Timesys Tutorial

Embedded Linux Development

Contents

CHAPTER 1

CHAPTER 2

CHAPTER 1

Introduction to Embedded Linux Development

Linux is the operating system of choice for almost all new embedded device projects today. Linux provides a powerful, flexible kernel and runtime infrastructure that is continuously being improved by the open source community and extended by hardware vendors to support new processors, buses, devices, and protocols. Cost-conscious and time-critical embedded hardware projects can take advantage of its freedom from downstream licensing and redistribution costs, while decreasing development and prototyping time by leveraging the vast amount of system software, middleware, and application software whose source code is freely available for Linux. Embedded device projects can often reduce hardware costs by taking advantage of the power and flexibility that a true multi-tasking operating system brings to embedded devices. The Linux kernel and associated open source infrastructure is the heart of a new ecosystem for embedded operating system, infrastructure, and application prototyping, optimization, and deployment.

Unfortunately, power, flexibility, and easy availability do not mean that manually creating and maintaining a device-specific Linux platform (commonly known as a "Roll Your Own" — RYO — Linux platform) is an easy task. As many RYO projects come to realize, having to assemble all of the software and tools required for an embedded device can be time-consuming in the short term, and requires continued attention to bug fixes and enhancements to the open source software that you are using in the long term. Companies need a significant level of Linux expertise to get started: finding compilers that produce executables for a target device, selecting open source software packages to satisfy device requirements, determining what packages those packages require and the order in which they must be compiled, how to customize the "right" kernel for your hardware, how to assemble a kernel, system software, and custom software into a format that a device can use, and so on.

While the Timesys FREE and PRO Editions automatically provide much of that expertise for you, let's first establish a common vocabulary for Linux, embedded Linux, and Linux software development. The next few sections introduce basic Linux terminology, highlighting the basic components of Linux on any platform and the software that is required to configure, compile, and deploy Linux for a specific device in order to satisfy your project requirements.

This chapter provides introductory information about Linux and embedded Linux systems that you can skip if you are already familiar with Linux and its organization.

Core Linux Concepts

If you're new to the Linux operating system, the first thing that you need to know is that the term "Linux" specifically refers to the operating system itself, better known as the "Linux kernel." Originally written by Linus Torvalds for the Intel® 80386 architecture, the source code for today's Linux kernel is the result of contributions from thousands of private and corporate developers. The Linux kernel is written in the C programming language, and it supports all major computer architectures. It includes device drivers for thousands of devices, and it is easily extended to support new devices and protocols. The only differences between the version of the Linux kernel that you are running on a desktop system and an embedded Linux kernel are the architecture and processor(s) for which it was compiled, and the device drivers that are available to the kernel.

By itself, an operating system kernel isn't very usable without infrastructure that enables the kernel to start processes and threads, applications that those processes and threads can execute, and system services that support communication with available devices using supported protocols. This infrastructure is stored in a Linux filesystem, and is made up of open source software, much of which is from the Open Software Foundation's GNU project. The combination of a Linux kernel and its software infrastructure is therefore often referred to as "GNU/Linux."

A **Linux platform** is a custom combination of a Linux kernel, system infrastructure, and applications that run on a specific computer architecture and associated processor. A **Linux distribution** is a complete, generic set of these components from a specific Linux vendor that you can customize to create an individual Linux platform. Linux distributions for desktop and server computer systems typically include an installer that simplifies deployment on available hardware, include a customizable mechanism for booting installed systems, and include some mechanism for updating those systems once they have been installed. As discussed in the next section, Linux distributions that target embedded development consist of the same sorts of components, which you then compile a specific set of components that makes up a Linux platform that satisfies the requirements of a specific embedded hardware device and the services that it must provide.

Core Embedded Linux Concepts

An "embedded Linux distribution" is a Linux distribution that is designed to be customized for the size and hardware constraints of embedded devices, and includes software packages that support a variety of services and applications on those devices. A key differentiator between desktop/server and embedded Linux distributions is that desktop and server software is typically compiled on the platform where it will execute, while embedded Linux distributions are usually compiled on one platform but are intended to be executed on another. The software used to compile the Linux kernel and its infrastructure is referred to as a "toolchain," as discussed in detail in the next section.

Introducing Toolchains

A "toolchain" is a compiler and associated utilities that enable developers to produce a kernel, system software, and application software that run on some specific target hardware. As mentioned in the previous section, toolchains for traditional desktop and server systems execute on those systems and produce executables and libraries that will run on that same hardware. Because of the hardware constraints of embedded devices, most embedded Linux development takes place on desktop systems that may have a different system architecture and processor than that of the device for which they are being compiled. Compilers that run on one type of system but produce compiled code for another are therefore known as "cross-compilers."

Just as a Linux kernel for one architecture and processor has its roots in the same source code, the compilers that are commonly used to cross-compile embedded Linux software are based on the same source code as those used to build desktop and server software. The next section discusses the GNU Compiler Collection (GCC), which is the package that provides these compilers, highlighting some additional options that these compilers provide when building software for embedded systems.

GNU Compiler Collection Overview

After the Linux kernel, the GNU Compiler Collection (GCC) is probably the most significant piece of open source software ever developed. While other compilers are freely and commercially available for Linux systems, none have the extensive support for multiple architectures and processors that is provided by GCC. The term GCC is context-sensitive, because the name of the C compiler that is part of the GNU Compiler Collection is itself "gcc" (GNU C Compiler). This wasn‘t confusing when the C programming language was the only supported language, but today‘s GNU Compiler Collection includes compilers for many other languages such as Ada, C, C++, Fortran, Java, Objective-C, and Objective-C++.

The GNU C Compiler, gcc, itself depends on packages such as the binary utilities package, "binutils," which provides the GNU assembler (gas), a linker/loader (ld), and many other utilities used for manipulating executables, constructing and optimizing libraries, and other compilation-related tasks.

The GNU C compiler is used to compile the Linux kernel, any custom bootloader that you are using with an embedded project and all of the system services and applications that make up the in-memory and permanent filesystems that you use in an embedded Linux project. (All of these topics are discussed later in this document.) When compiling anything other than a kernel or bootloader, the GNU C Compiler also requires a runtime library of C language functions, generically referred to as a "C Library," which is discussed in the next section.

C Library Overview and Comparison

A C library defines and supports the system calls that make requests of the kernel, and also provides higher-level functions that create and terminate processes, create and open files, allocate and de-allocate memory, print character strings, and so on. On Linux systems, a standard C library is typically provided as a shared library, and must be present for the system to execute compiled C applications.

- glibc — the standard C Library that is used with the GNU C compiler is the GNU C Library (glibc). The glibc library provides a rich and robust set of C language functions, and is therefore fairly large. Compiling applications that use the GNU C library can therefore produce large executables whose size is not a problem on desktop or server systems, but which can push the memory and storage constraints of many embedded devices.

Because of the size requirements of applications that are compiled against the GNU C Library, several other C libraries are available for use with the GNU C compiler. These libraries were primarily developed to reduce runtime requirements by producing smaller executables for embedded development projects. Common alternate C libraries are the following:

- **dietlibc** — a small standard C library used to produce static executable that do not use shared libraries.
- **eglibc** — a variant of the GNU C Library that is designed for use on embedded systems and to provide binary compatibility with applications that are compiled for use with glibc.
- **Newlib** — a small standard C library that is used in many free, commercial, and vendor-distributed toolchains for embedded development, and is also used by the popular Cygwin project, which is a set of tools and API layer for Linux applications in the Microsoft Windows environment.
- **uClibc** — a small standard C library used to produce static or shared-library executables

To simplify using the GNU C compiler with a different C library, Linux distributions that target embedded development typically provide toolchains that have been built with that C library and therefore use that library automatically. The Timesys Web and Desktop Factory tools include both glibc-based and uClibc-based toolchains.

Boot Loaders for Linux

A bootloader is a small application that is executed when a computer system is powered on, loads an executable image into memory, and then begins its execution. On Linux-based systems, bootloaders typically load the Linux kernel. Because they require significant knowledge about and interaction with the underlying hardware, bootloaders are often specific to the computer architecture on a specific system.

Bootloaders are an important consideration when developing embedded systems, depending on the capabilities of the board and processor on which an embedded system is based. For example, on most Intel® x86 or x86_64 hardware, the bootloader can rely on a Basic Input/Output System (BIOS) to perform device initialization, and can therefore focus on higher-level functions such as selecting a specific kernel to load. On most non-x86 hardware, the bootloader initializes the hardware itself, loads kernel or standalone application images from ROM, Flash, or a storage device, and passes any required options to the kernel before beginning its execution in memory.

A single-stage bootloader loads a single executable into memory and begins its execution. On embedded Linux systems, a single-stage bootloader loads the Linux kernel. A multi-stage bootloader essentially consists of two or more bootloaders, the first of which loads the second, and so on, eventually loading a specific executable, which is the Linux kernel on embedded Linux systems.

