LINUTRONIX

Yocto or Debian
Build system for Embedded Devices

White Paper

Executive Summary:

With Yocto, a seemingly de-facto standard for generating (embedded) Linux BSPs has established itself in recent years. However, you should keep in mind that Yocto is a tool to create your own distribution (for your device). What you notice first, is the flexibility to quickly make your own adaptations and also to create variants.

In the long run over the life cycle of a device this means that all of the maintenance of security patches has to be done by the user. The resources required for this can take on considerable proportions and more than just eliminate the apparent advantages of the initial product launch.

The distributions commonly used in the IT world have still not been accepted in the embedded world for various reasons. Debian, as for example, actually is suitable for the embedded market because it supports various CPU architectures. The seemingly lacking flexibility in the configuration of individual software packages (e.g. own translation, own selection of packages) can be created by additional tools like ELBE. With ELBE a Debian based BSP is at least as flexible and easy to create as with Yocto. Even better, once a BSP has been created, it can be completely restored to exactly the same version at any time, thanks to the XML file, which contains all the necessary information.

Due to the Debian sources being permanently maintained, the maintenance of the BSP created in this way can be done directly with the patches stored there for the individual packages. This significantly reduces the effort for security and bug fixes.

Although at the beginning of a project seemingly "cheaper" and "faster", a Yocto based approach to creating a BSP over the lifetime of a product will ultimately be significantly more expensive than creating an identical BSP using Debian packages.

Embedded developers are often faced with the task of having to commission new hardware for a new project. The task to be solved consists essentially of porting the bootloader, operating system and third-party software components to the new hardware. The way this is carried out should be

- reproducible
- maintainable in the sense of updatability in the later product life cycle
- secure in terms of data security
- and without problems related to the used CPU architecture.

Let's first clarify the term „Linux“ at this point. Linux is, strictly speaking, an operating system kernel. The combination of (Linux) kernel and all the other software like drivers, daemons, utilities (i.e. the root file system (rfs)), bootloader etc. tends to be called Linux, but should be called Linux distribution. In reference to a specific hardware platform, this is often referred to as a Board Support Package (BSP).

Fig. 1 shows the typical structure of an embedded Linux platform as described above.

To get to a platform like the one shown here, but also to develop relating application programs (userland code), you need a so-called toolchain, which is executed on a development computer. A toolchain is a systematic collection of programs that are used to create a product (usually another program or a system of programs). The name explains itself as the tool programs are usually used one by one in the form of a chain.

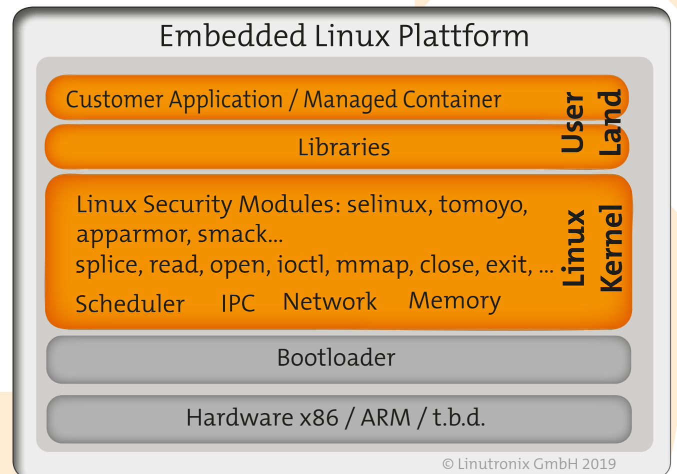


Figure 1: Embedded Linux Platform (BSP, Distribution)

A simple toolbox for software development includes a text editor for creating the source code, a compiler, a linker for creating executable programs, libraries for accessing the public routines of the operating system, and a debugger.

