Managing Embedded Software Development

Thomas E. Besemer

Concepts and Approaches

Copyright © 2001 ICF Ventures, Inc. All rights reserved. This document may be distributed within engineering environments freely, provided that the authorship is preserved. Commercial usage and/or publication prohibited without written consent from the author. For additional information, please contact the author through http://www.tbcorp.com

1.0 Introduction

There is a tremendous amount of information available which discusses the technical aspects of developing embedded systems. A topic that is often neglected is the unique approaches, from a technical management perspective, required to successfully coordinate the development of embedded systems.

The approaches required to developing an embedded system are unique in the following key ways:

- Almost always the environment is cross development oriented, with host tools used to build binary images which run on a target processor
- The target processor often requires a customized operating systems to be installed on it
- The target processor is often a proprietary design, being developed simultaneously with the embedded application software
- Individual team members on the project are often fluent with native tools such as Visual C++ from Microsoft, but not familiar with target runtime environments such as Tornado and VxWorks from Wind River Systems

Embedded systems are often built on top of operating systems capable of running multiple tasks at one time. Managing the development of an embedded system has the same characteristics; multiple project tasks, all related but each specific in functionality, that must be coordinated by the technical managers of the project. This coordination requires that hardware be developed and/or provided simultaneously with a software design, a host development environment be put in place while system software designs are done and implementation proceeds. The orchestration of this process requires savvy leaders

capable of juggling multiple issues at any given time. Figure 1 provides a graphical example of the embedded project process flow.

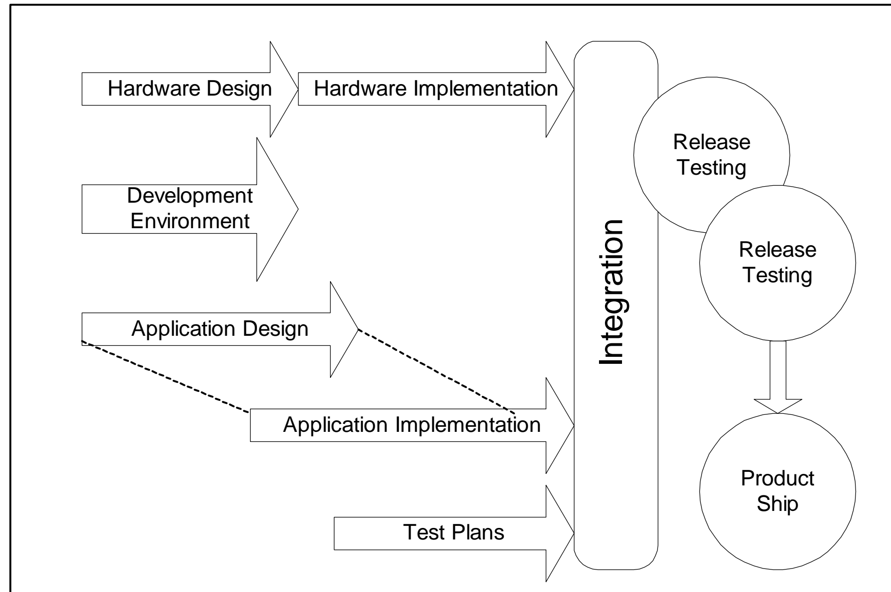

Figure 1 - Embedded Project Flow

This paper is designed to help those in a leadership position better understand the processes necessary for success, and provide some solid guidelines on how to proceed. For Project Managers, numerous solid and tested approaches are presented. For Executive level personal, these overall concepts will aid them in their ongoing evaluations of progress of a project.

1.1 Cross Development

Cross development of an embedded application requires that a host system be configured with tools which support the cross development effort, and some form of a target processor to run the developed application. The host environment, in most cases, is either Unix, Linux or Windows. The target environment is often a proprietary design, with evaluation boards available for both hardware design reference, and initial testing of the embedded application. A typical configuration contains a target environment which is based on a processor architecture that differs from that host development environment. Through this, the target application code will not natively execute on the host, and host tools me be capable of generating executable images compatible with the target CPU architecture.

1.1.1 Target Runtimes

Target environments require some sort of runtime environment. Typically, this is an off the shelf operating system such as VxWorks, Nucleus or Linux. This runtime environment is a critical component to the development of the target embedded system; it must

be developed early in the project, often runs first on some sort of evaluation hardware, and must be extremely stable to ensure the application developers have a solid platform for testing.

This target environment has a component of it which can be referred to as the Board Support Package (BSP). The BSP provides the “glue” which binds the operating system to the specific underlying hardware. For example, when coming out of reset, basic target board initialization must occur. This initialization is responsible for getting memory operational and putting peripheral devices into a quiescent state. Often, a serial port is configured as a console so that boot messages can be displayed, and user interaction with the operating system can take place. Network devices may be initialized by this layer (certain operating systems, such as VxWorks or Linux, will provide advanced initialization during their start-up).

The BSP provides a platform for the operating system to execute. In a Windows environment, the equivalent to the BSP is the BIOS, which does configuration of the underlying PC hardware, loads the Windows from the local hard disk and passes control to it.

Without a good planing and execution, this target environment can be the source of many subsequent problems during the application development. It is such a key component that it requires early stabilization, and clear control over the source code. An extremely important concept is to task this aspect of the project with a person who is fluent in both hardware and software, and able to work closely with both the hardware and software groups - when there are problems, especially on proprietary hardware, a person who is able to evaluate these system problems and help diagnose whether the issues are hardware, software or both is going to save the developers and project managers many hours of frustration.

1.1.2 Host Environments

The target environment (BSP) is always developed within some sort of host development environment. This host environment consists of, at a minimum, cross development tools. These tools, and this host environment, must be extremely stable early on in the process.

In many organizations, management of the host environments is done through an internal IT group. This IT group is responsible for correct configuration of the host operating system, is responsible for mail operation, responsible for backups, and responsible for account management.

