Articles » Internet of Things » Boards / Embedded devices » Raspberry Pi

Getting Mono Running on a Raspberry Pi Using Yocto

Richard Dunkley, 9 Nov 2014 CPOL
★★★★★ 5.00 (1 vote)

Demonstrates how to use the Yocto Project to get a custom Linux operating system with Mono up and running on the Raspberry Pi development board.

Introduction

This article is a tutorial that describes how to get a custom Linux baseline and Mono (a cross platform .NET runtime environment) up and running on a Raspberry Pi development board. It is part of a larger article that explains how to use the Yocto Project to establish a custom Linux baseline across multiple hardware configurations. The outcome of this tutorial is that our Raspberry Pi development board will be setup to run C# applications and libraries. These applications and libraries can then be developed using a high level Integrated Development Environment (IDE) such as Visual Studio, SharpDevelop, or Xamarin Studio. This allows software developers with limited experience with embedded environments to write software for an embedded Internet of Things (IoT) device.

Setup

The basic setup of the Raspberry Pi (power connections, jumper settings, serial output, etc.) is not covered in this article since it is assumed that an owner of a board wishing to use this tutorial would have a basic understanding of these hookups from the supplied quick start guides or reference manuals provided with the board. The focus of this article is using Yocto to build the desired environment targeting the development board.

The following software and hardware components are used in this tutorial:

- Linux development system of your choosing (tutorial uses Mint Linux 17 64bit)
- Poky Build System v1.6.2 (Daisy)
- meta-raspberrypi metadata layer (Daisy branch)
- meta-mono metadata layer (Daisy branch)
- Raspberry Pi Development Board

The main Yocto Project tool that we will be using is the Poky build system. In order to use this tool we need a Linux development environment. Most mainstream Linux distributions are supported. I used Mint 17

64bit for mine. The Yocto Project website also has a "Build Appliance" which contains a VMWare Virtual Image of a Linux system all setup to play with. If using Poky on your own Linux environment be sure to install the required dependencies (see the Quick Start or the Reference Manual on the Yocto Project website for more information www.yoctoproject.org). The packages I had to install were the following: g++, texinfo, libsdl1.2-dev, texi2html, cvs, subversion, bison, flex, and mono-complete (to build Mono).

I also recommend you read some of the Quick Start documentation on the Yocto Project website. I will outline the steps to create boot images and files for the Raspberry Pi in this article, but a more in depth understanding of the capabilities and features of Yocto can be obtained from Yocto's online documentation.

Using the code

Commands that should be run at the terminal on the Linux development system side are shown with the $ prompt:

```
$
```

Commands that should be run on the development board are shown with the # prompt:

```
#
```

File modifications will use "..." to represent the text in the file before and after the section referred to. This will allow us to show the sections in the files that need to be changed without displaying the entire file contents.

```
...

Text in the file

...
```

Tutorial

Create a folder for the Raspberry Pi poky build environment:

```
$ mkdir ~/rpi
```

Clone the git Yocto project core into the poky working directory:

```
$ git clone -b daisy git://git.yoctoproject.org/poky.git ~/rpi
```

We now need to pull in the board support package associated with our processor or board. The meta-raspberrypi board support package will work. This metadata layer is located on the Yocto project website. If you cannot locate a BSP for your target board using the Yocto website, try googling "yocto <board or processor> bsp" and you may find a community maintained one.

Clone the meta-raspberrypi board support package into the poky working directory. Note: when pulling in a board support package, be sure to read the README file associated with the package or the repo "About" section online. Sometimes they have additional steps that must be taken in order to use the BSP.

```
$ git clone -b daisy git://git.yoctoproject.org/meta-raspberrypi ~/rpi/meta-raspberrypi
```

The repo "About" section outlines that we also need the meta-multimedia layer from OpenEmbedded. This layer is contained in the meta-openembedded package. We can clone this package using the following command:

```
$ git clone -b daisy git://git.openembedded.org/meta-openembedded ~/rpi/meta-openembedded
```

Clone the meta-mono metadata layer into the poky working directory:

```
$ git clone -b daisy git://git.yoctoproject.org/meta-mono ~/rpi/meta-mono
```

Source the build environment script to create the configuration files:

```
$ cd ~/rpi
$ source oe-init-build-env
```

