM <string>[,<operand>]

This directive causes the assembler to generate a line of symbolic debugging information. Lines that describe data symbol positions are written to a temporary file which is later combined into the object module by the compiler utility. Line that describe C source line positions are written directly into the object module.

5.6.3 _DGROUP DEFINE DATA GROUP

Syntax

_DGROUP <symbol>

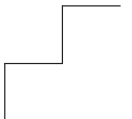
The compiler addresses global data via the A5 register. The global data is divided into two segments, idata for initialized data and udata for uninitialized data. In order to address two different segments off one register, a virtual segment or “group” called “data” is used. The A5 register points at data, and the link editor ensures that idata and udata are located in one 64k byte area.

This directive causes the assembler to emit a group definition into the object module. The name symbol (always “data”) is the group name. The statement puts the udata and/or idata segments into the group data if they are present.

CHAPTER

6

MACRO OPERATIONS AND CONDITIONAL ASSEMBLY



6

CHAPTER

This chapter describes the macro and the conditional assembly capabilities of the assembler. These features can be used in any program.

6.1 MACRO OPERATIONS

Programming applications frequently involve the coding of a repeated pattern of instructions that, within themselves, contain variable entries at each iteration of the pattern, or basic coding patterns subject to conditional assembly at each occurrence. In either case, macros provide a shorthand notation for handling these patterns. Having determined the iterated pattern, the programmer can, within the macro, designate fields of any statement as variable. Thereafter, by invoking a macro, the programmer can use the entire pattern as many times as needed, substituting different parameters for the designated variable portions of the statements.

Macro usage can be divided into two basic parts: definition and expansion.

When the pattern is defined, it is given a name. This name becomes the mnemonic by which the macro is subsequently invoked (called). The name of a macro definition should not be the same as an existing instruction mnemonic, an assembler directive, or a previously defined macro.

Expansion occurs when the previously defined macro is called (invoked). The macro call causes source statements to be generated. The generated statements may contain substitutable arguments. The statements that may be generated by a macro call are relatively unrestricted as to type. They can be any processor instruction, almost any assembler directive, or any previously defined macro. Source statements generated by a macro call are subject to the same conditions and restrictions as programmer-generated statements.

The invocation of a macro requires that the macro name appear in the operation field of a source statement. Most arguments are placed in the operand field. Appropriate arguments selected according to the macro definition cause the assembler to produce in-line coding variations of the macro definition.

The effect of a macro call is the same as an open subroutine in that it produces in-line code to perform a predefined function. The in-line code is inserted in the normal flow of the program so that the generated instructions are executed in-line with the rest of the program each time the macro is called.

6.1.1 MACRO DEFINITION

The definition of a macro consists of three parts:

1. The header: *<label>* MACRO

The *<label>* of the MACRO statement is the “name” by which the macro is later invoked. This name must be a unique class 1 symbol. A macro name may not have a period (.) as any character other than the first.

2. The body

The body of a macro is a sequence of standard source statements. Macro parameters are defined by the appearance of argument designators within these source statements. Legal macro-generated statements include the set of Motorola 68000 family assembly language instructions, assembler directives, structured syntax statements, and calls to other, previously defined macros. However, macro definitions may not be nested.

3. The terminator: ENDM

6.1.2 MACRO INVOCATION

The form of a macro call is:

[<label>] <name>[.<qualifier>] [<parameters>]

Although a macro may be referenced by another macro prior to its definition in the source module, the macro must be defined before its first in-line expansion. The name of the called macro appears in the operation field of the source statement; parameters may appear as qualifiers to the macro name and/or in the operand field of the source statement, separated by commas.

The macro call produces in-line code at the location of the invocation, according to the macro definition and the parameters specified in the macro call. The source statements so generated are then assembled, subject to the same conditions and restrictions affecting any source statement. Nested macro calls are also expanded at this time.

6.1.3 MACRO PARAMETER DEFINITION AND USE

Up to 36 different, substitutable arguments may appear in the source statements which constitute the body of a macro. These arguments are replaced by the corresponding parameters in a subsequent call to that macro.

Arguments are designated by a backslash character (\), followed by a digit (0 through 9) or an upper case letter (A through Z). Argument designator \0 refers to the qualifier appended to the macro name; parameters in the operand field of the macro call refer to argument designations \1 through \9 and \A through \Z, in that order.

Argument substitution at the time of a macro call is handled as a literal (string) substitution. The string corresponding to a given parameter is substituted literally whenever that argument designator occurs in a source statement as the macro is expanded. Each statement generated in this expansion is assembled in-line. (Note that, if a qualifier is present, argument 0 begins with the first character following the period which separates the qualifier from the macro name).

It is possible to specify a null argument in a macro call by an empty string (not a blank); except for 0, it must still be separated from other parameters by a comma. In the case of a null argument referenced as a size code, the default size code (W) is implied; when a null argument itself is passed as an argument in a nested macro call, a null argument is passed. All parameters have a default value of null at the time of macro call.

If an argument has multiple parts or contains commas or blanks, the entire argument must be enclosed within angle brackets (< and >). Such arguments must still be separated from other arguments by commas. A bracketed argument with no intervening character is treated as a null argument. Embedded brackets must occur in pairs. Parameter 0 may not be bracketed and, hence, may not contain blanks (although commas are legal). Note that a macro argument may not contain the characters "<" or ">" unless they occur as part of the argument bracketing.

6.1.4 LABELS WITHIN MACROS

To avoid the problem of multiply-defined labels resulting from multiple calls to a macro which employs labels in its source statements, the programmer may direct the assembler to generate unique labels on each call to a macro.

Assembler-generated labels include a string of the form .nnn, where nnn is a 3-digit value. The programmer may request an assembler-generated label by specifying \@ in a label field within a macro body. Each successive label definition which specifies a \@ directive will generate successive values of .nnn, thereby creating unique labels on repeated macro calls. Note that \@ may be preceded or succeeded by additional characters for clarity and to prevent ambiguity.

References to an assembler-generated label always refer to the label of the given form defined in the current level of the current macro expansion. Such a label is referenced as an operand by specifying the same character string as that which defines the label.

6.1.5 THE MEXIT DIRECTIVE

The MEXIT directive terminates the macro source statement generation during expansion. It may be used within a conditional assembly structure to skip any remaining source lines up to the ENDM directive. All conditional assembly structures pending within the macro currently being expanded are also terminated by the MEXIT directive. The MEXIT Directive takes an optional expression. It exits if the expression is true (non-zero). If the MEXIT Directive is not given an argument, comments must be delimited with a '!'.

