MontaVista Linux 6

WHITE PAPER

Streamlining the Embedded Linux Development Process

Using MontaVista Linux 6 to get the most out of open source software and improve development efficiencies

ABSTRACT:

The Linux operating system has become the dominant choice for new development projects over any other embedded OS for many different reasons. This paper will outline the challenges of embedded Linux development and the obstacles that can push software engineers off the path of rapidly delivering embedded products to market.

It will then provide an overview of MontaVista Linux 6 and show examples of how MontaVista Linux 6 helps to streamline the embedded Linux development process, delivering a faster time to market.

montavista™

Table of Contents

Overview 3
Challenges of Embedded Linux Development 3
  Assembling a Software Base 3
  Creating a Development Environment 4
  Keeping Current 4
  Dangers of These Challenges 5
MontaVista Linux 6: Solving These Challenges with a Streamlined Process 5
  Software Development Kit 6
    MontaVista Integration Platform 6
    Cross-development Toolchains 7
    MontaVista DevRocket 7
  Market Specific Distributions 7
    Content Collections 8
    Pre-built Binaries 8
  MontaVista Zone Content Server 9
Working with Projects 9
  Creating a Project 9
  Build Concepts 10
  Building a Project 11
  Deploying Project Images 12
Custom Project Configuration 12
  Pre-configured Images 12
  Local Configuration Files 12
  Configuration Variables 12
  Adding packages to an Image 13
  Kernel Configuration 13
  Adding Custom Applications 13
  Configuration management 14
  Dynamic Fetching 15
Summary: Meeting the Challenges of Embedded Linux Development with MontaVista Linux 6 16
  Assembling a Software Base 16
  Creating a Development Environment 16
  Keeping Current with Software Changes 16

montavista™

Overview

The Linux operating system is increasingly being used for embedded software projects. In fact, it's become the dominant choice for new development projects over any other embedded OS for many different reasons. The rapid rate of innovation and development, direct access to the source code, the breadth of options available in the community have all contributed to embedded Linux's popularity. However, there are numerous obstacles that can push software engineers off the path of quickly delivering embedded products to market. Each of these obstacles serves to disrupt what should be a streamlined process where engineers can quickly step through the development cycle.

This paper will outline the challenges of embedded Linux development. It will then provide an overview of MontaVista Linux 6 and show examples of how MontaVista Linux 6 helps to streamline the embedded Linux development process for a faster time to market.

Challenges of Embedded Linux Development

Let's take a look at the primary challenges of embedded Linux development that every developer has to address at some point. They are:

- Assembling a software base
- Creating a development environment
- Keeping current with software changes

Additionally, the normal challenges of any embedded software development project or process must be addressed in the Linux environment as well:

- Configuration of the operating system platform
- Integration of custom applications
- Optimization for target hardware

Assembling a Software Base

One of the great strengths of open source software is the thriving and varied community of developers and software and the many alternatives this presents. However, a developer can invest many man-hours investigating the best solution for a particular development environment. In face a substantial amount of time can be spend merely understanding what the alternatives are. A developer on an embedded Linux project can waste a great deal of time simply surveying and selecting from the available components.

Commercial embedded developers may be more accustomed to proprietary products that provide a complete solution for a particular target board. In general, one does not find a complete solution in the open source community for embedded projects. Simply bringing up a system for the first time requires selecting multiple components:

- A bootloader
- A kernel base
- A toolchain
- A basic application environment

It may be necessary to acquire these from multiple independent source locations. For example, MontaVista aggregates code from over 200 different open source projects for its distributions. Once downloaded, these components have to be ported to the target

> MontaVista aggregates code from over 200 different open source projects for its distributions. Once downloaded, these components have to be ported to the target hardware, and then integrated with each other and maintained throughout the product life-cycle.

hardware, and then integrated with each other and maintained throughout the product life-cycle.

Some semiconductor vendors provide their own Linux distributions, which integrate a number of these items to provide a useful starting point. However, to fully leverage their hardware, these distributions often customize the kernel and other components in ways that are not compatible with other components developed in the broader community. Combining technologies from different sources may therefore present complicated integration issues down the road. This often makes it difficult to make use of important frameworks or applications from the open source community, and delays the start of actual product development.

