A Firmware Development Standard

John Breitenbach
john@atlantex-usa.com

Atlantex Corp.
156 Dwight Avenue
Hillsdale, NJ 07642
(201) 664-3445
fax (201) 664-5840

Table of Contents

Scope

This document defines the standard way all programmers will create embedded firmware. Every programmer is expected to be intimately familiar with the Standard, and to understand and accept these requirements. All consultants and contractors will also adhere to this Standard.

The reason for the Standard is to insure all Company-developed firmware meets minimum levels of readability and maintainability. Source code has two equally-important functions: it must *work*, and it must clearly *communicate how it works* to a future programmer or the future version of yourself. Just as a standard English grammar and spelling makes prose readable, standardized coding conventions ease readability of one's firmware.

Part of every code review is to insure the reviewed modules and functions meet the requirements of the Standard. Code that does not meet this Standard will be rejected.

We recognize that no Standard can cover every eventuality. There may be times where it makes sense to take exception to one or more of the requirements incorporated in this document. Every exception must meet the following requirements:

- *Clear Reasons* - Before making an exception to the Standard, the programmer(s) will clearly spell out and understand the reasons involved, and will communicate these reasons to the project manager. The reasons must involve clear benefit to the project and/or Company; stylistic motivations, or programmer preferences and idiosyncrasies are not adequate reasons for making an exception.
- *Approval* - The project manager will approve all exceptions made
- *Documentation* - The effected module or function will have the exception clearly documented in the comments, so during code reviews and later maintenance the current and future technical staff understand the reasons for the exception, and the nature of the exception.

Projects

Directory Structure

To simplify use of a version control system, and to deal with unexpected programmer departures and sicknesses, every programmer involved with each project will maintain identical directory structures for the source code associated with the project.

The general "root" directory for a project takes the form:

`/projects/company/project-name/`

where

- "`/projects`" is the root of all firmware developed by the Company. By keeping all projects under one general directory version control and backup is simplified; it also reduces the size of the computer's root directory.
- "`/company`" is the name of client's company
- "`/project-name`" is the formal name of the project under development.

Required directories:

`/projects/company/project-name/tools` - compilers, linkers, assemblers used by this project. All tools will be checked into the VCS so in 5 to 10 years, when a change is required, the (now obsolete and unobtainable) tools will still be around. It's impossible to recompile and retest the project code every time a new version of the compiler or assembler comes out; the only alternative is to preserve old versions, forever, in the VCS.

`/projects/company/project-name/headers` - all header files, such as .h or assemble include files, go here.

`/projects/company/project-name/source` - source code. This may be further broken down into header, C, and assembly directories. The MAKE files are also stored here.

`/projects/company/project-name/object` - object code, including compiler/assembler objects and the linked and located binaries.

`/projects/company/project-name/test` - This directory is the one, and only one, that is not checked into the VCS and whose subdirectory layout is entirely up to the individual programmer. It contains work-in-progress, which is generally restricted to a single module. When the module is released to the VCS or the rest of the development team, the developer must clean out the directory and eliminate any file that is duplicated in the VCS.

Version File

Each project will have a special module that provides firmware version name, version date, and part number (typically the part number on the ROM chips). This module will list, in order (with the newest changes at the top of the file), all changes made from version to version of the released code.

Remember that the production or repair departments may have to support these products for years or decades. Documentation gets lost and ROM labels may come adrift. To make it possible to

correlate problems to ROM versions, even after the version label is long gone, the Version file should generate only one bit of "code" - a string that indicates, in ASCII, the current ROM version. Some day in the future a technician - or yourself! - may then be able to identify the ROM by dumping the ROM's contents. An example definition is:

```
# undef VERSION
# define VERSION "Version 1.30"
```

Example:

```
/*************************************************
* Version Module - Project SAMPLE
*
* Copyright 1997 Company
* All Rights Reserved
*
* The information contained herein is confidential
* property of Company. The use, copying, transfer or
* disclosure of such information is prohibited except
* by express written agreement with Company.
*
* 12/18/97 - Version 1.3 - ROM ID 78-130
*           Modified module AD_TO_D to fix scaling
*           algorithm; instead of y=mx, it now
*           computes y=mx+b.
* 10/29/97 - Version 1.2 - ROM ID 78-120
*           Changed modules DISPLAY_LED and READ_DIP
*           to incorporate marketing's request for a
*           diagnostics mode.
* 09/03/97 - Version 1.1 - ROM ID 78-110
*           Changed module ISR to properly handle
*           non-reentrant math problem.
* 07/12/97 - Version 1.0 - ROM ID 78-100
*           Initial release
*************************************************/
# undef VERSION
# define VERSION "Version 1.30"
```

