16-BIT LANGUAGE TOOLS LIBRARIES

Note the following details of the code protection feature on Microchip devices:

- Microchip products meet the specification contained in their particular Microchip Data Sheet.
- Microchip believes that its family of products is one of the most secure families of its kind on the market today, when used in the intended manner and under normal conditions.
- There are dishonest and possibly illegal methods used to breach the code protection feature. All of these methods, to our knowledge, require using the Microchip products in a manner outside the operating specifications contained in Microchip's Data Sheets. Most likely, the person doing so is engaged in theft of intellectual property.
- Microchip is willing to work with the customer who is concerned about the integrity of their code.
- Neither Microchip nor any other semiconductor manufacturer can guarantee the security of their code. Code protection does not mean that we are guaranteeing the product as "unbreakable."

Code protection is constantly evolving. We at Microchip are committed to continuously improving the code protection features of our products. Attempts to break Microchip's code protection feature may be a violation of the Digital Millennium Copyright Act. If such acts allow unauthorized access to your software or other copyrighted work, you may have a right to sue for relief under that Act.

Trademarks

QUALITY MANAGEMENT SYSTEM
CERTIFIED BY DNV
ISO/TS 16949:2002

Microchip received ISO/TS-16949:2002 certification for its worldwide headquarters, design and wafer fabrication facilities in Chandler and Tempe, Arizona, Gresham, Oregon and Mountain View, California. The Company's quality system processes and procedures are for its PIC® MCUs and dsPIC® DSCs, KEELOQ® code hopping devices, Serial EEPROMs, microperipherals, nonvolatile memory and analog products. In addition, Microchip's quality system for the design and manufacture of development systems is ISO 9001:2000 certified.

Table of Contents

Preface

NOTICE TO CUSTOMERS

All documentation becomes dated, and this manual is no exception. Microchip tools and documentation are constantly evolving to meet customer needs, so some actual dialogs and/or tool descriptions may differ from those in this document. Please refer to our web site (www.microchip.com) to obtain the latest documentation available.

Documents are identified with a "DS" number. This number is located on the bottom of each page, in front of the page number. The numbering convention for the DS number is "DSXXXXXA", where "XXXXX" is the document number and "A" is the revision level of the document.

For the most up-to-date information on development tools, see the MPLAB® IDE on-line help. Select the Help menu, and then Topics to open a list of available on-line help files.

INTRODUCTION

This chapter contains general information that will be useful to know before using 16-bit libraries. Items discussed include:

- Document Layout
- Conventions Used in this Guide
- Recommended Reading
- The Microchip Web Site
- Development Systems Customer Change Notification Service
- Customer Support

DOCUMENT LAYOUT

This document describes how to use GNU language tools to write code for 16-bit applications. The document layout is as follows:

- **Chapter 1: Library Overview** – gives an overview of libraries. Some are described further in this document, while others are described in other documents or on-line Help files.
- **Chapter 2: DSP Library** – lists the library functions for DSP operation.
- **Chapter 3: Standard C Library with Math Functions** – lists the library functions and macros for standard C operation.
- **Appendix A: ASCII Character Set**

CONVENTIONS USED IN THIS GUIDE

The following conventions may appear in this documentation:

DOCUMENTATION CONVENTIONS

Description	Represents	Examples
Arial font:		
Italic	Referenced books	*MPLAB® IDE User's Guide*
	Emphasized text	...is the *only* compiler...
Initial caps	A window	the Output window
	A dialog	the Settings dialog
	A menu selection	select Enable Programmer
Quotes	A field name in a window or dialog	"Save project before build"
Underlined, italic with right angle bracket	A menu path	*File>Save*
Bold	A dialog button	Click **OK**
	A tab	Click the **Power** tab
Text in angle brackets < >	A key on the keyboard	Press <Enter>, <F1>
Courier New font:		
Plain	Sample source code	`#define START`
	Filenames	`autoexec.bat`
	File paths	`c:\mcc18\h`
	Keywords	`_asm, _endasm, static`
	Command-line options	`-Opa+, -Opa-`
	Bit values	`0, 1`
	Constants	`0xFF, 'A'`
Italic	A variable argument	*`file`*`.o`, where *`file`* can be any valid filename
Square brackets [ ]	Optional arguments	`mpasmwin [options]` *`file`* `[options]`
Curly brackets and pipe character: { | }	Choice of mutually exclusive arguments; an OR selection	`errorlevel {0|1}`
Ellipses...	Replaces repeated text	`var_name [, var_name...]`
	Represents code supplied by user	`void main (void) { ... }`

RECOMMENDED READING

This documentation describes how to use 16-bit libraries. Other useful documents are listed below. The following Microchip documents are available and recommended as supplemental reference resources.

Readme Files

For the latest information on Microchip tools, read the associated Readme files (HTML files) included with the software.

16-Bit Language Tools Getting Started (DS70094)

A guide to installing and working with the Microchip language tools (MPLAB ASM30, MPLAB LINK30 and MPLAB C30) for 16-bit devices. Examples using the 16-bit simulator SIM30 (a component of MPLAB SIM) are provided.

MPLAB® ASM30, MPLAB® LINK30 and Utilities User's Guide (DS51317)

A guide to using the 16-bit assembler, MPLAB ASM30, object linker, MPLAB LINK30 and various utilities, including MPLAB LIB30 archiver/librarian.

MPLAB® C30 C Compiler User's Guide (DS51284)

A guide to using the 16-bit C compiler. MPLAB LINK30 is used with this tool.

Device-Specific Documentation

The Microchip website contains many documents that describe 16-bit device functions and features. Among these are:

- Individual and family data sheets
- Family reference manuals
- Programmer's reference manuals

C Standards Information

American National Standard for Information Systems – *Programming Language – C*. American National Standards Institute (ANSI), 11 West 42nd. Street, New York, New York, 10036.

This standard specifies the form and establishes the interpretation of programs expressed in the programming language C. Its purpose is to promote portability, reliability, maintainability and efficient execution of C language programs on a variety of computing systems.

C Reference Manuals

Harbison, Samuel P. and Steele, Guy L., *C A Reference Manual*, Fourth Edition, Prentice-Hall, Englewood Cliffs, N.J. 07632.

Kernighan, Brian W. and Ritchie, Dennis M., *The C Programming Language*, Second Edition. Prentice Hall, Englewood Cliffs, N.J. 07632.

Kochan, Steven G., *Programming In ANSI C*, Revised Edition. Hayden Books, Indianapolis, Indiana 46268.

Plauger, P.J., *The Standard C Library*, Prentice-Hall, Englewood Cliffs, N.J. 07632.

Van Sickle, Ted., *Programming Microcontrollers in C*, First Edition. LLH Technology Publishing, Eagle Rock, Virginia 24085.

THE MICROCHIP WEB SITE

Microchip provides online support via our web site at www.microchip.com. This web site is used as a means to make files and information easily available to customers. Accessible by using your favorite Internet browser, the web site contains the following information:

- **Product Support** – Data sheets and errata, application notes and sample programs, design resources, user's guides and hardware support documents, latest software releases and archived software
- **General Technical Support** – Frequently Asked Questions (FAQs), technical support requests, online discussion groups, Microchip consultant program member listing
- **Business of Microchip** – Product selector and ordering guides, latest Microchip press releases, listing of seminars and events, listings of Microchip sales offices, distributors and factory representatives

DEVELOPMENT SYSTEMS CUSTOMER CHANGE NOTIFICATION SERVICE

Microchip's customer notification service helps keep customers current on Microchip products. Subscribers will receive e-mail notification whenever there are changes, updates, revisions or errata related to a specified product family or development tool of interest.

To register, access the Microchip web site at www.microchip.com, click on Customer Change Notification and follow the registration instructions.

The Development Systems product group categories are:

- **Compilers** – The latest information on Microchip C compilers and other language tools. These include the MPLAB C18 and MPLAB C30 C compilers; MPASM™ and MPLAB ASM30 assemblers; MPLINK™ and MPLAB LINK30 object linkers; and MPLIB™ and MPLAB LIB30 object librarians.
- **Emulators** – The latest information on Microchip in-circuit emulators.This includes the MPLAB REAL ICE™, MPLAB ICE 2000 and MPLAB ICE 4000 in-circuit emulators.
- **In-Circuit Debuggers** – The latest information on the Microchip in-circuit debugger, MPLAB ICD 2.
- **MPLAB® IDE** – The latest information on Microchip MPLAB IDE, the Windows® Integrated Development Environment for development systems tools. This list is focused on the MPLAB IDE, MPLAB IDE Project Manager, MPLAB Editor and MPLAB SIM simulator, as well as general editing and debugging features.
- **Programmers** – The latest information on Microchip programmers. These include the MPLAB PM3 and PRO MATE® II device programmers and the PICSTART® Plus, PICkit™ 1 and PICkit 2 development programmers.

CUSTOMER SUPPORT

Users of Microchip products can receive assistance through several channels:

- Distributor or Representative
- Local Sales Office
- Field Application Engineer (FAE)
- Technical Support

Customers should contact their distributor, representative or field application engineer (FAE) for support. Local sales offices are also available to help customers. A listing of sales offices and locations is included in the back of this document.

Technical support is available through the web site at: http://support.microchip.com

NOTES:

Chapter 1. Library Overview

1.1 INTRODUCTION

A library is a collection of functions grouped for reference and ease of linking. See the "*MPLAB® ASM30, MPLAB® LINK30 and Utilities User's Guide"* (DS51317) for more information about making and using libraries.

1.1.1 Assembly Code Applications

Free versions of the 16-bit language tool libraries are available from the Microchip web site. DSP and 16-bit peripheral libraries are provided with object files and source code. A math library containing functions from the standard C header file <`math.h`> is provided as an object file only. The complete standard C library is provided with the MPLAB C30 C compiler.

1.1.2 C Code Applications

The 16-bit language tool libraries are included in the `lib` subdirectory of the MPLAB C30 C compiler install directory, which is by default:

```
C:\Program Files\Microchip\MPLAB C30\lib
```

These libraries can be linked directly into an application with MPLAB LINK30.

1.1.3 Chapter Organization

This chapter is organized as follows:

- OMF-Specific Libraries/Start-up Modules
- Start-up Code
- DSP Library
- 16-Bit Peripheral Libraries
- Standard C Libraries (with Math Functions)
- MPLAB C30 Built-in Functions

1.2 OMF-SPECIFIC LIBRARIES/START-UP MODULES

Library files and start-up modules are specific to OMF (Object Module Format). An OMF can be one of the following:

- COFF – This is the default.
- ELF – The debugging format used for ELF object files is DWARF 2.0.

There are two ways to select the OMF:

1. Set an environment variable called `PIC30_OMF` for all tools.
2. Select the OMF on the command line when invoking the tool, i.e., `-omf=`*`omf`* or `-momf=`*`omf`* .

16-bit tools will first look for generic library files when building your application (no OMF specification). If these cannot be found, the tools will look at your OMF specifications and determine which library file to use.

As an example, if `libdsp.a` is not found and no environment variable or command-line option is set, the file `libdsp-coff.a` will be used by default.

1.3 START-UP CODE

In order to initialize variables in data memory, the linker creates a data initialization template. This template must be processed at start-up, before the application proper takes control. For C programs, this function is performed by the start-up modules in `libpic30-coff.a` (either `crt0.o` or `crt1.o`) or `libpic30-elf.a` (either `crt0.eo` or `crt1.eo`). Assembly language programs can utilize these modules directly by linking with the desired start-up module file. The source code for the start-up modules is provided in corresponding `.s` files.

The primary start-up module (`crt0`) initializes all variables (variables without initializers are set to zero as required by the ANSI standard) except for variables in the persistent data section. The alternate start-up module (`crt1`) performs no data initialization.

For more on start-up code, see the "*MPLAB® ASM30, MPLAB® LINK30 and Utilities User's Guide"* (DS51317) and, for C applications, the "*MPLAB® C30 C Compiler User's Guide"* (DS51284).

1.4 DSP LIBRARY

The DSP library (`libdsp-`*`omf`*`.a`) provides a set of digital signal processing operations to a program targeted for execution on a dsPIC30F digital signal controller (DSC). In total, 49 functions are supported by the DSP Library.

1.5 16-BIT PERIPHERAL LIBRARIES

The 16-bit software and hardware peripheral libraries provide functions and macros for setting up and controlling 16-bit peripherals. These libraries are processor-specific and of the form `libp`*`Device`*`-`*`omf`*`.a`, where *`Device`* is the 16-bit device number (e.g., `libp30F6014-coff.a` for the dsPIC30F6014 device) and *`omf`* is either `coff` or `elf`.

Documentation for these libraries is provided in HTML Help files. Examples of use are also provided in each file. By default, the documentation is found in:

`C:\Program Files\Microchip\MPLAB C30\docs`

1.6 STANDARD C LIBRARIES (WITH MATH FUNCTIONS)

A complete set of ANSI-89 conforming libraries are provided. The standard C library files are `libc-`*`omf`*`.a` (written by Dinkumware, an industry leader) and `libm-`*`omf`*`.a` (math functions, written by Microchip).

Additionally, some 16-bit standard C library helper functions, and standard functions that must be modified for use with 16-bit devices, are in `libpic30-`*`omf`*`.a`.

A typical C application will require all three libraries.

1.7 MPLAB C30 BUILT-IN FUNCTIONS

The MPLAB C30 C compiler contains built-in functions that, to the developer, work like library functions. These functions are listed in the "*MPLAB® C30 C Compiler Users' Guide*" (DS51284).

Chapter 2. DSP Library

2.1 INTRODUCTION

The DSP Library provides a set of digital signal processing operations to a program targeted for execution on a dsPIC30F/33F digital signal controller. The library has been designed to provide you, the C software developer, with efficient implementation of the most common signal processing functions. In total, 52 functions are supported by the DSP Library.

A primary goal of the library is to minimize the execution time of each function. To achieve this goal, the DSP Library is predominantly written in optimized assembly language. By using the DSP Library, you can realize significant gains in execution speed over equivalent code written in ANSI C. Additionally, since the DSP Library has been rigorously tested, using the DSP Library will allow you to shorten your application development time.

2.1.1 Assembly Code Applications

A free version of this library and its associated header file is available from the Microchip web site. Source code is included.

2.1.2 C Code Applications

The MPLAB C30 C compiler install directory (`c:\program files\microchip\mplab c30`) contains the following subdirectories with library-related files:

- `lib` – DSP library/archive files
- `src\dsp` – source code for library functions and a batch file to rebuild the library
- `support\h` – header file for DSP library

2.1.3 Chapter Organization

This chapter is organized as follows:

- Using the DSP Library
- Vector Functions
- Window Functions
- Matrix Functions
- Filtering Functions
- Transform Functions
- Control Functions
- Miscellaneous Functions

2.2 USING THE DSP LIBRARY

2.2.1 Building with the DSP Library

Building an application which utilizes the DSP Library requires only two files: `dsp.h` and `libdsp-omf.a`. `dsp.h` is a header file which provides all the function prototypes, `#defines` and `typedefs` used by the library. `libdsp-omf.a` is the archived library file which contains all the individual object files for each library function. (See **Section 1.2 "OMF-Specific Libraries/Start-up Modules"** for more on OMF-specific libraries.)

When compiling an application, `dsp.h` must be referenced (using `#include`) by all source files which call a function in the DSP Library or use its symbols or `typedefs`. When linking an application, `libdsp-omf.a` must be provided as an input to the linker (using the `--library` or `-l` linker switch) such that the functions used by the application may be linked into the application.

The linker will place the functions of the DSP library into a special text section named `.libdsp`. This may be seen by looking at the map file generated by the linker.

2.2.2 Memory Models

The DSP Library is built with the "small code" and "small data" memory models to create the smallest library possible. Since a few DSP library functions are written in C and make use of the compiler's floating-point library, the MPLAB C30 linker script files place the `.libm` and `.libdsp` text sections next to each other. This ensures that the DSP library may safely use the RCALL instruction to call the required floating-point routines in the floating-point library.

2.2.3 DSP Library Function Calling Convention

All the object modules within the DSP Library are compliant with the C compatibility guidelines for the dsPIC30F/33F DSC and follow the function call conventions documented in the Microchip "*MPLAB® C30 C Compiler User's Guide"* (DS51284). Specifically, functions may use the first eight working registers (W0 through W7) as function arguments. Any additional function arguments are passed through the stack.

The working registers W0 to W7 are treated as scratch memory, and their values may not be preserved after the function call. On the other hand, if any of the working registers W8 to W13 are used by a function, the working register is first saved, the register is used and then its original value is restored upon function return. The return value of a (non void) function is available in working register W0 (also referred to as WREG). When needed, the run time software stack is used following the C system stack rules described in the *MPLAB® C30 Compiler User's Guide*. Based on these guidelines, the object modules of the DSP Library can be linked to either a C program, an assembly program or a program which combines code in both languages.

2.2.4 Data Types

The operations provided by the DSP Library have been designed to take advantage of the DSP instruction set and architectural features of the dsPIC30F/33F DSC. In this sense, most operations are computed using fractional arithmetic.

The DSP Library defines a fractional type from an integer type:

```
#ifndef fractional
typedef int fractional;
#endif
```

The `fractional` data type is used to represent data that has 1 sign bit, and 15 fractional bits. Data which uses this format is commonly referred to as "1.15" data.

For functions which use the multiplier, results are computed using the 40-bit accumulator, and “9.31” arithmetic is utilized. This data format has 9 sign/magnitude bits and 31 fractional bits, which provides for extra computational headroom above the range (-1.00 to ~+1.00) provided by the 1.15 format. Naturally when these functions provide a result, they revert to a `fractional` data type, with 1.15 format.

The use of fractional arithmetic imposes some constraints on the allowable set of values to be input to a particular function. If these constraints are ensured, the operations provided by the DSP Library typically produce numerical results correct to 14 bits. However, several functions perform implicit scaling to the input data and/or output results, which may decrease the resolution of the output values (when compared to a floating-point implementation).

A subset of operations in the DSP Library, which require a higher degree of numerical resolution, do operate in floating-point arithmetic. Nevertheless, the results of these operations are transformed into fractional values for integration with the application. The only exception to this is the `MatrixInvert` function which computes the inversion of a floating-point matrix in floating-point arithmetic, and provides the results in floating-point format.

2.2.5 Data Memory Usage

The DSP Library performs no allocation of RAM, and leaves this task to you. If you do not allocate the appropriate amount of memory and align the data properly, undesired results will occur when the function executes. In addition, to minimize execution time, the DSP Library will do no checking on the provided function arguments (including pointers to data memory), to determine if they are valid. The user may refer to example projects that utilize the DSP library functions, in order to ascertain proper usage of functions. MPLAB IDE-based example projects/workspaces have been provided in the installation folder of the MPLAB C30 toolsuite.

Most functions accept data pointers as function arguments, which contain the data to be operated on, and typically also the location to store the result. For convenience, most functions in the DSP Library expect their input arguments to be allocated in the default RAM memory space (X-Data or Y-Data), and the output to be stored back into the default RAM memory space. However, the more computational intensive functions require that some operands reside in X-Data and Y-Data (or program memory and Y-Data), so that the operation can take advantage of the dual data fetch capability of the 16-bit architecture.

2.2.6 CORCON Register Usage

Many functions of the DSP Library place the dsPIC30F/33F device into a special operating mode by modifying the CORCON register. On the entry of these functions, the CORCON register is pushed to the stack. It is then modified to correctly perform the desired operation, and lastly the CORCON register is popped from the stack to preserve its original value. This mechanism allows the library to execute as correctly as possible, without disrupting CORCON setting.

When the CORCON register is modified, it is typically set to 0x00F0. This places the dsPIC30F/33F device into the following operational mode:

- DSP multiplies are set to used signed and fractional data
- Accumulator saturation is enabled for Accumulator A and Accumulator B
- Saturation mode is set to 9.31 saturation (Super Saturation)
- Data Space Write Saturation is enabled
- Program Space Visibility disabled
- Convergent (unbiased) rounding is enabled

For a detailed explanation of the CORCON register and its effects, refer to the "*dsPIC30F Family Reference Manual*" (DS70046).

2.2.7 Overflow and Saturation Handling

The DSP Library performs most computations using 9.31 saturation, but must store the output of the function in 1.15 format. If during the course of operation the accumulator in use saturates (goes above 0x7F FFFF FFFF or below 0x80 0000 0000), the corresponding saturation bit (SA or SB) in the STATUS register will be set. This bit will stay set until it is cleared. This allows you to inspect SA or SB after the function executes and to determine if action should be taken to scale the input data to the function.

Similarly, if a computation performed with the accumulator results in an overflow (the accumulator goes above 0x00 7FFF FFFF or below 0xFF 8000 0000), the corresponding overflow bit (OA or OB) in the STATUS register will be set. Unlike the SA and SB status bits, OA and OB will not stay set until they are cleared. These bits are updated each time an operation using accumulator is executed. If exceeding this specified range marks an important event, you are advised to enable the Accumulator Overflow Trap via the OVATE, OVBTE and COVTE bits in the INTCON1 register. This will have the effect of generating an Arithmetic Error Trap as soon as the Overflow condition occurs, and you may then take the required action.

2.2.8 Integrating with Interrupts and an RTOS

The DSP Library may easily be integrated into an application which utilizes interrupts or an RTOS, yet certain guidelines must be followed. To minimize execution time, the DSP Library utilizes `DO` loops, `REPEAT` loops, Modulo addressing and Bit-Reversed addressing. Each of these components is a finite hardware resource on the 16-bit device, and the background code must consider the use of each resource when disrupting execution of a DSP Library function.

When integrating with the DSP Library, you must examine the Function Profile of each function description to determine which resources are used. If a library function will be interrupted, it is your responsibility to save and restore the contents of all registers used by the function, including the state of the `DO`, `REPEAT` and special addressing hardware. Naturally this also includes saving and restoring the contents of the CORCON and Status registers.

2.2.9 Rebuilding the DSP Library

A batch file named `makedsplib.bat` is provided to rebuild the DSP library. The MPLAB C30 compiler is required to rebuild the DSP library, and the batch file assumes that the compiler is installed in the default directory, `c:\Program Files\Microchip\MPLAB C30\`. If your language tools are installed in a different directory, you must modify the directories in the batch file to match the location of your language tools.

2.3 VECTOR FUNCTIONS

This section presents the concept of a fractional vector, as considered by the DSP Library, and describes the individual functions which perform vector operations.

2.3.1 Fractional Vector Operations

A fractional vector is a collection of numerical values, the vector elements, allocated contiguously in memory, with the first element at the lowest memory address. One word of memory (two bytes) is used to store the value of each element, and this quantity must be interpreted as a fractional number represented in the 1.15 data format.

A pointer addressing the first element of the vector is used as a handle which provides access to each of the vector values. The address of the first element is referred to as the base address of the vector. Because each element of the vector is 16 bits, the base address *must* be aligned to an even address.

The one dimensional arrangement of a vector accommodates to the memory storage model of the device, so that the nth element of an N-element vector can be accessed from the vector's base address BA as:

$BA + 2(n - 1)$, for $1 \leq n \leq N$.

The factor of 2 is used because of the byte addressing capabilities of the 16-bit device.

Unary and binary fractional vector operations are implemented in this library. The operand vector in a unary operation is called the source vector. In a binary operation the first operand is referred to as the source one vector, and the second as the source two vector. Each operation applies some computation to one or several elements of the source vector(s). Some operations produce a result which is a scalar value (also to be interpreted as a 1.15 fractional number), while other operations produce a result which is a vector. When the result is also a vector, this is referred to as the destination vector.

Some operations resulting in a vector allow computation in place. This means the results of the operation are placed back into the source vector (or the source one vector for binary operations). In this case, the destination vector is said to (physically) replace the source (one) vector. If an operation can be computed in place, it is indicated as such in the comments provided with the function description.

For some binary operations, the two operands can be the same (physical) source vector, which means the operation is applied to the source vector and itself. If this type of computation is possible for a given operation, it is indicated as such in the comments provided with the function description.

Some operations can be both self applicable and computed in place.

All the fractional vector operations in this library take as an argument the cardinality (number of elements) of the operand vector(s). Based on the value of this argument the following assumptions are made:

a) The sum of sizes of all the vectors involved in a particular operation falls within the range of available data memory for the target device.
b) In the case of binary operations, the cardinalities of both operand vectors *must* obey the rules of vector algebra (particularly, see remarks for the `VectorConvolve` and `VectorCorrelate` functions).
c) The destination vector *must* be large enough to accept the results of an operation.

2.3.2 User Considerations

a) No boundary checking is performed by these functions. Out of range cardinalities (including zero length vectors) as well as nonconforming use of source vector

sizes in binary operations may produce unexpected results.

b) The vector addition and subtraction operations could lead to saturation if the sum of corresponding elements in the source vector(s) is greater than $1-2^{-15}$ or smaller than -1.0. Analogously, the vector dot product and power operations could lead to saturation if the sum of products is greater than $1-2^{-15}$ or smaller than -1.0.

c) It is recommended that the STATUS Register (SR) be examined after completion of each function call. In particular, users can inspect the SA, SB and SAB flags after the function returns to determine if saturation occurred.

d) All the functions have been designed to operate on fractional vectors allocated in default RAM memory space (X-Data or Y-Data).

e) Operations which return a destination vector can be nested, so that for instance if:

   a = Op1 (b, c), with b = Op2 (d), and c = Op3 (e, f), then

   a = Op1 (Op2 (d), Op3 (e, f))

2.3.3 Additional Remarks

The description of the functions limits its scope to what could be considered the regular usage of these operations. However, since no boundary checking is performed during computation of these functions, you have the freedom to interpret the operation and its results as it fits some particular needs.

For instance, while computing the `VectorMax` function, the length of the source vector could be greater than *`numElems`*. In this case, the function would be used to find the maximum value *only* among the first *`numElems`* elements of the source vector.

As another example, you may be interested in replacing *`numElems`* elements of a destination vector located between `N` and `N+`*`numElems`*`-1`, with *`numElems`* elements from a source vector located between elements `M` and `M+`*`numElems`*`-1`. Then, the `VectorCopy` function could be used as follows:

```
fractional* dstV[DST_ELEMS] = {...};
fractional* srcV[SRC_ELEMS] = {...};
int n = NUM_ELEMS;
int N = N_PLACE;   /* NUM_ELEMS+N ≤ DST_ELEMS */
int M = M_PLACE;   /* NUM_ELEMS+M ≤ SRC_ELEMS */
fractional* dstVector = dstV+N;
fractional* srcVector = srcV+M;

dstVector = VectorCopy (n, dstVector, srcVector);
```

Also in this context, the `VectorZeroPad` function can operate in place, where now *`dstV = srcV`*, *`numElems`* is the number of elements at the beginning of source vector to preserve, and `numZeros` the number of elements at the vector tail to set to zero.

Other possibilities can be exploited from the fact that no boundary checking is performed.

2.3.4 Individual Functions

In what follows, the individual functions implementing vector operations are described.

VectorAdd

Description:	`VectorAdd` adds the value of each element in the source one vector with its counterpart in the source two vector, and places the result in the destination vector.
Include:	`dsp.h`

VectorAdd (Continued)

Prototype:

```
extern fractional* VectorAdd (
    int numElems,
    fractional* dstV,
    fractional* srcV1,
    fractional* srcV2
);
```

Arguments:

numElems	number of elements in source vectors
dstV	pointer to destination vector
srcV1	pointer to source one vector
srcV2	pointer to source two vector

Return Value:

Pointer to base address of destination vector.

Remarks:

If the absolute value of *srcV1*[n] + *srcV2*[n] is larger than $1-2^{-15}$, this operation results in saturation for the n-th element.
This function can be computed in place.
This function can be self applicable.

Source File:

vadd.s

Function Profile:

System resources usage:

W0..W4	used, not restored
ACCA	used, not restored
CORCON	saved, used, restored

DO and REPEAT instruction usage:
1 level DO instructions
no REPEAT instructions

Program words (24-bit instructions):
13

Cycles (including C-function call and return overheads):
17 + 3(*numElems*)

VectorConvolve

Description: `VectorConvolve` computes the convolution between two source vectors, and stores the result in a destination vector. The result is computed as follows:

$$y(n) = \sum_{k=0}^{n} x(k)h(n-k), \text{ for } 0 \le n < M$$

$$y(n) = \sum_{k=n-M+1}^{n} x(k)h(n-k), \text{ for } M \le n < N$$

$$y(n) = \sum_{k=n-M+1}^{N-1} x(k)h(n-k), \text{ for } N \le n < N+M-1$$

where x(k) = source one vector of size N, h(k) = source two vector of size M (with M ≤ N).

Include: `dsp.h`

Prototype:

```
extern fractional* VectorConvolve (
   int numElems1,
   int numElems2,
   fractional* dstV,
   fractional* srcV1,
   fractional* srcV2
);
```

Arguments:

numElems1	number of elements in source one vector
numElems2	number of elements in source two vector
dstV	pointer to destination vector
srcV1	pointer to source one vector
srcV2	pointer to source two vector

Return Value: Pointer to base address of destination vector.

Remarks: The number of elements in the source two vector *must* be less than or equal to the number of elements in the source one vector.
The destination vector *must* already exist, with exactly *numElems1*+*numElems2*-1 number of elements.
This function can be self applicable.

Source File: `vcon.s`

VectorConvolve (Continued)

Function Profile: System resources usage:

W0..W7	used, not restored
W8..W10	saved, used, restored
ACCA	used, not restored
CORCON	saved, used, restored

`DO` and `REPEAT` instruction usage:
2 level `DO` instructions
no `REPEAT` instructions

Program words (24-bit instructions):
58

Cycles (including C-function call and return overheads):
For N = *numElems1*, and M = *numElems2*,

$$28 + 13M + 6\sum_{m=1}^{M} m + (N - M)(7 + 3M)\text{, for M < N}$$

$$28 + 13M + 6\sum_{m=1}^{M} m\text{, for M = N}$$

.

VectorCopy

Description: `VectorCopy` copies the elements of the source vector into the beginning of an (already existing) destination vector, so that:
dstV[n] = *srcV*[n], 0 ≤ n < *numElems*

Include: `dsp.h`

Prototype:
```
extern fractional* VectorCopy (
   int numElems,
   fractional* dstV,
   fractional* srcV
);
```

Arguments:

numElems	number of elements in source vector
dstV	pointer to destination vector
srcV	pointer to source vector

Return Value: Pointer to base address of destination vector.

Remarks: The destination vector *must* already exist. Destination vectors *must* have, at least, *numElems* elements, but could be longer.
This function can be computed in place. See Additional Remarks at the end of the section for comments on this mode of operation.

Source File: `vcopy.s`

Function Profile: System resources usage:

W0..W3	used, not restored

`DO` and `REPEAT` instruction usage:
no `DO` instructions
1 level `REPEAT` instructions

Program words (24-bit instructions):
6

Cycles (including C-function call and return overheads):
12 + *numElems*

VectorCorrelate

Description: `VectorCorrelate` computes the correlation between two source vectors, and stores the result in a destination vector. The result is computed as follows:

$$r(n) = \sum_{k=0}^{N-1} x(k)y(k+n), \text{ for } 0 \le n < N + M - 1$$

where x(k) = source one vector of size N, y(k) = source two vector of size M (with M ≤ N).

Include: `dsp.h`

Prototype:

```
extern fractional* VectorCorrelate (
   int numElems1,
   int numElems2,
   fractional* dstV,
   fractional* srcV1,
   fractional* srcV2
);
```

Arguments:

numElems1	number of elements in source one vector
numElems2	number of elements in source two vector
dstV	pointer to destination vector
srcV1	pointer to source one vector
srcV2	pointer to source two vector

Return Value: Pointer to base address of destination vector.

Remarks: The number of elements in the source two vector *must* be less than or equal to the number of elements in the source one vector.
The destination vector *must* already exist, with exactly *numElems1*+*numElems2*-1 number of elements.
This function can be self applicable.
This function uses `VectorConvolve`.

Source File: `vcor.s.s`

Function Profile: System resources usage:
W0..W7 used, not restored,
plus resources from `VectorConvolve`

`DO` and `REPEAT` instruction usage:
1 level `DO` instructions
no `REPEAT` instructions,
plus `DO`/`REPEAT` instructions from `VectorConvolve`

Program words (24-bit instructions):
14,
plus program words from `VectorConvolve`

Cycles (including C-function call and return overheads):
19 + floor(M / 2) * 3, with M = *numElems2*,
plus cycles from `VectorConvolve`.

Note: In the description of `VectorConvolve` the number of cycles reported includes 4 cycles of C-function call overhead. Thus, the number of actual cycles from `VectorConvolve` to add to `VectorCorrelate` is 4 less than whatever number is reported for a stand-alone `VectorConvolve`.

VectorDotProduct

Description: `VectorDotProduct` computes the sum of the products between corresponding elements of the source one and source two vectors.

Include: `dsp.h`

Prototype:
```
extern fractional VectorDotProduct (
   int numElems,
   fractional* srcV1,
   fractional* srcV2
);
```

Arguments:

numElems	number of elements in source vectors
srcV1	pointer to source one vector
srcV2	pointer to source two vector

Return Value: Value of the sum of products.

Remarks: If the absolute value of the sum of products is larger than $1-2^{-15}$, this operation results in saturation.
This function can be self applicable.

Source File: `vdot.s`

Function Profile: System resources usage:

W0..W2	used, not restored
W4..W5	used, not restored
ACCA	used, not restored
CORCON	saved, used, restored

`DO` and `REPEAT` instruction usage:
1 level `DO` instructions
no `REPEAT` instructions

Program words (24-bit instructions):
13

Cycles (including C-function call and return overheads):
17 + 3(*numElems*)

.

VectorMax

Description: `VectorMax` finds the last element in the source vector whose value is greater than or equal to any previous vector element. Then, it outputs that maximum value and the index of the maximum element.

Include: `dsp.h`

Prototype:
```
extern fractional VectorMax (
   int numElems,
   fractional* srcV,
   int* maxIndex
);
```

Arguments:

numElems	number of elements in source vector
srcV	pointer to source vector
maxIndex	pointer to holder for index of (last) maximum element

Return Value: Maximum value in vector.

Remarks: If *srcV*`[i] = `*srcV*`[j] = maxVal`, and `i < j`, then `*`*maxIndex*` = j`.

Source File: `vmax.s`

VectorMax (Continued)

Function Profile: System resources usage:
W0..W5 used, not restored

`DO` and `REPEAT` instruction usage:
no `DO` instructions
no `REPEAT` instructions

Program words (24-bit instructions):
13

Cycles (including C-function call and return overheads):
14
if *numElems* = 1
20 + 8(*numElems* – 2)
if *srcV*[n] ≤ *srcV*[n + 1], 0 ≤ n < *numElems* – 1
19 + 7(*numElems* – 2)
if *srcV*[n] > *srcV*[n + 1], 0 ≤ n < *numElems* – 1

VectorMin

Description: `VectorMin` finds the last element in the source vector whose value is less than or equal to any previous vector element. Then, it outputs that minimum value and the index of the minimum element.

Include: `dsp.h`

Prototype:
```
extern fractional VectorMin (
    int numElems,
    fractional* srcV,
    int* minIndex
);
```

Arguments:

numElems	number of elements in source vector
srcV	pointer to source vector
minIndex	pointer to holder for index of (last) minimum element

Return Value: Minimum value in vector.

Remarks: If *srcV*[i] = *srcV*[j] = minVal, and i < j, then **minIndex* = j.

Source File: `vmin.s`

Function Profile: System resources usage:
W0..W5 used, not restored

`DO` and `REPEAT` instruction usage:
no `DO` instructions
no `REPEAT` instructions

Program words (24-bit instructions):
13

Cycles (including C-function call and return overheads):
14
if *numElems* = 1
20 + 8(*numElems* – 2)
if *srcV*[n] ≥ *srcV*[n + 1], 0 ≤ n < *numElems* – 1
19 + 7(*numElems* – 2)
if *srcV*[n] < *srcV*[n + 1], 0 ≤ n < *numElems* – 1

VectorMultiply

Description: `VectorMultiply` multiplies the value of each element in source one vector with its counterpart in source two vector, and places the result in the corresponding element of destination vector.

Include: `dsp.h`

Prototype:

```
extern fractional* VectorMultiply (
    int numElems,
    fractional* dstV,
    fractional* srcV1,
    fractional* srcV2
);
```

Arguments:

numElems	number of elements in source vector
dstV	pointer to destination vector
srcV1	pointer to source one vector
srcV2	pointer to source two vector

Return Value: Pointer to base address of destination vector.

Remarks: This operation is also known as vector element-by-element multiplication.
This function can be computed in place.
This function can be self applicable.

Source File: `vmul.s`

Function Profile: System resources usage:

W0..W5	used, not restored
ACCA	used, not restored
CORCON	saved, used, restored

`DO` and `REPEAT` instruction usage:
1 level `DO` instructions
no `REPEAT` instructions

Program words (24-bit instructions):
14

Cycles (including C-function call and return overheads):
17 + 4(*numElems*)

VectorNegate

Description: `VectorNegate` negates (changes the sign of) the values of the elements in the source vector, and places them in the destination vector.

Include: `dsp.h`

Prototype:

```
extern fractional* VectorNeg (
    int numElems,
    fractional* dstV,
    fractional* srcV
);
```

Arguments:

numElems	number of elements in source vector
dstV	pointer to destination vector
srcV	pointer to source vector

Return Value: Pointer to base address of destination vector.

Remarks: The negated value of 0x8000 is set to 0x7FFF.
This function can be computed in place.

Source File: `vneg.s`

VectorNegate (Continued)

Function Profile: System resources usage:

W0..W5	used, not restored
ACCA	used, not restored
CORCON	saved, used, restored

`DO` and `REPEAT` instruction usage:
1 level `DO` instructions
no `REPEAT` instructions

Program words (24-bit instructions):
16

Cycles (including C-function call and return overheads):
19 + 4(*numElems*)

VectorPower

Description: `VectorPower` computes the power of a source vector as the sum of the squares of its elements.

Include: `dsp.h`

Prototype:

```
extern fractional VectorPower (
   int numElems,
   fractional* srcV
);
```

Arguments:

numElems	number of elements in source vector
srcV	pointer to source vector

Return Value: Value of the vector's power (sum of squares).

Remarks: If the absolute value of the sum of squares is larger than $1-2^{-15}$, this operation results in saturation
This function can be self applicable.

Source File: `vpow.s`

Function Profile: System resources usage:

W0..W2	used, not restored
W4	used, not restored
ACCA	used, not restored
CORCON	saved, used, restored

`DO` and `REPEAT` instruction usage:
no `DO` instructions
1 level `REPEAT` instructions

Program words (24-bit instructions):
12

Cycles (including C-function call and return overheads):
16 + 2(*numElems*)

VectorScale

Description: `VectorScale` scales (multiplies) the values of all the elements in the source vector by a scale value, and places the result in the destination vector.

Include: `dsp.h`

Prototype:
```
extern fractional* VectorScale (
   int numElems,
   fractional* dstV,
   fractional* srcV,
   fractional sclVal
);
```

Arguments:

numElems	number of elements in source vector
dstV	pointer to destination vector
srcV	pointer to source vector
sclVal	value by which to scale vector elements

Return Value: Pointer to base address of destination vector.

Remarks: `sclVal` must be a fractional number in 1.15 format.
This function can be computed in place.

Source File: `vscl.s`

Function Profile: System resources usage:

W0..W5	used, not restored
ACCA	used, not restored
CORCON	saved, used, restored

`DO` and `REPEAT` instruction usage:
1 level `DO` instructions
no `REPEAT` instructions

Program words (24-bit instructions):
14

Cycles (including C-function call and return overheads):
18 + 3(*numElems*)

VectorSubtract

Description: `VectorSubtract` subtracts the value of each element in the source two vector from its counterpart in the source one vector, and places the result in the destination vector.

Include: `dsp.h`

Prototype:
```
extern fractional* VectorSubtract (
   int numElems,
   fractional* dstV,
   fractional* srcV1,
   fractional* srcV2
);
```

Arguments:

numElems	number of elements in source vectors
dstV	pointer to destination vector
srcV1	pointer to source one vector (minuend)
srcV2	pointer to source two vector (subtrahend)

Return Value: Pointer to base address of destination vector.

Remarks: If the absolute value of *srcV1*`[n] - `*srcV2*`[n]` is larger than $1\text{-}2^{-15}$, this operation results in saturation for the `n`-th element.
This function can be computed in place.
This function can be self applicable.

VectorSubtract (Continued)

Source File: vsub.s

Function Profile: System resources usage:

W0..W4	used, not restored
ACCA	used, not restored
ACCB	used, not restored
CORCON	saved, used, restored

DO and REPEAT instruction usage:
1 level DO instructions
no REPEAT instructions

Program words (24-bit instructions):
14

Cycles (including C-function call and return overheads):
17 + 4(*numElems*)

VectorZeroPad

Description: VectorZeroPad copies the source vector into the beginning of the (already existing) destination vector, and then fills with zeros the remaining numZeros elements of destination vector:
dstV[n] = *srcV*[n], 0 ≤ n < *numElems*
dstV[n] = 0, *numElems*≤n<*numElems*+*numZeros*

Include: dsp.h

Prototype:
```
extern fractional* VectorZeroPad (
   int numElems,
   int numZeros,
   fractional* dstV,
   fractional* srcV
);
```

Arguments:

numElems	number of elements in source vector
numZeros	number of elements to fill with zeros at the tail of destination vector
dstV	pointer to destination vector
srcV	pointer to source vector

Return Value: Pointer to base address of destination vector.

Remarks: The destination vector *must* already exist, with exactly *numElems* + *numZeros* number of elements.
This function can be computed in place. See Additional Remarks at the beginning of the section for comments on this mode of operation.
This function uses VectorCopy.

Source File: vzpad.s

VectorZeroPad (Continued)

Function Profile:

System resources usage:
W0..W6 used, not restored
plus resources from `VectorCopy`

`DO` and `REPEAT` instruction usage:
no `DO` instructions
1 level `REPEAT` instructions
plus `DO`/`REPEAT` from `VectorCopy`

Program words (24-bit instructions):
13,
plus program words from `VectorCopy`

Cycles (including C-function call and return overheads):
18 + *numZeros*
plus cycles from `VectorCopy`.

Note: In the description of `VectorCopy`, the number of cycles reported includes 3 cycles of C-function call overhead. Thus, the number of actual cycles from `VectorCopy` to add to `VectorCorrelate` is 3 less than whatever number is reported for a stand-alone `VectorCopy`.

2.4 WINDOW FUNCTIONS

A window is a vector with a specific value distribution within its domain ($0 \leq n <$ *numElems*). The particular value distribution depends on the characteristics of the window being generated.

Given a vector, its value distribution may be modified by applying a window to it. In these cases, the window *must* have the same number of elements as the vector to modify.

Before a vector can be windowed, the window must be created. Window initialization operations are provided which generate the values of the window elements. For higher numerical precision, these values are computed in floating-point arithmetic, and the resulting quantities stored as 1.15 fractionals.

To avoid excessive overhead when applying a window operation, a particular window could be generated once and used many times during the execution of the program. Thus, it is advisable to store the window returned by any of the initialization operations in a permanent (static) vector.

2.4.1 User Considerations

a) All the window initialization functions have been designed to generate window vectors allocated in default RAM memory space (X-Data or Y-Data).
b) The windowing function is designed to operate on vectors allocated in default RAM memory space (X-Data or Y-Data).
c) It is recommended that the STATUS Register (SR) be examined after completion of each function call.
d) Since the window initialization functions are implemented in C, consult the electronic documentation included in the release for up-to-date cycle count information.

2.4.2 Individual Functions

In what follows, the individual functions implementing window operations are described.

BartlettInit

Description:	`BartlettInit` initializes a Barlett window of length *numElems*.
Include:	`dsp.h`

Prototype:	`extern fractional* BartlettInit (`
	`  int numElems,`
	`  fractional* window`
	`);`

Arguments:	*numElems* number of elements in window *window* pointer to window to be initialized
Return Value:	Pointer to base address of initialized window.
Remarks:	The *window* vector *must* already exist, with exactly *numElems* number of elements.
Source File:	`initbart.c`

BartlettInit (Continued)

Function Profile: System resources usage:

W0..W7	used, not restored
W8..W14	saved, used, not restored

`DO` and `REPEAT` instruction usage:
None

Program words (24-bit instructions):
See the file “Readme for dsPIC Language Tools Libraries.txt” for this information.

Cycles (including C-function call and return overheads):
See the file “Readme for dsPIC Language Tools Libraries.txt” for this information.

BlackmanInit

Description: `BlackmanInit` initializes a Blackman (3 terms) window of length *`numElems`*.

Include: `dsp.h`

Prototype:

```
extern fractional* BlackmanInit (
   int numElems,
   fractional* window
);
```

Arguments:

`numElems`	number of elements in window
`window`	pointer to window to be initialized

Return Value: Pointer to base address of initialized window.

Remarks: The *`window`* vector *must* already exist, with exactly *`numElems`* number of elements.

Source File: `initblck.c`

Function Profile: System resources usage:

W0..W7	used, not restored
W8..W14	saved, used, not restored

`DO` and `REPEAT` instruction usage:
None

Program words (24-bit instructions):
See the file “readme.txt” in pic30_tools\src\dsp for this information.

Cycles (including C-function call and return overheads):
See the file “readme.txt” in pic30_tools\src\dsp for this information.

HammingInit

Description: `HammingInit` initializes a Hamming window of length *`numElems`*.

Include: `dsp.h`

Prototype:
```
extern fractional* HammingInit (
   int numElems,
   fractional* window
);
```

Arguments:

`numElems`	number of elements in window
`window`	pointer to window to be initialized

Return Value: Pointer to base address of initialized window.

Remarks: The *`window`* vector *must* already exist, with exactly *`numElems`* number of elements.

Source File: `inithamm.c`

Function Profile: System resources usage:

W0..W7	used, not restored
W8..W14	saved, used, not restored

`DO` and `REPEAT` instruction usage:
None

Program words (24-bit instructions):
See the file “readme.txt” in pic30_tools\src\dsp for this information.

Cycles (including C-function call and return overheads):
See the file “readme.txt” in pic30_tools\src\dsp for this information.

HanningInit

Description: `HanningInit` initializes a Hanning window of length *`numElems`*.

Include: `dsp.h`

Prototype:
```
extern fractional* HanningInit (
   int numElems,
   fractional* window
);
```

Arguments:

`numElems`	number of elements in window
`window`	pointer to window to be initialized

Return Value: Pointer to base address of initialized window.

Remarks: The *`window`* vector *must* already exist, with exactly *`numElems`* number of elements.

Source File: `inithann.c`

Function Profile: System resources usage:

W0..W7	used, not restored
W8..W14	saved, used, not restored

`DO` and `REPEAT` instruction usage:
None

Program words (24-bit instructions):
See the file “readme.txt” in pic30_tools\src\dsp for this information.

Cycles (including C-function call and return overheads):
See the file “readme.txt” in pic30_tools\src\dsp for this information.

KaiserInit

Description: `KaiserInit` initializes a Kaiser window with shape determined by argument *`betaVal`* and of length *`numElems`*.

Include: `dsp.h`

Prototype:
```
extern fractional* KaiserInit (
   int numElems,
   fractional* window,
   float betaVal
);
```

Arguments:

`numElems`	number of elements in window
`window`	pointer to window to be initialized
`betaVal`	window shaping parameter

Return Value: Pointer to base address of initialized window.

Remarks: The *`window`* vector *must* already exist, with exactly *`numElems`* number of elements.

Source File: `initkais.c`

Function Profile: System resources usage:

W0..W7	used, not restored
W8..W14	saved, used, not restored

`DO` and `REPEAT` instruction usage:
None

Program words (24-bit instructions):
See the file “readme.txt” in pic30_tools\src\dsp for this information.

Cycles (including C-function call and return overheads):
See the file “readme.txt” in pic30_tools\src\dsp for this information.

VectorWindow

Description: `VectorWindow` applies a window to a given source vector, and stores the resulting windowed vector in a destination vector.

Include: `dsp.h`

Prototype:
```
extern fractional* VectorWindow (
   int numElems,
   fractional* dstV,
   fractional* srcV,
   fractional* window
);
```

Arguments:

`numElems`	number of elements in source vector
`dstV`	pointer to destination vector
`srcV`	pointer to source vector
`window`	pointer to initialized window

Return Value: Pointer to base address of destination vector.

Remarks: The *`window`* vector *must* have already been initialized, with exactly *`numElems`* number of elements.
This function can be computed in place.
This function can be self applicable.
This function uses `VectorMultiply`.