In a cross development environment, there is a secondary, but extremely important level of administration and management required: management of the cross development environment. Seldom does the internal corporate IT group have the required technical background to manage the cross tools. Thus, it is important that a strong team member be given responsibility for this task.

1.2 Scheduling

Project scheduling is a balancing act that needs to find equilibrium between market driven needs and internal resource capability. Many organizations try to work with one

schedule: a schedule that is market driven. This is a very important schedule, as without a deliverable product, revenue issues may prevent ongoing growth.

Through this, there is often a tendency to schedule the project based on what is needed, not on what can actually be accomplished. This results in a culture that can be extremely frustrating for both the managers, and the developers, as the neither reach their goals. Often a good way to work around this is through two schedules:

- A schedule based on the market requirements
- A schedule based on what the developers feel they can accomplish

These are two very different schedules, each with their own trappings. A market driven schedule that the development group does not endorse frustrates the developer, while a schedule based on capabilities by the developer frustrates the management. The key to success is having both, and managing the internal trade-offs necessary to *find a viable and reachable* development plan and strategy.

1.3 Essentials of Program Management

Program management is an art form. It's tied very strongly to scheduling. It's a balancing act between what you want, and what you can do. It's about resources: money and people. Money you either have or you don't. Human resources? Most environments have a staff that are either selected before the project begins, or during the projects execution.

A trapping many organizations get themselves into is feeling that Program Management is about telling their developers what is needed, and when. This approach is about managing the project, not the staff.

1.3.1 Getting Buy-In

"*An Idea Imposed is an Idea Opposed*." This is a pretty powerful phrase with a lot of truth in it. There are other ways to look at this phrase: "*Tell me, I'll forget, show me, I'll remember, involve me, I'll understand.*"

There's a lot of aspects involved in development an embedded application, and often many components are in transition or change: hardware isn't stable, schedules are moving around, requirements are changing. One thing that doesn't go through big changes is the staff you have involved in the project. The success of an embedded project is dependent upon many things, but key is availability of reasonable target hardware, and enthusiasm on the part of the staff to work with the changes.

Nobody works well in defeat, everybody works well when they are heard. It is extremely important that the people responsible for development of the embedded application are buying into the requirements that management puts in place. It gives them a sense of ownership, and makes them part of the process.

1.3.2 Project Ownership

A big part of "buy in" is project ownership. Individual responsibility. People like to take pride in their work. They like to feel as though at the end of each day they did contribute

to the greater end goal. They need to “own” a piece of the project and feel as though they can do a good job on that.

The author has yet to see an embedded project where the requirements did not shift several times during the development phase. Most embedded projects go through changes in requirements during their development, as time to market is critical, and often the market shifts in the middle of development. This is the business model of most development processes; it’s an ongoing education for both the management and the staff.

This ties back directly to “buy in” of a component by the individual developer. If they are in the loop, and “own” the portions of the project, they will want individual success. They will also want to be part of the decision process for the change, as owners of any project want to feel the sense of accomplishment that comes through being involved in the underlying decision making process.

Project Ownership and Buy In are the key ingredients to success. Try to involve the people doing the work in the decision making processes, give them a project they own, and watch them perform well.

1.4 Summary

This section touches on the fundamental components that require strong management approaches while developing a successful embedded system, providing an overview of each of them. The subsequent sections delve into these subjects in great detail, providing the reader with proven approaches that work.

2.0 The Development Environment

It may seem odd to place a great deal of emphasis on the development environment in a document that concentrates on the approaches to managing an embedded project, but in reality, the underlying development environment is central to the success or failure of an embedded project. The author has witnessed numerous projects over the years that have failed simply out of respect for the development environment, and more so, neglect of the underlying details fundamental to a viable development environment. This section provides in detail the essential elements behind configuring and deploying and reliable, robust and productive development environment.

A embedded development environment consists of several key elements:

- Host tools capable of generating target executable images
- Target hardware execution environments
- Potentially, a host based simulation environment
- Configuration Management and Source Code Control tools
- Makefiles an/or other tools to build the executable images
- Common utilities global to the project

Most important is that responsibility for the development environment be assigned to one of the more senior and experienced team members on the project - a component as crucial to the success of a project as the development environment requires a clear owner. More so, it requires a clear understanding by an organizations management that the development environment is non-trivial, and will require ongoing support and maintenance from one or more highly competent individuals.

2.1 Host Development Tools

The most predominate host environments are Windows, Unix or Linux. Dependent upon the cross development tools the user will employ, there may be a choice in what serves as the best host environment. For example, Tornado from Wind River systems is available for both Windows and Unix, while Linux target development are almost always constrained to a Linux host environment. Additional constraints may be focused on what the organizations internal IT department is comfortable managing; many organizations are PC oriented, and desire their engineering groups to work with PCs (as opposed to Unix or Linux).

Each of these different host environments has advantages and disadvantages.

2.1.1 Unix Server Hosted Development

A Unix server inherently has a lot of advantages to it:

- The host development tools and environment is installed once, configured, and is readily available to individual users on a centralized server
- Updates to the development environment do not need to propagate to multiple workstations
- Robust scripting capability exists native to Unix
- Simplicity in doing daily, weekly and monthly backups due to all users working on a common filesystem
- Strong control over permissions in the system, ensuring that individual users do not inadvertently corrupt the development environment
- Availability of numerous GNU based tools, such as CVS for Source Code Control, or GCC for native code compilation and execution

Some disadvantages include:

- In a non-Unix shop, reluctance or inability by the IT department to administrate the server
- Does not run Microsoft documentation tools, such as Word or PowerPoint
- May not easily integrate into the corporate email environment
- Increasing difficulty to hire Unix capable engineers
- Server cost can prove costly for larger groups of developers

Also worth noting is that Unix historically has been perceived, incorrectly, as difficult to administrate, and needing of a full time administrator. The releases of Unix available the past few years have been increasingly easy to administrate, and often, if set up correctly, require absolutely no daily administration.

