SOFTWARE DEVELOPMENT and STANDARDS MANUAL

SLTF CONSULTING

Technology with Business Sense

6316 Distant Rock Path
Columbia, MD 21045
410.799.3915
410.799.5951 Fax
info@sltf.com
www.sltf.com

TABLE OF CONTENTS

INTRODUCTION

This Software Standards Manual sets forth standards for all software development. This Manual's purpose is to ensure that we consistently deliver quality software to our clients while at the same time addressing each client's particular needs and requirements. These rules are not arbitrary, but have been proven over numerous developments to help with the following:

- Improving programming efficiency.
- Allowing the re-use of software functions.
- Improving software documentation.
- Making it possible to modify old programs without spending weeks relearning old code.
- Reducing errors by improving the readability of algorithms.

Programmers are expensive and the goal of any software design should be the reduction in time needed to complete the task. Careful analysis and design of the project are crucial for minimizing the expensive debugging time. By generating structured, well-defined code and algorithms, we can considerably reduce software errors and omissions.

Each client has their own needs and requirements for software documentation, and we will do our best to address these. This Manual sets forth an idealized software development procedure. Not all clients will want this nor can all clients afford the additional development costs incurred by these procedures. Appendix A has a form entitled DEVELOPMENT ACTIVITIES CHECK OFF FORM. This form allows the client to tailor the software development documentation that we do for them, thereby helping them control their costs and helping them to meet their industry's requirements. Throughout this Manual, imperative words such as *must*, are meant for the idealized development case. The CHECK OFF FORM is the controlling document for the type and level of documentation needed for each client's project.

STRUCTURED PROGRAMMING

DEFINITIONS

To allow everyone to speak the same development process language, the following definitions will be used throughout this document:

PROGRAM: A logically complete collection of modules working together to solve a problem.

MODULE: A collection of functions or data implementing one idea or concept. Ideally, a module is one disk file, or if the file is too large, several related files.

FUNCTION: A subroutine or other related program portion that completely expresses one idea or abstraction. Functions are short, sweet, and to the point and should be shorter than one printed page.

INTERFACE: The method used for the function or module to communicate with the rest of the program.

PROGRAM DEVELOPMENT PROCESS

Developers will adhere to the following procedures:

- Talk with the client/end-user and understand what he/she wants!
- Learn his/her terminology, technology, and lingo.
- Agree on or prepare a REQUIREMENTS document.
- Prepare a DESIGN document.
- Review – does the DESIGN document satisfy the given problem? Can the algorithm be improved?
- Code, debug, and test each module separately.
- Integrate the modules together.
- Debug the system.
- Write an operator's manual. This manual must explain how the operator will interact with the software, not necessarily how the designed instrument operates.

Expensive programmers should spend their time developing algorithms, not program code. Algorithm developments make it very easy to allow major

changes, but once you have written code, changing an algorithm is expensive. Essential to the algorithm development is a complete and clear understanding of the problem. During this entire sequence, the programmer will maintain the documentation and at the project's end, all documents will be accurate. The programmer must update the documentation during the development process, not at the end when the program code is finally working.

MODULES

A module is probably one of the most important features found in a program. It groups ideas, expresses the programmer's wishes, and is useful as a memory aid. A module can force organization into a discipline that seems to thrive on chaos. The problem, though, is that most programmers don't know how to write a good module. The following should hopefully lay down some rules that will make it easier to write a good module.

A module groups related functions together into a cohesive unit that expresses the implementation of a single concept or idea. In other words, a module "encapsulates" related functions. An example would be various functions used to maintain a FIFO buffer (see Appendix B). The goal should always be a module that can be reused in other development projects.

The concept of a module sounds easy but in practice it takes a lot of effort and self-discipline. The first question that you have to answer before designing a module is what characteristic(s) will relate these functions together? To answer this question, you must know what the development project needs are. Again using the FIFO example, the chances are very good that if you need one FIFO in a project then there will probably be other needs for it too. The choice here is to either hard-code for one particular FIFO or instead, code for any number of FIFOs that the project might need. The latter technique will allow reusability whereas the former technique restricts your design efforts to just this one project, which is a waste of your time and therefore more expensive to the client.