Example

```
SAV2      MACRO
           MOVE.L    \1, SAVET    SAVE 1ST ARGUMENT
           MOVE.L    \2, SAVET+4  SAVE 2ND ARGUMENT
           IFC       3,           IS THERE A 3RD
*                                     ARGUMENT?
           FAIL      10000        DID THE ASSEMBLER
*                                     FAIL THRU HERE?
           MEXIT                                NOEXIT FROM MACRO
           ENDC
           MOVE.L    \3, SAVET+8  SAVE 3RD ARGUMENT
ENDM
```

6.1.6 THE NARG SYMBOL

The symbol NARG is a special symbol when referenced within a macro expansion. The value assigned to NARG is the index of the last argument passed to the macros in the parameter list (including nulls). NARG is undefined outside of macro expansion and may be referenced as a Class 1 or 2 user-defined symbol outside of a macro expansion.

6.1.7 IMPLEMENTATION OF MACRO DEFINITION

When the sequence of source statements:

```
MAC1  MACRO
      <stmt1>
      <stmt2>
      .
      .
      .
      <stmtn>
      ENDM
```

is encountered in a source program, the following actions are performed:

1. The symbol table is checked for a Class 1 symbol entry of “MAC1”. If such an entry is already present, a redefined symbol warning is generated; otherwise, an entry is placed in the symbol table, identifying MAC1 as a macro.
2. Starting with the line following the MACRO directive, each line of the macro body is saved in a character sequence identified with MAC1. In the example, stmt1 through stmtn are saved in this manner. No object code is produced at this time.
3. Normal processing resumes with the line following the ENDM directive.

6.1.8 IMPLEMENTATION OF MACRO EXPANSION

When the statement:

```
MAC1.<qualifier>
<param1>,<param2>,...,<paramn>
```


is encountered in a source program calling the previously defined macro MAC1 (above), the following actions are performed:

1. The line is scanned for parameters which are saved as literals or null values, one such value in each of the 36 parameter record fields. No object code is produced.
2. Macro expansion consists of the retrieval of the source lines which comprise the macro body. Each line is retrieved in turn, with special character pairs replaced by parameter strings or assembler-generated label strings.

If a backslash character \ is followed by either a digit (0 through 9) or an upper case letter (A through Z), the two characters are replaced by the literal string which corresponds to that parameter on the macro invocation line(s).

A character sequence which includes \@ is replaced by an assembler-generated label. An assembler-generated label is uniquely identified by the characters preceding and/or appended to the \@ sequence and the macro invocation in which the reference occurs. Such labels may appear anywhere in the source line and always refer to the current macro expansion.

3. When a line has been completely expanded, it is assembled as any other source input line. At this time, any errors in the syntax of the expanded assembly code are found. Expanded lines longer than 80 characters are truncated, and an error is generated.

If a nested macro call is encountered, the nested macro expansion takes place recursively. There is no set limit to the depth of macro call nesting.

6.2 CONDITIONAL ASSEMBLY

Conditional assembly allows the programmer to write a comprehensive source program that can cover many conditions. Assembly conditions may be specified through the use of arguments in the case of macros and through definition of symbols via the SET and EQU directives. Variations of parameters can then cause assembly of only those parts necessary for the specified conditions.

The I/O section of a program, for example, will vary, depending on the target environment. Conditional assembly directives can include or exclude an I/O section, based on a flag set at the beginning of the assembly.

6.2.1 CONDITIONAL ASSEMBLY STRUCTURE

There are two conditional assembly structures available: IFC-ELSEC-ENDC and REPEATC-ENDR. IFC-ELSEC-ENDC blocks allow conditional assembly and are valid in any part of an assembly language program. REPEATC-ENDR blocks also allow conditional assembly, and are only valid within a macro definition.

The ELSEC, REPEATC, and ENDR constructs are only allowed if the “select TASKING extensions” option is in effect. See the *68000 Family Assembler* chapter in the *User's Manual* for a description of these extensions.

The IF conditional assembly structure consists of three parts:

1. *The header.* There are two conditional header clauses recognized by the assembler. The first form compares the equality of two strings:

```
IFxx <string1> , <string2>
```

“xx” specifies either the string compare (C) condition or the string not compare (NC) condition, representing string equality and inequality, respectively. The result of the string comparison, along with the “xx” condition, determines whether the body of the conditional structure will be assembled. Either string may contain embedded commas or spaces. An apostrophe that occurs within a string must be specified by double apostrophes.

The second form of the conditional clause compares with an expression against zero:

```
IFxx <expression>
```

“xx” specifies a conditional relation between the expression and the value zero. The result of this comparison at assembly time determines whether the body of the conditional structure will be assembled. Valid conditional relation codes include:

| | | |
|----|--------------|---|
| EQ | expression= | 0 |
| NE | expression<> | 0 |
| LT | expression< | 0 |
| LE | expression<= | 0 |
| GT | expression> | 0 |
| GE | expression>= | 0 |

Because of the nature of this comparison, the expression must be absolute. No forward references are allowed.

2. *The body.* The body of the conditional assembly structure consists of a sequence of standard source statements. There is no set limit to the depth of conditional assembly nesting; if such nesting occurs, an ENDC terminator must be specified for each structure.

There may be an ELSE clause in the body. The keyword for for this is ELSEC.



ELSEC may only be used if the “select TASKING extensions” option is in effect. See the *Assembler* chapter in the *User’s Manual* for more information.

3. *The terminator* ENDC. When an IFxx directive is encountered, the specified condition is evaluated. If the condition is true, the statements constituting the body of the conditional assembly structure are each assembled in turn. If the relation is false, the entire conditional assembly structure is ignored; the ignored lines are not included in the assembly listing. By specifying the OPT NOCL option, the header and terminator lines are ignored for listing purposes.

IFxx and ENDC directives may not be labeled.

The REPEATC-ENDR construct has a similar structure:

1. *The header:* REPEATC <expr1>[,<expr2>]. Both <expr1> and <expr2> must be assembly time absolute expressions. No forward references are allowed. If <expr1> is equal to zero (false), then statements up to the ENDR are ignored. Otherwise, the statements are assembled and the assembler repeats the process again until <expr1> is equal to zero. A REPEATC block stops iterating when the specified expression maximum, <expr2> is reached. If <expr2> is not specified, then the REPEATC block stops after 255 iterations.
2. *The body.* The body of a REPEATC–ENDR can contain any assembly language statements, including complete IFC–ELSEC–ENDC and REPEATC–ENDR constructs. IFC–ENDIF and REPEATC–ENDR blocks may not cross the boundary of a macro expansion or the boundaries of each other.
3. *The terminator:* ENDR. This terminates the body of the REPEATC construct.

Testing for null parameters may be done via the string compare form of the conditional assembly. To assemble conditionally if parameter 1 is null, either of the following is correct:

```
IFC " , '\1'
```

or

```
IFC '\1' , "
```

To assemble conditionally if a parameter is present, use either of the IFNC formats analogous to the above two.

A conditional assembly structure is also terminated by a MEXIT directive. It is an error if a conditional assembly block is not terminated in the same macro call and at the same level that it was begun in.

6.2.2 EXAMPLE OF MACRO AND CONDITIONAL ASSEMBLY USAGE

The following example illustrates most of the features of macro and conditional assembly structures. The assembly code is shown as it appears, without line numbers or object code. Note that angle brackets (< >) shown in examples are required characters.