2.1.2 Windows Hosted Development

The advantages to using Windows for a host development environment are numerous:

- Readily available hardware at a reasonable cost
- Almost always integrates easily into the corporate IT groups plan
- Allows the use of Microsoft documentation tools such as Word or PowerPoint
- Availability of a large talent base of engineers comfortable in the Windows environment
- Availability of numerous text editors which end users enjoy for code development

However, there can be several significant disadvantages to using Windows for the host environment:

- Typically, the cross development tools must be installed on each individual machine, complicating management of the tools
- Individual users have a tendency to “customize” their development machines, which may cause problems with the cross development tools
- Lacking in scripting capability
- Difficult to perform backups on each machine, this leaving the project vulnerable to code being developed on the local drives, and either inadvertently being corrupted by developers, or lost during a system/disk crash
- Lack of simple centralized administration capability

2.1.3 Linux Hosted Development

Linux has emerged the past two years as a viable embedded platform. There are several strains of Linux which runs on standard PC hardware, serving well as the host environment, and there is a vast selection of cross development tools available in the public domain.

Some of the advantages of a Linux environment are:

- Extremely economical
- Vast amounts of public domain, GPL software available
- A lot of interest in the technology, and thus, a talent pool available of engineers eager to work with Linux
- Increasing support by commercial vendors for development tools
- It’s all GNU, and thus, open

Some of the disadvantages are:

- The technology is new, and thus, stable versions of either host or target configurations are limited at times
- Does not integrate well with non-linux IT groups
- Many commercial tools, such as Tornado from Wind River systems, or ClearCase from Rational Software, are not available
- Lack of a talent pool of highly experienced developers

- Concerns over GPL and protection of Intellectual Property

A characteristic unique to Linux is that if Linux is selected as the target operating system, it almost requires that Linux be used as the host development environment. This is due to the fact that Linux natively hosts the entire configuration and source code for cross development, and vendors do not currently provide for alternate hosts for linux targets.

2.1.4 Mixed Mode Development

A mixed mode development environment, comprised of a centralized Unix[1] server for the core development tools, and standard Windows based PCs for individual users, is often an extremely attractive solution for many developers. A configuration of this nature has the following characteristics:

- One central Unix server which contains the cross development and configuration management tools
- Multiple client PCs which are integrated into the corporate IT environment
- The Unix file system is exported via Samba, and mapped into the client PCs, appearing as native Windows disk or network destination
- Client PC users are able to login to the Unix server either via telnet sessions, or through X-Windows client software

The advantages to a configuration such as this include:

- All software and project files reside in the developers home directory under Unix, and are easily backed up
- Client PCs may run the editor of the individual developers choice
- Developers do not require an extensive background in Unix, and non-Unix users can be trained quickly
- Native Microsoft documentation tools, such as Word and PowerPoint are easily available to the developer
- Centralized administration of the development environment
- Robust software development and scripting tools available for power users, yet hidden from average grade developers

The one significant argument against this environment that comes up is that you still need a Unix system administrator. The realities of this are not as difficult as one may think; a correctly configured Sun system can run for months and months without any administration needs, and most often, the technical lead person who is responsible for host tool configuration can easily provide the type system administration required.

The most significant argument towards a configuration of this nature is the absolutely reliability of the host development environment. This development environment can be configured and tested very early in a projects life cycle, and if done well, will rarely

1. In many cases, Linux may be used in the same capacity as Unix, provided that the developers tools will correctly run under Linux.

require updates. Additionally, the need to replicate individual workstations, as required in a Windows based development environment, is avoided; this saves a significant amount of time, and ensures that every single developer working on the project is using a standardized, stable development environment.

2.2 Target Hardware Issues

This is an extremely critical component of the process of developing an embedded system: when is target hardware available? In some cases, off the shelf hardware is being used, and is available from the start. More typically, custom, proprietary hardware is designed and built for the system, and this process occurs simultaneously with the development of the application software.

Proprietary designs are often derived from a reference board of some form, available from a vendor. Hardware designers are able to leverage the core design as a starting point for their own design, and software developers are able to work with the reference boards as a target to develop the Board Support Package (BSP) infrastructure, as well as run application code.

2.2.1 BSP Development

The Board Support Package (BSP) provides the lowest level of support for the target hardware. It provides the necessary interface code to support the operating system, and often, contains the Device Drivers which provide an interface between the application software and the underlying hardware. Figure 2 depicts the relationship between the BSP and the other system components and modules.

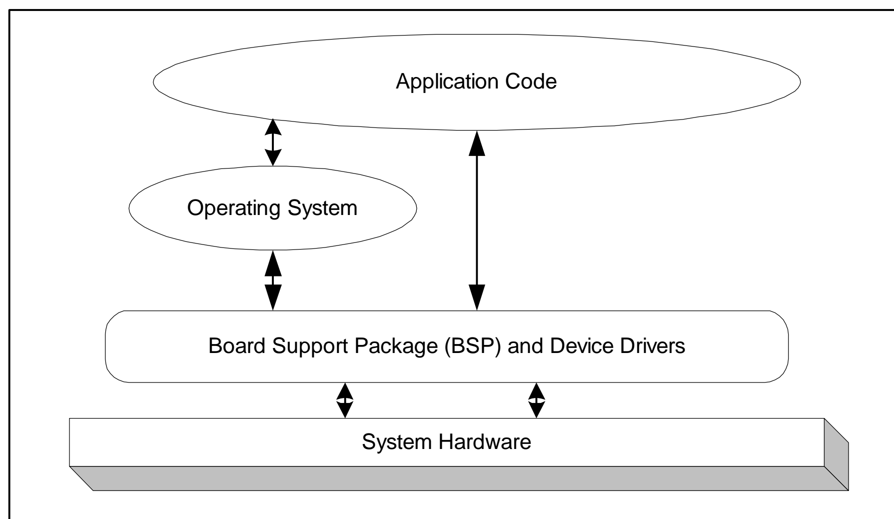

Figure 2 - Board Support Package Positioning

Many reference designs are available with a BSP for one or more operating systems such as VxWorks or Linux. This BSP often provides an excellent starting point for developer, but will generally require additions to support the users specific needs, or

specific peripheral devices; it is extremely important for management to understand that these vendor supplied BSPs are not ending points, or products, but starting points for the projects internal BSP.