Add the meta-raspberrypi board support package layer and Mono metadata layer to the build environment. Do this by opening up the ~/rpi/build/conf/bblayers.conf file and adding the layers to the BBLAYERS variable after meta-yocto-bsp. The file should look something like the following when finished:

```
...
  /home/<username>/rpi/meta-yocto-bsp \
  /home/<username>/rpi/meta-raspberrypi \
  /home/<username>/rpi/meta-mono \
  "
...
```

Now change the MACHINE variable to the Raspberry Pi board. This tells the build environment which board to target when building the Linux environment. To do this edit the ~/rpi/build/conf/local.conf file. Scan down the file until you reach the following part:

```
...
#MACHINE ?= "edgerouter"
#
# This sets the default machine to be qemux86 if no other machine is selected:
MACHINE ??= "qemux86"
...
```

Add the Raspberry Pi reference as the MACHINE above the default so the file reads:

```
...
#MACHINE ?= "edgerouter"
MACHINE ?= "raspberrypi"
#
# This sets the default machine to be qemux86 if no other machine is selected:
MACHINE ??= "qemux86"
...
```

This sets the build environment to target the Raspberry Pi. If you don't know what the machine name is you can look under the BSP folder (meta-raspberrypi) subfolder: conf/machine (~/rpi/meta-raspberrypi /conf/machine). The names of the .conf files in this folder are all the names of the machines that can be targeted by the board support package. Also note that the default machine is qemux86. QEMU is an open source machine emulator. You can use this emulator to try out the custom built Linux distribution prior to actually targeting a specific board.

You may want to build a minimal image at this point to determine if your build environment is setup correctly. You can do this by running "bitbake core-image-minimal". We are going to skip this step to move onto building the minimal image with Mono included.

There are lots of ways to mix and match packages and recipes. The Mono metadata layer already provides us with some recipes to add Mono to any image we want to create; however, here we are going to create are own image recipe that includes the minimal Linux image with the Mono runtime included. This means the GUI elements of Mono (classes that rely on libgdiplus) are not going to work because the basic minimal image does not include GDI. Since a lot of IoT applications may not contain a graphical interface we decided this was the best approach to show the minimal requirements to run C# on a board.

The first thing we need to do is create an include file that tells the build environment what we need to include. You can do this by running the following:

```
$ (cat << EOF
> IMAGE_INSTALL += "mono mono-helloworld"
> EOF
> ) > ~/rpi/meta-mono/recipes-mono/images/rpi-hwup-image-mono.inc
```

This command generates the file in the meta-mono layer. You can also create your own layer for this. If you compare this include file with the ~/rpi/meta-mono/recipes-mono/images/core-image-mono.inc file you'll notice that the only difference is the libgdiplus dependency is left out. Also note that we named this include file differently than in the other development board tutorials. meta-raspberry provides its own basic boot image recipes. The rpi-hwup-image recipe uses the build environment's default core-image-minimal recipe but adds some kernel modules to it. We will use the rpi-hwup-image for our target recipe instead of the core-image-minimal in order to get these additional board specific modules.

Now let's create the actual bitbake recipe file. You can do this by running the following:

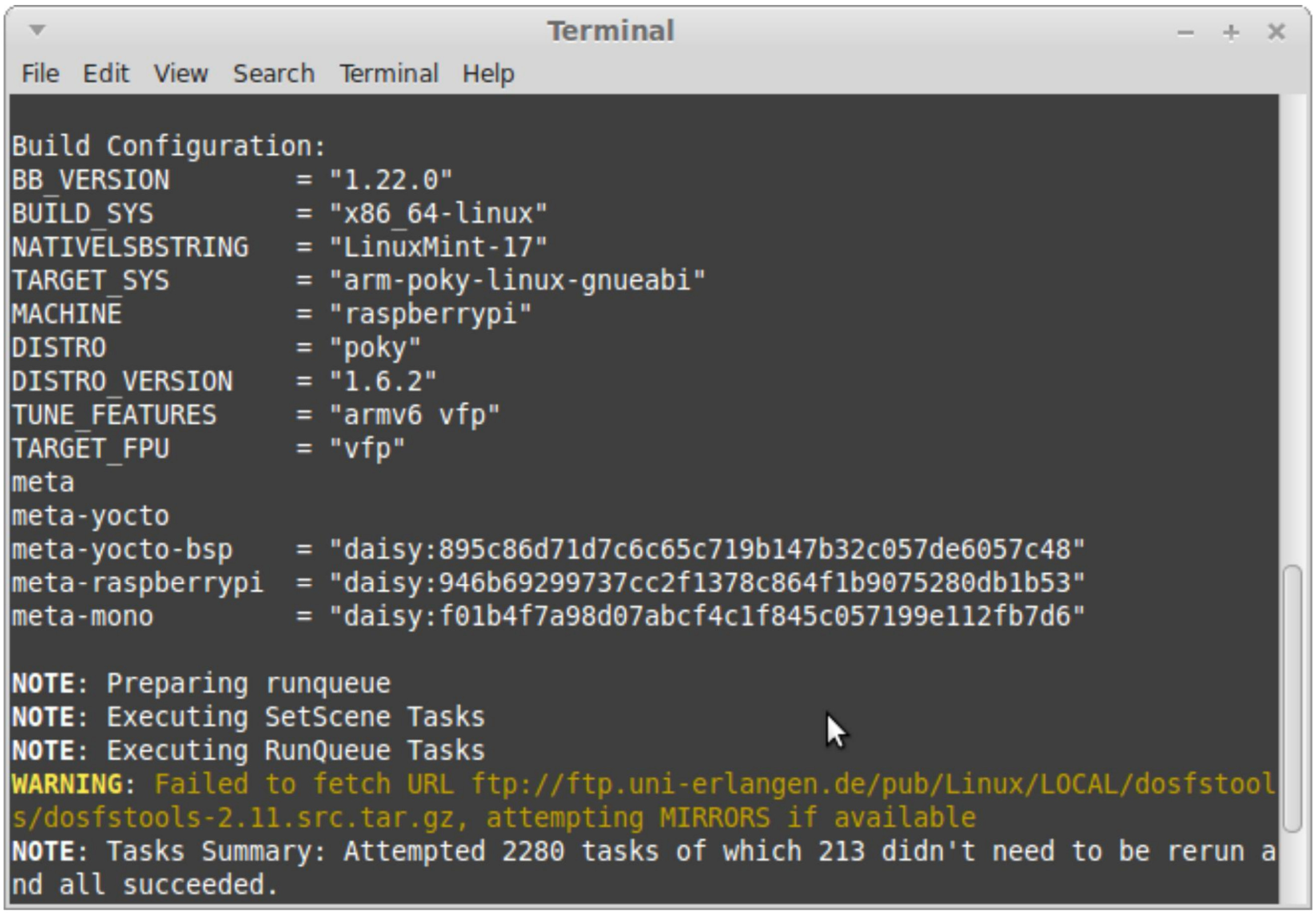
```
$ (cat << EOF
> require recipes-core/images/rpi-hwup-image.bb
> require rpi-hwup-image-mono.inc
> EOF
> ) > ~/rpi/meta-mono/recipes-mono/images/rpi-hwup-image-mono.bb
```

This command generates the bitbake recipe file. This recipe is fairly simple. The first line tells the build environment to include the recipe to make the rpi-hwup-image image. The second line tells the environment to add the Mono files to the image.

We now have our image recipe and are ready to bake. Type the following to build the image (make sure you are in the build directory):

```
$ bitbake rpi-hwup-image-mono
```

This process may take a very long time depending on your environment and internet connection. It will first parse all the recipes and determine what source code needs to be downloaded from the various repositories. Note, even though we only included two recipes in our recipe file these two recipes represent a hierarchy of a large amount of dependent recipes. The parsing traverses the dependencies and determines all the files that need to be downloaded. It downloads the source code and caches it locally. The advantage is that subsequent builds that may require the source do not need to go out to the internet to obtain them. The disadvantage of this is depending on the recipe you are trying to build you may have a large amount of files that get downloaded and stored. After this build process our ~/rpi folder grew to 21GB. This is representative of the amount of storage required for even a very minimal build.



```
Build Configuration:
BB_VERSION        = "1.22.0"
BUILD_SYS         = "x86_64-linux"
NATIVELSBSTRING   = "LinuxMint-17"
TARGET_SYS        = "arm-poky-linux-gnueabi"
MACHINE           = "raspberrypi"
DISTRO            = "poky"
DISTRO_VERSION    = "1.6.2"
TUNE_FEATURES     = "armv6 vfp"
TARGET_FPU        = "vfp"
meta
meta-yocto
meta-yocto-bsp    = "daisy:895c86d71d7c6c65c719b147b32c057de6057c48"
meta-raspberrypi  = "daisy:946b69299737cc2f1378c864f1b9075280db1b53"
meta-mono         = "daisy:f01b4f7a98d07abcf4c1f845c057199e112fb7d6"

NOTE: Preparing runqueue
NOTE: Executing SetScene Tasks
NOTE: Executing RunQueue Tasks
WARNING: Failed to fetch URL ftp://ftp.uni-erlangen.de/pub/Linux/LOCAL/dosfstool
s/dosfstools-2.11.src.tar.gz, attempting MIRRORS if available
NOTE: Tasks Summary: Attempted 2280 tasks of which 213 didn't need to be rerun a
nd all succeeded.
```