Example of Nested Macros

```

MAC0                MACRO
                     MOVE.\0  \1
                     CLR.L    \2
                     ENDM

MAC1                MACRO
                     MOVE.\    #\1,D\2
                     IF\3      \1          CONDITIONAL
                     ADD.\     #1,D\2
                     IF\3      \1-5        NESTED CONDITIONAL
                     ADD.\0     #2,D\2     \4
                     ENDC          END NESTED CONDITIONAL
                     ENDC          END CONDITIONAL

LAB\@               CLR.L      D1
                     MOVE.\0    D\2,(A0)+
                     B\3        L\@END
                     BRA        LAB\@

L\@END              \5.\0     #1,D\2
                     IFLE       \1
                     MAC0.\0    <D\2,(A0)>,A\2 NESTED MACRO CALL
                     ENDC
                     ENDM

                     OPT        MEX,NOCL
                     MAC1.L      7,3,GT,<TEST PASSES>,ADD
* Expansion is equivalent to following lines
                     MOVE.L      #7,D3
                     ADD.L       #1,D3
                     ADD.L       #2,D3          TEST PASSES

LAB.001             CLR.L      D1
                     MOVE.L      D3,(A0)+
                     BGT         L.002END
                     BRA         LAB.001

L.002END            ADD.L       #1,D3

                     MAC1.W      0,6,NE,<ERROR HERE>,SUB
* Expansion is equivalent to following lines
                     MOVE.W      #0,D6
LAB.003             CLR.L      D1
                     MOVE.W      D6,(A0)+
                     BNE         L.004END
                     BRA         LAB.003

L.004END            SUB.W       #1,D6
                     MOVE.W      D6,(A0)
                     CLR.L       A6

```

Examples of REPEATC-ENDR

```

NUMSTR      MACRO
X            SET          1
             IFGT         X-9
             FAIL         "Argument to NUMSTR out of range"
             ENDC
             IFLT         X
             FAIL         "Argument to NUMSTR out of range"
             ENDC
             REPEATC      1,X
             DC.B         0+X          SAME AS X IF 0<=X<=9
X            SET          X-1
             ENDR
             DC.B         0
             ENDM

             NUMST        3

```

* Expansion is equivalent to following lines

```

             DC.B         0+3
             DC.B         0+2
             DC.B         0+1
             DC.B         0

```

```

POWERS      MACRO
X            SET          \2
             REPEATC      1,\1
             DC.\1        X
X            SET          X*\2
             ENDR
             ENDM

             POWERS.W     4,4

```

* Expansion is equivalent to following lines

```

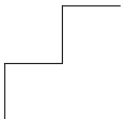
             DC.W         4
             DC.W         16
             DC.W         64
             DC.W         256

```



CHAPTER 7

STRUCTURED CONTROL STATEMENTS



7 | CHAPTER

An assembly language provides an instruction set for performing certain rudimentary operations. These operations, in turn, may be combined into control structures — such as loops (for, repeat, while) or conditional branches (if-then, if-then-else). To simplify the process of coding these constructs, this assembler accepts formal, high level directives that specify these control structures, generating, in turn, the appropriate assembly language instructions for their efficient implementation. This use of structured control statement directives improves the readability of assembly language programs, without compromising the desirable aspects of programming in an assembly language.

7.1 KEYWORD SYMBOLS

The following Class 1 symbols, used in the structured syntax, are reserved keywords (directives):

| | | |
|------|------|--------|
| ELSE | ENDW | REPEAT |
| ENDF | FOR | UNTIL |
| ENDI | IF | WHILE |

The following symbols are required in the structured syntax. All keywords are reserved:

| | | |
|-----|--------|----|
| AND | DOWNT0 | TO |
| BY | OR | |
| DO | THEN | |



AND and OR are reserved instruction mnemonics, however.

7.2 SYNTAX

This section describes the formats for the IF, FOR, REPEAT, and WHILE statements. They are spaced to show the line separations required for Class 1 symbol usage. Syntactic variables used in the formats are as follows:

<expression> A simple or compound expression (see the *Simple and Compound Expressions* section).

<stmtlist> Zero or more assembler directives, structured control statements, or executable instructions.



An assembler directive (see the *Assembler Directives* chapter) occurring within a structured control statement is examined exactly once – at assembly time. Thus, the presence of a directive within a FOR, REPEAT, or WHILE statement does not imply repeated occurrence of an assembler directive; nor does the presence of a directive within an IF-THEN-ELSE statement imply a conditional assembly structure (see the *Structured Control Statements* chapter).

For correct recognition, the statements in <stmtlist> must not appear on the same line as the structured syntax symbols.

<size> The value B, W, or L, indicating a data size of byte, word, or longword, respectively. With the keyword FOR, <size> is a single code applying to <op1>, <op2>, <op3>, and <op4>. With the keywords IF, UNTIL, and WHILE, <size> indicates the size of the operand comparison in the subsequent simple expression (refer to paragraph 5.3.4 for a compound expression). Note that structured syntax statements rely on the underlying opcodes and the restrictions these opcodes place on arguments to the statements. For example, the structured syntax statement

```
FOR.B D7 = #0 to #255 DO
```

generates code without warning but does not execute as expected. This is because the comparison opcode CMP does a signed comparison and hence deals with numbers in the range **-128 ... 127** instead of **0 ... 255**.

<extent> The value S or L, indicating that the branch extent is short or long, respectively. This is appended to the keywords THEN, ELSE, and DO, to force the appropriate extent of the generated forward branch over the subsequent <stmtlist>. The default extent for the Motorola 68020-plus is determined by the forward branch option directive (OPT BRS, OPT BRB, OPT BRW, or OPT BRL) currently in effect.

<op1> A user-defined operand whose memory/register location holds the FOR counter. **This must be a data or address register.**

- <op2>** The initial value of the FOR counter. The effective address may be any mode. Immediate operands must be preceded by a # sign.
- <op3>** The terminating value for the FOR counter. The effective address may be any mode. Immediate operands must be preceded by a # sign.
- <op4>** The step (increment/decrement) for the FOR counter each time through the loop. If not specified, it defaults to a value of #1. The effective address may be any mode. Immediate operands must be preceded by a # sign.

7.2.1 IF STATEMENT

Syntax

IF[.<size>] <expression> **THEN**[.<extent>]
 <stmtlist>
ENDI

OR:

IF[.<size>] <expression> **THEN**[.<extent>]
 <stmtlist>
ELSE[.<extent>]
 <stmtlist>
ENDI

If <expression> is true, execute the <stmtlist> following THEN;

If <expression> is false, execute <stmtlist> following ELSE, if present, or advance to next instruction.

Notes

- If an operand comparison <expression> is specified, the condition codes are set and tested before execution of <stmtlist>.
- In the case of nested IF-THEN-ELSE statements, each ELSE refers to the closest IF-THEN.