In environments where proprietary hardware design is derived from a reference board, the reference board can be used as a solid platform for prototyping up the product BSP. Although specific peripherals may change on the proprietary environment, the basic infrastructure for the BSP will remain the same:

- Operating system independent initialization code to bring the board out of reset
- Some sort of boot loader capable of loading the operating system into memory for execution
- Basic Serial Port (console) and/or network support to interact with the target during board bringup

Since hardware development will occur in parallel to application code development, stabilization of a BSP environment early on in the process is extremely important. It provides a test bed for bringing up the actual target hardware, and provides a runtime environment for early testing of the application code prior to arrival and deployment of the actual target processor.

It is very possible to stabilize the BSP environment prior to arrival of the actual target hardware, thus providing a ready to run image that can be used to start debugging the hardware. Additionally, it is easily possible to abstract the hardware from the application software through careful design and usage of Device Drivers, thus allowing the application code to run on the reference board.

2.2.2 Using Reference Boards

While reference boards do not always contain the exact set of peripherals as the production target board, reference boards can still provide an extremely valuable testing environment for application software. This is achieved through careful design of the BSP, and clear abstraction of the underlying hardware from the application code through Device Drivers.

Additionally, robust simulation environments can be configured. For example, let's assume that a proprietary board contains a custom network interface that does not exist on the reference design. Through this network interlace, communications is established between some sort of remote connected computer. Through careful Device Driver design, a native reference board Network Interface can be used, and some sort of host based application can be put in place to "simulate" the actual physical device and interconnection. With this configuration, the actual embedded application code, abstracted from the hardware, may be coded and tested prior to the availability of production hardware.

Figure 3 presents the relationship between a reference boards BSP, and potential abstraction layers to provide a test bed for the actual application development.

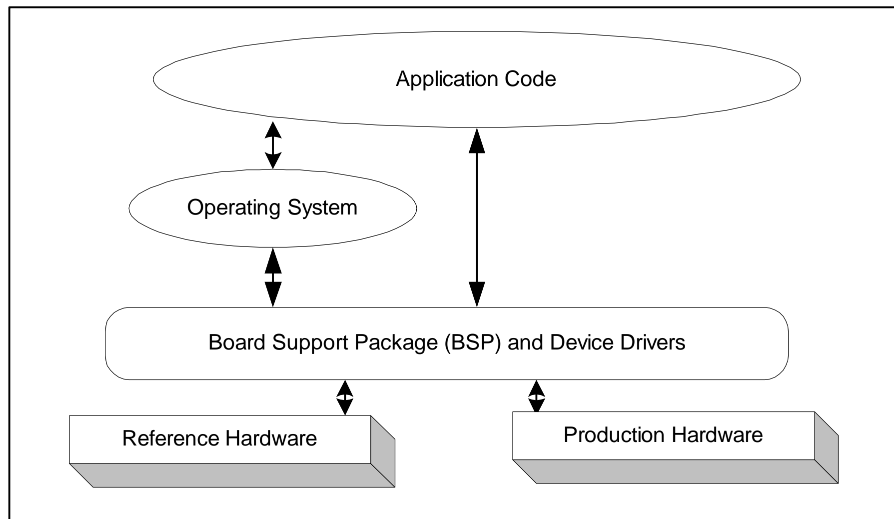

Figure 3 - BSP, Reference Board and Abstraction

Each application and configuration will have it's own unique requirements, yet with some simple steps, abstractions can be done to allow the upper layer software to be developed and unit tested prior to integration on the actual target. This can give the developer a significant head start on the unit testing, easing integration struggles towards the end of the project.

2.3 Host Based Simulation

Another extremely powerful concept is host based simulation. With this, the developer is able to unit test significant portions of their application either prior to the availability of target hardware, or in parallel with the usage of the target hardware. There are several advantages to this:

- For larger groups, the need to provide a target for each developer is avoided
- All systems are built up of small subroutines or functional modules; these modules represent threads of C code, and these threads can easily be tested using native host tools, thus ensuring their stability prior to integration
- Ease of development; not all programmers are comfortable working with real hardware, and at times, the real hardware is not stable

The key to moving down this path is rooted in abstraction from the target hardware functionality through Device Driver simulators. Additionally, abstraction from the operating system calls may be necessary when the host system does not contain native calls which are compatible with the target operating system. The approaches require that some additional software modules be developed, modules which will not be part of the production system. Some will argue that this is a diversion, yet many, many times the amount of time saved through providing the user with a clean development environment far outweighs the cost of development of the simulation and abstraction layers.

Figure 4 provides a conceptual view of an abstraction configuration to allow application code to operate in a Windows environment.

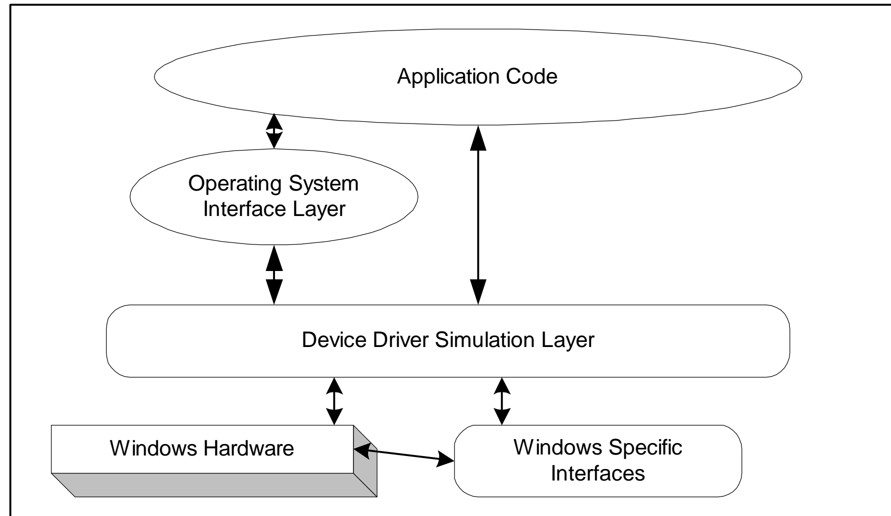

Figure 4 - Windows Based Host Simulation

Developers using Linux for their target environment have a great advantage when it comes to host based simulation: the host operating system (Linux) has the same system calls available as the target operating system. Pseudo Device Drivers can easily be put in place to give the application developer the I/O operating as the actual target.