In addition, semiconductor vendors generally do not provide support, maintenance or updates for their distributions. If they do, it's typically only for their largest customers, and then fairly limited

Creating a Development Environment

Most software development in the broader Linux community is based around "self-hosted development." That is, the host system (where development is done) and the target system (where the application will run) are the same. Because of this, little attention is often paid to a strict separation between the host and target environments. This frequently results in the application software having dependencies on the host environment.

For embedded developers, "cross-development" is the rule, with significant differences between the host and target environments. The host and target may be running different operating systems, and often different processors. The target hardware, deliberately limited to meet costs, is often not capable of supporting the workstation-level storage, processing power and graphics capabilities that are desirable in a modern development system.

In order to create a complete development environment, an embedded Linux developer may have to assemble his own cross-development environment. Doing this includes the following tasks:

- Acquire a toolchain for the target hardware architecture
- Create a build environment around this toolchain
- Add the bootloader, kernel and base application software to this environment
- Eliminate host system dependencies from the application software
- Port the base software to the target hardware architecture
- Integrate debugging and analysis tools

The build environment needs to be flexible, in order to support the many disparate build frameworks that are in use in the open source community. The build environment also needs to insulate the product build from the host environment, so that a consistent product build can be produced on different host systems with reliable, repeatable results.

The base software selected must further be optimized for the target hardware and the target application. This task may include integrating or writing kernel drivers for the target hardware devices, as well as adding specialized open source or proprietary software applications and frameworks.

Keeping Current

Due to the rapid rate of innovation in the open source community, keeping up-to-date on changes to different software projects may involve following many disparate sources such as:

- Community mailing lists
- Vendor hardware mailing lists
- Security forums
- Web sites
- Source repositories

Software changes in the open source community are often distributed in the form of "patches," which describe the source code differences from a previously released version of the software. If local changes have been made to the

original community software, either by the developer or the vendor who provided the base code, there may be difficulties in applying the community changes to the current software.

Vendor and community distributions will also usually only fix bugs on the latest version of a distribution, and it can be a time-intensive task to back-port these fixes to a released embedded system that is based on an older version. Sometimes bugs are addressed by systemic rather than narrowly targeted changes to the software, which not only makes back-porting more difficult, but also can introduce interface or behavioral changes that can require modification of other software in the system. Repeated thorough testing is necessary to ensure that the system continues to operate as intended, and that new issues are not introduced by the updates.

The integration of such changes is a continuous challenge not just during the development process, but also after release and over the lifetime of the product. Many members of the open source community may not be interested in helping with these issues, as they prefer to encourage developers to update to the latest version of the software. This tendency becomes stronger the older the base software version becomes, so products with long life spans in particular may require more direct expertise in the later years of their product life.

Dangers of These Challenges

Underestimating these challenges may lead to projects with ballooning costs, protracted schedule delays, or in a worst case scenario, both.

For instance, development of significant Linux expertise may be required to create a new distribution in-house. This often includes adding staff to focus on maintaining the open source software, not doing new development. This issue is frequently overlooked, as existing staff and management may underestimate the effort required to do this in a production environment, believing that open source software is an off-the-shelf solution.

…development of significant Linux expertise may be required to create a new distribution in-house. This often **includes adding staff to focus on maintaining the open source software, not doing new development.** This issue is frequently overlooked, as existing staff and management may underestimate the effort required to do this in a production environment, believing that open source software is an off-the-shelf solution.

The time required to develop such a project is also often miscalculated. Training of staff may be slower than expected due to poor, out-of-date, or non-existent documentation. Open source or semiconductor vendor code may be less robust than required for a production environment, or may not account for the limitations of an embedded environment, requiring additional development time and/or staff. Linux drivers may be unavailable for new or proprietary hardware, requiring additional development time, and perhaps highly skilled developers with specialized expertise.