Source File: `dowindow.s`

VectorWindow (Continued)

Function Profile:

System resources usage:
resources from `VectorMultiply`

`DO` and `REPEAT` instruction usage:
no `DO` instructions
no `REPEAT` instructions,
plus `DO`/`REPEAT` from `VectorMultiply`

Program words (24-bit instructions):
3,
plus program words from `VectorMultiply`

Cycles (including C-function call and return overheads):
9,
plus cycles from `VectorMultiply`.

Note: In the description of `VectorMultiply` the number of cycles reported includes 3 cycles of C-function call overhead. Thus, the number of actual cycles from `VectorMultiply` to add to `VectorWindow` is 3 less than whatever number is reported for a stand-alone `VectorMultiply`.

2.5 MATRIX FUNCTIONS

This section presents the concept of a fractional matrix, as considered by the DSP Library, and describes the individual functions which perform matrix operations.

2.5.1 Fractional Matrix Operations

A fractional matrix is a collection of numerical values, the matrix elements, allocated contiguously in memory, with the first element at the lowest memory address. One word of memory (two bytes) is used to store the value of each element, and this quantity must be interpreted as a fractional number represented in 1.15 format.

A pointer addressing the first element of the matrix is used as a handle which provides access to each of the matrix values. The address of the first element is referred to as the base address of the matrix. Because each element of the matrix is 16 bits, the base address *must* be aligned to an even address.

The two dimensional arrangement of a matrix is emulated in the memory storage area by placing its elements organized in row major order. Thus, the first value in memory is the first element of the first row. It is followed by the rest of the elements of the first row. Then, the elements of the second row are stored, and so on, until all the rows are in memory. This way, the element at row r and column c of a matrix with R rows and C columns is located from the matrix base address BA at:

$$BA + 2(C(r - 1) + c - 1), \text{ for } 1 \leq r \leq R, 1 \leq c \leq C.$$

Note that the factor of 2 is used because of the byte addressing capabilities of the 16-bit device.

Unary and binary fractional matrix operations are implemented in this library. The operand matrix in a unary operation is called the source matrix. In a binary operation the first operand is referred to as the source one matrix, and the second matrix as the source two matrix. Each operation applies some computation to one or several elements of the source matrix(ces). The operations result in a matrix, referred to as the destination matrix.

Some operations resulting in a matrix allow computation in place. This means the results of the operation is placed back into the source matrix (or the source one matrix for a binary operation). In this case, the destination matrix is said to (physically) replace the source (one) matrix. If an operation can be computed in place, it is indicated as such in the comments provided with the function description.

For some binary operations, the two operands can be the same (physical) source matrix, which means the operation is applied to the source matrix and itself. If this type of computation is possible for a given operation, it is indicated as such in the comments provided with the function description.

Some operations can be self applicable and computed in place.

All the fractional matrix operations in this library take as arguments the number of rows and the number of columns of the operand matrix(ces). Based on the values of these argument the following assumptions are made:

a) The sum of sizes of all the matrices involved in a particular operation falls within the range of available data memory for the target device.
b) In the case of binary operations the number of rows and columns of the operand matrices *must* obey the rules of vector algebra; i.e., for matrix addition and subtraction the two matrices must have the same number of rows and columns, while for matrix multiplication, the number of columns of the first operand must be the same as the number of rows of the second operand. The source matrix to the inversion operation must be square (the same number of rows as of columns), and non-singular (its determinant different than zero).
c) The destination matrix *must* be large enough to accept the results of an operation.

2.5.2 User Considerations

a) No boundary checking is performed by these functions. Out of range dimensions (including zero row and/or zero column matrices) as well as nonconforming use of source matrix sizes in binary operations may produce unexpected results.
b) The matrix addition and subtraction operations could lead to saturation if the sum of corresponding elements in the source(s) matrix(ces) is greater than $1-2^{-15}$ or smaller than -1.
c) The matrix multiplication operation could lead to saturation if the sum of products of corresponding row and column sets results in a value greater than $1-2^{-15}$ or smaller than -1.
d) It is recommended that the STATUS Register (SR) is examined after completion of each function call. In particular, users can inspect the SA, SB and SAB flags after the function returns to determine if saturation occurred.
e) All the functions have been designed to operate on fractional matrices allocated in default RAM memory space (X-Data or Y-Data).
f) Operations which return a destination matrix can be nested, so that for instance if:

   a = Op1 (b, c), with b = Op2 (d), and c = Op3 (e, f), then

   a = Op1 (Op2 (d), Op3 (e, f))

2.5.3 Additional Remarks

The description of the functions limits its scope to what could be considered the regular usage of these operations. However, since no boundary checking is performed during computation of these functions, you have the freedom to interpret the operation and its results as it fits some particular needs.

For instance, while computing the `MatrixMultiply` function, the dimensions of the intervening matrices does not necessarily need to be {*numRows1*, *numCos1Rows2*} for source one matrix, {*numCols1Rows2*, *numCols2*} for source two matrix, and {*numRows1*, *numCols2*} for destination matrix. In fact, all that is needed is that their sizes are large enough so that during computation the pointers do no exceed over their memory range.

As another example, when a source matrix of dimension {*numRows*, *numCols*} is transposed, the destination matrix has dimensions {*numCols*, *numRows*}. Thus, properly speaking the operation can be computed in place *only* if source matrix is square. Nevertheless, the operation can be successfully applied in place to non square matrices; all that needs to be kept in mind is the *implicit* change of dimensions.

Other possibilities can be exploited from the fact that no boundary checking is performed.

2.5.4 Individual Functions

In what follows, the individual functions implementing matrix operations are described.

MatrixAdd

Description: `MatrixAdd` adds the value of each element in the source one matrix with its counterpart in the source two matrix, and places the result in the destination matrix.

Include: `dsp.h`

Prototype:

```
extern fractional* MatrixAdd (
    int numRows,
    int numCols,
    fractional* dstM,
    fractional* srcM1,
    fractional* srcM2
);
```

Arguments:

numRows	number of rows in source matrices
numCols	number of columns in source matrices
dstM	pointer to destination matrix
srcM1	pointer to source one matrix
srcM2	pointer to source two matrix

Return Value: Pointer to base address of destination matrix.

Remarks: If the absolute value of *srcM1*`[r][c]`+*srcM2*`[r][c]` is larger than $1-2^{-15}$, this operation results in saturation for the `(r,c)`-th element.
This function can be computed in place.
This function can be self applicable.

Source File: `madd.s`

Function Profile: System resources usage:

W0..W4	used, not restored
ACCA	used, not restored
CORCON	saved, used, restored

`DO` and `REPEAT` instruction usage:
1 level `DO` instructions
no `REPEAT` instructions

Program words (24-bit instructions):
14

Cycles (including C-function call and return overheads):
20 + 3(*numRows* * *numCols*)

MatrixMultiply

Description: `MatrixMultiply` performs the matrix multiplication between the source one and source two matrices, and places the result in the destination matrix. Symbolically:

$$dstM[i][j] = \sum_{k}(srcM1[i][k])(srcM2i[k][j])$$

where:
$0 \le i < numRows1$
$0 \le j < numCols2$
$0 \le k < numCols1Rows2$

Include: `dsp.h`

Prototype:
```
extern fractional* MatrixMultiply (
    int numRows1,
    int numCols1Rows2,
    int numCols2,
    fractional* dstM,
    fractional* srcM1,
    fractional* srcM2
);
```

Arguments:

numRows1	number of rows in source one matrix
numCols1Rows2	number of columns in source one matrix; which *must* be the same as number of rows in source two matrix
numCols2	number of columns in source two matrix
dstM	pointer to destination matrix
srcM1	pointer to source one matrix
srcM2	pointer to source two matrix

Return Value: Pointer to base address of destination matrix.

Remarks: If the absolute value of

$$\sum_{k}(srcM1[i][k])(srcM2i[k][j])$$

is larger than $1\text{-}2^{-15}$, this operation results in saturation for the `(i,j)`-th element.
If the source one matrix is squared, then this function can be computed in place and can be self applicable. See Additional Remarks at the beginning of the section for comments on this mode of operation.

Source File: `mmul.s`

Function Profile: System resources usage:

W0..W7	used, not restored
W8..W13	saved, used, restored
ACCA	used, not restored
CORCON	saved, used, restored

`DO` and `REPEAT` instruction usage:
2 level `DO` instructions
no `REPEAT` instructions

Program words (24-bit instructions):
35

Cycles (including C-function call and return overheads):
36 + *numRows1* * (8 + *numCols2* * (7 + 4 * *numCols1Rows2*))

MatrixScale

Description: `MatrixScale` scales (multiplies) the values of all elements in the source matrix by a scale value, and places the result in the destination matrix.

Include: `dsp.h`

Prototype:
```
extern fractional* MatrixScale (
   int numRows,
   int numCols,
   fractional* dstM,
   fractional* srcM,
   fractional sclVal
);
```

Arguments:

numRows	number of rows in source matrix
numCols	number of columns in source matrix
dstM	pointer to destination matrix
srcM	pointer to source matrix
sclVal	value by which to scale matrix elements

Return Value: Pointer to base address of destination matrix.

Remarks: This function can be computed in place.

Source File: `mscl.s`

Function Profile: System resources usage:

W0..W5	used, not restored
ACCA	used, not restored
CORCON	saved, used, restored

`DO` and `REPEAT` instruction usage:
1 level `DO` instructions
no `REPEAT` instructions

Program words (24-bit instructions):
14

Cycles (including C-function call and return overheads):
20 + 3(*numRows* * *numCols*)

MatrixSubtract

Description: `MatrixSubtract` subtracts the value of each element in the source two matrix from its counterpart in the source one matrix, and places the result in the destination matrix.

Include: `dsp.h`

Prototype:
```
extern fractional* MatrixSubtract (
   int numRows,
   int numCols,
   fractional* dstM,
   fractional* srcM1,
   fractional* srcM2
);
```

Arguments:

numRows	number of rows in source matrix(ces)
numCols	number of columns in source matrix(ces)
dstM	pointer to destination matrix
srcM1	pointer to source one matrix (minuend)
srcM2	pointer to source two matrix (subtrahend)

Return Value: Pointer to base address of destination matrix.

MatrixSubtract (Continued)

Remarks: If the absolute value of *srcM1*[r][c]-*srcM2*[r][c] is larger than $1-2^{-15}$, this operation results in saturation for the (r,c)-th element.
This function can be computed in place.
This function can be self applicable.

Source File: msub.s

Function Profile: System resources usage:

W0..W4	used, not restored
ACCA	used, not restored
ACCB	used, not restored
CORCON	saved, used, restored

DO and REPEAT instruction usage:
1 level DO instructions
no REPEAT instructions

Program words (24-bit instructions):
15

Cycles (including C-function call and return overheads):
20 + 4(*numRows* * *numCols*)

MatrixTranspose

Description: MatrixTranspose transposes the rows by the columns in the source matrix, and places the result in destination matrix. In effect:
dstM[i][j] = *srcM*[j][i],
0 ≤ i < *numRows*, 0 ≤ j < *numCols*.

Include: dsp.h

Prototype:
```
extern fractional* MatrixTranspose (
   int numRows,
   int numCols,
   fractional* dstM,
   fractional* srcM
);
```

Arguments:

numRows	number of rows in source matrix
numCols	number of columns in source matrix
dstM	pointer to destination matrix
srcM	pointer to source matrix

Return Value: Pointer to base address of destination matrix.

Remarks: If the source matrix is square, this function can be computed in place. See Additional Remarks at the beginning of the section for comments on this mode of operation.

Source File: mtrp.s

Function Profile: System resources usage:

W0..W5	used, not restored

DO and REPEAT instruction usage:
2 level DO instructions
no REPEAT instructions

Program words (24-bit instructions):
14

Cycles (including C-function call and return overheads):
16 + *numCols* * (6 + (*numRows*-1) * 3)

2.5.5 Matrix Inversion

The result of inverting a non-singular, square, fractional matrix is another square matrix (of the same dimension) whose element values are not necessarily constrained to the discrete fractional set {-1, ..., $1-2^{-15}$}. Thus, no matrix inversion operation is provided for fractional matrices.

However, since matrix inversion is a very useful operation, an implementation based on floating-point number representation and arithmetic is provided within the DSP Library. Its description follows.

MatrixInvert

Description: `MatrixInvert` computes the inverse of the source matrix, and places the result in the destination matrix.

Include: `dsp.h`

Prototype:

```
extern float* MatrixInvert (
   int numRowsCols,
   float* dstM,
   float* srcM,
   float* pivotFlag,
   int* swappedRows,
   int* swappedCols
);
```

Arguments:

numRowCols	number of rows and columns in (square) source matrix
dstM	pointer to destination matrix
srcM	pointer to source matrix

Required for internal use:

pivotFlag	pointer to a length *numRowsCols* vector
swappedRows	pointer to a length *numRowsCols* vector
swappedCols	pointer to a length *numRowsCols* vector

Return Value: Pointer to base address of destination matrix, or `NULL` if source matrix is singular.

Remarks: Even though the vectors *pivotFlag*, *swappedRows*, and *swappedCols*, are for internal use only, they must be allocated prior to calling this function.
If source matrix is singular (determinant equal to zero) the matrix does not have an inverse. In this case the function returns `NULL`.
This function can be computed in place.

Source File: `minv.s` (assembled from C code)

Function Profile: System resources usage:

W0..W7	used, not restored
W8, W14	saved, used, restored

`DO` and `REPEAT` instruction usage:
None

Program words (24-bit instructions):
See the file “readme.txt” in pic30_tools\src\dsp for this information.

Cycles (including C-function call and return overheads):
See the file “readme.txt” in pic30_tools\src\dsp for this information.

2.6 FILTERING FUNCTIONS

This section presents the concept of a fractional filter, as considered by the DSP Library, and describes the individual functions which perform filter operations. The user may refer to example projects that utilize the DSP library filtering functions, in order to ascertain proper usage of functions. MPLAB IDE-based example projects/workspaces have been provided in the installation folder of the MPLAB C30 toolsuite.

2.6.1 Fractional Filter Operations

Filtering the data sequence represented by fractional vector x[n] ($0 \le n < N$) is equivalent to solving the difference equation:

$$y[n] + \sum_{p=1}^{P-1} (-a[p])(y[n-p]) = \sum_{m=0}^{M-1} (b[m])(x[n-m])$$

for every nth sample, which results into the filtered data sequence y[n]. In this sense, the fractional filter is characterized by the fractional vectors a[p] ($0 \le p < P$) and b[m] ($0 \le m < M$), referred to as the set of filter coefficients, which are designed to induce some pre-specified changes in the signal represented by the input data sequence.

When filtering it is important to know and manage the past history of the input and output data sequences (x[n], $-M + 1 \le n < 0$, and y[n], $-P + 1 \le n < 0$), which represent the initial conditions of the filtering operation. Also, when repeatedly applying the filter to contiguous sections of the input data sequence it is necessary to remember the final state of the last filtering operation (x[n], $N - M + 1 \le n < N - 1$, and y[n], $N - P + 1 \le n < N - 1$). This final state is then taken into consideration for the calculations of the next filtering stage. Accounting for the past history and current state is required in order to perform a correct filtering operation.

The management of the past history and current state of a filtering operation is commonly implemented via additional sequences (also fractional vectors), referred to as the filter delay line. Prior to applying the filter operation, the delay describes the past history of the filter. After performing the filtering operation, the delay contains a set of the most recently filtered data samples, and of the most recent output samples. (Note that to ensure correct operation of a particular filter implementation, it is advisable to initialize the delay values to zero by calling the corresponding initialization function.)

In the filter implementations provided with the DSP Library the input data sequence is referred to as the sequence of source samples, while the resulting filtered sequence is called the destination samples. The filter coefficients (a,b) and delay are usually thought of as making up a filter structure. In all filter implementations, the input and output data samples may be allocated in default RAM memory space (X-Data or Y-Data). Filter coefficients may reside either in X-Data memory or program memory, and filter delay values must be accessed *only* from Y-Data.

2.6.2 FIR and IIR Filter Implementations

The properties of a filter depend on the value distribution of its coefficients. In particular, two types of filters are of special interest: Finite Impulse Response (FIR) filters, for which $a[m] = 0$ when $1 \le m < M$, and Infinite Impulse Response (IIR) filters, those such that $a[0] \ne 0$, and $a[m] \ne 0$ for some m in {1, ..., M}. Other classifications within the FIR and IIR filter families account for the effects that the operation induces on input data sequences.

Furthermore, even though filtering consists on solving the difference equation stated above, several implementations are available which are more efficient than direct computation of the difference equation. Also, some other implementations are designed to execute the filtering operation under the constrains imposed by fractional arithmetic.

All these considerations lead to a proliferation of filtering operations, of which a subset is provided by the DSP Library.

2.6.3 Single Sample Filtering

The filtering functions provided in the DSP Library are designed for block processing. Each filter function accepts an argument named `numSamps` which indicates the number of words of input data (block size) to operate on. If single sample filtering is desired, you may set `numSamps` to 1. This will have the effect of filtering one input sample, and the function will compute a single output sample from the filter.

2.6.4 User Considerations

All the fractional filtering operations in this library rely on the values of either input parameters or data structure elements to specify the number of samples to process, and the sizes of the coefficients and delay vectors. Based on these values the following assumptions are made:

a) The sum of sizes of all the vectors (sample sequences) involved in a particular operation falls within the range of available data memory for the target device.

b) The destination vector *must* be large enough to accept the results of an operation.

c) No boundary checking is performed by these functions. Out of range sizes (including zero length vectors) as well as nonconforming use of source vectors and coefficient sets may produce unexpected results.

d) It is recommended that the STATUS Register (SR) is examined after completion of each function call. In particular, users can inspect the SA, SB and SAB flags after the function returns to determine if saturation occurred.

e) Operations which return a destination vector can be nested, so that for instance if:

   a = Op1 (b, c), with b = Op2 (d), and c = Op3 (e, f), then

   a = Op1 (Op2 (d), Op3 (e, f))

2.6.5 Individual Functions

In what follows, the individual functions implementing filtering operations are described. For further discussions on digital filters, please consult Alan Oppenheim and Ronald Schafer's "*Discrete-Time Signal Processing"*, Prentice Hall, 1989. For implementation details of Least Mean Square FIR filters, please refer to T. Hsia's "*Convergence Analysis of LMS and NLMS Adaptive Algorithms"*, Proc. ICASSP, pp. 667-670, 1983, as well as Sangil Park and Garth Hillman's "*On Acoustic-Echo Cancellation Implementation with Multiple Cascadable Adaptive FIR Filter Chips"*, Proc. ICASSP, 1989.

FIRStruct

Structure: `FIRStruct` describes the filter structure for any of the FIR filters.

Include: `dsp.h`

Declaration:

```
typedef struct {
  int numCoeffs;
  fractional* coeffsBase;
  fractional* coeffsEnd;
  int coeffsPage;
  fractional* delayBase;
  fractional* delayEnd;
  fractional* delay;
} FIRStruct;
```

Parameters:

numCoeffs	number of coefficients in filter (also M)
coeffsBase	base address for filter coefficients (also h)
coeffsEnd	end address for filter coefficients
coeffsPage	coefficients buffer page number
delayBase	base address for delay buffer
delayEnd	end address for delay buffer
delay	current value of delay pointer (also d)

Remarks:

Number of coefficients in filter is M.
Coefficients, h[m], defined in $0 \leq m < M$, either within X-Data or program memory.

Delay buffer d[m], defined in $0 \leq m < M$, *only* in Y-Data.

If coefficients are stored in X-Data space, *coeffsBase* points to the actual address where coefficients are allocated. If coefficients are stored in program memory, *coeffsBase* is the offset from the program page boundary containing the coefficients to the address in the page where coefficients are allocated. This latter value can be calculated using the inline assembly operator `psvoffset()`.

coeffsEnd is the address in X-Data space (or offset if in program memory) of the last byte of the filter coefficients buffer.

If coefficients are stored in X-Data space, *coeffsPage* must be set to 0xFF00 (defined value `COEFFS_IN_DATA`). If coefficients are stored in program memory, it is the program page number containing the coefficients. This latter value can be calculated using the inline assembly operator `psvpage()`.

delayBase points to the actual address where the delay buffer is allocated.

delayEnd is the address of the last byte of the filter delay buffer.

FIRStruct (Continued)

When the coefficients and delay buffers are implemented as circular increasing modulo buffers, both *coeffsBase* and *delayBase* *must* be aligned to a 'zero' power of two address (*coeffsEnd* and *delayEnd* are odd addresses). Whether these buffers are implemented as circular increasing modulo buffers or not is indicated in the remarks section of each FIR filter function description.

When the coefficients and delay buffers are not implemented as circular (increasing) modulo buffers, *coeffsBase* and *delayBase* *do not need to* be aligned to a 'zero' power of two address, and the values of *coeffsEnd* and *delayEnd* are ignored within the particular FIR Filter function implementation.

FIR

Description: FIR applies an FIR filter to the sequence of source samples, places the results in the sequence of destination samples, and updates the delay values.

Include: dsp.h

Prototype:

```
extern fractional* FIR (
   int numSamps,
   fractional* dstSamps,
   fractional* srcSamps,
   FIRStruct* filter
);
```

Arguments:

numSamps	number of input samples to filter (also N)
dstSamps	pointer to destination samples (also y)
srcSamps	pointer to source samples (also x)
filter	pointer to FIRStruct filter structure

Return Value: Pointer to base address of destination samples.

Remarks: Number of coefficients in filter is M.
Coefficients, h[m], defined in $0 \leq m < M$, implemented as a circular increasing modulo buffer.
Delay, d[m], defined in $0 \leq m < M$, implemented as a circular increasing modulo buffer.
Source samples, x[n], defined in $0 \leq n < N$.
Destination samples, y[n], defined in $0 \leq n < N$.
(See also FIRStruct, FIRStructInit and FIRDelayInit.)

Source File: fir.s

FIR (Continued)

Function Profile: System resources usage:

W0..W6	used, not restored
W8, W10	saved, used, restored
ACCA	used, not restored
CORCON	saved, used, restored
MODCON	saved, used, restored
XMODSTRT	saved, used, restored
XMODEND	saved, used, restored
YMODSTRT	saved, used, restored
PSVPAG	saved, used, restored (only if coefficients in P memory)

`DO` and `REPEAT` instruction usage:
1 level `DO` instructions
1 level `REPEAT` instructions

Program words (24-bit instructions):
55

Cycles (including C-function call and return overheads):
53 + N(4+M), or
56 + N(8+M) if coefficients in P memory.

Example Please refer to the MPLAB C30 installation folder for a sample project demonstrating the use of this function.

FIRDecimate

Description: `FIRDecimate` decimates the sequence of source samples at a rate of R to 1; or equivalently, it downsamples the signal by a factor of R. Effectively,
y[n] = x[Rn].
To diminish the effect of aliasing, the source samples are first filtered and then downsampled. The decimated results are stored in the sequence of destination samples, and the delay values updated.

Include: `dsp.h`

Prototype:

```
extern fractional* FIRDecimate (
   int numSamps,
   fractional* dstSamps,
   fractional* srcSamps,
   FIRStruct* filter,
   int rate
);
```

Arguments:

numSamps	number of *output* samples (also N, N = Rp, p integer)
dstSamp	pointer to destination samples (also y)
srcSamps	pointer to source samples (also x)
filter	pointer to `FIRStruct` filter structure
rate	rate of decimation (downsampling factor, also R)

Return Value: Pointer to base address of destination samples.

FIRDecimate (Continued)

Remarks: Number of coefficients in filter is M, with M an integer multiple of R.
Coefficients, h[m], defined in $0 \le m < M$, not implemented as a circular modulo buffer.
Delay, d[m], defined in $0 \le m < M$, not implemented as a circular modulo buffer.
Source samples, x[n], defined in $0 \le n < NR$.
Destination samples, y[n], defined in $0 \le n < N$.
(See also `FIRStruct`, `FIRStructInit`, and `FIRDelayInit`.)

Source File: `firdecim.s`

Function Profile: System resources usage:

W0..W7	used, not restored
W8..W12	saved, used, restored
ACCA	used, not restored
CORCON	saved, used, restored
PSVPAG	saved, used, restored (only if coefficients in P memory)

`DO` and `REPEAT` instruction usage:
1 level `DO` instructions
1 level `REPEAT` instructions

Program words (24-bit instructions):
48

Cycles (including C-function call and return overheads):
45 + N(10 + 2M), or
48 + N(13 + 2M) if coefficients in P memory.

FIRDelayInit

Description: `FIRDelayInit` initializes to zero the delay values in an `FIRStruct` filter structure.

Include: `dsp.h`

Prototype:

```
extern void FIRDelayInit (
    FIRStruct* filter
);
```

Arguments: *`filter`* pointer to `FIRStruct` filter structure.

Remarks: See description of `FIRStruct` structure above.
Note: FIR interpolator's delay is initialized by function `FIRInterpDelayInit`.

Source File: `firdelay.s`

Function Profile: System resources usage:

W0..W2	used, not restored

`DO` and `REPEAT` instruction usage:
no `DO` instructions
1 level `REPEAT` instructions

Program words (24-bit instructions):
7

Cycles (including C-function call and return overheads):
11 + M

FIRInterpolate

Description: `FIRInterpolate` interpolates the sequence of source samples at a rate of 1 to R; or equivalently, it upsamples the signal by a factor of R. Effectively,
y[n] = x[n/R].
To diminish the effect of aliasing, the source samples are first upsampled and then filtered. The interpolated results are stored in the sequence of destination samples, and the delay values updated.

Include: `dsp.h`

Prototype:
```
extern fractional* FIRInterpolate (
   int numSamps,
   fractional* dstSamps,
   fractional* srcSamps,
   FIRStruct* filter,
   int rate
);
```

Arguments:

numSamps	number of input samples (also N, N = Rp, p integer)
dstSamps	pointer to destination samples (also y)
srcSamps	pointer to source samples (also x)
filter	pointer to `FIRStruct` filter structure
rate	rate of interpolation (upsampling factor, also R)

Return Value: Pointer to base address of destination samples.

Remarks: Number of coefficients in filter is M, with M an integer multiple of R.
Coefficients, h[m], defined in $0 \le m < M$, not implemented as a circular modulo buffer.
Delay, d[m], defined in $0 \le m < M/R$, not implemented as a circular modulo buffer.
Source samples, x[n], defined in $0 \le n < N$.
Destination samples, y[n], defined in $0 \le n < NR$.
(See also `FIRStruct`, `FIRStructInit`, and `FIRInterpDelayInit`.)

Source File: `firinter.s`

Function Profile: System resources usage:

W0..W7	used, not restored
W8..W13	saved, used, restored
ACCA	used, not restored
CORCON	saved, used, restored
PSVPAG	saved, used, restored (only if coefficients in P memory)

`DO` and `REPEAT` instruction usage:
2 level `DO` instructions
1 level `REPEAT` instructions

Program words (24-bit instructions):
63

Cycles (including C-function call and return overheads):
45 + 6(M / R) + N(14 + M / R + 3M + 5R), or
48 + 6(M / R) + N(14 + M / R + 4M + 5R) if coefficients in P memory.

FIRInterpDelayInit

Description: `FIRInterpDelayInit` initializes to zero the delay values in an `FIRStruct` filter structure, optimized for use with an FIR interpolating filter.

Include: `dsp.h`

Prototype:
```
extern void FIRDelayInit (
   FIRStruct* filter,
   int rate
);
```

Arguments:

filter	pointer to `FIRStruct` filter structure
rate	rate of interpolation (upsampling factor, also R)

Remarks: Delay, d[m], defined in $0 \leq m < M/R$, with M the number of filter coefficients in the interpolator.
See description of `FIRStruct` structure above.

Source File: `firintdl.s`

Function Profile: System resources usage:
W0..W4 used, not restored

`DO` and `REPEAT` instruction usage:
no `DO` instructions
1 level `REPEAT` instructions

Program words (24-bit instructions):
13

Cycles (including C-function call and return overheads):
10 + 7M/R

FIRLattice

Description: `FIRLattice` uses a lattice structure implementation to apply an FIR filter to the sequence of source samples. It then places the results in the sequence of destination samples, and updates the delay values.

Include: `dsp.h`

Prototype:
```
extern fractional* FIRLattice (
   int numSamps,
   fractional* dstSamps,
   fractional* srcSamps,
   FIRStruct* filter
);
```

Arguments:

numSamps	number of input samples to filter (also N)
dstSamps	pointer to destination samples (also y)
srcSamps	pointer to source samples (also x)
filter	pointer to `FIRStruct` filter structure

Return Value: Pointer to base address of destination samples.

Remarks: Number of coefficients in filter is M.
Lattice coefficients, k[m], defined in $0 \leq m < M$, not implemented as a circular modulo buffer.
Delay, d[m], defined in $0 \leq m < M$, not implemented as a circular modulo buffer.
Source samples, x[n], defined in $0 \leq n < N$.
Destination samples, y[n], defined in $0 \leq n < N$.
(See also `FIRStruct`, `FIRStructInit` and `FIRDelayInit`.)

Source File: `firlatt.s`

FIRLattice (Continued)

Function Profile: System resources usage:

W0..W7	used, not restored
W8..W12	saved, used, restored
ACCA	used, not restored
ACCB	used, not restored
CORCON	saved, used, restored
PSVPAG	saved, used, restored (only if coefficients in P memory)

`DO` and `REPEAT` instruction usage:
2 level `DO` instructions
no `REPEAT` instructions

Program words (24-bit instructions):
50

Cycles (including C-function call and return overheads):
41 + N(4 + 7M)
44 + N(4 + 8M) if coefficients in P memory

FIRLMS

Description: `FIRLMS` applies an adaptive FIR filter to the sequence of source samples, stores the results in the sequence of destination samples, and updates the delay values.
The filter coefficients are also updated, at a sample-per-sample basis, using a Least Mean Square algorithm applied according to the values of the reference samples.

Include: `dsp.h`

Prototype:

```
extern fractional* FIRLMS (
   int numSamps,
   fractional* dstSamps,
   fractional* srcSamps,
   FIRStruct* filter,
   fractional* refSamps,
   fractional muVal
);
```

Arguments:

numSamps	number of input samples (also N)
dstSamps	pointer to destination samples (also y)
srcSamps	pointer to source samples (also x)
filter	pointer to `FIRStruct` filter structure
refSamps	pointer to reference samples (also r)
muVal	adapting factor (also mu)

Return Value: Pointer to base address of destination samples.

FIRLMS (Continued)

Remarks: Number of coefficients in filter is M.
Coefficients, h[m], defined in $0 \leq m < M$, implemented as a circular increasing modulo buffer.
delay, d[m], defined in $0 \leq m < M-1$, implemented as a circular increasing modulo buffer.
Source samples, x[n], defined in $0 \leq n < N$.
Reference samples, r[n], defined in $0 \leq n < N$.
Destination samples, y[n], defined in $0 \leq n < N$.
Adaptation:

$h\_m[n] = h\_m[n - 1] + mu * (r[n] - y[n]) * x[n - m]$,
for $0 \leq n < N$, $0 \leq m < M$.

The operation could result in saturation if the absolute value of (r[n] - y[n]) is greater than or equal to one.
Filter coefficients *must not* be allocated in program memory, because in that case their values could not be adapted. If filter coefficients are detected as allocated in program memory the function returns `NULL`.
(See also `FIRStruct`, `FIRStructInit` and `FIRDelayInit`.)

Source File: `firlms.s`

Function Profile: System resources usage:

W0..W7	used, not restored
W8..W12	saved, used, restored
ACCA	used, not restored
ACCB	used, not restored
CORCON	saved, used, restored
MODCON	saved, used, restored
XMODSTRT	saved, used, restored
XMODEND	saved, used, restored
YMODSTRT	saved, used, restored

`DO` and `REPEAT` instruction usage:
2 level `DO` instructions
1 level `REPEAT` instructions

Program words (24-bit instructions):
76

Cycles (including C-function call and return overheads):
61 + N(13 + 5M)

FIRLMSNorm

Description: `FIRLMSNorm` applies an adaptive FIR filter to the sequence of source samples, stores the results in the sequence of destination samples, and updates the delay values.
The filter coefficients are also updated, at a sample-per-sample basis, using a Normalized Least Mean Square algorithm applied according to the values of the reference samples.

Include: `dsp.h`

FIRLMSNorm (Continued)

Prototype:

```
extern fractional* FIRLMSNorm (
    int numSamps,
    fractional* dstSamps,
    fractional* srcSamps,
    FIRStruct* filter,
    fractional* refSamps,
    fractional muVal,
    fractional* energyEstimate
);
```

Arguments:

numSamps	number of input samples (also N)
dstSamps	pointer to destination samples (also y)
srcSamps	pointer to source samples (also x)
filter	pointer to `FIRStruct` filter structure
refSamps	pointer to reference samples (also r)
muVal	adapting factor (also mu)
energyEstimate	estimated energy value for the last M input signal samples, with M the number of filter coefficients

Return Value: Pointer to base address of destination samples.

Remarks: Number of coefficients in filter is M.
Coefficients, h[m], defined in $0 \le m < M$, implemented as a circular increasing modulo buffer.
delay, d[m], defined in $0 \le m < M$, implemented as a circular increasing modulo buffer.
Source samples, x[n], defined in $0 \le n < N$.
Reference samples, r[n], defined in $0 \le n < N$.
Destination samples, y[n], defined in $0 \le n < N$.
Adaptation:

h_m[n] = h_m[n – 1] + nu[n] * (r[n] – y[n]) * x[n – m],
for $0 \le n < N$, $0 \le m < M$,
where nu[n] = mu / (mu + E[n])
with $E[n]=E[n-1] + (x[n])^2 – (x[n – M + 1])^2$ an estimate of input signal energy.

On start up, *energyEstimate* should be initialized to the value of E[-1] (zero the first time the filter is invoked). Upon return, *energyEstimate* is updated to the value E[N – 1] (which may be used as the start up value for a subsequent function call if filtering an extension of the input signal).
The operation could result in saturation if the absolute value of (r[n] – y[n]) is greater than or equal to one.
Note: Another expression for the energy estimate is:
$E[n] = (x[n])^2 + (x[n-1])^2 + ... + (x[n – M + 2])^2$.
Thus, to avoid saturation while computing the estimate, the input sample values should be bound so that

$$\sum_{m=0}^{-M+2} (x[n+m])^2 < 1 \text{, for } 0 \le n < N.$$

Filter coefficients *must not* be allocated in program memory, because in that case their values could not be adapted. If filter coefficients are detected as allocated in program memory the function returns `NULL`. (See also `FIRStruct`, `FIRStructInit` and `FIRDelayInit`.)

Source File: `firlmsn.s`

FIRLMSNorm (Continued)

Function Profile: System resources usage:

W0..W7	used, not restored
W8..W13	saved, used, restored
ACCA	used, not restored
ACCB	used, not restored
CORCON	saved, used, restored
MODCON	saved, used, restored
XMODSTRT	saved, used, restored
XMODEND	saved, used, restored
YMODSTRT	saved, used, restored

`DO` and `REPEAT` instruction usage:
2 level `DO` instructions
1 level `REPEAT` instructions

Program words (24-bit instructions):
91

Cycles (including C-function call and return overheads):
66 + N(49 + 5M)

FIRStructInit

Description: `FIRStructInit` initializes the values of the parameters in an `FIRStruct` FIR Filter structure.

Include: `dsp.h`

Prototype:
```
extern void FIRStructInit (
  FIRStruct* filter,
  int numCoeffs,
  fractional* coeffsBase,
  int coeffsPage,
  fractional* delayBase
);
```

Arguments:

filter	pointer to `FIRStruct` filter structure
numCoeffs	number of coefficients in filter (also M)
coeffsBase	base address for filter coefficients (also h)
coeffsPage	coefficient buffer page number
delayBase	base address for delay buffer

Remarks: See description of `FIRStruct` structure above.
Upon completion, `FIRStructInit` initializes the `coeffsEnd` and `delayEnd` pointers accordingly. Also, `delay` is set equal to `delayBase`.

Source File: `firinit.s`

Function Profile: System resources usage:

W0..W5	used, not restored

`DO` and `REPEAT` instruction usage:
no `DO` instructions
no `REPEAT` instructions

Program words (24-bit instructions):
10

Cycles (including C-function call and return overheads):
19

IIRCanonic

Description: `IIRCanonic` applies an IIR filter, using a cascade of canonic (direct form II) biquadratic sections, to the sequence of source samples. It places the results in the sequence of destination samples, and updates the delay values.

Include: `dsp.h`

Prototype:

```
typedef struct {
  int numSectionsLess1;
  fractional* coeffsBase;
  int coeffsPage;
  fractional* delayBase;
  int initialGain;
  int finalShift;
} IIRCanonicStruct;

extern fractional* IIRCanonic (
   int numSamps,
   fractional* dstSamps,
   fractional* srcSamps,
   IIRCanonicStruct* filter
);
```

Arguments: Filter structure:

numSectionsLess1	1 less than number of cascaded second order (biquadratic) sections (also S-1)
coeffsBase	pointer to filter coefficients (also {a, b}), either within X-Data or program memory
coeffsPage	coefficients buffer page number, or `0xFF00 (defined value COEFFS_IN_DATA)` if coefficients in data space
delayBase	pointer to filter delay (also d), *only* in Y-Data
initialGain	initial gain value
finalShift	output scaling (shift left)

Filter Description:

numSamps	number of input samples to filter (also N)
dstSamps	pointer to destination samples (also y)
srcSamps	pointer to source samples (also x)
filter	pointer to `IIRCanonicStruct` filter structure

Return Value: Pointer to base address of destination samples.

Remarks: There are 5 coefficients per second order (biquadratic) sections arranged in the ordered set {a2[s], a1[s], b2[s], b1[s], b0[s]}, $0 \le s < S$.
Coefficient values should be generated with dsPICFD filter design package from Momentum Data Systems, Inc., or similar tool.
The delay is made up of two words of filter state per section {d1[s], d2[s]}, $0 \le s < S$.
Source samples, x[n], defined in $0 \le n < N$.
Destination samples, y[n], defined in $0 \le n < N$.
Initial gain value is applied to each input sample prior to *entering* the filter structure.
The output scale is applied as a shift to the output of the filter structure prior to storing the result in the output sequence. It is used to restore the filter gain to 0 dB. Shift count may be zero; if not zero, it represents the number of bits to shift: negative indicates shift left, positive is shift right.

Source File: `iircan.s`

IIRCanonic (Continued)

Function Profile: System resources usage:

W0..W7	used, not restored
W8..W11	saved, used, restored
ACCA	used, not restored
CORCON	saved, used, restored
PSVPAG	saved, used, restored

`DO` and `REPEAT` instruction usage:
2 level `DO` instructions
1 level `REPEAT` instructions

Program words (24-bit instructions):
42

Cycles (including C-function call and return overheads):
36 + N(8 + 7S), or
39 + N(9 + 12S) if coefficients in program memory.

IIRCanonicInit

Description: `IIRCanonicInit` initializes to zero the delay values in an `IIRCanonicStruct` filter structure.

Include: `dsp.h`

Prototype:
```
extern void IIRCanonicInit (
    IIRCanonicStruct* filter
);
```

Arguments: Filter structure:
(See description of `IIRCanonic` function).

Initialization Description:

filter	pointer to `IIRCanonicStruct` filter structure

Remarks: Two words of filter state per second order section {d1[s], d2[s]}, $0 \leq s < S$.

Source File: `iircan.s`

Function Profile: System resources usage:

W0, W1	used, not restored

`DO` and `REPEAT` instruction usage:
1 level `DO` instructions
no `REPEAT` instructions

Program words (24-bit instructions):
7

Cycles (including C-function call and return overheads):
10 + S2.

IIRLattice

Description: `IIRLattice` uses a lattice structure implementation to apply an IIR filter to the sequence of source samples. It then places the results in the sequence of destination samples, and updates the delay values.

Include: `dsp.h`

Prototype:

```
typedef struct {
  int order;
  fractional* kappaVals;
  fractional* gammaVals;
  int coeffsPage;
  fractional* delay;
} IIRLatticeStruct;

extern fractional* IIRLattice (
   int numSamps,
   fractional* dstSamps,
   fractional* srcSamps,
   IIRLatticeStruct* filter
);
```

Arguments: Filter structure:

order	filter order (also M, M ≤ N; see `FIRLattice` for N)
kappaVals	base address for lattice coefficients (also k), either in X-Data or program memory
gammaVals	base address for ladder coefficients (also g), either in X-Data or program memory. If `NULL`, the function will implement an all-pole filter.
coeffsPage	coefficients buffer page number, or 0xFF00 `(defined value COEFFS_IN_DATA)` if coefficients in data space
delay	base address for delay (also d), *only* in Y-Data

Filter Description:

numSamps	number of input samples to filter (also N, N Š M; see `IIRLatticeStruct` for M)
dstSamps	pointer to destination samples (also y)
srcSamps	pointer to source samples (also x)
filter	pointer to `IIRLatticeStruct` filter structure

Return Value: Pointer to base address of destination samples.

Remarks: Lattice coefficients, k[m], defined in 0 ≤ m ≤ M.
Ladder coefficients, g[m], defined in 0 ≤ m ≤ M (unless if implementing an all-pole filter).
Delay, d[m], defined in 0 ≤ m ≤ M.
Source samples, x[n], defined in 0 ≤ n < N.
Destination samples, y[n], defined in 0 ≤ n < N.
Note: The fractional implementation provided with this library is prone to saturation. Design and test the filter “off-line” using a floating-point implementation such as the OCTAVE model at the end of this section. Then, the intermediate forward and backward values should be monitored during the floating-point execution in search for levels outside the [-1, 1) range. If any one of the intermediate values spans outside of that range, the maximum absolute value should be used to scale the input signal prior to applying the fractional filter in real-time; i.e., multiply the signal by the inverse of that maximum. This scaling should prevent the fractional implementation from saturating.

Source File: `iirlatt.s`

IIRLattice (Continued)

Function Profile: System resources usage:

W0..W7	used, not restored
W8..W13	saved, used, restored
ACCA	used, not restored
ACCB	used, not restored
CORCON	saved, used, restored

`DO` and `REPEAT` instruction usage:
2 level `DO` instructions
no `REPEAT` instructions

Program words (24-bit instructions):
76

Cycles (including C-function call and return overheads):
46 + N(16 + 7M), or
49 + N(20 + 8M) if coefficients in program memory.

If implementing an all-pole filter:
46 + N(16 + 6M), or
49 + N(16 + 7M) if coefficients in program memory

IIRLatticeInit

Description: `IIRLatticeInit` initializes to zero the delay values in an `IIRLatticeStruct` filter structure.

Include: `dsp.h`

Prototype:
```
extern void IIRLatticeInit (
   IIRLatticeStruct* filter
);
```

Arguments: Filter structure:
(See description of `IIRLattice` function).

Initialization Description:

filter	pointer to `IIRLatticeStruct` filter structure.

Source File: `iirlattd.s`

Function Profile: System resources usage:

W0..W2	used, not restored

`DO` and `REPEAT` instruction usage:
no `DO` instructions
1 level `REPEAT` instructions

Program words (24-bit instructions):
6

Cycles (including C-function call and return overheads):
10 + M

IIRTransposed

Description: `IIRTransposed` applies an IIR filter, using a cascade of transposed (direct form II) biquadratic sections, to the sequence of source samples. It places the results in the sequence of destination samples, and updates the delay values.

Include: `dsp.h`

Prototype:

```
typedef struct {
  int numSectionsLess1;
  fractional* coeffsBase;
  int coeffsPage;
  fractional* delayBase1;
  fractional* delayBase2;
  int finalShift;
} IIRTransposedStruct;

extern fractional* IIRTransposed (
   int numSamps,
   fractional* dstSamps,
   fractional* srcSamps,
   IIRTransposedStruct* filter
);
```

Arguments: Filter structure:

numSectionsLess1	1 less than number of cascaded second order (biquadratic) sections (also S-1)
coeffsBase	pointer to filter coefficients (also {a, b}), either in X-Data or program memory
coeffsPage	coefficient buffer page number, or 0xFF00 `(defined value COEFFS_IN_DATA)` if coefficients in data space
delayBase1	pointer to filter state 1, with one word of delay per second order section (also d1), *only* in Y-Data
delayBase2	pointer to filter state 2, with one word of delay per second order section (also d2), *only* in Y-Data
finalShift	output scaling (shift left)

Filter Description:

numSamps	number of input samples to filter (also N)
dstSamps	pointer to destination samples (also y)
srcSamps	pointer to source samples (also x)
filter	pointer to `IIRTransposedStruct` filter structure

Return Value: Pointer to base address of destination samples.

Remarks: There are 5 coefficients per second order (biquadratic) section arranged in the ordered set {b0[s], b1[s], a1[s], b2[s],a2[s]}, $0 \le s < S$.
Coefficient values should be generated with dsPICFD filter design package from Momentum Data Systems, Inc., or similar tool.
The delay is made up of two independent buffers, each buffer containing one word of filter state per section {d2[s], d1[s]}, $0 \le s < S$.
Source samples, x[n], defined in $0 \le n < N$.
Destination samples, y[n], defined in $0 \le n < N$.
The output scale is applied as a shift to the output of the filter structure prior to storing the result in the output sequence. It is used to restore the filter gain to 0 dB. Shift count may be zero; if not zero, it represents the number of bits to shift: negative indicates shift left, positive is shift right.

Source File: `iirtrans.s`

IIRTransposed (Continued)

Function Profile: System resources usage:

W0..W7	used, not restored
W8..W11	saved, used, restored
ACCA	used, not restored
ACCB	used, not restored
CORCON	saved, used, restored
PSVPAG	saved, used, restored

`DO` and `REPEAT` instruction usage:
2 level `DO` instructions
1 level `REPEAT` instructions

Program words (24-bit instructions):
48

Cycles (including C-function call and return overheads):
35 + N(11 + 11S), or
38 + N(9 + 17S) if coefficients in P memory.
S is number of second order sections.

Example Please refer to the MPLAB C30 installation folder for a sample project demonstrating the use of this function.

IIRTransposedInit

Description: `IIRTransposedInit` initializes to zero the delay values in an `IIRTransposedStruct` filter structure.

Include: `dsp.h`

Prototype:
```
extern void IIRTransposedInit (
   IIRTransposedStruct* filter
);
```

Arguments: Filter structure:
(See description of `IIRTransposed` function).

Initialization Description:
`filter` pointer to `IIRTransposedStruct` filter structure.

Remarks: The delay is made up of two independent buffers, each buffer containing one word of filter state per section {d2[s], d1[s]}, $0 \leq s < S$.

Source File: `iirtrans.s`

Function Profile: System resources usage:

W0..W2	used, not restored

`DO` and `REPEAT` instruction usage:
1 level `DO` instructions
no `REPEAT` instructions

Program words (24-bit instructions):
8

Cycles (including C-function call and return overheads):
11 + 2S,
S is number of second order sections.

Example Please refer to the MPLAB C30 installation folder for a sample project demonstrating the use of this function.

2.6.6 OCTAVE model for analysis of `IIRLattice` filter

The following OCTAVE model may be used to examine the performance of an IIR Lattice Filter prior to using the fractional implementation provided by the function `IIRLattice`.

IIRLattice OCTAVE model

```
function [out, del, forward, backward] = iirlatt (in, kappas, gammas, delay)
## FUNCTION.-
## IIRLATT: IIR Fileter Lattice implementation.
##
##     [out, del, forward, backward] = iirlatt (in, kappas, gammas, delay)
##
##     forward: records intermediate forward values.
##     backward: records intermediate backward values.

#.....................................................................

## Get implicit parameters.
numSamps = length(in); numKapps = length(kappas);
if (gammas != 0)
   numGamms = length(gammas);
else
   numGamms = 0;
endif
numDels = length(delay); filtOrder = numDels-1;

## Error check.
if (numGamms != 0)
   if (numGamms != numKapps)
      fprintf ("ERROR! %d should be equal to %d.\n", numGamms, numKapps);
      return;
   endif
endif
if (numDels != numKapps)
   fprintf ("ERROR! %d should equal to %d.\n", numDels, numKapps);
   return;
endif

## Initialize.
M = filtOrder; out = zeros(numSamps,1); del = delay;
forward = zeros(numSamps*M,1); backward = forward; i = 0;

## Filter samples.
for n = 1:numSamps
   ## Get new sample.
   current = in(n);
```

```
   ## Lattice structure.
   for m = 1:M
      after     = current  - kappas(M+1-m) * del(m+1);
      del(m)    = del(m+1) + kappas(M+1-m) * after;
      i = i+1;
      forward(i) = current;
      backward(i) = after;
      current   = after;
   end
   del(M+1) = after;

   ## Ladder structure (computes output).
   if (gammas == 0)
      out(n) = del(M+1);
   else
      for m = 1:M+1
         out(n) = out(n) + gammas(M+2-m)*del(m);
      endfor
   endif
endfor

## Return.
return;

#.....................................................................

endfunction
```

2.7 TRANSFORM FUNCTIONS

This section presents the concept of a fractional transform, as considered by the DSP Library, and describes the individual functions which perform transform operations. The user may refer to example projects that utilize the DSP library Transform functions, in order to ascertain proper usage of functions. Example MPLAB IDE-based projects/workspaces have been provided in the installation folder of the MPLAB C30 toolsuite.