For groups using an off the shelf operating system such as VxWorks from Wind River Systems, they have a few options. First, the Tornado tool suite provides a host based simulator which provides all operating system calls. An alternative is to abstract the calls to the operating system through a lightweight interface layer, and simulate the runtime using pthreads (available in both Unix and Windows environments).

The most important aspect of host based simulation is the stability of it. Correctly implemented, it can provide a very stable development environment for module and unit testing, avoiding late evenings on early (alpha) hardware releases.

2.4 Configuration Management

Configuration Management, and Source Code Control, is more than a set of tools: it is a philosophy. It is based on a tool, and there are many available in many different forms, but mostly, it is a set of policies developed for the project and agreed upon by all team members.

Far too often you hear a couple of arguments from developers (and at times, Managers). One, we haven’t started coding, so we don’t need it. The other one is “the code’s not done, so I don’t want to check it in.”

Putting Configuration Management in place is part of setting up the host development environment; it should be a tool considered just as important as the cross compiler. Set it up early, test it, ensure it's stable and be ready for the coding to start. Use it for your design documents. Don't wait for trouble to happen during coding, showing you that you need it (this trouble will happen early) - get the job done up front before you start the project, and put it behind you.

2.4.1 Usage Guidelines

Every group will have a different philosophy, but one good rule of thumb is "*if it's not checked in, it's not available in any means to any person other than the developer of the file and/or module.*" Configuration management is a way to share modules and files in a controlled fashion. It prevents problems by giving the developers history on the project, and individual project files; if "today's" build does not work, then check out code and/or modules from yesterday, or last week as a straightforward means of starting to isolate the location of the problems or erratic behavior.

Additionally, people make mistakes. Files get deleted or corrupted by accident; having them checked into a central repository (which is backed up) provides a safety net for the developer, and the entire project. And, depending upon how branches and merges are handled, code does not need to work correctly to be checked in - it can be checked in as part of the evolution of a module, providing history and safety. Once modules are completed and tested, they can be merged and/or made available to the group at large.

2.4.2 CM Tools

There are a large variety of options for Configuration Management. CVS and RCS are free (they're GNU), run under Windows, Unix and Linux, and CVS has several Graphical User Interfaces available for it. Both are widely used, and are well documented. Commercial tools can range from middle of the road options (PVCS or StarTeam) to high end, robust options (ClearCase). What works best for each user depends on their budget, and their needs.

The most important step to take early in the project is evaluate the options, make some decisions, and get the tools in place and tested. Standardize a project development directory structure, publish it, and ensure that everybody is working in a consistent way. It's going to save a lot of trouble down the road.

2.5 System Build Environments

The build environment (sysgen) is another key component of the development environment. Application code developers must rely on it to build project wide test and production images, and, potentially, local developer private builds. Production groups, such as the QA group, may want to be able to automate the build process, producing nightly builds and reports. Like other components of the development environment, designing, implementing and testing a build environment up front is going to save the project many hours and headaches down the road.

There are two common type build environments: Makefiles, and Integrated Development Environments.

2.5.1 Makefiles

Most commonly found in Unix or Linux type environments, Makefiles are text files that are processed by the Make utility to translate source code into object and/or executable files. A set of rules, dictated by the Make utility, defines the operation of the Makefiles.

A Makefile based build environment has numerous advantages:

- Support for multiple build targets, i.e., production, test and individual developer targets.
- Easily automated, as the build (Make) is most commonly invoked from the command line (even in Windows, Make utilities such as Opus Make allow builds to happen from the DOS Command Prompt).
- Most vendors of commercial products, such as Operating Systems or Cross Compilers support the Make environment.
- Easily integrates into a CM environment.

However, there are some disadvantages to a Makefile based environment:

- Makefiles are not straight-forward concepts, or implementations, and it can be difficult to find non-Unix/Linux personal who are "Makefile" literate.
- Makefiles do not, typically, integrate into Integrated Development Environments (IDE).

2.5.2 Integrated Development Environments (IDE)

An IDE is most typically found in Windows type environments. A good example of this is the Visual Studio by Microsoft. An IDE integrates the Compiler, the build tools and the Text Editor into one GUI based environment.

There are numerous advantages to an IDE based environment:

- Straight-forward operation without in-depth training.
- A large talent pool available that is comfortable in an IDE environment.
- Can potentially, when using off the shelf tools, provide a large head start to a projects development environment (Makefiles do not need to be developed).

The disadvantages can include:

- May not easily integrate into an automated build environment.
- Cannot easily be scripted from command line tools.
- Difficulty in managing with a CM tool.

2.6 Common Utilities

There are numerous common utilities that exist in any embedded project, such as:

- Ring buffer handlers
- Linked list handlers
- Diagnostic message handlers and loggers (robust "printf()" type capability)

- Memory management
- Exception handlers

Every project will have a set of common utilities, specific to each project. It is very possible that during the design of a project, these utilities can be defined. When implementation begins, these should be the first components coded, as these components can be utilized by all the developers.

2.7 Coding Standards

Successful projects have coding standards. Always. They help ensure consistency in the implementation, thus making it easy for group members to share code, and understand components of the system developed by other members of the project. The most successful set of coding standards are those that are agreed upon by the bulk of the team members of the project; every developer has their own preferences, and due to such, at times it can be hard to get everybody to agree to a uniform set of project coding standards.