MontaVista Linux 6: Solving These Challenges with a Streamlined Process

MontaVista Linux 6 (MVL6) provides a unique new approach to the challenges of embedded Linux development. It meets embedded developers where they are in the development cycle, providing a complete embedded Linux distribution and new developer tools to enable faster time to development. It enables developers to build from source to more easily customize their software stack and add product-differentiating features. The key components of MVL6 are:

- Software Development Kit
- Market Specific Distributions
- MontaVista Zone Content Server

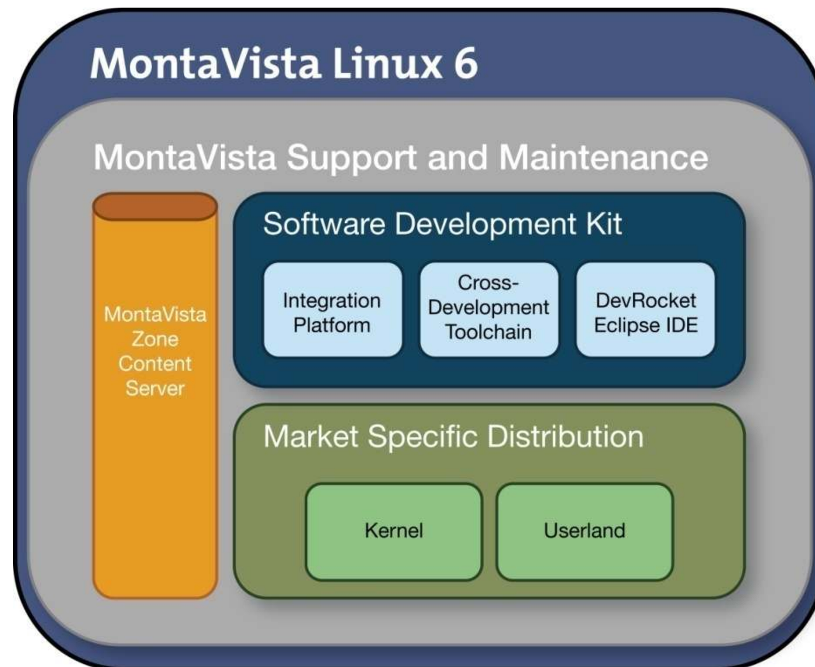

Figure 1- MontaVista Linux 6 Components

A MontaVista Linux 6 user begins by downloading and installing the Software Development Kit (SDK). It provides tools that are then used to download a Market-Specific Distribution (MSD) that includes optimized target software in the form of content collections. These content collections are accessed online from the MontaVista Zone Content Server, which provides a mirror of both source code and binaries. These are used to perform the initial build of the MSD, as well as future builds. To see the process at work view an online demonstration at http://bit.ly/TY9qD.

Software Development Kit

The SDK consists of:

- MontaVista Integration Platform
- Cross-development toolchains
- MontaVista DevRocket Integrated Development Environment (IDE)

Each of these is provided as an executable installer image that is downloaded from the MontaVista Zone Content Server. Let's take a look at each of these in more detail.

MontaVista Integration Platform

The MontaVista Integration Platform (MVIP) provides the utilities used to manage a MontaVista Linux 6 project. This includes:

- Creating a project for a specific MSD
- Downloading and updating MSD software content from the MontaVista Zone Content Server
- Configuring and building a project

The MVIP integrates the open source BitBake tool, used by the OpenEmbedded community project. One of the key benefits of basing MVIP on BitBake is access to an active community of developers of open source embedded software. MontaVista is contributing to the BitBake and OpenEmbedded projects to enhance the existing capabilities of BitBake in order to meet the needs of commercial developers. These enhancements include improved

control of locations where software is fetched from, incremental build support using pre-built binaries, and integration with debugging and analysis tools.

Other tools in the MVIP add project management capabilities and easy update support. These tools allow managing multiple versions of each content collection. Mirror site management tools also allow a completely self-contained build to be created in the local environment, with all software used in the build controlled and managed by either the project team or a site administrator. This helps to ensure you can generate repeatable builds now, and in the future.

Cross-development Toolchains

An optimized toolchain is provided for each supported target architecture. The toolchain includes:

- Compiler, assembler, linker, and associated development tools
- Critical target system libraries
- Basic debugging tools

Each toolchain is configured to operate in a cross development environment, using the libraries and header files appropriate for the target system, independent of the host environment.