2.7.1 Fractional Transform Operations

A fractional transform is a linear, time invariant, discrete operation that when applied to a fractional time domain sample sequence, results in a fractional frequency in the frequency domain. Conversely, inverse fractional transform operation, when applied to frequency domain data, results in its time domain representation.

A set of transforms (and a subset of inverse transforms) are provided by the DSP Library. The first set applies a Discrete Fourier transform (or its inverse) to a complex data set (see below for a description of fractional complex values). The second set applies a Type II Discrete Cosine Transform (DCT) to a real valued sequence. These transforms have been designed to either operate out-of-place, or in-place. The former type populates an output sequence with the results of the transformation. In the latter, the input sequence is (physically) replaced by the transformed sequence. For out-of-place operations, enough memory to accept the results of the computation must be provided.

The transforms make use of transform factors (or constants) which must be supplied to the transforming function during its invocation. These factors, which are complex data sets, are computed in floating-point arithmetic, and then transformed into fractionals for use by the operations. To avoid excessive computational overhead when applying a transformation, a particular set of transform factors could be generated once and used many times during the execution of the program. Thus, it is advisable to store the factors returned by any of the initialization operations in a permanent (static) complex vector. It is also advantageous to generate the factors “off-line”, and place them in program memory, and use them when the program is later executing. This way, not only cycles, but also RAM memory is saved when designing an application which involves transformations.

2.7.2 Fractional Complex Vectors

A complex data vector is represented by a data set in which every pair of values represents an element of the vector. The first value in the pair is the real part of the element, and the second its imaginary part. Both the real and imaginary parts are stored in memory using one word (two bytes) for each, and must be interpreted as 1.15 fractionals. As with the fractional vector, the fractional complex vector stores its elements consecutively in memory.

The organization of data in a fractional complex vector may be addressed by the following data structure:

```
#ifdef fractional
#ifndef fractcomplex
typedef struct {
  fractional real;
  fractional imag;
} fractcomplex;
#endif
#endif
```

2.7.3 User Considerations

a) No boundary checking is performed by these functions. Out of range sizes (including zero length vectors) as well as nonconforming use of source complex vectors and factor sets may produce unexpected results.

b) It is recommended that the STATUS Register (SR) is examined after completion of each function call. In particular, users can inspect the SA, SB and SAB flags after the function returns to determine if saturation occurred.

c) The input and output complex vectors involved in the family of transformations *must* be allocated in Y-Data memory. Transforms factors may be allocated either in X-Data or program memory.

d) Because bit reverse addressing requires the vector set to be modulo aligned, the input and output complex vectors in operations using either explicitly or implicitly the `BitReverseComplex` function must be properly allocated.

e) Operations which return a destination complex vector can be nested, so that for instance if:

   a = Op1 (b, c), with b = Op2 (d), and c = Op3 (e, f), then

   a = Op1 (Op2 (d), Op3 (e, f)).

In what follows, the individual functions implementing transform and inverse transform operations are described.

2.7.4 Individual Functions

BitReverseComplex

Description: `BitReverseComplex` reorganizes the elements of a complex vector in bit reverse order.

Include: `dsp.h`

Prototype:
```
extern fractcomplex* BitReverseComplex (
   int log2N,
   fractcomplex* srcCV
);
```

Arguments:

log2N	based 2 logarithm of N (number of complex elements in source vector)
srcCV	pointer to source complex vector

Return Value: Pointer to base address of source complex vector.

Remarks: N *must* be an integer power of 2.
The *srcCV* vector must be allocated at a modulo alignment of N.
This function operates in place.

Source File: `bitrev.s`

Function Profile: System resources usage:

W0..W7	used, not restored
MODCON	saved, used, restored
XBREV	saved, used, restored

`DO` and `REPEAT` instruction usage:
1 level `DO` instructions
no `REPEAT` instructions

Program words (24-bit instructions):
27

Cycles (including C-function call and return overheads):
See below:

Example Please refer to the MPLAB C30 installation folder for a sample project demonstrating the use of this function.

Transform Size	# Complex Elements	# Cycles
32 point	32	245
64 point	64	485
128 point	128	945
256 point	256	1905

CosFactorInit

Description: `CosFactorInit` generates the first half of the set of cosine factors required by a Type II Discrete Cosine Transform, and places the result in the complex destination vector. Effectively, the set contains the values:

$CN(k) = e^{j\frac{\pi k}{2N}}$, where $0 \le k < N/2$.

Include: `dsp.h`

Prototype:
```
extern fractcomplex* CosFactorInit (
   int log2N,
   fractcomplex* cosFactors
);
```

Arguments:

log2N	based 2 logarithm of N (number of complex factors needed by a DCT)
cosFactors	pointer to complex cosine factors

Return Value: Pointer to base address of cosine factors.

Remarks: N *must* be an integer power of 2.
Only the first N/2 cosine factors are generated.
A complex vector of size N/2 *must* have already been allocated and assigned to *cosFactors* prior to invoking the function. The complex vector *should* reside in X-Data memory.
Factors are computed in floating-point arithmetic and converted to 1.15 complex fractionals.

Source File: `initcosf.c`

Function Profile: System resources usage:

W0..W7	used, not restored
W8..W14	saved, used, restored

`DO` and `REPEAT` instruction usage:
None

Program words (24-bit instructions):
See the file “readme.txt” in pic30_tools\src\dsp for this information.

Cycles (including C-function call and return overheads):
See the file “readme.txt” in pic30_tools\src\dsp for this information.

DCT

Description: `DCT` computes the Discrete Cosine Transform of a source vector, and stores the results in the destination vector.

Include: `dsp.h`

Prototype:
```
extern fractional* DCT (
   int log2N,
   fractional* dstV,
   fractional* srcV,
   fractcomplex* cosFactors,
   fractcomplex* twidFactors,
   int factPage
);
```

DCT (Continued)

Arguments:	*log2N*	based 2 logarithm of N (number of complex elements in source vector)
	dstCV	pointer to destination vector
	srcCV	pointer to source vector
	cosFactors	pointer to cosine factors
	twidFactors	pointer to twiddle factors
	factPage	memory page for transform factors
Return Value:	Pointer to base address of destination vector.	
Remarks:	N *must* be an integer power of 2. This function operates out of place. A vector of size 2N elements, *must* already have been allocated and assigned to *dstV*. The *dstV* vector must be allocated at a modulo alignment of N. The results of computation are stored in the first N elements of the destination vector. To avoid saturation (overflow) during computation, the values of the source vector *should* be in the range [-0.5, 0.5]. Only the first N/2 cosine factors are needed. Only the first N/2 twiddle factors are needed. If the transform factors are stored in X-Data space, *cosFactors* and *twidFactors* point to the actual address where the factors are allocated. If the transform factors are stored in program memory, *cosFactors* and *twidFactors* are the offset from the program page boundary where the factors are allocated. This latter value can be calculated using the inline assembly operator `psvoffset()`. If the transform factors are stored in X-Data space, *factPage* must be set to 0xFF00 (defined value `COEFFS_IN_DATA`). If they are stored in program memory, *factPage* is the program page number containing the factors. This latter value can be calculated using the inline assembly operator `psvpage()`. The twiddle factors *must* be initialized with `conjFlag` set to a value different than zero. Only the first N/2 cosine factors are needed. Output is scaled by the factor $1/(\sqrt{2N})$	
Source File:	`dctoop.s`	

DCT (Continued)

Function Profile: System resources usage:

W0..W5 used, not restored

plus system resources from VectorZeroPad, and DCTIP.

DO and REPEAT instruction usage:

no DO instructions

no REPEAT instructions

plus DO/REPEAT instructions from VectorZeroPad, and DCTIP.

Program words (24-bit instructions):

16

plus program words from VectorZeroPad, and DCTIP.

Cycles (including C-function call and return overheads):

22

plus cycles from VectorZeroPad, and DCTIP.

Note: In the description of VectorZeroPad the number of cycles reported includes 4 cycles of C-function call overhead. Thus, the number of actual cycles from VectorZeroPad to add to DCT is 4 less than whatever number is reported for a stand-alone VectorZeroPad. In the same way, the number of actual cycles from DCTIP to add to DCT is 3 less than whatever number is reported for a stand-alone DCTIP.

DCTIP

Description: DCTIP computes the Discrete Cosine Transform of a source vector in place.

Include: dsp.h

Prototype:

```
extern fractional* DCTIP (
   int log2N,
   fractional* srcV,
   fractcomplex* cosFactors,
   fractcomplex* twidFactors,
   int factPage
);
```

Arguments:

log2N	based 2 logarithm of N (number of complex elements in source vector)
srcCV	pointer to source vector
cosFactors	pointer to cosine factors
twidFactors	pointer to twiddle factors
factPage	memory page for transform factors

Return Value: Pointer to base address of destination vector.

DCTIP (Continued)

Remarks:

N *must* be an integer power of 2.
This function expects that the source vector has been zero padded to length 2N.
The *srcV* vector must be allocated at a modulo alignment of N.
The results of computation are stored in the first N elements of source vector.
To avoid saturation (overflow) during computation, the values of the source vector *should* be in the range [-0.5, 0.5].
Only the first N / 2 cosine factors are needed.
Only the first N / 2 twiddle factors are needed.
If the transform factors are stored in X-Data space, *cosFactors* and *twidFactors* point to the actual address where the factors are allocated. If the transform factors are stored in program memory, *cosFactors* and *twidFactors* are the offset from the program page boundary where the factors are allocated. This latter value can be calculated using the inline assembly operator `psvoffset()`.
If the transform factors are stored in X-Data space, *factPage* must be set to 0xFF00 (defined value `COEFFS_IN_DATA`). If they are stored in program memory, *factPage* is the program page number containing the factors. This latter value can be calculated using the inline assembly operator `psvpage()`.
The twiddle factors *must* be initialized with `conjFlag` set to a value different than zero.

Output is scaled by the factor $1/(\sqrt{2N})$.

Source File: `dctoop.s`

Function Profile:

System resources usage:

W0..W7	used, not restored
W8..W13	saved, used, restored
ACCA	used, not restored
CORCON	saved, used, restored
PSVPAG	saved, used, restored (only if coefficients in P memory)

`DO` and `REPEAT` instruction usage:
1 level `DO` instructions
1 level `REPEAT` instructions
plus `DO`/`REPEAT` instructions from `IFFTComplexIP`.

Program words (24-bit instructions):
92
plus program words from `IFFTComplexIP`.

Cycles (including C-function call and return overheads):
71 + 10N, or
73 + 11N if factors in program memory,
plus cycles from `IFFTComplexIP`

Note: In the description of `IFFTComplexIP` the number of cycles reported includes 4 cycles of C-function call overhead. Thus, the number of actual cycles from `IFFTComplexIP` to add to `DCTIP` is 4 less than whatever number is reported for a stand-alone `IFFTComplexIP`.

FFTComplex

Description:

`FFTComplex` computes the Discrete Fourier Transform of a source complex vector, and stores the results in the destination complex vector.

FFTComplex (Continued)

Include: `dsp.h`

Prototype:
```
extern fractcomplex* FFTComplex (
   int log2N,
   fractcomplex* dstCV,
   fractcomplex* srcCV,
   fractcomplex* twidFactors,
   int factPage
);
```

Arguments:

log2N	based 2 logarithm of N (number of complex elements in source vector)
dstCV	pointer to destination complex vector
srcCV	pointer to source complex vector
twidFactors	base address of twiddle factors
factPage	memory page for transform factors

Return Value: Pointer to base address of destination complex vector.

Remarks: N *must* be an integer power of 2.
This function operates out of place. A complex vector, large enough to receive the results of the operation, *must* already have been allocated and assigned to *dstCV*.
The *dstCV* vector must be allocated at a modulo alignment of N.
The elements in source complex vector are expected in natural order.
The elements in destination complex vector are generated in natural order.
To avoid saturation (overflow) during computation, the magnitude of the values of the source complex vector *should* be in the range [-0.5, 0.5].
Only the first N/2 twiddle factors are needed.
If the twiddle factors are stored in X-Data space, *twidFactors* points to the actual address where the factors are allocated. If the twiddle factors are stored in program memory, *twidFactors* is the offset from the program page boundary where the factors are allocated. This latter value can be calculated using the inline assembly operator `psvoffset()`.
If the twiddle factors are stored in X-Data space, *factPage* must be set to 0xFF00 (defined value `COEFFS_IN_DATA`). If they are stored in program memory, *factPage* is the program page number containing the factors. This latter value can be calculated using the inline assembly operator `psvpage()`.
The twiddle factors *must* be initialized with `conjFlag` set to zero.
Output is scaled by the factor 1/N.

Source File: `fftoop.s`

FFTComplex (Continued)

Function Profile: System resources usage:

W0..W4 used, not restored

plus system resources from `VectorCopy`, `FFTComplexIP`, and `BitReverseComplex`.

`DO` and `REPEAT` instruction usage:

no `DO` instructions

no `REPEAT` instructions

plus `DO`/`REPEAT` instructions from `VectorCopy`, `FFTComplexIP`, and `BitReverseComplex`.

Program words (24-bit instructions):

17

plus program words from `VectorCopy`, `FFTComplexIP`, and `BitReverseComplex`.

Cycles (including C-function call and return overheads):

23

plus cycles from `VectorCopy`, `FFTComplexIP`, and `BitReverseComplex`.

Note: In the description of `VectorCopy` the number of cycles reported includes 3 cycles of C-function call overhead. Thus, the number of actual cycles from `VectorCopy` to add to `FFTComplex` is 3 less than whatever number is reported for a stand-alone `VectorCopy`. In the same way, the number of actual cycles from `FFTComplexIP` to add to `FFTComplex` is 4 less than whatever number is reported for a stand-alone `FFTComplexIP`. And those from `BitReverseComplex` are 2 less than whatever number is reported for a stand-alone `FFT-Complex`.

FFTComplexIP

Description: `FFTComplexIP` computes the Discrete Fourier Transform of a source complex vector in place..

Include: `dsp.h`

Prototype:

```
extern fractcomplex* FFTComplexIP (
   int log2N,
   fractcomplex* srcCV,
   fractcomplex* twidFactors,
   int factPage
);
```

Arguments:

log2N	based 2 logarithm of N (number of complex elements in source vector)
srcCV	pointer to source complex vector
twidFactors	base address of twiddle factors
factPage	memory page for transform factors

Return Value: Pointer to base address of source complex vector.

FFTComplexIP (Continued)

Remarks: N *must* be an integer power of 2.
The elements in source complex vector are expected in natural order.
The resulting transform is stored in bit reverse order.
To avoid saturation (overflow) during computation, the magnitude of the values of the source complex vector should be in the range [-0.5, 0.5].
Only the first N/2 twiddle factors are needed.
If the twiddle factors are stored in X-Data space, *twidFactors* points to the actual address where the factors are allocated. If the twiddle factors are stored in program memory, *twidFactors* is the offset from the program page boundary where the factors are allocated. This latter value can be calculated using the inline assembly operator `psvoffset()`.
If the twiddle factors are stored in X-Data space, *factPage* must be set to 0xFF00 (defined value `COEFFS_IN_DATA`). If they are stored in program memory, *factPage* is the program page number containing the factors. This latter value can be calculated using the inline assembly operator `psvpage()`.
The twiddle factors *must* be initialized with `conjFlag` set to zero.
Output is scaled by the factor 1/N.

Source File: `fft.s`

Function Profile: System resources usage:

W0..W7	used, not restored
W8..W13	saved, used, restored
ACCA	used, not restored
ACCB	used, not restored
CORCON	saved, used, restored
PSVPAG	saved, used, restored (only if coefficients in P memory)

`DO` and `REPEAT` instruction usage:
2 level `DO` instructions
no `REPEAT` instructions

Program words (24-bit instructions):
59

Cycles (including C-function call and return overheads):
See table below

Example: Please refer to the MPLAB C30 installation folder for a sample project demonstrating the use of this function.

Transform Size	# Cycles if Twiddle Factors in X-mem	# Cycles if Twiddle Factors in P-mem
32 point	1,633	1,795
64 point	3,739	4,125
128 point	8,485	9,383
256 point	19,055	21,105

IFFTComplex

Description: `IFFTComplex` computes the Inverse Discrete Fourier Transform of a source complex vector, and stores the results in the destination complex vector.

Include: `dsp.h`

IFFTComplex (Continued)

Prototype:

```
extern fractcomplex* IFFTComplex (
   int log2N,
   fractcomplex* dstCV,
   fractcomplex* srcCV,
   fractcomplex* twidFactors,
   int factPage
);
```

Arguments:

log2N	based 2 logarithm of N (number of complex elements in source vector)
dstCV	pointer to destination complex vector
srcCV	pointer to source complex vector
twidFactors	base address of twiddle factors
factPage	memory page for transform factors

Return Value:

Pointer to base address of destination complex vector.

Remarks:

N *must* be an integer power of 2.
This function operates out of place. A complex vector, large enough to receive the results of the operation, *must* already have been allocated and assigned to *dstCV*.
The *dstCV* vector must be allocated at a modulo alignment of N.
The elements in source complex vector are expected in natural order.
The elements in destination complex vector are generated in natural order.
To avoid saturation (overflow) during computation, the magnitude of the values of the source complex vector *should* be in the range [-0.5, 0.5].
If the twiddle factors are stored in X-Data space, *twidFactors* points to the actual address where the factors are allocated. If the twiddle factors are stored in program memory, *twidFactors* is the offset from the program page boundary where the factors are allocated. This latter value can be calculated using the inline assembly operator `psvoffset()`.
If the twiddle factors are stored in X-Data space, *factPage* must be set to 0xFF00 (defined value `COEFFS_IN_DATA`). If they are stored in program memory, *factPage* is the program page number containing the factors. This latter value can be calculated using the inline assembly operator `psvpage()`.
The twiddle factors *must* be initialized with `conjFlag` set to a value other than zero.
Only the first N/2 twiddle factors are needed.

Source File:

`ifftoop.s`

IFFTComplex (Continued)

Function Profile: System resources usage:

W0..W4 used, not restored

plus system resources from `VectorCopy`, and `IFFTComplexIP`.

`DO` and `REPEAT` instruction usage:

no `DO` instructions

no `REPEAT` instructions

plus `DO`/`REPEAT` instructions from `VectorCopy`, and `IFFTComplexIP`.

Program words (24-bit instructions):

12

plus program words from `VectorCopy`, and `IFFTComplexIP`.

Cycles (including C-function call and return overheads):

15

plus cycles from `VectorCopy`, and `IFFTComplexIP`.

Note: In the description of `VectorCopy` the number of cycles reported includes 3 cycles of C-function call overhead. Thus, the number of actual cycles from `VectorCopy` to add to `IFFTComplex` is 3 less than whatever number is reported for a stand-alone `VectorCopy`. In the same way, the number of actual cycles from `IFFTComplexIP` to add to `IFFTComplex` is 4 less than whatever number is reported for a stand-alone `IFFTComplexIP`.

IFFTComplexIP

Description: `IFFTComplexIP` computes the Inverse Discrete Fourier Transform of a source complex vector in place..

Include: `dsp.h`

Prototype:
```
extern fractcomplex* IFFTComplexIP (
   int log2N,
   fractcomplex* srcCV,
   fractcomplex* twidFactors,
   int factPage
);
```

Arguments:

log2N	based 2 logarithm of N (number of complex elements in source vector)
srcCV	pointer to source complex vector
twidFactors	base address of twiddle factors
factPage	memory page for transform factors

Return Value: Pointer to base address of source complex vector.

Remarks: N *must* be an integer power of 2.
The elements in source complex vector are expected in bit reverse order. The resulting transform is stored in natural order.
The *srcCV* vector must be allocated at a modulo alignment of N.
To avoid saturation (overflow) during computation, the magnitude of the values of the source complex vector *should* be in the range [-0.5, 0.5].
If the twiddle factors are stored in X-Data space, *twidFactors* points to the actual address where the factors are allocated. If the twiddle factors are stored in program memory, *twidFactors* is the offset from the program page boundary where the factors are allocated. This latter value can be calculated using the inline assembly operator `psvoffset()`.
If the twiddle factors are stored in X-Data space, *factPage* must be set to 0xFF00 (defined value `COEFFS_IN_DATA`). If they are stored in program memory, *factPage* is the program page number containing the factors. This latter value can be calculated using the inline assembly operator `psvpage()`.
The twiddle factors *must* be initialized with `conjFlag` set to a value other than zero.
Only the first N/2 twiddle factors are needed.

Source File: `ifft.s`

IFFTComplexIP (Continued)

Function Profile: System resources usage:

W0..W3 used, not restored
plus system resources from FFTComplexIP, and BitReverseComplex.

DO and REPEAT instruction usage:

no DO instructions
no REPEAT instructions
plus DO/REPEAT instructions from FFTComplexIP, and BitReverseComplex.

Program words (24-bit instructions):

11
plus program words from FFTComplexIP, and BitReverseComplex.

Cycles (including C-function call and return overheads):

15
plus cycles from FFTComplexIP, and BitReverseComplex.

Note: In the description of FFTComplexIP the number of cycles reported includes 3 cycles of C-function call overhead. Thus, the number of actual cycles from FFTComplexIP to add to IFFTComplexIP is 3 less than whatever number is reported for a stand-alone FFTComplexIP. In the same way, the number of actual cycles from BitReverseComplex to add to IFFTComplexIP is 2 less than whatever number is reported for a stand-alone BitReverseComplex.

SquareMagnitudeCplx

Description: SquareMagnitudeCplx computes the squared magnitude of each element in a complex source vector.

Include: dsp.h

Prototype:

```
extern fractional* SquareMagnitudeCplx (
    int numElems,
  fractcomplex* srcV,
  fractional* dstV
);
```

Arguments:

numElems	number of elements in the complex source vector
srcV	pointer to complex source vector
dstV	pointer to real destination vector

Return Value: Pointer to base address of destination vector.

Remarks: If the sum of squares of the real and imaginary parts of a complex element in the source vector is larger than $1-2^{-15}$, this operation results in saturation.
This function can be used to operate in-place on a source data set.

Source File: cplxsqrmag.s

SquareMagnitudeCplx (Continued)

Function Profile: System resources usage:

W0..W2	used, not restored
W4, W5, W10	saved, used, restored
ACCA	used, not restored
CORCON	saved, used, restored

`DO` and `REPEAT` instruction usage:
1 level `DO` instructions
no `REPEAT` instructions

Program words (24-bit instructions):
19

Cycles (including C-function call and return overheads):
20 + 3(*numElems*)

Example: Please refer to the MPLAB C30 installation folder for a sample project demonstrating the use of this function.

TwidFactorInit

Description: `TwidFactorInit` generates the first half of the set of twiddle factors required by a Discrete Fourier Transform or Discrete Cosine Transform, and places the result in the complex destination vector. Effectively, the set contains the values:

$WN(k) = e^{-j\frac{2\pi k}{N}}$, where $0 \le k \le N/2$, for *conjFlag* = 0

$WN(k) = e^{j\frac{2\pi k}{N}}$, where $0 \le k \le N/2$, for *conjFlag* ! = 0

Include: `dsp.h`

Prototype:

```
extern fractcomplex* TwidFactorInit (
   int log2N,
   fractcomplex* twidFactors,
   int conjFlag
);
```

Arguments:

log2N	based 2 logarithm of N (number of complex factors needed by a DFT)
twidFactors	pointer to complex twiddle factors
conjFlag	flag to indicate whether or not conjugate values are to be generated

Return Value: Pointer to base address of twiddle factors.

Remarks: N *must* be an integer power of 2.
Only the first N/2 twiddle factors are generated.
The value of `conjFlag` determines the sign in the argument of the exponential function. For forward Fourier Transforms, *conjFlag* should be set to '`0`'. For inverse Fourier Transforms and Discrete Cosine Transforms, *conjFlag* should be set to '`1`'.
A complex vector of size N/2 must have already been allocated and assigned to *twidFactors* prior to invoking the function. The complex vector *should* be allocated in X-Data memory.
Factors computed in floating-point arithmetic and converted to 1.15 complex fractionals.

Source File: `inittwid.c`

TwidFactorInit (Continued)

Function Profile: System resources usage:

W0..W7	used, not restored
W8..W14	saved, used, restored

`DO` and `REPEAT` instruction usage:
None

Program words (24-bit instructions):
See the file “readme.txt” in pic30_tools\src\dsp for this information.

Cycles (including C-function call and return overheads):
See the file “readme.txt” in pic30_tools\src\dsp for this information.

Example: Please refer to the MPLAB C30 installation folder for a sample project demonstrating the use of this function.

2.8 CONTROL FUNCTIONS

This section describes functions provided in the DSP library that aid the implementation of closed-loop control systems.

2.8.1 Proportional Integral Derivative (PID) Control

A complete discussion of Proportional Integral Derivative (PID) controllers is beyond the scope of this discussion, but this section will try to provide you with some guidelines for tuning PID controllers.

2.8.1.1 PID CONTROLLER BACKGROUND

A PID controller responds to an error signal in a closed control loop and attempts to adjust the controlled quantity in order to achieve the desired system response. The controlled parameter can be any measurable system quantity, such as speed, voltage or current. The output of the PID controller can control one or more system parameters that will affect the controlled system quantity. For example, a speed control loop in a Sensorless Brushless DC motor application can control the PWM duty cycle directly or it can set the current demand for an inner control loop that regulates the motor currents. The benefit of the PID controller is that it can be adjusted empirically by adjusting one or more gain values and observing the change in system response.

A digital PID controller is executed at a periodic sampling interval and it is assumed that the controller is executed frequently enough so that the system can be properly controlled. For example, the current controller in the Sensorless Brushless DC motor application is executed every PWM cycle, since the motor can change very rapidly. The speed controller in such an application is executed at the medium event rate (100 Hz), because motor speed changes will occur relatively slowly due to mechanical time constants.

The error signal is formed by subtracting the desired setting of the parameter to be controlled from the actual measured value of that parameter. This sign of the error indicates the direction of change required by the control input.

The Proportional (P) term of the controller is formed by multiplying the error signal by a P gain. This will cause the PID controller to produce a control response that is a function of the error magnitude. As the error signal becomes larger, the P term of the controller becomes larger to provide more correction.

The effect of the P term will tend to reduce the overall error as time elapses. However, the effect of the P term will reduce as the error approaches zero. In most systems, the error of the controlled parameter will get very close to zero, but will not converge. The result is a small remaining steady state error. The Integral (I) term of the controller is used to fix small steady state errors. The I term takes a continuous running total of the error signal. Therefore, a small steady state error will accumulate into a large error value over time. This accumulated error signal is multiplied by an I gain factor and becomes the I output term of the PID controller.

The Differential (D) term of the PID controller is used to enhance the speed of the controller and responds to the rate of change of the error signal. The D term input is calculated by subtracting the present error value from a prior value. This delta error value is multiplied by a D gain factor that becomes the D output term of the PID controller. The D term of the controller produces more control output the faster the system error is changing.

It should be noted that not all PID controllers will implement the D or, less commonly, the I terms. For example, the speed controller in a Brushless DC motor application described by Microchip Application Note AN901 does not have a D term due to the rel-

atively slow response time of motor speed changes. In this case, the D term could cause excessive changes in PWM duty cycle that could affect the operation of the sensorless algorithm and produce over current trips.

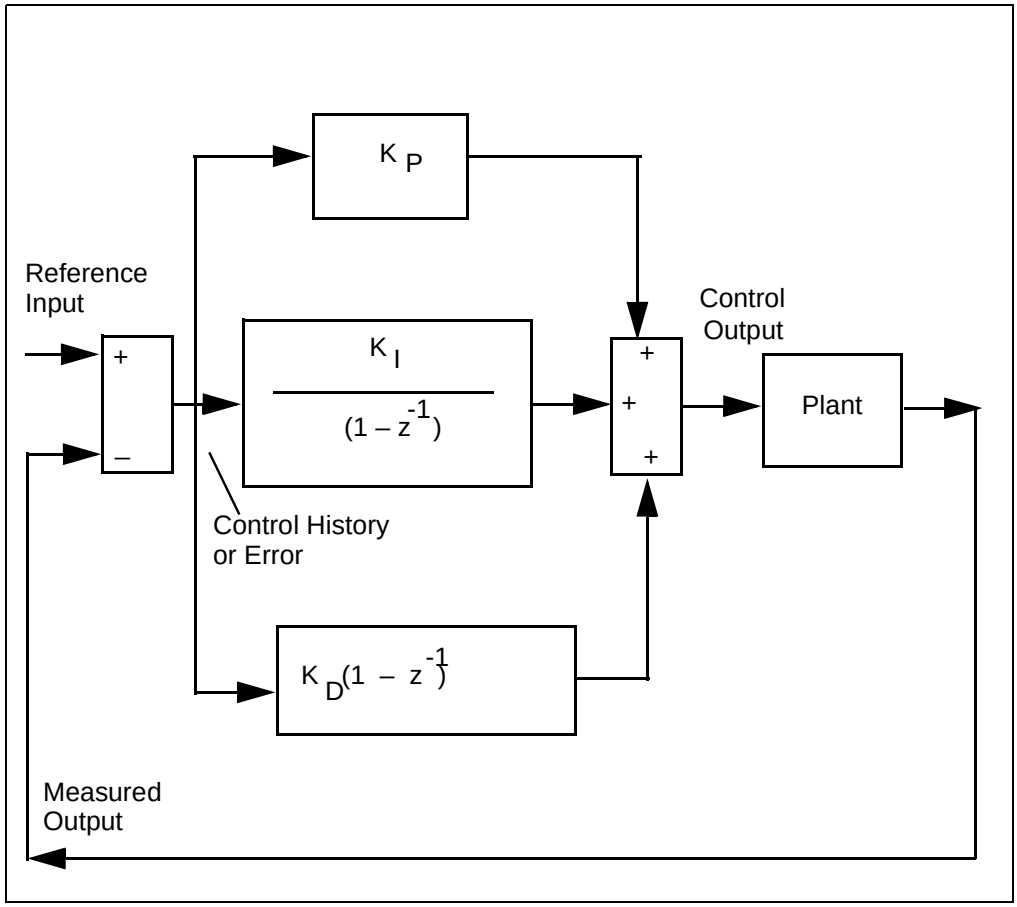
2.8.1.2 ADJUSTING PID GAINS

The P gain of a PID controller will set the overall system response. When first tuning a controller, the I and D gains should be set to zero. The P gain can then be increased until the system responds well to set-point changes without excessive overshoot or oscillations. Using lower values of P gain will 'loosely' control the system, while higher values will give 'tighter' control. At this point, the system will probably not converge to the set-point.

After a reasonable P gain is selected, the I gain can be slowly increased to force the system error to zero. Only a small amount of I gain is required in most systems. Note that the effect of the I gain, if large enough, can overcome the action of the P term, slow the overall control response, and cause the system to oscillate around the set-point. If this occurs, reducing the I gain and increasing the P gain will usually solve the problem.

After the P and I gains are set, the D gain can be set. The D term will speed up the response of control changes, but it should be used sparingly because it can cause very rapid changes in the controller output. This behavior is called 'set-point kick'. The set-point kick occurs because the difference in system error becomes instantaneously very large when the control set-point is changed. In some cases, damage to system hardware can occur. If the system response is acceptable with the D gain set to zero, you can probably omit the D term.

FIGURE 2-1: PID CONTROL SYSTEM



2.8.1.3 PID LIBRARY FUNCTIONS AND DATA STUCTURES

The DSP library provides a PID Controller function, `PID ( tPID* )`, to perform a PID operation. The function uses a data structure defined in the header file dsp.h, which has the following form:

```
typedef struct {
  fractional* abcCoefficients;
  fractional* controlHistory;
  fractional controlOutput;
  fractional measuredOutput;
  fractional controlReference;
} tPID;
```

Prior to invoking the `PID()` function, the application should initialize the data structure of type `tPID`. This is done in the following steps:

1. Calculate Coefficients from PID Gain values

   The element `abcCoefficients` in the data structure of type `tPID` is a pointer to A, B & C coefficients located in X-data space. These coefficients are derived from the PID gain values, Kp, Ki and Kd, shown in Figure 2-1, as follows:

   ```
   A = Kp + Ki + Kd
   B = -(Kp + 2*Kd)
   C = Kd
   ```

   To derive the A, B and C coefficients, the DSP library provides a function, `PIDCoeffCalc.`

2. Clear the PID State Variables

   The structural element `controlHistory` is a pointer to a history of 3 samples located in Y-space, with the first sample being the most recent (current). These samples constitute a history of current and past differences between the Reference Input and the Measured Output of the plant function. The PIDInit function clears the elements pointed to by `controlHistory`. It also clears the `controlOutput` element in the `tPID` data structure.

2.8.2 Individual Functions

PIDInit	
Description:	This routine clears the delay line elements in the 3-element array located in Y-space and pointed to by `controlHistory`. It also clears the current PID output element, `controlOutput`.
Include:	`dsp.h`
Prototype:	`void PIDInit ( tPID *`*`fooPIDStruct`*` );`
Arguments:	*`fooPIDStruct`* a pointer to a PID data structure of type tPID
Return Value:	void.
Source File:	`pid.s`

PIDInit (Continued)

Function Profile: System resources usage:

W0..W4	used, not restored
ACCA, ACCB	used, not restored
CORCON	saved, used, restored

`DO` and `REPEAT` instruction usage:
0 level `DO` instructions
0 `REPEAT` instructions

Program words (24-bit instructions):
11

Cycles (including C-function call and return overheads):
13

PIDCoeffCalc

Description: `PIDInit` computes the PID coefficients based on values of Kp, Ki and Kd provided by the user.
abcCoefficients[0] = Kp + Ki + Kd
abcCoefficients[1] = -(Kp + 2*Kd)
abcCoefficients[2] = Kd
This routine also clears the delay line elements in the array `ControlDifference`, as well as clears the current PID output element, `ControlOutput`.

Include: `dsp.h`

Prototype: `void PIDCoeffCalc ( fractional *fooPIDGainCoeff, tPID *fooPIDStruct )`

Arguments:

fooPIDGainCoeff	pointer to input array containing Kp, Ki, Kd coefficients in order [Kp, Ki, Kd]
fooPIDStruct	pointer to a PID data structure of type tPID

Return Value: Void.

Remarks: PID Coefficient array elements may be subject to saturation depending on values of Kp, Ki and Kd.

Source File: `pid.s`

Function Profile: System resources usage:

W0..W2	used, not restored
ACCA, ACCB	used, not restored
CORCON	saved, used, restored

`DO` and `REPEAT` instruction usage:
0 level `DO` instructions
0 `REPEAT` instructions

Program words (24-bit instructions):
18

Cycles (including C-function call and return overheads):
20

PID

Description: `PID` computes the `controlOutput` element of the data structure `tPID`:
controlOutput[n] = controlOutput[n-1]
+ controlHistory[n] * abcCoefficient[0]
+ controlHistory[n-1] * abcCoefficient[1]
+ controlHistory[n-2] * abcCoefficient[2]
where,
abcCoefficient[0] = Kp + Ki + Kd
abcCoefficient[1] = -(Kp + 2*Kd)
abcCoefficient[2] = Kd
ControlHistory[n] =
MeasuredOutput[n] - ReferenceInput[n]

Include: `dsp.h`

Prototype: `extern void PID ( tPID* fooPIDStruct );`

Arguments: *`fooPIDStruct`* pointer to a PID data structure of type tPID

Return Value: Pointer to *`fooPIDStruct`*

Remarks: `controlOutput` element is updated by the `PID()` routine. The `controlOutput` will be subject to saturation.

Source File: `pid.s`

Function Profile: System resources usage:

W0..W5	used, not restored
W8,W10	saved, used, restored
ACCA	used, not restored
CORCON	saved, used, restored

`DO` and `REPEAT` instruction usage:
0 level `DO` instructions
0 `REPEAT` instructions

Program words (24-bit instructions):
28

Cycles (including C-function call and return overheads):
30

2.9 MISCELLANEOUS FUNCTIONS

This section describes other helpful functions provided in the DSP library.

Fract2Float

Description: `Fract2Float` converts a 1.15 fractional value to an IEEE floating-point value.

Include: `dsp.h`

Prototype: `extern float Fract2Float (fractional aVal);`

Arguments: *aVal* 1.15 fractional number in the implicit range [-1,(+ 1 – 2^{-15})]

Return Value: IEEE floating-point value in the range [-1, (+ 1 – 2^{-15})]

Remarks: None

Source File: `flt2frct.c`

Function Profile: System resources usage:

W0..W7	used, not restored
W8..W14	saved, used, restored

`DO` and `REPEAT` instruction usage:
None

Program words (24-bit instructions):
See the file “readme.txt” in pic30_tools\src\dsp for this information.

Cycles (including C-function call and return overheads):
See the file “readme.txt” in pic30_tools\src\dsp for this information.

Float2Fract

Description: `Float2Fract` converts an IEEE floating-point value to a 1.15 fractional number.

Include: `dsp.h`

Prototype: `extern fractional Float2Fract (float aVal);`

Arguments: *aVal* Floating-point number in the range [-1,(+ 1 – 2^{-15})]

Return Value: 1.15 Fractional value in the range [-1, (+ 1 – 2^{-15})]

Remarks: The conversion is performed using convergent rounding and saturation mechanisms.

Source File: `flt2frct.c`

Function Profile: System resources usage:

W0..W7	used, not restored
W8..W14	saved, used, restored

`DO` and `REPEAT` instruction usage:
None

Program words (24-bit instructions):
See the file “readme.txt” in pic30_tools\src\dsp for this information.

Cycles (including C-function call and return overheads):
See the file “readme.txt” in pic30_tools\src\dsp for this information.

NOTES:

Chapter 3. Standard C Libraries with Math Functions

3.1 INTRODUCTION

Standard ANSI C library functions are contained in the libraries `libc-`*`omf`*`.a` and `libm-`*`omf`*`.a` (math functions), where *`omf`* will be `coff` or `elf` depending upon the selected object module format.

Additionally, some 16-bit standard C library helper functions, and standard functions that must be modified for use with 16-bit devices, are in the library `libpic30-`*`omf`*`.a`.

3.1.1 Assembly Code Applications

A free version of the math functions library and header file is available from the Microchip web site. No source code is available with this free version.

3.1.2 C Code Applications

The MPLAB C30 C compiler install directory (`c:\Program Files\Microchip\MPLAB C30`) contains the following subdirectories with library-related files:

- `lib` – standard C library files
- `src\libm` – source code for math library functions, batch file to rebuild the library
- `support\h` – header files for libraries

In addition, there is a file, `ResourceGraphs.pdf`, which contains diagrams of resources used by each function, located in `lib`.

3.1.3 Chapter Organization

This chapter is organized as follows:

- Using the Standard C Libraries

libc-*omf*.a

- <assert.h> diagnostics
- <ctype.h> character handling
- <errno.h> errors
- <float.h> floating-point characteristics
- <limits.h> implementation-defined limits
- <locale.h> localization
- <setjmp.h> non-local jumps
- <signal.h> signal handling
- <stdarg.h> variable argument lists
- <stddef.h> common definitions
- <stdio.h> input and output
- <stdlib.h> utility functions
- <string.h> string functions
- <time.h> date and time functions

libm-*omf*.a

- <math.h> mathematical functions

libpic30-*omf*.a

- pic30-libs

3.2 USING THE STANDARD C LIBRARIES

Building an application which utilizes the standard C libraries requires two types of files: header files and library files.

3.2.1 Header Files

All standard C library entities are declared or defined in one or more standard headers (See list in **Section 3.1.3 "Chapter Organization"**.) To make use of a library entity in a program, write an include directive that names the relevant standard header.

The contents of a standard header is included by naming it in an include directive, as in:

```
#include <stdio.h> /* include I/O facilities */
```

The standard headers can be included in any order. Do not include a standard header within a declaration. Do not define macros that have the same names as keywords before including a standard header.

A standard header never includes another standard header.

3.2.2 Library Files

The archived library files contain all the individual object files for each library function.

When linking an application, the library file must be provided as an input to the linker (using the `--library` or `-l` linker option) such that the functions used by the application may be linked into the application.

A typical C application will require three library files: `libc-`*`omf`*`.a`, `libm-`*`omf`*`.a`, and `libpic30-`*`omf`*`.a`. (See **Section 1.2 "OMF-Specific Libraries/Start-up Modules"** for more on OMF-specific libraries.) These libraries will be included automatically if linking is performed using the MPLAB C30 compiler.

Note:	Some standard library functions require a heap. These include the standard I/O functions that open files and the memory allocation functions. See the "*MPLAB® ASM30, MPLAB® LINK30 and Utilities User's Guide"* (DS51317) and "*MPLAB® C30 C Compiler User's Guide"* (DS51284) for more information on the heap.

3.3 <ASSERT.H> DIAGNOSTICS

The header file `assert.h` consists of a single macro that is useful for debugging logic errors in programs. By using the assert statement in critical locations where certain conditions should be true, the logic of the program may be tested.

Assertion testing may be turned off without removing the code by defining NDEBUG before including `<assert.h>`. If the macro `NDEBUG` is defined, `assert()` is ignored and no code is generated.

assert

Description: If the expression is false, an assertion message is printed to stderr and the program is aborted.

Include: `<assert.h>`

Prototype: `void assert(int expression);`

Argument: *expression* The expression to test.

Remarks: The expression evaluates to zero or non-zero. If zero, the assertion fails, and a message is printed to stderr. The message includes the source file name (`__FILE__`), the source line number (`__LINE__`), the expression being evaluated and the message. The macro then calls the function `abort()`. If the macro `_VERBOSE_DEBUGGING` is defined, a message will be printed to stderr each time `assert()` is called.

Example:

```
#include <assert.h> /* for assert */

int main(void)
{
  int a;

  a = 2 * 2;
  assert(a == 4); /* if true-nothing prints */
  assert(a == 6); /* if false-print message */
                  /* and abort */
}
```

Output:

```
sampassert.c:9 a == 6 -- assertion failed
ABRT
```

with `_VERBOSE_DEBUGGING` defined:

```
sampassert.c:8 a == 4 -- OK
sampassert.c:9 a == 6 -- assertion failed
ABRT
```

3.4 <CTYPE.H> CHARACTER HANDLING

The header file `ctype.h` consists of functions that are useful for classifying and mapping characters. Characters are interpreted according to the Standard C locale.

isalnum

Description: Test for an alphanumeric character.

Include: `<ctype.h>`

Prototype: `int isalnum(int c);`

Argument: *c* The character to test.

Return Value: Returns a non-zero integer value if the character is alphanumeric; otherwise, returns a zero.

Remarks: Alphanumeric characters are included within the ranges A-Z, a-z or 0-9.

Example:

```
#include <ctype.h> /* for isalnum */
#include <stdio.h> /* for printf */

int main(void)
{
  int ch;

  ch = '3';
  if (isalnum(ch))
    printf("3 is an alphanumeric\n");
  else
    printf("3 is NOT an alphanumeric\n");

  ch = '#';
  if (isalnum(ch))
    printf("# is an alphanumeric\n");
  else
    printf("# is NOT an alphanumeric\n");
}
```

Output:

```
3 is an alphanumeric
# is NOT an alphanumeric
```

isalpha

Description: Test for an alphabetic character.

Include: <ctype.h>

Prototype: `int isalpha(int c);`

Argument: *c* The character to test.

Return Value: Returns a non-zero integer value if the character is alphabetic; otherwise, returns zero.

Remarks: Alphabetic characters are included within the ranges A-Z or a-z.

isalpha (Continued)

Example:

```
#include <ctype.h> /* for isalpha */
#include <stdio.h> /* for printf */

int main(void)
{
  int ch;

  ch = 'B';
  if (isalpha(ch))
    printf("B is alphabetic\n");
  else
    printf("B is NOT alphabetic\n");

  ch = '#';
  if (isalpha(ch))
    printf("# is alphabetic\n");
  else
    printf("# is NOT alphabetic\n");
}
```

Output:

```
B is alphabetic
# is NOT alphabetic
```

iscntrl

Description: Test for a control character.

Include: <ctype.h>

Prototype: `int iscntrl(int c);`

Argument: *c* character to test.

Return Value: Returns a non-zero integer value if the character is a control character; otherwise, returns zero.

Remarks: A character is considered to be a control character if its ASCII value is in the range 0x00 to 0x1F inclusive, or 0x7F.

Example:

```
#include <ctype.h> /* for iscntrl */
#include <stdio.h> /* for printf */

int main(void)
{
  char ch;

  ch = 'B';
  if (iscntrl(ch))
    printf("B is a control character\n");
  else
    printf("B is NOT a control character\n");

  ch = '\t';
  if (iscntrl(ch))
    printf("A tab is a control character\n");
  else
    printf("A tab is NOT a control character\n");
}
```

Output:

```
B is NOT a control character
A tab is a control character
```

isdigit

Description: Test for a decimal digit.

Include: `<ctype.h>`

Prototype: `int isdigit(int c);`

Argument: *c* character to test.

Return Value: Returns a non-zero integer value if the character is a digit; otherwise, returns zero.

Remarks: A character is considered to be a digit character if it is in the range of '0'- '9'.

Example:

```
#include <ctype.h> /* for isdigit */
#include <stdio.h> /* for printf */

int main(void)
{
  int ch;

  ch = '3';
  if (isdigit(ch))
    printf("3 is a digit\n");
  else
    printf("3 is NOT a digit\n");

  ch = '#';
  if (isdigit(ch))
    printf("# is a digit\n");
  else
    printf("# is NOT a digit\n");
}
```

Output:

```
3 is a digit
# is NOT a digit
```

isgraph

Description: Test for a graphical character.

Include: `<ctype.h>`

Prototype: `int isgraph (int c);`

Argument: *c* character to test

Return Value: Returns a non-zero integer value if the character is a graphical character; otherwise, returns zero.

Remarks: A character is considered to be a graphical character if it is any printable character except a space.

Example:

```
#include <ctype.h> /* for isgraph */
#include <stdio.h> /* for printf */

int main(void)
{
  int ch;
```

isgraph (Continued)

```
  ch = '3';
  if (isgraph(ch))
    printf("3 is a graphical character\n");
  else
    printf("3 is NOT a graphical character\n");

  ch = '#';
  if (isgraph(ch))
    printf("# is a graphical character\n");
  else
    printf("# is NOT a graphical character\n");

  ch = ' ';
  if (isgraph(ch))
    printf("a space is a graphical character\n");
  else
   printf("a space is NOT a graphical character\n");
}
```

Output:

```
3 is a graphical character
# is a graphical character
a space is NOT a graphical character
```

islower

Description: Test for a lower case alphabetic character.

Include: <ctype.h>

Prototype: `int islower (int c);`

Argument: *c* character to test

Return Value: Returns a non-zero integer value if the character is a lower case alphabetic character; otherwise, returns zero.

Remarks: A character is considered to be a lower case alphabetic character if it is in the range of 'a'-'z'.

Example:

```
#include <ctype.h> /* for islower */
#include <stdio.h> /* for printf */

int main(void)
{
  int ch;

  ch = 'B';
  if (islower(ch))
    printf("B is lower case\n");
  else
    printf("B is NOT lower case\n");

  ch = 'b';
  if (islower(ch))
    printf("b is lower case\n");
  else
    printf("b is NOT lower case\n");
}
```

Output:

```
B is NOT lower case
b is lower case
```

isprint

Description: Test for a printable character (includes a space).

Include: `<ctype.h>`

Prototype: `int isprint (int c);`

Argument: *c* character to test

Return Value: Returns a non-zero integer value if the character is printable; otherwise, returns zero.

Remarks: A character is considered to be a printable character if it is in the range 0x20 to 0x7e inclusive.

Example:

```
#include <ctype.h> /* for isprint */
#include <stdio.h> /* for printf */

int main(void)
{

  int ch;

  ch = '&';
  if (isprint(ch))
    printf("& is a printable character\n");
  else
    printf("& is NOT a printable character\n");

  ch = '\t';
  if (isprint(ch))
    printf("a tab is a printable character\n");
  else
    printf("a tab is NOT a printable character\n");
}
```

Output:

```
& is a printable character
a tab is NOT a printable character
```

ispunct

Description: Test for a punctuation character.

Include: `<ctype.h>`

Prototype: `int ispunct (int c);`

Argument: *c* character to test

Return Value: Returns a non-zero integer value if the character is a punctuation character; otherwise, returns zero.