However, not having them can indicate the start of a project that is coded without consistency, and without solid and open communication between team members: two components which can, if not present, kill a project before it starts.

2.8 Documentation and System Design

Everybody agrees to the necessity for it, but nobody really wants to do it. Some people aren't good at it, so either struggle with it, or avoid it. In environments with tight schedules, it's often the first casualty of the project, when it really can be one of the greatest aids to development.

2.8.1 Design Documentation

Code should never become the central problem in a project; it should be one of the steps taken during the implementation of a successful project. Yet often bad code is the root of all project problems and program slips. Bad code starts without a design (or with a bad design), and quickly grows into an unmanageable monstrosity that nobody wants to deal with, throw out (even though at times the best thing that can be done for a project gone awry is to quickly discard of the existing code, without remorse!).

Design documentation is about understanding what you need to code. It represents the blueprints of a project (would you build a house without a plan? Would you start building the walls for a house before the foundation was laid?). It provides a means for individual developers to communicate their plans and ideas, and provides a means to work through problems and flawed concepts. It provides the glue that translates the system requirements into the implementation requirements.

Even the best of programmers make mistakes in their approaches. The best way to identify problems before they are coded is to design the code, and have reviews on the designs. Additionally, embedded systems, by nature, are multi-tasking, and comprised of multiple modules which must interact with other modules without stalls and miss-communications.

The only way to ensure that you are going to write code that works, code that can be unit tested, and code that integrates without hassle *is to design it*. These designs must include clear statements of functionality, API's which other modules will design and code to, and an internal breakdown of how the actual implementation will occur.

Often, the hard and complex design work is done by a small group of senior people. With clear design documents, junior developers can be employed to perform the actual coding of modules, as they will have a clear path.

There is often a tendency of managers to feel uneasy that little coding is happening early on in the project, while lot's of talking and brainstorming is being done. It's extremely important to understand that you will design the project; *either before you start it, or when you re-write it*.

2.8.2 Keeping it Current

Requirements change for many reasons; marketing identifies new features, implementation uncovers miss-understandings of how the hardware works, or the target market, in general, shifts. Design documents, especially those with API's, become useless relics of the past if they are not kept current, yet in the heat of implementation, it requires a great deal of discipline to keep documentation current.

One very viable approach to keeping the documentation current is to put the documentation in the code, and extract it into document format (such as into HTML). This is accomplished through clear coding standards that include defined procedure, structure and file headers.

With defined headers, users may develop simple tools which post-process the C files and extract the documentation. This can be scripted to run every night (thus providing an internal set of web pages which all users can access), or done by hand, massaging the output into professional looking documentation.

Wind River Systems generates the bulk of their reference documentation through scripts and coding standards such as this. The author has developed tools to generate HTML and Unix Man Page documentation. Certain Java tools provide a set of rules which allow this type documentation generation.

Listing 1 shows an example procedure, with header, which can be passed easily through an auto-documentation extraction tool.

```
/*+
 *  NAME
 *      sampleCall - Example procedure
 *
 *  DESCRIPTION
 *      This call provides an example of how the auto-documentation
 *      generator works.  Procedures and Data Structures which
 *      follow this code header format are processed with the
 *      utility <mpgen>, which produces Unix man page output,
 *      providing online reference material which grows with
 *      the projects, and stays current.
 *
 *      Other formats available are HTML, and FrameMaker mif format.
 *
 *  RETURNS
 *      Returns ERROR if a problem is encountered, else OK.
 *
```

```
 *  SEE ALSO
 *      The documentation in this section of the EDT Reference
 *      Manual.
-*/

STATUS  sampleCall( int  p1,   /* Input parameter 1 */
                    int  p2,   /* Input Parameter 2 */
                  )
{
    return( OK );
}
```

Listing 1 - Source Code Suitable for Auto-Documentation

Example 1 shows the resultant output from the example in Listing 1, after passing through a tool which generates Adobe FrameMaker (MIF) format which from the C source code.

NAME
sampleCall - Example procedure

SYNOPSIS

```
STATUS  sampleCall( int  p1,   /* Input parameter 1 */
                    int  p2,   /* Input Parameter 2 */
                  )
```

DESCRIPTION
This call provides an example of how the auto-documentation generator works. Procedures and Data Structures which follow this code header format are processed with the utility <mpgen>.

Other formats available are HTML, and FrameMaker mif format.

RETURNS
Returns ERROR if a problem is encountered, else OK.

SEE ALSO
The documentation in this section of the EDT Reference Manual.

SOURCE FILE
smpl_code.c

Example 1 - Output from Listing 1 Example

The key concept behind this is to make documentation as painless and easy as possible. For developers, headers are extra work, but if it’s an enforced project rule (subject to periodic code reviews). However, it can prove to be a relatively painless path to keeping the documentation current.

This concept is part of setting up the host development environment; it’s putting the tools in place to aid the implementation, and this is just another tool that once done and tested, will provide months of stable usage, and valuable information.

3.0 Product Execution

Execution is everything. It's also the most predominate failure point for start-up companies. There are three key components of project execution which can make or break a project:

1. Approaches to staffing
2. Approaches to scheduling
3. Approaches to testing and integration

3.1 Staffing

A discussion in regards to staffing is beyond the scope of this document; staffing is comprised of many variables, such as company culture, department culture, manager preferences, budget and existing resources. The most important concept for any manager to be aware of is that one strong and experienced technical lead can really help keep a project going on a daily basis. A person such as this should be experience in the following areas:

- The Operating System being used on the target
- The host development tools and environment
- Design and integration of a project of similar size, complexity and capability
- Strong system analysis and debugging skills

This person should be responsible for the development environment, driving the coding standards and project architecture, and in general, day to day operations of the project.

3.2 Scheduling

What's a schedule? Is it a means to identify every task in a project, and help coordinate the development of the project? Is it a means to track the flow of a project, and then start to learn what your groups capabilities are, and from that, learn how to predict your own performance? Is it a document which helps you identify what you want to do versus what you can do? Is it something that the engineering group (team members) have helped generate?