MontaVista DevRocket

MontaVista DevRocket is an Eclipse-based Integrated Development Environment (IDE) that delivers a set of tools designed to streamline and automate common embedded Linux development and analysis tasks. As a set of standard Eclipse plug-ins, DevRocket can easily integrate into your existing Eclipse environment or install as a complete IDE. MontaVista plug-ins provide advanced tools for:

- Integration with MVL6 projects
- 4Remote target access
- Debugging and analysis
- Profiling and performance evaluation

Market Specific Distributions

The MontaVista Linux 6 target software is delivered as a Market Specific Distribution (MSD). Unlike a board support package (BSP) or Linux support package (LSP), an MSD provides an optimized solution for a specific hardware platform and its target market. Each MSD consists of a Linux kernel and the appropriate device drivers, libraries and applications for that processor.

Each MSD combines the best of the open source and semiconductor Linux technology for the platform. It is feature compatible with the full breadth of capabilities provided by the semiconductor's Linux technology. MontaVista provides continuous updates that integrate fixes from the open source and vendor implementations and adds key features needed to provide a complete solution for the hardware's target market. For example, a particular target board may support accelerated graphics or multimedia capabilities that require customized target software. A consumer device may require a graphical user interface and Bluetooth wireless software, while a data plane device may require remote management and network services software.

As mentioned previously, the majority of the MontaVista Linux 6 target software and host utility code is not provided during the SDK installation. It is provided as downloadable content collections on the MontaVista Zone Content Server. We'll discuss building an MSD later on when we talk about projects.[1]

[1] For more detailed information on MVL6 components, see the MontaVista Linux 6 Technical Brief

Content Collections

A content collection is a group of MVIP recipes (discussed in more detail below) and supporting files. Separate collections are provided for the kernel and application software. Distinct application software functionality is also provided in separate collections. Collections are distributed and updated independently.

The collections are structured as archives which contain patches that are applied to base open source project release archives. If desired, these base source archives may be downloaded separately from the MontaVista Zone Content Server. A collection archive may then be updated to implement changes as additional patches without requiring a change to the base source archive or requiring it to be downloaded again. This saves time and storage space compared to downloading a complete archive of the source code each time.

A developer creates a new project based on a particular MSD using the content tools provided by the MVIP. These tools contact the MontaVista Zone Content Server to determine the right set of "collections" that comprise the MSD code. The "mvl-project" tool creates a directory for the new project which contains configuration files that specify the content collections to be used for the project. It downloads the content collections themselves, and can optionally download all the base source archives and pre-built binaries for the MSD and add them to the project directory. This project directory is intended to contain all the needed configuration information to create a reproducible project build. The developer is freed of the burden of downloading content from potentially hundreds of different sites, and doesn't have to worry about the validity of the downloaded content, or his initial build failing three-quarters of the way through due to some missing content file.

This project directory is intended to contain all the needed configuration information to create a reproducible project build. The developer is **freed of the burden of downloading content from potentially hundreds of different sites**, and doesn't have to worry about the validity of the downloaded content, or his initial build failing three-quarters of the way through due to some missing content file.

Pre-built Binaries

Pre-built binaries are supported to allow reuse of unchanged binaries from previous builds to accelerate the build process. These binaries act as cached intermediate build information.

The configuration settings for the build are compiled into a configuration signature that is associated with each pre-built binary. If a pre-built binary with the correct configuration signature is available, the build process can use it rather than performing a complete new build. If the configuration has been changed so that no matching pre-built binary is found, the build process will automatically perform a complete build to create a new binary.

The MontaVista Zone Content Server supplies pre-built binaries for the default configuration of each MSD. The use of pre-built binaries can allow a build to complete very quickly when there are relatively few configuration changes. This provides a very quick startup scenario for new projects using supported hardware in the default configuration.