The next two sections provide an overview of the bootloaders that are commonly used on embedded systems that use x86 (and x86_64) or other architectures. All of these bootloaders are supported by the Timesys Desktop Factory.

x86-Specific Boot Loaders

Because desktop and Linux server platforms typically use x86 or x86_64 processors, it is not surprising that multiple bootloaders are available for these related architectures. Linux systems originally used bootloaders called LOADLIN (for DOS and Windows filesystem compatibility) and LILO (Linux Loader). Both of these had limitations based on the hardware and filesystem capabilities of the time, and are therefore no longer used.

Today, Linux distributions for x86 and x86_64 platforms typically use one of two different bootloaders, depending on the boot media and boot process that you want to support. These bootloaders are the following:

- **Grand Unified Boot Loader (GRUB)** — the most common bootloader used on Linux desktop and server systems, GRUB is a multi-stage, chainloading bootloader that is most commonly used to facilitate interaction with the boot process. The original GRUB bootloader has been replaced by GNU GRUB, also known as GRUB 2.
- **Syslinux** — a family of light-weight multi-stage bootloaders that supports booting from DOS/Windows or Linux filesystems (SYSLINUX), the network (PXELINUX), and DVD and CD-ROM devices (ISOLINUX). Syslinux is typically used to enable booting from the network or from USB, CD-ROM, and DVD devices.

Though GRUB is traditionally viewed as a large bootloader, it provides an extremely flexible boot environment that can be very convenient during development, and can boot from types of Linux filesystems that are not supported by Syslinux. Syslinux is considerably smaller, but is better used in boot environments where predefined or no interaction with the boot process is required.

A bootloader development kit for Intel® processors, BLDK, is available from Intel® and simplifies the development of custom bootloaders for basic embedded system initialization. BLDK-based bootloaders are not supported in the Timesys Desktop Factory to guarantee conformance with standard Linux boot processes and simplify using Linux platforms that are developed using the Desktop Factory across multiple hardware platforms.

Das U-Boot, the Platform-Independent Boot Loader

Das U-Boot is a highly-configurable multi-stage bootloader that was designed for use with embedded Linux and is the bootloader of choice on non-x86 systems. U-Boot supports booting most ARM, Coldfire, MIPS, PPC, XScale, and similar embedded systems, and can also be used on x86 systems. U-Boot's flexible configuration makes it easy to customize for specific board and processor requirements while keeping the bootloader as small as possible.

Platform-Specific Boot Loaders

Different embedded system hardware provides different amounts of system resources for storing and executing a bootloader. The Timesys Desktop Factory includes support for the following bootloaders for non-x86/x86_64 platforms:

- **APEX** — a bootloader that is used on some ARM-based embedded systems.
- **at91bootstrap** — a first-stage bootloader that is commonly used on AT91 ARM systems, often as a first-stage bootloader for U-Boot, which can then perform more complex system initialization tasks before loading a Linux kernel.
- **at91bootstrap-3** — an extended version of the at91bootstrap bootloader, the at91bootstrap-3 bootloader expands the locations from which a Linux kernel can be loaded and the protocols that can be used to retrieve the kernel.
- **blob** — an open source bootloader that is used with some StrongARM processors.
- **x-loader** — a small single-stage bootloader that is derived from the U-Boot code base. The x-loader boot loader is commonly used on platforms with small amounts of static memory, such as OMAP platforms, and is often used as a first-stage bootloader for U-Boot, which can then perform more complex system initialization tasks before loading a Linux kernel.
- **yaboot** — a PowerPC bootloader for older Open Firmware machines, which are rarely encountered in new embedded Linux projects.

Working with the Linux Kernel

The Linux kernel is the heart of any Linux installation, embedded or otherwise. The kernel is responsible for memory allocation, process and thread creation, management, and scheduling, data exchange with onboard hardware and peripheral devices, and supports the protocols necessary for interaction with other systems, as needed.

Desktop and server Linux systems need to support a large number of onboard and peripheral devices because it is impossible to predict the hardware that will present in a random desktop or server deployment. Similarly, such systems also need to support a huge range of communication and data exchange protocols so that they can be used for a large number of different purposes. Embedded devices typically require support for a specific set of devices, peripherals, and protocols, depending on the hardware that is present in a given device and the intended purpose of that device.

The Linux kernel is highly configurable in terms of the architecture that it is compiled for and the processors and devices that it supports. Building a kernel that supported every possible device and protocol would produce a huge kernel that would include support for many devices and protocols that it would probably never need. Such an “uber-kernel” would also have correspondingly large memory and storage requirements. To minimize the size and associated requirements of the kernel, the kernel can be built with a configurable set of device drivers, which are the kernel components that enable the kernel to interact with specific types and instances of onboard and peripheral hardware.

Configuring the kernel for the architecture, processor, and specific hardware requirements of an embedded device is the foundation of any embedded Linux project. Optimizing the kernel to support specific hardware and protocols minimizes its size and memory/storage requirements, and can positively impact performance. At the same time, limiting the kernel to a specific set of hardware and protocols can also limit its flexibility in supporting additional hardware or increasing protocol requirements.

The key to balancing hardware-specific optimization and flexible support for additional devices and protocols lies in how the kernel supports device and protocol drivers, as described in the next section.

Working with Device Drivers

The Linux kernel can use device drivers in one of two ways. They can either be compiled into the kernel or can be dynamically loaded on demand when delivered as loadable kernel modules (LKMs). Loadable kernel modules are compiled object files that are compiled to work with a specific kernel and that can be dynamically loaded into a running instance of that kernel in response to detecting new hardware or protocol requirements. LKMs are stored outside the kernel as files in a physical or in-memory filesystem that the kernel has access to, and have the ".ko" (kernel object) extension to make it easy to identify them.

Though dynamically loading (and unloading) kernel modules have some minimal kernel overhead, the cost of that overhead is offset by the flexibility that LKMs provide during both development and deployment. When developing device drivers for any new hardware that is associated with an embedded Linux project, developing those drivers as LKMs enables in-progress drivers to be loaded, tested, unloaded, and updated without rebuilding the kernel itself. The LKM model is also extremely useful during device deployment and use because it enables the kernel to load device and protocol drivers on-demand, which is often critical for consumer electronics devices with external network, storage, or general peripheral interfaces such as USB, that provide a general interface that can be used by many different devices.

Kernel and associated device driver configuration is therefore largely a matter of differentiating between mandatory and on-demand devices and protocols. On Linux desktop or server systems, support for everyday devices like PATA or SATA hard drives is typically compiled into the kernel because those devices are present in most computer systems and you can't load a device driver from hardware that you can't access. As a general rule, device drivers for the hardware that is required to boot a system are usually built into the Linux kernel so that those drivers are automatically available. Once a Linux system boots and can access an in-memory or device-based file system, the drivers for almost any other hardware can be loaded from LKMs that are stored in that filesystem. Device drivers for other hardware and protocols, which may only be required when an embedded device is being used in a specific way, are therefore commonly developed and deployed as loadable kernel modules.

Incorporating new devices drivers into the source code for the Linux kernel has licensing implications for those drivers, but not for most system or application software that uses them. See Intellectual Property and Embedded Linux for information about Linux and open source software licensing.

The Linux kernel is configured using an integrated mechanism known as **kconfig**, which simplifies architecture, processor, and device driver selection, and automatically satisfies dependencies between different drivers. The Timesys Desktop Factory provides integrated kconfig support, and also extends this simple, usable configuration mechanism to the other portions of an embedded Linux development project, such as operating system infrastructure and application selection and support, as discussed later in this document.

Interfacing with Devices Under Linux

Providing a flexible mechanism for loading and using device drivers requires that a Linux system is able to access associated devices in a similarly flexible fashion. Linux systems access devices through special files known as device nodes, which must be present before a related device can be accessed.

Traditionally, the Linux device nodes that a Linux system might need to access were pre-created in the `/dev` directory in an in-memory or storage-based Linux filesystem. Pre-creating device nodes is still a good solution for embedded systems that only need to access a limited set of devices, but is often not suitable for more flexible embedded systems (or for Linux desktop and server systems) to which additional hardware can be dynamically attached.

Rather than wasting space by pre-creating (and storing) every possible device node, modern Linux systems support an on-demand device creation mechanism known as **udev.** A process which is started as part of the operating system infrastructure for the kernel monitors the existing hardware in a system and watches for new hardware detection events. When new hardware is detected, this daemon creates mandatory device nodes by executing one or more rules that are appropriate for the class of hardware or specific device that was detected.

Completely dynamic device creation depends on the order in which hardware is initialized and recognized by a Linux system, which may not always occur in the same sequence. If not addressed, this could cause, for example, Ethernet interfaces to be named in a different sequence. This would make it difficult to configure the system to handle those devices differently or use them for specific purposes, because you could not be sure of their names. To eliminate this sort of problem by providing the right compromise between static and dynamic device creation, the udev daemon creates, updates, and saves sets of rules for certain classes of detected devices. When a Linux system is restarted, these rules are used like any other udev rules in response to hardware detection. However, because they also contain specific name assignments, the device-to-name mapping is the same after each system restart.