It should be all of these things. What it should not be is a statement by management to the developers of what needs to be done, and when it needs to be done. *Management must remember that successful program management requires a lot more than simply waving a the magic wand of "tell the troops to get it done by Friday"!*

3.2.1 Project Needs versus Engineering Capability

There is a big difference between what you want and need, versus what you can actually do. A lot of organizations miss this subtle point, and schedule based on what they need to do. This approach results in long hours by the developers to meet the unrealistic schedule, causing compromise (marginal workmanship) in implementation, frustration through missed dates, and the planting of seeds of discontent (happy developers write stable code).

Certainly, a project should be a challenge for everybody involved; a set of tough goals to accomplish, and in the end, a sense of pride in having accomplished those tough goals. But a project, and a schedule, should be about what you are capable of doing at any given time; you'd never try to run a marathon if you haven't taken a walk around the block in a few years, let alone trained for the marathon.

In order to successfully execute a project, you must base your goals on what your capabilities are, not on what your desires are. As a consultant, the author has been brought into many groups towards the end of the project to help get it on track. Often the first thing encountered is that everybody is working in defeat: the managers don't have faith in the developers because the developers have not met any scheduled dates or milestones. The developers have been working long hours, producing sub-standard modules, and have an advesarial relationship with the management. And everybody feels defeated.

This situation falls out of an unrealistic schedule, a schedule that did not take into account the capabilities of the engineering group. A schedule that was based on project needs, not developer capabilities. Everybody loses in a situation like this; customers are let down because marketing and sales has made commitments they could not meet. Managers are frustrated because the project grows further behind each day. Developers are frustrated because they do not perceive a management team that wants them to succeed.

There are two ways to work around this:

- Schedule it based on what you need, identify how much work you have to do, and staff the project accordingly
- Get the staff involved in the schedule, and develop a schedule which has buy in, a schedule which is reachable by the group you have

Schedules need to be viable, working components of a project which change based on evaluation of progress and evaluation of your own capabilities. They should provide a means of tracking progress, not imposing direction.

3.2.2 Tracking Progress

Tracking progress is very different than imposing progress; it is about learning your capabilities. It is about starting with a schedule which the group has bought into, and then on a weekly basis, evaluating how well you are progressing. It's an education for everybody, and allows management to make decisions on resource utilization.

The best thing a management group can learn how to do is predict the performance of the development group. The best way to accomplish this is to work with a realistic schedule, track progress and identify problem areas. If week after week the results of tracking progress shows continued and increased program slips in certain areas, the management group will be able to evaluate the best approach to improve the performance in these areas.

You cannot manage a project you do not have visibility into, and the only way to gain that visibility is work with realistic schedules which help you track progress. This helps

avoid the biggest pitfall of all time in software development: the false hope based on the thought rooted in "*I'm confident that it's all going to come together in the end*."

3.2.3 Managing Program Slips

Program slips are best managed by either changing the requirements for the end deliverable, or adding resources to the problem. In cases where the slips are centered around third party software or hardware, it may be important to get the vendor involved very early in the problem (versus waiting a week or two to see if things improve).

Consultants and contractors may be able to help get through some short term slips, but can only be easily utilized if you have clear design documentation; these are short term people who you want to aim at a problem, and have them help you get it resolved. This means they must be able to get up to speed rapidly, and the best way to ensure this is to have clean design documentation. It's extremely important to give any consultants a clear set of objectives to work from; consultants work by the hour, and thus it is important to give consultants a clear set of tasks to perform (even if one of those tasks is helping define what tasks need to be done!).

There is a tendency among groups to manage program slips by skipping two essential steps in the project: design and unit testing. Without a design, the coding approaches quickly turn into "stream of thought" type design, which rarely works. Without unit testing, integration becomes a frustrating, problem filled adventures which almost always results in an unstable code base.

The best approach is to identify the hot spots well before integration and QA, and put your best resources on those. A good rule of thumb to keep in mind is this: with software, it's going to be *on time, cheap or bug free - pick any two*.

3.3 Integration and Testing

Past design, integration and testing can be the most difficult and important aspects of an embedded project. Integration is a difficult phase, as code modules of differing functions, written by programmers with differing skill levels and approaches, are pulled together and made to run as one system. Highly developed test plans can a strong asset during this phase, as they provide a stable reference point to measure progress, bug fixes, changes and feature enhancements.

3.3.1 Test Plans

Successful projects have at least two levels of test plans:

1. Unit test plans
2. Integration test plans

3.3.2 Unit Testing

Unit test plans are internal to the group, and should be designed to test individual modules prior to integration. A good time to develop these test plans is during the design phase; as each module is designed, and any necessary API's developed, unit test plans can be developed specifically for each module.

These test plans do not need to be complicated; their primary purpose is to serve as a checklist to validate each module in the system. In most cases, specific test code will have to be developed to support each of these test plans, and the best way to approach using these test plans and unit test code is through peer review; it's a way to transfer knowledge between team members (which will help integration), as well as validate the performance and status of each module (thus, helping identify with certainty the status of the project in regards to the schedule).

A common failure, especially in environments and projects where time to market is extremely tight, is to skip this phase. The irony is that in one form or another, this phase is always performed: all the code modules in the system must be debugged prior to the system operating correctly and stable! *It's extremely important to understand that it's a lot easier to debug individual modules through unit testing, than debugging larger, more complicated system problems that result from integration with untested modules*.

A few points to remember about unit testing:

- For modules that are abstracted from the underlying hardware, often the unit testing can be done on the host, in a simulation environment; this can prove to be a real time saver.
- Unit test code should be developed in parallel with the module, and should have some sort of simple user interface to allow it to be invoked from the projects diagnostic console.
- Unit test code should do more than initiate tests on a given module; it should be able to query the internal state of the module, reporting the modules status. During integration, this can really help understand the status of the system as a whole.
- The unit (module) test code is just as important a piece of software as the actual; treat it that way.