Make and Project Files

Every executable will be generated via a MAKE file, or the equivalent supported by the tool chain selected. The MAKE file includes all of the information needed to automatically build the entire ROM image. This includes compiling and assembling source files, linking, locating (if needed), and whatever else must be done to produce a final ROM image.

An alternative version of the MAKE file may be provided to generate debug versions of the code. Debug versions may include special diagnostic code, or might have a somewhat different format of the binary image for use with debugging tools.

In integrated development environments (like Visual C++) specify a PROJECT file that is saved with the source code to configure all MAKE-like dependencies.

In no case is any tool *ever* to be invoked by typing in a command, as invariably command line arguments "accumulate" over the course of a project… only to be quickly forgotten once version 1.0 ships.

Startup Code

Most ROM code, especially when a C compiler is used, requires an initial startup module that sets up the compiler's runtime package and initializes certain hardware on the processor itself, including chip selects, wait states, etc.

Startup code generally comes from the compiler or locator vendor, and is then modified by the project team to meet specific needs of the project. It is invariably compiler- and locator-specific. Therefore, the first modification made to the startup code is an

initial comment that describes the version numbers of all tools (compiler, assembler, linker, and locator) used.

Vendor-supplied startup code is notoriously poorly documented. To avoid creating difficult-to-track problems, *never* delete a line of code from the startup module. Simply comment-out unneeded lines, being careful to put a note in that you were responsible for disabling the specific lines. This will ease re-enabling the code in the future (for example, if you disable the floating point package initialization, one day it may need to be brought back in).

Many of the peripherals may be initialized in the startup module. Be careful when using automatic code generation tools provided by the processor vendor (tools that automate chip select setup, for example). Since many processor boot with RAM chip selects disabled, always include the chip select and wait state code in-line (not as a subroutine). Be careful to initialize these selects at the very top of the module, to allow future subroutine calls to operate, and since some debugging tools will not operate reliably until these are set up.

Stack and Heap Issues

Always initialize the stack on an *even* address. Resist the temptation to set it to a odd value like 0xffff, since on a word machine an odd stack will cripple system performance.
Since few programmers have a reasonable way to determine maximum stack requirements, always assume your estimates will be incorrect. For each stack in the system, make sure the initialization code fills the entire amount of memory allocated to the stack with the value 0x55. Later, when debugging, you can view the stack and detect stack overflows by seeing no blocks of 0x55 in that region. Be sure, though, that the code that fills the stack with 0x55 automatically detects the stack's size, so a late night stack size change will not destroy this useful tool.

Embedded systems are often intolerant of heap problems. Dynamically allocating ad freeing memory may, over time, fragment the heap to the point that the program crashes due to an inability to allocate more RAM. (Desktop programs are much less susceptible to this as they typically run for much shorter periods of time).

So, be wary of the use of the malloc() function. When using a new tool chain examine the malloc function, if possible, to see if it implements garbage collection to release fragmented blocks (note that this may bring in another problem, as during garbage collection the system may not be responsive to interrupts). *Never* blindly assume that allocating and freeing memory is cost- or problem-free.

If you chose to use malloc(), *always* check the return value and safely crash (with diagnostic information) if it fails.

When using C, if possible include Walter Bright's MEM package (www.snippets.org/mem.txt) with the code, at least for the debugging.
MEM provides:

- ISO/ANSI verification of allocation/reallocation functions
- Logging of all allocations and frees
- Verifications of Frees
- Detection of pointer over- and under-runs.
- Memory leak detection
- Pointer checking
- Out of memory handling

Modules

General

A *Module* is a single file of source code that contains one or more functions or routines, as well as the variables needed to support the functions.

Each module contains a number of *related* functions. For instance, an A/D converter module may include all A/D drivers in a single file. Grouping functions in this manner makes it easier to find relevant sections of code, and allows more effective encapsulation.