The udev mechanism provides flexibility when required, but is not mandatory. For embedded devices that will only access a specific set of devices (such as on-board devices), pre-creating static device nodes for that hardware eliminates the need to run a udev daemon, store udev rules, and so on, reducing both system memory and storage requirements. On the other hand, on embedded devices that are designed to be used with peripheral devices that you cannot necessarily predict, such as random USB devices, the udev mechanism provides the initial device nodes required for a default system, and the flexibility that the device requires in the real world.

Configuring Kernel Boot Options

The Linux kernel is a fascinating topic, but is primarily a facilitator for system services, applications, and user interaction with a system. The resources that are available to a given Linux kernel are therefore often specified as options when booting that kernel. The default values for these boot options are typically set when configuring the kernel, but can also be overridden by supplying those values as arguments to a bootloader or to the kernel itself when it is loaded into memory.

Being able to override default values that the kernel requires, such as memory size, boot device location, and other device-specific settings provides significant flexibility without requiring that you continuously recompile the kernel to specify different, pre-set values.

Working with File Systems

A file system (AKA **filesystem**) is a container for system and application programs and data that can be accessed by the kernel and applications in a consistent fashion. Filesystems must be **mounted** in order to be accessed, which simply means that they must be associated with a Linux directory. A Linux directory that is used for mounting a filesystem is known as a **mount point**, but is a standard Linux directory in all respects.

Relatively small file systems can be created in system memory, but do not support persistent data because they are recreated in memory after any system restart. File systems that must support persistent user and configuration data or non-default applications are created on persistent storage devices, such as appropriately-formatted NAND or NOR flash or traditional disk-based storage devices, regardless of whether that storage is local to a device or available over some sort of network.

Initial RAM Disks and File Systems

Part of the boot process for any Linux system requires access to one or more file systems that contain the executables for the services and applications that the system provides. When booting, most Linux systems first load an in-memory file system that contains a relatively small number of system-level executables that start basic services, explicitly load device drivers that are required later in the boot process, and so on. The final commands that are executed from these in-memory file systems often mount persistent storage from some location, depending on the capabilities and purpose of that system.

The use of an in-memory file system during the Linux boot process is common practice, but is not mandatory. Embedded Linux platforms that do not need the initialization capabilities that are provided by these file systems can directly mount and use their root file system. Root file systems are discussed in the next section.

In-memory file systems that are typically used during the boot process are known as an initial RAM disk (**initrd**) or initial RAM file system (**initramfs**), depending on how they are organized:

- **initial RAM disk** — a pre-created, fixed size file system image (optionally compressed) that is loaded into memory, requires a specific driver for the type of filesystem used in the initrd, and requires some amount of memory and processing time to copy programs from the filesystem into memory so that they can be executed.
- **initial RAM file system** — typically created by directly expanding a compressed file into the page cache used by the kernel, and therefore does not require a file system driver, is very fast, and does not have a fixed size because the size of the page cache can continue to grow, as required.

Most modern Linux systems use an initramfs because of the optimizations that are provided by the initramfs format and how it is loaded. Initial RAM file systems can either be built into the kernel or can be identified via the initrd kernel boot option when loading a kernel and beginning its execution.

Root File System

All Linux systems mount a filesystem on the directory `/`, known as the root directory. This filesystem, known as the root filesystem, contains the system infrastructure and applications which that Linux system can execute.

As discussed in the previous section, Linux systems can mount and use multiple root filesystems, though not at the same time. Linux systems typically mount an in-memory file system for use during the boot process. The initramfs filesystem that is initially used as a root filesystem by most 2.6 and later Linux systems is not a filesystem per se, but is a compressed archive file in the Linux `cpio` archive format that is uncompressed into the kernel's page cache, which is then mounted as a temporary filesystem. This in-memory filesystem begins the Linux boot process (discussed in Linux System Initialization and Startup Process) and can either be used as the actual root filesystem for an embedded Linux system that does not need to preserve information across system restarts, or can simply be used as an initial root filesystem that then mounts a storage-based file system to provide the system infrastructure and runtime environment that is used while that Linux system is running.

Storage-based file systems are file systems that are located on a non-transient storage device, are "formatted" using one of the many different file system types that are supported by Linux, and are then mounted and accessed directly by that system's runtime infrastructure and applications. The next sections introduce the most common filesystems used in embedded Linux projects and the types of storage that they are traditionally used with.

Traditional Storage-Based File System Types and Formats

Linux supports many different types of file systems, each of which provides different internal capabilities, though they all simply appear to contain files and directories to developers and users. The most common Linux file systems are the **ext2**, **ext3**, and **ext4** filesystems which, as their names suggest, are subsequent versions of the same extended filesystem that add new capabilities to their predecessors. The **ext2** filesystem is the canonical Linux filesystem, while **ext3** adds journaling capabilities to protect the integrity of the file system and updates to it. Journaling filesystems queue filesystem updates in an on-disk log to preserve both their integrity and the integrity of the filesystem itself in the event of an unexpected system restart. The **ext4** is also a journaling filesystem, but adds new allocation methods that improve efficiency and performance when allocating large amounts of storage.

The ext2, ext3, and ext4 filesystems, along with other types of Linux file systems such as JFS, ReiserFS, and XFS, are designed and optimized for use on traditional storage media such as hard drives. With the exception of ext2, all of these filesystems are journaling filesystems that support fast system restarts, but the specific type of filesystem that you decide to use in an embedded Linux project is a fundamental that requires some amount of expertise with the administrative utilities that are associated with each type of filesystem.

Many embedded systems do not include hard drives, but use filesystems that are contained in on-board storage such as NAND or NOR flash. Though flash and other on-board media filesystems can be formatted in standard Linux filesystem formats, they are more commonly formatted using types of filesystems that are designed for use on the specific types of storage devices and which provide media-specific functionality. The next section discusses the Flash-based file systems that are commonly used in embedded Linux projects.

Flash System Types and Formats

As discussed in the previous section, Linux supports many different types of filesystems, each of which has its own advantages, disadvantages, and caveats. The previous section introduced the types of Linux filesystems that are commonly used on hard drive-based Linux desktop and server systems, but there are also a number of different Linux filesystems that are designed for use in embedded systems, where the most common storage medium is Flash.

Most of the types of filesystems used in embedded Linux projects focus on minimizing filesystem size and providing media-specific capabilities. For example, flash devices can only be written to a finite number of times, and therefore usually provide internal mechanisms such as **wear-leveling**, which try to evenly distribute writes to the underlying flash media to maximize its usable lifespan. Interestingly enough, some of the Linux filesystems that are designed for embedded use address underlying media issues by not being writable, which means that they can only be updated by writing an entirely new filesystem image to the associated storage.

The different types of Flash filesystems that are commonly used in embedded Linux projects and are therefore supported by embedded development tools from Timesys are the following:

- **cramfs** — the Compressed RAM file system is a simple read-only filesystem that is designed to minimize size by maximizing the efficient use of underlying storage. Files on cramfs file systems are compressed in units that match the Linux page size (typically 4096 bytes or 4 MB, based on kernel version and configuration) to provide efficient random access to file contents.
- **jffs2** — the Journaling Flash File System, version 2, is a log-based filesystem that, as the name suggests, is designed for use on NOR and NAND flash devices with special attention to flash-oriented issues such as wear-leveling.

- **squashfs** — like cramfs, Squashfs is a compressed, read-only filesystem that was designed for use on low memory or limited storage size environments such as embedded Linux systems.
- **ubifs** — the Unsorted Block Image File System (UBIFS) is a relatively new filesystem that was designed for use on Flash devices. Ubifs generally provides better performance than JFFS2 on larger NAND flash devices, and also supports write caching to provide additional performance improvements.
- **yaffs2** — Yet another Flash File System, version 2, provides a fast and robust file system for large NAND and NOR Flash devices. Yaffs2 requires less RAM to hold filesystem state information than filesystems such as JFFS2, and also generally provides better performance if the filesystem is being written to frequently.

Read-only flash filesystems such as cramfs and squashfs are commonly used on stateless embedded devices that do not need to preserve data across system restarts. Filesystem directories such as `/tmp` that are used as temporary file creation by many Linux services must still be writable if the services that the system supports require that directory. Writable locations such as `/tmp` can always be mounted on a small RAM disk to support temporary file creation in a primarily read-only environment.

Read/write filesystems such as jffs2, ubifs, and yaffs2 are used on stateful embedded devices, where service or application state data is saved and reused across system restarts. Selecting the type of writable Flash filesystem to use in a particular embedded project is more closely tied to the physical characteristics and technology used in the underlying Flash storage. JFFS2 typically performs best on smaller Flash partitions because it stores filesystem index information in memory and therefore requires that all underlying storage blocks be scanned before a filesystem can be mounted. Such scanning will increase system restart time on systems with larger amounts of Flash.