Remarks: A character is considered to be a punctuation character if it is a printable character which is neither a space nor an alphanumeric character. Punctuation characters consist of the following:

! " # $ % & ' ( ) ; < = > ? @ [\ ] * + , - . / : ^ _ { | } ~

ispunct (Continued)

Example:

```
#include <ctype.h> /* for ispunct */
#include <stdio.h> /* for printf */

int main(void)
{
  int ch;

  ch = '&';
  if (ispunct(ch))
    printf("& is a punctuation character\n");
  else
    printf("& is NOT a punctuation character\n");

  ch = '\t';
  if (ispunct(ch))
    printf("a tab is a punctuation character\n");
  else
    printf("a tab is NOT a punctuation character\n");
}
```

Output:

```
& is a punctuation character
a tab is NOT a punctuation character
```

isspace

Description: Test for a white-space character.

Include: <ctype.h>

Prototype: int isspace (int *c*);

Argument: *c* character to test

Return Value: Returns a non-zero integer value if the character is a white-space character; otherwise, returns zero.

Remarks: A character is considered to be a white-space character if it is one of the following: space (' '), form feed ('\f'), newline ('\n'), carriage return ('\r'), horizontal tab ('\t'), or vertical tab ('\v').

Example:

```
#include <ctype.h> /* for isspace */
#include <stdio.h> /* for printf */

int main(void)
{

  int ch;

  ch = '&';
  if (isspace(ch))
    printf("& is a white-space character\n");
  else
    printf("& is NOT a white-space character\n");

  ch = '\t';
  if (isspace(ch))
    printf("a tab is a white-space character\n");
  else
    printf("a tab is NOT a white-space character\n");
}
```

isspace (Continued)

Output:

```
& is NOT a white-space character
a tab is a white-space character
```

isupper

Description: Test for an upper case letter.

Include: <ctype.h>

Prototype: int isupper (int *c*);

Argument: *c* character to test

Return Value: Returns a non-zero integer value if the character is an upper case alphabetic character; otherwise, returns zero.

Remarks: A character is considered to be an upper case alphabetic character if it is in the range of ‘A’-‘Z’.

Example:

```
#include <ctype.h> /* for isupper */
#include <stdio.h> /* for printf */

int main(void)
{
  int ch;

  ch = 'B';
  if (isupper(ch))
    printf("B is upper case\n");
  else
    printf("B is NOT upper case\n");

  ch = 'b';
  if (isupper(ch))
    printf("b is upper case\n");
  else
    printf("b is NOT upper case\n");
}
```

Output:

```
B is upper case
b is NOT upper case
```

isxdigit

Description: Test for a hexadecimal digit.

Include: <ctype.h>

Prototype: int isxdigit (int *c*);

Argument: *c* character to test

Return Value: Returns a non-zero integer value if the character is a hexadecimal digit; otherwise, returns zero.

Remarks: A character is considered to be a hexadecimal digit character if it is in the range of ‘0’-‘9’, ‘A’-‘F’, or ‘a’-‘f’. Note: The list does not include the leading 0x because 0x is the prefix for a hexadecimal number but is not an actual hexadecimal digit.

isxdigit (Continued)

Example:

```
#include <ctype.h> /* for isxdigit */
#include <stdio.h> /* for printf */

int main(void)
{
  int ch;

  ch = 'B';
  if (isxdigit(ch))
    printf("B is a hexadecimal digit\n");
  else
    printf("B is NOT a hexadecimal digit\n");

  ch = 't';
  if (isxdigit(ch))
    printf("t is a hexadecimal digit\n");
  else
    printf("t is NOT a hexadecimal digit\n");
}
```

Output:

```
B is a hexadecimal digit
t is NOT a hexadecimal digit
```

tolower

Description: Convert a character to a lower case alphabetical character.

Include: <ctype.h>

Prototype: `int tolower (int c);`

Argument: *c* The character to convert to lower case.

Return Value: Returns the corresponding lower case alphabetical character if the argument was originally upper case; otherwise, returns the original character.

Remarks: Only upper case alphabetical characters may be converted to lower case.

Example:

```
#include <ctype.h> /* for tolower */
#include <stdio.h> /* for printf */

int main(void)
{
  int ch;

  ch = 'B';
  printf("B changes to lower case %c\n",
         tolower(ch));

  ch = 'b';
  printf("b remains lower case %c\n",
         tolower(ch));

  ch = '@';
  printf("@ has no lower case, ");
  printf("so %c is returned\n", tolower(ch));
}
```

tolower (Continued)

Output:

```
B changes to lower case b
b remains lower case b
@ has no lower case, so @ is returned
```

toupper

Description: Convert a character to an upper case alphabetical character.

Include: <ctype.h>

Prototype: int toupper (int *c*);

Argument: *c* The character to convert to upper case.

Return Value: Returns the corresponding upper case alphabetical character if the argument was originally lower case; otherwise, returns the original character.

Remarks: Only lower case alphabetical characters may be converted to upper case.

Example:

```
#include <ctype.h> /* for toupper */
#include <stdio.h> /* for printf */

int main(void)
{

  int ch;

  ch = 'b';
  printf("b changes to upper case %c\n",
          toupper(ch));

  ch = 'B';
  printf("B remains upper case %c\n",
          toupper(ch));

  ch = '@';
  printf("@ has no upper case, ");
  printf("so %c is returned\n", toupper(ch));
}
```

Output:

```
b changes to upper case B
B remains upper case B
@ has no upper case, so @ is returned
```

3.5 <ERRNO.H> ERRORS

The header file `errno.h` consists of macros that provide error codes that are reported by certain library functions (see individual functions). The variable `errno` may return any value greater than zero. To test if a library function encounters an error, the program should store the value zero in `errno` immediately before calling the library function. The value should be checked before another function call could change the value. At program start-up, `errno` is zero. Library functions will never set `errno` to zero.

EDOM

Description: Represents a domain error.

Include: `<errno.h>`

Remarks: `EDOM` represents a domain error, which occurs when an input argument is outside the domain in which the function is defined.

ERANGE

Description: Represents an overflow or underflow error.

Include: `<errno.h>`

Remarks: `ERANGE` represents an overflow or underflow error, which occurs when a result is too large or too small to be stored.

errno

Description: Contains the value of an error when an error occurs in a function.

Include: `<errno.h>`

Remarks: The variable `errno` is set to a non-zero integer value by a library function when an error occurs. At program start-up, `errno` is set to zero. `Errno` should be reset to zero prior to calling a function that sets it.

3.6 <FLOAT.H> FLOATING-POINT CHARACTERISTICS

The header file `float.h` consists of macros that specify various properties of floating-point types. These properties include number of significant figures, size limits, and what rounding mode is used.

DBL_DIG

Description:	Number of decimal digits of precision in a double precision floating-point value
Include:	`<float.h>`
Value:	6 by default, 15 if the switch `-fno-short-double` is used
Remarks:	By default, a double type is the same size as a float type (32-bit representation). The `-fno-short-double` switch allows the IEEE 64-bit representation to be used for a double precision floating-point value.

DBL_EPSILON

Description:	The difference between 1.0 and the next larger representable double precision floating-point value
Include:	`<float.h>`
Value:	1.192093e-07 by default, 2.220446e-16 if the switch `-fno-short-double` is used
Remarks:	By default, a double type is the same size as a float type (32-bit representation). The `-fno-short-double` switch allows the IEEE 64-bit representation to be used for a double precision floating-point value.

DBL_MANT_DIG

Description:	Number of base-`FLT_RADIX` digits in a double precision floating-point significand
Include:	`<float.h>`
Value:	24 by default, 53 if the switch `-fno-short-double` is used
Remarks:	By default, a double type is the same size as a float type (32-bit representation). The `-fno-short-double` switch allows the IEEE 64-bit representation to be used for a double precision floating-point value.

DBL_MAX

Description:	Maximum finite double precision floating-point value
Include:	`<float.h>`
Value:	3.402823e+38 by default, 1.797693e+308 if the switch `-fno-short-double` is used
Remarks:	By default, a double type is the same size as a float type (32-bit representation). The `-fno-short-double` switch allows the IEEE 64-bit representation to be used for a double precision floating-point value.

DBL_MAX_10_EXP

Description: Maximum integer value for a double precision floating-point exponent in base 10

Include: `<float.h>`

Value: 38 by default, 308 if the switch `-fno-short-double` is used

Remarks: By default, a double type is the same size as a float type (32-bit representation). The `-fno-short-double` switch allows the IEEE 64-bit representation to be used for a double precision floating-point value.

DBL_MAX_EXP

Description: Maximum integer value for a double precision floating-point exponent in base `FLT_RADIX`

Include: `<float.h>`

Value: 128 by default, 1024 if the switch `-fno-short-double` is used

Remarks: By default, a double type is the same size as a float type (32-bit representation). The `-fno-short-double` switch allows the IEEE 64-bit representation to be used for a double precision floating-point value.

DBL_MIN

Description: Minimum double precision floating-point value

Include: `<float.h>`

Value: 1.175494e-38 by default, 2.225074e-308 if the switch `-fno-short-double` is used

Remarks: By default, a double type is the same size as a float type (32-bit representation). The `-fno-short-double` switch allows the IEEE 64-bit representation to be used for a double precision floating-point value.

DBL_MIN_10_EXP

Description: Minimum negative integer value for a double precision floating-point exponent in base 10

Include: `<float.h>`

Value: -37 by default, -307 if the switch `-fno-short-double` is used

Remarks: By default, a double type is the same size as a float type (32-bit representation). The `-fno-short-double` switch allows the IEEE 64-bit representation to be used for a double precision floating-point value.

DBL_MIN_EXP

Description: Minimum negative integer value for a double precision floating-point exponent in base `FLT_RADIX`

Include: `<float.h>`

Value: -125 by default, -1021 if the switch `-fno-short-double` is used

Remarks: By default, a double type is the same size as a float type (32-bit representation). The `-fno-short-double` switch allows the IEEE 64-bit representation to be used for a double precision floating-point value.

FLT_DIG

Description: Number of decimal digits of precision in a single precision floating-point value

Include: `<float.h>`

Value: 6

FLT_EPSILON

Description: The difference between 1.0 and the next larger representable single precision floating-point value

Include: `<float.h>`

Value: 1.192093e-07

FLT_MANT_DIG

Description: Number of base-`FLT_RADIX` digits in a single precision floating-point significand

Include: `<float.h>`

Value: 24

FLT_MAX

Description: Maximum finite single precision floating-point value

Include: `<float.h>`

Value: 3.402823e+38

FLT_MAX_10_EXP

Description: Maximum integer value for a single precision floating-point exponent in base 10

Include: `<float.h>`

Value: 38

FLT_MAX_EXP

Description: Maximum integer value for a single precision floating-point exponent in base FLT_RADIX

Include: <float.h>

Value: 128

FLT_MIN

Description: Minimum single precision floating-point value

Include: <float.h>

Value: 1.175494e-38

FLT_MIN_10_EXP

Description: Minimum negative integer value for a single precision floating-point exponent in base 10

Include: <float.h>

Value: -37

FLT_MIN_EXP

Description: Minimum negative integer value for a single precision floating-point exponent in base FLT_RADIX

Include: <float.h>

Value: -125

FLT_RADIX

Description: Radix of exponent representation

Include: <float.h>

Value: 2

Remarks: The base representation of the exponent is base-2 or binary.

FLT_ROUNDS

Description: Represents the rounding mode for floating-point operations

Include: <float.h>

Value: 1

Remarks: Rounds to the nearest representable value

LDBL_DIG

Description: Number of decimal digits of precision in a long double precision floating-point value

Include: <float.h>

Value: 15

LDBL_EPSILON

Description: The difference between 1.0 and the next larger representable long double precision floating-point value

Include: `<float.h>`

Value: 2.220446e-16

LDBL_MANT_DIG

Description: Number of base-FLT_RADIX digits in a long double precision floating-point significand

Include: `<float.h>`

Value: 53

LDBL_MAX

Description: Maximum finite long double precision floating-point value

Include: `<float.h>`

Value: 1.797693e+308

LDBL_MAX_10_EXP

Description: Maximum integer value for a long double precision floating-point exponent in base 10

Include: `<float.h>`

Value: 308

LDBL_MAX_EXP

Description: Maximum integer value for a long double precision floating-point exponent in base `FLT_RADIX`

Include: `<float.h>`

Value: 1024

LDBL_MIN

Description: Minimum long double precision floating-point value

Include: `<float.h>`

Value: 2.225074e-308

LDBL_MIN_10_EXP

Description: Minimum negative integer value for a long double precision floating-point exponent in base 10

Include: `<float.h>`

Value: -307

LDBL_MIN_EXP

Description:	Minimum negative integer value for a long double precision floating-point exponent in base `FLT_RADIX`
Include:	`<float.h>`
Value:	-1021

3.7 <LIMITS.H> IMPLEMENTATION-DEFINED LIMITS

The header file `limits.h` consists of macros that define the minimum and maximum values of integer types. Each of these macros can be used in `#if` preprocessing directives.

CHAR_BIT

Description:	Number of bits to represent type `char`
Include:	`<limits.h>`
Value:	8

CHAR_MAX

Description:	Maximum value of a `char`
Include:	`<limits.h>`
Value:	127

CHAR_MIN

Description:	Minimum value of a `char`
Include:	`<limits.h>`
Value:	-128

INT_MAX

Description:	Maximum value of an `int`
Include:	`<limits.h>`
Value:	32767

INT_MIN

Description:	Minimum value of an `int`
Include:	`<limits.h>`
Value:	-32768

LLONG_MAX

Description:	Maximum value of a `long long int`
Include:	`<limits.h>`
Value:	9223372036854775807

LLONG_MIN

Description: Minimum value of a `long long int`
Include: `<limits.h>`
Value: -9223372036854775808

LONG_MAX

Description: Maximum value of a `long int`
Include: `<limits.h>`
Value: 2147483647

LONG_MIN

Description: Minimum value of a `long int`
Include: `<limits.h>`
Value: -2147483648

MB_LEN_MAX

Description: Maximum number of bytes in a multibyte character
Include: `<limits.h>`
Value: 1

SCHAR_MAX

Description: Maximum value of a `signed char`
Include: `<limits.h>`
Value: 127

SCHAR_MIN

Description: Minimum value of a `signed char`
Include: `<limits.h>`
Value: -128

SHRT_MAX

Description: Maximum value of a `short int`
Include: `<limits.h>`
Value: 32767

SHRT_MIN

Description: Minimum value of a `short int`
Include: `<limits.h>`
Value: -32768

UCHAR_MAX

Description: Maximum value of an `unsigned char`
Include: `<limits.h>`
Value: 255

UINT_MAX

Description: Maximum value of an `unsigned int`
Include: `<limits.h>`
Value: 65535

ULLONG_MAX

Description: Maximum value of a `long long unsigned int`
Include: `<limits.h>`
Value: 18446744073709551615

ULONG_MAX

Description: Maximum value of a `long unsigned int`
Include: `<limits.h>`
Value: 4294967295

USHRT_MAX

Description: Maximum value of an `unsigned short int`
Include: `<limits.h>`
Value: 65535

3.8 <LOCALE.H> LOCALIZATION

This compiler defaults to the C locale and does not support any other locales; therefore it does not support the header file `locale.h`. The following would normally be found in this file:

- struct lconv
- NULL
- LC_ALL
- LC_COLLATE
- LC_CTYPE
- LC_MONETARY
- LC_NUMERIC
- LC_TIME
- localeconv
- setlocale

3.9 <SETJMP.H> NON-LOCAL JUMPS

The header file `setjmp.h` consists of a type, a macro and a function that allow control transfers to occur that bypass the normal function call and return process.

jmp_buf

Description: A type that is an array used by `setjmp` and `longjmp` to save and restore the program environment.

Include: `<setjmp.h>`

Prototype:
```
typedef int jmp_buf[_NSETJMP];
```

Remarks: `_NSETJMP` is defined as 16 + 2 that represents 16 registers and a 32-bit return address.

setjmp

Description: A macro that saves the current state of the program for later use by `longjmp`.

Include: `<setjmp.h>`

Prototype:
```
#define setjmp(jmp_buf env)
```

Argument: *env* variable where environment is stored

Return Value: If the return is from a direct call, `setjmp` returns zero. If the return is from a call to `longjmp`, `setjmp` returns a non-zero value.
Note: If the argument *val* from `longjmp` is 0, `setjmp` returns 1.

Example: See `longjmp`.

longjmp

Description: A function that restores the environment saved by `setjmp`.

Include: `<setjmp.h>`

Prototype:
```
void longjmp(jmp_buf env, int val);
```

Arguments: *env* variable where environment is stored

val value to be returned to setjmp call.

Remarks: The value parameter *val* should be non-zero. If `longjmp` is invoked from a nested signal handler (that is, invoked as a result of a signal raised during the handling of another signal), the behavior is undefined.

3.10 <SIGNAL.H> SIGNAL HANDLING

The header file `signal.h` consists of a type, several macros and two functions that specify how the program handles signals while it is executing. A signal is a condition that may be reported during the program execution. Signals are synchronous, occurring under software control via the `raise` function.

A signal may be handled by:

- Default handling (`SIG_DFL`); the signal is treated as a fatal error and execution stops
- Ignoring the signal (`SIG_IGN`); the signal is ignored and control is returned to the user application
- Handling the signal with a function designated via `signal`.

By default all signals are handled by the default handler, which is identified by `SIG_DFL`.

The type `sig_atomic_t` is an integer type that the program access atomically. When this type is used with the keyword `volatile`, the signal handler can share the data objects with the rest of the program.

sig_atomic_t

Description:	A type used by a signal handler
Include:	`<signal.h>`
Prototype:	`typedef int sig_atomic_t;`

SIG_DFL

Description:	Used as the second argument and/or the return value for `signal` to specify that the default handler should be used for a specific signal.
Include:	`<signal.h>`

SIG_ERR

Description:	Used as the return value for `signal` when it cannot complete a request due to an error.
Include:	`<signal.h>`

SIG_IGN

Description:	Used as the second argument and/or the return value for `signal` to specify that the signal should be ignored.
Include:	`<signal.h>`

SIGABRT

Description: Name for the abnormal termination signal.

Include: `<signal.h>`

Prototype: `#define SIGABRT`

Remarks: `SIGABRT` represents an abnormal termination signal and is used in conjunction with `raise` or `signal`. The default `raise` behavior (action identified by `SIG_DFL`) is to output to the standard error stream:

```
abort - terminating
```

See the example accompanying `signal` to see general usage of signal names and signal handling.

Example:

```
#include <signal.h> /* for raise, SIGABRT */
#include <stdio.h> /* for printf */

int main(void)
{
  raise(SIGABRT);
  printf("Program never reaches here.");
}
```

Output:

```
ABRT
```

Explanation:
ABRT stands for “abort”.

SIGFPE

Description: Signals floating-point error such as for division by zero or result out of range.

Include: `<signal.h>`

Prototype: `#define SIGFPE`

Remarks: `SIGFPE` is used as an argument for `raise` and/or `signal`. When used, the default behavior is to print an arithmetic error message and terminate the calling program. This may be overridden by a user function that defines the signal handler actions. See `signal` for an example of a user defined function.

Example:

```
#include <signal.h> /* for raise, SIGFPE */
#include <stdio.h> /* for printf */

int main(void)
{
  raise(SIGFPE);
  printf("Program never reaches here");
}
```

Output:

```
FPE
```

Explanation:
FPE stands for “floating-point error”.

SIGILL

Description: Signals illegal instruction.

Include: `<signal.h>`

Prototype: `#define SIGILL`

Remarks: `SIGILL` is used as an argument for `raise` and/or `signal`. When used, the default behavior is to print an invalid executable code message and terminate the calling program. This may be overridden by a user function that defines the signal handler actions. See `signal` for an example of a user defined function.

Example:

```
#include <signal.h> /* for raise, SIGILL */
#include <stdio.h> /* for printf */

int main(void)
{
  raise(SIGILL);
  printf("Program never reaches here");
}
```

Output:

```
ILL
```

Explanation:
ILL stands for “illegal instruction”.

SIGINT

Description: Interrupt signal.

Include: `<signal.h>`

Prototype: `#define SIGINT`

Remarks: `SIGINT` is used as an argument for `raise` and/or `signal`. When used, the default behavior is to print an interruption message and terminate the calling program. This may be overridden by a user function that defines the signal handler actions. See `signal` for an example of a user defined function.

Example:

```
#include <signal.h> /* for raise, SIGINT */
#include <stdio.h> /* for printf */

int main(void)
{
  raise(SIGINT);
  printf("Program never reaches here.");
}
```

Output:

```
INT
```

Explanation:
INT stands for “interruption”.

SIGSEGV

Description: Signals invalid access to storage.

Include: `<signal.h>`

Prototype: `#define SIGSEGV`

Remarks: `SIGSEGV` is used as an argument for `raise` and/or `signal`. When used, the default behavior is to print an invalid storage request message and terminate the calling program. This may be overridden by a user function that defines the signal handler actions. See `signal` for an example of a user defined function.

Example:

```
#include <signal.h> /* for raise, SIGSEGV */
#include <stdio.h> /* for printf */

int main(void)
{
  raise(SIGSEGV);
  printf("Program never reaches here.");
}
```

Output:

```
SEGV
```

Explanation:
SEGV stands for “invalid storage access”.

SIGTERM

Description: Signals a termination request

Include: `<signal.h>`

Prototype: `#define SIGTERM`

Remarks: `SIGTERM` is used as an argument for `raise` and/or `signal`. When used, the default behavior is to print a termination request message and terminate the calling program. This may be overridden by a user function that defines the signal handler actions. See `signal` for an example of a user defined function.

Example:

```
#include <signal.h> /* for raise, SIGTERM */
#include <stdio.h> /* for printf */

int main(void)
{
  raise(SIGTERM);
  printf("Program never reaches here.");
}
```

Output:

```
TERM
```

Explanation:
TERM stands for “termination request”.

raise

Description: Reports a synchronous signal.

Include: `<signal.h>`

Prototype: `int raise(int sig);`

Argument: *sig* signal name

Return Value: Returns a 0 if successful; otherwise, returns a non-zero value.

Remarks: `raise` sends the signal identified by sig to the executing program.

Example:

```
#include <signal.h>   /* for raise, signal, */
                      /* SIGILL, SIG_DFL    */
#include <stdlib.h>   /* for div, div_t     */
#include <stdio.h>    /* for printf         */
#include <p30f6014.h> /* for INTCON1bits */

void __attribute__((__interrupt__))
_MathError(void)
{
  raise(SIGILL);
  INTCON1bits.MATHERR = 0;
}

void illegalinsn(int idsig)
{
  printf("Illegal instruction executed\n");
  exit(1);
}
int main(void)
{
  int x, y;
  div_t z;

  signal(SIGILL, illegalinsn);
  x = 7;
  y = 0;
  z = div(x, y);
  printf("Program never reaches here");
}
```

Output:

```
Illegal instruction executed
```

Explanation:

This example requires the linker script `p30f6014.gld`. There are three parts to this example.

First, an interrupt handler is written for the interrupt vector `_MathError` to handle a math error by sending an illegal instruction `signal (SIGILL)` to the executing program. The last statement in the interrupt handler clears the exception flag.

Second, the function `illegalinsn` will print an error message and call `exit`.

Third, in `main`, `signal (SIGILL, illegalinsn)` sets the handler for `SIGILL` to the function `illegalinsn`.

When a math error occurs, due to a divide by zero, the `_MathError` interrupt vector is called, which in turn will raise a signal that will call the handler function for `SIGILL`, which is the function `illegalinsn`. Thus error messages are printed and the program is terminated.

signal

Description: Controls interrupt signal handling.

Include: `<signal.h>`

Prototype: `void (*signal(int sig, void(*func)(int)))(int);`

Arguments: *sig* signal name

func function to be executed

Return Value: Returns the previous value of *func.*

Example:

```
#include <signal.h> /* for signal, raise, */
                    /* SIGINT, SIGILL, */
                    /* SIG_IGN, and SIGFPE */
#include <stdio.h>  /* for printf */

/* Signal handler function */
void mysigint(int id)
{
  printf("SIGINT received\n");
}

int main(void)
{
  /* Override default with user defined function */
  signal(SIGINT, mysigint);
  raise(SIGINT);

  /* Ignore signal handler */
  signal(SIGILL, SIG_IGN);
  raise(SIGILL);
  printf("SIGILL was ignored\n");

  /* Use default signal handler */
  raise(SIGFPE);
  printf("Program never reaches here.");
}
```

Output:

```
SIGINT received
SIGILL was ignored
FPE
```

Explanation:

The function `mysigint` is the user-defined signal handler for `SIGINT`. Inside the main program, the function `signal` is called to set up the signal handler (`mysigint`) for the signal `SIGINT` that will override the default actions. The function `raise` is called to report the signal `SIGINT`. This causes the signal handler for `SIGINT` to use the user-defined function (`mysigint`) as the signal handler so it prints the "`SIGINT received`" message.

Next, the function `signal` is called to set up the signal handler `SIG_IGN` for the signal `SIGILL`. The constant `SIG_IGN` is used to indicate the signal should be ignored. The function `raise` is called to report the signal `SIGILL` that is ignored.

The function `raise` is called again to report the signal `SIGFPE`. Since there is no user defined function for `SIGFPE`, the default signal handler is used so the message "`FPE`" is printed (which stands for "`arithmetic error - terminating`"). Then the calling program is terminated. The `printf` statement is never reached.

3.11 <STDARG.H> VARIABLE ARGUMENT LISTS

The header file `stdarg.h` supports functions with variable argument lists. This allows functions to have arguments without corresponding parameter declarations. There must be at least one named argument. The variable arguments are represented by ellipses (...). An object of type `va_list` must be declared inside the function to hold the arguments. `va_start` will initialize the variable to an argument list, `va_arg` will access the argument list, and `va_end` will end the use of the argument.

va_list

Description: The type `va_list` declares a variable that will refer to each argument in a variable-length argument list.

Include: `<stdarg.h>`

Example: See `va_arg`.

va_arg

Description: Gets the current argument

Include: `<stdarg.h>`

Prototype: `#define va_arg(va_list ap, Ty)`

Argument: *ap* pointer to list of arguments

Ty type of argument to be retrieved

Return Value: Returns the current argument

Remarks: `va_start` must be called before `va_arg`.

Example:

```
#include <stdio.h>   /* for printf */
#include <stdarg.h>  /* for va_arg, va_start,
                        va_list, va_end */

void tprint(const char *fmt, ...)
{
  va_list ap;

  va_start(ap, fmt);
  while (*fmt)
  {
    switch (*fmt)
    {
```

va_arg (Continued)

```
                case '%':
                      fmt++;
                      if (*fmt == 'd')
                      {
                        int d = va_arg(ap, int);
                        printf("<%d> is an integer\n",d);
                      }
                      else if (*fmt == 's')
                      {
                        char *s = va_arg(ap, char*);
                        printf("<%s> is a string\n", s);
                      }
                      else
                      {
                        printf("%%%c is an unknown format\n",
                             *fmt);
                      }
                      fmt++;
                      break;
                default:
                      printf("%c is unknown\n", *fmt);
                      fmt++;
                      break;
            }
        }
        va_end(ap);
}

int main(void)
{
  tprint("%d%s.%c", 83, "This is text.", 'a');
}
```

Output:

```
<83> is an integer
<This is text.> is a string
. is unknown
%c is an unknown format
```

va_end

Description: Ends the use of *ap*.

Include: <stdarg.h>

Prototype: #define va_end(va_list *ap*)

Argument: *ap* pointer to list of arguments

Remarks: After a call to va_end, the argument list pointer *ap* is considered to be invalid. Further calls to va_arg should not be made until the next va_start. In MPLAB C30, va_end does nothing, so this call is not necessary but should be used for readability and portability.

Example: See va_arg.

va_start

Description: Sets the argument pointer *ap* to first optional argument in the variable-length argument list

Include: <stdarg.h>

Prototype: #define va_start(va_list *ap*, *last_arg*)

Argument: *ap* pointer to list of arguments

last_arg last named argument before the optional arguments

Example: See va_arg.

3.12 <STDDEF.H> COMMON DEFINITIONS

The header file stddef.h consists of several types and macros that are of general use in programs.

ptrdiff_t

Description: The type of the result of subtracting two pointers.

Include: <stddef.h>

size_t

Description: The type of the result of the sizeof operator.

Include: <stddef.h>

wchar_t

Description: A type that holds a wide character value.

Include: <stddef.h>

NULL

Description: The value of a null pointer constant.

Include: <stddef.h>

offsetof

Description: Gives the offset of a structure member from the beginning of the structure.

Include: `<stddef.h>`

Prototype: `#define offsetof(T, mbr)`

Arguments:

T name of structure

mbr name of member in structure T

Return Value: Returns the offset in bytes of the specified member (*mbr*) from the beginning of the structure.

Remarks: The macro `offsetof` is undefined for bitfields. An error message will occur if bitfields are used.

Example:

```
#include <stddef.h> /* for offsetof */
#include <stdio.h>  /* for printf */

struct info {
  char item1[5];
  int item2;
  char item3;
  float item4;
};

int main(void)
{
  printf("Offset of item1 = %d\n",
         offsetof(struct info,item1));
  printf("Offset of item2 = %d\n",
         offsetof(struct info,item2));
  printf("Offset of item3 = %d\n",
         offsetof(struct info,item3));
  printf("Offset of item4 = %d\n",
         offsetof(struct info,item4));
}
```

Output:

```
Offset of item1 = 0
Offset of item2 = 6
Offset of item3 = 8
Offset of item4 = 10
```

Explanation:

This program shows the offset in bytes of each structure member from the start of the structure. Although `item1` is only 5 bytes (`char item1[5]`), padding is added so the address of `item2` falls on an even boundary. The same occurs with `item3`; it is 1 byte (`char item3`) with 1 byte of padding.

3.13 <STDIO.H> INPUT AND OUTPUT

The header file `stdio.h` consists of types, macros and functions that provide support to perform input and output operations on files and streams. When a file is opened it is associated with a stream. A stream is a pipeline for the flow of data into and out of files. Because different systems use different properties, the stream provides more uniform properties to allow reading and writing of the files.

Streams can be text streams or binary streams. Text streams consist of a sequence of characters divided into lines. Each line is terminated with a newline ('`\n`') character. The characters may be altered in their internal representation, particularly in regards to line endings. Binary streams consist of sequences of bytes of information. The bytes transmitted to the binary stream are not altered. There is no concept of lines - the file is just a series of bytes.

At start-up three streams are automatically opened: `stdin`, `stdout`, and `stderr`. `stdin` provides a stream for standard input, `stdout` is standard output and `stderr` is the standard error. Additional streams may be created with the `fopen` function. See `fopen` for the different types of file access that are permitted. These access types are used by `fopen` and `freopen`.

The type `FILE` is used to store information about each opened file stream. It includes such things as error indicators, end-of-file indicators, file position indicators, and other internal status information needed to control a stream. Many functions in the `stdio` use `FILE` as an argument.

There are three types of buffering: unbuffered, line buffered and fully buffered. Unbuffered means a character or byte is transferred one at a time. Line buffered collects and transfers an entire line at a time (i.e., the newline character indicates the end of a line). Fully buffered allows blocks of an arbitrary size to be transmitted. The functions `setbuf` and `setvbuf` control file buffering.

The `stdio.h` file also contains functions that use input and output formats. The input formats, or scan formats, are used for reading data. Their descriptions can be found under `scanf`, but they are also used by `fscanf` and `sscanf`. The output formats, or print formats, are used for writing data. Their descriptions can be found under `printf`. These print formats are also used by `fprintf`, `sprintf`, `vfprintf`, `vprintf` and `vsprintf`.

3.13.1 Compiler Options

Certain compiler options may affect how standard I/O performs. In an effort to provide a more tailored version of the formatted I/O routines, the tool chain may convert a call to a `printf` or `scanf` style function to a different call. The options are summarized below:

- The `-msmart-io` option, when enabled, will attempt to convert `printf`, `scanf` and other functions that use the input output formats to an integer only variant. The functionality is the same as that of the C standard forms, minus the support for floating-point output. `-msmart-io=0` disables this feature and no conversion will take place. `-msmart-io=1` or `-msmart-io` (the default) will convert a function call if it can be proven that an I/O function will never be presented with a floating-point conversion. `-msmart-io=2` is more optimistic than the default and will assume that non-constant format strings or otherwise unknown format strings will not contain a floating-point format. In the event that `-msmart-io=2` is used with a floating-point format, the format letter will appear as literal text and its corresponding argument will not be consumed.

- `-fno-short-double` will cause the compiler to generate calls to formatted I/O routines that support `double` as if it were a `long double` type.

Mixing modules compiled with these options may result in a larger executable size, or incorrect execution if large and small double-sized data is shared across modules.

3.13.2 Customizing STDIO

The standard I/O relies on helper functions described in **Section 3.18 “pic30-libs”**. These functions include `read()`, `write()`, `open()`, and `close()` which are called to read, write, open or close handles that are associated with standard I/O `FILE` pointers. The sources for these libraries are provided for you to customize as you wish.

The simplest way to redirect standard I/O to the pheripheral of your choice is to select one of the default handles already in use. Also, you could open files with a specific name, via `fopen()`, by rewriting `open()` to return a new handle to be recognized by `read()` or `write()`, as appropriate.

If only a specific peripheral is required, then you could associate handle `1 == stdout`, or `2 == stderr`, to another peripheral by writing the correct code to talk to the interested peripheral.

A complete generic solution might be:

```
/* should be in a header file */
enum my_handles {
    handle_stdin,
    handle_stdout,
    handle_stderr,
    handle_can1,
    handle_can2,
    handle_spi1,
    handle_spi2,
};

int __attribute__((__weak__, __section__(".libc"))) open(const char
*name, int access, int mode) {
     switch (name[0]) {
       case 'i' : return handle_stdin;
       case 'o' : return handle_stdout;
       case 'e' : return handle_stderr;
       case 'c' : return handle_can1;
       case 'C' : return handle_can2;
       case 's' : return handle_spi1;
       case 'S' : return handle_spi2;
       default: return handle_stderr;
    }
}
```

Single letters were used in this example because they are faster to check and use less memory. However, if memory is not an issue, you could use `strcmp` to compare full names.

In `write()`, you would write:

```
write(int handle, void *buffer, unsigned int len) {
     int i;
     volatile UxMODEBITS *umode = &U1MODEbits;
     volatile UxSTABITS *ustatus = &U1STAbits;
     volatile unsigned int *txreg = &U1TXREG;
     volatile unsigned int *brg = &U1BRG;

     switch (handle)
     {
```

```
        default:
        case 0:
        case 1:
        case 2:
           if ((__C30_UART != 1) && (&U2BRG)) {
              umode = &U2MODEbits;
              ustatus = &U2STAbits;
              txreg = &U2TXREG;
              brg = &U2BRG;
           }
           if ((umode->UARTEN) == 0)
           {
              *brg = 0;
              umode->UARTEN = 1;
           }
           if ((ustatus->UTXEN) == 0)
           {
              ustatus->UTXEN = 1;
           }
           for (i = len; i; --i)
           {
              while ((ustatus->TRMT) ==0);
              *txreg = *(char*)buffer++;
           }
           break;
        case handle_can1:  /* code to support can1 */
           break;
        case handle_can2:  /* code to support can2 */
           break;
        case handle_spi1:  /* code to support spi1 */
           break;
        case handle_spi2:  /* code to support spi2 */
           break;
        }
        return(len);
}
```

where you would fill in the appropriate code as specified in the comments.

Now you can use the generic C STDIO features to write to another port:

```
FILE *can1 = fopen("c","w");
     fprintf(can1,"This will be output through the can\n");
```

3.13.3 STDIO Functions

FILE	
Description:	Stores information for a file stream.
Include:	`<stdio.h>`

fpos_t	
Description:	Type of a variable used to store a file position.
Include:	`<stdio.h>`

size_t

Description: The result type of the `sizeof` operator.
Include: `<stdio.h>`

_IOFBF

Description: Indicates full buffering.
Include: `<stdio.h>`
Remarks: Used by the function `setvbuf`.

_IOLBF

Description: Indicates line buffering.
Include: `<stdio.h>`
Remarks: Used by the function `setvbuf`.

_IONBF

Description: Indicates no buffering.
Include: `<stdio.h>`
Remarks: Used by the function `setvbuf`.

BUFSIZ

Description: Defines the size of the buffer used by the function `setbuf`.
Include: `<stdio.h>`
Value: 512

EOF

Description: A negative number indicating the end-of-file has been reached or to report an error condition.
Include: `<stdio.h>`
Remarks: If an end-of-file is encountered, the end-of-file indicator is set. If an error condition is encountered, the error indicator is set. Error conditions include write errors and input or read errors.

FILENAME_MAX

Description: Maximum number of characters in a filename including the null terminator.
Include: `<stdio.h>`
Value: 260

FOPEN_MAX

Description: Defines the maximum number of files that can be simultaneously open

FOPEN_MAX (Continued)

Include: <stdio.h>

Value: 8

Remarks: stderr, stdin and stdout are included in the FOPEN_MAX count.

L_tmpnam

Description: Defines the number of characters for the longest temporary filename created by the function tmpnam.

Include: <stdio.h>

Value: 16

Remarks: L_tmpnam is used to define the size of the array used by tmpnam.

NULL

Description: The value of a null pointer constant

Include: <stdio.h>

SEEK_CUR

Description: Indicates that fseek should seek from the current position of the file pointer

Include: <stdio.h>

Example: See example for fseek.

SEEK_END

Description: Indicates that `fseek` should seek from the end of the file.

Include: `<stdio.h>`

Example: See example for `fseek`.

SEEK_SET

Description: Indicates that `fseek` should seek from the beginning of the file.

Include: `<stdio.h>`

Example: See example for `fseek`.

stderr

Description: File pointer to the standard error stream.

Include: `<stdio.h>`

stdin

Description: File pointer to the standard input stream.

Include: `<stdio.h>`

stdout

Description: File pointer to the standard output stream.

Include: `<stdio.h>`

TMP_MAX

Description: The maximum number of unique filenames the function `tmpnam` can generate.

Include: `<stdio.h>`

Value: 32

clearerr

Description: Resets the error indictor for the stream

Include: `<stdio.h>`

Prototype: `void clearerr(FILE *stream);`

Argument: *stream* stream to reset error indicators

Remarks: The function clears the end-of-file and error indicators for the given stream (i.e., `feof` and `ferror` will return false after the function `clearerr` is called).

Example:

```
/* This program tries to write to a file that is */
/* readonly. This causes the error indicator to  */
/* be set.  The function ferror is used to check */
/* the error indicator. The function clearerr is */
/* used to reset the error indicator so the next */
/* time ferror is called it will not report an   */
/* error.                                        */
#include <stdio.h> /* for ferror, clearerr,  */
                   /* printf, fprintf, fopen,*/
                   /* fclose, FILE, NULL     */

int main(void)
{
  FILE *myfile;

  if ((myfile = fopen("sampclearerr.c", "r")) ==
       NULL)
    printf("Cannot open file\n");
  else
  {
    fprintf(myfile, "Write this line to the "
            "file.\n");
    if (ferror(myfile))
      printf("Error\n");
    else
      printf("No error\n");
    clearerr(myfile);
    if (ferror(myfile))
      printf("Still has Error\n");
    else
      printf("Error indicator reset\n");

    fclose(myfile);
  }
}
```

Output:

```
Error
Error indicator reset
```

fclose

Description: Close a stream.

Include: `<stdio.h>`

Prototype: `int fclose(FILE *stream);`

Argument: *stream* pointer to the stream to close

Return Value: Returns 0 if successful; otherwise, returns EOF if any errors were detected.

Remarks: `fclose` writes any buffered output to the file.

Example:

```
#include <stdio.h> /* for fopen, fclose,     */
                   /* printf,FILE, NULL, EOF */

int main(void)
{
  FILE *myfile1, *myfile2;
  int y;

  if ((myfile1 = fopen("afile1", "w+")) == NULL)
    printf("Cannot open afile1\n");
  else
  {
    printf("afile1 was opened\n");

    y = fclose(myfile1);
    if (y == EOF)
      printf("afile1 was not closed\n");
    else
      printf("afile1 was closed\n");
  }
}
```

Output:

```
afile1 was opened
afile1 was closed
```

feof

Description: Tests for end-of-file

Include: `<stdio.h>`

Prototype: `int feof(FILE *stream);`

Argument: *stream* stream to check for end-of-file

Return Value: Returns non-zero if stream is at the end-of-file; otherwise, returns zero.

Example:

```
#include <stdio.h> /* for feof, fgetc, fputc, */
                   /* fopen, fclose, FILE,   */
                   /* NULL */

int main(void)
{
  FILE *myfile;
  int y = 0;

  if( (myfile = fopen( "afile.txt", "rb" )) == NULL )
    printf( "Cannot open file\n" );
  else
  {
    for (;;)
    {
      y = fgetc(myfile);
      if (feof(myfile))
        break;
      fputc(y, stdout);
    }
    fclose( myfile );
  }
}
```

Input:

Contents of `afile.txt` (used as input):

```
This is a sentence.
```

Output:

```
This is a sentence.
```

ferror

Description: Tests if error indicator is set.

Include: `<stdio.h>`

Prototype: `int ferror(FILE *stream);`

Argument: *stream* pointer to FILE structure

Return Value: Returns a non-zero value if error indicator is set; otherwise, returns a zero.

Example:

```
/* This program tries to write to a file that is */
/* readonly. This causes the error indicator to  */
/* be set.  The function ferror is used to check */
/* the error indicator and find the error. The   */
/* function clearerr is used to reset the error  */
/* indicator so the next time ferror is called   */
/* it will not report an error.                  */

#include <stdio.h> /* for ferror, clearerr, */
                   /* printf, fprintf,      */
                   /* fopen, fclose,        */
                   /* FILE, NULL            */


int main(void)
{
  FILE *myfile;

  if ((myfile = fopen("sampclearerr.c", "r")) ==
       NULL)
    printf("Cannot open file\n");
  else
  {
    fprintf(myfile, "Write this line to the "
            "file.\n");
    if (ferror(myfile))
      printf("Error\n");
    else
      printf("No error\n");
    clearerr(myfile);
    if (ferror(myfile))
      printf("Still has Error\n");
    else
      printf("Error indicator reset\n");

    fclose(myfile);
  }
}
```

Output:

```
Error
Error indicator reset
```

fflush

Description: Flushes the buffer in the specified stream.

Include: `<stdio.h>`

Prototype: `int fflush(FILE *stream);`

Argument: *stream* pointer to the stream to flush.

Return Value: Returns EOF if a write error occurs; otherwise, returns zero for success.

Remarks: If stream is a null pointer, all output buffers are written to files. `fflush` has no effect on an unbuffered stream.

fgetc

Description: Get a character from a stream

Include: `<stdio.h>`

Prototype: `int fgetc(FILE *stream);`

Argument: *stream* pointer to the open stream

Return Value: Returns the character read or EOF if a read error occurs or end-of-file is reached.

Remarks: The function reads the next character from the input stream, advances the file-position indicator and returns the character as an `unsigned char` converted to an `int`.

Example:

```
#include <stdio.h> /* for fgetc, printf, */
                   /* fclose, FILE,      */
                   /* NULL, EOF          */

int main(void)
{
  FILE *buf;
  char y;

  if ((buf = fopen("afile.txt", "r")) == NULL)
    printf("Cannot open afile.txt\n");
  else
  {
    y = fgetc(buf);
    while (y != EOF)
    {
      printf("%c|", y);
      y = fgetc(buf);
    }
    fclose(buf);
  }
}
```

Input:

Contents of `afile.txt` (used as input):

```
Short
Longer string
```

Output:

```
S|h|o|r|t|
|L|o|n|g|e|r| |s|t|r|i|n|g|
|
```

fgetpos

Description: Gets the stream's file position.

Include: `<stdio.h>`

Prototype: `int fgetpos(FILE *stream, fpos_t *pos);`

Arguments:

stream	target stream
pos	position-indicator storage

Return Value: Returns 0 if successful; otherwise, returns a non-zero value.

Remarks: The function stores the file-position indicator for the given stream in `*pos` if successful, otherwise, `fgetpos` sets `errno`.

Example:

```
/* This program opens a file and reads bytes at */
/* several different locations. The fgetpos     */
/* function notes the 8th byte. 21 bytes are    */
/* read then 18 bytes are read. Next the        */
/* fsetpos function is set based on the         */
/* fgetpos position and the previous 21 bytes   */
/* are reread.                                  */

#include <stdio.h> /* for fgetpos, fread,    */
                   /* printf, fopen, fclose, */
                   /* FILE, NULL, perror,    */
                   /* fpos_t, sizeof         */

int main(void)
{
  FILE   *myfile;
  fpos_t pos;
  char   buf[25];

  if ((myfile = fopen("sampfgetpos.c", "rb")) ==
       NULL)
    printf("Cannot open file\n");
  else
  {
   fread(buf, sizeof(char), 8, myfile);
    if (fgetpos(myfile, &pos) != 0)
      perror("fgetpos error");
    else
    {
      fread(buf, sizeof(char), 21, myfile);
      printf("Bytes read: %.21s\n", buf);
      fread(buf, sizeof(char), 18, myfile);
      printf("Bytes read: %.18s\n", buf);
    }

  if (fsetpos(myfile, &pos) != 0)
    perror("fsetpos error");

  fread(buf, sizeof(char), 21, myfile);
  printf("Bytes read: %.21s\n", buf);
  fclose(myfile);
  }
}
```

Output:

```
Bytes read: program opens a file
Bytes read: and reads bytes at
Bytes read: program opens a file
```

fgets

Description: Get a string from a stream

Include: `<stdio.h>`

Prototype: `char *fgets(char *s, int n, FILE *stream);`

Arguments:

s	pointer to the storage string
n	maximum number of characters to read
stream	pointer to the open stream.

Return Value: Returns a pointer to the string *s* if successful; otherwise, returns a null pointer

Remarks: The function reads characters from the input stream and stores them into the string pointed to by *s* until it has read n-1 characters, stores a newline character or sets the end-of-file or error indicators. If any characters were stored, a null character is stored immediately after the last read character in the next element of the array. If `fgets` sets the error indicator, the array contents are indeterminate.

Example:

```
#include <stdio.h> /* for fgets, printf, */
                   /* fopen, fclose,     */
                   /* FILE, NULL         */
#define MAX 50

int main(void)
{
  FILE *buf;
  char s[MAX];

  if ((buf = fopen("afile.txt", "r")) == NULL)
    printf("Cannot open afile.txt\n");
  else
  {
    while (fgets(s, MAX, buf) != NULL)
    {
      printf("%s|", s);
    }
    fclose(buf);
  }
}
```

Input:

Contents of `afile.txt` (used as input):

```
Short
Longer string
```

Output:

```
Short
|Longer string
|
```

fopen

Description: Opens a file.

Include: `<stdio.h>`

Prototype: `FILE *fopen(const char *filename, const char *mode);`

Arguments:

filename name of the file

mode type of access permitted

Return Value: Returns a pointer to the open stream. If the function fails a null pointer is returned.

Remarks: Following are the types of file access:

`r` - opens an existing text file for reading

`w` - opens an empty text file for writing. (An existing file will be overwritten.)

`a` - opens a text file for appending. (A file is created if it doesn't exist.)

`rb` - opens an existing binary file for reading.

`wb` - opens an empty binary file for writing. (An existing file will be overwritten.)

`ab` - opens a binary file for appending. (A file is created if it doesn't exist.)

`r+` - opens an existing text file for reading and writing.

`w+` - opens an empty text file for reading and writing. (An existing file will be overwritten.)

`a+` - opens a text file for reading and appending. (A file is created if it doesn't exist.)

`r+b` or `rb+` - opens an existing binary file for reading and writing.

`w+b` or `wb+` - opens an empty binary file for reading and writing. (An existing file will be overwritten.)

`a+b` or `ab+` - opens a binary file for reading and appending. (A file is created if it doesn't exist.)