Encapsulation - hiding the details of a function's operation, and keeping the variables used by the function local - is absolutely essential. Though C and assembly language don't explicitly support encapsulation, with careful coding you can get all of the benefits of this powerful idea as do people using OOP languages.

In C and assembly language you can define all variables and RAM inside the modules that use those values. Encapsulate the data by defining each variable for the scope of the functions that use these variables only. Keep them private within the function, or within the module, that uses them.

Modules tend to grow large enough that they are unmanageable. Keep module sizes under 1000 lines to insure tools (source debuggers, compilers, etc.) are not stressed to the point they become slow or unreliable, and to ease searching.

High risk and low risk code

Code is divided into two groups: high risk and low risk. High risk code is dependent on the target's particular hardware scheme. Examples include low-level functions that interface with specific hardware devices, such as device drivers. Low risk code, while application-specific, does not depend on the underlying hardware structure and could be ported easily to a new hardware platform. Examples of low risk code would be the upper layers of the protocol stacks.

Keep high risk and low risk code in separate modules. This localizes any changes necessary when the underlying hardware changes, and makes it easier to port application code to future projects.

Templates

To encourage a uniform module look and feel, create module templates named "`module_template.c`", "`module_template.h`" and "`module_template.asm`", stored in the source directory, that becomes part of the code base maintained by the VCS. Use one of these files as the base for all new modules. The module template includes a standardized form for the header (the comment block preceding all code), a standard spot for file includes and module-wide declarations, function prototypes and macros. The templates also include the standard format for functions.

Module Names

Though long module names are a wonderful aid to identifying what-goes-where, all too many compilers and debuggers don't properly handle names longer than 8 characters. In some cases this may be a fault inherent in the object file format or a debugging file. Limit names to 8 characters or less.

Never include the project's name or acronym as part of each module name. It's much better to use separate directories for each project.

Big projects may require many dozens of modules; scrolling through a directory listing looking for the one containing function main() can be frustrating and confusing. Therefore store function main() in a module named main.c or main.asm.

File extensions will be:

C Source Code	FileName.c
C Header File	FileName.h
Assembler files	FileName.s
Assembler include files	FileName.inc
Buld rules for make	Filename.mak
Directory Contents	README

Variables

Names

Regardless of language, use long names to clearly specify the variable's meaning. If your tools do not support long names, get new tools.

Variable names will be as descriptive as possible. Pointers will have the suffix "Ptr" attached to the variable name. Variable names start with a lowercase character; subsequent words in variable names are capitalized.

The ANSI C specification restricts the use of names that begin with an underscore and either an uppercase letter or another underscore (_[A-Z_][0-9A-Za-z_]). Much compiler runtime code also starts with leading underscores. To avoid confusion, never name a variable or function with a leading underscore.

These names are also reserved by ANSI for its future expansion:

E[0-9A-Z][0-9A-Za-z]*	errno values
is[a-z][0-9A-Za-z]*	Character classification
to[a-z][0-9A-Za-z]*	Character manipulation
LC_[0-9A-Za-z_]*	Locale
SIG[_A-Z][0-9A-Za-z_]*	Signals
str[a-z][0-9A-Za-z_]*	String manipulation
mem[a-z][0-9A-Za-z_]*	Memory manipulation
wcs[a-z][0-9A-Za-z_]*	Wide character manipulation

Example variable names

Not acceptable	Acceptable
Skt	originalSocket
Ec	errorCode
*p	*incomingPacketPtr
Len	packetLen

Acceptable variable prefixes, suffixes and abbreviations

Len	as a suffix for length
Ptr	as a suffix for pointers
num	as a prefix for a count (eg. numPacketsReceived

Global Variables

All too often C and especially assembly programs have one huge module with all of the variable definitions. Though it may seem nice to organize variables in a common spot, the peril is these are all then global in scope. Global variables are responsible for much undebuggable code, reentrancy problems, global warming and male pattern baldness. Avoid them!

Real time code may occasionally require a few - and only a few - global variables to insure reasonable response to external events. Every global variable must be approved by the project manager.

When globals are used, put all of them into a single module. They are so problematic that it's best to clearly identify the sin via the name `globals.c` or `globals.asm`. All global variables ***must*** be prefixed with 'g_" to clearly identify them.