Ubifs and yaffs2 filesystems are better suited to larger NAND and NOR Flash devices. As its name suggests, ubifs can only be used with Flash controllers that support the Unsorted Block Image (UBI) volume management interface for access to underlying standard Memory Technology Device (MTD) Flash hardware. On existing embedded hardware, UBI support is therefore an initial decision point when choosing between the ubifs and yaffs2 filesystems. When designing new embedded device hardware, Flash filesystem characteristics can help drive hardware selection. As the newest of these filesystems, ubifs is most actively under development, supports filesystem data compression, and generally provides the best I/O performance on large, writable MTD flash devices.

Linux System Initialization and Startup Process

Previous sections of this document referred to the system infrastructure that is stored in a Linux filesystem and provides runtime and administrative support for Linux systems. This section introduces the components of that infrastructure and how they are used during the Linux boot process.

On most embedded Linux systems, the program loaded into memory by a boot monitor is the Linux kernel. This is typically a compressed kernel image that is loaded directly into memory. This compressed kernel is prefixed by some instructions that perform initial hardware and execution environment setup (stack definition, page table initialization, starting the swapper, etc.), uncompress the kernel image into high memory, store any initial RAM disk or initial RAM filesystem in memory for subsequent access, and then begin execution of the uncompressed kernel.

The uncompressed kernel sets up interrupts, performs additional hardware initialization, and then uncompresses and mounts any initial RAM disk or filesystem that was found in the kernel image or specified on the kernel command line using the initrd keyword:

If an initial RAM disk or filesystem is found or specified, the system follows the sequence of events described in Initial RAM Disk/Filesystem Initialization.

If no initial RAM disk or initial RAM filesystem is found in the kernel or identified using the initrd parameter, the kernel mounts the root filesystem identified on the kernel command-line, and then starts the standard system initialization process, as described later in The Linux init Process**.**

Initial RAM Disk/Filesystem Initialization

Once an initial RAM disk or filesystem is mounted, the next step in the boot process for any Linux system is to execute an initialization file that is provided in the RAM disk or filesystem. The name of this initialization file depends on whether an initial RAM disk or filesystem is being used:

- **Initial RAM filesystem** — Linux systems that boot with an initial RAM filesystem execute the file `/init` in that filesystem
- **Initial RAM disk** — Linux systems that boot with an initial RAM disk execute the file `/linuxrc` in that filesystem

These files are typically text files that contain Linux shell commands, and are therefore known as **shell scripts**, or more simply **scripts**, which use a specified Linux shell to execute other Linux commands and scripts. These commands and scripts set variables for the system's execution environment, load specific device drivers, and initialize subsystems that may be required to access storage devices and their contents, such as software RAID or volumes that are managed by LVM (the Linux Volume Manager).

The `/init` or `/linuxrc` files can also be a version of the `/sbin/init` program that follows the standard initialization process described in The Linux init Process.

Executing the `/sbin/init` program directly is sometimes done when the initial RAM disk will be your run-time filesystem, as is the case for many embedded systems. However, most embedded systems execute `/init` as a shell script. The flexibility provided by running a shell script that can execute other commands and scripts provides a very flexible boot mechanism that can do a significant amount of system configuration before eventually spawning the `/sbin/init` program.

On Linux systems where an initial RAM filesystem or disk is used temporarily before a storage-based root filesystem, the last steps in the `/init` or `/linuxrc` script typically:

1. mount the storage-based root filesystem identified in the kernel, in the script, or using the root= kernel boot parameter,
2. transfer virtual filesystems that the kernel uses to provide user-space access to hardware and system state (`/sys`) and process management (`/proc`) to the new root filesystem by remounting them in that root filesystem, and
3. begin using that root filesystem as the new root filesystem by executing a kernel helper application called `run-init` that switches to the new root filesystem by executing the `chroot` (change root) system call and then starts a specified initialization program (typically `/sbin/init`) in the context of that filesystem, as discussed in The Linux init Process.

Some desktop and server systems do not use the traditional init mechanism, but use similar initialization mechanisms such as upstart (Ubuntu 6.10 and greater, RHEL 6, Fedora® 14) or systemd (Fedora 15). These alternate initialization mechanisms are designed to be less sequential and more event-driven than the traditional init mechanism, but are not commonly used in embedded Linux systems.

The Linux init Process

After loading the kernel and mounting a run-time root filesystem, Linux systems traditionally execute a system application known as the init (initialization) process, which is found in `/sbin/init`. The init process is process number 1 on a Linux system, and is therefore the ancestor of all other processes on that system. This initialization mechanism predates Linux, and was introduced in System V Unix. It is therefore commonly referred to as SysVinit.

Embedded Linux systems that are created using the Timesys Desktop and Web Factory typically use a small-footprint initialization mechanism that largely works like SysVinit, but is actually provided by the BusyBox framework. The BusyBox init mechanism eliminates the overhead of some of the capabilities of the traditional SysVinit mechanism that are rarely, if ever, used in embedded Linux systems. See for detailed information about BusyBox. The Timesys Desktop and Web Factory also support the full, standard SysVinit mechanism.

When using SysVinit or a work-alike, the `/sbin/init` program reads the file `/etc/inittab` to identify the way in which the system should boot and the scripts and programs that it should start. Any line in this file that begins with a hash-mark ('#') is treated as a comment, and is not executed. Entries in this file that are actually executed are made up of four colon-separated fields of the following form:

```
<id>:<runlevels>:<action>:<process>
```

- **id** — the TTY (Linux terminal) on which the associated process should run. If specified, the contents of this field are appended to `/dev/`. If not specified, the default Linux console, `/dev/console` is used. If a serial console is being used, all entries with non-empty id fields are ignored.
- **runlevels** — one or more integer values that identify the Linux **runlevels** for which the associated command should run. The BusyBox init mechanism does not support runlevels, which means that any value in this field is ignored. See for more information.
- **action** — one of the following values:
  - ***askfirst*** — displays a "please press enter to activate this console" message before executing the specified process, and waits for enter to be pressed before starting that process. Rarely used on embedded systems, except during debugging.
  - ***ctrlaltdel*** — identifies a process to be executed in response to the Ctrl-Alt-Delete input sequence.
  - ***once*** — identifies a process that is started once when the system boots and is not respawned if it exits. This value is often used for applications that initialize system resources or services prior to their use by another process.
  - ***respawn*** — starts the specified process, and automatically restarts that process if it exits
  - ***restart*** — identifies a process to be executed in response to a system restart request. If used, this typically identifies the full pathname of the `init` program.
  - ***shutdown*** — identifies a process to be executed in response to a system shutdown request.
  - ***sysinit*** — the specified process is started when the system boots

    - ***wait*** — the specified process is started when a matching runlevel is entered, and the init process waits for the process to complete before proceeding

- **process** — the full path-name of the program or script to execute

The most important entries on systems that use the BusyBox SysVinit work-alike are the following:

```
::sysinit:/etc/init.d/rcS
::shutdown:/etc/init.d/rcK
```

The first of these entries runs the `/etc/init.d/rcS` script when the system boots. The second of these runs the `/etc/init.d/rcK` script when the system is being shut down. Both of these scripts execute other script files that begin with the letter S (startup) or K (kill), followed by a two-digit integer value to enforce the order in which they are executed. These script files are also located in the `/etc/init.d` directory. Script files beginning with the letter S are executed in increasing numerical order, while those beginning with the letter K are executed in decreasing numerical order. Example file names are `S01-udevd`, `S02-mount`, and so on.

When running system startup scripts through the `/etc/init.d/rcS` script, that script also executes the `/etc/rc.local` script after processing all startup scripts. This file is typically used to execute system-specific startup commands, enabling customization of the startup process for a specific machine without making general changes to the standard startup scripts for that system.

Linux Root Filesystem Contents and Organization

The root filesystem on any Linux system contains programs, libraries, scripts, and data files. Like any Linux filesystem, the root filesystem is organized hierarchically using directories that can contain files and other directories. The names of these directories and their organization conform to the Linux Filesystem Hierarchy Standard (FHS), which is not duplicated here for brevity.

Root filesystems on embedded Linux systems typically take advantage of both specific projects and general Linux functionality to minimize their size and simplify their contents. Regardless of the amount of persistent storage that may be available to an embedded Linux system, using a root filesystem conservatively simplifies both maintenance and debugging.

The next few sections discuss applications and techniques to consider when creating a root filesystem for use on an embedded Linux system.

Introducing BusyBox

When first creating a root filesystem, most embedded Linux projects start with a filesystem that is initially populated with programs that are based on the BusyBox package, which is a multi-call binary program that can perform the function of most basic GNU Linux system utilities. A multi-call binary is one that performs different functions based on the name by which the binary is executed. Multi-call binaries can be executed with different names by copying a program to a new filename (which must be executable) or, more typically, by creating hard or soft links with the name(s) to the existing program. When starting embedded Linux projects, the BusyBox utility and specially-named links to its executable are often the sole components of your root filesystem, along with some standard device nodes if you are not using the udev dynamic device creation mechanism.