Example:

```
#include <stdio.h> /* for fopen, fclose, */
                   /* printf, FILE,      */
                   /* NULL, EOF          */

int main(void)
{
  FILE *myfile1, *myfile2;
  int y;
```

fopen (Continued)

```
  if ((myfile1 = fopen("afile1", "r")) == NULL)
    printf("Cannot open afile1\n");
  else
  {
    printf("afile1 was opened\n");
    y = fclose(myfile1);
    if (y == EOF)
      printf("afile1 was not closed\n");
    else
      printf("afile1 was closed\n");
  }

  if ((myfile1 = fopen("afile1", "w+")) == NULL)
    printf("Second try, cannot open afile1\n");
  else
  {
    printf("Second try, afile1 was opened\n");
    y = fclose(myfile1);
    if (y == EOF)
      printf("afile1 was not closed\n");
    else
      printf("afile1 was closed\n");
  }

  if ((myfile2 = fopen("afile2", "w+")) == NULL)
    printf("Cannot open afile2\n");
  else
  {
    printf("afile2 was opened\n");
    y = fclose(myfile2);
    if (y == EOF)
      printf("afile2 was not closed\n");
   else
      printf("afile2 was closed\n");
  }

}
```

Output:

```
Cannot open afile1
Second try, afile1 was opened
afile1 was closed
afile2 was opened
afile2 was closed
```

Explanation:

`afile1` must exist before it can be opened for reading (`r`) or the `fopen` function will fail. If the `fopen` function opens a file for writing (`w+`) it does not have to already exist. If it doesn't exist, it will be created and then opened.

fprintf

Description: Prints formatted data to a stream.

Include: `<stdio.h>`

Prototype: `int fprintf(FILE *stream, const char *format, ...);`

Arguments:

stream	pointer to the stream in which to output data
format	format control string
...	optional arguments

Return Value: Returns number of characters generated or a negative number if an error occurs.

Remarks: The format argument has the same syntax and use that it has in `print`.

Example:

```
#include <stdio.h> /* for fopen, fclose, */
                   /* fprintf, printf,   */
                   /* FILE, NULL         */

int main(void)
{
  FILE *myfile;
  int y;
  char s[]="Print this string";
  int x = 1;
  char a = '\n';

  if ((myfile = fopen("afile", "w")) == NULL)
    printf("Cannot open afile\n");
  else
  {
    y = fprintf(myfile, "%s %d time%c", s, x, a);

    printf("Number of characters printed "
           "to file = %d",y);

    fclose(myfile);
  }
}
```

Output:

```
Number of characters printed to file = 25
```

Contents of `afile`:

```
Print this string 1 time
```

fputc

Description: Puts a character to the stream.

Include: `<stdio.h>`

Prototype: `int fputc(int c, FILE *stream);`

Arguments:

c	character to be written
stream	pointer to the open stream

Return Value: Returns the character written or EOF if a write error occurs.

Remarks: The function writes the character to the output stream, advances the file-position indicator and returns the character as an `unsigned char` converted to an `int`.

Example:

```
#include <stdio.h> /* for fputc, EOF, stdout */

int main(void)
{
  char *y;
  char buf[] = "This is text\n";
  int x;

  x = 0;

  for (y = buf; (x != EOF) && (*y != '\0'); y++)
  {
    x = fputc(*y, stdout);
    fputc('|', stdout);
  }
}
```

Output:

```
T|h|i|s| |i|s| |t|e|x|t|
|
```

fputs

Description: Puts a string to the stream.

Include: `<stdio.h>`

Prototype: `int fputs(const char *s, FILE *stream);`

Arguments:

s	string to be written
stream	pointer to the open stream

Return Value: Returns a non-negative value if successful; otherwise, returns EOF.

Remarks: The function writes characters to the output stream up to but not including the null character.

Example:

```
#include <stdio.h> /* for fputs, stdout */

int main(void)
{
  char buf[] = "This is text\n";

  fputs(buf,stdout);
  fputs("|",stdout);
}
```

Output:

```
This is text
|
```

fread

Description: Reads data from the stream.

Include: `<stdio.h>`

Prototype: `size_t fread(void *ptr, size_t size, size_t nelem, FILE *stream);`

Arguments:

ptr	pointer to the storage buffer
size	size of item
nelem	maximum number of items to be read
stream	pointer to the stream

Return Value: Returns the number of complete elements read up to *nelem* whose size is specified by size.

Remarks: The function reads characters from a given stream into the buffer pointed to by *ptr* until the function stores *size * nelem* characters or sets the end-of-file or error indicator. `fread` returns n/size where n is the number of characters it read. If n is not a multiple of size, the value of the last element is indeterminate. If the function sets the error indicator, the file-position indicator is indeterminate.

Example:

```
#include <stdio.h> /* for fread, fwrite,     */
                   /* printf, fopen, fclose, */
                   /* sizeof, FILE, NULL     */

int main(void)
{
  FILE *buf;
  int x, numwrote, numread;
  double nums[10], readnums[10];

  if ((buf = fopen("afile.out", "w+")) != NULL)
  {
    for (x = 0; x < 10; x++)
    {
      nums[x] = 10.0/(x + 1);
      printf("10.0/%d = %f\n", x+1, nums[x]);
    }

    numwrote = fwrite(nums, sizeof(double),
                      10, buf);
    printf("Wrote %d numbers\n\n", numwrote);
    fclose(buf);

 }
  else
    printf("Cannot open afile.out\n");
```

fread (Continued)

```
  if ((buf = fopen("afile.out", "r+")) != NULL)
  {
    numread = fread(readnums, sizeof(double),
                    10, buf);
    printf("Read %d numbers\n", numread);
    for (x = 0; x < 10; x++)
    {
      printf("%d * %f = %f\n", x+1, readnums[x],
             (x + 1) * readnums[x]);
    }
    fclose(buf);
  }
  else
    printf("Cannot open afile.out\n");
}
```

Output:

```
10.0/1 = 10.000000
10.0/2 = 5.000000
10.0/3 = 3.333333
10.0/4 = 2.500000
10.0/5 = 2.000000
10.0/6 = 1.666667
10.0/7 = 1.428571
10.0/8 = 1.250000
10.0/9 = 1.111111
10.0/10 = 1.000000
Wrote 10 numbers

Read 10 numbers
1 * 10.000000 = 10.000000
2 * 5.000000 = 10.000000
3 * 3.333333 = 10.000000
4 * 2.500000 = 10.000000
5 * 2.000000 = 10.000000
6 * 1.666667 = 10.000000
7 * 1.428571 = 10.000000
8 * 1.250000 = 10.000000
9 * 1.111111 = 10.000000
10 * 1.000000 = 10.000000
```

Explanation:

This program uses `fwrite` to save 10 numbers to a file in binary form. This allows the numbers to be saved in the same pattern of bits as the program is using which provides more accuracy and consistency. Using `fprintf` would save the numbers as text strings which could cause the numbers to be truncated. Each number is divided into 10 to produce a variety of numbers. Retrieving the numbers with `fread` to a new array and multiplying them by the original number shows the numbers were not truncated in the save process.

freopen

Description: Reassigns an existing stream to a new file.

Include: `<stdio.h>`

Prototype: `FILE *freopen(const char *filename, const char *mode, FILE *stream);`

Arguments:

filename	name of the new file
mode	type of access permitted
stream	pointer to the currently open stream

Return Value: Returns a pointer to the new open file. If the function fails a null pointer is returned.

Remarks: The function closes the file associated with the stream as though `fclose` was called. Then it opens the new file as though `fopen` was called. `freopen` will fail if the specified stream is not open. See `fopen` for the possible types of file access.

Example:

```
#include <stdio.h> /* for fopen, freopen, */
                   /* printf, fclose,     */
                   /* FILE, NULL          */

int main(void)
{
  FILE *myfile1, *myfile2;
  int y;

  if ((myfile1 = fopen("afile1", "w+")) == NULL)
    printf("Cannot open afile1\n");
  else
  {
    printf("afile1 was opened\n");

    if ((myfile2 = freopen("afile2", "w+",
         myfile1)) == NULL)
    {
      printf("Cannot open afile2\n");
      fclose(myfile1);
    }
    else
    {
      printf("afile2 was opened\n");
      fclose(myfile2);
    }
  }
}
```

Output:

```
afile1 was opened
afile2 was opened
```

Explanation:

This program uses `myfile2` to point to the stream when `freopen` is called so if an error occurs, `myfile1` will still point to the stream and can be closed properly. If the `freopen` call is successful, `myfile2` can be used to close the stream properly.

fscanf

Description: Scans formatted text from a stream.

Include: `<stdio.h>`

fscanf (Continued)

Prototype: `int fscanf(FILE *stream, const char *format, ...);`

Arguments:

stream	pointer to the open stream from which to read data
format	format control string
...	optional arguments

Return Value: Returns the number of items successfully converted and assigned. If no items are assigned, a 0 is returned. EOF is returned if end-of-file is encountered before the first conversion or if an error occurs.

Remarks: The format argument has the same syntax and use that it has in `scanf`.

Example:

```
#include <stdio.h> /* for fopen, fscanf,   */
                   /* fclose, fprintf,     */
                   /* fseek, printf, FILE, */
                   /* NULL, SEEK_SET       */

int main(void)
{
  FILE *myfile;
  char s[30];
  int x;
  char a;

  if ((myfile = fopen("afile", "w+")) == NULL)
    printf("Cannot open afile\n");
  else
  {
    fprintf(myfile, "%s %d times%c",
            "Print this string", 100, '\n');

    fseek(myfile, 0L, SEEK_SET);

    fscanf(myfile, "%s", s);
    printf("%s\n", s);
    fscanf(myfile, "%s", s);
    printf("%s\n", s);
    fscanf(myfile, "%s", s);
    printf("%s\n", s);
    fscanf(myfile, "%d", &x);
    printf("%d\n", x);
    fscanf(myfile, "%s", s);
    printf("%s\n", s);
    fscanf(myfile, "%c", a);
    printf("%c\n", a);

    fclose(myfile);
  }
}
```

Input:

Contents of `afile`:

```
Print this string 100 times
```

Output:

```
Print
this
string
100
times
```

fseek

Description: Moves file pointer to a specific location.

Include: `<stdio.h>`

Prototype: `int fseek(FILE *stream, long offset, int mode);`

Arguments:

stream	stream in which to move the file pointer.
offset	value to add to the current position
mode	type of seek to perform

Return Value: Returns 0 if successful; otherwise, returns a non-zero value and set errno.

Remarks: mode can be one of the following:

SEEK_SET – seeks from the beginning of the file

SEEK_CUR – seeks from the current position of the file pointer

SEEK_END – seeks from the end of the file

Example:

```
#include <stdio.h> /* for fseek, fgets,      */
                   /* printf, fopen, fclose, */
                   /* FILE, NULL, perror,    */
                   /* SEEK_SET, SEEK_CUR,    */
                   /* SEEK_END               */

int main(void)
{
  FILE *myfile;
  char s[70];
  int  y;

  myfile = fopen("afile.out", "w+");
  if (myfile == NULL)
    printf("Cannot open afile.out\n");
  else
  {
    fprintf(myfile, "This is the beginning, "
                    "this is the middle and "
                    "this is the end.");

    y = fseek(myfile, 0L, SEEK_SET);
    if (y)
      perror("Fseek failed");
    else
    {
      fgets(s, 22, myfile);
      printf("\"%s\"\n\n", s);
    }

    y = fseek(myfile, 2L, SEEK_CUR);
    if (y)
      perror("Fseek failed");
    else
    {
      fgets(s, 70, myfile);
      printf("\"%s\"\n\n", s);
    }
```

fseek (Continued)

```
    y = fseek(myfile, -16L, SEEK_END);
    if (y)
      perror("Fseek failed");
    else
    {
      fgets(s, 70, myfile);
      printf("\"%s\"\n", s);
    }
    fclose(myfile);
  }
}
```

Output:

```
"This is the beginning"

"this is the middle and this is the end."

"this is the end."
```

Explanation:

The file `afile.out` is created with the text, "This is the beginning, this is the middle and this is the end".

The function `fseek` uses an offset of zero and `SEEK_SET` to set the file pointer to the beginning of the file. `fgets` then reads 22 characters which are "This is the beginning", and adds a null character to the string.

Next, `fseek` uses an offset of two and `SEEK_CURRENT` to set the file pointer to the current position plus two (skipping the comma and space). `fgets` then reads up to the next 70 characters. The first 39 characters are "this is the middle and this is the end". It stops when it reads EOF and adds a null character to the string.

FInally, `fseek` uses an offset of negative 16 characters and `SEEK_END` to set the file pointer to 16 characters from the end of the file. `fgets` then reads up to 70 characters. It stops at the EOF after reading 16 characters "this is the end". and adds a null character to the string.

fsetpos

Description:	Sets the stream's file position.
Include:	`<stdio.h>`
Prototype:	`int fsetpos(FILE *stream, const fpos_t *pos);`
Arguments:	*stream* target stream *pos* position-indicator storage as returned by an earlier call to `fgetpos`
Return Value:	Returns 0 if successful; otherwise, returns a non-zero value.
Remarks:	The function sets the file-position indicator for the given stream in `*`*pos* if successful; otherwise, `fsetpos` sets *errno*.

fsetpos (Continued)

Example:

```
/* This program opens a file and reads bytes at */
/* several different locations. The fgetpos     */
/* function notes the 8th byte. 21 bytes are    */
/* read then 18 bytes are read. Next the        */
/* fsetpos function is set based on the         */
/* fgetpos position and the previous 21 bytes   */
/* are reread.                                  */

#include <stdio.h> /* for fgetpos, fread,    */
                   /* printf, fopen, fclose, */
                   /* FILE, NULL, perror,    */
                   /* fpos_t, sizeof         */

int main(void)
{
  FILE   *myfile;
  fpos_t pos;
  char   buf[25];

  if ((myfile = fopen("sampfgetpos.c", "rb")) ==
                      NULL)
    printf("Cannot open file\n");
  else
  {
    fread(buf, sizeof(char), 8, myfile);
    if (fgetpos(myfile, &pos) != 0)
      perror("fgetpos error");
    else
    {
      fread(buf, sizeof(char), 21, myfile);
      printf("Bytes read: %.21s\n", buf);
      fread(buf, sizeof(char), 18, myfile);
      printf("Bytes read: %.18s\n", buf);
    }

  if (fsetpos(myfile, &pos) != 0)
    perror("fsetpos error");

  fread(buf, sizeof(char), 21, myfile);
  printf("Bytes read: %.21s\n", buf);
  fclose(myfile);
  }
}
```

Output:

```
Bytes read: program opens a file
Bytes read: and reads bytes at
Bytes read: program opens a file
```

ftell

Description: Gets the current position of a file pointer.

Include: `<stdio.h>`

Prototype: `long ftell(FILE *stream);`

Argument: *stream* stream in which to get the current file position

Return Value: Returns the position of the file pointer if successful; otherwise, returns -1.

Example:

```
#include <stdio.h> /* for ftell, fread,      */
                   /* fprintf, printf,       */
                   /* fopen, fclose, sizeof, */
                   /* FILE, NULL */

int main(void)
{
  FILE *myfile;
  char s[75];
  long y;

  myfile = fopen("afile.out", "w+");
  if (myfile == NULL)
    printf("Cannot open afile.out\n");
  else
  {
    fprintf(myfile,"This is a very long sentence "
                    "for input into the file named "
                   "afile.out for testing.");

    fclose(myfile);

    if ((myfile = fopen("afile.out", "rb")) != NULL)
    {
      printf("Read some characters:\n");
      fread(s, sizeof(char), 29, myfile);
      printf("\t\"%s\"\n", s);

      y = ftell(myfile);
      printf("The current position of the "
             "file pointer is %ld\n", y);
      fclose(myfile);
    }
  }
}
```

Output:

```
Read some characters:
        "This is a very long sentence "
The current position of the file pointer is 29
```

fwrite

Description: Writes data to the stream.

Include: `<stdio.h>`

Prototype:
```
size_t fwrite(const void *ptr, size_t size,
 size_t nelem, FILE *stream);
```

Arguments:

ptr	pointer to the storage buffer
size	size of item
nelem	maximum number of items to be read
stream	pointer to the open stream

Return Value: Returns the number of complete elements successfully written, which will be less than *nelem* only if a write error is encountered.

Remarks: The function writes characters to a given stream from a buffer pointed to by *ptr* up to *nelem* elements whose size is specified by *size*. The file position indicator is advanced by the number of characters successfully written. If the function sets the error indicator, the file-position indicator is indeterminate.

Example:
```
#include <stdio.h> /* for fread, fwrite,     */
                   /* printf, fopen, fclose, */
                   /* sizeof, FILE, NULL     */

int main(void)
{
  FILE *buf;
  int x, numwrote, numread;
  double nums[10], readnums[10];

  if ((buf = fopen("afile.out", "w+")) != NULL)
  {
    for (x = 0; x < 10; x++)
    {
      nums[x] = 10.0/(x + 1);
      printf("10.0/%d = %f\n", x+1, nums[x]);
    }

    numwrote = fwrite(nums, sizeof(double),
                      10, buf);
    printf("Wrote %d numbers\n\n", numwrote);
    fclose(buf);

  }
  else
    printf("Cannot open afile.out\n");
```

fwrite (Continued)

```
  if ((buf = fopen("afile.out", "r+")) != NULL)
  {
    numread = fread(readnums, sizeof(double),
                    10, buf);
    printf("Read %d numbers\n", numread);
    for (x = 0; x < 10; x++)
    {
      printf("%d * %f = %f\n", x+1, readnums[x],
             (x + 1) * readnums[x]);
    }
    fclose(buf);
  }
  else
    printf("Cannot open afile.out\n");
}
```

Output:

```
10.0/1 = 10.000000
10.0/2 = 5.000000
10.0/3 = 3.333333
10.0/4 = 2.500000
10.0/5 = 2.000000
10.0/6 = 1.666667
10.0/7 = 1.428571
10.0/8 = 1.250000
10.0/9 = 1.111111
10.0/10 = 1.000000
Wrote 10 numbers

Read 10 numbers
1 * 10.000000 = 10.000000
2 * 5.000000 = 10.000000
3 * 3.333333 = 10.000000
4 * 2.500000 = 10.000000
5 * 2.000000 = 10.000000
6 * 1.666667 = 10.000000
7 * 1.428571 = 10.000000
8 * 1.250000 = 10.000000
9 * 1.111111 = 10.000000
10 * 1.000000 = 10.000000
```

Explanation:

This program uses `fwrite` to save 10 numbers to a file in binary form. This allows the numbers to be saved in the same pattern of bits as the program is using which provides more accuracy and consistency. Using `fprintf` would save the numbers as text strings, which could cause the numbers to be truncated. Each number is divided into 10 to produce a variety of numbers. Retrieving the numbers with `fread` to a new array and multiplying them by the original number shows the numbers were not truncated in the save process.

getc

Description: Get a character from the stream.

Include: `<stdio.h>`

Prototype: `int getc(FILE *stream);`

Argument: *stream* pointer to the open stream

Return Value: Returns the character read or EOF if a read error occurs or end-of-file is reached.

Remarks: `getc` is the same as the function `fgetc`.

Example:

```
#include <stdio.h> /* for getc, printf, */
                   /* fopen, fclose,    */
                   /* FILE, NULL, EOF   */

int main(void)
{
  FILE *buf;
  char y;

  if ((buf = fopen("afile.txt", "r")) == NULL)
    printf("Cannot open afile.txt\n");
  else
  {
    y = getc(buf);
    while (y != EOF)
    {
      printf("%c|", y);
      y = getc(buf);
    }
    fclose(buf);
  }
}
```

Input:

Contents of `afile.txt` (used as input):

```
Short
Longer string
```

Output:

```
S|h|o|r|t|
|L|o|n|g|e|r| |s|t|r|i|n|g|
|
```

getchar

Description: Get a character from `stdin`.

Include: `<stdio.h>`

Prototype: `int getchar(void);`

Return Value: Returns the character read or EOF if a read error occurs or end-of-file is reached.

Remarks: Same effect as `fgetc` with the argument `stdin`.

Example:

```
#include <stdio.h> /* for getchar, printf */

int main(void)
{
  char y;

    y = getchar();
    printf("%c|", y);
    y = getchar();
    printf("%c|", y);
    y = getchar();
    printf("%c|", y);
    y = getchar();
    printf("%c|", y);
    y = getchar();
    printf("%c|", y);
}
```

Input:

Contents of `UartIn.txt` (used as stdin input for simulator):

```
Short
Longer string
```

Output:

```
S|h|o|r|t|
```

gets

Description: Get a string from `stdin`.

Include: `<stdio.h>`

Prototype: `char *gets(char *s);`

Argument: *s* pointer to the storage string

Return Value: Returns a pointer to the string *s* if successful; otherwise, returns a null pointer

Remarks: The function reads characters from the stream `stdin` and stores them into the string pointed to by *s* until it reads a newline character (which is not stored) or sets the end-of-file or error indicators. If any characters were read, a null character is stored immediately after the last read character in the next element of the array. If `gets` sets the error indicator, the array contents are indeterminate.

gets (Continued)

Example:

```
#include <stdio.h> /* for gets, printf */

int main(void)
{
  char y[50];

  gets(y) ;
  printf("Text: %s\n", y);
}
```

Input:

Contents of `UartIn.txt` (used as stdin input for simulator):

```
Short
Longer string
```

Output:

```
Text: Short
```

perror

Description: Prints an error message to `stderr`.

Include: `<stdio.h>`

Prototype: `void perror(const char *s);`

Argument: *s* string to print

Return Value: None.

Remarks: The string s is printed followed by a colon and a space. Then an error message based on `errno` is printed followed by an newline

Example:

```
#include <stdio.h> /* for perror, fopen, */
                   /* fclose, printf,    */
                   /* FILE, NULL         */

int main(void)
{
  FILE *myfile;

  if ((myfile = fopen("samp.fil", "r+")) == NULL)
    perror("Cannot open samp.fil");
  else
    printf("Success opening samp.fil\n");

  fclose(myfile);
}
```

Output:

```
Cannot open samp.fil:  file open error
```

printf

Description: Prints formatted text to `stdout`.

Include: `<stdio.h>`

Prototype: `int printf(const char *format, ...);`

Arguments:

format	format control string
...	optional arguments

Return Value: Returns number of characters generated or a negative number if an error occurs.

Remarks: There must be exactly the same number of arguments as there are format specifiers. If the are less arguments than match the format specifiers, the output is undefined. If there are more arguments than match the format specifiers, the remaining arguments are discarded. Each format specifier begins with a percent sign followed by optional fields and a required type as shown here:

```
%[flags][width][.precision][size]type
```

`flags`

-	left justify the value within a given field width
0	Use 0 for the pad character instead of space (which is the default)
+	generate a plus sign for positive signed values
space	generate a space or signed values that have neither a plus nor a minus sign
#	to prefix 0 on an octal conversion, to prefix 0x or 0X on a hexadecimal conversion, or to generate a decimal point and fraction digits that are otherwise suppressed on a floating-point conversion

`width`

specify the number of characters to generate for the conversion. If the asterisk (*) is used instead of a decimal number, the next argument (which must be of type `int`) will be used for the field width. If the result is less than the field width, pad characters will be used on the left to fill the field. If the result is greater than the field width, the field is expanded to accommodate the value without padding.

`precision`

The field width can be followed with dot (.) and a decimal integer representing the precision that specifies one of the following:

- minimum number of digits to generate on an integer conversion
- number of fraction digits to generate on an e, E, or f conversion
- maximum number of significant digits to generate on a g or G conversion
- maximum number of characters to generate from a C string on an s conversion

If the period appears without the integer the integer is assumed to be zero. If the asterisk (*) is used instead of a decimal number, the next argument (which must be of type int) will be used for the precision.

printf (Continued)

`size`	
h modifier –	used with type d, i, o, u, x, X; converts the value to a `short int` or `unsigned short int`
h modifier –	used with n; specifies that the pointer points to a `short int`
l modifier –	used with type d, i, o, u, x, X; converts the value to a `long int` or `unsigned long int`
l modifier –	used with n; specifies that the pointer points to a `long int`
l modifier –	used with c; specifies a wide character
l modifier –	used with type e, E, f, F, g, G; converts the value to a `double`
ll modifier –	used with type d, i, o, u, x, X; converts the value to a `long long int` or `unsigned long long int`
ll modifier –	used with n; specifies that the pointer points to a `long long int`
L modifier –	used with e, E, f, g, G; converts the value to a `long double`
`type`	
d, i	`signed int`
o	`unsigned int` in octal
u	`unsigned int` in decimal
x	`unsigned int` in lowercase hexadecimal
X	`unsigned int` in uppercase hexadecimal
e, E	`double` in scientific notation
f	`double` decimal notation
g, G	`double` (takes the form of e, E or f as appropriate)
c	`char` - a single character
s	`string`
p	value of a pointer
n	the associated argument shall be an integer pointer into which is placed the number of characters written so far. No characters are printed.
%	A % character is printed

Example:

```
#include <stdio.h> /* for printf */

int main(void)
{
  /* print a character right justified in a 3  */
  /* character space.                          */
  printf("%3c\n", 'a');

  /* print an integer, left justified (as      */
  /* specified by the minus sign in the format */
  /* string) in a 4 character space. Print a   */
  /* second integer that is right justified in */
  /* a 4 character space using the pipe (|) as */
  /* a separator between the integers.         */
  printf("%-4d|%4d\n", -4, 4);

  /* print a number converted to octal in 4    */
  /* digits.                                   */
  printf("%.4o\n", 10);
```

printf (Continued)

```
  /* print a number converted to hexadecimal   */
  /* format with a 0x prefix.                  */
  printf("%#x\n", 28);

  /* print a float in scientific notation      */
  printf("%E\n", 1.1e20);

  /* print a float with 2 fraction digits      */
  printf("%.2f\n", -3.346);

  /* print a long float with %E, %e, or %f     */
  /* whichever is the shortest version         */
  printf("%Lg\n", .02L);
}
```

Output:

```
  a
-4  |   4
0012
0x1c
1.100000E+20
-3.35
0.02
```

putc

Description: Puts a character to the stream.

Include: `<stdio.h>`

Prototype: `int putc(int c, FILE *stream);`

Arguments:

c	character to be written
stream	pointer to FILE structure

Return Value: Returns the character or EOF if an error occurs or end-of-file is reached.

Remarks: `putc` is the same as the function `fputc`.

Example:

```
#include <stdio.h> /* for putc, EOF, stdout */

int main(void)
{

  char *y;
  char buf[] = "This is text\n";
  int x;

  x = 0;

for (y = buf; (x != EOF) && (*y != '\0'); y++)
  {
    x = putc(*y, stdout);
    putc('|', stdout);
  }
}
```

Output:

```
T|h|i|s| |i|s| |t|e|x|t|
|
```

putchar

Description: Put a character to `stdout`.

Include: `<stdio.h>`

Prototype: `int putchar(int c);`

Argument: *c* character to be written

Return Value: Returns the character or EOF if an error occurs or end-of-file is reached.

Remarks: Same effect as `fputc` with `stdout` as an argument.

Example:

```
#include <stdio.h> /* for putchar, printf, */
                   /* EOF, stdout          */

int main(void)
{

  char *y;
  char buf[] = "This is text\n";
  int x;

  x = 0;


  for (y = buf; (x != EOF) && (*y != '\0'); y++)
    x = putchar(*y);
}
```

Output:

```
This is text
```

puts

Description: Put a string to `stdout`.

Include: `<stdio.h>`

Prototype: `int puts(const char *s);`

Argument: *s* string to be written

Return Value: Returns a non-negative value if successful; otherwise, returns EOF.

Remarks: The function writes characters to the stream `stdout`. A newline character is appended. The terminating null character is not written to the stream.

Example:

```
#include <stdio.h> /* for puts */

int main(void)
{
  char buf[] = "This is text\n";

  puts(buf);
  puts("|");
}
```

Output:

```
This is text

|
```

remove

Description: Deletes the specified file.

Include: `<stdio.h>`

Prototype: `int remove(const char *filename);`

Argument: *filename* name of file to be deleted.

Return Value: Returns 0 if successful, -1 if not.

Remarks: If filename does not exist or is open, `remove` will fail.

Example:

```
#include <stdio.h> /* for remove, printf */

int main(void)
{
  if (remove("myfile.txt") != 0)
    printf("Cannot remove file");
  else
    printf("File removed");
}
```

Output:

```
File removed
```

rename

Description: Renames the specified file.

Include: `<stdio.h>`

Prototype: `int rename(const char *old, const char *new);`

Arguments: *old* pointer to the old name

new pointer to the new name.

Return Value: Return 0 if successful, non-zero if not.

Remarks: The new name must not already exist in the current working directory, the old name must exist in the current working directory.

Example:

```
#include <stdio.h> /* for rename, printf */

int main(void)
{
  if (rename("myfile.txt","newfile.txt") != 0)
    printf("Cannot rename file");
  else
    printf("File renamed");
}
```

Output:

```
File renamed
```

rewind

Description: Resets the file pointer to the beginning of the file.

Include: `<stdio.h>`

Prototype: `void rewind(FILE *stream);`

Argument: *stream* stream to reset the file pointer

Remarks: The function calls `fseek(stream, 0L, SEEK_SET)` and then clears the error indicator for the given stream.

Example:

```
#include <stdio.h> /* for rewind, fopen, */
                   /* fscanf, fclose,    */
                   /* fprintf, printf,   */
                   /* FILE, NULL         */

int main(void)
{
  FILE *myfile;
  char s[] = "cookies";
  int x = 10;

  if ((myfile = fopen("afile", "w+")) == NULL)
    printf("Cannot open afile\n");
  else
  {
    fprintf(myfile, "%d %s", x, s);
    printf("I have %d %s.\n", x, s);

    /* set pointer to beginning of file */
    rewind(myfile);
    fscanf(myfile, "%d %s", &x, &s);
    printf("I ate %d %s.\n", x, s);

    fclose(myfile);
  }
}
```

Output:

```
I have 10 cookies.
I ate 10 cookies.
```

scanf

Description: Scans formatted text from `stdin`.

Include: `<stdio.h>`

Prototype: `int scanf(const char *format, ...);`

Argument:

format	format control string
...	optional arguments

Return Value: Returns the number of items successfully converted and assigned. If no items are assigned, a 0 is returned. EOF is returned if an input failure is encountered before the first.

Remarks: Each format specifier begins with a percent sign followed by optional fields and a required type as shown here:

```
%[*][width][modifier]type
```

`*`
indicates assignment suppression. This will cause the input field to be skipped and no assignment made.

`width`
specify the maximum number of input characters to match for the conversion not including white space that can be skipped.

`modifier`

h modifier – used with type d, i, o, u, x, X; converts the value to a `short int` or `unsigned short int`.

h modifier – used with n; specifies that the pointer points to a `short int`

l modifier – used with type d, i, o, u, x, X; converts the value to a `long int` or `unsigned long int`

l modifier – used with n; specifies that the pointer points to a `long int`

l modifier – used with c; specifies a wide character

l modifier – used with type e, E, f, F, g, G; converts the value to a `double`

ll modifier – used with type d, i, o, u, x, X; converts the value to a `long long int` or `unsigned long long int`

ll modifier – used with n; specifies that the pointer points to a `long long int`

L modifier – used with e, E, f, g, G; converts the value to a `long double`

scanf (Continued)

type	
d,i	`signed int`
o	`unsigned int` in octal
u	`unsigned int` in decimal
x	`unsigned int` in lowercase hexadecimal
X	`unsigned int` in uppercase hexadecimal
e,E	`double` in scientific notation
f	`double` decimal notation
g,G	`double` (takes the form of e, E or f as appropriate)
c	`char` - a single character
s	string
p	value of a pointer
n	the associated argument shall be an integer pointer into, which is placed the number of characters read so far. No characters are scanned.
[...]	character array. Allows a search of a set of characters. A caret (^) immediately after the left bracket ([ ) inverts the scanset and allows any ASCII character except those specified between the brackets. A dash character (-) may be used to specify a range beginning with the character before the dash and ending the character after the dash. A null character can not be part of the scanset.
%	A % character is scanned

Example:

```
#include <stdio.h> /* for scanf, printf */

int main(void)
{
  int number, items;
  char letter;
  char color[30], string[30];
  float salary;

  printf("Enter your favorite number, "
         "favorite letter, ");
  printf("favorite color desired salary "
         "and SSN:\n");
  items = scanf("%d %c %[A-Za-z] %f %s", &number,
  &letter, &color, &salary, &string);

  printf("Number of items scanned = %d\n", items);
  printf("Favorite number = %d, ", number);
  printf("Favorite letter = %c\n", letter);
  printf("Favorite color = %s, ", color);
  printf("Desired salary = $%.2f\n", salary);
  printf("Social Security Number = %s, ", string);
}
```

Input:

Contents of `UartIn.txt` (used as stdin input for simulator):

```
5 T Green 300000 123-45-6789
```

Output:

```
Enter your favorite number, favorite letter,
favorite color, desired salary and SSN:
Number of items scanned = 5
Favorite number = 5, Favorite letter = T
Favorite color = Green, Desired salary = $300000.00
Social Security Number = 123-45-6789
```

setbuf

Description: Defines how a stream is buffered.

Include: `<stdio.h>`

Prototype: `void setbuf(FILE *stream, char *buf);`

Arguments:

stream	pointer to the open stream
buf	user allocated buffer

Remarks: `setbuf` must be called after `fopen` but before any other function calls that operate on the stream. If *buf* is a null pointer, `setbuf` calls the function `setvbuf(stream, 0, _IONBF, BUFSIZ)` for no buffering; otherwise `setbuf` calls `setvbuf(stream, buf, _IOFBF, BUFSIZ)` for full buffering with a buffer of size `BUFSIZ`. See `setvbuf`.

Example:

```
#include <stdio.h> /* for setbuf, printf, */
                   /* fopen, fclose,      */
                   /* FILE, NULL, BUFSIZ  */

int main(void)
{
  FILE *myfile1, *myfile2;
  char buf[BUFSIZ];

  if ((myfile1 = fopen("afile1", "w+")) != NULL)
  {
    setbuf(myfile1, NULL);
    printf("myfile1 has no buffering\n");
    fclose(myfile1);
  }

  if ((myfile2 = fopen("afile2", "w+")) != NULL)
  {
    setbuf(myfile2, buf);
    printf("myfile2 has full buffering");
    fclose(myfile2);
  }

}
```

Output:

```
myfile1 has no buffering
myfile2 has full buffering
```

setvbuf

Description: Defines the stream to be buffered and the buffer size.

Include: `<stdio.h>`

Prototype:
```
int setvbuf(FILE *stream, char *buf, int mode,
size_t size);
```

Arguments:

stream	pointer to the open stream
buf	user allocated buffer
mode	type of buffering
size	size of buffer

Return Value: Returns 0 if successful

Remarks: `setvbuf` must be called after `fopen` but before any other function calls that operate on the stream. For mode use one of the following:
`_IOFBF` – for full buffering
`_IOLBF` – for line buffering
`_IONBF` – for no buffering

Example:
```
#include <stdio.h> /* for setvbuf, fopen, */
                   /* printf, FILE, NULL, */
                   /* _IONBF, _IOFBF      */

int main(void)
{
  FILE *myfile1, *myfile2;
  char buf[256];

  if ((myfile1 = fopen("afile1", "w+")) != NULL)
  {
    if (setvbuf(myfile1, NULL, _IONBF, 0) == 0)
      printf("myfile1 has no buffering\n");
    else
      printf("Unable to define buffer stream "
             "and/or size\n");
  }
  fclose(myfile1);

  if ((myfile2 = fopen("afile2", "w+")) != NULL)
  {
    if (setvbuf(myfile2, buf, _IOFBF, sizeof(buf)) ==
        0)
      printf("myfile2 has a buffer of %d "
             "characters\n", sizeof(buf));
    else
      printf("Unable to define buffer stream "
             "and/or size\n");
  }
  fclose(myfile2);
}
```

Output:
```
myfile1 has no buffering
myfile2 has a buffer of 256 characters
```

sprintf

Description:	Prints formatted text to a string
Include:	`<stdio.h>`
Prototype:	`int sprintf(char *s, const char *format, ...);`
Arguments:	*s* storage string for output
	format format control string
	... optional arguments
Return Value:	Returns the number of characters stored in *s* excluding the terminating null character.
Remarks:	The format argument has the same syntax and use that it has in `printf`.
Example:	

```
#include <stdio.h> /* for sprintf, printf */

int main(void)
{
  char sbuf[100], s[]="Print this string";
  int x = 1, y;
  char a = '\n';

  y = sprintf(sbuf, "%s %d time%c", s, x, a);

  printf("Number of characters printed to "
         "string buffer = %d\n", y);
  printf("String = %s\n", sbuf);
}
```

Output:

```
Number of characters printed to string buffer = 25
String = Print this string 1 time
```

sscanf

Description:	Scans formatted text from a string
Include:	`<stdio.h>`
Prototype:	`int sscanf(const char *s, const char *format, ...);`
Arguments:	*s* storage string for input
	format format control string
	... optional arguments
Return Value:	Returns the number of items successfully converted and assigned. If no items are assigned, a 0 is returned. EOF is returned if an input error is encountered before the first conversion.
Remarks:	The format argument has the same syntax and use that it has in `scanf`.

sscanf (Continued)

Example:

```
#include <stdio.h> /* for sscanf, printf */

int main(void)
{
  char s[] = "5 T green 3000000.00";
  int number, items;
  char letter;
  char color[10];
  float salary;

  items = sscanf(s, "%d %c %s %f", &number, &letter,
              &color, &salary);

  printf("Number of items scanned = %d\n", items);
  printf("Favorite number = %d\n", number);
  printf("Favorite letter = %c\n", letter);
  printf("Favorite color = %s\n", color);
  printf("Desired salary = $%.2f\n", salary);

}
```

Output:

```
Number of items scanned = 4
Favorite number = 5
Favorite letter = T
Favorite color = green
Desired salary = $3000000.00
```

tmpfile

Description: Creates a temporary file

Include: `<stdio.h>`

Prototype: `FILE *tmpfile(void)`

Return Value: Returns a stream pointer if successful; otherwise, returns a NULL pointer.

Remarks: `tmpfile` creates a file with a unique filename. The temporary file is opened in `w+b` (binary read/write) mode. It will automatically be removed when `exit` is called; otherwise the file will remain in the directory.

Example:

```
#include <stdio.h> /* for tmpfile, printf, */
                   /* FILE, NULL           */

int main(void)
{
  FILE *mytempfile;

  if ((mytempfile = tmpfile()) == NULL)
    printf("Cannot create temporary file");
  else
    printf("Temporary file was created");
}
```

Output:

```
Temporary file was created
```

tmpnam

Description: Creates a unique temporary filename

Include: `<stdio.h>`

Prototype: `char *tmpnam(char *s);`

Argument: *s* pointer to the temporary name

Return Value: Returns a pointer to the filename generated and stores the filename in *s*. If it can not generate a filename, the NULL pointer is returned.

Remarks: The created filename will not conflict with an existing file name. Use `L_tmpnam` to define the size of array the argument of `tmpnam` points to.

Example:

```
#include <stdio.h> /* for tmpnam, L_tmpnam, */
                   /* printf, NULL          */

int main(void)
{
  char *myfilename;
  char mybuf[L_tmpnam];
  char *myptr = (char *) &mybuf;

  if ((myfilename = tmpnam(myptr)) == NULL)
    printf("Cannot create temporary file name");
  else
    printf("Temporary file %s was created",
           myfilename);
}
```

Output:

```
Temporary file ctm00001.tmp was created
```

ungetc

Description: Pushes character back onto stream.

Include: `<stdio.h>`

Prototype: `int ungetc(int c, FILE *stream);`

Argument: *c* character to be pushed back

stream pointer to the open stream

Return Value: Returns the pushed character if successful; otherwise, returns EOF

Remarks: The pushed back character will be returned by a subsequent read on the stream. If more than one character is pushed back, they will be returned in the reverse order of their pushing. A successful call to a file positioning function (`fseek`, `fsetpos` or `rewind`) cancels any pushed back characters. Only one character of pushback is guaranteed. Multiple calls to `ungetc` without an intervening read or file positioning operation may cause a failure.

ungetc (Continued)

Example:

```
#include <stdio.h> /* for ungetc, fgetc,     */
                   /* printf, fopen, fclose, */
                   /* FILE, NULL, EOF        */

int main(void)
{
  FILE *buf;
  char y, c;

  if ((buf = fopen("afile.txt", "r")) == NULL)
    printf("Cannot open afile.txt\n");
  else
  {
    y = fgetc(buf);
    while (y != EOF)
    {
      if (y == 'r')
      {
        c = ungetc(y, buf);
        if (c != EOF)
        {
          printf("2");
          y = fgetc(buf);
        }
      }
      printf("%c", y);
      y = fgetc(buf);
    }
    fclose(buf);
  }
}
```

Input:

Contents of `afile.txt` (used as input):

```
Short
Longer string
```

Output:

```
Sho2rt
Longe2r st2ring
```

vfprintf

Description: Prints formatted data to a stream using a variable length argument list.

Include:
```
<stdio.h>
<stdarg.h>
```

Prototype:
```
int vfprintf(FILE *stream, const char *format,
va_list ap);
```

Arguments:

stream	pointer to the open stream
format	format control string
ap	pointer to a list of arguments

Return Value: Returns number of characters generated or a negative number if an error occurs.

Remarks: The format argument has the same syntax and use that it has in `printf`.
To access the variable length argument list, the *ap* variable must be initialized by the macro `va_start` and may be reinitialized by additional calls to `va_arg`. This must be done before the `vfprintf` function is called. Invoke `va_end` after the function returns. For more details see `stdarg.h`.

Example:
```
#include <stdio.h>  /* for vfprintf, fopen, */
                    /* fclose, printf,      */
                    /* FILE, NULL           */

#include <stdarg.h> /* for va_start,        */
                    /* va_list, va_end      */

FILE *myfile;

void errmsg(const char *fmt, ...)
{
  va_list ap;

  va_start(ap, fmt);
  vfprintf(myfile, fmt, ap);
  va_end(ap);
}

int main(void)
{
  int num = 3;

  if ((myfile = fopen("afile.txt", "w")) == NULL)
    printf("Cannot open afile.txt\n");
  else
  {
    errmsg("Error: The letter '%c' is not %s\n", 'a',
           "an integer value.");
    errmsg("Error: Requires %d%s%c", num,
           " or more characters.", '\n');
  }
  fclose(myfile);
}
```

Output:
Contents of `afile.txt`
```
Error: The letter 'a' is not an integer value.
Error: Requires 3 or more characters.
```

vprintf

Description: Prints formatted text to `stdout` using a variable length argument list

Include: `<stdio.h>`

`<stdarg.h>`

Prototype: `int vprintf(const char *format, va_list ap);`

Arguments:

format	format control string
ap	pointer to a list of arguments

Return Value: Returns number of characters generated or a negative number if an error occurs.

Remarks: The format argument has the same syntax and use that it has in `printf`.
To access the variable length argument list, the *ap* variable must be initialized by the macro `va_start` and may be reinitialized by additional calls to `va_arg`. This must be done before the `vprintf` function is called. Invoke `va_end` after the function returns. For more details see `stdarg.h`

Example:

```
#include <stdio.h>   /* for vprintf, printf */
#include <stdarg.h>  /* for va_start,       */
                     /* va_list, va_end     */

void errmsg(const char *fmt, ...)
{
  va_list ap;

  va_start(ap, fmt);
  printf("Error: ");
  vprintf(fmt, ap);
  va_end(ap);
}

int main(void)
{
  int num = 3;

  errmsg("The letter '%c' is not %s\n", 'a',
         "an integer value.");
  errmsg("Requires %d%s\n", num,
         " or more characters.\n");
}
```

Output:

```
Error: The letter 'a' is not an integer value.
Error: Requires 3 or more characters.
```

vsprintf

Description: Prints formatted text to a string using a variable length argument list

Include: `<stdio.h>`

`<stdarg.h>`

Prototype: `int vsprintf(char *s, const char *format, va_list ap);`

Arguments:

s	storage string for output
format	format control string
ap	pointer to a list of arguments

Return Value: Returns number of characters stored in *s* excluding the terminating null character.

Remarks: The format argument has the same syntax and use that it has in `printf`.
To access the variable length argument list, the *ap* variable must be initialized by the macro `va_start` and may be reinitialized by additional calls to `va_arg`. This must be done before the `vsprintf` function is called. Invoke `va_end` after the function returns. For more details see `stdarg.h`

Example:

```
#include <stdio.h>   /* for vsprintf, printf */
#include <stdarg.h>  /* for va_start,        */
                     /* va_list, va_end      */

void errmsg(const char *fmt, ...)
{
  va_list ap;
  char buf[100];

  va_start(ap, fmt);
  vsprintf(buf, fmt, ap);
  va_end(ap);
  printf("Error: %s", buf);
}

int main(void)
{
  int num = 3;

  errmsg("The letter '%c' is not %s\n", 'a',
         "an integer value.");
  errmsg("Requires %d%s\n", num,
         " or more characters.\n");
}
```

Output:

```
Error: The letter 'a' is not an integer value.
Error: Requires 3 or more characters.
```

3.14 <STDLIB.H> UTILITY FUNCTIONS

The header file `stdlib.h` consists of types, macros and functions that provide text conversions, memory management, searching and sorting abilities, and other general utilities.

div_t

Description:	A type that holds a quotient and remainder of a signed integer division with operands of type `int`.
Include:	`<stdlib.h>`
Prototype:	`typedef struct { int quot, rem; } div_t;`
Remarks:	This is the structure type returned by the function `div`.

ldiv_t

Description:	A type that holds a quotient and remainder of a signed integer division with operands of type `long`.
Include:	`<stdlib.h>`
Prototype:	`typedef struct { long quot, rem; } ldiv_t;`
Remarks:	This is the structure type returned by the function `ldiv`.

size_t

Description:	The type of the result of the `sizeof` operator.
Include:	`<stdlib.h>`

wchar_t

Description:	A type that holds a wide character value.
Include:	`<stdlib.h>`

EXIT_FAILURE

Description:	Reports unsuccessful termination.
Include:	`<stdlib.h>`
Remarks:	`EXIT_FAILURE` is a value for the exit function to return an unsuccessful termination status
Example:	See `exit` for example of use.

EXIT_SUCCESS

Description:	Reports successful termination
Include:	`<stdlib.h>`
Remarks:	`EXIT_SUCCESS` is a value for the exit function to return a successful termination status.
Example:	See `exit` for example of use.

MB_CUR_MAX

Description: Maximum number of characters in a multibyte character

Include: `<stdlib.h>`

Value: 1

NULL

Description: The value of a null pointer constant

Include: `<stdlib.h>`

RAND_MAX

Description: Maximum value capable of being returned by the `rand` function

Include: `<stdlib.h>`

Value: 32767

abort

Description: Aborts the current process.

Include: `<stdlib.h>`

Prototype: `void abort(void);`

Remarks: `abort` will cause the processor to reset.

Example:

```
#include <stdio.h>  /* for fopen, fclose, */
                    /* printf, FILE, NULL */
#include <stdlib.h> /* for abort          */

int main(void)
{
  FILE *myfile;

  if ((myfile = fopen("samp.fil", "r")) == NULL)
  {
    printf("Cannot open samp.fil\n");
    abort();
  }
  else
    printf("Success opening samp.fil\n");

  fclose(myfile);
}
```

Output:

```
Cannot open samp.fil
ABRT
```

abs

Description: Calculates the absolute value.

Include: `<stdlib.h>`

Prototype: `int abs(int i);`

Argument: *i* integer value

Return Value: Returns the absolute value of *i*.

Remarks: A negative number is returned as positive; a positive number is unchanged.

Example:

```
#include <stdio.h>  /* for printf */
#include <stdlib.h> /* for abs    */

int main(void)
{
  int i;

  i = 12;
  printf("The absolute value of  %d is  %d\n",
          i, abs(i));

  i = -2;
  printf("The absolute value of  %d is   %d\n",
          i, abs(i));

  i = 0;
  printf("The absolute value of   %d is   %d\n",
          i, abs(i));
}
```

Output:

```
The absolute value of  12 is  12
The absolute value of  -2 is   2
The absolute value of   0 is   0
```

atexit

Description: Registers the specified function to be called when the program terminates normally.

Include: `<stdlib.h>`

Prototype: `int atexit(void(*func)(void));`

Argument: *func* function to be called

Return Value: Returns a zero if successful; otherwise, returns a non-zero value.

Remarks: For the registered functions to be called, the program must terminate with the `exit` function call.

Example:

```
#include <stdio.h>  /* for scanf, printf */
#include <stdlib.h> /* for atexit, exit  */

void good_msg(void);
void bad_msg(void);
void end_msg(void);
```

atexit (Continued)

```
int main(void)
{
  int number;

  atexit(end_msg);
  printf("Enter your favorite number:");
  scanf("%d", &number);
  printf(" %d\n", number);
  if (number == 5)
  {
    printf("Good Choice\n");
    atexit(good_msg);
    exit(0);
  }
  else
  {
    printf("%d!?\n", number);
    atexit(bad_msg);
    exit(0);
  }
}

void good_msg(void)
{
  printf("That's an excellent number\n");
}

void bad_msg(void)
{
  printf("That's an awful number\n");
}

void end_msg(void)
{
  printf("Now go count something\n");
}
```

Input:

With contents of `UartIn.txt` (used as `stdin` input for simulator):

```
5
```

Output:

```
Enter your favorite number: 5
Good Choice
That's an excellent number
Now go count something
```

Input:

With contents of `UartIn.txt` (used as `stdin` input for simulator):

```
42
```

Output:

```
Enter your favorite number: 42
42!?
That's an awful number
Now go count something
```

atof

Description: Converts a string to a double precision floating-point value.

