Jukka Kokko

SOFTWARE BUILD AND RELEASE MANAGEMENT FOR A WIRELESS PRODUCT WITH OPEN SOURCE TOOLS

SOFTWARE BUILD AND RELEASE MANAGEMENT FOR A WIRELESS PRODUCT WITH OPEN SOURCE TOOLS

Jukka Kokko
Master's thesis
Autumn 2013
Degree Programme in Information Technology
Oulu University of Applied Sciences

ABSTRACT

Oulu University of Applied Sciences
Degree Programme in Information Technology

Author: Jukka Kokko
Title of thesis: Software Build and Release Management for a Wireless Product with Open Source Tools
Supervisor: Markku Rahikainen
Term and year when the thesis was submitted: Autumn 2013 Number of pages: 43

The object of this research was to test and select the best open source tools and processes for a project working with a complex wireless device, from a build and release management point of view. Some of the investigations were started on 2005; most of the work has been done during the year 2013 for Elektrobit Wireless Oy.

The work consisted of studying and testing out multiple different open source tools used on build and release management process. The studied tools were Git, SVN and Mercurial as software configuration management systems, Bitbake, Yocto, Apache ANT and Apache Maven as build frameworks, Gerrit and Review Board as code review systems and CruiseControl and Jenkins as build automation systems. Also hardware selection, test automation, error and feature management related tools, delivery methods and effects of the project's development model were considered.

The result was a bit mixed as the best combination of tools depends on the project, but for example a project that uses agile methods and does software for a wireless embedded Linux device should use Git as a software configuration management system, Yocto as a build framework, Gerrit as a code review system and Jenkins as a build automation system. This kind of selection was considered to be the best option even if most of the studied applications would have been adequate to use.

Keywords:

Build management, software configuration management, build framework, code review

CONTENTS

TERMS AND ABBREVIATIONS

Build Framework	A set of tools that handles compilation from downloading code to generating an end package for a large amount of code.
GSM	Global System for Mobile Communications. A standard set to describe protocols for digital cellular networks.
Handheld	A small wireless telephone using GSM network, also called as a cellular phone.
Base station	Part of a GSM network, communicates with handhelds.
Development effort	The estimated effort that the project will require.
PC	Personal Computer. A general-purpose computer that can be used to write programs.
CI	Continuous Integration. A development model where all changes are merged together several times a day.
RAM	Random Access Memory. Memory that can be both red and written.
Codebase	The source code for software system and/or application.
RISC	Reduced instruction set computing. A processor design strategy that relies on simple instruction sets.
ARM	An architecture that describes a family of RISC-based computer processors.
SAS	Serial Attached Small Computer System Interface. A point-to-point serial protocol for data transfer.
Rpm	Rotations per minute. The speed of a rotating disk.
PCIE	Peripheral Component Interconnect Express. A high-speed serial computer expansion bus standard.
Bit	A basic unit of information in computing.
Hertz	A basic unit of frequency.
SPARC	Scalable Processor Architecture. A RISC-based commercial processor family.
C#	An object-oriented programming language mostly used for Windows specific programs.
SCM	Software Configuration Management. A system that handles all changes done to the source code in a controlled way.
HTTP	Hypertext Transfer Protocol. An application protocol commonly used with Internet.
SSH	Secure Shell. A cryptographic network protocol, used to connect two locations securely over the existing connections.
FSFS	File System atop the File System. A file system with version control, the current default storage system for SVN.
FOSS	Free and Open Source Software. Software that can be classified as free and open source.
IP	Internet Protocol. A communications protocol used in the Internet.

LDAP	Lightweight Directory Access Protocol. An access protocol for accessing and maintaining distributed directory information services over an IP network.
OpenID	Open standard Identification system. An authorization system developed to work with co-operating web sites
REST API	Representational State Transfer Application Protocol Interface. An interface to programmatically communicate via HTTP.
SQL	Structured Query Language. A special language for managing relational database's data.
NIS	Network Information Service. A server based authentication system.
TTCN	Testing and Test Control Notation. A programming language used for testing communication protocols.

PREFACE

This thesis work was carried out as part of developing a complex wireless device as a build specialist at Elektrobit Wireless Oy. Some parts of the work were also carried out after office hours. The thesis documentation was created after office hours during the year 2013.