A developer can also use pre-built binaries generated from their own builds for both system software and locally developed content. They can be used to improve the efficiency of successive internal builds of a project as development continues, or to provide a stable and fast-building base for multiple dependent projects on a shared common platform.

montavista™

MontaVista Zone Content Server

The MontaVista Zone Content server provides access to the content collections for all MSDs. From the MVIP he initiates the `mvl-project` tool, which provides the ability to:

- List available MSDs
- List the collections associated with an MSD
- List the available versions of each collection
- Download collection releases, base source archives, and pre-built binaries
- Update collection releases, base source archives, and pre-built binaries

When a developer is logged in, he will see only the MSD's he has access to. The `mvl-mirror` tool in the MVIP provides the ability to copy content from the MontaVista Zone Content Server to one or more local content servers. Maintaining a local mirror of all pertinent source can benefit companies in multiple ways:

- All software is controlled locally, and no changes can be introduced without explicit local action
- All software can be placed under source control, and included in regular backups
- Individual developers do not require Internet access to manage projects or perform builds. Internet access is required only when the mirror site is updated
- Builds can be performed with greater consistency and reliability when not dependent on external network accesses

> The MontaVista Zone Content Server supplies pre-built binaries for the default configuration of each MSD. The use of pre-built binaries can allow a build to complete very quickly when there are relatively few configuration changes. This provides a very quick startup scenario for new projects using supported hardware in the default configuration.

Local content mirrors help you maintain control over all software so that it may be placed under source control, be included in regular backups, not require Internet access and improve build performance.

You can configure your local content server environment to match your project requirements. The local content server can contain a single MSD or multiple MSDs, and can contain multiple releases of each collection. A development team that supports multiple projects for multiple targets can maintain all the software in a centrally managed location. A large company with multiple project teams can likewise maintain, update, and manage the software centrally for all teams, perhaps with a separate administrator. A site can also have multiple content servers, each supporting different project teams if so desired.

Where the MontaVista Zone Content Server is mentioned in the preceding or following text, a local content server mirror can generally be substituted.

Working with Projects

Now that we've described all the components of MVL6, we are ready to begin a project.

Creating a Project

Once the SDK is installed and the `bin` directory is added to the user's PATH, a project can be created using a simple command:

```
mvl-project -m msd_name project_dir
```

where *msd_name* is the identifier for a particular MSD and *project_dir* is the directory that should be created for the new project. For instance:

```
mvl-project -m x86-target-2.6.28 $HOME/myproject
```

would create a new directory *$HOME/myproject* with a default project configuration for the *x86-target-2.6.28* MSD. For other platforms such as TI, Freescale, ARM, MIPS, etc. the MSD's have a similar naming convention.

As the project is created, the `mvl-project` tool will:

- Download the kernel and application software content collections associated with the MSD from the MontaVista Zone Content Server
- Set up default configuration files for the project
- Create the setup.sh script to establish necessary settings in the user's environment

Additional options can also cause all the associated base source archives and pre-built binaries to be downloaded into the project directory. If these are not downloaded at project creation time, they can be fetched as needed during the build from the MontaVista Zone Content Server.

The next step is to build the project.

Build Concepts

The MVIP build environment supports scripts, classes and other metadata used for building target software and host utilities. The content collections provide information in this form. Understanding a few common concepts will help to grasp the structure.

Recipes are BitBake script files that define how to build a particular target object. Actions can be defined for various stages of the build process, including:

- Fetch**:** downloads the base source archives
- Patch**:** applies patches containing modifications to the base source archives
- Configure**:** configures build settings for a particular package
- Compile**:** compiles the software
- Install**:** creates the desired installation structure for the built package contents
- Package**:** bundles the installed package contents into archives
- Clean**:** removes temporary files created by the build process

Classes can be used to define actions that are common for a large number of recipes. For instance, a class can be used to define the common build actions for software based on the open source autotools build environment. Individual package recipes can simply include a class by reference, and replace or extend the default actions only where necessary.

Dependencies can be used to declare that a given package requires the support of another package, either at build time or at run time. The build process attempts to ensure that any build time dependencies are built before the dependent packages, and that run time dependencies are automatically included in any resulting images.

Tasks are recipes that are associated with no unique software; they contain only dependencies and sometimes build actions. Tasks can be used to provide an easy way to include a complex set of functionality constructed from a number of different packages. By declaring dependencies on specific packages through a task, users of the task are freed of having to know these details themselves.