SDK – a Software Development Kit is a fundamental collection of necessary components such as compilers, utilities and information to develop software for an operating system. The basis for this is the so-called toolchain. The exact content of an SDK can differ significantly from case to case, as it varies in complexity depending on the subject matter. Usually the kit contains at least the necessary APIs for an existing system and documentation, which give the developer information about available interfaces or the general structure. Some build systems for Linux provide application developers with an SDK for easy use.

How does a developer get a BSP / distribution for the hardware he uses?

There are several different procedures. Apart from the so-called Build Your Own Linux, you can either use an existing distribution or create your own.

Let's take a look at two common examples of these approaches. Yocto on the one hand is a tool for creating your own distribution. And Debian on the other hand is a widespread distribution that is also used as a basis for many other distributions. Debian itself, however, is not a tool for creating a distribution.

Let's first look at Yocto. Further information can be found at the following link:

www.yoctoproject.org

Yocto defines itself as follows: The Yocto project is defined as „an open source collaboration project that provides templates, tools and methods to help you create custom Linux-based systems for embedded products regardless of the hardware architecture." It is a collection of recipes, configuration values, and dependencies used to create a custom Linux runtime image tailored to your specific needs.

A typical build process with Yocto looks like this:

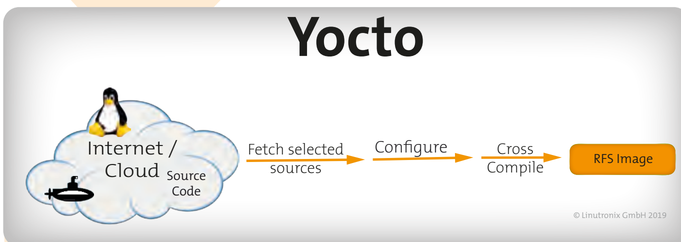


Figure 2: Yocto Build Process

In a first step, the required software packages are defined, then searched on the Internet and then down-loaded to the development computer / server. The loaded software components are cross translated, whereby the configuration(s) for the compilation process can be specified individually. After the compilation, the components are assembled, depending on the storage medium used there, to an executable image for the target system. All these processes are controlled by the so-called „recipes" of Yocto.

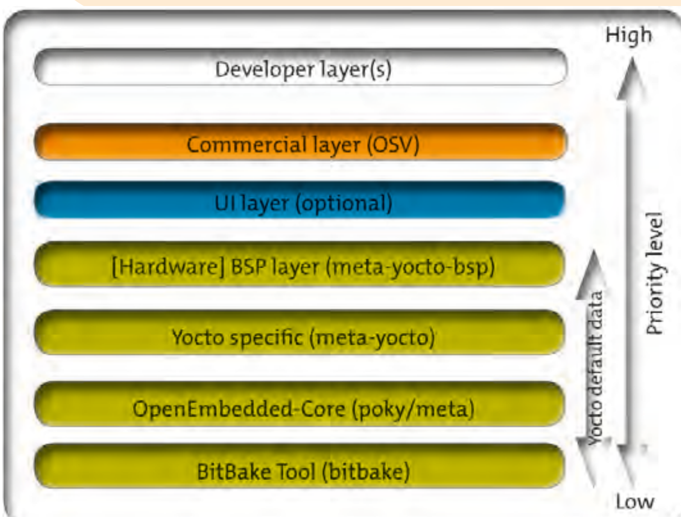


Figure 3: Yocto Layer Structure

But how does building a distribution / BSP with Debian work?

Here you have to decide between using Debian as it is or using a tool like ELBE (Embedded Linux Build System).

Further information about Debian can be found at the following link:

https://www.debian.org/

Further information about ELBE can be found at the following link:

https://elbe-rfs.org/

Figure 5 shows what a typical „classic" build based on Debian may look like.