A complete list of the standard GNU utilities that the BusyBox application can emulate would be both long and almost immediately outdated. However, some of the more commonly-used are the following: ar, brctl, the bz2, compress, gz, uuencode, xz, and zip utilities, chmod, chown, chroot, cp, dmesg, du, fdisk, find, the grep utilities, httpd, ifconfig, ifdown, ifup, init, kill, ln, losetup, ls, mkdir, mknod, mkswap, the modprobe utilities, mount, mv, netstat, nslookup, ping, pivot_root, poweroff, reboot, renice, route, rpm, sed, sh, sort, su, swapon and swapoff, sync, tar, telnet, tftp, top, traceroute, umount, and wget.

Using BusyBox to function as utilities such as these saves both disk space and build time because standalone versions of these utilities do not need to be built and installed in your embedded Linux system's root filesystem. The Timesys Desktop Factory configuration tool simplifies configuring your embedded Linux build to use BusyBox to function as an optimal set of GNU utilities when you select the **BusyBox Init** template.

The BusyBox version of the `/sbin/init` program does not support all of the capabilities provided by traditional SysVinit. The capabilities that it does not support are rarely relevant on embedded Linux systems. For example, the primary SysVinit capability that is not supported in the BusyBox init work-a-like that is used on most embedded systems is the idea of **run levels**. Run levels are a way of defining sets of applications that the system administrator believes should be run when the system is being used in a certain way. Most embedded systems are only used in a single way — to provide a specific set of services — and therefore don't need to support special administrative environments.

System Infrastructure for Applications

Libraries are collections of pre-compiled code that can be used by any application that is linked to them, and provide a common way of maintaining sets of functions that can be used by multiple applications. Applications that use library functions resolve references to those functions differently depending on the type of library that the executable is being linked with. Linux systems support three basic types of libraries: static, shared, and dynamically loaded libraries:

- **static libraries** — Static libraries are the oldest and simplest form of code libraries, and are linked at compile-time into applications that require them. Static libraries typically have the .a (archive) or .sa (static archive) extension.

  Static linking results in standalone applications that are easy to deploy due to reduced dependencies, but also creates larger applications because they contain a copy of all of the functions from that library that are used by that specific application. This eliminates the potential for code-sharing that shared and dynamically linked libraries provide, and also increases the complexity of application updates because each application that is statically linked with a library must be relinked with any new version of that library that is released.

- **shared libraries** — Shared libraries are centralized code libraries that an application loads and links to at runtime, rather than at compile time. Shared libraries typically have the file extension .so (shared object).

  Applications that use shared libraries must be able to locate and resolve function references to those libraries when an application is executed, which introduces dependencies that must be considered when deploying embedded applications and systems. However, they can provide significant advantages on embedded systems:

  - ***Simplified application maintenance*** — Shared libraries can be updated at any time, and all applications that use those libraries can automatically take advantage of bug fixes or enhanced functionality provided in newer versions of those libraries.
  - ***Reduced system disk space requirements*** — Applications that use shared libraries are smaller than statically linked versions of those applications because the library is not bundled into the application. This is especially significant in graphical applications that often use functions from a large number of different graphical toolkit libraries.

    - ***Reduced system memory requirements*** — Applications that access shared libraries still need to load the shared libraries into memory when these applications are executed. However, shared libraries need only be loaded into system memory once in order to be used by any application which uses that library. This reduces overall system memory requirements when more than one application that uses a specific shared library is being executed.

- **dynamically loaded libraries** — Dynamically loaded libraries are libraries that an application can load and reference at any time while it is running, without requiring the pre-loading that shared libraries require. Unlike static and shared libraries, which have different formats, dynamically loaded libraries differ from other types of libraries in terms of how an application uses them, rather than how the runtime environment or compiler uses them. Both static and shared libraries can be used as dynamically loaded libraries.

  Dynamically loaded libraries are commonly used in applications that use plug-ins or modules. Dynamically loaded libraries enable you to defer the overhead of loading specific functionality unless it is actually required, and also simplify separately developing, maintaining, and deploying plug-ins independent of the executable that actually invokes and loads them. Applications that use DL libraries employ a standard API for opening DL libraries, examining their contents, handling errors, and so on. In C language applications on Linux systems, this API is defined and made available by including the system header file `dlfcn.h`.

How you use libraries with your applications deserves special consideration on embedded systems, where minimizing both application size and system overhead are fundamental considerations that can have different effects on decisions regarding library use. Shared libraries provide significant advantages if your embedded system will be executing multiple applications that use the same libraries (such as multiple graphical applications), but can needlessly increase compilation complexity if those libraries are only used by a single application. Similarly, dynamically loaded libraries can simplify distributing add-on functionality to existing applications, but require using an additional API to load them and access the functions that they contain.

Starting Application(s) at Boot Time

Many embedded systems execute a single application. That application is typically late in the init process for that system, and therefore, how that application is started depends on the type of storage that is associated with the system, and how that system is being initialized:

- **RAM-based systems** — Systems that run from an in-memory filesystem such as an initial RAM disk or initial RAM filesystem typically start custom applications at the end of the `/linuxrc` or `/init` script, respectively.

- **Storage-based systems** — Systems that use the BusyBox or SysVinit initialization mechanisms typically start custom applications in one of several ways:

    - ***`/etc/rc.local`*** — this script is typically run after all standard system init scripts (located in the directory `/etc/init.d`) have been executed. This script provides a good place to start custom applications that do not require significant pre-configuration, and can therefore be started by appending a command or two to the end of this file.

    - ***via an existing init script*** — applications that depend upon daemons which are started by existing init scripts do not usually require their own startup scripts. For example, web-based applications that run in the context of a web server may simply be started or made available as a consequence of starting the associated daemon via its standard startup script in the `/etc/init.d` directory.

- ***via a custom init script*** — creating and using an application-specific init script (`/etc/init.d/`SNN-application) is commonly done when an application requires setting custom environment variables and other setup before launching the application.

Modifying a file that already exists in your RFS, such as `/etc/rc.local` or an existing init script, means that you will have to preserve your modifications each time you rebuild or reinstall your RFS. To simplify this, you can create an archive file that contains your modifications and other customizations. The Timesys Desktop Factory makes it easy to apply such archive files (in Linux `tar` format) on top of a freshly-created RFS.

When creating a custom init script or adding entries to an existing script to start your application, the commands used to start applications that do not automatically detach themselves must run in the background (using the '&' symbol) so that the script can complete but leave the application running. Detaching is typically done within a server application by using the standard Linux fork/exec model, or through a separate administrative application that spawns the server process.

When developing and debugging a custom embedded application or server, it is often convenient to start that application manually after logging in on the embedded system. However, simply starting an application in the background from a shell prompt using an ampersand runs that command in a new shell that is still dependent on the shell that it created it. Normally, when that parent shell terminates, the child shell and its subprocesses (your application or server) does too. The easiest way to prevent this is by prefixing the command that you want to run with the Linux `nohup` command (no hang-up), which causes your application to be started as a child of the init process on your system, making it independent of the shell from which it was started and enabling it to continue running after that shell terminates. The `nohup` command is part of the Linux coreutils package. You can also use the Bash shell's `disown` built-in command or a nanny program such as the `/sbin/start-stop-daemon` that is used to spawn daemon processed on many desktop and server Linux distributions.

Embedded Linux Development Roles

As you can see from previous sections, developing successful embedded Linux systems requires knowledge of several different domains. The first, and most obvious of these is knowledge about the specific applications that an embedded system is being designed to support, but embedded Linux development projects must also have a good understanding of how those applications will be integrated into the target system, how that system and your applications can be tested, and so on.

The next few sections introduce the distinct types of knowledge that are required for an Embedded Linux project to succeed. Unless your embedded Linux project has multiple team members with different domains of expertise, these domains of expertise do not have to be under the control of different people — they are more easily thought of as **roles** in an embedded project, and are often performed or supported sequentially by one or more developers rather than being the responsibility of distinct individuals.

Operating System and Platform Support

The Operating System and Platform Support role in an embedded development project is fundamental to any embedded Linux project. The team member(s) responsible for this role configures and creates the basic Linux platform that is required by the applications and services that are delivered by the embedded device. Configuring and building an optimal Linux and filesystem requires a good understanding of the target hardware for the project, whether a development board or custom, device-specific hardware.