Images are recipes that define deployable outputs from the build. These outputs can include bootloader and kernel binaries, as well as filesystem images containing system and user application software.

montavista™

Building a Project

Once a project is created, the default image can be built using just the commands:

```
cd $HOME/myproject
source setup.sh
bitbake default-image
```

In order to perform the build, BitBake will parse recipe information from the configured content collections, resolving dependencies by adding additional recipes to the image where necessary. It will then break down each recipe into specific tasks such as:

- Fetching and unpacking base source archives
- Applying patches
- Configuring the build settings
- Building the software
- Packaging the software into deployable units
- Deploying the software into target images

These correspond closely with the recipe actions noted above, with the addition of some system image creation tasks. These tasks will be scheduled and executed as necessary in order to perform the build.

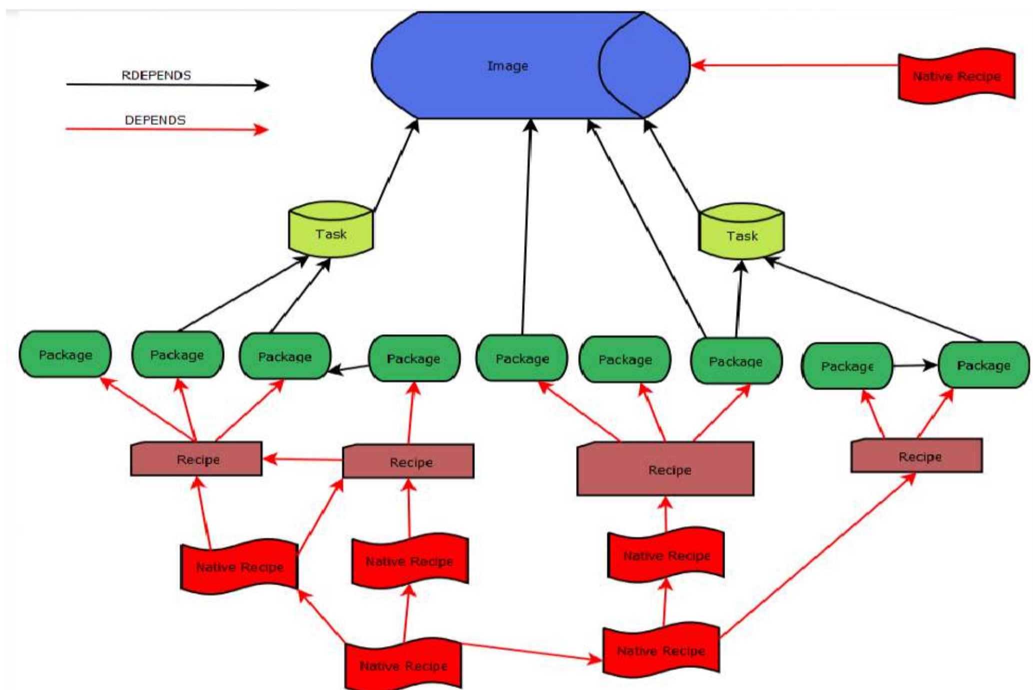

Figure 3 – Overview of MVL6 Project Build

During the build, base source archives will be downloaded as needed from the MontaVista Zone Content Server. As each package recipe is built, the configuration signature is generated and a check is made for appropriate pre-built binaries in the local project or on the content server. If an applicable pre-built binary is found, the step of building the software can be pre-empted. Otherwise, the build proceeds normally.

Deploying Project Images

Once the build is complete, the output images can be deployed to the target hardware and executed. Most commonly, the outputs are:

- A kernel binary image, which can be loaded using a target bootloader from a network location, flash memory or other media
- A filesystem image, which can be written to flash memory or other target media, or can be mounted from the host system using the NFS network filesystem protocol to provide quick turnaround for developers making and testing changes

Custom Project Configuration

Pre-configured Images

A number of pre-configured images are provided with MontaVista Linux 6. Each MSD can define default configuration settings for these images:

- Default-image: configures a system with many standard Linux capabilities
- Small-image: configures a simple working system, appropriate as a base for building up for a particular application
- Devel-image: configures a system suitable for development activities
- Initramfs-image: configures a very small environment suitable for an in-memory ramdisk

These images provide a starting point that can be extended with additional system and custom applications. The "small-image", in particular, is useful as the basis of a system that includes little more than the target applications required for the project.