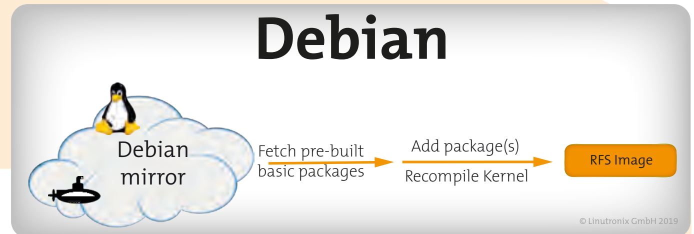


Figure 5: Debian Build Process

The user can quickly create a BSP based on binary packages (available from the Debian Mirror servers) by simply merging them together. The only way to include a personal configuration here is to replace the Debian kernel package by a self-compiled kernel.

Debian and customized adaptability

The quickness of achieving a (first) GNP in this way, however, compromises flexibility. You have to accept the packages in their binary form just as they are.
To change this and give the user full flexibility about his BSP while using Debian, Linutronix has developed the tool ELBE (Embedded Linux Build Environment). With ELBE, a build process for a target system appears as shown in Fig. 6.

To put it simple, ELBE allows you to retranslate any component, including your own software, and store it as a Debian package. This happens in a virtual machine and can be done both natively and crosswise. Native translation solves many problems that can occur during cross-platform build as a result of dependencies. The RFS itself can be created for many different media (SDcard, NAND, network, etc.), regardless of the method chosen for creation.

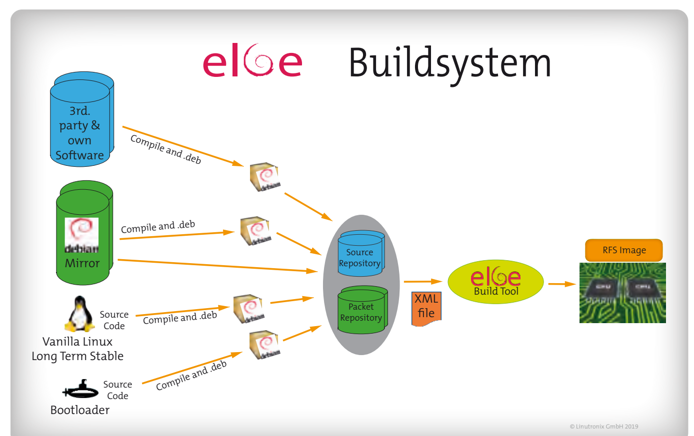


Figure 6: ELBE Build System - Overview

All important information, configurations etc. are stored in an XML file. This allows ELBE to easily reproduce the target image. The XML file in ELBE represents what Yocto calls the collection of recipes, layers and meta-layers, but more clearly and easily recognizable (no overwriting by later recipes or layers).

ELBE does give the Debian approach not only the same flexibility and configurability known as the characteristics of the Yocto approach, but additionally combines this with the advantages of a maintained distribution.

Maintenance – not only Security

The two approaches differ greatly in this respect. Yocto is, as mentioned above, not a distribution, but only a tool to create a distribution. Although there is a new Yocto version at regular intervals, its contents are not defined. Neither is there any guarantee that the personal distribution created with Yocto (my kernel and my RFS) will be completely covered by the new release. And since there is no definition of what is included in the new release, it can happen that a buggy software version is actually included in the new Yocto release, without security patches, etc.

The only remedy here is a manual check to see which components have changed in the new Yocto release, whether all the necessary security patches have been integrated, and, if this is not the case, do it yourself.
And even if a software, let's say a library, contains a required security fix, it doesn't mean that the new library version is ready to use. Because it may behave differently or an API might have been changed. All of this has to be checked and validated by the user.

Again, an intermediate release must be performed manually, if the approximately 6-month interval between two releases is too long. You would have to search, which software needs a security patch, search and apply the patch, recover this process for the entire RFS and then build the new board support package with Yocto.

Debian is quite a different story. Here the source versions the packages are built from, are maintained by the responsible maintainer as well as by the Debian Security Team. Even if the source project is further developed (e.g. a new version is created), the functional patches are collected by the maintainer and, if necessary ported back to the source level of the Debian version. This also applies to any security patches. These, too, will be ported back if necessary. So, you can be assured that over the years the software component is up to date and still compatible to the original version. Particularly with long-running products, this is an immense (cost) advantage, as further, time-consuming work on your own application, for instance, can be avoided.