Portability

Don't assume that the address of an int object is also the address of its least-significant byte. This is not true on big-endian machines. Thus, don't make the following mistake:

```
int c;
while ((c = getchar()) != EOF)
  write(file_descriptor, &c, 1);
```

Functions

Regardless of language, *keep functions small!* The ideal size is less than a page; in no case should a function ever exceed two pages. Break large functions into several smaller ones.

The only exception to this rule is the very rare case where real time constraints (or sometimes stack limitations) mandate long sequences of in-line code. The project manager must approve all such code… but first look hard for a more structured alternative!

Explicitly declare every parameter passed to each function. Clearly document the meaning of the parameter in the comments.

Define a prototype for every called functions, with the exception of those in the compiler's runtime library. Prototypes let the compiler catch the all-to-common errors of incorrect argument types and improper numbers of arguments. They are cheap insurance.

In general, function names should follow the variable naming protocol. Remember that functions are the "verbs" in programs - they *do* things. Incorporate the concept of "action words" into the variables' names. For example, use "readFromFifo" instead of "fifoData", or "sendToDisplay" instead of "displayOut".

Interrupt Service Routines

ISRs, though usually a small percentage of the code, are often the hardest bits of firmware to design and debug. Crummy ISRs will destroy the project schedule!

Decent interrupt routines, though, require properly designed hardware. Sometimes it's tempting to save a few gates by letting the external device just toggle the interrupt line for a few microseconds. This is unacceptable. Every interrupt must be latched until acknowledged, either by the processor's interrupt-acknowledge cycle (be sure the hardware acks the proper interrupt source), or via a handshake between the code and the hardware.

Use the non-maskable interrupt only for catastrophic events, like the apocalypse or immanent power failure. Many tools cannot properly debug NMI code. Worse, NMI is guaranteed to break non-reentrant code.

If at all possible, design a few spare I/O bits in the system. These are tremendously useful for measuring ISR performance.

Keep ISRs short! Long (too many lines of code) and slow are the twins of ISR disaster. Remember that *long* and *slow* may be disjoint; a five line ISR with a loop can be as much of a problem as a loop-free 500 line routine. When an ISR grows too large or too slow, spawn another task and exit. Large ISRs are a sure sign of a need to include a RTOS.

Budget time for each ISR. Before writing the routine, understand just how much time is available to service the interrupt. Base all of your coding on this, and then *measure* the resulting ISR performance to see if you met the system's need. Since every interrupt competes for CPU resources, that slow ISR that works is just as buggy as one with totally corrupt code.

Never allocate or free memory in an ISR unless you have a clear understanding of the behavior of the memory allocation routines. Garbage collection or the ill-behaved behavior of many runtime packages may make the ISR time non-deterministic.

On processors with interrupt vector tables, fill every entry of the table. Point those entries not used by the system to an error handler, so you've got a prayer of finding problems due to incorrectly-programmed vectors in peripherals.

Though non-reentrant code is always dangerous in a real time system, it's often unavoidable in ISRs. Hardware interfaces, for example, are often non-reentrant. Put all such code as close to the beginning of the ISR as possible, so you can then re-enable interrupts. Remember that as long as interrupts are off the system is not responding to external requests.

Comments

Code *implements* an algorithm; the comments *communicate* the code's operation to yourself and others. Adequate comments allow you to understand the system's operation without having to read the code itself.

Write comments in *clear English*. Use the sentence structure Miss Grandel tried to pound into your head in grade school. Avoid writing the Great American Novel; be concise yet explicit… but be complete.

Avoid long paragraphs. Use simple sentences: noun, verb, object. Use active voice: "Start_motor actuates the induction relay after a 4 second pause". Be complete. Good comments capture everything important about the problem at hand.

Use proper case. Using all caps or all lower case simply makes the comments harder to read.

Enter comments in C at block resolution and when necessary to clarify a line. Don't feel compelled to comment each line. It is much more natural to comment groups of lines which work together to perform a macro function. However, never assume that long variable names create "self documenting code". Self documenting code is an oxymoron, so add comments where needed to make the firmware's operation crystal clear. It should be possible to get a sense of the system's operation by reading only the comments.

Explain the meaning and function of every variable declaration. Every single one. Explain the return value, if any. Long variable names are merely an *aid* to understanding; accompany the descriptive name with a deep, meaningful, prose description.

