The right approach

When it comes to choosing between Embedded Linux or a proprietary real-time operating system, there are many different considerations to take into account, as **Marcus Nissemark** explains

An Embedded Linux distribution with its kernel and software packages provides a broad set of ready-made software, which is attractive to managers and developers. Although it requires considerable knowledge of the dependencies and configuration details, it is frequently used in the industry. A proprietary Real-Time Operating System (RTOS) with a minimal code base and limited middleware availability however, can be just as capable of meeting the same software requirements, but, with a different approach. There are many considerations when selecting between Embedded Linux or a proprietary operating system. Not just cost, but also complexity; ease of use; advanced compile and debug tools; time to market; developer productivity; and safety and security concepts.

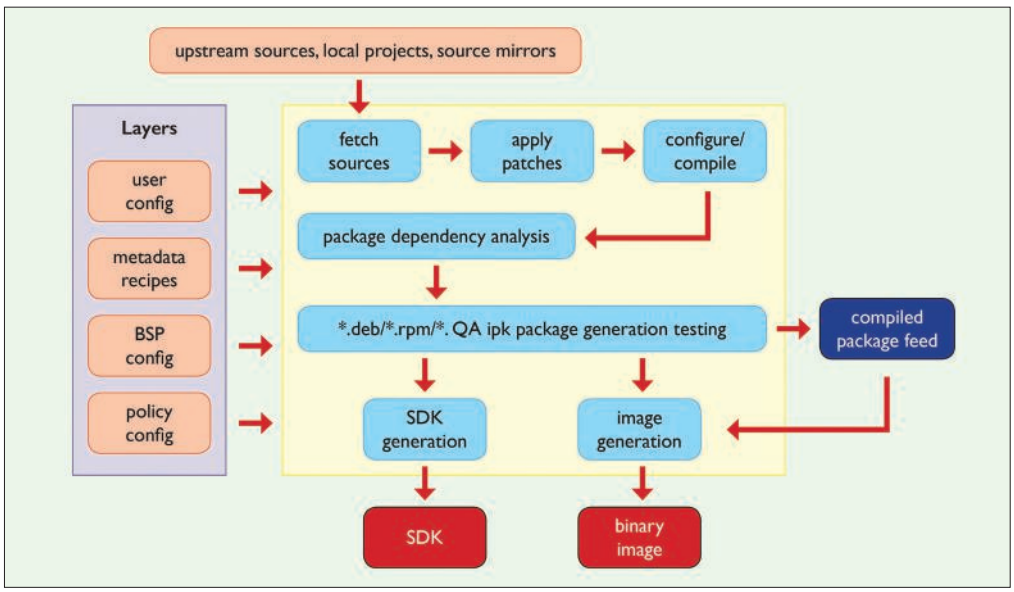

Figure 1. The Yocto Project workflow illustrating the complex scenario

Hardware choice

The challenge starts at the hardware level. Many popular SoCs support Linux, but it is often limited to the silicon vendors' dedication to deliver, support and upstream their Linux kernel and user space contributions, effectively supporting only a subset of the open-source components. For instance, NXP supports the popular NXP i.MX6 Cortex-A9 SoC through the Yocto project for a limited set of recent Linux kernels, or through the LTIB build tool for older releases. This may restrict users to one Linux kernel or middleware version, or one build system, and the features that this supports.

Depending on requirements, a user may end up having to implement functionality in existing middleware or port code to the build system. This is likely to cause significant unexpected work as the intention was to reuse existing software packages.

RTOS vendors normally work with silicon vendors, enabling each other's business by supporting the SoC in question. It is important to ask them about their support, as they often resort to their partner network and ecosystem for supporting extended middleware such as graphics or connectivity software. They can provide significant help and valuable information about the use case. They also can provide features like security enhancements, functional safety compliance, or hard real-time capabilities that a Linux solution cannot.

The requirements lead to the usage of the Yocto project and its build system. Yocto is an open-source project, driven by the Linux Foundation, whose focus is on improving the software development process when building Embedded Linux systems. It is a suitable solution for many Embedded Linux applications, as long as the work associated with it is understood.

The Yocto project can be tailored and configured, but requires significant knowledge, and is perceived, by some, as a bloated solution that needs a lot of disk storage and long build times. The default setup of a Yocto project build, downloads a significant set of build scripts and software, for example recipes and packages that are used for configuring, building and installing components to your Embedded Linux system.

The Yocto recipes will incorporate a cross-compiler from a third party or build one from source. If this is not managed correctly, it can create a configuration mismatch.