Comparing the two build system approaches

The following chapter contains a (incomplete) list of the most important points of Yocto and Debian (Elbe) relevant for a developer.

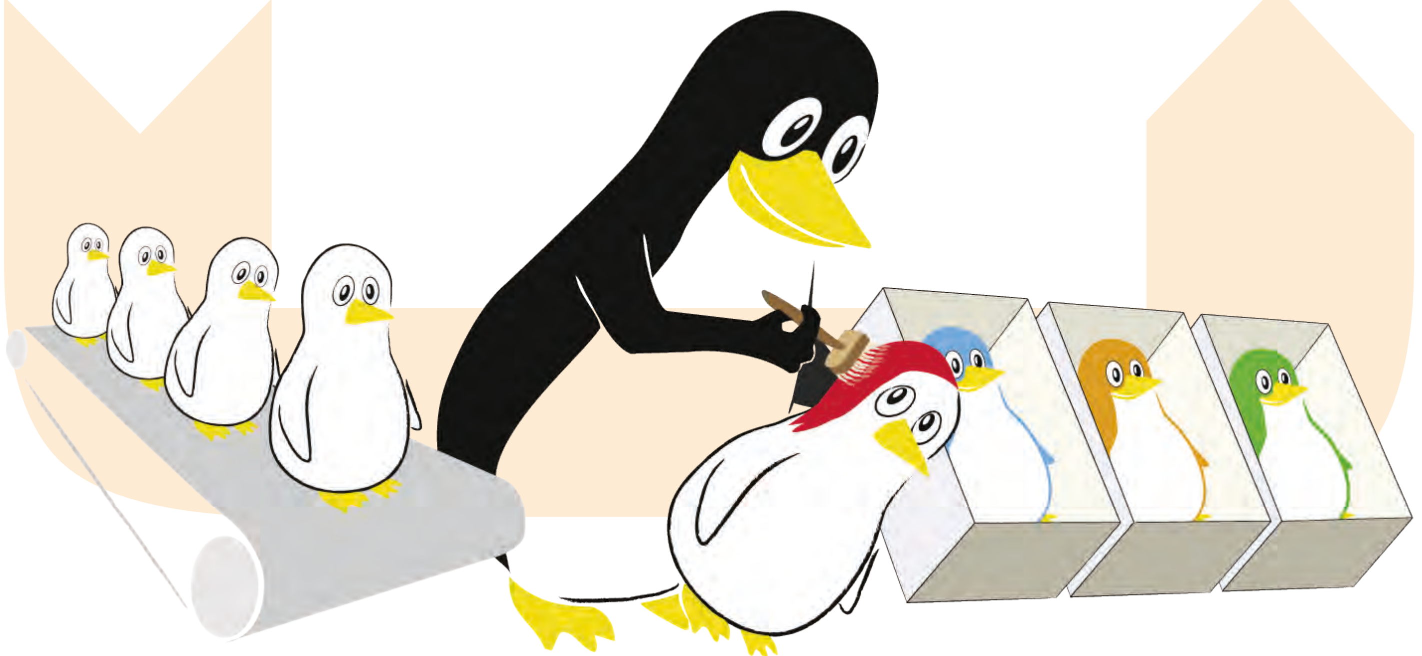
Task	Yocto	ELBE (based on Debian)
Compiling	Cross translation on the host Many Open Source packages can be cross-translated today without much effort regarding any dependencies. Possibly, dependencies to the host system can be built into the BSP (linking host libraries etc.).	Native or cross translation on the host Target system is created in a virtual machine either cross or native (in a Qemu environment); translating on the target is no longer necessary. The virtual machine decouples the target system 100% from the host and avoids any problems with native translation.
Additional packages or features (Enhancement of functionality)	Customize recipes to create new packages. Then install packages through package manager (Yocto supports them) or create a complete new BSP	Either by Package Manager on the target (installing the additional functions/packages) or by creating a new BSP or by creating an update image (and subsequently by updating on the target)
Individual configurations and customization of features	Each software package comes with its own recipe and/or Metalayer. This allows you to define for each package how it is built. All changes to configurations in Yocto can be tracked between two versions.	Using the original (binary) Debian packages, you can quickly create a first BSP. A specific customization of the packages is not possible. ELBE allows you to recompile each individual package with its own specific configuration (see above). The configuration is stored in XML, and changes are traceable. Each build remains reproducible.