3.3.3 Integration

Integration is generally one of two things:

1. A point of time where great satisfaction and real progress can be felt. Each of the components starts to be pulled in, and the end functionality of the system comes alive.
2. A point in time filled with long and late hours where nothing but defeat and frustration is felt. As each of the components in the system are pulled in, performance degrades, reliability degrades, and the ability to manage the integration phase is lost.

How integration goes depends on two things:

1. How much time and effort put into design.
2. How much time and effort put into unit testing.

There are a few simple rules for integration that can really help:

- There should be a coordinator of the effort; a person who determines what modules are integrated, and when. A person who has the authority to pull modules out and request further unit testing on them if necessary. Often, this role is best done by the technical lead of the project.

- Regression testing of previously integrated modules should be an ongoing project, ensuring that new code modules being brought into the system do not cause failure of code that has already been validated and tested.
- The same overall functional test plan that is used to by a QA group to validate correct product operation should serve as the guideline for progress (as opposed to individual programmers making a determination that the system is working - code authors rarely find logistical problems in their implementations, as it does exactly what they think it should do!!).

Most important is that integration steps should be tied to the CM tools; for each major step forward, some sort of internal "tag" should be applied to the code tree, ensuring known check and test points are noted. Since numerous modules are pulled in during this phase, it's very easy to go from a stable, working system to an unstable, broken system in short order - make sure you can always get back to what works, and know what was integrated since the last stable point.

3.3.4 Functional Testing and Quality Assurance

Functional test plans are used to validate the stability of the project, and the correct functionality of the project. They should be derived from project functional specifications or system requirements specifications. If an internal QA group exists, they can prove to be the best origin for test plans of this nature.

Although these test plans should be available to the development team, and used by the development team during integration, the best approach is to periodically provide a separate test group formal ("tagged") releases of the system for test. Simple release notes should be provided by the development group to the test group during this phase; these release notes help the test group identify what is believed to be working and included, helping focus testing efforts.

By working with "tagged" releases, the test group can generate reports of anomalies against each build, and developers can then perform fixes and enhancements against these reports for inclusion in subsequent releases. It's very important to be extremely organized during this phase of the project; developers need to understand what a "bug" is reported against, as they may or may not have already addressed the issues. Testers should not be testing features that are known not to work.

Most important is reproducibility of problems. This is why the concept of known, "tagged" releases is important. Reports of anomalies should not only include a description of the issue, but a detailed description of how to reproduce the issue. When a developer is assigned the task of investigating any particular anomaly report, the first thing they will want to try is see if the problem/issue still exists *in the release which they are currently working with*; if so, then they can go ahead and take steps to address the issue. If not, they will want to work with the same version of the system that the testers found the problem/issue on, and make a determination of whether it's been fixed correctly in the current version, or merits further investigation.

Careful, controlled and deliberate steps are the keys to success in this phase.

3.3.5 Bug Tracking

Bug tracking is an essential part of the development and product deployment process. Software, by nature, has bugs, deficiencies, errors and omissions. This applies to every software project, whether written by average people, below average people, or highly experienced, above average people: it's the nature of the beast.

It is extremely important to manage the "bug list" of a project; certain bugs will be of higher priority than other bugs. For example, a problem that causes the product to "hang" every 5 minutes is going to be of much higher priority than a spelling error in a diagnostic message that is displayed once or twice a week. More so, the people who are responsible for fixing reported problems are generally busy with feature enhancements, ongoing integration or other project tasks. Therefor, it's important to manage the "bug list" like any other aspect of the project - prioritize it, schedule and assign resources, and then verify completion of the task.

Many Configuration Management tools come with defect tracking components. These can be extremely useful to the developer, and the capabilities of this aspect of a CM tool should be taken into account when evaluating the CM tool early in the project. At a minimum, a spread sheet can be used, or simply a ascii file with a list of problems, if the CM tool does not support bug tracking.

The most important message is *manage the bugs like any other component of the project*: Thirty 1 day bug fixes add up to thirty days of effort. A common failure by management is to assume that they just have a handful of bugs, and those bugs will get fixed while normal progress continues on a project; this illusion can cause a great deal of friction between the development staff and the management, as well as result in program/schedule slips.

Plan for bugs.

A typical cycle for a "bug" in a project is through the following steps:

1. A formal release is given to a test group.
2. Testing identifies one or more "bugs" in the release.
3. Reports are generated describing the issue, and how to reproduce it.
4. Project management periodically reviews the reports, and assign resources to each report.
5. Individual team members attempt to reproduce the issue, and if necessary, make changes to the code to fix the issue.
6. On the next formal release provided to the test group, a set of release notes include what bugs (reports) have been addressed.
7. The test group verifies that the problem/issue has been fixed; if so, the issue is closed. If not, the issue is re-submitted through an updated report, or new report.

A few rules of thumb that help keep the test phase running smoothly:

- Testers should test independently, and not interfere with developers unless absolutely necessary. Both groups have jobs to do.

- Developers should not be “yanked around” to fix problems as they arise; this kills productivity. Schedule the bug fixes like any other component of the project.
- Without robust test plans, you can’t honestly evaluate the status of a project.
- If you didn’t do your unit testing prior to integration, you will be doing it during functional testing. It will take longer at this phase.
- Make sure you are doing testing against formal releases of code; developers passing “custom builds” to testers is the first step down the slippery slope of loss of control over the integration and deployment of a project. There is a natural tendency for developers to do this, as well as managers - they like to see problems cleared up. Clear them up with control.

4.0 Summary

Developing and embedded system is based on coordinating numerous complicated projects operating in parallel. This document outlined concepts that have been proven to work, but the author certainly does not wish to imply that the concepts presented represent the only path. Managers and developers are strongly encouraged to work with each other, communicate with each other and develop ideas, concepts and plans which every single person involved in the project can accept, ideas which all involved can support.

As maybe the most important thing to remember was stated early in this document: *an idea imposed is an idea opposed*. Communications is the key to success in any relationship, whether it be an embedded system or not.