I would like to thank Timo Torvikoski for enabling the thesis work on behalf of Elektrobit Wireless Oy, Markku Rahikainen for acting as a supervising teacher and Kaija Posio for acting as a language instructor. I would also like to thank my colleagues for the information given during thesis work and especially Vesa Vuohtoniemi for acting as a thesis supervisor.

Oulu, Finland, October 2013
Jukka Kokko

1 INTRODUCTION

The aim of the thesis work was to present tools and processes used on software build and release management for a project working with a wireless product (handheld, base station, etc.) by using open source tools and selecting the best combination of them. The client currently has multiple different setups, and the aim of the thesis work was to investigate and possibly improve the client's current build and release practices.

The research question for this thesis work was which combination of the available open source tools and processes would the best for build and release management on a project working with a complex wireless device? The outcome of this research was fit for few of the current projects, but some projects with different specification might be better off with a different combination. Also, the client's clients and collaborators might dictate what tools to use at certain phases of the development.

2 BUILD AND RELEASE MANAGEMENT

Build and release management is part of a software development effort. Build management includes compiling and packaging the software (14, p. 2); release management includes creating and delivering software releases. These two terms are often described separately, but in practice the same people handle both roles in parallel.

Build management is responsible for setting up, maintaining and developing the build framework. The build framework usually includes used tools for version control, code compilation and packaging but may also include the used hardware setup. Build management may also be responsible for setting up the code branches of version control when needed, and also maintaining the sidetracks of the development. Depending on the build manager's proficiency, usually both build framework and software debugging responsibilities are included in this role.

Release management is responsible for creating a distributable package and its release notes, and delivering them as a software release. The used tools on this stage are usually determined by the release recipient. The distributable package usually consists of the end product binaries that often require a special signing process, but may also include some debugging related files and the source code. The release note describes what is in the delivery, how to use it, what to take into account, etc. The release manager also needs to communicate with the project client along with the project management in order to know what to deliver, when to deliver and why something is not working or is behind the schedule.

3 HARDWARE SELECTION

3.1 Compilation Hardware Selection

When a build management related computing work is done - code compilation, target creation, etc. - it takes up a lot of resources. Therefore, the used PC hardware should be selected to be as powerful as the budget constraints allow. Also, the multiple parallel compilation requirements need to be taken into account, especially when the selected implementation process is Continuous Integration or similar.

A complex wireless device codebase size may differ from 0.1 to 30 gigabytes. This and the build time requirements effect on how much resources per build is needed. Also, the used operating system and used compilation tools have an effect. Table 1 shows some results of empirical testing; for example if the codebase size is 5 gigabytes, the target platform is ARM based, the computer operating system is Linux and build time requirement is one hour, the hardware resource requirement is about 8x3.4GHz cores, 8GB of RAM and 100GB of disk space. The exact requirement differs from project to project; for example if the project uses a lot of pre-compiled binaries, then the hardware resources can be a lot less. Also, some projects require a lot of disk space activity; in such case even the 15 000rpm SAS disks might be a bottleneck and faster options should be evaluated, such as PCIE RAM disks.

TABLE 1. Estimated codebase size effect on hardware requirements; build time 1 hour, ARM, Linux.

	5GB codebase	15GB codebase	30GB codebase
Processor	8 Core 3.4GHz	32 Core 3.4GHz	96 Core 3.4GHz
RAM	8GB	24GB	64GB
Disk Space	100GB	200GB	350GB

With modern build tools several builds can be done simultaneously on the same computer without a significant build time penalty. For example a project with a 5GB codebase can run 2 builds at the same time with the 5GB codebase setup from Table 1, and 6 builds at the same time with the 15GB codebase setup from Table 1.

Also the computer's operating system has an effect on the selected hardware. For example, if Solaris is selected to be the operating system due to availability or due to a large amount of processor cores needed, then SPARC architecture should be considered as an option.

The operating system selection is usually dictated by user preferences but the operating system has also an effect on the used tools; for example most ARM compilers work directly on Linux and via an emulator such as Cygwin on Windows. This has some effect on the build time. Also, some programming languages, such as C#, have good development and compilation tools only for a certain operating system. Some of the code that is included in the project may not be compatible with the 64-bit architecture, therefore also multiple different operating system versions may be needed.

When requirements and development branches change during the product development effort, more compilation capable hardware is usually needed. If the codebase size and the build time requirements allow, a bunch of virtual servers hosted on a rack servers (or in cloud) eases and speeds up adding and removing hardware resources to the project.