Example

```

IF.L D1 <LT> #10 THEN
MOVE D5, D6
ENDI

```

7.2.2 FLOATING-POINT STRUCTURED ASSEMBLER SYNTAX FOR THE IF STATEMENT

Syntax

```

IF[.<fmt>]    FPN <fpcc> <ea>    THEN
IF[.<fmt>]    <ea> <fpcc> FPN    THEN
IF.X          FPN <fpcc> FPM    THEN
IF           <fpcc>              THEN

```

where:

<fmt> = B | W | L | S | D | X | P (W is default)
 <fpcc> = A floating-point condition code, as defined in Table
 NO TAG.

This directive is similar to the non-floating-point IF syntax, except that the floating-point condition codes are used. When the assembler expands the structured IF statement with a floating-point condition code, <fpcc>, it must choose the true IEEE inverse of <fpcc>. For example, the code generated for:

```
IF.X FP3 <FGT> #3.3 THEN
```

(where GT is one value of fpcc and #3.3 is a required constant value)

would be:

```

FCMP.X    #3.3,FP3
FBNGT     ELSECLAUSE
....
true clause code
BRA       PAST
ELSECLAUSE
....
false clause code
PAST
....

```



The branch following the FCMP is a FBNGT rather than a FBLE because FBNGT is the IEEE inverse of FBGT.

7.2.3 FOR STATEMENT

Syntax

```
FOR[.<size>] <op1> = <op2> TO <op3>  
    [BY <op4>] DO[.<extent>] <stmtlist>  
ENDF
```

OR:

```
FOR[.<size>] <op1>=<op2> DOWNTO  
    <op3> [BY <op4>] DO[.<extent>] <stmtlist>  
ENDF
```

These counting loops utilize a user-defined operand, <op1>, for the loop counter. FOR-TO allows counting upward, while FOR-DOWNTO allows counting downward. In both loops, the user may specify the step size, <op4>, or elect the default step size of #1. The FOR-TO loop is not executed if <op2> is greater than <op3> upon entry. Similarly, the FOR-DOWNTO loop is not executed if <op2> is less than <op3>.

Notes

- The condition codes are set and tested before each execution of <stmtlist>. This happens even if <stmtlist> is not executed.
- A step size of #1 may not be meaningful if the counter, <op1>, is used to index through word or longword size data.
- The FOR structure generates a move, a compare, and either an add or subtract. Therefore, if any of the four operands is an address register, <size> may not be B (byte).
- <op1> must be a data or address register.

Example

```
FOR COUNT = #4 TO #40 BY #4 DO  
    NOP      loop 10 times by steps of 4  
ENDF
```

7.2.4 REPEAT STATEMENT

Syntax

```
REPEAT
    <stmtlist>
UNTIL[.<size>] <expression>
```

<stmtlist> is executed repeatedly until <expression> is true.

Notes:

- The <stmtlist> is executed at least once, even if <expression> is true upon entry.
- If an operand comparison <expression> is specified, the condition codes are set and tested following each execution of <stmtlist>.

Example

```
REPEAT
    MOVE (A6)+, (A5)+
UNTIL <EQ>
```

7.2.5 WHILE STATEMENT

Syntax

```
WHILE[.<size>] <expression> DO[.<extent>]
    <stmtlist>
ENDW
```

The <expression> is tested before execution of <stmtlist>. While <expression> is true, <stmtlist> is executed repeatedly.

Notes:

- If <expression> is false upon entry, <stmtlist> is not executed.
- If an operand comparison <expression> is specified, the condition codes are set and tested before each execution of <stmtlist>. The condition codes are set and tested even if <stmtlist> is not executed.

Example

```
WHILE.B (A3) <NE> D2 DO
    MOVE.B (A5)+, D3
ENDW
```

7.3 SIMPLE AND COMPOUND EXPRESSIONS

Expressions are an integral part of IF, REPEAT, and WHILE statements. An expression may be simple or compound. A compound expression consists of no more than two simple expressions joined by AND or OR.

7.3.1 SIMPLE EXPRESSIONS

Simple expressions are concerned with the bits of the Condition Code Register (CCR). These expressions are of two types. The first type merely tests conditions currently specified by the contents of the CCR. The second type set up a comparison of two operands to set the condition codes, and afterwards tests the codes.

7.3.2 CONDITION CODE EXPRESSIONS

Fourteen tests (identical to those in the Bcc instruction) may be performed, based on the CCR condition codes. The condition codes, in this case, are preset by either a user-generated instruction or a structured operand-comparison expression. Each test is expressed in the structured control statement by a mnemonic enclosed in angle brackets (<\^>) as follows:

| | |
|------|------|
| <CC> | <LS> |
| <CS> | <LT> |
| <EQ> | <MI> |
| <GE> | <NE> |
| <GT> | <PL> |
| <HI> | <VC> |
| <LE> | <VS> |

Example

```
IF <EQ> THEN
    CLR.L D2
ENDI

REPEAT
    SUB D4,D3
UNTIL <LT>
```


7.3.3 OPERAND COMPARISON EXPRESSIONS

Two operands may be compared in a simple expression, with subsequent transfer of control based on that comparison. Such a comparison takes the form:

`<op1> <cc> <op2>`

where `<cc>` is a condition mnemonic enclosed in angle brackets, specifying the relation to be tested between `<op1>` and `<op2>`. When processed by the assembler, this expression translates to a compare instruction. For example,

`CMP <op1>, <op2>`

followed by a branch instruction (Bcc) which tests the relation specified. `<op1>` is normally, but not necessarily assigned to the first (leftmost) operand and `<op2>` to the second (rightmost) operand of the compare instruction.

Notes:

- A size may be specified for the comparison by appending a data size code (B, W, or L) to the directive, with W being the default. The only restriction is that a byte size code (B) may not be used in conjunction with an address register direct operand.
- Compare instructions require certain effective addressing modes for their operands. These modes are listed in Table 7-1. However, if the operands, `<op1>` and `<op2>`, are not listed in an order that generates a legal compare instruction (Table 7-1), but generates a legal compare if the operand order is reversed, the assembler reverses the operands when expanding the expression. To maintain the nature of the relation specified, the condition operator is adjusted, if necessary. For example,

`D2 <GT> #5`

is adjusted by the assembler to the equivalent of

`#5 <LE> D2`

Likewise,

```
A2 <EQ> (A5)
```

is adjusted to the equivalent of

```
(A5) <NE> A2.
```

This processing allows the user the flexibility of specifying the most meaningful operand order in the expression.

| | First Operand | Second Operand |
|------|------------------------|-------------------------|
| CMP | (All) | Data register direct |
| | (All) | Address register direct |
| CMPA | Immediate | (Data alterable) |
| CMPM | Postincrement register | Postincrement |
| | indirect | register indirect |

Table 7-1: Effective Compare Instruction Addressing Modes

If the operands, either as stated or reversed, do not yield a legal compare instruction, an error will result. For example, the statement:

```
IF (A1) <NE> (A2) THEN
```

results in an illegal address mode error during expansion. To avoid this error, a MOVE is required to effect a legal operand, such as:

```
MOVE (A2),D2 IF (A1) <NE> D2 THEN
```

Example

```
WHILE.B (A3) <NE> D2 DO          THIS EXPRESSION
    MOVE.B (A5)+,D2              IS LEGAL AS STATED.
ENDW

IF      D7 <LT> #10 THEN          THIS EXPRESSION
    BSR SUBR1                    IS REVERSED
ELSE
    MULS #2,D7
ENDI
```

7.3.4 COMPOUND EXPRESSIONS

A compound expression consists of two simple expressions joined by a logical operator. The Boolean value of the compound expression is determined by the Boolean values of the simple expressions and the nature of the logical operator (AND or OR).