To continue this example, one way to allow (essentially) an unlimited number of FIFOs in your development is to create function **fifo_open** into which you pass the length of the FIFO and the element size. This function can then allocate the correct amount of space on the heap and will return a pointer to a structure. This structure will contain all the information necessary for other FIFO functions to correctly work with this one particular FIFO. Adding data to this FIFO then becomes a call to function **fifo_add** into which you pass a pointer to the structure and a pointer to the data being added. **fifo_add** again is general purpose and reusable since nothing about any particular FIFO is hard-coded. In the same vein, you can retrieve data from the FIFO through function **fifo_get** into which you pass the pointer to the FIFO structure and a pointer to where the data will go. When the program finishes with a FIFO, a call to **fifo_close** reclaims the heap space.

These functions, **fifo_open**, **fifo_add**, **fifo_get**, **fifo_close**, and maybe some others like **fifo_length**, all belong in one module. These functions all relate to one idea (a FIFO) and all share a common interface (through the FIFO structure).

A module comprises one or more functions with an interface and maybe private data. The inner workings of the module must be hidden from the rest of the program. For example, it's not important to the program how the module handles the FIFOs. If you need a particular feature of the FIFO and it's not available, you must either add it to the module or find another way of solving the problem. This takes discipline but the net result is an encapsulated idea, any problems or changes being isolated to this one module, and a module that has a much better chance of being reused.

One pitfall to avoid is designing the module for one particular application or project. Every time you design a module, ask yourself if there is any possibility that this module, or any parts of it, could ever be used elsewhere. If it could, plan for it. If a module doesn't include the I/O for the particular project, there's a much better chance that you might be able to reuse the module. I/O is very project specific but a module such as FIFO couldn't care less about it since it doesn't use I/O. However, a module named ICON might be very dependent on the I/O. Isolate the I/O dependent sections, even within a module. It's much better to have 95% of a module reusable because of the I/O dependent sections being isolated than having to change 100% of the module because the I/O on a new project is different. This philosophy might sound wasteful of computer memory and execution speed but these are now so cheap compared to software costs that they shouldn't always be the prime driving force.

So how do you force a program to use your beautifully crafted module? Headers. Create for each module (or if necessary, multiply related modules) a header that specifies the interface to the modules and any particular data structures. For FIFO, this includes the prototype definitions for each globally accessible function and the definition of the data structure. Any functions that need a FIFO will include this header interface definition. This header must be the sole method of interfacing into the module; if it's not then the module is not encapsulated nor reusable, and was a waste to design. Programmer's time is extremely expensive; use it wisely and don't look for shortcuts. As Pippin said, "Short cuts make long delays."

If a module's interface header should change (no programmer's perfect, right?), this mustn't cause you to recompile the entire program just to make sure you catch every dependent module. In the MAKE file, include the dependencies of each module's interface. This way, if an interface should change, only those modules using the particular interface will be recompiled, not all the modules, thereby saving an enormous amount of time. Some

language environments can keep track of these interdependencies automatically – use it.

One important feature of a module is its length. Longer is not better. Compilation time is expensive; the longer the module, the longer it takes to compile. Most program changes that require a recompilation of a module are for minor changes, not involving lots of code lines. Try to restrict your modules to 400 lines of code. If the module needs to be longer, split it into multiple modules, all with a related name. If the FIFO module was 1,000 lines long (for whatever reason), split it into three modules, FIFO1, FIFO2, and FIFO3. The sum of the parts will be more than the whole because of the interface header files required, but this will create smaller modules with quicker compilations, without sacrificing the integrity of the module concept. You wouldn't type the entire program in each time you needed to do a change to FIFO so why force the compiler to work harder than it has to?

FUNCTIONS

Functions are the backbone of every program. This is where all the work within the program takes place. Poorly coded functions create nothing but headaches, both during debugging and during the maintenance phase. The following are the general rules that you must follow when designing functions:

- A function implements one idea or abstraction.
- Its algorithm should be obvious from the program code and header description.
- Each function must begin with the standard header.
- Each function must contain complete documentation in the code by the frequent use of comments.

Anyone can write functions but what sets excellent program code apart are good, well-crafted functions. The following items, though not complete, contribute towards a "good" function:

- Algorithms are simple and easily understood from the code.
- Sensible function and variable names.
- Each function should, if possible, occupy no more than one page in the listing.
- Keep compound conditional statements to a minimum.
- Frequent use of comments within the program code.