Local Configuration Files

A tremendous amount of project customization can be performed using the single project file "conf/local.conf". A simple version of this file is created by the `mvl-project` tool. Many system options can be modified by setting *configuration variables* in this file. A full system configuration can be defined in this single location. Keeping configuration changes localized in this way can make it easier to update collections with newer versions without encountering conflicts.

For more complex operations, complete custom recipes and classes can be created. The project directory "collections/custom" is included in the project search path to allow custom recipes to be added to this directory easily.

Configuration Variables

Many system options can be modified by setting *configuration variables* in the local.conf file. A variable can be set by a simple declaration:

```
KERNEL_CONFIG_KGDB = "y"
```

For many system options, adding to the default rather than overriding it is preferable. It is possible to add to a variable by using the "+=" operator or adding the "_append" string to the name:

```
CFLAGS += " -Wall"
CFLAGS_append = " -Wall"
```

Variable settings can also be directed to a specific recipe, as opposed to being set globally. In this example, the CFLAGS variable will be modified only for the "ncurses" recipe:

```
CFLAGS_ncurses += " -Wall"
```

The CFLAGS variable can be used to customize compiler options system-wide or for a single recipe. There are many such variables; for instance, configure script options for software which uses the autotools framework can be set as follows:

```
EXTRA_OECONF_links += " --enable-javascript"
```

Adding packages to an Image

Perhaps the most common customization is to add selected packages to an image. This is easily done as follows in local.conf:

```
IMAGE_INSTALL_append = " ntp"
```

The object can be any package included in the search path. This can be a package included in the MontaVista Linux 6 collections or a custom one defined by the developer, either locally developed or downloaded from the open source community. It can also be a task; as mentioned earlier, this may be a more convenient way to include selected functionality without needing to list all the specific packages required.

Kernel Configuration

For an embedded Linux system, it is often desirable to adjust the kernel configuration for the target system.

The Linux kernel configuration is controlled by a set of configuration variables, somewhat like the MVIP configuration variables. In fact, the MVIP makes it possible to set kernel configuration variables directly from the local.conf file. For instance, the CONFIG_KGDB variable can be set as follows:

```
KERNEL_CONFIG_KGDB = "y"
```

It is also possible to replace the whole kernel configuration with a specified configuration file from the project directory (represented by the TOPDIR variable):

```
KERNEL_CONFIG = "${TOPDIR}/kernel.config"
```

Other facilities provide the ability to apply patches to the kernel and to make other changes.

Adding Custom Applications

Almost any embedded Linux project will require the addition of custom applications. It will normally be necessary to add a recipe for each custom application. Using a supported software framework can make this very easy.

For instance, the recipe for `cpio`, which uses the autotools framework, looks like this:

```
DESCRIPTION = "GNU cpio is a program to manage archives of files."
HOMEPAGE = "http://www.gnu.org/software/cpio/"
SECTION = "console"
LICENSE = "GPL"
PR = "r4"

DEPENDS += " texinfo-native "
```

```
SRC_URI = "${GNU_MIRROR}/cpio/cpio-${PV}.tar.gz \
           file://install.patch;patch=1"
S = "${WORKDIR}/cpio-${PV}"

inherit autotools

do_install () {
        autotools_do_install
        install -d ${D}${base_bindir}/
        mv ${D}${bindir}/cpio ${D}${base_bindir}/cpio.${PN}
        mv ${D}${libexecdir}/rmt ${D}${libexecdir}/rmt.${PN}
}

pkg_postinst_${PN} () {
        update-alternatives --install ${base_bindir}/cpio cpio
cpio.${PN} 100
        update-alternatives --install ${libexecdir}/rmt rmt rmt.${PN}
50
}

pkg_prerm_${PN} () {
        update-alternatives --remove cpio cpio.${PN}
        update-alternatives --remove rmt rmt.${PN}
}
```

The details of writing recipes are beyond the scope of this paper, but the above shows a number of useful features:

- The SRC_URI specifies the original location of the base source archive, as well as a local patch file to be applied
- Most of the work for implementing an autotools recipe can be done by the "inherit autotools" directive
- Selected actions can be overridden, such as for the do_install action
- Build-time dependencies can be specified in the DEPENDS variable

Many other powerful capabilities are available:

- The SRC_URI can specify files from a local directory or a remote web server. It can even access files directly from several source control systems, and can be extended to support others
- The RDEPENDS variable can specify run-time dependencies that should be included in the target image to support the packages built from this recipe
- The build system automatically allocates the installed files to multiple packages based on file type and installation location. These allocations can be overridden and modified by the recipe to create whatever packages are desired and allocate files to these packages as needed

Refer to the OpenEmbedded documentation for more information on creating or modifying recipes.