Include: `<stdlib.h>`

Prototype: `double atof(const char *s);`

Argument: *s* pointer to the string to be converted

Return Value: Returns the converted value if successful; otherwise, returns 0.

Remarks: The number may consist of the following:

```
[whitespace] [sign] digits [.digits]
  [ { e | E }[sign]digits]
```

optional `whitespace`, followed by an optional `sign` then a sequence of one or more `digits` with an optional decimal point, followed by one or more optional `digits` and an optional `e` or `E` followed by an optional signed exponent. The conversion stops when the first unrecognized character is reached. The conversion is the same as `strtod(s,0,0)` except it does no error checking so `errno` will not be set.

Example:

```
#include <stdio.h>  /* for printf */
#include <stdlib.h> /* for atof   */

int main(void)
{
  char a[] = " 1.28";
  char b[] = "27.835e2";
  char c[] = "Number1";
  double x;

  x = atof(a);
  printf("String = \"%s\" float = %f\n", a, x);

  x = atof(b);
  printf("String = \"%s\" float = %f\n", b, x);

  x = atof(c);
  printf("String = \"%s\"  float = %f\n", c, x);
}
```

Output:

```
String = "1.28"     float = 1.280000
String = "27.835:e2" float = 2783.500000
String = "Number1"  float = 0.000000
```

atoi

Description: Converts a string to an integer.

Include: `<stdlib.h>`

Prototype: `int atoi(const char *s);`

Argument: *s* string to be converted

Return Value: Returns the converted integer if successful; otherwise, returns 0.

Remarks: The number may consist of the following:

```
[whitespace] [sign] digits
```

optional `whitespace`, followed by an optional `sign` then a sequence of one or more `digits`. The conversion stops when the first unrecognized character is reached. The conversion is equivalent to `(int) strtol(s,0,10)` except it does no error checking so `errno` will not be set.

Example:

```
#include <stdio.h>  /* for printf */
#include <stdlib.h> /* for atoi   */

int main(void)
{

  char a[] = " -127";
  char b[] = "Number1";
  int x;

  x = atoi(a);
  printf("String = \"%s\"\tint = %d\n", a, x);

  x = atoi(b);
  printf("String = \"%s\"\tint = %d\n", b, x);
}
```

Output:

```
String = " -127"        int = -127
String = "Number1"      int = 0
```

atol

Description: Converts a string to a long integer.

Include: `<stdlib.h>`

Prototype: `long atol(const char *s);`

Argument: *s* string to be converted

Return Value: Returns the converted long integer if successful; otherwise, returns 0

Remarks: The number may consist of the following:

```
[whitespace] [sign] digits
```

optional `whitespace`, followed by an optional `sign` then a sequence of one or more `digits`. The conversion stops when the first unrecognized character is reached. The conversion is equivalent to `(int) strtol(s,0,10)` except it does no error checking so `errno` will not be set.

atol (Continued)

Example:

```
#include <stdio.h>  /* for printf */
#include <stdlib.h> /* for atol   */

int main(void)
{
  char a[] = " -123456";
  char b[] = "2Number";
  long x;

  x = atol(a);
  printf("String = \"%s\"  int = %ld\n", a, x);

  x = atol(b);
  printf("String = \"%s\"  int = %ld\n", b, x);
}
```

Output:

```
String = " -123456"     int = -123456
String = "2Number"      int = 2
```

bsearch

Description: Performs a binary search

Include: `<stdlib.h>`

Prototype:

```
void *bsearch(const void *key, const void *base,
 size_t nelem, size_t size,
 int (*cmp)(const void *ck, const void *ce));
```

Arguments:

key	object to search for
base	pointer to the start of the search data
nelem	number of elements
size	size of elements
cmp	pointer to the comparison function
ck	pointer to the key for the search
ce	pointer to the element being compared with the key.

Return Value: Returns a pointer to the object being searched for if found; otherwise, returns NULL.

Remarks: The value returned by the compare function is <0 if *ck* is less than *ce*, 0 if *ck* is equal to *ce*, or >0 if *ck* is greater than *ce*.
In the following example, `qsort` is used to sort the list before `bsearch` is called. `bsearch` requires the list to be sorted according to the comparison function. This `comp` uses ascending order.

bsearch (Continued)

Example:

```
#include <stdlib.h> /* for bsearch, qsort */
#include <stdio.h>  /* for printf, sizeof */

#define NUM 7

int comp(const void *e1, const void *e2);

int main(void)
{
  int list[NUM] = {35, 47, 63, 25, 93, 16, 52};
  int x, y;
  int *r;

  qsort(list, NUM, sizeof(int), comp);

  printf("Sorted List:   ");
  for (x = 0; x < NUM; x++)
    printf("%d  ", list[x]);


  y = 25;
  r = bsearch(&y, list, NUM, sizeof(int), comp);
  if (r)
    printf("\nThe value %d was found\n", y);
  else
    printf("\nThe value %d was not found\n", y);

  y = 75;
  r = bsearch(&y, list, NUM, sizeof(int), comp);
  if (r)
    printf("\nThe value %d was found\n", y);
  else
    printf("\nThe value %d was not found\n", y);
}

int comp(const void *e1, const void *e2)
{

  const int * a1 = e1;
  const int * a2 = e2;

  if (*a1 < *a2)
    return -1;
  else if (*a1 == *a2)
    return 0;
 else
  return 1;
}
```

Output:

```
Sorted List:   16  25  35  47  52  63  93
The value 25 was found

The value 75 was not found
```

calloc

Description: Allocates an array in memory and initializes the elements to 0.

Include: `<stdlib.h>`

Prototype: `void *calloc(size_t nelem, size_t size);`

Arguments:

nelem	number of elements
size	length of each element

Return Value: Returns a pointer to the allocated space if successful; otherwise, returns a null pointer.

Remarks: Memory returned by `calloc` is aligned correctly for any size data element and is initialized to zero.

Example:

```
/* This program allocates memory for the      */
/* array 'i' of long integers and initializes */
/* them to zero.                              */

#include <stdio.h>  /* for printf, NULL */
#include <stdlib.h> /* for calloc, free */

int main(void)
{
  int x;
  long *i;

  i = (long *)calloc(5, sizeof(long));
  if (i != NULL)
  {
    for (x = 0; x < 5; x++)
      printf("i[%d] = %ld\n", x, i[x]);
    free(i);
  }
  else
    printf("Cannot allocate memory\n");
}
```

Output:

```
i[0] = 0
i[1] = 0
i[2] = 0
i[3] = 0
i[4] = 0
```

div

Description: Calculates the quotient and remainder of two numbers

Include: `<stdlib.h>`

Prototype: `div_t div(int numer, int denom);`

Arguments:

numer	numerator
denom	denominator

Return Value: Returns the quotient and the remainder.

Remarks: The returned quotient will have the same sign as the numerator divided by the denominator. The sign for the remainder will be such that the quotient times the denominator plus the remainder will equal the numerator (quot * denom + rem = numer). Division by zero will invoke the math exception error, which by default, will cause a reset. Write a math error handler to do something else.

div (Continued)

Example:

```
#include <stdlib.h> /* for div, div_t */
#include <stdio.h>  /* for printf     */

void __attribute__((__interrupt__))
_MathError(void)
{
  printf("Illegal instruction executed\n");
  abort();
}

int main(void)
{
  int x, y;
  div_t z;

  x = 7;
  y = 3;
  printf("For div(%d, %d)\n", x, y);
  z = div(x, y);
  printf("The quotient is %d and the "
         "remainder is %d\n\n", z.quot, z.rem);

  x = 7;
  y = -3;
  printf("For div(%d, %d)\n", x, y);
  z = div(x, y);
  printf("The quotient is %d and the "
         "remainder is %d\n\n", z.quot, z.rem);

  x = -5;
  y = 3;
  printf("For div(%d, %d)\n", x, y);
  z = div(x, y);
  printf("The quotient is %d and the "
         "remainder is %d\n\n", z.quot, z.rem);

  x = 7;
  y = 7;
  printf("For div(%d, %d)\n", x, y);
  z = div(x, y);
  printf("The quotient is %d and the "
         "remainder is %d\n\n", z.quot, z.rem);

  x = 7;
  y = 0;
  printf("For div(%d, %d)\n", x, y);
  z = div(x, y);
  printf("The quotient is %d and the "
         "remainder is %d\n\n", z.quot, z.rem);
}
```

div (Continued)

Output:

```
For div(7, 3)
The quotient is 2 and the remainder is 1

For div(7, -3)
The quotient is -2 and the remainder is 1

For div(-5, 3)
The quotient is -1 and the remainder is -2

For div(7, 7)
The quotient is 1 and the remainder is 0

For div(7, 0)
Illegal instruction executed
ABRT
```

exit

Description: Terminates program after clean up.

Include: `<stdlib.h>`

Prototype: `void exit(int status);`

Argument: *status* exit status

Remarks: `exit` calls any functions registered by `atexit` in reverse order of registration, flushes buffers, closes stream, closes any temporary files created with `tmpfile`, and resets the processor. This function is customizable. See `pic30-libs`.

Example:

```
#include <stdio.h>  /* for fopen, printf, */
                    /* FILE, NULL         */
#include <stdlib.h> /* for exit           */

int main(void)
{
  FILE *myfile;

  if ((myfile = fopen("samp.fil", "r" )) == NULL)
  {
    printf("Cannot open samp.fil\n");
    exit(EXIT_FAILURE);
  }
  else
  {
    printf("Success opening samp.fil\n");
    exit(EXIT_SUCCESS);
  }
  printf("This will not be printed");
}
```

Output:

```
Cannot open samp.fil
```

free

Description: Frees memory.

Include: `<stdlib.h>`

Prototype: `void free(void *ptr);`

Argument: *ptr* points to memory to be freed

Remarks: Frees memory previously allocated with `calloc`, `malloc`, or `realloc`. If `free` is used on space that has already been deallocated (by a previous call to `free` or by `realloc`) or on space not allocated with `calloc`, `malloc`, or `realloc`, the behavior is undefined.

Example:

```
#include <stdio.h>  /* for printf, sizeof, */
                    /* NULL                */
#include <stdlib.h> /* for malloc, free    */

int main(void)
{
  long *i;

  if ((i = (long *)malloc(50 * sizeof(long))) ==
       NULL)
    printf("Cannot allocate memory\n");
  else
  {
    printf("Memory allocated\n");
    free(i);
    printf("Memory freed\n");
  }
}
```

Output:

```
Memory allocated
Memory freed
```

getenv

Description: Get a value for an environment variable.

Include: `<stdlib.h>`

Prototype: `char *getenv(const char *name);`

Argument: *name* name of environment variable

Return Value: Returns a pointer to the value of the environment variable if successful; otherwise, returns a null pointer.

Remarks: This function must be customized to be used as described (see `pic30-libs`). By default there are no entries in the environment list for `getenv` to find.

getenv (Continued)

Example:

```
#include <stdio.h>  /* for printf, NULL */
#include <stdlib.h> /* for getenv       */

int main(void)
{
  char *incvar;

  incvar = getenv("INCLUDE");
  if (incvar != NULL)
    printf("INCLUDE environment variable = %s\n",
           incvar);
  else
    printf("Cannot find environment variable "
           "INCLUDE ");
}
```

Output:

```
Cannot find environment variable INCLUDE
```

labs

Description: Calculates the absolute value of a long integer.

Include: `<stdlib.h>`

Prototype: `long labs(long i);`

Argument: *i* long integer value

Return Value: Returns the absolute value of *i*.

Remarks: A negative number is returned as positive; a positive number is unchanged.

Example:

```
#include <stdio.h>  /* for printf */
#include <stdlib.h> /* for labs   */

int main(void)
{
  long i;

  i = 123456;
  printf("The absolute value of %7ld is %6ld\n",
         i, labs(i));

  i = -246834;
  printf("The absolute value of %7ld is %6ld\n",
         i, labs(i));

  i = 0;
  printf("The absolute value of %7ld is %6ld\n",
         i, labs(i));
}
```

Output:

```
The absolute value of  123456 is 123456
The absolute value of -246834 is 246834
The absolute value of       0 is      0
```

ldiv

Description: Calculates the quotient and remainder of two long integers.

Include: <stdlib.h>

Prototype: ldiv_t ldiv(long *numer*, long *denom*);

Arguments:

numer	numerator
denom	denominator

Return Value: Returns the quotient and the remainder.

Remarks: The returned quotient will have the same sign as the numerator divided by the denominator. The sign for the remainder will be such that the quotient times the denominator plus the remainder will equal the numerator (quot * denom + rem = numer). If the denominator is zero, the behavior is undefined.

Example:

```
#include <stdlib.h> /* for ldiv, ldiv_t */
#include <stdio.h>  /* for printf       */

int main(void)
{
  long x,y;
  ldiv_t z;

  x = 7;
  y = 3;
  printf("For ldiv(%ld, %ld)\n", x, y);
  z = ldiv(x, y);
  printf("The quotient is %ld and the "
         "remainder is %ld\n\n", z.quot, z.rem);

  x = 7;
  y = -3;
  printf("For ldiv(%ld, %ld)\n", x, y);
  z = ldiv(x, y);
  printf("The quotient is %ld and the "
         "remainder is %ld\n\n", z.quot, z.rem);

  x = -5;
  y = 3;
  printf("For ldiv(%ld, %ld)\n", x, y);
  z = ldiv(x, y);
  printf("The quotient is %ld and the "
         "remainder is %ld\n\n", z.quot, z.rem);

  x = 7;
  y = 7;
  printf("For ldiv(%ld, %ld)\n", x, y);
  z = ldiv(x, y);
  printf("The quotient is %ld and the "
         "remainder is %ld\n\n", z.quot, z.rem);

  x = 7;
  y = 0;
  printf("For ldiv(%ld, %ld)\n", x, y);
  z = ldiv(x, y);
  printf("The quotient is %ld and the "
         "remainder is %ld\n\n",
         z.quot, z.rem);
}
```

ldiv (Continued)

Output:

```
For ldiv(7, 3)
The quotient is 2 and the remainder is 1

For ldiv(7, -3)
The quotient is -2 and the remainder is 1

For ldiv(-5, 3)
The quotient is -1 and the remainder is -2

For ldiv(7, 7)
The quotient is 1 and the remainder is 0

For ldiv(7, 0)
The quotient is -1 and the remainder is 7
```

Explanation:
In the last example (`ldiv(7,0)`) the denominator is zero, the behavior is undefined.

malloc

Description: Allocates memory.

Include: `<stdlib.h>`

Prototype: `void *malloc(size_t size);`

Argument: *size* number of characters to allocate

Return Value: Returns a pointer to the allocated space if successful; otherwise, returns a null pointer.

Remarks: `malloc` does not initialize memory it returns.

Example:

```
#include <stdio.h>  /* for printf, sizeof, */
                    /* NULL                */
#include <stdlib.h> /* for malloc, free    */

int main(void)
{
  long *i;

  if ((i = (long *)malloc(50 * sizeof(long))) ==
      NULL)
    printf("Cannot allocate memory\n");
  else
  {
    printf("Memory allocated\n");
    free(i);
    printf("Memory freed\n");
  }
}
```

Output:

```
Memory allocated
Memory freed
```

mblen

Description: Gets the length of a multibyte character. (See Remarks.)

Include: `<stdlib.h>`

Prototype: `int mblen(const char *s, size_t n);`

Arguments:

s	points to the multibyte character
n	number of bytes to check

Return Value: Returns zero if s points to a null character; otherwise, returns 1.

Remarks: MPLAB C30 does not support multibyte characters with length greater than 1 byte.

mbstowcs

Description: Converts a multibyte string to a wide character string. (See Remarks.)

Include: `<stdlib.h>`

Prototype: `size_t mbstowcs(wchar_t *wcs, const char *s, size_t n);`

Arguments:

wcs	points to the wide character string
s	points to the multibyte string
n	the number of wide characters to convert.

Return Value: Returns the number of wide characters stored excluding the null character.

Remarks: `mbstowcs` converts *n* number of wide characters unless it encounters a null wide character first. MPLAB C30 does not support multibyte characters with length greater than 1 byte.

mbtowc

Description: Converts a multibyte character to a wide character. (See Remarks.)

Include: `<stdlib.h>`

Prototype: `int mbtowc(wchar_t *pwc, const char *s, size_t n);`

Arguments:

pwc	points to the wide character
s	points to the multibyte character
n	number of bytes to check

Return Value: Returns zero if *s* points to a null character; otherwise, returns 1

Remarks: The resulting wide character will be stored at *pwc*. MPLAB C30 does not support multibyte characters with length greater than 1 byte.

qsort

Description: Performs a quick sort.

Include: `<stdlib.h>`

Prototype:
```
void qsort(void *base, size_t nelem, size_t size,
 int (*cmp)(const void *e1, const void *e2));
```

Arguments:

base	pointer to the start of the array
nelem	number of elements
size	size of the elements
cmp	pointer to the comparison function
e1	pointer to the key for the search
e2	pointer to the element being compared with the key

Remarks: `qsort` overwrites the array with the sorted array. The comparison function is supplied by the user. In the following example, the list is sorted according to the comparison function. This `comp` uses ascending order.

Example:
```
#include <stdlib.h> /* for qsort  */
#include <stdio.h>  /* for printf */

#define NUM 7

int comp(const void *e1, const void *e2);

int main(void)
{
  int list[NUM] = {35, 47, 63, 25, 93, 16, 52};
  int x;

  printf("Unsorted List: ");
  for (x = 0; x < NUM; x++)
    printf("%d  ", list[x]);

  qsort(list, NUM, sizeof(int), comp);

  printf("\n");
  printf("Sorted List:   ");
  for (x = 0; x < NUM; x++)
    printf("%d  ", list[x]);

}

int comp(const void *e1, const void *e2)
{

  const int * a1 = e1;
  const int * a2 = e2;

  if (*a1 < *a2)
    return -1;
  else if (*a1 == *a2)
    return 0;
  else
    return 1;
}
```

Output:
```
Unsorted List: 35  47  63  25  93  16  52
Sorted List:   16  25  35  47  52  63  93
```

rand

Description: Generates a pseudo-random integer.

Include: `<stdlib.h>`

Prototype: `int rand(void);`

Return Value: Returns an integer between 0 and `RAND_MAX`.

Remarks: Calls to this function return pseudo-random integer values in the range [0,`RAND_MAX`]. To use this function effectively, you must seed the random number generator using the `srand` function. This function will always return the same sequence of integers when no seeds are used (as in the example below) or when identical seed values are used. (See `srand` for seed example.)

Example:

```
#include <stdio.h>  /* for printf */
#include <stdlib.h> /* for rand   */

int main(void)
{
  int x;

  for (x = 0; x < 5; x++)
    printf("Number = %d\n", rand());
}
```

Output:

```
Number = 21422
Number = 2061
Number = 16443
Number = 11617
Number = 9125
```

Notice if the program is run a second time, the numbers are the same. See the example for `srand` to seed the random number generator.

realloc

Description: Reallocates memory to allow a size change.

Include: `<stdlib.h>`

Prototype: `void *realloc(void *ptr, size_t size);`

Arguments:

ptr	points to previously allocated memory
size	new size to allocate to

Return Value: Returns a pointer to the allocated space if successful; otherwise, returns a null pointer.

Remarks: If the existing object is smaller than the new object, the entire existing object is copied to the new object and the remainder of the new object is indeterminate. If the existing object is larger than the new object, the function copies as much of the existing object as will fit in the new object. If `realloc` succeeds in allocating a new object, the existing object will be deallocated; otherwise, the existing object is left unchanged. Keep a temporary pointer to the existing object since `realloc` will return a null pointer on failure.

realloc (Continued)

Example:

```
#include <stdio.h> /* for printf, sizeof, NULL */
#include <stdlib.h> /* for realloc, malloc, free */

int main(void)
{
  long *i, *j;

  if ((i = (long *)malloc(50 * sizeof(long)))
       == NULL)
    printf("Cannot allocate memory\n");
  else
  {
    printf("Memory allocated\n");
    /* Temp pointer in case realloc() fails */
    j = i;

    if ((i = (long *)realloc(i, 25 * sizeof(long)))
         == NULL)
    {
      printf("Cannot reallocate memory\n");
       /* j pointed to allocated memory */
      free(j);
    }
    else
    {
      printf("Memory reallocated\n");
      free(i);
    }
  }
}
```

Output:

```
Memory allocated
Memory reallocated
```

srand

Description: Set the starting seed for the pseudo-random number sequence.

Include: `<stdlib.h>`

Prototype: `void srand(unsigned int seed);`

Argument: *seed* starting value for the pseudo-random number sequence

Return Value: None

Remarks: This function sets the starting seed for the pseudo-random number sequence generated by the `rand` function. The `rand` function will always return the same sequence of integers when identical seed values are used. If `rand` is called with a seed value of 1, the sequence of numbers generated will be the same as if `rand` had been called without `srand` having been called first.

Example:

```
#include <stdio.h>  /* for printf      */
#include <stdlib.h> /* for rand, srand */

int main(void)
{
  int x;

  srand(7);
  for (x = 0; x < 5; x++)
    printf("Number = %d\n", rand());
}
```

Output:

```
Number = 16327
Number = 5931
Number = 23117
Number = 30985
Number = 29612
```

strtod

Description: Converts a partial string to a floating-point number of type double.

Include: `<stdlib.h>`

Prototype: `double strtod(const char *s, char **endptr);`

Arguments: *s* string to be converted

endptr pointer to the character at which the conversion stopped

Return Value: Returns the converted number if successful; otherwise, returns 0.

Remarks: The number may consist of the following:

```
[whitespace] [sign] digits [.digits]
  [ { e | E }[sign]digits]
```

optional `whitespace`, followed by an optional `sign`, then a sequence of one or more `digits` with an optional decimal point, followed by one or more optional `digits` and an optional `e` or `E` followed by an optional signed exponent.

`strtod` converts the string until it reaches a character that cannot be converted to a number. *endptr* will point to the remainder of the string starting with the first unconverted character.

If a range error occurs, `errno` will be set.

strtod (Continued)

Example:

```
#include <stdio.h>  /* for printf */
#include <stdlib.h> /* for strtod */

int main(void)
{
  char *end;
  char a[] = "1.28 inches";
  char b[] = "27.835e2i";
  char c[] = "Number1";
  double x;

  x = strtod(a, &end);
  printf("String = \"%s\"  float = %f\n", a, x );
  printf("Stopped at: %s\n\n", end );

  x = strtod(b, &end);
  printf("String = \"%s\" float = %f\n", b, x );
  printf("Stopped at: %s\n\n", end );

  x = strtod(c, &end);
  printf("String = \"%s\"   float = %f\n", c, x );
  printf("Stopped at: %s\n\n", end );
}
```

Output:

```
String = "1.28 inches"  float = 1.280000
Stopped at:  inches

String = "27.835e2i" float = 2783.500000
Stopped at: i

String = "Number1"   float = 0.000000
Stopped at: Number1
```

strtol

Description: Converts a partial string to a long integer.

Include: `<stdlib.h>`

Prototype: `long strtol(const char *s, char **endptr, int base);`

Arguments:

s	string to be converted
endptr	pointer to the character at which the conversion stopped
base	number base to use in conversion

Return Value: Returns the converted number if successful; otherwise, returns 0.

Remarks: If *base* is zero, `strtol` attempts to determine the base automatically. It can be octal, determined by a leading zero, hexadecimal, determined by a leading 0x or 0X, or decimal in any other case. If base is specified `strtol` converts a sequence of digits and letters a-z (case insensitive), where a-z represents the numbers 10-36. Conversion stops when an out of base number is encountered. *endptr* will point to the remainder of the string starting with the first unconverted character. If a range error occurs, `errno` will be set.

Example:

```
#include <stdio.h>  /* for printf */
#include <stdlib.h> /* for strtol */

int main(void)
{
  char *end;
  char a[] = "-12BGEE";
  char b[] = "1234Number";
  long x;

  x = strtol(a, &end, 16);
  printf("String = \"%s\"  long = %ld\n", a, x );
  printf("Stopped at: %s\n\n", end );

  x = strtol(b, &end, 4);
  printf("String = \"%s\"  long = %ld\n", b, x );
  printf("Stopped at: %s\n\n", end );
}
```

Output:

```
String = "-12BGEE"  long = -299
Stopped at: GEE

String = "1234Number"  long = 27
Stopped at: 4Number
```

strtoul

Description: Converts a partial string to an unsigned long integer.

Include: `<stdlib.h>`

Prototype: `unsigned long strtoul(const char *s, char **endptr, int base);`

Arguments:

s	string to be converted
endptr	pointer to the character at which the conversion stopped
base	number base to use in conversion

Return Value: Returns the converted number if successful; otherwise, returns 0.

Remarks: If *base* is zero, `strtol` attempts to determine the base automatically. It can be octal, determined by a leading zero, hexadecimal, determined by a leading 0x or 0X, or decimal in any other case. If base is specified `strtol` converts a sequence of digits and letters a-z (case insensitive), where a-z represents the numbers 10-36. Conversion stops when an out of base number is encountered. *endptr* will point to the remainder of the string starting with the first unconverted character. If a range error occurs, `errno` will be set.

Example:

```
#include <stdio.h>  /* for printf  */
#include <stdlib.h> /* for strtoul */

int main(void)
{
  char *end;
  char a[] = "12BGET3";
  char b[] = "0x1234Number";
  char c[] = "-123abc";
  unsigned long x;

  x = strtoul(a, &end, 25);
  printf("String = \"%s\"  long = %lu\n", a, x );
  printf("Stopped at: %s\n\n", end );

  x = strtoul(b, &end, 0);
  printf("String = \"%s\"  long = %lu\n", b, x );
  printf("Stopped at: %s\n\n", end );

  x = strtoul(c, &end, 0);
  printf("String = \"%s\"  long = %lu\n", c, x );
  printf("Stopped at: %s\n\n", end );
}
```

Output:

```
String = "12BGET3"  long = 429164
Stopped at: T3

String = "0x1234Number"  long = 4660
Stopped at: Number

String = "-123abc"  long = 4294967173
Stopped at: abc
```

system

Description: Execute a command.

Include: `<stdlib.h>`

Prototype: `int system(const char *s);`

Argument:

s	command to be executed

Remarks: This function must be customized to be used as described (see `pic30-libs`). By default `system` will cause a reset if called with anything other than NULL. `system(NULL)` will do nothing.

Example:

```
/* This program uses system */
/* to TYPE its source file. */

#include <stdlib.h> /* for system */

int main(void)
{
  system("type sampsystem.c");
}
```

Output:

```
System(type sampsystem.c) called: Aborting
```

wctomb

Description: Converts a wide character to a multibyte character. (See Remarks.)

Include: `<stdlib.h>`

Prototype: `int wctomb(char *s, wchar_t wchar);`

Arguments:

s	points to the multibyte character
wchar	the wide character to be converted

Return Value: Returns zero if s points to a null character; otherwise, returns 1.

Remarks: The resulting multibyte character is stored at *s*. MPLAB C30 does not support multibyte characters with length greater than 1 character.

wcstombs

Description: Converts a wide character string to a multibyte string. (See Remarks.)

Include: `<stdlib.h>`

Prototype:

```
size_t wcstombs(char *s, const wchar_t *wcs,
 size_t n);
```

Arguments:

s	points to the multibyte string
wcs	points to the wide character string
n	the number of characters to convert

Return Value: Returns the number of characters stored excluding the null character.

Remarks: `wcstombs` converts *n* number of multibyte characters unless it encounters a null character first. MPLAB C30 does not support multibyte characters with length greater than 1 character.

3.15 <STRING.H> STRING FUNCTIONS

The header file `string.h` consists of types, macros and functions that provide tools to manipulate strings.

size_t

Description: The type of the result of the `sizeof` operator.

Include: `<string.h>`

NULL

Description: The value of a null pointer constant.

Include: `<string.h>`

memchr

Description: Locates a character in a buffer.

Include: `<string.h>`

Prototype: `void *memchr(const void *s, int c, size_t n);`

Arguments:

s	pointer to the buffer
c	character to search for
n	number of characters to check

Return Value: Returns a pointer to the location of the match if successful; otherwise, returns null.

Remarks: `memchr` stops when it finds the first occurrence of *c* or after searching *n* number of characters.

Example:

```
#include <string.h> /* for memchr, NULL */
#include <stdio.h>  /* for printf       */

int main(void)
{
  char buf1[50] = "What time is it?";
  char ch1 = 'i', ch2 = 'y';
  char *ptr;
  int res;

  printf("buf1 : %s\n\n", buf1);

  ptr = memchr(buf1, ch1, 50);
if (ptr != NULL)
  {
   res = ptr - buf1 + 1;
   printf("%c found at position %d\n", ch1, res);
  }
  else
    printf("%c not found\n", ch1);
```

memchr (Continued)

```
  printf("\n");

  ptr = memchr(buf1, ch2, 50);
if (ptr != NULL)
  {
    res = ptr - buf1 + 1;
    printf("%c found at position %d\n", ch2, res);
  }
  else
    printf("%c not found\n", ch2);
}
```

Output:

```
buf1 : What time is it?

i found at position 7

y not found
```

memcmp

Description: Compare the contents of two buffers.

Include: `<string.h>`

Prototype: `int memcmp(const void *s1, const void *s2, size_t n);`

Arguments:

s1	first buffer
s2	second buffer
n	number of characters to compare

Return Value: Returns a positive number if *s1* is greater than *s2*, zero if *s1* is equal to *s2*, or a negative number if *s1* is less than *s2*.

Remarks: This function compares the first *n* characters in *s1* to the first *n* characters in *s2* and returns a value indicating whether the buffers are less than, equal to or greater than each other.

Example:

```
#include <string.h> /* memcmp     */
#include <stdio.h>  /* for printf */

int main(void)
{
  char buf1[50] = "Where is the time?";
  char buf2[50] = "Where did they go?";
  char buf3[50] = "Why?";
  int res;

  printf("buf1 : %s\n", buf1);
  printf("buf2 : %s\n", buf2);
  printf("buf3 : %s\n\n", buf3);

  res = memcmp(buf1, buf2, 6);
  if (res < 0)
    printf("buf1 comes before buf2\n");
  else if (res == 0)
    printf("6 characters of buf1 and buf2 "
           "are equal\n");
  else
    printf("buf2 comes before buf1\n");
```

memcmp (Continued)

```
  printf("\n");

  res = memcmp(buf1, buf2, 20);
  if (res < 0)
    printf("buf1 comes before buf2\n");
  else if (res == 0)
    printf("20 characters of buf1 and buf2 "
           "are equal\n");
  else
    printf("buf2 comes before buf1\n");

  printf("\n");

  res = memcmp(buf1, buf3, 20);
  if (res < 0)
    printf("buf1 comes before buf3\n");
  else if (res == 0)
    printf("20 characters of buf1 and buf3 "
           "are equal\n");
  else
    printf("buf3 comes before buf1\n");
}
```

Output:

```
buf1 : Where is the time?
buf2 : Where did they go?
buf3 : Why?

6 characters of buf1 and buf2 are equal

buf2 comes before buf1

buf1 comes before buf3
```

memcpy

Description: Copies characters from one buffer to another.

Include: `<string.h>`

Prototype: `void *memcpy(void *dst , const void *src , size_t n);`

Arguments:

dst	buffer to copy characters to
src	buffer to copy characters from
n	number of characters to copy

Return Value: Returns *dst*.

Remarks: memcpy copies *n* characters from the source buffer *src* to the destination buffer *dst*. If the buffers overlap, the behavior is undefined.

Example:

```
#include <string.h> /* memcpy     */
#include <stdio.h>  /* for printf */

int main(void)
{
  char buf1[50] = "";
  char buf2[50] = "Where is the time?";
  char buf3[50] = "Why?";

  printf("buf1 : %s\n", buf1);
  printf("buf2 : %s\n", buf2);
  printf("buf3 : %s\n\n", buf3);

  memcpy(buf1, buf2, 6);
  printf("buf1 after memcpy of 6 chars of "
         "buf2: \n\t%s\n", buf1);

  printf("\n");

  memcpy(buf1, buf3, 5);
  printf("buf1 after memcpy of 5 chars of "
         "buf3: \n\t%s\n", buf1);
}
```

Output:

```
buf1 :
buf2 : Where is the time?
buf3 : Why?

buf1 after memcpy of 6 chars of buf2:
        Where

buf1 after memcpy of 5 chars of buf3:
        Why?
```

memmove

Description: Copies n characters of the source buffer into the destination buffer, even if the regions overlap.

Include: `<string.h>`

Prototype: `void *memmove(void *s1, const void *s2, size_t n);`

Arguments:

s1	buffer to copy characters to (destination)
s2	buffer to copy characters from (source)
n	number of characters to copy from *s2* to *s1*

Return Value: Returns a pointer to the destination buffer

Remarks: If the buffers overlap, the effect is as if the characters are read first from *s2* then written to *s1* so the buffer is not corrupted.

Example:

```
#include <string.h> /* for memmove */
#include <stdio.h>  /* for printf  */

int main(void)
{
  char buf1[50] = "When time marches on";
  char buf2[50] = "Where is the time?";
  char buf3[50] = "Why?";

  printf("buf1 : %s\n", buf1);
  printf("buf2 : %s\n", buf2);
  printf("buf3 : %s\n\n", buf3);

  memmove(buf1, buf2, 6);
  printf("buf1 after memmove of 6 chars of "
         "buf2: \n\t%s\n", buf1);

  printf("\n");

  memmove(buf1, buf3, 5);
  printf("buf1 after memmove of 5 chars of "
         "buf3: \n\t%s\n", buf1);
}
```

Output:

```
buf1 : When time marches on
buf2 : Where is the time?
buf3 : Why?

buf1 after memmove of 6 chars of buf2:
        Where ime marches on

buf1 after memmove of 5 chars of buf3:
        Why?
```

memset

Description: Copies the specified character into the destination buffer.

Include: `<string.h>`

Prototype: `void *memset(void *s, int c, size_t n);`

Arguments:

s	buffer
c	character to put in buffer
n	number of times

Return Value: Returns the buffer with characters written to it.

Remarks: The character *c* is written to the buffer *n* times.

Example:

```
#include <string.h> /* for memset */
#include <stdio.h>  /* for printf */

int main(void)
{
  char buf1[20] = "What time is it?";
  char buf2[20] = "";
  char ch1 = '?', ch2 = 'y';
  char *ptr;
  int res;

  printf("memset(\"%s\", \'%c\',4);\n", buf1, ch1);
  memset(buf1, ch1, 4);
  printf("buf1 after memset: %s\n", buf1);

  printf("\n");
  printf("memset(\"%s\", \'%c\',10);\n", buf2, ch2);
  memset(buf2, ch2, 10);
  printf("buf2 after memset: %s\n", buf2);
}
```

Output:

```
memset("What time is it?", '?',4);
buf1 after memset: ???? time is it?

memset("", 'y',10);
buf2 after memset: yyyyyyyyyy
```

strcat

Description: Appends a copy of the source string to the end of the destination string.

Include: `<string.h>`

Prototype: `char *strcat(char *s1, const char *s2);`

Arguments:

s1	null terminated destination string to copy to
s2	null terminated source string to be copied

Return Value: Returns a pointer to the destination string.

Remarks: This function appends the source string (including the terminating null character) to the end of the destination string. The initial character of the source string overwrites the null character at the end of the destination string. If the buffers overlap, the behavior is undefined.

Example:

```
#include <string.h> /* for strcat, strlen */
#include <stdio.h>  /* for printf         */

int main(void)
{
  char buf1[50] = "We're here";
  char buf2[50] = "Where is the time?";

  printf("buf1 : %s\n", buf1);
  printf("\t(%d characters)\n\n", strlen(buf1));
  printf("buf2 : %s\n", buf2);
  printf("\t(%d characters)\n\n", strlen(buf2));

  strcat(buf1, buf2);
  printf("buf1 after strcat of buf2: \n\t%s\n",
         buf1);
  printf("\t(%d characters)\n", strlen(buf1));

  printf("\n");

  strcat(buf1, "Why?");
  printf("buf1 after strcat of \"Why?\": \n\t%s\n",
         buf1);
  printf("\t(%d characters)\n", strlen(buf1));
}
```

Output:

```
buf1 : We're here
        (10 characters)

buf2 : Where is the time?
        (18 characters)

buf1 after strcat of buf2:
        We're hereWhere is the time?
        (28 characters)

buf1 after strcat of "Why?":
        We're hereWhere is the time?Why?
        (32 characters)
```

strchr

Description: Locates the first occurrence of a specified character in a string.

Include: `<string.h>`

Prototype: `char *strchr(const char *s, int c);`

Arguments:

s	pointer to the string
c	character to search for

Return Value: Returns a pointer to the location of the match if successful; otherwise, returns a null pointer.

Remarks: This function searches the string *s* to find the first occurrence of the character *c*.

Example:

```
#include <string.h> /* for strchr, NULL */
#include <stdio.h>  /* for printf       */

int main(void)
{
  char buf1[50] = "What time is it?";
  char ch1 = 'm', ch2 = 'y';
  char *ptr;
  int res;

  printf("buf1 : %s\n\n", buf1);

  ptr = strchr(buf1, ch1);
if (ptr != NULL)
  {
    res = ptr - buf1 + 1;
    printf("%c found at position %d\n", ch1, res);
  }
  else
    printf("%c not found\n", ch1);

  printf("\n");

  ptr = strchr(buf1, ch2);
if (ptr != NULL)
  {
    res = ptr - buf1 + 1;
    printf("%c found at position %d\n", ch2, res);
  }
  else
    printf("%c not found\n", ch2);
}
```

Output:

```
buf1 : What time is it?

m found at position 8

y not found
```

strcmp

Description: Compares two strings.

Include: `<string.h>`

Prototype: `int strcmp(const char *s1, const char *s2);`

Arguments:

s1	first string
s2	second string

Return Value: Returns a positive number if *s1* is greater than *s2*, zero if *s1* is equal to *s2*, or a negative number if *s1* is less than *s2*.

Remarks: This function compares successive characters from *s1* and *s2* until they are not equal or the null terminator is reached.

Example:

```
#include <string.h> /* for strcmp */
#include <stdio.h>  /* for printf */

int main(void)
{
  char buf1[50] = "Where is the time?";
  char buf2[50] = "Where did they go?";
  char buf3[50] = "Why?";
  int res;

  printf("buf1 : %s\n", buf1);
  printf("buf2 : %s\n", buf2);
  printf("buf3 : %s\n\n", buf3);

  res = strcmp(buf1, buf2);
  if (res < 0)
    printf("buf1 comes before buf2\n");
  else if (res == 0)
    printf("buf1 and buf2 are equal\n");
  else
    printf("buf2 comes before buf1\n");

  printf("\n");

  res = strcmp(buf1, buf3);
  if (res < 0)
    printf("buf1 comes before buf3\n");
  else if (res == 0)
    printf("buf1 and buf3 are equal\n");
  else
    printf("buf3 comes before buf1\n");

  printf("\n");

  res = strcmp("Why?", buf3);
  if (res < 0)
    printf("\"Why?\" comes before buf3\n");
  else if (res == 0)
    printf("\"Why?\" and buf3 are equal\n");
  else
    printf("buf3 comes before \"Why?\"\n");
}
```

strcmp (Continued)

Output:

```
buf1 : Where is the time?
buf2 : Where did they go?
buf3 : Why?

buf2 comes before buf1

buf1 comes before buf3

"Why?" and buf3 are equal
```

strcoll

Description: Compares one string to another. (See Remarks.)

Include: `<string.h>`

Prototype: `int strcoll(const char *s1, const char *s2);`

Arguments:

s1	first string
s2	second string

Return Value: Using the locale-dependent rules, it returns a positive number if *s1* is greater than *s2*, zero if *s1* is equal to *s2*, or a negative number if *s1* is less than *s2*.

Remarks: Since MPLAB C30 does not support alternate locales, this function is equivalent to `strcmp`.

strcpy

Description: Copy the source string into the destination string.

Include: `<string.h>`

Prototype: `char *strcpy(char *s1, const char *s2);`

Arguments:

s1	destination string to copy to
s2	source string to copy from

Return Value: Returns a pointer to the destination string.

Remarks: All characters of *s2* are copied, including the null terminating character. If the strings overlap, the behavior is undefined.

Example:

```
#include <string.h> /* for strcpy, strlen */
#include <stdio.h>  /* for printf         */

int main(void)
{
  char buf1[50] = "We're here";
  char buf2[50] = "Where is the time?";
  char buf3[50] = "Why?";

  printf("buf1 : %s\n", buf1);
  printf("buf2 : %s\n", buf2);
  printf("buf3 : %s\n\n", buf3);

  strcpy(buf1, buf2);
  printf("buf1 after strcpy of buf2: \n\t%s\n\n",
         buf1);
```

strcpy (Continued)

```
  strcpy(buf1, buf3);
  printf("buf1 after strcpy of buf3: \n\t%s\n",
          buf1);
}
```

Output:

```
buf1 : We're here
buf2 : Where is the time?
buf3 : Why?

buf1 after strcpy of buf2:
        Where is the time?

buf1 after strcpy of buf3:
        Why?
```

strcspn

Description: Calculate the number of consecutive characters at the beginning of a string that are not contained in a set of characters.

Include: `<string.h>`

Prototype: `size_t strcspn(const char *s1, const char *s2);`

Arguments:

s1	pointer to the string to be searched
s2	pointer to characters to search for

Return Value: Returns the length of the segment in *s1* not containing characters found in *s2*.

Remarks: This function will determine the number of consecutive characters from the beginning of *s1* that are not contained in *s2*.

Example:

```
#include <string.h> /* for strcspn */
#include <stdio.h>  /* for printf  */

int main(void)
{
  char str1[20] = "hello";
  char str2[20] = "aeiou";
  char str3[20] = "animal";
  char str4[20] = "xyz";
  int res;

  res = strcspn(str1, str2);
  printf("strcspn(\"%s\", \"%s\") = %d\n",
          str1, str2, res);

  res = strcspn(str3, str2);
  printf("strcspn(\"%s\", \"%s\") = %d\n",
          str3, str2, res);

  res = strcspn(str3, str4);
  printf("strcspn(\"%s\", \"%s\") = %d\n",
          str3, str4, res);
}
```

Output:

```
strcspn("hello", "aeiou") = 1
strcspn("animal", "aeiou") = 0
strcspn("animal", "xyz") = 6
```

strcspn (Continued)

Explanation:
In the first result, `e` is in *s2* so it stops counting after `h`.
In the second result, `a` is in *s2*.
In the third result, none of the characters of *s1* are in *s2* so all characters are counted.

strerror

Description: Gets an internal error message.

Include: `<string.h>`

Prototype: `char *strerror(int errcode);`

Argument: *errcode* number of the error code

Return Value: Returns a pointer to an internal error message string corresponding to the specified error code *errcode*.

Remarks: The array pointed to by `strerror` may be overwritten by a subsequent call to this function.

Example:

```
#include <stdio.h>  /* for fopen, fclose, */
                    /* printf, FILE, NULL */
#include <string.h> /* for strerror       */
#include <errno.h>  /* for errno          */

int main(void)
{
  FILE *myfile;

  if ((myfile = fopen("samp.fil", "r+")) == NULL)
    printf("Cannot open samp.fil: %s\n",
            strerror(errno));
  else
    printf("Success opening samp.fil\n");
  fclose(myfile);
}
```

Output:

```
Cannot open samp.fil: file open error
```

strlen

Description: Finds the length of a string.

Include: `<string.h>`

Prototype: `size_t strlen(const char *s);`

Argument: *s* the string

Return Value: Returns the length of a string.

Remarks: This function determines the length of the string, not including the terminating null character.

strlen (Continued)

Example:

```
#include <string.h> /* for strlen */
#include <stdio.h>  /* for printf */

int main(void)
{
  char str1[20] = "We are here";
  char str2[20] = "";
  char str3[20] = "Why me?";

  printf("str1 : %s\n", str1);
  printf("\t(string length = %d characters)\n\n",
         strlen(str1));
  printf("str2 : %s\n", str2);
  printf("\t(string length = %d characters)\n\n",
         strlen(str2));
  printf("str3 : %s\n", str3);
  printf("\t(string length = %d characters)\n\n\n",
         strlen(str3));

}
```

Output:

```
str1 : We are here
        (string length = 11 characters)

str2 :
        (string length = 0 characters)

str3 : Why me?
        (string length = 7 characters)
```

strncat

Description: Append a specified number of characters from the source string to the destination string.

Include: <string.h>

Prototype: char *strncat(char *s1, const char *s2, size_t n);

Arguments:

s1	destination string to copy to
s2	source string to copy from
n	number of characters to append

Return Value: Returns a pointer to the destination string.

Remarks: This function appends up to *n* characters (a null character and characters that follow it are not appended) from the source string to the end of the destination string. If a null character is not encountered, then a terminating null character is appended to the result. If the strings overlap, the behavior is undefined.

Example:

```
#include <string.h> /* for strncat, strlen */
#include <stdio.h>  /* for printf          */

int main(void)
{
  char buf1[50] = "We're here";
  char buf2[50] = "Where is the time?";
  char buf3[50] = "Why?";
```

strncat (Continued)

```
  printf("buf1 : %s\n", buf1);
  printf("\t(%d characters)\n\n", strlen(buf1));
  printf("buf2 : %s\n", buf2);
  printf("\t(%d characters)\n\n", strlen(buf2));
  printf("buf3 : %s\n", buf3);
  printf("\t(%d characters)\n\n\n", strlen(buf3));

  strncat(buf1, buf2, 6);
  printf("buf1 after strncat of 6 characters "
         "of buf2: \n\t%s\n", buf1);
  printf("\t(%d characters)\n", strlen(buf1));

  printf("\n");

  strncat(buf1, buf2, 25);
  printf("buf1 after strncat of 25 characters "
         "of buf2: \n\t%s\n", buf1);
  printf("\t(%d characters)\n", strlen(buf1));

  printf("\n");

  strncat(buf1, buf3, 4);
  printf("buf1 after strncat of 4 characters "
         "of buf3: \n\t%s\n", buf1);
  printf("\t(%d characters)\n", strlen(buf1));
}
```

Output:

```
buf1 : We're here
      (10 characters)

buf2 : Where is the time?
      (18 characters)

buf3 : Why?
      (4 characters)

buf1 after strncat of 6 characters of buf2:
      We're hereWhere
      (16 characters)

buf1 after strncat of 25 characters of buf2:
      We're hereWhere Where is the time?
      (34 characters)

buf1 after strncat of 4 characters of buf3:
  We're hereWhere Where is the time?Why?
      (38 characters)
```

strncmp

Description: Compare two strings, up to a specified number of characters.

Include: `<string.h>`

Prototype:
```
int strncmp(const char *s1, const char *s2,
 size_t n);
```

Arguments:

s1	first string
s2	second string
n	number of characters to compare

Return Value: Returns a positive number if *s1* is greater than *s2*, zero if *s1* is equal to *s2*, or a negative number if *s1* is less than *s2*.

Remarks: `strncmp` returns a value based on the first character that differs between *s1* and *s2*. Characters that follow a null character are not compared.

Example:
```
#include <string.h> /* for strncmp */
#include <stdio.h>  /* for printf  */

int main(void)
{
  char buf1[50] = "Where is the time?";
  char buf2[50] = "Where did they go?";
  char buf3[50] = "Why?";
  int res;

  printf("buf1 : %s\n", buf1);
  printf("buf2 : %s\n", buf2);
  printf("buf3 : %s\n\n", buf3);

  res = strncmp(buf1, buf2, 6);
  if (res < 0)
    printf("buf1 comes before buf2\n");
  else if (res == 0)
    printf("6 characters of buf1 and buf2 "
           "are equal\n");
  else
   printf("buf2 comes before buf1\n");

  printf("\n");

  res = strncmp(buf1, buf2, 20);
  if (res < 0)
    printf("buf1 comes before buf2\n");
  else if (res == 0)
    printf("20 characters of buf1 and buf2 "
           "are equal\n");
  else
    printf("buf2 comes before buf1\n");
```

strncmp (Continued)

```
  printf("\n");

  res = strncmp(buf1, buf3, 20);
  if (res < 0)
    printf("buf1 comes before buf3\n");
  else if (res == 0)
    printf("20 characters of buf1 and buf3 "
           "are equal\n");
else
    printf("buf3 comes before buf1\n");
}
```

Output:

```
buf1 : Where is the time?
buf2 : Where did they go?
buf3 : Why?

6 characters of buf1 and buf2 are equal

buf2 comes before buf1

buf1 comes before buf3
```

strncpy

Description: Copy characters from the source string into the destination string, up to the specified number of characters.