- Keep global data to a minimum. Let the function restrict access to the data.

- One executable program statement on each line.

For the sake of conformity and to make it easier for someone else to quickly understand the program code, we need to have some constraints made on variable names. These constraints aren't meant to interfere with the programmer's task or creativity but are instead needed for helping comprehend the program at a later time or by someone else:

- Simple incrementers used within loops shall use the FORTRAN integer letters i, j, k, l, m, and n. If you need something other than a simple variable, give it an appropriate name.

- Use the variables x, y, and z as temporary floating point variables. Again, if a more descriptive name makes more sense, use it.

- In graphics work, x and y can be used to indicate a particular pixel location and used as a loop incrementer.

DOCUMENTATION

Self-documenting code is one of the objectives. However, you must maintain clear and up-to-date documentation for all projects. The documentation's purpose is to prevent confusion and misunderstanding in what is being developed. Even though this is a laudable goal, some companies will not want to spend the money for a formal documentation process. As such, you must address each client's individual needs in addition to your company's needs. The check-off sheet in Appendix A will outline the documentation needed for each client.

Some clients will have their own documentation procedures and of course we will have to follow their requirements. That does not mean that you can ignore this document though. Work with the client to agree on a mutually acceptable solution that encompasses their documentation needs with your company's developmental needs.

The following outlines the preferred procedure for documenting projects.

REQUIREMENTS DOCUMENT

Before you can start any software project, both the software development manager and the end-user (e.g., client) must agree on a REQUIREMENTS DOCUMENT. Ideally, the end-user creates this document but in most instances it ends up being developed by the software designer for the end-user. On the basis of the initial discussion(s) with the end-user, the software developer will generate this document describing his/her perceived understanding of the task. The software developer will then submit this document to the end-user for review. The end-user's comments will be incorporated into a new release of the REQUIREMENTS DOCUMENT and resubmitted to the end-user. This process will be repeated until both the end-user and the software development manager have agreed on the document, that is, they have agreed on what's being designed.

The REQUIREMENTS DOCUMENT is ideally a list of requirements that the design must meet. A narrative document is OK, but you still must have some easy mechanism to pull the requirements out from the document. A list of the salient features might suffice for this. Each requirement should have some place for you to check off that the design has met the requirement.

DESIGN DOCUMENT

Once the end-user has accepted the REQUIREMENTS DOCUMENT, the software developer must create the DESIGN DOCUMENT detailing how he/she will design the software to achieve the requirements contained in the

REQUIREMENTS DOCUMENT. The DESIGN DOCUMENT should list all inputs and outputs, algorithms, formulas, high level functionality, abstractions, and user interface.

Whenever possible, you should include sample screens, and user control and interfacing to the proposed design with the DESIGN DOCUMENT. A suitable substitute is a prototyping program running on a PC. The screens should demonstrate how the end-user will interact with the design and how the program will present data to the end-user. The goal again is that the end-user must know, before starting any design, what is being designed and that he/she agrees that your design is the best fit for the problem. If you prepare the REQUIREMENTS DOCUMENT, you can instead include this section on the user interface in that document.

Most of this document is not relevant to the end-user but you must circulate it for his/her approval.

SYSTEM TEST DOCUMENT

While designing the DESIGN DOCUMENT, the software developer must also create a SYSTEM TEST DOCUMENT. This document will detail how the software developer will test each module and the entire system. Included also will be the expected actions of each key press and each input signal, including analog signals. You must also list any specific timing constraints. List also any test equipment needed to verify the integrity of the software. Wherever possible, make the system testable by non-programmers. You must show the SYSTEM TEST DOCUMENT to the end-user and also get their approval. The module testing might not be of much use to him/her but the system test section will be the acceptance criteria for the end-user.

GENERAL PROJECT DOCUMENTATION

Once you have defined the modules, you must maintain a module list to aid management in assessing the status of a project. This is a standard form (see Appendix A). When you have coded and debugged a module, place a check in the appropriate column.