The two simple expressions are evaluated in the order in which they are given. However, if an AND separates the expressions and the first expression is false, the second expression is not evaluated. Likewise, if an OR separates the expressions and the first expression is true, the second expression does not need to be evaluated, and the condition codes reflect the result of only the first simple expression.

A size may be specified for each operand comparison expression. The size of the comparison for the first expression may be appended to the directive, while the size of the comparison for the second expression may be appended to the keyword AND or OR. For example, in the statement:

```
IF.L D3 <GT> (A0) OR.B #Q <EQ> BUFFER1
```

the first comparison (between D3 and (A0)) is a longword comparison, and the second (between #Q and BUFFER1) is a byte comparison.

7.4 SOURCE LINE FORMATTING

The format of structured source statements is more restricted than the format of basic statements. The following paragraphs discuss the formatting requirements of structured statements as well as their appearance in the assembly listing.

7.4.1 CLASS 1 SYMBOL USAGE

Class 1 symbols are the assembler directives (including macro names), instruction mnemonics, and the structured control directives. Only one of these symbols is recognized on each source line. Thus, each directive (reserved keyword) of a structured control statement and each executable instruction generated by the programmer must be written on a separate source line. The following source line, for example, is in error:

```
REPEAT MOVE (A5),D2 UNTIL <EQ>
```

The MOVE and UNTIL symbols and their operands are not recognized as class 1 symbols, but are treated as part of the comment field of the REPEAT directive. Likewise, the following lines are in error:

```
IF      <VS> THEN JSR OVERFLOW
ELSE    JMP  (A3) ENDI
```

The JSR, JMP, and ENDI symbols and their operands are not recognized because they come after the THEN and ELSE keywords and are treated as comments. The correct format for these lines is as follows:

```
REPEAT
    MOVE (A5),D2
UNTIL <EQ>
```

and:

```
IF <VS> THEN
    JSR  OVERFLOW
ELSE
    JMP  (A3)
ENDI
```

7.4.2 NESTING OF STRUCTURED STATEMENTS

Structured statements may be nested as desired to create multilevel control structures. An example of such nesting is the following:

```
IF <EQ> THEN
    REPEAT
        MOVE    D0,(A5)+
        ADDQ     #4,D0
        MOVE.L   A4,(A4)+
    UNTIL.L A5 <LE> A4

    ELSE.L

        FOR D2 = #10 TO #20 BY #2 DO
            WHILE D4 <LE> D2 AND D4 <LT> #100 DO
                MOVE.L 10 (A3,D4.W),(A5)+
                ADDQ    #2,D4
            ENDW
        ENDF
    ENDI
```



The indentation shown above is not necessary for nested structure statements; it just makes the code easier to read.

7.5 EFFECTS ON THE USER'S ENVIRONMENT

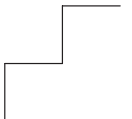
If the `-p` option is passed on the command line, the generated code of the structured control expansions is listed. There may be three items found in this code that will affect the user's environment:

- During assembly, local labels beginning with `|` (pipe bar) are generated. These labels use the same increment counter (.nnn) as local labels in macros. They are stored in the symbol table, but can not be duplicated in user-defined labels.
- In the FOR loop, `<op1>` is a user-defined symbol. When exiting the loop, the memory/register assigned to this symbol contains the value which caused the exit from the loop.
- Compare instructions (Table 7-1) are generated by the assembler whenever two operands are tested relationally in a structured statement. At run-time, however, **these assembler-generated instructions set the condition codes of the CCR** (in the case of a loop, the condition codes are set repeatedly). Users must keep in mind the effects of this when writing code that references the CCR within or following a structured statement.

CHAPTER

8

POSITION- INDEPENDENT CODE



8

CHAPTER

This chapter contains sections on Forcing Position Independence, Base-Displacement Addressing, and Base-Displacement in Conjunction with Forced Position Independence.

8.1 FORCING POSITION INDEPENDENCE

When creating a relocatable program module, it is often desirable to ensure that all references to operands in relocatable sections are position-independent effective addresses, i.e., no absolute addresses occur as effective addresses for such references. To avoid absolute effective address formats, it is necessary to ensure that all memory operand references are resolved by the assembler or the linker into one of the program counter relative or address register indirect addressing modes. The ORG directive should also be avoided.

To override an absolute address mode when resolving the effective address format of an operand, the following formats may be used to force program counter relative addressing:

- Forcing program counter with displacement:

An operand of the form:

`LABEL(PC)`

is resolved as a PC with displacement effective address, either by the assembler or by the linker. If LABEL cannot be resolved into a 16-bit displacement from the program counter, an error is generated.

- Forcing PC with index plus displacement:

An operand of the form:

`LABEL(PC,Rn)`

is resolved as a PC with index plus displacement effective address by the assembler. If LABEL cannot be resolved into an 8-bit displacement from the program counter, an error is generated.

8.2 BASE-DISPLACEMENT ADDRESSING

Although PC relative addresses have the advantage of position independence, such address formats often are not the most meaningful to the programmer when debugging an assembled module. There are many times when a programmer would prefer to see an address relative to a specified base — i.e., in a base-displacement format. This is especially true when addressing tables, arrays, and other data structures. Base-displacement references to a given location are “base relative” and, therefore, fixed with respect to a given base address; PC relative references to that same location are different in each instruction.

