An Introduction to Yocto

Author(s):

Plextek Services Limited, London Road, Great Chesterford, Essex CB10 1NY, United Kingdom

Plextek

An Introduction to Yocto

Who needs Yocto and what does it do for you?
To begin by teaching granny to suck eggs, in order to understand Yocto you need to understand what a Linux distribution (commonly known as a 'distro') is.

The first thing to realise is that unlike MS Windows or MacOS there is no owner and no canonical "version" of Linux but rather a sea of different, often competing, components (primarily but not exclusively applications) that can be glued together to create it.

You can do this for yourself but very few people have the expertise, the time or the motivation to do it. As a result a number of organisations such as Ubuntu, Red Hat and Debian have emerged in order to provide pre-canned component collections, which are referred to as Linux distros.

The very first set of components required to build a Linux distro originated from the Free Software Foundation and is known as the GNU[1] Operating System. The one component that they were initially unable to develop for themselves was the operating system kernel, so they decided to adopt the Linux kernel which was developed and is still maintained by Linus Torvalds. The combination of GNU components and the Linux kernel is referred to as GNU/Linux – generally shortened to Linux (much to the chagrin of the Free Software Foundation).

However you will not find a pure "GNU/Linux" distro because in addition to GNU/Linux (and in some cases instead of it), many additional components from other sources are also included in every Linux distro in order to make it fit for purpose. For convenience, components with a common purpose (e.g. a set of networking utilities or a programming language toolset) are grouped together into packages that are installed and managed as a single entity. The package is therefore the basic unit of a distro.

Each distro defines a minimum set of packages that must be installed to make it run at all, along with a collection of other packages that may be installed/un-installed at the discretion of the user. Distro developers define a set of packages that are installed by default on a new system. After installation, the user is free to add further packages, or delete some of the pre-installed ones, in order to fine tune the system to their precise requirements.

If you want a Linux distro to run on standard PC architecture hardware, you have a huge number of choices including: Ubuntu, Red Hat, Debian, Mint, SuSE and many more besides. If you want a Linux distro to run on an embedded hardware platform that is not PC architecture your choices have historically been far more limited. These have generally tended to boil down to whatever the hardware manufacturers chose to offer on a 'take it or leave it' basis. Making changes to the Linux distro on an embedded system was thus difficult or impossible for the user and upgrades were provided as and when the manufacturers got around to it, if at all. This remains the case for many devices, particularly IoT devices such as access points, routers, TVs and set top boxes as well as mobile devices based on Android (which has a bare bones Linux distro carefully concealed under the hood).

A key part of the problem has been the sheer complexity of creating and maintaining embedded distros. While many organisations – including some very large businesses – exist to do this for PCs, they are only able to do so due to the relatively limited variety of processors and hardware architectures on which their software has to run. In the embedded world, processors abound, hardware platforms are legion and mutually incompatible kernel forks have proliferated, especially on ARM based hardware. This has historically made it impossible for "conventional" Linux distros to support them. Lately, a lot of effort has been put into eliminating

1. The name GNU is a recursive acronym that stands for "GNU's Not Unix". Who says software developers lack a sense of humour?

the multiple kernel forks and merging them back into the mainline Linux kernel. However, this is a very difficult task that has resulted in a lot of angry outbursts from Linus Torvalds and others and it remains a work in progress.

Another big problem for the embedded space is that application packages are written and maintained by many different groups around the world, working independently of (and in many cases in competition with) one another. These teams have all had a clear common interest in supporting Linux on x86 processors but, historically, other processors were much lower down on, or completely off, their radar. In recent years, as embedded Linux has become the OS of choice for various classes of embedded devices, this situation has improved and most packages can now be built for a variety of different processors, particularly ARM architectures which are now competing head-on with x86 in many different markets.

Another big problem in this area is how to keep track of all of the application packages as they are updated and ensure that they still build and run correctly on every platform to which they have been ported. Doing this properly is an enormous undertaking that strains the resources of even large organisations.

Numerous attempts have been made to address these problems by means of toolsets designed to build embedded distros starting from source code, 'Buildroot' being probably the best known example. These toolsets use cross-compilers running under Linux on PC build hosts to generate the embedded distro code.