Keep all source code for a project in its own project subdirectory on your disk. Don't combine different projects within the subdirectory. If a project comprises multiple computers, each computer's software will be in its own subdirectory under the project's umbrella subdirectory. For example, if a project has two computers, you might label the directories \PROJECT\CPU1 and \PROJECT\CPU2. In other, simpler words, Organize! Organize! Organize! Make sure that the disk structure makes sense today and six months in the future.

All PROMs, disks, and other media will have version numbers. Version numbers will be of the form X.YY. YY will increment by 01 with each new minor revision. Increment X by 1 and set YY to 00 for each major change.

The software developer will maintain on disk a version list summary for each project. This summary will list the version number, date of change, a short description of the change, the modules changed, and the programmer's initials. The version number should be an ASCII string that is automatically compiled into the program code if the number changes. Figure 2 shows a sample header file. All systems with any type of display will indicate the program version number on power-up or program initiation.

```
#undef VERSION
#define VERSION "Version 1.23"  /* 09/22/1992 SBR
1. Changed calibration so that in DEMO mode the
   tonometer heater is not turned on.
2. Changed screen print to add patient ID and case number
   to
   each printout.
*/


#undef VERSION
#define VERSION "Version 1.24"  /* 10/01/1992 SBR
1. Changed constant storage and retrieval to speed it up.
2. Fix - problem with possible two cursor lines was fixed in
   draw_it_task.
3. Sped up the time to go from cursor back to trending mode
   as a consequence of 2 above.
*/
```

Figure 1 – Sample version history list

All documentation must be neat, in standard English, and must convey the information such that someone not familiar with the project can read the documentation and understand how to change the software. It is not acceptable to have poorly worded descriptions, spelling errors, grammatical errors, or anything else that would be an embarrassment to an educated person.

MODULE DEFINITION

Each module will include a copyright notice at the module beginning using the format in Figure 2. If the end-user is your company, fill in the copyright line with your company's legal name. This must be the first item in each module.

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

        COPYRIGHT 199_  (end-user name)

        DEVELOPED BY:    [your company name]
                         [your company address]
                         [your company phone]

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

Figure 2 – Copyright notice

Below the copyright message, each module will have a header describing the higher-level functionality of the module. For example, a module concerning interfacing to a monitor screen might consist of functions that deal with the cursor, screen writing and reading, and character attributes. You must list here, too, any special conditions, files, or environments needed to use this module. Obviously, if the module is a header, global data, or of a similar type, you will not need all the above information.

FUNCTION DEFINITION

Each function must have a standard header introducing it of the form as shown in Figure 3. The function description can be multi-line and in most situations it will be.

```
/************************************************
Routine Name:
Author:
Date Created:
Description:
Input:
Output:
************************************************/
```

Figure 3 – Function description

CODE DOCUMENTATION

All program code will include a reasonable number of comments. If the selected language allows, comments should be at the end of the program line or the program lines can be split by an introductory comment. It should be possible to follow the program by reading the comments. Do not continue comments at ends of program lines onto the next line of code (no running narratives).

TESTING

There are two sections to program testing. The first is testing of each module in the program. This is similar to conventional debugging. The second is testing of the complete program or instrument. This tests not only how well the modules behave together but it also tests how well the software fits the application.

MODULE TESTING

It is absolutely essential that after the software developer codes each module, that he/she debugs each module. Don't wait until you code all the modules before testing them – test each module as you write them. If you wait, an oversight in one module might have a ripple effect throughout all the other coded modules requiring a lot of effort compared to coding and testing each module separately. If you follow the encapsulation techniques mentioned earlier, the testing should be easy and straightforward.

The software developer must test each module before incorporating it into the program. Sometimes it is not possible for you to test a module stand-alone. In this case, you must identify a technique that unequivocally tests the module. You may also test a module using a "test" program to verify its functionality, or you can check it with a software debugger for proper operation. Whichever route you take, you must document the testing technique and the applicable test results. If you generate a separate test program, you must save this as part of the documentation.