The team member(s) in this role is/are responsible for the following:

- **Toolchain** — Identifying the appropriate C library for the project, and building or obtaining the cross-compiler and associated libraries for the target architecture and processor.
- **Bootloader** — Selecting an appropriate bootloader, identifying and implementing low-level system configuration tasks that are required before loading the kernel, loading the kernel (and filesystem, when appropriate) into appropriate system memory, and beginning kernel execution.
- **Kernel and Device Drivers** — Correctly configuring the kernel for the target architecture and processor, enabling required drivers, developing and integrating any custom devices drivers that are required by the target hardware, choosing between integrated device drivers or loadable kernel modules and identifying and resolving tradeoffs between size, performance, flexibility, and simplicity, and enabling an initial RAM filesystem or RAM disk.
- **Initial RAM Filesystem or RAM Disk** — Configuring the in-memory filesystem that will be used during the kernel boot process, identifying any custom software requirements for that filesystem (such as specific device drivers or applications), creating the initialization script for that filesystem, and creating that filesystem in the correct format for use by the kernel.
- **RFS** — Unless the device will only use an initial RAM filesystem or RAM disk, building the root filesystem for the target device requires selecting, configuring, compiling, and installing:
    - the system’s initialization (init) mechanism,
    - networking support (as required),
    - device drivers that will be available in the RFS as loadable kernel modules, including custom device drivers,
    - the system infrastructure that is required by the embedded system in order to support both the system itself and the applications and services that it must deliver,
    - any local or remote system logging mechanism used by that infrastructure,
    - the system infrastructure, such as a web server, that is required by any user-accessible administrative utilities, and
    - any utilities and libraries required by the application(s) and services that the embedded system delivers to the end user.

The team member(s) that fills this role designs and delivers a stable Linux platform that supports the applications and services that a target embedded device requires. These components can either be configured, compiled, and integrated manually, or can be produced by tools such as the Timesys Desktop Factory, which provides a single, consistent interface for configuring, building, and deploying those components for desktop use (the tool chain) and on the target system.

Application Development

The Application Development role in an embedded development project leverages the toolchains and underlying system provided by the team members(s) in the Operating System and Platform Support role. The Application Development role is perhaps even more critical, because the team member(s) in that role develop and deliver the applications, services, and any associated administrative tools that define the functionality of a given embedded Linux device and differentiate that device from others.

The team member(s) in this role is/are responsible for:

- developing any custom applications that are required on the embedded system,
- developing any custom or customized libraries that are required by those custom applications, and
- developing any administrative interface that is required in order to configure and use the capabilities of those custom applications.

The team members(s) in the Application Development role must work closely and interactively with those in the Operating System and Platform Support and Integration and Quality Assurance roles. Application development and debugging requires that applications must be integrated into the system's initialization and startup process, that any resources that they require are available, and that those resources can be successfully accessed by the application(s). Application debugging may also require temporary or iterative changes to a system's default logging mechanism to enable remote logging during the debugging process.

Integration and Quality Assurance

The Integration and Quality Assurance role in an embedded development project is responsible for integrating the products of the Operating System and Platform Support and application Development roles, and creating deployable and testable embedded systems that deliver the applications and services that make an embedded device unique. The team member(s) responsible for this role must have a good general understanding of the target hardware, embedded Linux system contents and deployment mechanism, and the capabilities of the custom applications and services that the embedded device that is being developed will deliver.

The team member(s) in this role is/are responsible for:

- working with OS and AD roles to produce versioned release candidates and archives of associated source code,
- deploying release candidates on test systems for integration and general quality assurance testing,
- developing and executing system and application-specific tests to ensure the quality, consistency, and performance of the services that the embedded systems and custom applications provide, and
- producing final images for ROM/Flash burn-in by manufacturing — after completing QA and certifying a release of the system and applications

The team member(s) that fills this role integrates the products of other embedded development roles, and deploys, tests, and exercises release candidates for an embedded Linux project.

Developing test code and associated infrastructure for exercising custom device drivers (if required), custom applications, and custom or related services is a part of any good internal software development project. However, developing system test software for an operating system and related system infrastructure often requires significant Linux knowledge and expertise that may not be required for traditional, application-specific testing.

After scripting local and remote test scenarios for custom software and drivers that are used on your embedded Linux platform, a popular solution for testing a Linux system itself is the open source test software that has been developed as part of the Linux Test Project (LTP). The LTP is a set of several thousand tests that are designed to exercise the Linux kernel and related features. Which of those tests you use depends on the capabilities and contents of a specific embedded Linux project, as some of the tests in the LTP may exercise features that are not germane to your project or system's capabilities.

Intellectual Property and Embedded Linux

Proprietary embedded operating system vendors delight in creating fear, uncertainty, and doubt (FUD) about the use of open source operating system and application software, especially regarding the possibility of *contaminating* your intellectual property. In this case, “contamination” means that you would have to release your intellectual property under an open source license, and can only occur if you base your software on or integrate your software into existing open source software that is released under a specific class of open source license. Luckily, you can prevent this sort of “contamination” in exactly the same way that you can avoid getting a speeding ticket — by following the rules, which in this case are the terms of the licenses under which the open source software that you are using is released.

Different open source software packages are released under a variety of licenses, all of which are designed to encourage its use, reuse, and continued improvement. Open source licenses can be divided into several general categories, including:

- Licenses that require the free redistribution of any changes or additions that you make to software that is released under that license, and and also require you to distribute your changes (or offer to do so) under that same license. The best known license of this type is the GNU General Public License (GPL), of which there are several versions.
- Licenses that enable you to link against software libraries or use software APIs that are released under a different license, without imposing any licensing requirements on your software. The best known licenses of this type are the GNU Library or “Lesser” General Public License (LGPL) and Mozilla Public License (MPL).
- Licenses that enable you to use or modify open source software without imposing any licensing or redistribution requirements beyond including copyright and licensing information for the open source software that you are using. The best known licenses of this type are the BSD and MIT licenses.

Licenses of the first type, specifically the GPL, are the open source licenses that FUD-mongers identify as “dangerous” and “viral.” Neither of these is true. You can avoid violating such licenses by following a simple rule: **Never integrate any source code that is released under the GPL into your proprietary source code.** Linux device drivers for proprietary hardware are the most common development effort that triggers redistribution concerns. If directly compiled into the Linux kernel (which is released under the GPL V2 license), the source code for those device drivers must be released under the GPL. Big deal. Proprietary embedded operating system vendors do not impose that sort of requirement, but you’re literally paying for the privilege of preventing other developers from using your proprietary hardware, which is a strange business practice if you’re trying to sell that hardware.

Some interpretations of licenses such as the GPL state that you can avoid releasing the source code for device drivers and other kernel software by only delivering those device drivers or other kernel code as loadable kernel modules, and never directly compiling such code into the kernel. If this is your plan, you should discuss this with your corporate attorneys before beginning development, and certainly before distributing or otherwise releasing your software.

This section was not written by an attorney and can’t provide a complete discussion of all available open source software licenses and their nuances — there are entire books devoted to those topics. For detailed information about open source licenses and a complete list of commonly-used open source licenses, see the Open Source Initiative.

Summary

This section provided an overview of common Linux and embedded Linux concepts, discussing the many software components of an embedded device that uses the Linux operating system. The beauty of using Linux-based open source software in your embedded development projects is that the complete source code for all of that software is freely available and eliminates the licensing and redistribution costs that are associated with most proprietary embedded operating systems and development tools.

Ultimately, there are still costs associated with using open source software. General application development costs aside, the cost of using Linux and other open source software in your embedded development projects falls into one of two categories:

- **Personnel costs** — Your development team must have sufficient Linux expertise to build cross-compilers, build and deploy custom versions of the Linux kernel, and build and deploy one or more root filesystems for your embedded Linux devices. Tasks such as these may require personnel and long-term maintenance commitments that fall outside your core areas of business and technical expertise.
- **Specialized software costs** — Embedded Linux software vendors such as Timesys provide complete, integrated, and easy-to-use solutions such as the Timesys PRO Edition that enable development teams to focus on custom software development, rather than requiring that they also become Linux wizards.

Many device manufacturers have discovered that developing and supporting Linux themselves can be a daunting task. When problems are encountered or specialized expertise is required, cost savings can evaporate quickly if projects stall in various phases of the development cycle. With rising complexity and shortened product market windows, device manufacturers are increasingly turning to commercial Embedded Linux experts such as Timesys to provide Linux expertise through offerings such as the Timesys PRO Edition and TimeStorm IDE which include commercial support, and through Timesys expert professional services.

CHAPTER 2

Building Embedded Systems with Timesys Tools

Timesys provides cloud-based and desktop development tools and associated subscription services that include technical support to simplify and expedite embedded Linux development projects. Different Timesys offerings satisfy the development, debugging, and deployment requirements of the embedded Linux platform and application development roles. Timesys subscription services help ensure that the Open Source Software (OSS) on which your projects depend is up to date, automatically delivering the latest open source updates, fixes, and enhancements so that you can focus on your own development projects. Subscribers also have access to the embedded Linux development experts at Timesys, who are available to answer questions and provide the advice you need to succeed.

Timesys development tools for embedded Linux provide a powerful and consistent development environment that helps insulate your platform and application development efforts from underlying hardware changes, simplifying the move from development boards or reference hardware to the custom hardware that you have designed for final deployment and delivery to customers.