Comment assembly language blocks, and any line that is not crystal clear. The worst comments are those that say "move AX to BX" on a MOV instruction! Reasonable commenting practices will yield about one comment on every other line of assembly code.

Though it's useful to highlight sections of comments with string's of asterisks, never have characters on the right side of a block of comments. It's too much trouble to maintain proper spacing as the comments later change. In other words, this is not allowed:

```
/*********************************************
*  This comment incorrectly uses right-hand     *
* asterisks                                     *
*********************************************/
```

The correct form is:

```
/*********************************************
*  This comment does not use right-hand
* asterisks
*********************************************/
```

If your compiler supports comments starting with C++ style "//", use them.

Coding Conventions

General

No line may ever be more than 120 characters.

Don't use absolute path names when including header files. Use the form `#include <module/name>` to get public header files from a standard place.

Never, ever use “magic numbers”. Instead, first understand where the number comes from, then define it in a constant, and then document your understanding of the number in the constant’s declaration.

Spacing and Indentation

Put a space after every keyword, unless a semicolon is the next character, but never between function names and the argument list.

Put a space after each comma in argument lists and after the semicolons separating expressions in a `for` statement.

Put a space before and after every binary operator (like +, -, etc.). Never put a space between a unary operator and its operand (e.g., unary minus).

Put a space before and after pointer variants (star, ampersand) in declarations. Precede pointer variants with a space, but have no following space, in expressions.

Indent C code in increments of four spaces. Tabs stops should be set to 5, 9, 14, etc. Use tab characters rather than spaces.

Always place the # in a preprocessor directive in column 1.

C Formatting

Never nest IF statements more than two deep; deep nesting quickly becomes incomprehensible. It’s better to call a function, or even better to replace complex Ifs with a SWITCH statement.

Place braces so the opening brace is the last thing on the line, and place the closing brace first, like:

```
if (result > a_to_d) {
  do a bunch of stuff
}
```

Note that the closing brace is on a line of its own, except when it is followed by a continuation of the same statement, such as:

```
do {
  body of the loop
} while (condition);
```

When an `if-else` statement is nested in another `if` statement, always put braces around the `if-else` to make the scope of the first `if` clear.

Always surround the block of an `if` or `while` statement with braces, even if there is only a single line of code in the block. This makes the coder's intent absolutely certain to the reader, and makes it easier to add debugging code within the `if` or `while`.

For example:

```
if (packetBufferPtr > PACKET_BUFFER_END) {
      packetBufferPtr = PACKET_BUFFER_START ;
}
```

When splitting a line of code, indent the second line like this:

```
function (float arg1, int arg2, long arg3,
          int arg4)
```

or,

```
if (long_variable_name && constant_of_some_sort == 2
    && another_condition)
```

Use too many parenthesis. Never let the compiler resolve precedence; explicitly declare precedence via parenthesis.

Never make assignments inside `if` statements. E.g., don't write:

```
if ((foo = (char *) malloc (sizeof *foo)) == 0)
  fatal ("virtual memory exhausted");
```

instead, write:

```
foo = (char *) malloc (sizeof *foo);
if (foo == 0)
  fatal ("virtual memory exhausted")
```

If you use `#ifdef` to select among a set of configuration options, add a final `#else` clause containing a `#error` directive so that the compiler will generate an error message if none of the options has been defined:

```
#ifdef sun
#define USE_MOTIF
#elif hpux
#define USE_OPENLOOK
#else
#error unknown machine type
#endif
```

Assembly Formatting

Tab stops in assembly language are as follows:

- Tab 1: column 9
- Tab 2: column 17
- Tab 3: column 25

Note that these are all in increments of 8, for editors that don't support explicit tab settings. A large gap - 16 columns - is between the operands and the comments.

Place labels on lines by themselves, like this:

```
label:
      mov   r1, r2            ; r1=pointer to I/O
```

Precede and follow comment blocks with semicolon lines:

```
;
;  Comment block that shows how comments stand
; out from the code when preceded and followed by
; "blank" lines.
;
```

Never run a comment between lines of code. For example, do not write like this:

```
      mov   r1, r2      ; Now we set r1 to the value
      add   r3, [data]  ; we read back in read_ad
```

Instead, use either a comment block, or a line without an instruction, like this:

```
      mov   r1, r2      ; Now we set r1 to the value
                        ; we read back in read_ad
      add   r3, [data]
```

Be wary of macros. Though useful, macros can quickly obfuscate meaning. Do pick very meaningful names for macros.