Base-displacement addressing must be handled explicitly by the programmer. For example, if the following data area is declared:

| | | |
|--------|------|------|
| TEMP | DS | \$40 |
| CONST | DC | \$10 |
| ARRAY1 | DS.L | \$10 |
| ARRAY2 | DS.L | \$10 |
| RESULT | DS.L | \$10 |

the programmer may choose to load A6 with the address of TEMP and make references to the other data locations as displacements from this base address. For example, to move the first element of ARRAY1 to D1, the programmer may specify:

```
MOVE.L      ARRAY1-TEMP(A6),D1
```

Indexing with the low order contents of D0 may be added (as the array index):

```
MOVE.L      ARRAY1-TEMP(A6,D0),D1
```

8.3 BASE-DISPLACEMENT IN CONJUNCTION WITH FORCED POSITION INDEPENDENCE

Complete code-position independence can be achieved by using base-displacement addressing in conjunction with the PCS option and the forced PC relative addressing scheme outlined in the *Forcing Position Independence* section. Although these techniques can be used to avoid all undesired absolute address formats, there are significant limitations of PC relative addressing in a position-independent program, as noted below:

- *PC with displacement:*
PC with displacement effective addresses (for the 68000 and 68010) are restricted by the 16-bit displacement field. A displacement greater than 32K byte from the current PC cannot be resolved in this format.
- *PC with index plus displacement:*
The displacement field here is restricted to eight bits (for the 68000 and 68010), limiting the range of this format to a 128-byte displacement from the current PC. The displacement may be relocatable.
- *Operands in the alterable addressing category:*
Neither PC relative mode is allowed as an alterable operand. This is a significant limitation in instructions which require an alterable operand, such as the destination operand in a MOVE instruction.

By appropriate use of base registers, these limitations can be overcome.



POSITION-INDEPENDENT CODE

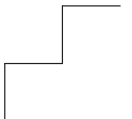
APPENDIX

CHARACTER SET

A



TASKING



A

APPENDIX

This appendix lists the ASCII characters recognized by the assembler.

1 CHARACTERS RECOGNIZED

The characters recognized by the assembler are listed below. The ASCII codes for these characters are shown on the following pages:

- The upper case letters A through Z
- The lower case letters a through z
- The digits 0 through 9
- Five arithmetic operators: +, -, *, /, and %
- The logical operators: >>, <<, &, and !
- Parentheses used in expressions ()
- Characters used as special prefixes:
 - # (pound sign) specifies the immediate mode of addressing
 - \$ (dollar sign) specifies a hexadecimal number
 - @ (commercial “at”) specifies an octal number
 - % (percent) specifies a binary number
 - ' (apostrophe) specifies an ASCII literal character
- The special characters used in macros: <, >, /, and @
- Four separating characters:
 - (space)
 - (tab)
 - , (comma)
 - . (period)
- A comment in a source statement may include any characters with ASCII values from (hexadecimal) 20 through 7E.
- Character used as a special suffix:
 - : (colon) specifies the end of a label

2 ASCII CHARACTER SET

| Character | Comments | Hex Value |
|-----------|-------------------|-----------|
| NUL | Null or tape feed | 00 |
| SOH | Start of Heading | 01 |
| STX | Start of Text | 02 |
| ETX | End of Text | 03 |

| Character | Comments | Hex Value |
|-----------|----------------------------|-----------|
| EOT | End of Transmission | 04 |
| ENQ | Enquire | 05 |
| ACK | Acknowledge | 06 |
| BEL | Bell | 07 |
| BS | Backspace | 08 |
| HT | Horizontal Tab | 09 |
| LF | Line Feed | 0A |
| VT | Vertical Tab | 0B |
| FF | Form Feed | 0C |
| RETURN | Carriage Return | 0D |
| SO | Shift Out (to red ribbon) | 0E |
| SI | Shift In (to black ribbon) | 0F |
| DLE | Data Link Escape | 10 |
| DC1 | Device Control 1 | 11 |
| DC2 | Device Control 2 | 12 |
| DC3 | Device Control 3 | 13 |
| DC4 | Device Control 4 | 14 |
| NAK | Negative Acknowledge | 15 |
| SYN | Synchronous idle | 16 |
| ETB | End of Transmission Block | 17 |
| CAN | Cancel | 18 |
| EM | End of Medium | 19 |
| SUB | Substitute | 1A |
| ESC | Escape, prefix | 1B |
| FS | File Separator | 1C |
| GS | Group Separator | 1D |
| RS | Record Separator | 1E |
| US | Unit Separator | 1F |
| SP | Space or blank | 20 |
| ! | Exclamation point | 21 |
| " | Quotation mark | 22 |

| Character | Comments | Hex Value |
|-----------|------------------------|-----------|
| # | Number sign | 23 |
| \$ | Dollar sign | 24 |
| % | Percent sign | 25 |
| & | Ampersand | 26 |
| ' | Apostrophe | 27 |
| (| Opening parenthesis | 28 |
|) | Closing parenthesis | 29 |
| * | Asterisk | 2A |
| + | Plus sign | 2B |
| — | Hyphen (minus) | 2D |
| . | Period (decimal point) | 2E |
| / | Slant | 2F |
| 0 | Digit 0 | 30 |
| 1 | Digit 1 | 31 |
| 2 | Digit 2 | 32 |
| 3 | Digit 3 | 33 |
| 4 | Digit 4 | 34 |
| 5 | Digit 5 | 35 |
| 6 | Digit 6 | 36 |
| 7 | Digit 7 | 37 |
| 8 | Digit 8 | 38 |
| 9 | Digit 9 | 39 |
| : | Colon | 3A |
| ; | Semicolon | 3B |
| < | Less than | 3C |
| = | Equals | 3D |
| > | Greater than | 3E |
| ? | Question mark | 3F |
| @ | Commercial at | 40 |
| A | Upper case letter A | 41 |
| B | Upper case letter B | 42 |

| Character | Comments | Hex Value |
|-----------|---------------------|-----------|
| C | Upper case letter C | 43 |
| D | Upper case letter D | 44 |
| E | Upper case letter E | 45 |
| F | Upper case letter F | 46 |
| G | Upper case letter G | 47 |
| H | Upper case letter H | 48 |
| I | Upper case letter I | 49 |
| J | Upper case letter J | 4A |
| K | Upper case letter K | 4B |
| L | Upper case letter L | 4C |
| M | Upper case letter M | 4D |
| N | Upper case letter N | 4E |
| O | Upper case letter O | 4F |
| P | Upper case letter P | 50 |
| Q | Upper case letter Q | 51 |
| R | Upper case letter R | 52 |
| S | Upper case letter S | 53 |
| T | Upper case letter T | 54 |
| U | Upper case letter U | 55 |
| V | Upper case letter V | 56 |
| W | Upper case letter W | 57 |
| X | Upper case letter X | 58 |
| Y | Upper case letter Y | 59 |
| Z | Upper case letter Z | 5A |
| [| Opening bracket | 5B |
| \ | Reverse slant | 5C |
|] | Closing bracket | 5D |
| ^ | Circumflex | 5E |
| _ | Underline | 5F |
| ' | Quotation mark | 60 |
| a | Lower case letter a | 61 |

| Character | Comments | Hex Value |
|-----------|---------------------|-----------|
| b | Lower case letter b | 62 |
| c | Lower case letter c | 63 |
| d | Lower case letter d | 64 |
| e | Lower case letter e | 65 |
| f | Lower case letter f | 66 |
| g | Lower case letter g | 67 |
| h | Lower case letter h | 68 |
| i | Lower case letter i | 69 |
| j | Lower case letter j | 6A |
| k | Lower case letter k | 6B |
| l | Lower case letter l | 6C |
| m | Lower case letter m | 6D |
| n | Lower case letter n | 6E |
| o | Lower case letter o | 6F |
| p | Lower case letter p | 70 |
| q | Lower case letter q | 71 |
| r | Lower case letter r | 72 |
| s | Lower case letter s | 73 |
| t | Lower case letter t | 74 |
| u | Lower case letter u | 75 |
| v | Lower case letter v | 76 |
| w | Lower case letter w | 77 |
| x | Lower case letter x | 78 |
| y | Lower case letter y | 79 |
| z | Lower case letter z | 7A |
| { | Opening brace | 7B |
| | Vertical line | 7C |
| } | Closing brace | 7D |
| ~ | Equivalent | 7E |
| DEL | Delete | 7F |