Introducing LinuxLink

LinuxLink is the product family for all of the embedded Linux development tools and associated subscription services from Timesys. The LinuxLink umbrella covers a variety of specialized tools and associated subscription services that satisfy the requirements of different embedded Linux development roles throughout the life of an embedded Linux project. In the following sections, we will provide a quick overview of the LinuxLink FREE Edition, PRO Edition and TimeStorm IDE.

LinuxLink FREE Edition

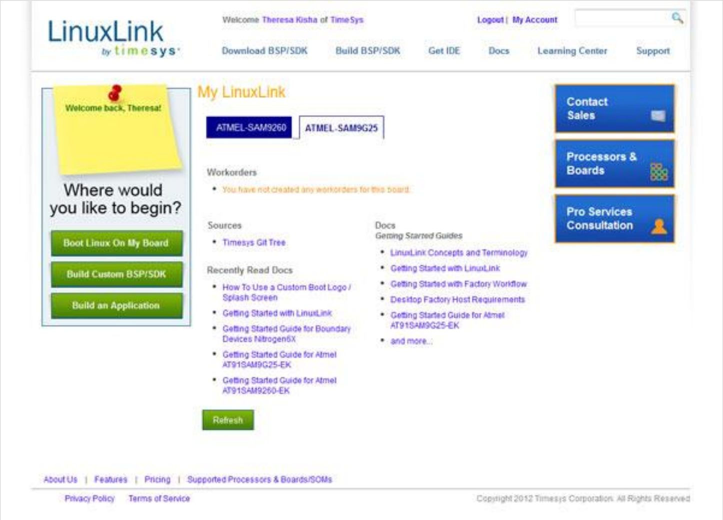
LinuxLink FREE Edition is a free, cloud-based development environment for reference hardware and vendor-specific development boards. While most hardware manufacturers deliver custom Linux platforms with their hardware, those vendor-specific platforms typically only provide a proof-of-concept subset of the open source software that you will need for prototyping your embedded Linux project. They are also rarely kept up to date, and therefore may not include the latest kernel updates and fixes to the open source software packages that they include. In addition, vendor-supplied Linux platforms are often based on different embedded Linux build environments (Buildroot, LTIB, OpenEmbedded, and so on), each with its own learning curve, configuration tools, build environment, and build products.

LinuxLink Free Edition provides web-based access to up-to-date kernel and open source package source code, making it easy to quickly prototype and benchmark both Linux and your applications on development kits and reference hardware. LinuxLink Free Edition is especially useful during hardware platform selection and comparison, providing a consistent set of kernel and software sources that:

- simplifies hardware platform comparison by delivering identical software platforms for different development kits and reference hardware,
- reduces prototyping time by providing a consistent Linux build environment across all supported architectures and processors,
- encourages accurate embedded Linux and application prototyping by providing access to a wide range of open source packages in Timesys' online repository,
- eliminates delays associated with manually installing, configuring, and maintaining a local build environment, and

- encourages efficient collaboration by hosting build tools, configuration data, and platform prototypes in a cloud-based infrastructure that is continuously available to all of your developers.

The image, below, shows the Web-based interface that you will see when accessing a LinuxLink FREE Edition subscription for the first time.



LinuxLink by timesys

Welcome Theresa Kisha of TimeSys

Logout | My Account

Download BSP/SDK | Build BSP/SDK | Get IDE | Docs | Learning Center | Support

Welcome back, Theresa!

Where would you like to begin?

Boot Linux On My Board

Build Custom BSP/SDK

Build an Application

My LinuxLink

ATMEL-SAM9260 | ATMEL-SAM9G25

Workorders

- You have not created any workorders for this board.

Sources

- Timesys Git Tree

Recently Read Docs

- How To Use a Custom Boot Logo / Splash Screen
- Getting Started with LinuxLink
- Getting Started Guide for Boundary Devices Nitrogen6X
- Getting Started Guide for Atmel AT91SAM9G25-EK
- Getting Started Guide for Atmel AT91SAM9260-EK

Docs
Getting Started Guides

- LinuxLink Concepts and Terminology
- Getting Started with LinuxLink
- Getting Started with Factory Workflow
- Desktop Factory Host Requirements
- Getting Started Guide for Atmel AT91SAM9G25-EK
- and more...

Refresh

Contact Sales

Processors & Boards

Pro Services Consultation

About Us | Features | Pricing | Supported Processors & Boards/SOMs

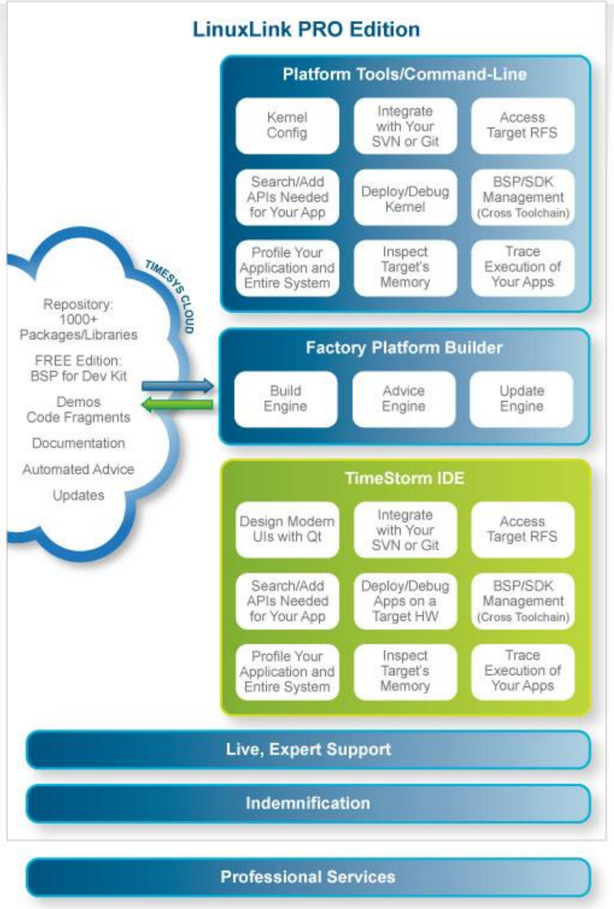
Privacy Policy   Terms of Service

Copyright 2012 Timesys Corporation. All Rights Reserved

LinuxLink PRO Edition

LinuxLink PRO Edition provides a powerful desktop build environment for embedded Linux platform customization and application development, debugging, and optimization, improving your turnaround time for configuring, compiling, and deploying new platforms and applications:

- **Platforms** — LinuxLink PRO Edition's Desktop Factory is based on the core technologies used by the web-based LinuxLink FREE Edition, but executes locally, on a desktop system. More importantly, the Desktop Factory dramatically extends the platform configuration and customization capabilities of FREE Edition, making it easy with which to experiment and refine common platform development tasks such as kernel and RFS size minimization and general performance optimization.
- **Applications** — LinuxLink PRO includes TimeStorm, a graphical Integrated Development Environment (IDE) for embedded Linux applications that is based on the industry-standard Eclipse IDE. TimeStorm is designed to satisfy the specialized requirements of embedded application development, deployment, debugging, and testing, within a familiar Eclipse framework.

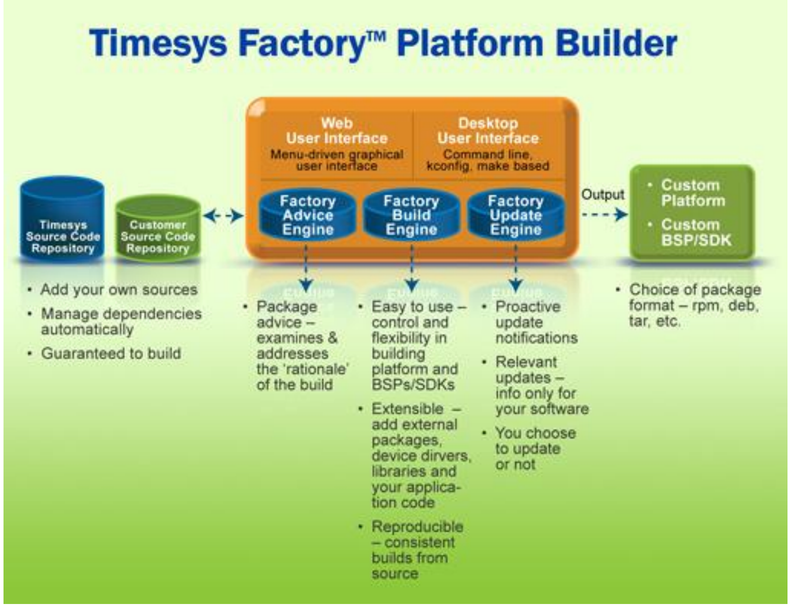

LinuxLink PRO Edition
Platform Tools/Command-Line
Kernel Config
Integrate with Your SVN or Git
Access Target RFS
Search/Add APIs Needed for Your App
Deploy/Debug Kernel
BSP/SDK Management (Cross Toolchain)
Profile Your Application and Entire System
Inspect Target's Memory
Trace Execution of Your Apps
TIMESYS CLOUD
Repository: 1000+ Packages/Libraries
FREE Edition: BSP for Dev Kit
Demos Code Fragments
Documentation
Automated Advice
Updates
Factory Platform Builder
Build Engine
Advice Engine
Update Engine
TimeStorm IDE
Design Modern UIs with Qt
Integrate with Your SVN or Git
Access Target RFS
Search/Add APIs Needed for Your App
Deploy/Debug Apps on a Target HW
BSP/SDK Management (Cross Toolchain)
Profile Your Application and Entire System
Inspect Target's Memory
Trace Execution of Your Apps
Live, Expert Support
Indemnification
Professional Services

Desktop Factory

LinuxLink PRO Edition provides the right tools for all of your embedded development efforts — fast, menu-based configuration tools for platform builders, and a familiar graphical IDE for application developers. The Desktop Factory tool in LinuxLink PRO Edition enables platform builders to:

- quickly customize, rebuild, and redeploy your bootloader, kernel, and root filesystem,
- customize BusyBox and the embedded system startup mechanism, and
- switch between toolchains that use different C libraries to reflect hardware changes and easily explore size, capability, and performance tradeoffs.