Task	Yocto	ELBE (based on Debian)
Platform customization	Complete platform can be configured and customized due to recipes and meta-layers. Reuse over different hardware platforms possible (e.g. for x86 and ARM based hardware; here boot loader and kernel configuration must be adapted, with the other parts a new compilation is sufficient)	Bootloader and kernel must be adapted, RFS can be reused after recompilation thanks to multi-arch architecture.
Learning curve	Steep, because of Meta-Layer concept Recipes Cross compilation, Dependencies of the packages	Straight Debian – quite flat ELBE - demanding, but not steep XML description Cross or native compilation Build custom Debian packages Dependencies of the packages
Reproducibility	Quite powerful, especially if the sources are on a local mirror server Automatic build possible	Quite powerful, if the sources or binary packages are on a local mirror server. Automatic build possible with ELBE
Host Tool Dependency	Existing; use of containers reduces this dependency to the fact that a Linux Docker image must be executable	With ELBE, there is no dependency because of the virtual machine;
Scalability and Automation	Build and test can be automated, regardless of the development of recipes and meta-layers. Their development can be split among several persons (scalable).	Build and test can be automated; XML files can be worked on by different persons
License Issue	A summary of used licenses can be generated, also on SPDX basis	A summary of used licenses can be generated, also on SPDX basis
Testing options	No dependence on Yocto	No dependency on Debian and/or ELBE
Debugging	All Linux features are supported	All Linux features are supported
Boot Time Optimization	Possible, because each SW package can be individually configured and compiled; assembly of the target image can be defined and thus optimized	Possible, because each SW package can be individually configured and compiled; assembly of the target image can be defined and thus optimized
RootFile System (RFS) Size	Since the packages to be integrated can be defined individually, the RFS size can be determined („only what is really needed").	Since the packages to be integrated can be defined individually, the RFS size can be determined („only what is really needed").
Maintainability (Feature, Security)	(Almost) no support by Yocto Project except the ½ annual releases; to what extent the own BSP is affected or not has to be checked in each case. (Almost) no security support through patches Maintenance effort very high	Regular bug and feature fixes by Debian Project Regular security patches through Debian project Maintenance effort therefore at a minimum
SDK (Toolchain)	Will be built	Will be built

Advantages of ELBE

With the use of Debian, the industrial environment now can benefit from the many advantages in the server and desktop world. The ELBE build system helps the user to adapt Debian to his application and to keep the build process reproducible. With ELBE and Debian the advantages are:

- Use of a long-lived technology
- Maintained packages
- Flexibility through the availability of a large number of packages and through simple integration of new packages on the target system
- Easy creation of images for different storage media and boot scenarios
- Easy adaptability and extensibility
- Reproducibility of the build process
- Easily create an SDK
- Easy handling of all license obligations



Are you interested? Would you like to learn more about our products and solutions? Simply contact us via telephone or email.

WP_2019 V1.0

LINUTRONIX GMBH
Bahnhofstrasse 3 | D-88690 Uhldingen - Mühlhofen
Telefon +49 7556 25 999 0 | Fax +49 7556 25 999 99
sales@linutronix.de | www.linutronix.de

LINUTRONIX
LINUX FOR INDUSTRY