The recommendations for the build hardware for ARM based complex wireless device project are:

- Use any powerful enough hardware that you have lying around.
- Use virtual servers to balance load if possible.
- If there is enough proficiency in the project team, select Linux as the operating system.

3.2 Support System Hardware Selection

Build and release management usually also requires a separate hardware for certain supporting systems such as build automation, version control, error management or code review systems. Hardware requirements for these differ depending on the selected application for each use. Also, the operating system selection should be done according to each system's requirements.
For example, build automation can be done on a compilation system hardware, but if multiple compilations are needed, then separate build automation hardware is also needed. This hardware can be a virtual machine but it requires relatively more memory and disk space.

As software configuration management systems are based on databases, therefore they require a lot of memory and disk space but usually only an average processing power. However, the amount of users served simultaneously might require also a lot of processing, therefore on large projects the SCM servers should be dedicated servers.

Error management systems are also based on databases thus they have similar hardware requirements as version control systems. The amount of data is usually a bit less than on SCM system but the usage level might be a lot higher, therefore a dedicated server is also recommended. Error management is usually a separate team so this hardware requirement might come from them, but the build and release team works closely with them.

The hardware requirements of a Code review system vary a lot depending on the used applications. If the used application is light, a simple virtual machine with few processor cores and few megabytes of memory will suffice. If the code review application is complex and heavily automated such as Gerrit, it should be installed on a dedicated server.

4 CONFIGURATION MANAGEMENT TOOL SELECTION

4.1 Introduction and Requirements

SCM or software configuration management is a system that handles all changes done to the source code in a controlled way (25, p. 1). An SCM system can also be used for collaborating development from multiple sites. A software project can be done without a software configuration management but it will benefit even a one-man project. Any software project that has a number of developers definitely needs some kind of version control and change distribution system, so software configuration management is a must.

Wireless device software projects usually are large or very large-scale projects employing hundreds or even thousands of people all over the world. Also, the used software development model is usually waterfall, iterative, agile or any mix of these. These attributes mandate some requirements for the used SCM system. It needs to be scalable enough, fairly easy to use and to support a replication between different local systems. It also needs to be reliable and traceable and communicate with other systems. One of the SCM system's most important features in development effort is branching; in almost any project at some point the development work needs to be continued towards future versions while finalizing one delivery. Also, as all open source projects, it needs an active community to support in problem cases and an adequate license to be used in a commercial software development.

The needed scalability of the SCM system depends on the size of the codebase on the project. Most modern SCM systems can handle fairly large projects with standard server hardware; only the largest projects may need to rely on large solutions containing dedicated servers and specialized software such as IBM Rational Clearcase (18). One way to ensure that SCM system's scalability is sufficient for the project is to divide the project into subprojects that will have their own SCM instances.

SCM needs to be fairly easy to use especially for the parts that an average developer is doing; getting other people's changes and implementing their own changes. This is because an SCM system should help producing the working software, not making developers' lives harder. Some parts can be harder to do but it still needs to be possible; few people can obtain a SCM specialist role.

Replication support between separate systems may be needed if the project teams are located globally and communication between the developers and a single server is not fast enough for daily work. The project setup dictates if the replication support needs to be real-time (or as close to it as possible) but small delays in updates are often tolerated.

Reliability and traceability requirement means that all changes that SCM handles needs to be kept intact, including the information who has done and what. Any unexpectedly lost change might cause critical problems to the whole project, not to mention the developers loosing trust to the system. Some SCM systems allow deleting or modifying existing changes. This feature should be used only when it is absolutely necessary as the result may be similar to an unexpectedly lost change.

Communication with other systems is important as the developers benefit from a single tool usage. For example Eclipse (7) can be configured to use SCM systems instead of only a local work area. Also, build automation requires an SCM system that can communicate both ways. For example a submitted change in SCM may be used to automatically start a build.

4.2 Git

Git is a distributed SCM system and one of the most popular open source version control systems, designed for speed, robustness and a large amount of data. It is fairly complex and very flexible but normal development work is easy to do. One of the major differences to many other SCM systems is that each checkout from master codebase is its own repository. This way each user has their own copy of the codebase. Getting the changes back to the master codebase requires taking other people's changes into account, and the development work can be done even without a continuous Internet connection. A local copy of the codebase also makes it easier to implement different code branches at will as well as merging branches back together (9).