Embedded Linux projects are typically prototyped on development kits or other reference hardware, but are eventually deployed on custom hardware based on the same architecture, processors, and auxiliary hardware, often in system-on-a-chip (SoC) form. Regardless of where you are in the hardware examination, selection, and customization cycle, the LinuxLink PRO Edition provides a consistent platform development environment that easily mirrors changes in your underlying hardware by enabling you to switch or rebuild toolchains while retaining all other aspects of your existing platform's configuration.


Timesys Factory™ Platform Builder
Web
User Interface
Menu-driven graphical user interface
Desktop
User Interface
Command line, kconfig, make based
Timesys Source Code Repository
Customer Source Code Repository
Factory Advice Engine
Factory Build Engine
Factory Update Engine
Output
• Custom Platform
• Custom BSP/SDK
• Add your own sources
• Manage dependencies automatically
• Guaranteed to build
• Package advice – examines & addresses the 'rationale' of the build
• Easy to use – control and flexibility in building platform and BSPs/SDKs
• Extensible – add external packages, device dirvers, libraries and your application code
• Reproducible – consistent builds from source
• Proactive update notifications
• Relevant updates – info only for your software
• You choose to update or not
• Choice of package format – rpm, deb, tar, etc.

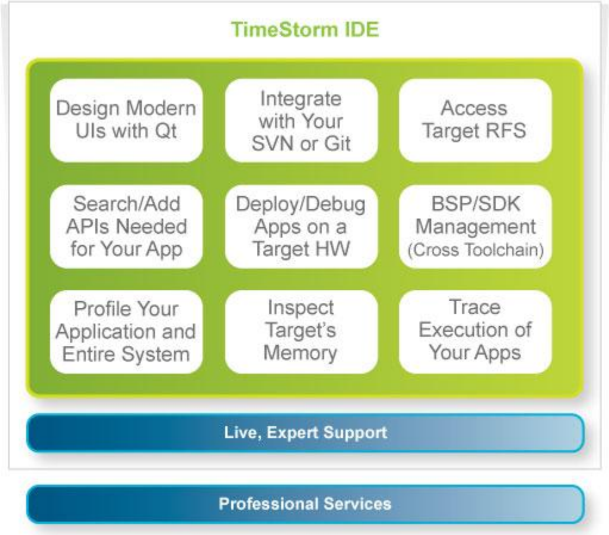
TimeStorm Eclipse-based IDE

Customized for the requirements of embedded application development, the TimeStorm IDE provides significant productivity and workflow enhancements for embedded developers. The TimeStorm IDE:

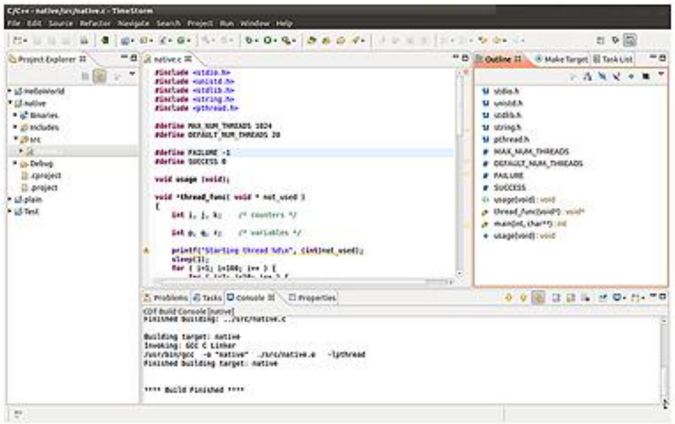
- simplifies the cross-compilation process by providing target hardware and cross-compilation toolchain management plugins,
- enables easy integration of other Eclipse plugins that are available from the open source community or third parties,
- includes templates for standard C and C++ applications, and
- makes it easy to install, execute, and debug applications on your target hardware from your desktop system.

Unlike buy-once, update-yearly products, LinuxLink PRO uses a subscription model that guarantees continuous updates to the open source software from the Timesys repository that you are using in your project. The latest versions of the Desktop Factory and TimeStorm are always available from the LinuxLink Web site. TimeStorm also leverages the built-in update capabilities of the Eclipse IDE to provide instant access to updates in any TimeStorm or Eclipse plugin that you are using.

A Timesys LinuxLink PRO Edition subscription includes direct access to Timesys embedded Linux experts and support personnel to answer your questions without resorting to a search engine. Your PRO Edition subscription also includes access to an expanded set of online documentation and tutorials on embedded Linux development topics.


TimeStorm IDE
Design Modern UIs with Qt
Integrate with Your SVN or Git
Access Target RFS
Search/Add APIs Needed for Your App
Deploy/Debug Apps on a Target HW
BSP/SDK Management (Cross Toolchain)
Profile Your Application and Entire System
Inspect Target's Memory
Trace Execution of Your Apps
Live, Expert Support
Professional Services

Embedded application developers can purchase additional licenses for the TimeStorm IDE after purchasing one or more LinuxLink PRO subscriptions. As discussed earlier, embedded Linux projects are typically prototyped on development kits or other reference hardware, but are eventually deployed on custom hardware. Purchasing additional seats of TimeStorm can jumpstart your application development process by enabling multiple developers to take advantage of all of the custom embedded capabilities of TimeStorm.



```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <pthread.h>

#define MAX_NUM_THREADS 1024
#define DEFAULT_NUM_THREADS 20

#define FAILURE -1
#define SUCCESS 0

void usage (void);

void *thread_func( void * not_used )
{
    int i, j, k;    /* counters */

    int a, b, c;    /* variables */

    printf("Starting thread %d\n", (int)not_used);
    sleep(1);
    for ( i=1; i<100; i++ ) {
```

```
CDT Build Console [native]
Finished building: ../src/native.c

Building target: native
Invoking: GCC C Linker
/usr/bin/gcc  -o "native"  ./src/native.o   -lpthread
Finished building target: native


**** Build Finished ****
```

TimeStorm IDE

Development kits from hardware vendors often use different embedded development models, ranging from roll-your-own environments to open source embedded development environments such as Buildroot, LTIB, OpenEmbedded, and many more. Each of these embedded development environments has its own learning curve, and mastering multiple approaches to embedded development extends and delays your hardware evaluation and selection process. Purchasing additional TimeStorm seats provides a consistent development environment that is independent of different vendors' approaches to embedded Linux development.

Each additional TimeStorm license includes direct access to Timesys' embedded Linux experts and support personnel to answer your questions and provide advice on embedded application development.

Summary

LinuxLink FREE and LinuxLink PRO Editions, along with TimeStorm IDE, provide you with a complete spectrum of supported tools and embedded development expertise that is the product of years of experience with embedded Linux platform and application development. Regardless of where you are in the embedded device and application development process, Timesys LinuxLink products provide significant time and cost savings that can reduce time to market, increasing profits and maximizing your return on investment (ROI) in both project personnel and technology selection.

About Timesys

Timesys is the provider of LinuxLink, a high-productivity software development framework that dramatically simplifies and speeds up embedded Linux application development. The LinuxLink framework includes the Linux kernel, cross-toolchain, application development IDE, an award winning build system called Factory, a vast library of middleware packages, software stacks and libraries, documentation and expert technical support. LinuxLink enables development teams to consistently build and maintain a custom, open source embedded Linux platform through regularly updated Linux sources, proven middleware packages, and a scriptable GNU-based build environment. LinuxLink reduces the time, resources, risk and cost associated with building a product based on open source Linux. Get more information at: www.timesys.com

Timesys Corporation
428 Forbes Avenue, Suite 2100
Pittsburgh, PA 15219
UNITED STATES
1.866.392.4897
412.232.3250
412.232.9818 (Fax)