Include: `<string.h>`

Prototype: `char *strncpy(char *s1, const char *s2, size_t n);`

Arguments:

s1	destination string to copy to
s2	source string to copy from
n	number of characters to copy

Return Value: Returns a pointer to the destination string.

Remarks: Copies *n* characters from the source string to the destination string. If the source string is less than *n* characters, the destination is filled with null characters to total *n* characters. If *n* characters were copied and no null character was found then the destination string will not be null-terminated. If the strings overlap, the behavior is undefined.

Example:

```
#include <string.h> /* for strncpy, strlen */
#include <stdio.h>  /* for printf          */

int main(void)
{
  char buf1[50] = "We're here";
  char buf2[50] = "Where is the time?";
  char buf3[50] = "Why?";
  char buf4[7]  = "Where?";

  printf("buf1 : %s\n", buf1);
  printf("buf2 : %s\n", buf2);
  printf("buf3 : %s\n", buf3);
  printf("buf4 : %s\n", buf4);
```

strncpy (Continued)

```
  strncpy(buf1, buf2, 6);
  printf("buf1 after strncpy of 6 characters "
         "of buf2: \n\t%s\n", buf1);
  printf("\t( %d characters)\n", strlen(buf1));

  printf("\n");

  strncpy(buf1, buf2, 18);
  printf("buf1 after strncpy of 18 characters "
         "of buf2: \n\t%s\n", buf1);
  printf("\t( %d characters)\n", strlen(buf1));

  printf("\n");

  strncpy(buf1, buf3, 5);
  printf("buf1 after strncpy of 5 characters "
         "of buf3: \n\t%s\n", buf1);
  printf("\t( %d characters)\n", strlen(buf1));

  printf("\n");

  strncpy(buf1, buf4, 9);
  printf("buf1 after strncpy of 9 characters "
         "of buf4: \n\t%s\n", buf1);
  printf("\t( %d characters)\n", strlen(buf1));
}
```

Output:

```
buf1 : We're here
buf2 : Where is the time?
buf3 : Why?
buf4 : Where?
buf1 after strncpy of 6 characters of buf2:
      Where here
      ( 10 characters)

buf1 after strncpy of 18 characters of buf2:
      Where is the time?
      ( 18 characters)

buf1 after strncpy of 5 characters of buf3:
      Why?
      ( 4 characters)

buf1 after strncpy of 9 characters of buf4:
      Where?
      ( 6 characters)
```

strncpy (Continued)

Explanation:

Each buffer contains the string shown, followed by null characters for a length of 50. Using `strlen` will find the length of the string up to but not including the first null character.

In the first example, 6 characters of `buf2` (“Where “) replace the first 6 characters of `buf1` ("We're ") and the rest of `buf1` remains the same ("here" plus null characters).

In the second example, 18 characters replace the first 18 characters of `buf1` and the rest remain null characters.

In the third example, 5 characters of `buf3` ("Why?" plus a null terminating character) replace the first 5 characters of `buf1`. `buf1` now actually contains ("Why?", 1 null character, " is the time?", 32 null characters). `strlen` shows 4 characters because it stops when it reaches the first null character.

In the fourth example, since `buf4` is only 7 characters `strncpy` uses 2 additional null characters to replace the first 9 characters of `buf1`. The result of `buf1` is 6 characters ("Where?") followed by 3 null characters, followed by 9 characters ("the time?"), followed by 32 null characters.

strpbrk

Description: Search a string for the first occurrence of a character from a specified set of characters.

Include: `<string.h>`

Prototype: `char *strpbrk(const char *s1, const char *s2);`

Arguments:

s1	pointer to the string to be searched
s2	pointer to characters to search for

Return Value: Returns a pointer to the matched character in *s1* if found; otherwise, returns a null pointer.

Remarks: This function will search *s1* for the first occurrence of a character contained in *s2*.

Example:

```
#include <string.h> /* for strpbrk, NULL */
#include <stdio.h>  /* for printf        */

int main(void)
{
  char str1[20] = "What time is it?";
  char str2[20] = "xyz";
  char str3[20] = "eou?";
  char *ptr;
  int res;

  printf("strpbrk(\"%s\", \"%s\")\n", str1, str2);
  ptr = strpbrk(str1, str2);
if (ptr != NULL)
  {
    res = ptr - str1 + 1;
    printf("match found at position %d\n", res);
  }
  else
    printf("match not found\n");
```

strpbrk (Continued)

```
  printf("\n");

  printf("strpbrk(\"%s\", \"%s\")\n", str1, str3);
  ptr = strpbrk(str1, str3);
if (ptr != NULL)
  {
    res = ptr - str1 + 1;
    printf("match found at position %d\n", res);
  }
  else
    printf("match not found\n");
}
```

Output:

```
strpbrk("What time is it?", "xyz")
match not found

strpbrk("What time is it?", "eou?")
match found at position 9
```

strrchr

Description: Search for the last occurrence of a specified character in a string.

Include: `<string.h>`

Prototype: `char *strrchr(const char *s, int c);`

Arguments:

s	pointer to the string to be searched
c	character to search for

Return Value: Returns a pointer to the character if found; otherwise, returns a null pointer.

Remarks: The function searches the string *s*, including the terminating null character, to find the last occurrence of character *c*.

Example:

```
#include <string.h> /* for strrchr, NULL */
#include <stdio.h>  /* for printf        */

int main(void)
{
  char buf1[50] = "What time is it?";
  char ch1 = 'm', ch2 = 'y';
  char *ptr;
  int res;

  printf("buf1 : %s\n\n", buf1);

  ptr = strrchr(buf1, ch1);
if (ptr != NULL)
  {
    res = ptr - buf1 + 1;
    printf("%c found at position %d\n", ch1, res);
  }
  else
    printf("%c not found\n", ch1);
```

strrchr (Continued)

```
  printf("\n");

  ptr = strrchr(buf1, ch2);
if (ptr != NULL)
  {
    res = ptr - buf1 + 1;
    printf("%c found at position %d\n", ch2, res);
  }
  else
    printf("%c not found\n", ch2);
}
```

Output:

```
buf1 : What time is it?

m found at position 8

y not found
```

strspn

Description: Calculate the number of consecutive characters at the beginning of a string that are contained in a set of characters.

Include: `<string.h>`

Prototype: `size_t strspn(const char *s1, const char *s2);`

Arguments:

s1	pointer to the string to be searched
s2	pointer to characters to search for

Return Value: Returns the number of consecutive characters from the beginning of *s1* that are contained in *s2*.

Remarks: This function stops searching when a character from *s1* is not in *s2*.

Example:

```
#include <string.h> /* for strspn */
#include <stdio.h>  /* for printf */

int main(void)
{
  char str1[20] = "animal";
  char str2[20] = "aeiounm";
  char str3[20] = "aimnl";
  char str4[20] = "xyz";
  int res;

  res = strspn(str1, str2);
  printf("strspn(\"%s\", \"%s\") = %d\n",
          str1, str2, res);

  res = strspn(str1, str3);
  printf("strspn(\"%s\", \"%s\") = %d\n",
          str1, str3, res);

  res = strspn(str1, str4);
  printf("strspn(\"%s\", \"%s\") = %d\n",
          str1, str4, res);
}
```

strspn (Continued)

Output:

```
strspn("animal", "aeiounm") = 5
strspn("animal", "aimnl") = 6
strspn("animal", "xyz") = 0
```

Explanation:

In the first result, `l` is not in *s2*.
In the second result, the terminating null is not in *s2*.
In the third result, `a` is not in *s2* , so the comparison stops.

strstr

Description: Search for the first occurrence of a string inside another string.

Include: `<string.h>`

Prototype: `char *strstr(const char *s1, const char *s2);`

Arguments:

s1	pointer to the string to be searched
s2	pointer to substring to be searched for

Return Value: Returns the address of the first element that matches the substring if found; otherwise, returns a null pointer.

Remarks: This function will find the first occurrence of the string *s2* (excluding the null terminator) within the string *s1*. If *s2* points to a zero length string, *s1* is returned.

Example:

```
#include <string.h> /* for strstr, NULL */
#include <stdio.h>  /* for printf       */

int main(void)
{
  char str1[20] = "What time is it?";
  char str2[20] = "is";
  char str3[20] = "xyz";
  char *ptr;
  int res;

  printf("str1 : %s\n", str1);
  printf("str2 : %s\n", str2);
  printf("str3 : %s\n\n", str3);

  ptr = strstr(str1, str2);
if (ptr != NULL)
  {
   res = ptr - str1 + 1;
   printf("\"%s\" found at position %d\n",
          str2, res);
  }
  else
    printf("\"%s\" not found\n", str2);
```

strstr (Continued)

```
  printf("\n");

  ptr = strstr(str1, str3);
if (ptr != NULL)
  {
    res = ptr - str1 + 1;
    printf("\"%s\" found at position %d\n",
           str3, res);
  }
  else
    printf("\"%s\" not found\n", str3);
}
```

Output:

```
str1 : What time is it?
str2 : is
str3 : xyz

"is" found at position 11

"xyz" not found
```

strtok

Description: Break a string into substrings, or tokens, by inserting null characters in place of specified delimiters.

Include: `<string.h>`

Prototype: `char *strtok(char *s1, const char *s2);`

Arguments:

s1	pointer to the null terminated string to be searched
s2	pointer to characters to be searched for (used as delimiters)

Return Value: Returns a pointer to the first character of a token (the first character in *s1* that does not appear in the set of characters of *s2*). If no token is found, the null pointer is returned.

Remarks: A sequence of calls to this function can be used to split up a string into substrings (or tokens) by replacing specified characters with null characters. The first time this function is invoked on a particular string, that string should be passed in *s1*. After the first time, this function can continue parsing the string from the last delimiter by invoking it with a null value passed in *s1*.

It skips all leading characters that appear in the string *s2* (delimiters), then skips all characters not appearing in *s2* (this segment of characters is the token), and then overwrites the next character with a null character, terminating the current token. The function `strtok` then saves a pointer to the character that follows, from which the next search will start. If `strtok` finds the end of the string before it finds a delimiter, the current token extends to the end of the string pointed to by *s1*. If this is the first call to `strtok`, it does not modify the string (no null characters are written to *s1*). The set of characters that is passed in *s2* need not be the same for each call to `strtok`.

If `strtok` is called with a non-null parameter for *s1* after the initial call, the string becomes the new string to search. The old string previously searched will be lost.

strtok (Continued)

Example:

```
#include <string.h> /* for strtok, NULL */
#include <stdio.h> / * for printf       */

int main(void)
{
  char str1[30] = "Here, on top of the world!";
  char delim[5] = ", .";
  char *word;
  int x;

  printf("str1 : %s\n", str1);
  x = 1;
  word = strtok(str1,delim);
  while (word != NULL)
  {
    printf("word %d: %s\n", x++, word);
    word = strtok(NULL, delim);
  }
}
```

Output:

```
str1 : Here, on top of the world!
word 1: Here
word 2: on
word 3: top
word 4: of
word 5: the
word 6: world!
```

strxfrm

Description: Transforms a string using the locale-dependent rules. (See Remarks.)

Include: `<string.h>`

Prototype: `size_t strxfrm(char *s1, const char *s2, size_t n);`

Arguments:

s1	destination string
s2	source string to be transformed
n	number of characters to transform

Return Value: Returns the length of the transformed string not including the terminating null character. If *n* is zero, the string is not transformed (*s1* may be a point null in this case) and the length of *s2* is returned.

Remarks: If the return value is greater than or equal to *n*, the content of *s1* is indeterminate. Since MPLAB C30 does not support alternate locales, the transformation is equivalent to `strcpy`, except that the length of the destination string is bounded by *n*-1.

3.16 <TIME.H> DATE AND TIME FUNCTIONS

The header file `time.h` consists of types, macros and functions that manipulate time.

clock_t

Description: Stores processor time values.

Include: `<time.h>`

Prototype:
```
typedef long clock_t
```

size_t

Description: The type of the result of the `sizeof` operator.

Include: `<time.h>`

struct tm

Description: Structure used to hold the time and date (calendar time).

Include: `<time.h>`

Prototype:
```
struct tm {
int tm_sec;/*seconds after the minute ( 0 to 61 )*/
           /*allows for up to two leap seconds*/
int tm_min;/*minutes after the hour ( 0 to 59 )*/
int tm_hour;/*hours since midnight ( 0 to 23 )*/
int tm_mday;/*day of month ( 1 to 31 )*/
int tm_mon;/*month ( 0 to 11 where January = 0 )*/
int tm_year;/*years since 1900*/
int tm_wday;/*day of week ( 0 to 6 where Sunday = 0 )*/
int tm_yday;/*day of year ( 0 to 365 where January 1 = 0 )*/
int tm_isdst;/*Daylight Savings Time flag*/
}
```

Remarks: If `tm_isdst is` a positive value, Daylight Savings is in effect. If it is zero, Daylight Saving time is not in effect. If it is a negative value, the status of Daylight Saving Time is not known.

time_t

Description: Represents calendar time values.

Include: `<time.h>`

Prototype:
```
typedef long time_t
```

CLOCKS_PER_SEC

Description: Number of processor clocks per second.

Include: `<time.h>`

Prototype:
```
#define CLOCKS_PER_SEC
```

Value: 1

Remarks: MPLAB C30 returns clock ticks (instruction cycles) not actual time.

NULL

Description: The value of a null pointer constant.

Include: `<time.h>`

asctime

Description: Converts the time structure to a character string.

Include: `<time.h>`

Prototype: `char *asctime(const struct tm *tptr);`

Argument: *tptr* time/date structure

Return Value: Returns a pointer to a character string of the following format:

```
DDD MMM dd hh:mm:ss YYYY
```

`DDD` is day of the week
`MMM` is month of the year
`dd` is day of the month
`hh` is hour
`mm` is minute
`ss` is second
`YYYY` is year

Example:

```
#include <time.h>  /* for asctime, tm */
#include <stdio.h> /* for printf      */

volatile int i;

int main(void)
{
  struct tm when;
  time_t whattime;

  when.tm_sec = 30;
  when.tm_min = 30;
  when.tm_hour = 2;
  when.tm_mday = 1;
  when.tm_mon = 1;
  when.tm_year = 103;

  whattime = mktime(&when);
  printf("Day and time is %s\n", asctime(&when));
}
```

Output:

```
Day and time is Sat Feb  1 02:30:30 2003
```

clock

Description: Calculates the processor time.

Include: `<time.h>`

Prototype: `clock_t clock(void);`

Return Value: Returns the number of clock ticks of elapsed processor time.

Remarks: If the target environment cannot measure elapsed processor time, the function returns -1, cast as a `clock_t`. (i.e. (`clock_t`) -1) By default, MPLAB C30 returns the time as instruction cycles.

clock (Continued)

Example:

```
#include <time.h>  /* for clock  */
#include <stdio.h> /* for printf */

volatile int i;

int main(void)
{
  clock_t start, stop;
  int ct;

  start = clock();
  for (i = 0; i < 10; i++)
  stop = clock();
  printf("start = %ld\n", start);
  printf("stop = %ld\n", stop);
}
```

Output:

```
start = 0
stop = 317
```

ctime

Description: Converts calendar time to a string representation of local time.

Include: `<time.h>`

Prototype: `char *ctime(const time_t *tod);`

Argument: *tod* pointer to stored time

Return Value: Returns the address of a string that represents the local time of the parameter passed.

Remarks: This function is equivalent to `asctime(localtime(tod))`.

Example:

```
#include <time.h>  /* for mktime, tm, ctime */
#include <stdio.h> /* for printf            */

int main(void)
{
  time_t whattime;
  struct tm nowtime;

  nowtime.tm_sec = 30;
  nowtime.tm_min = 30;
  nowtime.tm_hour = 2;
  nowtime.tm_mday = 1;
  nowtime.tm_mon = 1;
  nowtime.tm_year = 103;

  whattime = mktime(&nowtime);
  printf("Day and time %s\n", ctime(&whattime));
}
```

Output:

```
Day and time Sat Feb  1 02:30:30 2003
```

difftime

Description: Find the difference between two times.

Include: `<time.h>`

Prototype: `double difftime(time_t t1, time_t t0);`

Arguments:

`t1`	ending time
`t0`	beginning time

Return Value: Returns the number of seconds between `t1` and `t0`.

Remarks: By default, MPLAB C30 returns the time as instruction cycles so `difftime` returns the number of ticks between `t1` and `t0`.

Example:

```
#include <time.h>  /* for clock, difftime */
#include <stdio.h> /* for printf          */

volatile int i;

int main(void)
{
  clock_t start, stop;
  double elapsed;

  start = clock();
  for (i = 0; i < 10; i++)
  stop = clock();
  printf("start = %ld\n", start);
  printf("stop = %ld\n", stop);
  elapsed = difftime(stop, start);
  printf("Elapsed time = %.0f\n", elapsed);
}
```

Output:

```
start = 0
stop = 317
Elapsed time = 317
```

gmtime

Description: Converts calendar time to time structure expressed as Universal Time Coordinated (UTC) also known as Greenwich Mean Time (GMT).

Include: `<time.h>`

Prototype: `struct tm *gmtime(const time_t *tod);`

Argument:

`tod`	pointer to stored time

Return Value: Returns the address of the time structure.

Remarks: This function breaks down the `tod` value into the time structure of type `tm`. By default, MPLAB C30 returns the time as instruction cycles. With this default `gmtime` and `localtime` will be equivalent except `gmtime` will return `tm_isdst` (Daylight Savings Time flag) as zero to indicate that Daylight Savings Time is not in effect.

gmtime (Continued)

Example:

```
#include <time.h>  /* for gmtime, asctime, */
                   /* time_t, tm           */
#include <stdio.h> /* for printf           */

int main(void)
{
  time_t timer;
  struct tm *newtime;

  timer = 1066668182; /* Mon Oct 20 16:43:02 2003 */

  newtime = gmtime(&timer);
  printf("UTC time = %s\n", asctime(newtime));
}
```

Output:

```
UTC time = Mon Oct 20 16:43:02 2003
```

localtime

Description: Converts a value to the local time.

Include: `<time.h>`

Prototype: `struct tm *localtime(const time_t *tod);`

Argument: *tod* pointer to stored time

Return Value: Returns the address of the time structure.

Remarks: By default, MPLAB C30 returns the time as instruction cycles. With this default `localtime` and `gmtime` will be equivalent except `localtime` will return `tm_isdst` (Daylight Savings Time flag) as -1 to indicate that the status of Daylight Savings Time is not known.

Example:

```
#include <time.h>  /* for localtime,      */
                   /* asctime, time_t, tm */
#include <stdio.h> /* for printf          */

int main(void)
{
  time_t timer;
  struct tm *newtime;

  timer = 1066668182; /* Mon Oct 20 16:43:02 2003 */

  newtime = localtime(&timer);
  printf("Local time = %s\n", asctime(newtime));
}
```

Output:

```
Local time = Mon Oct 20 16:43:02 2003
```

mktime

Description: Converts local time to a calendar value.

Include: `<time.h>`

Prototype: `time_t mktime(struct tm *tptr);`

Argument: *tptr* a pointer to the time structure

Return Value: Returns the calendar time encoded as a value of `time_t`.

Remarks: If the calendar time cannot be represented, the function returns -1, cast as a `time_t` (i.e. (`time_t`) -1).

Example:

```
#include <time.h>  /* for localtime,   */
                   /* asctime, mktime, */
                   /* time_t, tm       */
#include <stdio.h> /* for printf       */

int main(void)
{
  time_t timer, whattime;
  struct tm *newtime;

  timer = 1066668182; /* Mon Oct 20 16:43:02 2003 */
  /* localtime allocates space for struct tm */
  newtime = localtime(&timer);
  printf("Local time = %s", asctime(newtime));

  whattime = mktime(newtime);
  printf("Calendar time as time_t = %ld\n",
          whattime);
}
```

Output:

```
Local time = Mon Oct 20 16:43:02 2003
Calendar time as time_t = 1066668182
```

strftime

Description: Formats the time structure to a string based on the format parameter.

Include: `<time.h>`

Prototype: `size_t strftime(char *s, size_t n, const char *format, const struct tm *tptr);`

Arguments:

s	output string
n	maximum length of string
format	format-control string
tptr	pointer to tm data structure

Return Value: Returns the number of characters placed in the array s if the total including the terminating null is not greater than *n*. Otherwise, the function returns 0 and the contents of array *s* are indeterminate.

Remarks: The format parameters follow:

%a abbreviated weekday name

%A full weekday name

%b abbreviated month name

%B full month name

%c appropriate date and time representation

%d day of the month (01-31)

%H hour of the day (00-23)

strftime (Continued)

%I hour of the day (01-12)

%j day of the year (001-366)

%m month of the year (01-12)

%M minute of the hour (00-59)

%p AM/PM designator

%S second of the minute (00-61)
allowing for up to two leap seconds

%U week number of the year where Sunday is the first day of week 1 (00-53)

%w weekday where Sunday is day 0 (0-6)

%W week number of the year where Monday is the first day of week 1 (00-53)

%x appropriate date representation

%X appropriate time representation

%y year without century (00-99)

%Y year with century

%Z time zone (possibly abbreviated) or no characters if time zone is unavailable

%% percent character %

Example:

```
#include <time.h>  /* for strftime, */
                   /* localtime,    */
                   /* time_t, tm    */
#include <stdio.h> /* for printf    */

int main(void)
{
  time_t timer, whattime;
  struct tm *newtime;
  char buf[128];

  timer = 1066668182; /* Mon Oct 20 16:43:02 2003 */
  /* localtime allocates space for structure */
  newtime = localtime(&timer);

  strftime(buf, 128, "It was a %A, %d days into the "
           "month of %B in the year %Y.\n", newtime);
  printf(buf);

  strftime(buf, 128, "It was %W weeks into the year "
           "or %j days into the year.\n", newtime);
  printf(buf);
}
```

Output:

```
It was a Monday, 20 days into the month of October in
the year 2003.
It was 42 weeks into the year or 293 days into the
year.
```

time

Description: Calculates the current calendar time.

Include: `<time.h>`

Prototype: `time_t time(time_t *tod);`

Argument: *tod* pointer to storage location for time

Return Value: Returns the calendar time encoded as a value of `time_t`.

Remarks: If the target environment cannot determine the time, the function returns -1, cast as a `time_t`. By default, MPLAB C30 returns the time as instruction cycles. This function is customizable. See `pic30-libs`.

Example:

```
#include <time.h>  /* for time   */
#include <stdio.h> /* for printf */

volatile int i;

int main(void)
{
  time_t ticks;

  time(0); /* start time */
  for (i = 0; i < 10; i++) /* waste time */
  time(&ticks); /* get time */
  printf("Time = %ld\n", ticks);
}
```

Output:

```
Time = 256
```

3.17 <MATH.H> MATHEMATICAL FUNCTIONS

The header file `math.h` consists of a macro and various functions that calculate common mathematical operations. Error conditions may be handled with a domain error or range error (see `errno.h`).

A domain error occurs when the input argument is outside the domain over which the function is defined. The error is reported by storing the value of `EDOM` in `errno` and returning a particular value defined for each function.

A range error occurs when the result is too large or too small to be represented in the target precision. The error is reported by storing the value of `ERANGE` in `errno` and returning `HUGE_VAL` if the result overflowed (return value was too large) or a zero if the result underflowed (return value is too small).

Responses to special values, such as NaNs, zeros, and infinities, may vary depending upon the function. Each function description includes a definition of the function's response to such values.

HUGE_VAL

Description: `HUGE_VAL` is returned by a function on a range error (e.g., the function tries to return a value too large to be represented in the target precision).

Include: `<math.h>`

Remarks: `-HUGE_VAL` is returned if a function result is negative and is too large (in magnitude) to be represented in the target precision. When the printed result is `+/- HUGE_VAL`, it will be represented by `+/- inf`.

acos

Description: Calculates the trigonometric arc cosine function of a double precision floating-point value.

Include: `<math.h>`

Prototype: `double acos (double x);`

Argument: *x* value between -1 and 1 for which to return the arc cosine

Return Value: Returns the arc cosine in radians in the range of 0 to pi (inclusive).

Remarks: A domain error occurs if *x* is less than -1 or greater than 1.

Example:

```
#include <math.h>  /* for acos           */
#include <stdio.h> /* for printf, perror */
#include <errno.h> /* for errno          */

int main(void)
{
  double x,y;

  errno = 0;
  x = -2.0;
  y = acos (x);
  if (errno)
    perror("Error");
  printf("The arccosine of %f is %f\n\n", x, y);
```

acos (Continued)

```
  errno = 0;
  x = 0.10;
  y = acos (x);
  if (errno)
    perror("Error");
  printf("The arccosine of %f is %f\n\n", x, y);
}
```

Output:

```
Error: domain error
The arccosine of -2.000000 is nan

The arccosine of 0.100000 is 1.470629
```

acosf

Description: Calculates the trigonometric arc cosine function of a single precision floating-point value.

Include: `<math.h>`

Prototype: `float acosf (float x);`

Argument: *x* value between -1 and 1

Return Value: Returns the arc cosine in radians in the range of 0 to pi (inclusive).

Remarks: A domain error occurs if *x* is less than -1 or greater than 1.

Example:

```
#include <math.h>  /* for acosf          */
#include <stdio.h> /* for printf, perror */
#include <errno.h> /* for errno          */

int main(void)
{
  float x, y;

  errno = 0;
  x = 2.0F;
  y = acosf (x);
  if (errno)
    perror("Error");
  printf("The arccosine of %f is %f\n\n", x, y);

  errno = 0;
  x = 0.0F;
  y = acosf (x);
  if (errno)
    perror("Error");
  printf("The arccosine of %f is %f\n", x, y);
}
```

Output:

```
Error: domain error
The arccosine of 2.000000 is nan

The arccosine of 0.000000 is 1.570796
```

asin

Description: Calculates the trigonometric arc sine function of a double precision floating-point value.

Include: `<math.h>`

Prototype: `double asin (double x);`

Argument: *x* value between -1 and 1 for which to return the arc sine

Return Value: Returns the arc sine in radians in the range of -pi/2 to +pi/2 (inclusive).

Remarks: A domain error occurs if *x* is less than -1 or greater than 1.

Example:

```
#include <math.h>  /* for asin           */
#include <stdio.h> /* for printf, perror */
#include <errno.h> /* for errno          */

int main(void)
{
  double x, y;

  errno = 0;
  x = 2.0;
  y = asin (x);
  if (errno)
    perror("Error");
  printf("The arcsine of %f is %f\n\n", x, y);

  errno = 0;
  x = 0.0;
  y = asin (x);
  if (errno)
    perror("Error");
  printf("The arcsine of %f is %f\n\n", x, y);
}
```

Output:

```
Error: domain error
The arcsine of 2.000000 is nan

The arcsine of 0.000000 is 0.000000
```

asinf

Description: Calculates the trigonometric arc sine function of a single precision floating-point value.

Include: `<math.h>`

Prototype: `float asinf (float x);`

Argument: *x* value between -1 and 1

Return Value: Returns the arc sine in radians in the range of -pi/2 to +pi/2 (inclusive).

Remarks: A domain error occurs if *x* is less than -1 or greater than 1.

Example:

```
#include <math.h>  /* for asinf          */
#include <stdio.h> /* for printf, perror */
#include <errno.h> /* for errno          */

int main(void)
{
  float x, y;
```

asinf (Continued)

```
  errno = 0;
  x = 2.0F;
  y = asinf(x);
  if (errno)
    perror("Error");
  printf("The arcsine of %f is %f\n\n", x, y);

  errno = 0;
  x = 0.0F;
  y = asinf(x);
  if (errno)
    perror("Error");
  printf("The arcsine of %f is %f\n\n", x, y);
}
```

Output:

```
Error: domain error
The arcsine of 2.000000 is nan

The arcsine of 0.000000 is 0.000000
```

atan

Description: Calculates the trigonometric arc tangent function of a double precision floating-point value.

Include: `<math.h>`

Prototype: `double atan (double x);`

Argument: *x* value for which to return the arc tangent

Return Value: Returns the arc tangent in radians in the range of -pi/2 to +pi/2 (inclusive).

Remarks: No domain or range error will occur.

Example:

```
#include <math.h>  /* for atan   */
#include <stdio.h> /* for printf */

int main(void)
{
  double x, y;

  x = 2.0;
  y = atan (x);
  printf("The arctangent of %f is %f\n\n", x, y);

  x = -1.0;
  y = atan (x);
  printf("The arctangent of %f is %f\n\n", x, y);
}
```

Output:

```
The arctangent of 2.000000 is 1.107149

The arctangent of -1.000000 is -0.785398
```

atanf

Description: Calculates the trigonometric arc tangent function of a single precision floating-point value.

Include: `<math.h>`

Prototype: `float atanf (float x);`

Argument: *x* value for which to return the arc tangent

Return Value: Returns the arc tangent in radians in the range of -pi/2 to +pi/2 (inclusive).

Remarks: No domain or range error will occur.

Example:

```
#include <math.h>  /* for atanf  */
#include <stdio.h> /* for printf */

int main(void)
{
  float x, y;

  x = 2.0F;
  y = atanf (x);
  printf("The arctangent of %f is %f\n\n", x, y);

  x = -1.0F;
  y = atanf (x);
  printf("The arctangent of %f is %f\n\n", x, y);
}
```

Output:

```
The arctangent of 2.000000 is 1.107149

The arctangent of -1.000000 is -0.785398
```

atan2

Description: Calculates the trigonometric arc tangent function of y/x.

Include: `<math.h>`

Prototype: `double atan2 (double y, double x);`

Arguments: *y* y value for which to return the arc tangent

x x value for which to return the arc tangent

Return Value: Returns the arc tangent in radians in the range of -pi to pi (inclusive) with the quadrant determined by the signs of both parameters.

Remarks: A domain error occurs if both *x* and *y* are zero or both *x* and *y* are +/- infinity.

Example:

```
#include <math.h>  /* for atan2          */
#include <stdio.h> /* for printf, perror */
#include <errno.h> /* for errno          */

int main(void)
{
  double x, y, z;
```

atan2 (Continued)

```
  errno = 0;
  x = 0.0;
  y = 2.0;
  z = atan2(y, x);
  if (errno)
    perror("Error");
  printf("The arctangent of %f/%f is %f\n\n",
          y, x, z);

  errno = 0;
  x = -1.0;
  y = 0.0;
  z = atan2(y, x);
  if (errno)
    perror("Error");
  printf("The arctangent of %f/%f is %f\n\n",
          y, x, z);

  errno = 0;
  x = 0.0;
  y = 0.0;
  z = atan2(y, x);
  if (errno)
    perror("Error");
  printf("The arctangent of %f/%f is %f\n\n",
          y, x, z);
}
```

Output:

```
The arctangent of 2.000000/0.000000 is 1.570796

The arctangent of 0.000000/-1.000000 is 3.141593

Error: domain error
The arctangent of 0.000000/0.000000 is nan
```

atan2f

Description: Calculates the trigonometric arc tangent function of y/x.

Include: `<math.h>`

Prototype: `float atan2f (float y, float x);`

Arguments:

y	y value for which to return the arc tangent
x	x value for which to return the arc tangent

Return Value: Returns the arc tangent in radians in the range of -pi to pi with the quadrant determined by the signs of both parameters.

Remarks: A domain error occurs if both *x* and *y* are zero or both *x* and *y* are +/- infinity.

Example:

```
#include <math.h>  /* for atan2f         */
#include <stdio.h> /* for printf, perror */
#include <errno.h> /* for errno          */

int main(void)
{
  float x, y, z;

  errno = 0;
  x = 2.0F;
  y = 0.0F;
  z = atan2f (y, x);
  if (errno)
    perror("Error");
  printf("The arctangent of %f/%f is %f\n\n",
          y, x, z);

  errno = 0;
  x = 0.0F;
  y = -1.0F;
  z = atan2f (y, x);
  if (errno)
    perror("Error");
  printf("The arctangent of %f/%f is %f\n\n",
          y, x, z);

  errno = 0;
  x = 0.0F;
  y = 0.0F;
  z = atan2f (y, x);
  if (errno)
    perror("Error");
  printf("The arctangent of %f/%f is %f\n\n",
          y, x, z);
}
```

Output:

```
The arctangent of 2.000000/0.000000 is 1.570796

The arctangent of 0.000000/-1.000000 is 3.141593

Error: domain error
The arctangent of 0.000000/0.000000 is nan
```

ceil

Description: Calculates the ceiling of a value.

Include: `<math.h>`

Prototype: `double ceil(double x);`

Argument: *x* a floating-point value for which to return the ceiling.

Return Value: Returns the smallest integer value greater than or equal to *x*.

Remarks: No domain or range error will occur. See `floor`.

Example:

```
#include <math.h>  /* for ceil   */
#include <stdio.h> /* for printf */

int main(void)
{
  double x[8] = {2.0, 1.75, 1.5, 1.25, -2.0,
                 -1.75, -1.5, -1.25};
  double y;
  int i;

  for (i=0; i<8; i++)
  {
    y = ceil (x[i]);
    printf("The ceiling for  %f is  %f\n", x[i], y);
  }
}
```

Output:

```
The ceiling for  2.000000 is  2.000000
The ceiling for  1.750000 is  2.000000
The ceiling for  1.500000 is  2.000000
The ceiling for  1.250000 is  2.000000
The ceiling for  -2.000000 is  -2.000000
The ceiling for  -1.750000 is  -1.000000
The ceiling for  -1.500000 is  -1.000000
The ceiling for  -1.250000 is  -1.000000
```

ceilf

Description: Calculates the ceiling of a value.

Include: `<math.h>`

Prototype: `float ceilf(float x);`

Argument: *x* floating-point value.

Return Value: Returns the smallest integer value greater than or equal to *x*.

Remarks: No domain or range error will occur. See `floorf`.

Example:

```
#include <math.h>  /* for ceilf  */
#include <stdio.h> /* for printf */

int main(void)
{
  float x[8] = {2.0F, 1.75F, 1.5F, 1.25F,
                -2.0F, -1.75F, -1.5F, -1.25F};
  float y;
  int i;

  for (i=0; i<8; i++)
  {
    y = ceilf (x[i]);
    printf("The ceiling for  %f is  %f\n", x[i], y);
  }
}
```

Output:

```
The ceiling for  2.000000 is  2.000000
The ceiling for  1.750000 is  2.000000
The ceiling for  1.500000 is  2.000000
The ceiling for  1.250000 is  2.000000
The ceiling for  -2.000000 is  -2.000000
The ceiling for  -1.750000 is  -1.000000
The ceiling for  -1.500000 is  -1.000000
The ceiling for  -1.250000 is  -1.000000
```

cos

Description: Calculates the trigonometric cosine function of a double precision floating-point value.

Include: `<math.h>`

Prototype: `double cos (double x);`

Argument: *x* value for which to return the cosine

Return Value: Returns the cosine of *x* in radians in the ranges of -1 to 1 inclusive.

Remarks: A domain error will occur if *x* is a NaN or infinity.

Example:

```
#include <math.h>  /* for cos            */
#include <stdio.h> /* for printf, perror */
#include <errno.h> /* for errno          */

int main(void)
{
  double x,y;
```

cos (Continued)

```
  errno = 0;
  x = -1.0;
  y = cos (x);
  if (errno)
    perror("Error");
  printf("The cosine of %f is %f\n\n", x, y);
  errno = 0;
  x = 0.0;
  y = cos (x);
  if (errno)
    perror("Error");
  printf("The cosine of %f is %f\n\n", x, y);
}
```

Output:

```
The cosine of -1.000000 is 0.540302

The cosine of 0.000000 is 1.000000
```

cosf

Description: Calculates the trigonometric cosine function of a single precision floating-point value.

Include: `<math.h>`

Prototype: `float cosf (float x);`

Argument: *x* value for which to return the cosine

Return Value: Returns the cosine of *x* in radians in the ranges of -1 to 1 inclusive.

Remarks: A domain error will occur if *x* is a NaN or infinity.

Example:

```
#include <math.h>  /* for cosf           */
#include <stdio.h> /* for printf, perror */
#include <errno.h> /* for errno          */

int main(void)
{
  float x, y;

  errno = 0;
  x = -1.0F;
  y = cosf (x);
  if (errno)
    perror("Error");
  printf("The cosine of %f is %f\n\n", x, y);

  errno = 0;
  x = 0.0F;
  y = cosf (x);
  if (errno)
    perror("Error");
  printf("The cosine of %f is %f\n\n", x, y);
}
```

cosf (Continued)

Output:

```
The cosine of -1.000000 is 0.540302

The cosine of 0.000000 is 1.000000
```

cosh

Description: Calculates the hyperbolic cosine function of a double precision floating-point value.

Include: `<math.h>`

Prototype: `double cosh (double x);`

Argument: *x* value for which to return the hyperbolic cosine

Return Value: Returns the hyperbolic cosine of *x*

Remarks: A range error will occur if the magnitude of *x* is too large.

Example:

```
#include <math.h>  /* for cosh           */
#include <stdio.h> /* for printf, perror */
#include <errno.h> /* for errno          */

int main(void)
{
  double x, y;

  errno = 0;
  x = -1.5;
  y = cosh (x);
  if (errno)
    perror("Error");
  printf("The hyperbolic cosine of %f is %f\n\n",
          x, y);

  errno = 0;
  x = 0.0;
  y = cosh (x);
  if (errno)
    perror("Error");
  printf("The hyperbolic cosine of %f is %f\n\n",
          x, y);

  errno = 0;
  x = 720.0;
  y = cosh (x);
  if (errno)
    perror("Error");
  printf("The hyperbolic cosine of %f is %f\n\n",
          x, y);
}
```

Output:

```
The hyperbolic cosine of -1.500000 is 2.352410

The hyperbolic cosine of 0.000000 is 1.000000

Error: range error
The hyperbolic cosine of 720.000000 is inf
```

coshf

Description: Calculates the hyperbolic cosine function of a single precision floating-point value.

Include: `<math.h>`

Prototype: `float coshf (float x);`

Argument: *x* value for which to return the hyperbolic cosine

Return Value: Returns the hyperbolic cosine of *x*

Remarks: A range error will occur if the magnitude of *x* is too large.

Example:

```
#include <math.h>  /* for coshf          */
#include <stdio.h> /* for printf, perror */
#include <errno.h> /* for errno          */

int main(void)
{
  float x, y;

  errno = 0;
  x = -1.0F;
  y = coshf (x);
  if (errno)
    perror("Error");
  printf("The hyperbolic cosine of %f is %f\n\n",
          x, y);

  errno = 0;
  x = 0.0F;
  y = coshf (x);
  if (errno)
    perror("Error");
  printf("The hyperbolic cosine of %f is %f\n\n",
          x, y);

  errno = 0;
  x = 720.0F;
  y = coshf (x);
  if (errno)
    perror("Error");
  printf("The hyperbolic cosine of %f is %f\n\n",
          x, y);
}
```

Output:

```
The hyperbolic cosine of -1.000000 is 1.543081

The hyperbolic cosine of 0.000000 is 1.000000

Error: range error
The hyperbolic cosine of 720.000000 is inf
```

exp

Description: Calculates the exponential function of x (e raised to the power x where x is a double precision floating-point value).

Include: `<math.h>`

Prototype: `double exp (double x);`

Argument: *x* value for which to return the exponential

Return Value: Returns the exponential of *x*. On an overflow, `exp` returns `inf` and on an underflow `exp` returns 0.

Remarks: A range error occurs if the magnitude of *x* is too large.

Example:

```
#include <math.h>  /* for exp            */
#include <stdio.h> /* for printf, perror */
#include <errno.h> /* for errno          */

int main(void)
{
  double x, y;

  errno = 0;
  x = 1.0;
  y = exp (x);
  if (errno)
    perror("Error");
  printf("The exponential of %f is %f\n\n", x, y);

  errno = 0;
  x = 1E3;
  y = exp (x);
  if (errno)
    perror("Error");
  printf("The exponential of %f is %f\n\n", x, y);

  errno = 0;
  x = -1E3;
  y = exp (x);
  if (errno)
    perror("Error");
  printf("The exponential of %f is %f\n\n", x, y);
}
```

Output:

```
The exponential of 1.000000 is 2.718282

Error: range error
The exponential of 1000.000000 is inf

Error: range error
The exponential of -1000.000000 is 0.000000
```

expf

Description: Calculates the exponential function of x (e raised to the power x where x is a single precision floating-point value).

Include: `<math.h>`

Prototype: `float expf (float x);`

Argument: *x* floating-point value for which to return the exponential

Return Value: Returns the exponential of *x*. On an overflow, `expf` returns `inf` and on an underflow exp returns 0.

Remarks: A range error occurs if the magnitude of *x* is too large.

Example:

```
#include <math.h>  /* for expf           */
#include <stdio.h> /* for printf, perror */
#include <errno.h> /* for errno          */

int main(void)
{
  float x, y;

  errno = 0;
  x = 1.0F;
  y = expf (x);
  if (errno)
    perror("Error");
  printf("The exponential of %f is %f\n\n", x, y);

  errno = 0;
  x = 1.0E3F;
  y = expf (x);
  if (errno)
    perror("Error");
  printf("The exponential of %f is %f\n\n", x, y);

  errno = 0;
  x = -1.0E3F;
  y = expf (x);
  if (errno)
    perror("Error");
  printf("The exponential of %f is %f\n\n", x, y);
}
```

Output:

```
The exponential of 1.000000 is 2.718282

Error: range error
The exponential of 1000.000000 is inf

Error: range error
The exponential of -1000.000000 is 0.000000
```

fabs

Description: Calculates the absolute value of a double precision floating-point value.

Include: `<math.h>`

Prototype: `double fabs(double x);`

Argument: *x* floating-point value for which to return the absolute value

Return Value: Returns the absolute value of *x*. (A negative number is returned as positive, a positive number is unchanged.)

Remarks: No domain or range error will occur.

Example:

```
#include <math.h>  /* for fabs   */
#include <stdio.h> /* for printf */

int main(void)
{
  double x, y;

  x = 1.75;
  y = fabs (x);
  printf("The absolute value of  %f is  %f\n", x, y);

  x = -1.5;
  y = fabs (x);
  printf("The absolute value of %f is  %f\n", x, y);
}
```

Output:

```
The absolute value of  1.750000 is  1.750000
The absolute value of -1.500000 is  1.500000
```

fabsf

Description: Calculates the absolute value of a single precision floating-point value.

Include: `<math.h>`

Prototype: `float fabsf(float x);`

Argument: *x* floating-point value for which to return the absolute value

Return Value: Returns the absolute value of *x*. (A negative number is returned as positive, a positive number is unchanged.)

Remarks: No domain or range error will occur.

Example:

```
#include <math.h>  /* for fabsf  */
#include <stdio.h> /* for printf */

int main(void)
{
  float x,y;

  x = 1.75F;
  y = fabsf (x);
  printf("The absolute value of  %f is  %f\n", x, y);

  x = -1.5F;
  y = fabsf (x);
  printf("The absolute value of %f is  %f\n", x, y);
}
```

Output:

```
The absolute value of  1.750000 is  1.750000
The absolute value of -1.500000 is  1.500000
```

floor

Description: Calculates the floor of a double precision floating-point value.

Include: `<math.h>`

Prototype: `double floor (double x);`

Argument: *x* floating-point value for which to return the floor.

Return Value: Returns the largest integer value less than or equal to *x*.

Remarks: No domain or range error will occur. See `ceil`.

Example:

```
#include <math.h>  /* for floor  */
#include <stdio.h> /* for printf */

int main(void)
{
  double x[8] = {2.0, 1.75, 1.5, 1.25, -2.0,
                 -1.75, -1.5, -1.25};
  double y;
  int i;

  for (i=0; i<8; i++)
  {
    y = floor (x[i]);
    printf("The ceiling for %f is %f\n", x[i], y);
  }
}
```

Output:

```
The  floor for 2.000000 is 2.000000
The  floor for 1.750000 is 1.000000
The  floor for 1.500000 is 1.000000
The  floor for 1.250000 is 1.000000
The  floor for -2.000000 is -2.000000
The  floor for -1.750000 is -2.000000
The  floor for -1.500000 is -2.000000
The  floor for -1.250000 is -2.000000
```

floorf

Description: Calculates the floor of a single precision floating-point value.

Include: `<math.h>`

Prototype: `float floorf(float x);`

Argument: *x* floating-point value.

Return Value: Returns the largest integer value less than or equal to *x*.

Remarks: No domain or range error will occur. See `ceilf`.

floorf (Continued)

Example:

```
#include <math.h>  /* for floorf */
#include <stdio.h> /* for printf */

int main(void)
{
  float x[8] = {2.0F, 1.75F, 1.5F, 1.25F,
                -2.0F, -1.75F, -1.5F, -1.25F};
  float y;
  int i;

  for (i=0; i<8; i++)
  {
    y = floorf (x[i]);
    printf("The floor for  %f is  %f\n", x[i], y);
  }
}
```

Output:

```
The floor for  2.000000 is  2.000000
The floor for  1.750000 is  1.000000
The floor for  1.500000 is  1.000000
The floor for  1.250000 is  1.000000
The floor for  -2.000000 is  -2.000000
The floor for  -1.750000 is  -2.000000
The floor for  -1.500000 is  -2.000000
The floor for  -1.250000 is  -2.000000
```

fmod

Description: Calculates the remainder of x/y as a double precision value.

Include: `<math.h>`

Prototype: `double fmod(double x, double y);`

Arguments:

x	a double precision floating-point value.
y	a double precision floating-point value.

Return Value: Returns the remainder of *x* divided by *y*.

Remarks: If *y* = 0, a domain error occurs. If *y* is non-zero, the result will have the same sign as *x* and the magnitude of the result will be less than the magnitude of *y*.

Example:

```
#include <math.h>  /* for fmod           */
#include <stdio.h> /* for printf, perror */
#include <errno.h> /* for errno          */

int main(void)
{
  double x,y,z;

  errno = 0;
  x = 7.0;
  y = 3.0;
  z = fmod(x, y);
  if (errno)
    perror("Error");
  printf("For fmod(%f, %f) the remainder is %f\n\n",
         x, y, z);
```

fmod (Continued)

```
  errno = 0;
  x = 7.0;
  y = 7.0;
  z = fmod(x, y);
  if (errno)
    perror("Error");
  printf("For fmod(%f, %f) the remainder is %f\n\n",
         x, y, z);

  errno = 0;
  x = -5.0;
  y = 3.0;
  z = fmod(x, y);
  if (errno)
    perror("Error");
  printf("For fmod(%f, %f) the remainder is %f\n\n",
         x, y, z);

  errno = 0;
  x = 5.0;
  y = -3.0;
  z = fmod(x, y);
  if (errno)
    perror("Error");
  printf("For fmod(%f, %f) the remainder is %f\n\n",
         x, y, z);

  errno = 0;
  x = -5.0;
  y = -5.0;
  z = fmod(x, y);
  if (errno)
    perror("Error");
  printf("For fmod(%f, %f) the remainder is %f\n\n",
         x, y, z);

  errno = 0;
  x = 7.0;
  y = 0.0;
  z = fmod(x, y);
  if (errno)
    perror("Error");
  printf("For fmod(%f, %f) the remainder is %f\n\n",
         x, y, z);
}
```

fmod (Continued)

Output:

```
For fmod(7.000000, 3.000000) the remainder is
1.000000

For fmod(7.000000, 7.000000) the remainder is
0.000000

For fmod(-5.000000, 3.000000) the remainder is
-2.000000

For fmod(5.000000, -3.000000) the remainder is
2.000000

For fmod(-5.000000, -5.000000) the remainder is
-0.000000

Error: domain error
For fmod(7.000000, 0.000000) the remainder is nan
```

fmodf

Description: Calculates the remainder of x/y as a single precision value.

Include: <math.h>

Prototype: float fmodf(float *x*, float *y*);

Arguments:

x a single precision floating-point value

y a single precision floating-point value

Return Value: Returns the remainder of *x* divided by *y*.

Remarks: If *y* = 0, a domain error occurs. If *y* is non-zero, the result will have the same sign as *x* and the magnitude of the result will be less than the magnitude of *y*.