Additionally, the step of building a cross-compiler for the target bloats

the host installation and build process with unnecessary files and extra time, and causes an unnecessary risk from a test and validation perspective. An RTOS like INTEGRITY usually requires a validated version of the toolchain/compiler that is mandatory to use, and some commercial Linux alternatives bundle this into their Yocto support. This helps to remove risks associated with toolchain configurations, and simplifies the process for the developer.

The next step is configuration management of the Embedded Linux system within Yocto. This consists of a Linux kernel, hardware related patches and a number of open source packages, each requiring their own configuration setup, build process and installation. It is easy to get lost trying to navigate between what the hardware vendor supports, what the user requirements are, and the actual available and tested software solutions.

Understanding all of the configuration possibilities and package dependencies is one of the bigger obstacles encountered when developing an Embedded Linux system. For instance, adding support for a new software package may bring in a dependency to another software package or another version of that package, which is not included in the build system. This quickly becomes hard to resolve, and causes a cascading effort of changes required in your system, none of which were initially planned for. This type of dependency dilemma is similar to a 'top down' approach to software package configuration management. Each component or package must be configured to require fewer dependencies or to scale to requirements. This can consume many hours.

For a proprietary RTOS, there are still dependencies to various versions and components supported by the vendor or their partner network, so choice is limited to the software solutions they support. However, the proprietary RTOS approach to configuration management is normally to include only the components required, using a simpler 'bottom-up' approach. In other words, far less time will be spent configuring components since they are already pre-configured for the specific use case by vendors. This helps reduce effort and puts focus on to the user's own software development.

Developer focus

Using Embedded Linux solutions leverages existing software capabilities and saves development time. If developers spend time on other activities, there is no developer focus. Tools beyond the compiler can address that, where powerful debuggers, Integrated Development Environments (IDEs), or other support tools make developers efficient. All Embedded Linux systems and some proprietary RTOSes rely on the well-performing GNU compiler gcc. The GNU debugger requires extensive configuration and knowledge to get it to support debugging properly, partly mitigated by the usage of Eclipse based configurations and pre-configured setups. In the end, many developers tend to use printf() debugging, mainly due to lack of better tools, or lack of knowledge or time to properly configure the debugging solution.

Generally, RTOS vendors put extra effort into the tools and their debugging solutions are pre-configured and work out of the box. These solutions are also bundled with additional analysis and performance monitoring tools that aid development. Application development for any operating system is a straightforward task when using common APIs like POSIX, socket APIs for connectivity, and OpenGL for advanced graphics. The application code does not depend on the choice of RTOS. Easy access and properly setup tools make developers more efficient, increasing their focus and productivity.

There can still be a need for education and support. RTOS vendors provide professional support services that really can make a difference when problems occur, and they provide professional training courses by their own personnel. Embedded Linux training is available from various stakeholders, including professional support services, and there is online support in abundance. Just remember that searching for suitable support may be a time-consuming challenge.

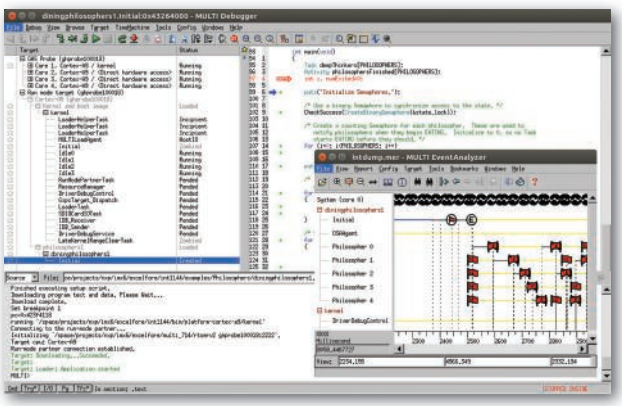

Figure 2. The MULTI Debugger from Green Hills Software

Conclusion

Embedded Linux is attractive at first sight because of its low upfront costs, but make sure you consider the following:

Time to market – Do not underestimate the time needed for problem resolution, configuration management and software package dependencies;
Developer efficiency – Efficient, easy to use and pre-configured tools make developers more efficient, allowing them to find and resolve more bugs and provide better software quality;
Engineering Time – Engineering time is a cost related to the project; developers should focus only on the actual application development; and
Alternative requirements – Functional safety, high security or real-time characteristics can be hidden features that strengthen your end product.

Author details
Marcus Nissemark is a Field Applications Engineer at Green Hills Software