Git scalability is quite good. It cannot use multiple hardware servers per project which limits the codebase size per project, but it can handle multiple different projects at once. The codebase size limit is quite large - for example the linux-kernel project uses a single Git repository - thus designing appropriate size subprojects should not be too hard. Handling different projects as one is usually done with the help of a separate program such as the Repo tool by Google (19).

Ease of use was not the focus when designing Git, thus mastering it may take months to study. However, enough knowledge of the Git usage in a developer role can be obtained in an hour or so by training, especially if the developer is familiar with some other SCM system. All that is needed is a basic knowledge on how the Git works and on 7 commands presented in Figure 1. Git is a command line application but it also has a number of user-friendly interface applications to help developers using it - for both Linux and Windows.

```
git clone git://git.openembedded.org/oecore
git pull
… do changes …
Git add –u
Git commit –m”test changes”
Git fetch –all
Git rebase origin/master
Git push origin HEAD:refs/heads/sandbox/mynewbranch
```

FIGURE 1. Git usage example.

Replication or synching between different locations is possible with Git using only a few commands. Basically it requires mirroring a local copy of one of the remote locations, fetching another locations' mirror metadata to same copy and pushing mirrored changes back to all projects that need to be synced. In most cases project requires a short custom script that is run either periodically or when a change is detected on any of the locations.

Reliability and traceability are quite good with Git, but it does have few possibilities to change one's submitted changes and forcing software branches to different positions, so that some submitted changes are deleted. Both of these require using a specific flag on correct command and the effects are noted on log, so this issue can be tackled with training and communication. Also big binary files can cause problems. Changes lost for some other reason were not seen during the project.

Git has a good support with multiple different tools, and the existence of a command line interface and a native support for protocols such as HTTP and SSH enables an easy integration with almost any related software. For example a code review, build automation and test automation can be triggered with a submitted change into the Git repository. Many widely used open source tools such as Jenkins and Eclipse support Git directly.

Codebase branching is extremely easy with Git. Any change can be submitted to its own branch at any point of development, and merging the branches back together is also quite easy. Only the different binary files and code changes in same lines need human interaction with default settings, and the merging behavior is easy to change to favor either one of the branches being merged. Example of branching and merging is shown in Figure 2, and Figure 3 shows the same thing graphically.

```
git checkout –b testbranch HEAD^^
… do changes …
git add –u
git commit –m”test”
git fetch –all
git merge origin/master-next
```

FIGURE 2. Git branching and merging.

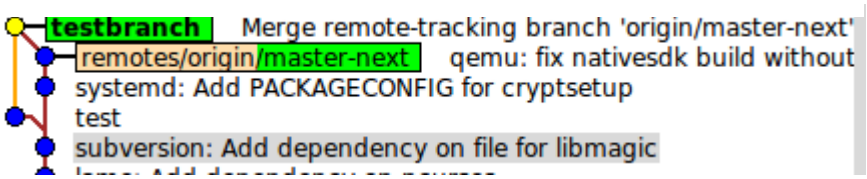

FIGURE 3. Git branching and merging graphically.

Git has an active community and its original developer, Linux Torvalds, continues to support it. The community makes not only error corrections but also implements new features such as better subproject handling. Also, the licensing, GNU General Public License version 2, allows almost any kind of project to use Git.

4.3 SVN

SVN or Apache Subversion is a centralized client/server SCM system designed to be reliable and simple to use (22). SVN has been in development since 2000, and it currently does yearly releases containing some new features and a bunch of error fixes thus it is in quite mature stage.

SVN scalability is adequate in many cases. SVN itself does not support scalability as it does not have the concept of a project; instead SVN uses a set storage subsystem for data; current default is FSFS that keeps the data in the operating system's file system (22). Scalability can be achieved by using operating system's scalability or by using webserver's load balance.

SVN has been designed to be simple and easy to use. All client sessions contain links to a server content, and if a modified file type is detected to be unmergeable, it is locked on the server preventing other people to do modifications while it is being checked out by someone. Most files can be modified simultaneously and merged automatically. For most people SVN usage consists of checking out files, doing modifications and checking them in again as demonstrated in Figure 4. Also branching is made easy. SVN also has multiple user-friendly interface applications available, some of them even integrate to the operating system's file manager interface.

```
svn co https://svn.myproject.com/repos/test/trunk/ test-trunk
cd test-trunk
echo "test" > test.txt
svn add test.txt
svn commit --username your-name --password your-password \
  --message "Trying out svn"
```

FIGURE 4. SVN command line usage example.