Independent of the language translator (assembler or compiler, and called compiler from now on), you must evaluate the compiler during initial system development if it has not been used before. For the first 500 lines of source code produced by the programmer, the programmer will evaluate the output of the compiler (and optimizer) for correct translations. In addition, the programmer will obtain from the compiler's manufacturer a list of all known bugs within the compiler and will, if at all possible, have his/her name on a list to receive all future bug reports. All bugs that the programmer finds and that are not on the bug report will be listed on an internal bug report, including ways to work around the problem. The programmer will send this internal bug report list to the compiler's manufacturer in a timely manner. If the manufacturer releases a new version of the compiler, you must check this new version for errors and compatibility with the older version. Occasionally, manufacturers release new versions that have more errors than what is being replaced.

Assembly language programs have special concerns since the programmer has complete control over all the computer's resources. There must be a convention as to which registers are saved and which can be destroyed within each module. You must also define stack usage (and, if applicable, heap

usage). Throughout the program, you must follow these conventions. When testing these modules, you must check the input and output conditions (registers, flags, etc.) with a debugger. Make sure the stack stays balanced!

All the module names will be entered into the MODULE LIST form, including the file it resides in, a brief description, the date it was tested, and how it was tested.

As you test and make each module work, enter the testing date and test method for each module in the Module List. If while testing a module you find a previously undetected error, fix the module, modify the testing procedure, and then retest the module. Once you test a module, don't change it. If you do change it, you need to retest it.

LINT TESTING

LINT is a standard utility program that identifies questionable sections in source code (at least in C and C++). It does this by looking at all the source code and header files, and identifying common programming mistakes, oversights, and questionable construct. It will not necessarily find program "bugs." Typical examples of what LINT finds includes uninitialized static variables, unused variables, and unintentional endless loops. An analogy for LINT is using a spelling checker – it can find spelling mistakes but it cannot find if you are using a wrong word, like *than* instead of *then*. You can easily tailor the LINT program to a specific coding style so that it does not log unrealistic problems.

The LINT procedure is an iterative process. It is not worth running LINT and then not changing the source code to reflect what it found. On the other hand, just because LINT says that something is questionable, this does not make it necessary to change the source code. The programmer knows the code better than LINT and therefore the programmer's intuition must always win out.

LINT is run not only on a module, but you also run it on the entire program. Unlike a compiler, LINT can find problems and inconsistencies across modules.

CODE REVIEW

After LINT evaluates the program source code, the next step is a Code Review by someone other than the programmer. A different programmer performs the Code Review, who looks at the source code in more detail than LINT can. The reviewer is looking, for example, for bad constructs, poor structure, ambiguity, commenting, and even poor algorithm implementation.

The reviewer will log all comments, suggestions, and problems either on a standard form or in a database. The reviewer will not amend or change the source code in any way. It is also not necessary to print out the entire program. The reviewer can do the review using a PC screen.

At the end of the code review, the reviewer will give all the forms to the programmer.

During the course of the code review, the reviewer may wish to test, check, or verify the operation of a particular function or module. The reasons for this might be because of code ambiguity, algorithmic questions, or just as a spot check. It is not necessary for the reviewer to explain why he/she tested a section of code. The reviewer will log any testing methods that he/she does and the results of the test.

Sometimes the reviewer will identify a section for testing but cannot devise a proper test. The reviewer will also log this information.

After the reviewer finishes the code review and any testing, he/she will give the results to the programmer. The programmer will then either fix the source code or not. Whatever the programmer decides to do must be logged onto each of the forms.

After the programmer finishes with the forms and changing the source code, the programmer gives the forms back to the reviewer. The reviewer will review the changes and make sure that the programmer has satisfactorily resolved or explained the questionable areas. The reviewer will again log onto the forms any additional comments concerning the source code. In other words, this is an iterative process where the iterations are finished when the source code satisfies the reviewer. Hopefully this should not take more than three iterations.

SYSTEM TESTING

Besides individually testing each module, you need to test the entire program to make not only sure that all the modules function together properly but that the software and instrument meet the needs of the end-user. Before performing system testing, you must create a SYSTEM TEST DOCUMENT. You must then test the system according to this document to prove that the system works. When the system passes its testing, only then can you deliver it to the end-user. Remember during the system testing that things other than software problems can make a system appear to fail its tests. For example, electronic problems can yield noisy results as can a poor A/D reading function. Different data fed into a mathematical function might only work sometimes, especially with regression algorithms. Sure, software a lot of times has bugs but keep an open mind and realize that other parameters, especially in a jury-rigged or simulated environment, might not truly represent the intended application.