Whilst one can debate the pros and cons of these toolsets (Google is your friend here), they have historically tended to suffer from a number of problems:

- The toolsets were difficult to configure and use and/or limited in their scope.
- Toolset developers typically select a particular version of each of the underlying packages and only occasionally update them. This means that security vulnerabilities and bugs may remain in the resulting distros for extended periods even though the package developers have already fixed them.
- The number of supported packages is often quite limited (e.g. a few hundred for Buildroot)
- Dependency management is often limited, so using the tools to build a functional distro requires quite a lot of knowledge and care on the part of the user.
- A user who wishes to customise packages in ways that were not anticipated by the toolset developers need to directly interact with the individual packages in order to modify them. This makes some packages considerably easier to adapt than others.
- Toolsets typically output complete system images but do not make the individual packages available separately. This means that software update requires the replacement of the entire system image. This can be tens of Megabytes, even if only one byte in one file has actually changed. This also means that post-installation changes to configuration files can easily be overwritten during the update process.
- Toolsets tend to have a single monolithic build configuration mechanism (e.g. menuconfig in Buildroot) which defines everything about the image, so re-use of common configurations in different projects is difficult.
- The toolsets rely on packages already installed on the build host (e.g. make, compilers, linkers etc.) and/or download pre-built cross-compilation tools from the Internet. This leads to a configuration control problem where each individual build host may be using different versions of tools. The result is that it can sometimes be difficult or even impossible to re-create the same target image on two different build hosts or on a single build host at different times, even if none of the source code from which the image is built has changed.
- For a long time there was no clear favourite build system and processor board manufacturers chose them seemingly at random. This created problems for OEMs wishing to use those boards in their products because, in the worst case, they would have to use a different toolset for every board.

The OpenEmbedded /Yocto Project addresses these issues:

- It is based on a custom-designed toolset which supports a sophisticated distro build mechanism based on layered scripts known as “recipes”. Each package has its own recipe or set of recipes. Users only need to learn how to write these recipes.
- Recipe maintenance is delegated to individuals associated with the teams that develop the corresponding packages. This means that Yocto support becomes simply another aspect of package maintenance for the corresponding team and all of the details of building an individual package are taken care of by someone who already understands the package.
- Each distro has at least one “image” recipe that defines which packages are included in the production image used during product manufacture along with a set of tools for creating deployable images in a variety of formats typically used by product manufacturing processes. It is also possible to extend the toolset to use other image generators if required.
- Everything is a package (including the Linux kernel) and all packages can be updated after initial deployment using package managers such as ‘RPM’ (which is of course itself just another updateable package).
- Yocto understands how to interact with version control systems such as Git and SVN so that it can fetch repositories from the package developers’ public servers and do fine grained version control as required.
- Yocto depends on a small number of tools that are already on the host build system just to get itself up and running and then compiles the other tools it needs from source code. This ensures consistent configuration control of the toolset used to build the packages as well as of the packages themselves. The build tools are themselves managed as packages so that they are re-built by Yocto in exactly the same way as the packages that are part of the distro. This means that their output should always be reproducible, regardless of the initial state of the host build system.
- Yocto has a very sophisticated dependency management/rebuild scheme that can generally detect automatically when anything changes and rebuild the affected packages.
- Support from major chip design companies such as Intel and ARM and board manufacturers such as Raspberry Pi means that Yocto is becoming the “obvious” toolset to choose. As a result, board manufacturers that do not support it are at increasing risk of being left behind.
- Individual system developers see Yocto as something that they should have on their CVs[2].
- There are now several thousand packages/recipes and many Board Support Packages (BSPs) ported to Yocto with more being added all the time.
- Yocto has licence and source code management mechanisms to help its users document package usage and avoid licence infringements.
- Yocto is designed to support the creation of small packages/images for resource limited hardware. In fact, the name ‘Yocto’ was chosen because it is the naming prefix for the smallest measurement scale (10^{-24}) in the SI system of units.

What is inside Yocto?

Yocto is essentially a set of tools running on a Linux build host that build embedded Linux distros, along with a set of recipes that tell the tools what to do. Given the right recipes, it is capable of building distros for: ARM, MIPS, PowerPC, x86 hardware and PC architectures. This covers the vast majority of processors that are capable of running Linux.

The key component is a tool called ‘bitbake’ (note the cookery theme). In essence, bitbake is a task scheduling tool that understands how to determine the order in which the build activities have to happen based on the dependencies within and between the individual packages.