The image above gives an example of what should be displayed on the terminal while the custom Linux image is being created.

When the process completes successfully you can find the required boot files in the ~/rpi/build /tmp/deploy/images/raspberrypi directory. There are a lot more files in this folder than are needed to boot the board. The build process creates an SD card image file that can be written directly to an SD card. This makes it much easier to setup the bootable SD card than it is for other BSP. The SD card image file is rpi-hwup-image-mono-raspberrypi.rpi-sdimg.

Copy the image file to an SD card using the following commands. Be sure to replace the 'sdc' for the device your SD card is mounted to (may be sda or sdb, etc.). Be careful to get the right device (you don't want to wipe out the wrong drive).

```
$ umount /dev/sdc1
$ umount /dev/sdc2
$ sudo dd if=~/rpi/build/tmp/deploy/images/raspberrypi/rpi-hwup-image-mono-raspberrypi.rpi-sdimg
of=/dev/sdc bs=1M
```

We are now setup to boot from the SD card. Eject the card and plug it into the Raspberry Pi. Apply power and wait for the system to boot. If the build was a success, the SD card boots up to a login that looks similar to the following:

```
Poky (Yocto Project Reference Distro) 1.6.2 raspberrypi /dev/tty1

raspberrypi login:
```

Use "root" as the username to login and you should get access to the shell. Now type the following to determine if Mono was setup properly:

```
# mono --version
```

It should show something similar to below:

```
Mono JIT compiler version 3.4.0 (tarball Sat Nov  8 10:52:14 MST 2014)
Copyright (C) 2002-2014 Novell, Inc, Xamarin Inc and Contributors. www.mono-project.com
        TLS:           __thread
        SIGSEGV:       normal
        Notifications: epoll
        Architecture:  armel,vfp
        Disabled:      none
        Misc:          softdebug
        LLVM:          supported, not enabled.
        GC:            sgen
```

Since we included the mono-helloworld recipe in our image you should be able to run the following command:

```
# mono /usr/lib/helloworld/helloworld.exe
```

If "HelloWorld" printed out on the console then you just successfully ran a .NET application on your development board.

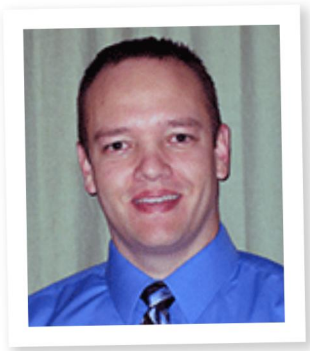
History

November 9th, 2014 - Initial Submission

License

Share

About the Author



Richard Dunkley
Engineer
United States

I earned my Bachelor of Science in Computer Engineering from Utah State University in 2004 with a minor in Mathematics and Computer Science. I also obtained a Masters of Science in Electrical Engineering from Utah State University with an emphasis in Embedded Systems. I am proficient in C#, C, assembly, and VHDL and have written a variety of applications targeting embedded processors. My experience includes robotics, LIDAR imaging, FPGAs, and various communication protocols. I currently work for the Air Force as the Technical Lead over a team of engineers at Hill Air Force Base, Utah.

Comments and Discussions

0 messages have been posted for this article Visit **http://www.codeproject.com/Articles/840489/Getting-Mono-Running-on-a-Raspberry-Pi-Using-Yocto** to post and view comments on this article, or click **here** to get a print view with messages.