Table A-1: ASCII character set



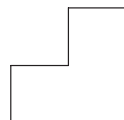
CHARACTER SET

INDEX

INDEX



TASKING



INDEX

Symbols

_tolower, 2-26
_toupper, 2-26-2-86

Numbers

68302, 2-9
68340, 2-10
68360, 2-11
68881, floating-point, 2-6, 2-12

A

A5 register, 2-5
A7 register, 2-5
abort, 2-27-2-86
abs, 2-27-2-86
access, 2-27
acos, 2-27-2-86
address modes, 4-16
asctime, 2-28-2-86
asin, 2-28-2-86
assert, 2-28
atan, 2-29-2-86
atan2, 2-29-2-86
atanh, 2-29-2-86
atexit, 2-30-2-86
atof, 2-30-2-86
atoi, 2-31-2-86
atol, 2-31-2-86

B

bsearch, 2-32-2-86

C

calloc, 2-32-2-86
ceil, 2-32-2-86
chdir, 2-33
clearerr, 2-33-2-86
clock, 2-33-2-86
close, 2-33
controls
 external symbol, 5-24-5-26
 external symbol definition (XDEF),
 5-24
 external symbol reference (XREF),
 5-25
 relocatable identification record
 (IDNT), 5-24
 internal assembly, 5-25-5-26
 declare external symbol
 (_BRINGIN), 5-26
 define data group (_DGROUP),
 5-26
 put out debugging information
 (_DEBSYM), 5-26
cos, 2-34-2-86
cosh, 2-34-2-86
ctime, 2-34

D

data initialization, 2-63
difftime, 2-34-2-86
directives
 assembly control, 5-3-5-26
 absolute origin (ORG), 5-7
 define offsets (OFFSET), 5-6
 enter named common section
 (COMMON), 5-4

include secondary file (INCLUDE), 5-6
program end (END), 5-5
relocatable ORG (RORG), 5-9
relocatable program section (SECTION), 5-10
reserve storage (RESERVE), 5-8
resume defined section (RESUME), 5-9
data definition/storage allocation, 5-14-5-26
 define constant (DC), 5-14
 define constant block (DCB), 5-17
 define storage (DS), 5-17
 specify command line (COMLINE), 5-14
symbol definition, 5-11-5-26
 define register list (REG), 5-13
 equate floating-point symbol value (FEQU), 5-12
 equate symbol value (EQU), 5-12
 set symbol value (SET), 5-13
terminate macro source statement generation (MEXIT), 6-6
 div, 2-35-2-86
 documentation, 1-3-1-4

 fclose, 2-36-2-86
 fcntl.h, open, 2-55
 feof, 2-36-2-86
 ferror, 2-36-2-86
 fflush, 2-36-2-86
 fgetc, 2-37
 fgetpos, 2-37-2-86
 fgets, 2-37-2-86
 file control block, 2-6
 floating-point
 constant notation, 4-12
 hex constant notation, 4-12
 floor, 2-37
 fmod, 2-38-2-86
 fopen, 2-38-2-86
 fprintf, 2-38-2-86
 fputc, 2-38
 fputs, 2-39-2-86
 fread, 2-39-2-86
 free, 2-39-2-86
 freopen, 2-39-2-86
 frexp, 2-40-2-86
 fscanf, 2-40-2-86
 fseek, 2-40-2-86
 fsetpos, 2-41-2-86
 ftell, 2-41-2-86
 fwrite, 2-42-2-86

E

exit, 2-35-2-86
 exp, 2-35-2-86
 expressions
 absolute, 4-15
 complex relocatable, 4-15
 simple relocatable, 4-15

F

fabs, 2-35-2-86

G

getc, 2-42-2-86
 getchar, 2-42
 getcwd, 2-42
 getenv, 2-43
 getl, 2-43
 gets, 2-43-2-86
 getw, 2-43-2-86
 global
 data, 2-5
 variable, 2-8

gmtime, 2-44-2-86

I

I/O, system, 2-6-2-25
 idata, 2-5
 initialization, routine, 2-11
 isalnum, 2-44-2-86
 isalpha, 2-44-2-86
 iscntrl, 2-45-2-86
 isdigit, 2-45-2-86
 isgraph, 2-45-2-86
 islower, 2-46-2-86
 isprint, 2-46-2-86
 ispunct, 2-46-2-86
 isspace, 2-47-2-86
 isupper, 2-47-2-86
 isxdigit, 2-47-2-86

L

labs, 2-47-2-86
 ldata, 2-6
 ldexp, 2-48-2-86
 ldiv, 2-48-2-86
 libraries that do not use A5, 2-6
 library, modification, 2-3
 localeconv, 2-48-2-86
 localtime, 2-49-2-86
 log, 2-49-2-86
 log10, 2-49-2-86
 log2, 2-49-2-86
 longjmp, 2-50-2-86
 lseek, 2-50

M

M68302ADS, 2-3, 2-4, 2-9
 M68340BCC, 2-3, 2-4, 2-10

M68360QUADS, 2-11
 macro calls, 4-5
 malloc, 2-50-2-86
 memccpy, 2-52-2-86
 memchr, 2-52-2-86
 memcmp, 2-53-2-86
 memcpy, 2-53-2-86
 memmove, 2-53-2-86
 memset, 2-53-2-86
 mktime, 2-54-2-86
 mnemonics
 directive, 4-5
 instruction, 4-5
 modf, 2-54-2-86

N

no-floats library, 2-13

O

offsetof, 2-54
 open, 2-55
 operators
 arithmetic operators, 4-13
 logical operators, 4-13
 shift operators, 4-13
 options, listing control and output,
 5-18-5-26
 assembler option (OPT), 5-20
 control listing file format
 (*FORMAT/NOFORMAT*), 5-19
 control listing generation
 (*LIST/NOLIST*), 5-19
 control pagination (PAGE/NOPAGE),
 5-23
 produce no object module (NOOBJ),
 5-19-5-20
 programmer generated error (FAIL),
 5-19

set subtitle (STTL), 5-23
set title (TTL), 5-23
space between source lines (SPC),
 5-23
specify listing line length (LLEN),
 5-19-5-20