2. Guilty as charged M’Lud

For instance, you have to build the cross compiler first and you have to build libraries before you build the applications that link to them. You can think of it as 'Make' on steroids (but don't strain the analogy because it is very different from, and much more capable, than Make).

Bitbake is a command line tool written in Python and is extensible by means of Python "classes" that give it an understanding of how to perform common build activities (e.g. downloading package source code from remote Git repositories).

The recipes for the individual packages are actually Linux shell scripts (sh rather than bash), using a large set of variables and functions provided by bitbake, such as its extension classes, user configuration files and the recipes themselves. They understand how to obtain, configure/adapt and build the source code for their packages.

Package developers (or Yocto support teams associated with them) provide the recipes for building their packages. Distro builders can append their own recipes that modify/extend or even replace the base package recipes in order to customise the package to suit their needs. To this end recipes are organised into prioritised layers, where higher priority layers (e.g. those written by Yocto users) are processed later than (and hence can modify) the lower priority layers provided by the package developers. By convention, layer names start with the string "meta-", so the layer for the Raspberry Pi is called meta-rasberrypi, the layer for web browsers is called meta-browser and so on.

A very large subset – although by no means all – of the packages developed for Linux on the PC have now been ported to Yocto and can be built for and deployed on any hardware platform that supports the necessary resources (because of course your hardware is going to need a display in order to use a video player package).

In addition to bitbake, Yocto supports a number of other useful scripts and tools. Of these, the most interesting is probably Toaster. This adds a web interface to Yocto, providing a means to configure and build distros and to visualise key information about them such as dependencies between packages.

What is Poky?

Poky is a workspace that contains both the basic OpenEmbedded toolset (bitbake, Toaster etc.), the core Linux reference layer (meta-poky) and a BSP layer (meta-yocto-bsp) which adds support for a number of common hardware platforms. It contains the recipes, configuration files etc. required to build a basic embedded Linux distro. For historical reasons, one further layer is required, which does not conform to the "meta-xxx" naming convention but is called openembedded-core. This contains core metadata that is common to all distros.

Other layers allow you to extend the distro by adding additional applications and board support packages for specific hardware platforms.

It is possible to create your own distro without using Poky as the starting point. However, a lot of wheels have to be re-invented in order to do this. There are a few layers such as meta-angstrom that do this for you. There are even layers that emulate some PC distros such as Debian (called meta-debian as you might expect); this supports the same system configuration and packages that you would get if you installed the corresponding Debian release (Jessie at the time of writing) on a PC.

How easy is Yocto to use?

It all depends. Poky provides a small number of pre-canned image recipes (which define the set of packages that go into the distro) such as core-image-minimal. Pointing bitbake to one of these will build a distro with no further work on the developer's part.

The first step beyond this is to write your own image recipe in order to define precisely the set of packages that you want in your distro. This step is fairly easy – you copy one of the pre-defined image recipes, give it a sensible name and edit it to change the laundry list of packages that it includes.

The next step is to add your own packages and/or to tweak the setups of existing packages (e.g. by adding platform specific configuration

files). This takes a bit more work because you have to understand how to write recipes. Yocto does provide a tool (devtool) for creating new recipes or modifying existing ones but it may not always do what you want and you still have to understand what lives inside a recipe.

By the time you get to this stage, I strongly recommend you get yourself some proper training or at least a mentor who knows Yocto. The Yocto documentation is very useful but, like most Open Source documentation, it is primarily for reference and good explanatory material is thin on the ground. Also, in the end, there's nothing quite like talking to somebody who already knows how to do it.

Conclusion

You might not think it from what I've just said, but in the end there's no right or wrong answer as to which toolset one should choose when building embedded Linux distros. Buildroot has its advantages over Yocto and other distro generator toolsets are available. Commercial distro builders such as Ubuntu are also now taking embedded devices seriously. If you want an easy life, and you are free to choose a hardware platform that supports them (such as the RaspberryPi or BeagleBone), you can simply take what they give you. However, for embedded developers who need fine grained control over what goes onto their platform, Yocto is the one to beat.

For myself, having used Yocto, Buildroot and commercial distros, the power and flexibility of Yocto gives it the edge when it comes to creating that finely honed product that will take the market by storm. I commend it to the house.