Tools

Computers

Do all PC-hosted development on machines running Windows 95 or NT only, to insure support for long file names, and to give a common OS between all team members.

If development under a DOS environment is required, do it in a Win 95/NT DOS window.

Maintain every bit of code under a version control system. In addition, the current compiler, assembler, linker, locator (if any) and debugger(s) will be checked into the VCS. Products have lifetimes measured in years or even decades, while tools tend to last months at best before new versions appear. It's impossible to recompile and retest all of the product code just because a new compiler version is out, so you've got to save the toolchain, under VCS lock and key.

The only downside of including tools in the VCS files is the additional disk space required. Disks are cheap; when more free space is required simply buy a larger disk. It's false economy to limp by with inadequate disk space.

Compilers et al

Leave all compiler, assembler and linker warnings and error message enabled. The module is unacceptable until it compiles cleanly, with no errors or warning messages. In the future a warning may puzzle a programmer, wasting time as he attempts to decide if it's important.

Write all C code to the ANSI standard. Never use vendor-defined extensions, which create problems when changing compilers. The only exception to this rule is when vendor-defined extensions are required to place ISR's at specific memory locations.

Never, ever, change the language's syntax or specification via macro substitutions.

Debugging

You have a choice: plan for bugs before writing the code, and build a debuggable product, or (surprise!) find bugs during test in a system that is impossible or difficult to troubleshoot. Expect bugs, and be bug-proactive in your design.

If at all possible, in all systems with a parts cost over a handful of dollars, allocate at least two, preferably more, parallel I/O bits to troubleshooting. Use these bits to measure ISR time (set one high on ISR entry and low on exit; measure time high on a scope), time consumed by other functions, idle time, and even entry/exit to functions.

If possible, include a spare serial port in the design. Then add a monitor - preferably a commercial product, but at least a low-level monitor that gives you some access to your code and hardware.

Debugging tools are notoriously problematic - unreliable, buggy, with long repair times. As CPU speeds increase the problems increase. Yet these tools are indispensable. Select a dual, complementary, debugging toolchain: perhaps and emulator and a monitor. Or an emulator and a background debugger. Be sure that both sets of tools use a common GUI. This will minimize the time needed to switch between tools, and will insure there will be no file conversion problems (debuggers use many hundreds of incompatible debug file formats).

When selecting tools, evaluate the following items:

- Support - is the vendor responsive and knowledgeable? Is the vendor likely to be around in a few months or years? If the unit fails, what is the *guaranteed* repair time?
- Intrusion - how much does the tool intrude on the system's operation? What is the impact on debugging strategies and development time?
- Does the tool run at full target speed, or will you have to slow things down? What is the impact?
- Will the mechanical connection between the tool and the target be reliable? It's quite tough to get a decent connection to many modern SMT and BGA processors.
- Interrupts/DMA - Will the tool let you debug ISRs? Are interrupts/DMA ever disabled unexpectedly? If the tool does not respond to interrupts/DMA when stopped at a breakpoint (very common), will this have a deleterious effect on your debugging?
- Tasking - If the product uses a RTOS, the tool must provide some support for that RTOS. Insure that the debugger itself is aware of the RTOS, and can display important task constructs in a high-level format. What happens if you set a breakpoint on a task - do the others continue to run? If not, what impact will this have on your development?
- Internal Peripherals - Is the tool aware of the CPU's internal peripherals? Many are; they let you look at the function of the peripherals at a very high level. Do timers stop running at a breakpoint (common)? Will this cause development problem?

Be wary of doing all of your development with the tool's downloader. Burn a ROM from time to time to make sure the code itself runs properly from ROM, and to insure the product properly addresses the ROMs.

Leave all debugging resources in the product when it ships. Disable them via a software flag so they lie latent, ready for action in case of a problem. Remember the Mars Pathfinder: JPL diagnosed and fixed a priority inversion bug while the unit was on Mars, using the RTOS's trace debug feature, which had been left in the product.