P

perror, 2-55-2-86
 pow, 2-55-2-86
 printf, 2-56-2-86
 putc, 2-60-2-86
 putchar, 2-61-2-86
 putl, 2-61-2-86
 puts, 2-61-2-86
 putw, 2-61-2-86

Q

qsort, 2-62-2-86

R

raise, 2-62
 rand, 2-62-2-86
 rcopy, 2-63-2-86
 read, 2-64
 realloc, 2-63-2-86
 remove, 2-64-2-86
 rename, 2-64-2-86
 ROM, 2-79
 run-time library, 2-1-2-25
 index file, 2-13
 modification, 2-3, 2-11-2-25
 object modules, 2-13-2-25
 routines, 2-14-2-25
 source code, 2-3

run-time library routine

_tolower, 2-26
 _toupper, 2-26
 abort, 2-27
 abs, 2-27
 access, 2-27
 acos, 2-27-2-86
 asctime, 2-28
 asin, 2-28
 assert, 2-28
 atan, 2-29
 atan2, 2-29
 atanh, 2-29
 atexit, 2-30
 atof, 2-30
 atoi, 2-31
 atol, 2-31
 bsearch, 2-32
 calloc, 2-32
 ceil, 2-32
 chdir, 2-33
 clearerr, 2-33
 clock, 2-33
 close, 2-33
 cos, 2-34
 cosh, 2-34
 ctime, 2-34
 difftime, 2-34
 div, 2-35
 exit, 2-35
 exp, 2-35
 fabs, 2-35
 fclose, 2-36
 feof, 2-36
 ferror, 2-36
 fflush, 2-36
 fgetc, 2-37, 2-42
 fgetpos, 2-37
 fgets, 2-37
 floor, 2-37
 fmod, 2-38
 fopen, 2-38

fprintf, 2-38
fputc, 2-38
fputs, 2-39
fread, 2-39
free, 2-39
freopen, 2-39
frexp, 2-40
fscanf, 2-40
fseek, 2-40
fsetpos, 2-41
ftell, 2-41
fwrite, 2-42
getchar, 2-42
getcwd, 2-42
getenv, 2-43
getl, 2-43
gets, 2-43
getw, 2-43
gmtime, 2-44
isalnum, 2-44
isalpha, 2-44
iscntrl, 2-45
isdigit, 2-45
isgraph, 2-45
islower, 2-46
isprint, 2-46
ispunct, 2-46
isspace, 2-47
isupper, 2-47
isxdigit, 2-47
labs, 2-47
ldexp, 2-48
ldiv, 2-48
localeconv, 2-48
localtime, 2-49
log, 2-49
log10, 2-49
log2, 2-49
longjmp, 2-50
lseek, 2-50
malloc, 2-50
mblen, 2-51
mbstowc, 2-52
mbstowcs, 2-51
memccpy, 2-52
memchr, 2-52
memcmp, 2-53
memcpy, 2-53
memmove, 2-53
memset, 2-53
mktime, 2-54
modf, 2-54
offsetof, 2-54
open, 2-55
perror, 2-55
pow, 2-55
printf, 2-56
putc, 2-60
putchar, 2-61
putl, 2-61
puts, 2-61
putw, 2-61
qsort, 2-62
raise, 2-62
rand, 2-62
rcopy, 2-63
read, 2-64
realloc, 2-63
remove, 2-64
rename, 2-64
rewind, 2-64
roupper, 2-83
scanf, 2-65
setbuf, 2-69
setjmp, 2-69
setlocale, 2-70
setvbuf, 2-70
signal, 2-71
sin, 2-71
sinh, 2-71
sprintf, 2-71
sqrt, 2-72
srand, 2-72
sscanf, 2-72
stat, 2-73
strcat, 2-73

strchr, 2-73
strcmp, 2-73
strcoll, 2-74
strcpy, 2-74
strcspn, 2-74
strerror, 2-74
strftime, 2-75
strlen, 2-76
strncat, 2-76
strncmp, 2-77
strncpy, 2-77
strpbrk, 2-77
strrchr, 2-78
strspn, 2-78
strstr, 2-78
strtod, 2-78
strtok, 2-79
strtol, 2-79
strtoul, 2-80
strxfrm, 2-80
swab, 2-81
system, 2-81
tan, 2-81
tanh, 2-81
time, 2-82
tmpfile, 2-82
tmpnam, 2-82
tolower, 2-82
ungetc, 2-83
unlink, 2-83
va_arg, 2-83
va_end, 2-84
va_start, 2-84
vfprintf, 2-84
vprintf, 2-85
vsprintf, 2-85
wcstombs, 2-85
wctomb, 2-86
write, 2-86

S

scanf, 2-65-2-86
 SEGBASE, 4-14
 SEGSIZE, 4-14
setbuf, 2-69-2-86
setjmp, 2-69-2-86
setlocale, 2-70-2-86
setvbuf, 2-70-2-86
sin, 2-71-2-86
sinh, 2-71-2-86
sprintf, 2-71-2-86
sqrt, 2-72-2-86
srand, 2-72-2-86
sscanf, 2-72-2-86
stat, 2-73
 statements
 assembler control, 3-4-3-5
 assembler directive, 3-4-3-5, 4-3
 data allocation, 3-4-3-5
 FOR, 7-7-7-14
 IF, 7-5-7-14
 instruction, 3-4-3-5
 REPEAT, 7-8-7-14
 source, 4-3, 4-4
 WHILE, 7-8-7-14
 storage allocation, 2-9-2-25
strcat, 2-73-2-86
strchr, 2-73-2-86
strcmp, 2-73-2-86
strcoll, 2-74-2-86
strcpy, 2-74-2-86
strcspn, 2-74-2-86
strerror, 2-74-2-86
strftime, 2-75-2-86
strlen, 2-76-2-86
strncat, 2-76-2-86
strncmp, 2-77

strncpy, 2-77-2-86
strpbrk, 2-77-2-86
strchr, 2-78-2-86
strspn, 2-78-2-86
strstr, 2-78-2-86
strtod, 2-78
strtok, 2-79-2-86
strtol, 2-79-2-86
strtoul, 2-80-2-86
strxfrm, 2-80-2-86
swab, 2-81-2-86
symbols, NARG, 6-7
system, 2-81-2-86
 initialization, 2-4-2-25

T

tan, 2-81-2-86
tanh, 2-81-2-86
time, 2-82-2-86
tmpfile, 2-82-2-86
tmpname, 2-82-2-86
tolower, 2-82-2-86
toupper, 2-83-2-86

U

udata, 2-5

ungetc, 2-83-2-86
unistd.h
 access, 2-27
 chdir, 2-33
 close, 2-33
 getcwd, 2-42
 lseek, 2-50
 read, 2-64
 stat, 2-73
 unlink, 2-83
 write, 2-86
unlink, 2-83
updating library, 2-12

V

va_arg, 2-83
va_end, 2-84
va_start, 2-84
vfprintf, 2-84-2-86
vprintf, 2-85-2-86
vsprintf, 2-85-2-86

W

write, 2-86