SOFTWARE CHANGE CONTROL PROCEDURE

Good software engineering controls requires that each programmer follow a Change Control Procedure. However, the Change Control Procedure does take time and some clients, being extremely strapped for cash or short-sighted, might elect not to use a Change Control Procedure. Additionally, some clients might require a modified Change Control Procedure, either to conform to their internal specifications or to their industry's requirements. You must serve the clients' needs and you can either adapt or try to guide your clients, but at the same time you cannot force a client to use these procedures. The following outlines a proposed standard procedure.

INITIATION

Changes can have many names including bug fixes, modifications, and improvements. The thing all these labels have in common is that, for some reason, the source code must change. Programmer's will not change any client's source code without the client's approval.

Communication with the client is normally very good. Because of this closeness, many times changes come verbally from the client. Obtaining written authorization from the client might be next to impossible and it can even be interpreted as our lack of trust in the client. Therefore, if the changes do come verbally, it is imperative that you and the client have an extremely clear understanding of what is required. If there is any chance of a misunderstanding, send the client a quick note stating the proposed changes. Do your best to avoid future problems!

CHANGE CONTROL DOCUMENT

We use a database for recording all changes to clients' software. The database's name is MUMS.MDB and runs with Microsoft Access under the Windows operating system. This one database is for all of our clients and is thus proprietary to your company. Clients can obtain copies of the database but the copy must only contain records relating to that one client.

The Control Document (form) is the top level document. On this document you record the client's name, the product name, and the version number of the program. In addition, you need to record what functionality you are changing in the program. The functionality is the program fix, change, or modification. This could be very broad such as "Change all messages to another language," or as specific as "Change the constant to 12.3." Remember, this is a top level document. The Detail Document will take care of each change made to the program.

Also on this document is a testing section. Within this section you need to record what testing you need to do to verify that the software changes work and that you have met the functionality described above. After you finish the testing, record on the document that you did the testing so that someone else looking at this document knows what was done. Obviously, if you tested the software change and if it did not work, you, as a professional programmer, would not allow the software to leave. However, not everyone else understands professional programmers and therefore, state the obvious.

Each Control Document has automatically assigned to it a Control Number. This is a unique number that identifies one specific functionality change. This is the number that clients' should refer to when they need further information.

CHANGE CONTROL DETAIL DOCUMENT

Programs are composed of many different parts, including modules, functions, headers, data, and comments. You need to record on the Detail Document each change you make to the program for the modification outlined in the Control Document. Use one Detail Document for each part of the program being changed. This therefore means that with each Control Document there can be many Detail Documents. Make sure that you put your name and the change date on each Detail Document.

REPORTS

All these database entries are done on the computer using forms within Access. Occasionally, a client might want a paper trail detailing the changes done for their product. The MUMS database has reporting capability and you can print a standard report. Just select the Control Numbers to print and the program will automatically print them out. These copies can go to the client either by fax or by mail.

APPENDIX A: DOCUMENTS

MODULE LIST

CLIENT: ____________________ DATE: ____________________

PROGRAM: ____________________ INITIALS: ____________

INSTRUMENT: ________________

Module Name	File Name	Description	Coded	Debugged

DEVELOPMENT ACTIVITIES CHECK OFF FORM

CLIENT OPTIONAL	REQUIRED
☐ Requirements Document	☑ Understand What The Client Wants
☐ Design Document	☑ Learn the Client's Lingo
☐ Module List	☑ Good Module Practices
☐ Operator's Manual	☑ Module Definitions
☐ Module Test Record	☑ Good Function Practices
☐ LINT Cleanup	☑ Function Definitions
☐ System Test Document	☑ Code Documentation
☐ Change Control Procedure	☑ Debug The System
	☑ Code Review

Comments:

Client: _____ Date: _____

SOFTWARE ENGINEERING
CONTROL DOCUMENT

Control No. 00001

Page 1 of –

Client

Product

Version No.

Client Change No.