Configuration management

Software configuration management refers to the systematic control of changes to a software product. It encompasses source control systems, defect reporting systems, and other documentation and processes. Proper configuration management is essential to create a well-understood product that remains manageable over its expected lifetime, as well as to protect the essential elements that comprise the product software.

A variety of different tools are in use for software configuration management. The MontaVista Linux 6 structure allows a tremendous amount of flexibility in the structure of a developer's environment in order to make it possible to integrate with different tools and architectures. You have the flexibility to take a centralized or more distributed approach.

The MontaVista Linux 6 structure allows a tremendous amount of flexibility in the structure of a developer's environment in order to make it possible to integrate with different tools and architectures. You have the flexibility to take a centralized or more distributed approach.

The project directory is designed so that it can contain all the configuration information required for the project, and this directory can be committed to source control. It is also possible to include custom recipes and configuration files in this directory, and even entire source code subdirectories. The `mvl-project` tools supports downloading the base source archives and pre-built binaries to this directory, so the entire project can be contained in this directory if so desired, and committed to source control from there.

For those who prefer a more distributed structure, custom recipes and configuration files can be stored outside of the project by adding the external location to the search path. Custom application source code can be accessed in a wide variety of ways, including:

- An archive file on a local or remote system
- A source tree on the local system, which may have been checked out from source control
- Directly accessed from a source control management system

It is possible to integrate support for various source control systems directly into the MVIP. During the build, recipes are then able to fetch a particular revision of software under source control during the build process, without requiring it to be checked out separately first.

Dynamic Fetching

For the MSD content collections, the base source archives and pre-built binaries can either be downloaded in bulk and stored locally, or they can be downloaded as needed during the build. This "dynamic fetching" capability allows a build to download only the sources and binaries needed for a particular project build, and only when needed. When used with a distributed structure, which stores the MontaVista Linux 6 software in a central location for reference by multiple projects, this allows for a very efficient use of space and simplified management for the individual projects.

The content collections can be accessed either from the MontaVista Zone Content Server or from one or more local content server mirrors. In the latter case, a single administrator can be responsible for updating the mirror content from the MontaVista Zone Content Server, with all other developers accessing the content from the local mirror.

montavista™

Summary: Meeting the Challenges of Embedded Linux Development with MontaVista Linux 6

To summarize, how does MontaVista Linux 6 help to meet the challenges of embedded Linux development as discussed in this paper? Let's take another look at each of the challenges we discussed at the beginning of this paper.

Assembling a Software Base

MontaVista Linux 6 provides a complete target environment for supported target hardware, including a configured kernel, a bootloader where necessary, target application software, and host tools.

Creating a Development Environment

MontaVista Linux 6 provides a powerful development environment, including optimized compilers and libraries, debugging and analysis tools, and a flexible build environment capable of building all target software and many host utilities from source. The build environment supports efficient incremental builds, access to multiple source control systems, and powerful extension capabilities.

Keeping Current with Software Changes

Regular updates of kernel and application content collections are provided through the MontaVista Zone Content server. These updates are easily integrated into existing projects using the included tools. The build configuration framework allows local customizations to be stored separately from the base software, which makes updates easier to integrate.

©2009 MontaVista Software, Inc. All rights reserved. Linux is a registered trademark of Linus Torvalds. MontaVista and DevRocket are trademarks or registered trademarks of MontaVista Software, Inc. All other names mentioned are trademarks, registered trademarks or service marks of their respective companies. MVL06WP0909

MontaVista Software, Inc.
2929 Patrick Henry Drive
Santa Clara, CA 95054
Tel: +1.408.572.8000
Fax: +1.408.572.8005
Email: sales@mvista.com
www.mvista.com