Example:

```
#include <math.h>  /* for fmodf          */
#include <stdio.h> /* for printf, perror */
#include <errno.h> /* for errno          */

int main(void)
{
  float x,y,z;

  errno = 0;
  x = 7.0F;
  y = 3.0F;
  z = fmodf (x, y);
  if (errno)
    perror("Error");
  printf("For fmodf (%f, %f) the remainder is"
         " %f\n\n", x, y, z);

  errno = 0;
  x = -5.0F;
  y = 3.0F;
  z = fmodf (x, y);
  if (errno)
    perror("Error");
  printf("For fmodf (%f, %f) the remainder is"
         " %f\n\n", x, y, z);
```

fmodf (Continued)

```
  errno = 0;
  x = 5.0F;
  y = -3.0F;
  z = fmodf (x, y);
  if (errno)
    perror("Error");
  printf("For fmodf (%f, %f) the remainder is"
          " %f\n\n", x, y, z);

  errno = 0;
  x = 5.0F;
  y = -5.0F;
  z = fmodf (x, y);
  if (errno)
    perror("Error");
  printf("For fmodf (%f, %f) the remainder is"
          " %f\n\n", x, y, z);

  errno = 0;
  x = 7.0F;
  y = 0.0F;
  z = fmodf (x, y);
  if (errno)
    perror("Error");
  printf("For fmodf (%f, %f) the remainder is"
          " %f\n\n", x, y, z);

  errno = 0;
  x = 7.0F;
  y = 7.0F;
  z = fmodf (x, y);
  if (errno)
    perror("Error");
  printf("For fmodf (%f, %f) the remainder is"
         " %f\n\n", x, y, z);
}
```

Output:

```
For fmodf (7.000000, 3.000000) the remainder is
1.000000

For fmodf (-5.000000, 3.000000) the remainder is
-2.000000

For fmodf (5.000000, -3.000000) the remainder is
2.000000

For fmodf (5.000000, -5.000000) the remainder is
0.000000

Error: domain error
For fmodf (7.000000, 0.000000) the remainder is nan

For fmodf (7.000000, 7.000000) the remainder is
0.000000
```

frexp

Description: Gets the fraction and the exponent of a double precision floating-point number.

Include: `<math.h>`

Prototype: `double frexp (double x, int *exp);`

Arguments:

x floating-point value for which to return the fraction and exponent

exp pointer to a stored integer exponent

Return Value: Returns the fraction, *exp* points to the exponent. If *x* is 0, the function returns 0 for both the fraction and exponent.

Remarks: The absolute value of the fraction is in the range of 1/2 (inclusive) to 1 (exclusive). No domain or range error will occur.

Example:

```
#include <math.h>  /* for frexp  */
#include <stdio.h> /* for printf */

int main(void)
{
  double x,y;
  int n;

  x = 50.0;
  y = frexp (x, &n);
  printf("For frexp of %f\n the fraction is %f\n ",
         x, y);
  printf("  and the exponent is %d\n\n", n);

  x = -2.5;
  y = frexp (x, &n);
  printf("For frexp of %f\n the fraction is %f\n ",
         x, y);
  printf("  and the exponent is %d\n\n", n);

  x = 0.0;
  y = frexp (x, &n);
  printf("For frexp of %f\n the fraction is %f\n ",
         x, y);
  printf("  and the exponent is %d\n\n", n);
}
```

Output:

```
For frexp of 50.000000
  the fraction is 0.781250
   and the exponent is 6

For frexp of -2.500000
  the fraction is -0.625000
   and the exponent is 2

For frexp of 0.000000
  the fraction is 0.000000
   and the exponent is 0
```

frexpf

Description: Gets the fraction and the exponent of a single precision floating-point number.

Include: `<math.h>`

Prototype: `float frexpf (float x, int *exp);`

Arguments:
x floating-point value for which to return the fraction and exponent
exp pointer to a stored integer exponent

Return Value: Returns the fraction, *exp* points to the exponent. If *x* is 0, the function returns 0 for both the fraction and exponent.

Remarks: The absolute value of the fraction is in the range of 1/2 (inclusive) to 1 (exclusive). No domain or range error will occur.

Example:

```
#include <math.h>  /* for frexpf */
#include <stdio.h> /* for printf */

int main(void)
{
  float x,y;
  int n;

  x = 0.15F;
  y = frexpf (x, &n);
  printf("For frexpf of %f\n  the fraction is %f\n ",
         x, y);
  printf("  and the exponent is %d\n\n", n);

  x = -2.5F;
  y = frexpf (x, &n);
  printf("For frexpf of %f\n  the fraction is %f\n ",
         x, y);
  printf("  and the exponent is %d\n\n", n);

  x = 0.0F;
  y = frexpf (x, &n);
  printf("For frexpf of %f\n  the fraction is %f\n ",
         x, y);
  printf("  and the exponent is %d\n\n", n);
}
```

Output:

```
For frexpf of 0.150000
  the fraction is 0.600000
   and the exponent is -2

For frexpf of -2.500000
  the fraction is -0.625000
   and the exponent is 2

For frexpf of 0.000000
  the fraction is 0.000000
   and the exponent is 0
```

ldexp

Description: Calculates the result of a double precision floating-point number multiplied by an exponent of 2.

Include: `<math.h>`

Prototype: `double ldexp(double x, int ex);`

Arguments:

x	floating-point value
ex	integer exponent

Return Value: Returns *x* * 2^*ex*. On an overflow, `ldexp` returns `inf` and on an underflow, `ldexp` returns 0.

Remarks: A range error will occur on overflow or underflow.

Example:

```
#include <math.h>  /* for ldexp          */
#include <stdio.h> /* for printf, perror */
#include <errno.h> /* for errno          */

int main(void)
{
  double x,y;
  int n;

  errno = 0;
  x = -0.625;
  n = 2;
  y = ldexp (x, n);
  if (errno)
    perror("Error");
  printf("For a number = %f and an exponent = %d\n",
         x, n);
  printf("  ldexp(%f, %d) = %f\n\n",
         x, n, y);

  errno = 0;
  x = 2.5;
  n = 3;
  y = ldexp (x, n);
  if (errno)
    perror("Error");
  printf("For a number = %f and an exponent = %d\n",
         x, n);
  printf("  ldexp(%f, %d) = %f\n\n",
         x, n, y);

  errno = 0;
  x = 15.0;
  n = 10000;
  y = ldexp (x, n);
  if (errno)
    perror("Error");
  printf("For a number = %f and an exponent = %d\n",
         x, n);
  printf("  ldexp(%f, %d) = %f\n\n",
         x, n, y);
}
```

ldexp (Continued)

Output:

```
For a number = -0.625000 and an exponent = 2
  ldexp(-0.625000, 2) = -2.500000

For a number = 2.500000 and an exponent = 3
  ldexp(2.500000, 3) = 20.000000

Error: range error
For a number = 15.000000 and an exponent = 10000
  ldexp(15.000000, 10000) = inf
```

ldexpf

Description: Calculates the result of a single precision floating-point number multiplied by an exponent of 2.

Include: `<math.h>`

Prototype: `float ldexpf(float x, int ex);`

Arguments:

x floating-point value

ex integer exponent

Return Value: Returns *x* * 2^*ex*. On an overflow, `ldexp` returns `inf` and on an underflow, `ldexp` returns 0.

Remarks: A range error will occur on overflow or underflow.

Example:

```
#include <math.h>  /* for ldexpf         */
#include <stdio.h> /* for printf, perror */
#include <errno.h> /* for errno          */

int main(void)
{
  float x,y;
  int n;

  errno = 0;
  x = -0.625F;
  n = 2;
  y = ldexpf (x, n);
  if (errno)
    perror("Error");
  printf("For a number = %f and an exponent = %d\n",
         x, n);
  printf("  ldexpf(%f, %d) = %f\n\n",
         x, n, y);

  errno = 0;
  x = 2.5F;
  n = 3;
  y = ldexpf (x, n);
  if (errno)
    perror("Error");
  printf("For a number = %f and an exponent = %d\n",
         x, n);
  printf("  ldexpf(%f, %d) = %f\n\n",
         x, n, y);
```

ldexpf (Continued)

```
  errno = 0;
  x = 15.0F;
  n = 10000;
  y = ldexpf (x, n);
  if (errno)
    perror("Error");
  printf("For a number = %f and an exponent = %d\n",
         x, n);
  printf("  ldexpf(%f, %d) = %f\n\n",
         x, n, y);
}
```

Output:

```
For a number = -0.625000 and an exponent = 2
  ldexpf(-0.625000, 2) = -2.500000

For a number = 2.500000 and an exponent = 3
  ldexpf(2.500000, 3) = 20.000000

Error: range error
For a number = 15.000000 and an exponent = 10000
  ldexpf(15.000000, 10000) = inf
```

log

Description: Calculates the natural logarithm of a double precision floating-point value.

Include: `<math.h>`

Prototype: `double log(double x);`

Argument: *x* any positive value for which to return the log

Return Value: Returns the natural logarithm of *x*. `-inf` is returned if *x* is 0 and NaN is returned if *x* is a negative number.

Remarks: A domain error occurs if $x \leq 0$.

Example:

```
#include <math.h>  /* for log            */
#include <stdio.h> /* for printf, perror */
#include <errno.h> /* for errno          */

int main(void)
{
  double x, y;

  errno = 0;
  x = 2.0;
  y = log (x);
  if (errno)
    perror("Error");
  printf("The natural logarithm of %f is %f\n\n",
          x, y);

  errno = 0;
  x = 0.0;
  y = log (x);
  if (errno)
    perror("Error");
  printf("The natural logarithm of %f is %f\n\n",
          x, y);

  errno = 0;
  x = -2.0;
  y = log (x);
  if (errno)
    perror("Error");
  printf("The natural logarithm of %f is %f\n\n",
          x, y);
}
```

Output:

```
The natural logarithm of 2.000000 is 0.693147

The natural logarithm of 0.000000 is -inf

Error: domain error
The natural logarithm of -2.000000 is nan
```

log10

Description: Calculates the base-10 logarithm of a double precision floating-point value.

Include: `<math.h>`

Prototype: `double log10(double x);`

Argument: *x* any double precision floating-point positive number

Return Value: Returns the base-10 logarithm of *x*. `-inf` is returned if *x* is 0 and NaN is returned if *x* is a negative number.

Remarks: A domain error occurs if $x \leq 0$.

Example:

```
#include <math.h>  /* for log10          */
#include <stdio.h> /* for printf, perror */
#include <errno.h> /* for errno          */

int main(void)
{
  double x, y;

  errno = 0;
  x = 2.0;
  y = log10 (x);
  if (errno)
    perror("Error");
  printf("The base-10 logarithm of %f is %f\n\n",
          x, y);

  errno = 0;
  x = 0.0;
  y = log10 (x);
  if (errno)
    perror("Error");
  printf("The base-10 logarithm of %f is %f\n\n",
          x, y);

  errno = 0;
  x = -2.0;
  y = log10 (x);
  if (errno)
    perror("Error");
  printf("The base-10 logarithm of %f is %f\n\n",
          x, y);
}
```

Output:

```
The base-10 logarithm of 2.000000 is 0.301030

The base-10 logarithm of 0.000000 is -inf

Error: domain error
The base-10 logarithm of -2.000000 is nan
```

log10f

Description: Calculates the base-10 logarithm of a single precision floating-point value.

Include: `<math.h>`

Prototype: `float log10f(float x);`

Argument: *x* any single precision floating-point positive number

Return Value: Returns the base-10 logarithm of *x*. `-inf` is returned if *x* is 0 and NaN is returned if *x* is a negative number.

Remarks: A domain error occurs if $x \leq 0$.

Example:

```
#include <math.h>  /* for log10f         */
#include <stdio.h> /* for printf, perror */
#include <errno.h> /* for errno          */

int main(void)
{
  float x, y;

  errno = 0;
  x = 2.0F;
  y = log10f(x);
  if (errno)
    perror("Error");
  printf("The base-10 logarithm of %f is %f\n\n",
          x, y);

  errno = 0;
  x = 0.0F;
  y = log10f(x);
  if (errno)
    perror("Error");
  printf("The base-10 logarithm of %f is %f\n\n",
          x, y);

  errno = 0;
  x = -2.0F;
  y = log10f(x);
  if (errno)
    perror("Error");
  printf("The base-10 logarithm of %f is %f\n\n",
          x, y);
}
```

Output:

```
The base-10 logarithm of 2.000000 is 0.301030

Error: domain error
The base-10 logarithm of 0.000000 is -inf

Error: domain error
The base-10 logarithm of -2.000000 is nan
```

logf

Description: Calculates the natural logarithm of a single precision floating-point value.

Include: `<math.h>`

Prototype: `float logf(float x);`

Argument: *x* any positive value for which to return the log

Return Value: Returns the natural logarithm of *x*. `-inf` is returned if *x* is 0 and NaN is returned if *x* is a negative number.

Remarks: A domain error occurs if $x \leq 0$.

Example:

```
#include <math.h>  /* for logf           */
#include <stdio.h> /* for printf, perror */
#include <errno.h> /* for errno          */

int main(void)
{
  float x, y;

  errno = 0;
  x = 2.0F;
  y = logf (x);
  if (errno)
    perror("Error");
  printf("The natural logarithm of %f is %f\n\n",
          x, y);

  errno = 0;
  x = 0.0F;
  y = logf (x);
  if (errno)
    perror("Error");
  printf("The natural logarithm of %f is %f\n\n",
          x, y);

  errno = 0;
  x = -2.0F;
  y = logf (x);
  if (errno)
    perror("Error");
  printf("The natural logarithm of %f is %f\n\n",
          x, y);
}
```

Output:

```
The natural logarithm of 2.000000 is 0.693147

The natural logarithm of 0.000000 is -inf

Error: domain error
The natural logarithm of -2.000000 is nan
```

modf

Description: Splits a double precision floating-point value into fractional and integer parts.

Include: `<math.h>`

Prototype: `double modf(double x, double *pint);`

Arguments:

x	double precision floating-point value
pint	pointer to a stored the integer part

Return Value: Returns the signed fractional part and *pint* points to the integer part.

Remarks: The absolute value of the fractional part is in the range of 0 (inclusive) to 1 (exclusive). No domain or range error will occur.

Example:

```
#include <math.h>  /* for modf   */
#include <stdio.h> /* for printf */

int main(void)
{
  double x,y,n;

  x = 0.707;
  y = modf (x, &n);
  printf("For %f the fraction is %f\n ", x, y);
  printf("  and the integer is %0.f\n\n", n);

  x = -15.2121;
  y = modf (x, &n);
  printf("For %f the fraction is %f\n ", x, y);
  printf("  and the integer is %0.f\n\n", n);
}
```

Output:

```
For 0.707000 the fraction is 0.707000
   and the integer is 0

For -15.212100 the fraction is -0.212100
   and the integer is -15
```

modff

Description: Splits a single precision floating-point value into fractional and integer parts.

Include: `<math.h>`

Prototype: `float modff(float x, float *pint);`

Arguments:

x	single precision floating-point value
pint	pointer to stored integer part

Return Value: Returns the signed fractional part and *pint* points to the integer part.

Remarks: The absolute value of the fractional part is in the range of 0 (inclusive) to 1 (exclusive). No domain or range error will occur.

Example:

```
#include <math.h>  /* for modff  */
#include <stdio.h> /* for printf */

int main(void)
{
  float x,y,n;

  x = 0.707F;
  y = modff (x, &n);
  printf("For %f the fraction is %f\n ", x, y);
  printf("  and the integer is %0.f\n\n", n);

  x = -15.2121F;
  y = modff (x, &n);
  printf("For %f the fraction is %f\n ", x, y);
  printf("  and the integer is %0.f\n\n", n);
}
```

Output:

```
For 0.707000 the fraction is 0.707000
   and the integer is 0

For -15.212100 the fraction is -0.212100
   and the integer is -15
```

pow

Description: Calculates x raised to the power y.

Include: `<math.h>`

Prototype: `double pow(double x, double y);`

Arguments:

x	the base
y	the exponent

Return Value: Returns *x* raised to the power *y* (*x*^*y*).

Remarks: If *y* is 0, `pow` returns 1. If *x* is 0.0 and *y* is less than 0 `pow` returns `inf` and a domain error occurs. If the result overflows or underflows, a range error occurs.

Example:

```
#include <math.h>  /* for pow            */
#include <stdio.h> /* for printf, perror */
#include <errno.h> /* for errno          */

int main(void)
{
  double x,y,z;

  errno = 0;
  x = -2.0;
  y = 3.0;
  z = pow (x, y);
  if (errno)
    perror("Error");
  printf("%f raised to %f is %f\n\n ", x, y, z);

  errno = 0;
  x = 3.0;
  y = -0.5;
  z = pow (x, y);
  if (errno)
    perror("Error");
  printf("%f raised to %f is %f\n\n ", x, y, z);

 errno = 0;
  x = 4.0;
  y = 0.0;
  z = pow (x, y);
  if (errno)
    perror("Error");
  printf("%f raised to %f is %f\n\n ", x, y, z);

  errno = 0;
  x = 0.0;
  y = -3.0;
  z = pow (x, y);
  if (errno)
    perror("Error");
  printf("%f raised to %f is %f\n\n ", x, y, z);
}
```

pow (Continued)

Output:

```
-2.000000 raised to 3.000000 is -8.000000

 3.000000 raised to -0.500000 is 0.577350

 4.000000 raised to 0.000000 is 1.000000

 Error: domain error
 0.000000 raised to -3.000000 is inf
```

powf

Description: Calculates x raised to the power y.

Include: <math.h>

Prototype: float powf(float *x*, float *y*);

Arguments:

x base

y exponent

Return Value: Returns *x* raised to the power *y* (*x*^*y*).

Remarks: If *y* is 0, powf returns 1. If *x* is 0.0 and *y* is less than 0 powf returns inf and a domain error occurs. If the result overflows or underflows, a range error occurs.

Example:

```
#include <math.h>  /* for powf           */
#include <stdio.h> /* for printf, perror */
#include <errno.h> /* for errno          */

int main(void)
{
  float x,y,z;

  errno = 0;
  x = -2.0F;
  y = 3.0F;
  z = powf (x, y);
  if (errno)
    perror("Error");
  printf("%f raised to %f is %f\n\n ", x, y, z);

  errno = 0;
  x = 3.0F;
  y = -0.5F;
  z = powf (x, y);
  if (errno)
    perror("Error");
  printf("%f raised to %f is %f\n\n ", x, y, z);

  errno = 0;
  x = 0.0F;
  y = -3.0F;
  z = powf (x, y);
  if (errno)
    perror("Error");
  printf("%f raised to %f is %f\n\n ", x, y, z);
}
```

powf (Continued)

Output:

```
-2.000000 raised to 3.000000 is -8.000000

 3.000000 raised to -0.500000 is 0.577350

 Error: domain error
 0.000000 raised to -3.000000 is inf
```

sin

Description: Calculates the trigonometric sine function of a double precision floating-point value.

Include: <math.h>

Prototype: double sin (double *x*);

Argument: *x* value for which to return the sine

Return Value: Returns the sine of *x* in radians in the ranges of -1 to 1 inclusive.

Remarks: A domain error will occur if t *x* is a NaN or infinity.

Example:

```
#include <math.h>  /* for sin            */
#include <stdio.h> /* for printf, perror */
#include <errno.h> /* for errno          */

int main(void)
{
  double x, y;

  errno = 0;
  x = -1.0;
  y = sin (x);
  if (errno)
    perror("Error");
  printf("The sine of %f is %f\n\n", x, y);

  errno = 0;
  x = 0.0;
  y = sin (x);
  if (errno)
    perror("Error");
  printf("The sine of %f is %f\n\n", x, y);
}
```

Output:

```
The sine of -1.000000 is -0.841471

The sine of 0.000000 is 0.000000
```

sinf

Description: Calculates the trigonometric sine function of a single precision floating-point value.

Include: `<math.h>`

Prototype: `float sinf (float x);`

Argument: *x* value for which to return the sine

Return Value: Returns the sin of *x* in radians in the ranges of -1 to 1 inclusive.

Remarks: A domain error will occur if *x* is a NaN or infinity.

Example:

```
#include <math.h>  /* for sinf           */
#include <stdio.h> /* for printf, perror */
#include <errno.h> /* for errno          */

int main(void)
{
  float x, y;

  errno = 0;
  x = -1.0F;
  y = sinf (x);
  if (errno)
    perror("Error");
  printf("The sine of %f is %f\n\n", x, y);

  errno = 0;
  x = 0.0F;
  y = sinf (x);
  if (errno)
    perror("Error");
  printf("The sine of %f is %f\n\n", x, y);
}
```

Output:

```
The sine of -1.000000 is -0.841471

The sine of 0.000000 is 0.000000
```

sinh

Description: Calculates the hyperbolic sine function of a double precision floating-point value.

Include: `<math.h>`

Prototype: `double sinh (double x);`

Argument: *x* value for which to return the hyperbolic sine

Return Value: Returns the hyperbolic sine of *x*

Remarks: A range error will occur if the magnitude of *x* is too large.

Example:

```
#include <math.h>  /* for sinh           */
#include <stdio.h> /* for printf, perror */
#include <errno.h> /* for errno          */

int main(void)
{
  double x, y;

  errno = 0;
  x = -1.5;
  y = sinh (x);
  if (errno)
    perror("Error");
  printf("The hyperbolic sine of %f is %f\n\n",
          x, y);

  errno = 0;
  x = 0.0;
  y = sinh (x);
  if (errno)
    perror("Error");
  printf("The hyperbolic sine of %f is %f\n\n",
          x, y);

  errno = 0;
  x = 720.0;
  y = sinh (x);
  if (errno)
    perror("Error");
  printf("The hyperbolic sine of %f is %f\n\n",
          x, y);
}
```

Output:

```
The hyperbolic sine of -1.500000 is -2.129279

The hyperbolic sine of 0.000000 is 0.000000

Error: range error
The hyperbolic sine of 720.000000 is inf
```

sinhf

Description: Calculates the hyperbolic sine function of a single precision floating-point value.

Include: `<math.h>`

Prototype: `float sinhf (float x);`

Argument: *x* value for which to return the hyperbolic sine

Return Value: Returns the hyperbolic sine of *x*

Remarks: A range error will occur if the magnitude of *x* is too large.

Example:

```
#include <math.h>  /* for sinhf          */
#include <stdio.h> /* for printf, perror */
#include <errno.h> /* for errno          */

int main(void)
{
  float x, y;

  errno = 0;
  x = -1.0F;
  y = sinhf (x);
  if (errno)
    perror("Error");
  printf("The hyperbolic sine of %f is %f\n\n",
          x, y);

  errno = 0;
  x = 0.0F;
  y = sinhf (x);
  if (errno)
    perror("Error");
  printf("The hyperbolic sine of %f is %f\n\n",
          x, y);
}
```

Output:

```
The hyperbolic sine of -1.000000 is -1.175201

The hyperbolic sine of 0.000000 is 0.000000
```

sqrt

Description: Calculates the square root of a double precision floating-point value.

Include: `<math.h>`

Prototype: `double sqrt(double x);`

Argument: *x* a non-negative floating-point value

Return Value: Returns the non-negative square root of *x*..

Remarks: If *x* is negative, a domain error occurs.

Example:

```
#include <math.h>  /* for sqrt           */
#include <stdio.h> /* for printf, perror */
#include <errno.h> /* for errno          */

int main(void)
{
  double x, y;

  errno = 0;
  x = 0.0;
  y = sqrt (x);
  if (errno)
    perror("Error");
  printf("The square root of %f is %f\n\n", x, y);

  errno = 0;
  x = 9.5;
  y = sqrt (x);
  if (errno)
    perror("Error");
  printf("The square root of %f is %f\n\n", x, y);

  errno = 0;
  x = -25.0;
  y = sqrt (x);
  if (errno)
    perror("Error");
  printf("The square root of %f is %f\n\n", x, y);
}
```

Output:

```
The square root of 0.000000 is 0.000000

The square root of 9.500000 is 3.082207

Error: domain error
The square root of -25.000000 is nan
```

sqrtf

Description: Calculates the square root of a single precision floating-point value.

Include: `<math.h>`

Prototype: `float sqrtf(float x);`

Argument: *x* non-negative floating-point value

Return Value: Returns the non-negative square root of *x*.

Remarks: If *x* is negative, a domain error occurs.

Example:

```
#include <math.h>  /* for sqrtf          */
#include <stdio.h> /* for printf, perror */
#include <errno.h> /* for errno          */

int main(void)
{
  double x;

  errno = 0;
  x = sqrtf (0.0F);
  if (errno)
    perror("Error");
  printf("The square root of 0.0F is %f\n\n", x);

  errno = 0;
  x = sqrtf (9.5F);
  if (errno)
    perror("Error");
  printf("The square root of 9.5F is %f\n\n", x);

  errno = 0;
  x = sqrtf (-25.0F);
  if (errno)
    perror("Error");
  printf("The square root of -25F is %f\n", x);
}
```

Output:

```
The square root of 0.0F is 0.000000

The square root of 9.5F is 3.082207

Error: domain error
The square root of -25F is nan
```

tan

Description: Calculates the trigonometric tangent function of a double precision floating-point value.

Include: `<math.h>`

Prototype: `double tan (double x);`

Argument: *x* value for which to return the tangent

Return Value: Returns the tangent of *x* in radians.

Remarks: A domain error will occur if *x* is a NaN or infinity.

Example:

```
#include <math.h>  /* for tan            */
#include <stdio.h> /* for printf, perror */
#include <errno.h> /* for errno          */

int main(void)
{
  double x, y;

  errno = 0;
  x = -1.0;
  y = tan (x);
  if (errno)
    perror("Error");
  printf("The tangent of %f is %f\n\n", x, y);

  errno = 0;
  x = 0.0;
  y = tan (x);
  if (errno)
    perror("Error");
  printf("The tangent of %f is %f\n\n", x, y);
}
```

Output:

```
The tangent of -1.000000 is -1.557408

The tangent of 0.000000 is 0.000000
```

tanf

Description: Calculates the trigonometric tangent function of a single precision floating-point value.

Include: `<math.h>`

Prototype: `float tanf (float x);`

Argument: *x* value for which to return the tangent

Return Value: Returns the tangent of *x*

Remarks: A domain error will occur if *x* is a NaN or infinity.

Example:

```
#include <math.h>  /* for tanf           */
#include <stdio.h> /* for printf, perror */
#include <errno.h> /* for errno          */

int main(void)
{
  float x, y;
```

tanf (Continued)

```
  errno = 0;
  x = -1.0F;
  y = tanf (x);
  if (errno)
    perror("Error");
  printf("The tangent of %f is %f\n\n", x, y);

  errno = 0;
  x = 0.0F;
  y = tanf (x);
  if (errno)
    perror("Error");
  printf("The tangent of %f is %f\n", x, y);
}
```

Output:

```
The tangent of -1.000000 is -1.557408

The tangent of 0.000000 is 0.000000
```

tanh

Description: Calculates the hyperbolic tangent function of a double precision floating-point value.

Include: `<math.h>`

Prototype: `double tanh (double x);`

Argument: *x* value for which to return the hyperbolic tangent

Return Value: Returns the hyperbolic tangent of *x* in the ranges of -1 to 1 inclusive.

Remarks: No domain or range error will occur.

Example:

```
#include <math.h>  /* for tanh   */
#include <stdio.h> /* for printf */

int main(void)
{
  double x, y;

  x = -1.0;
  y = tanh (x);
  printf("The hyperbolic tangent of %f is %f\n\n",
          x, y);

  x = 2.0;
  y = tanh (x);
  printf("The hyperbolic tangent of %f is %f\n\n",
          x, y);
}
```

Output:

```
The hyperbolic tangent of -1.000000 is -0.761594

The hyperbolic tangent of 2.000000 is 0.964028
```

tanhf

Description: Calculates the hyperbolic tangent function of a single precision floating-point value.

Include: `<math.h>`

Prototype: `float tanhf (float x);`

Argument: *x* value for which to return the hyperbolic tangent

Return Value: Returns the hyperbolic tangent of *x* in the ranges of -1 to 1 inclusive.

Remarks: No domain or range error will occur.

Example:

```
#include <math.h>  /* for tanhf  */
#include <stdio.h> /* for printf */

int main(void)
{
  float x, y;

  x = -1.0F;
  y = tanhf (x);
  printf("The hyperbolic tangent of %f is %f\n\n",
          x, y);

  x = 0.0F;
  y = tanhf (x);
  printf("The hyperbolic tangent of %f is %f\n\n",
          x, y);
}
```

Output:

```
The hyperbolic tangent of -1.000000 is -0.761594

The hyperbolic tangent of 0.000000 is 0.000000
```

3.18 PIC30-LIBS

The following functions are standard C library helper functions:

- `_exit` terminate program execution
- `brk` set the end of the process's data space
- `close` close a file
- `lseek` move a file pointer to a specified location
- `open` open a file
- `read` read data from a file
- `sbrk` extend the process's data space by a given increment
- `write` write data to a file

These functions are called by other functions in the standard C library and must be modified for the target application. The corresponding object modules are distributed in the `libpic30-`*`omf`*`.a` archive and the source code (for MPLAB C30) is available in the `src\pic30` folder.

Several standard C library functions must also be modified for the target application. They are:

- `getenv` get a value for an environment variable
- `remove` remove a file
- `rename` rename a file or directory
- `system` execute a command
- `time` get the system time

Although these functions are part of the standard C library, the object modules are distributed in the `libpic30-`*`omf`*`.a` archive and the source code (for MPLAB C30) is available in the `src\pic30` folder. These modules are not distributed as part of `libc-`*`omf`*`.a`.

Additional functions/constants to support a simulated UART are:

- `__attach_input_file` attach a file to the standard input
- `__close_input_file` close a file attached to the standard input
- `__delay32` provide a specified delay
- `__C30_UART` set the desired UART module

The corresponding object modules are distributed in the `libpic30-`*`omf`*`.a` archive and the source code (for MPLAB C30) is available in the `src\pic30` folder.

3.18.1 Rebuilding the libpic30-*omf*.a library

By default, the helper functions listed in this chapter were written to work with the `sim30` simulator. The header file, `simio.h`, defines the interface between the library and the simulator. It is provided so you can rebuild the libraries and continue to use the simulator. However, your application should not use this interface since the simulator will not be available to an embedded application.

The helper functions must be modified and rebuilt for your target application. The `libpic30-`*`omf`*`.a` library can be rebuild with the batch file named `makelib.bat`, which has been provided with the sources in `src\pic30`. Execute the batch file from a command window. Be sure you are in the `src\pic30` directory. Then copy the newly compiled file (`libpic30-`*`omf`*`.a`) into the lib directory.

3.18.2 Function Descriptions

This section describes the functions that must be customized for correct operation of the Standard C Library in your target environment. The default behavior section describes what the function does as it is distributed. The description and remarks describe what it typically should do.

__attach_input_file

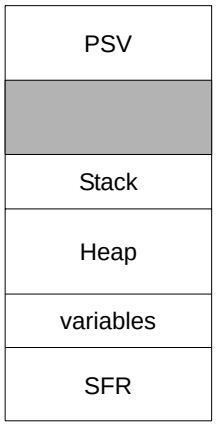
Description:	Attach a hosted file to the standard input stream.
Include:	None
Prototype:	`int __attach_input_file(const char *p);`
Argument:	*p* pointer to file
Remarks:	This function differs from the MPLAB IDE mechanism of providing an input file because it provides "on-demand" access to the file. That is, data will only be read from the file upon request and the asynchronous nature of the UART is not simulated. This function may be called more than once; any opened file will be closed. It is only appropriate to call this function in a simulated environment.
Default Behavior:	Allows the programmer to attach a hosted file to the standard input stream, `stdin`. The function will return 0 to indicate failure. If the file cannot be opened for whatever reason, standard in will remain connected (or be re-connected) to the simulated UART.
File:	`__attach_input_file.c`

brk

Description:	Set the end of the process's data space.
Include:	None
Prototype:	`int brk(void *endds);`
Argument:	*endds* pointer to the end of the data segment
Return Value:	Returns ‘0’ if successful, ‘-1’ if not.
Remarks:	`brk()` is used to dynamically change the amount of space allocated for the calling process's data segment. The change is made by resetting the process's break value and allocating the appropriate amount of space. The break value is the address of the first location beyond the end of the data segment. The amount of allocated space increases as the break value increases. Newly allocated space is uninitialized. This helper function is used by the Standard C Library function `malloc()`.

brk (Continued)

Default Behavior: If the argument *endds* is zero, the function sets the global variable `__curbrk` to the address of the start of the heap, and returns zero.
If the argument *endds* is non-zero, and has a value less than the address of the end of the heap, the function sets the global variable `__curbrk` to the value of *endds* and returns zero.
Otherwise, the global variable `__curbrk` is unchanged, and the function returns -1.
The argument *endds* must be within the heap range (see data space memory map below).



Notice that, since the stack is located immediately above the heap, using `brk()` or `sbrk()` has little effect on the size of the dynamic memory pool. The `brk()` and `sbrk()` functions are primarily intended for use in run-time environments where the stack grows downward and the heap grows upward.
The linker allocates a block of memory for the heap if the `-Wl,--heap=n` option is specified, where `n` is the desired heap size in characters. The starting and ending addresses of the heap are reported in variables `_heap` and `_eheap`, respectively.
For MPLAB C30, using the linker's heap size option is the standard way of controlling heap size, rather than relying on `brk()` and `sbrk()`.

File: `brk.c`

__C30_UART

Description: Constant that defines the default UART.

Include: N/A

Prototype: `int __C30_UART;`

Argument: N/A

Return Value: N/A

Remarks: Defines the default UART that `read()` and `write()` will use for `stdin` (unless a file has been attached), `stdout`, and `stdout`.

Default Behavior: By default, or with a value of 1, UART 1 will be used. Otherwise UART 2 will be used. `read()` and `write()` are the eventual destinations of the C standard I/O functions.

File: N/A

close

Description: Close a file.

Include: None

Prototype: `int close(int handle);`

Argument: *handle* handle referring to an opened file

Return Value: Returns '0' if the file is successfully closed. A return value of '-1' indicates an error.

Remarks: This helper function is called by the `fclose()` Standard C Library function.

Default Behavior: As distributed, this function passes the file handle to the simulator, which issues a close in the host file system.

File: `close.c`

__close_input_file

Description: Close a previously attached file.

Include: None

Prototype: `void __close_input_file(void);`

Argument: None

Remarks: None.

Default Behavior: This function will close a previously attached file and re-attach `stdin` to the simulated UART. This should occur before a reset to ensure that the file can be re-opened.

File: `__close_input_file.c`

__delay32

Description: Produce a delay of a specified number of clock cycles.

Include: None

Prototype: `void __delay32(unsigned long cycles);`

Argument: *cycles* number of cycles to delay.

Remarks: None.

Default Behavior: This function will effect a delay of the requested number of cycles. The minimum supported delay is 11 cycles (an argument of less than 11 will result in 11 cycles). The delay includes the `call` and `return` statements, but not any cycles required to set up the arugment (typically this would be two for a literal value).

File: `__delay32.c`

_exit

Description: Terminate program execution.

Include: None

Prototype: `void _exit (int status);`

Argument: *status* exit status

Remarks: This is a helper function called by the `exit()` Standard C Library function.

Default Behavior: As distributed, this function flushes stdout and terminates. The parameter status is the same as that passed to the `exit()` standard C library function.

File: `_exit.c`

getenv

Description: Get a value for an environment variable

Include: `<stdlib.h>`

Prototype: `char *getenv(const char *s);`

Argument: *s* name of environment variable

Return Value: Returns a pointer to the value of the environment variable if successful; otherwise, returns a null pointer.

Default Behavior: As distributed, this function returns a null pointer. There is no support for environment variables.

File: `getenv.c`

lseek

Description: Move a file pointer to a specified location.

Include: None

Prototype: `long lseek(int handle, long offset, int origin);`

Argument:

handle refers to an opened file

offset the number of characters from the origin

origin the position from which to start the seek. *origin* may be one of the following values (as defined in `stdio.h`):
SEEK_SET – Beginning of file.
SEEK_CUR – Current position of file pointer.
SEEK_END – End-of-file.

Return Value: Returns the offset, in characters, of the new position from the beginning of the file. A return value of '-1L' indicates an error.

Remarks: This helper function is called by the Standard C Library functions `fgetpos()`, `ftell()`, `fseek()`, `fsetpos`, and `rewind()`.

Default Behavior: As distributed, the parameters are passed to the host file system through the simulator. The return value is the value returned by the host file system.

File: `lseek.c`

open

Description: Open a file.

Include: None

Prototype: `int open(const char *name, int access, int mode);`

open (Continued)

Argument:	*name*	name of the file to be opened
	access	access method to open file
	mode	type of access permitted
Return Value:	If successful, the function returns a file handle, a small positive integer. This handle is then used on subsequent low-level file I/O operations. A return value of '-1' indicates an error.	
Remarks:	The access flag is a union of one of the following access methods and zero or more access qualifiers: 0 – Open a file for reading. 1 – Open a file for writing. 2 – Open a file for both reading and writing. The following access qualifiers must be supported: 0x0008 – Move file pointer to end-of-file before every write operation. 0x0100 – Create and open a new file for writing. 0x0200 – Open the file and truncate it to zero length. 0x4000 – Open the file in text (translated) mode. 0x8000 – Open the file in binary (untranslated) mode. The mode parameter may be one of the following: 0x0100 – Reading only permitted. 0x0080 – Writing permitted (implies reading permitted). This helper function is called by the Standard C Library functions `fopen()` and `freopen()`.	
Default Behavior:	As distributed, the parameters are passed to the host file system through the simulator. The return value is the value returned by the host file system. If the host system returns a value of '-1', the global variable errno is set to the value of the symbolic constant `EFOPEN` defined in `<errno.h>`.	
File:	`open.c`	

read

Description:	Read data from a file.	
Include:	None	
Prototype:	`int read(int handle, void *buffer, unsigned int len);`	
Argument:	*handle*	handle referring to an opened file
	buffer	points to the storage location for read data
	len	the maximum number of characters to read
Return Value:	Returns the number of characters read, which may be less than *len* if there are fewer than *len* characters left in the file or if the file was opened in text mode, in which case each carriage return-linefeed (CR-LF) pair is replaced with a single linefeed character. Only the single linefeed character is counted in the return value. The replacement does not affect the file pointer. If the function tries to read at end-of-file, it returns '0'. If the handle is invalid, or the file is not open for reading, or the file is locked, the function returns '-1'.	
Remarks:	This helper function is called by the Standard C Library functions `fgetc()`, `fgets()`, `fread()`, and `gets()`.	
Default Behavior:	As distributed, the parameters are passed to the host file system through the simulator. The return value is the value returned by the host file system.	
File:	`read.c`	

remove

Description:	Remove a file.
Include:	`<stdio.h>`
Prototype:	`int remove(const char *filename);`
Argument:	*filename* file to be removed
Return Value:	Returns '0' if successful, '-1' if unsuccessful.
Default Behavior:	As distributed, the parameters are passed to the host file system through the simulator. The return value is the value returned by the host file system.
File:	`remove.c`

rename

Description:	Rename a file or directory.
Include:	`<stdio.h>`
Prototype:	`int rename(const char *oldname, const char *newname);`
Argument:	*oldname* pointer to the old name *newname* pointer to the new name
Return Value:	Returns '0' if it is successful. On an error, the function returns a non-zero value.
Default Behavior:	As distributed, the parameters are passed to the host file system through the simulator. The return value is the value returned by the host file system.
File:	`rename.c`

sbrk

Description:	Extend the process's data space by a given increment.
Include:	None
Prototype:	`void * sbrk(int incr);`
Argument:	*incr* number of characters to increment/decrement
Return Value:	Return the start of the new space allocated, or '-1' for errors.
Remarks:	`sbrk()` adds *incr* characters to the break value and changes the allocated space accordingly. *incr* can be negative, in which case the amount of allocated space is decreased. `sbrk()` is used to dynamically change the amount of space allocated for the calling process's data segment. The change is made by resetting the process's break value and allocating the appropriate amount of space. The break value is the address of the first location beyond the end of the data segment. The amount of allocated space increases as the break value increases. This is a helper function called by the Standard C Library function `malloc()`.

sbrk (Continued)

Default Behavior: If the global variable `__curbrk` is zero, the function calls `brk()` to initialize the break value. If `brk()` returns -1, so does this function.
If the *`incr`* is zero, the current value of the global variable `__curbrk` is returned.
If the *`incr`* is non-zero, the function checks that the address (`__curbrk` + *`incr`*) is less than the end address of the heap. If it is less, the global variable `__curbrk` is updated to that value, and the function returns the unsigned value of `__curbrk`.
Otherwise, the function returns -1.
See the description of `brk()`.

File: `sbrk.c`

system

Description: Execute a command.

Include: `<stdlib.h>`

Prototype: `int system(const char *s);`

Argument: *`s`* command to be executed.

Default Behavior: As distributed, this function acts as a stub or placeholder for your function. If *`s`* is not NULL, an error message is written to stdout and the program will reset; otherwise, a value of -1 is returned.

File: `system.c`

time

Description: Get the system time.

Include: `<time.h>`

Prototype: `time_t time(time_t *timer);`

Argument: *`timer`* points to a storage location for time

Return Value: Returns the elapse time in seconds. There is no error return.

Default Behavior: As distributed, if timer2 is not enabled, it is enabled in 32-bit mode. The return value is the current value of the 32-bit timer2 register. Except in very rare cases, this return value is not the elapsed time in seconds.

File: `time.c`

write

Description: Write data to a file.

Include: None

Prototype:
```
int write(int handle, void *buffer,
 unsigned int count);
```

Argument:

handle	refers to an opened file
buffer	points to the storage location of data to be written
count	the number of characters to write.

Return Value: If successful, write returns the number of characters actually written. A return value of '-1' indicates an error.

Remarks: If the actual space remaining on the disk is less than the size of the buffer the function is trying to write to the disk, write fails and does not flush any of the buffer's contents to the disk. If the file is opened in text mode, each linefeed character is replaced with a carriage return – linefeed pair in the output. The replacement does not affect the return value.
This is a helper function called by the Standard C Library function `fflush()`.

Default Behavior: As distributed, the parameters are passed to the host file system through the simulator. The return value is the value returned by the host file system.

File: `write.c`

3.18.3 Examples of Use

EXAMPLE 3-1: UART1 I/O

```
#include <libpic30.h>      /* a new header file for
                              these defintions */
#include <stdio.h>

void main() {
  if (__attach_input_file("foo.txt")) {
    while (!feof(stdin)) {
      putchar(getchar());
    }
    __close_input_file();
  }
}
```

EXAMPLE 3-2: USING UART2

```
/* This program flashes a light and transmits a lot of messages at
   9600 8n1 through uart 2 using the default stdio provided
   by MPLAB C30. This is for a dsPIC33F on an Explorer 16(tm) board
   (and isn't very pretty) */

#include <libpic30.h>      /* a new header file for these
                              defintions */
#include <stdio.h>

#ifndef __dsPIC33F__
#error this is a 33F demo for the explorer 16(tm) board
#endif
#inlcude <p33Fxxxx.h>

_FOSCSEL(FNOSC_PRI );
_FOSC(FCKSM_CSDCMD & OSCIOFNC_OFF & POSCMD_XT);
_FWDT(FWDTEN_OFF);

main() {
 ODCA = 0;
 TRISAbits.TRISA6 = 0;
 __C30_UART=2;
 U2BRG = 38;
 U2MODEbits.UARTEN = 1;
 while (1) {
   __builtin_btg(&LATA,6);
   printf("Hello world %d\n",U2BRG);
 }
}
```

Appendix A. ASCII Character Set

TABLE A-1: ASCII CHARACTER SET

Least Significant Character \ Most Significant Character	0	1	2	3	4	5	6	7
0	NUL	DLE	Space	0	@	P	‘	p
1	SOH	DC1	!	1	A	Q	a	q
2	STX	DC2	"	2	B	R	b	r
3	ETX	DC3	#	3	C	S	c	s
4	EOT	DC4	$	4	D	T	d	t
5	ENQ	NAK	%	5	E	U	e	u
6	ACK	SYN	&	6	F	V	f	v
7	Bell	ETB	’	7	G	W	g	w
8	BS	CAN	(	8	H	X	h	x
9	HT	EM	)	9	I	Y	i	y
A	LF	SUB	*	:	J	Z	j	z
B	VT	ESC	+	;	K	[	k	{
C	FF	FS	,	<	L	\	l	|
D	CR	GS	-	=	M	]	m	}
E	SO	RS	.	>	N	^	n	~
F	SI	US	/	?	O	_	o	DEL

NOTES:

Index

D

E

F

G

H

M

N

WORLDWIDE SALES AND SERVICE

AMERICAS

Corporate Office
2355 West Chandler Blvd.
Chandler, AZ 85224-6199
Tel: 480-792-7200
Fax: 480-792-7277
Technical Support:
http://support.microchip.com
Web Address:
www.microchip.com

Atlanta
Duluth, GA
Tel: 678-957-9614
Fax: 678-957-1455

Boston
Westborough, MA
Tel: 774-760-0087
Fax: 774-760-0088

Chicago
Itasca, IL
Tel: 630-285-0071
Fax: 630-285-0075

Dallas
Addison, TX
Tel: 972-818-7423
Fax: 972-818-2924

Detroit
Farmington Hills, MI
Tel: 248-538-2250
Fax: 248-538-2260

Kokomo
Kokomo, IN
Tel: 765-864-8360
Fax: 765-864-8387

Los Angeles
Mission Viejo, CA
Tel: 949-462-9523
Fax: 949-462-9608

Santa Clara
Santa Clara, CA
Tel: 408-961-6444
Fax: 408-961-6445

Toronto
Mississauga, Ontario,
Canada
Tel: 905-673-0699
Fax: 905-673-6509

ASIA/PACIFIC

Asia Pacific Office
Suites 3707-14, 37th Floor
Tower 6, The Gateway
Habour City, Kowloon
Hong Kong
Tel: 852-2401-1200
Fax: 852-2401-3431

Australia - Sydney
Tel: 61-2-9868-6733
Fax: 61-2-9868-6755

China - Beijing
Tel: 86-10-8528-2100
Fax: 86-10-8528-2104

China - Chengdu
Tel: 86-28-8665-5511
Fax: 86-28-8665-7889

China - Fuzhou
Tel: 86-591-8750-3506
Fax: 86-591-8750-3521

China - Hong Kong SAR
Tel: 852-2401-1200
Fax: 852-2401-3431

China - Qingdao
Tel: 86-532-8502-7355
Fax: 86-532-8502-7205

China - Shanghai
Tel: 86-21-5407-5533
Fax: 86-21-5407-5066

China - Shenyang
Tel: 86-24-2334-2829
Fax: 86-24-2334-2393

China - Shenzhen
Tel: 86-755-8203-2660
Fax: 86-755-8203-1760

China - Shunde
Tel: 86-757-2839-5507
Fax: 86-757-2839-5571

China - Wuhan
Tel: 86-27-5980-5300
Fax: 86-27-5980-5118

China - Xian
Tel: 86-29-8833-7250
Fax: 86-29-8833-7256

ASIA/PACIFIC

India - Bangalore
Tel: 91-80-4182-8400
Fax: 91-80-4182-8422

India - New Delhi
Tel: 91-11-4160-8631
Fax: 91-11-4160-8632

India - Pune
Tel: 91-20-2566-1512
Fax: 91-20-2566-1513

Japan - Yokohama
Tel: 81-45-471- 6166
Fax: 81-45-471-6122

Korea - Gumi
Tel: 82-54-473-4301
Fax: 82-54-473-4302

Korea - Seoul
Tel: 82-2-554-7200
Fax: 82-2-558-5932 or
82-2-558-5934

Malaysia - Penang
Tel: 60-4-646-8870
Fax: 60-4-646-5086

Philippines - Manila
Tel: 63-2-634-9065
Fax: 63-2-634-9069

Singapore
Tel: 65-6334-8870
Fax: 65-6334-8850

Taiwan - Hsin Chu
Tel: 886-3-572-9526
Fax: 886-3-572-6459

Taiwan - Kaohsiung
Tel: 886-7-536-4818
Fax: 886-7-536-4803

Taiwan - Taipei
Tel: 886-2-2500-6610
Fax: 886-2-2508-0102

Thailand - Bangkok
Tel: 66-2-694-1351
Fax: 66-2-694-1350

EUROPE

Austria - Wels
Tel: 43-7242-2244-39
Fax: 43-7242-2244-393

Denmark - Copenhagen
Tel: 45-4450-2828
Fax: 45-4485-2829

France - Paris
Tel: 33-1-69-53-63-20
Fax: 33-1-69-30-90-79

Germany - Munich
Tel: 49-89-627-144-0
Fax: 49-89-627-144-44

Italy - Milan
Tel: 39-0331-742611
Fax: 39-0331-466781

Netherlands - Drunen
Tel: 31-416-690399
Fax: 31-416-690340

Spain - Madrid
Tel: 34-91-708-08-90
Fax: 34-91-708-08-91

UK - Wokingham
Tel: 44-118-921-5869
Fax: 44-118-921-5820

12/08/06