Type ○ New ○ Mod ○ Fix

Description

Test Procedure

Test Passed

Date

SOFTWARE ENGINEERING
CONTROL DETAIL

Control No. 00001

Page 2 of _

Module

Name

Type ○ Function ○ Data ○ Comment ○ Header

Status ○ New ○ Changed

Description

Programmer **Date**

APPENDIX B: SAMPLE FIFO MODULE

```
#ifndef __FIFO_H
#define __FIFO_H

/* fifo structure */

typedef struct {
  void *start;                  /* fifo data storage start location */
  void *temp;                   /* temporary buffer for returning data */
  int head;                     /* index to where new data goes */
  int tail;                     /* index to where oldest data comes from */
  int full;                     /* fifo full flag */
  int length;                   /* fifo maximum length */
  int size;                     /* fifo element size in bytes */
} FIFO;

/* fifo.c */
FIFO *fifo_open(int length, int size);
void fifo_close(FIFO *ptr);
void fifo_reset(FIFO *ptr, void *data);
void *fifo_read(FIFO *ptr);
void *fifo_write(FIFO *ptr, void *in);
int fifo_length(FIFO *ptr);

#endif
```

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

                    Copyright 1999 SLTF Consulting

                    Developed By:  SLTF Consulting
                                   Columbia, Maryland  21045
                                   (410) 799-3915

*************************************************************************/
#include <stdlib.h>
#include "fifo.h"

#define YES   1
#define NO    0
#define NULL  ((void *)0)
#define EOF   -1
typedef unsigned char BYTE;

/*************************************************************************
  Routine Name:    fifo_open
        Author:    Scott B. Rosenthal
  Date Created:    August 8, 1991
   Description:    This is called before the fifo can be used so that the structure for saving all the
                   fifo data can be set up.  This will return a pointer to the saved fifo data.  The
                   minimum fifo length is 1.  If the fifo is not needed anymore, use fifo_close to
                   remove the fifo data area from the heap.
         Input:    length -  The number of elements for the fifo to contain.  The minimum length
                   is 1.
                   size -    The size, in bytes, of each element in the fifo.
        Output:    pointer of type FIFO to the structure.  If NULL, there's not enough memory to
                   create this structure.
-------------------------------------------------------------------------------
                        Modification History
Modified By:            Date:                   Changes Implemented:

*************************************************************************/

FIFO *fifo_open(int length, int size)

{
  BYTE c[sizeof(double)];
  FIFO *ptr;

/* get memory for the fifo structure */

  if ( ((char *)ptr = malloc(sizeof(FIFO))) == NULL)
    return NULL;                /* return NULL if no memory available */

/* get memory for the fifo array */

  if ( ((char *)ptr->start = malloc(size * length)) == NULL) {
    free(ptr);                  /* return the memory for the structure */
    return NULL;                /* return NULL since no memory is available */
  }
```

```
/* get memory for the temporary buffer in the fifo structure */

  if ( ((char *)ptr->temp = malloc(size)) == NULL) {
    free(ptr->start);                 /* return the memory for the structure */
    free(ptr);                        /* return the memory for the structure */
    return NULL;                      /* return NULL since no memory is available */
  }

/* initialize the data in the structure */

  ptr->length = length;               /* initialize the fifo length */
  ptr->size = size;                   /* initialize the fifo size */
  memset(c, 0x00, sizeof(c[0]));      /* clear the buffer */
  fifo_reset(ptr, c);                 /* initialize the rest of the fifo data */
  return ptr;                         /* return a pointer to the structure */
}


/*************************************************************************
  Routine Name:    fifo_close
        Author:    Scott B. Rosenthal
   Date Created:   August 8, 1991
    Description:   This is called whenever a fifo is not needed anymore.  This will remove the fifo
                   information from the heap.  The argument is a pointer to the fifo information.
         Input:    ptr -  pointer to type FIFO.  This is the FIFO to close.
        Output:    None
-------------------------------------------------------------------------------------------
                        Modification History
Modified By:            Date:                   Changes Implemented:

*************************************************************************/

void fifo_close(FIFO *ptr)

{
  free(ptr->temp);                    /* free the temporary storage area */
  free(ptr->start);           /* free the data storage area */
  free(ptr);                          /* free the fifo structure */
}


/*************************************************************************
  Routine Name:    fifo_reset
        Author:    Scott B. Rosenthal
   Date Created:   August 8, 1991
    Description:   This will reset all the flags in the fifo structure and will initialize the data array
                   to 0x00
         Input:    ptr -  pointer to type FIFO.  This is the FIFO to reset.
                   in  - pointer to the data to fill the buffer with
        Output:    None
-------------------------------------------------------------------------------------------
                        Modification History
Modified By:            Date:                   Changes Implemented:

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

```
void fifo_reset(FIFO *ptr, void *in)

{
  BYTE *dptr;
  int i;

  ptr->head = 0;              /* point to the beginning of the array */
  ptr->tail = 0;                      /* point to the beginning of the array */
  ptr->full = NO;             /* the fifo is not full yet */
  dptr = ptr->start;                  /* point to the beginning */
  for (i = 0; i < ptr->length; i++) {  /* set the buffer to a known value */
    memcpy(dptr, in, ptr->size);     /* put in the newest data */
    dptr += ptr->size;
  }
}


/*************************************************************************
  Routine Name:    fifo_length
        Author:    Scott B. Rosenthal
   Date Created:   August 8, 1991
    Description:   This will return the current length of the fifo.  Pass the fifo structure pointer.
                   Be careful, the returned length can be 0.
         Input:    ptr -  pointer to type FIFO.  This is the FIFO to get the length of.
        Output:    None
-------------------------------------------------------------------------------------------
                          Modification History
Modified By:              Date:                    Changes Implemented:

*************************************************************************/

int fifo_length(FIFO *ptr)

{
  int i;

  if (ptr->full == YES) {    /* if the buffer is full */
    i = ptr->length;
  } else {
    i = ptr->head - ptr->tail;          /* get length of data in buffer */
    if (i < 0)                          /* if a wrap-around situation */
      i = ptr->length + i;
  }
  return i;                             /* return the fifo length */
}


/*************************************************************************
  Routine Name:    fifo_write
        Author:    Scott B. Rosenthal
   Date Created:   August 8, 1991
    Description:   This will write the data into the fifo data array if there's room.  If the array is full
                   and new data is still allowed to be written (stop_on_full), the data being
                   overwritten is returned.
```

```
            Input:    ptr -  pointer to type FIFO.  This is the FIFO to put the data in.
                      in  - pointer to the data to be saved in the fifo array.
           Output:    A pointer with the following possible values:
                      EOF - the array is full and no more data can be written into it.
                      NULL- The data was written into the array but the array wasn't full.
                      ??? - A pointer to the data that was bumped out of the array since new data was
                      being written into a full fifo.
-------------------------------------------------------------------------------------
                          Modification History
Modified By:              Date:                    Changes Implemented:

*************************************************************************/

void *fifo_write(FIFO *ptr, void *in)

{
  int full;
  BYTE *dptr;

  if ((full = ptr->full) == YES) {     /* save if full to begin with */
    return (void *)EOF;    /* nothing could be stored */
  }
  dptr = (BYTE *)ptr->start + (ptr->head * ptr->size);
  memcpy(dptr, in, ptr->size);       /* put in the newest data */
  if ( (ptr->head = ++ptr->head % ptr->length) == ptr->tail)
    ptr->full = YES;                 /* fifo is full now */
  return (full == YES) ? ptr->temp : NULL;          /* return pointer to the dropped data */
}


/*************************************************************************
  Routine Name:    fifo_read
        Author:    Scott B. Rosenthal
   Date Created:   August 8, 1991
    Description:   This will return the oldest data point from the fifo array.  If the array is empty,
                   an EOF pointer is returned.
            Input: ptr -  pointer to type FIFO.  This is the FIFO to get data from.
           Output: pointer to the data being retrieved.  If EOF, no data was available.
-------------------------------------------------------------------------------------
                          Modification History
Modified By:              Date:                    Changes Implemented:

*************************************************************************/
void *fifo_read(FIFO *ptr)

{
  BYTE *dptr;

  if (ptr->full == NO && ptr->tail == ptr->head)
    return (void *)EOF;    /* nothing in buffer to return */
  dptr = (BYTE *)ptr->start + (ptr->tail * ptr->size);
  memcpy(ptr->temp, dptr, ptr->size);       /* copy the oldest data */
  ptr->tail = ++ptr->tail % ptr->length;
  ptr->full = NO;            /* can't be full if we took one out */
  return ptr->temp;                  /* return a pointer to the data */
```