The replication of a SVN repository can be done in few ways; the simplest way but not the recommended one being the operating system's copy command thanks to the SVN's file system usage. The recommended way is to use an svnsync application that will create a slave repository from the master. If multiple master repositories or continuous synchronization is needed, then 3rd party applications need to be used.

SVN is reliable in most cases and has a good traceability. Most fatal problems usually relate to the used repository storage system; for example Berkeley DB can be sensitive to interruptions (22). Problems were not seen during the time of usage.

SVN is widely supported by multiple different tools. The existence of a command line interface and the native support for protocols such as HTTP enables easy integration with almost any related software. For example a code review, build automation and test automation can be triggered with a checked in change into the SVN repository. Many widely used open source tools such as Jenkins and Eclipse support SVN directly.

Branching on SVN requires copying one branch under a different directory as shown in Figure 5. This setup requires a quite good understanding what changes will be done in both branches during the separation, as merging the branches back together may be quite hard. Syncing other branches' changes to the development branch from time to time helps the issue.

```
svn copy trunk branches/test-branch
svn commit –m"Creating new branch"
[ save the commit id, say r123]
.. modify test-branch …
svn up
svn merge ^/trunk –r123:HEAD
```

FIGURE 5. SVN branching and merging.

SVN has a relatively active community that also meets face-to-face yearly, usually during the Hackathon event. The community does not only error corrections but also implements new features such as merging improvements. Also, the licensing, Apache License, allows almost any kind of project to use SVN.

4.4 Mercurial

Mercurial is a distributed SCM system designed to be easy to use and robust, have a high performance and be able to handle large amounts of data (16). It has been developed since 2005, it is quite mature and has a large amount of projects using it, such as Mozilla. Mercurial is quite easy to use, especially for users that have been using SVN, but not as flexible as Git. Mercurial is mostly done with Python so it is quite easily portable between operating systems. On Windows machine the Mercurial is one of the fastest SCM systems, and on Linux there is only a small difference between the Git and Mercurial performance.

Mercurial scalability is quite good. It cannot use multiple hardware servers per project which limits the codebase size per project, but it can handle multiple different projects at once. The codebase size limit is quite large: All files, metadata etc. should fit in RAM for efficiency thus designing appropriate size subprojects should not be too hard.

Ease of use was one of the main focus areas when developing Mercurial. Few hours of training is enough to work fluently with it, especially if the developer is familiar with some other SCM system. All that is needed is a basic knowledge on how the Mercurial works and on 4 commands presented in Figure 6. Mercurial is a command line application but it also has a number of user-friendly interface applications to help developers using it - for both Linux and Windows.

```
hg clone https://svn.myproject.com/repos/test/trunk/ test-trunk
cd test-trunk
echo "test" > test.txt
hg commit
```

FIGURE 6. Mercurial usage example.

The replication or synching between different locations is possible with Mercurial by using only a few commands. Basically it requires a local copy of one of the locations and pushing mirrored changes back to all projects that need to be synced. In most cases a project requires a short custom script that is run either periodically or when a change is detected at any of the locations.

Reliability and traceability are good with Mercurial, but it has a possibility to change once submitted changes. This requires using a specific flag in a correct command and the effects are noted on log so this issue can be tackled with training and communication. The changes lost for some other reason were not seen during the usage.

Mercurial has a good support with multiple different tools, and the existence of a command line interface and support for protocols such as HTTP enables easy integration with almost any related software. For example build automation can be triggered with a submitted change into the Mercurial repository. Many widely used open source tools such as Jenkins and Eclipse support Mercurial.

Branching is supported on Mercurial but it might require creating a new repository. This may be an issue in some cases as merging is possible only within a single repository. Handling different projects as one is not recommended but possible with a Subrepository feature, which would also allow other SCM systems' projects to be used. Branching and merging is demonstrated in Figure 7.

```
hg branch testbranch
hg ci -m "start testbranch"
echo "test2" >> test.txt
hg commit
cd ../test-2
hg pull ../test-trunk
hg merge
hg commit
```

FIGURE 7. Mercurial branching and merging.

Mercurial has an active community. The community does not only error corrections but also implements new features, and new releases come about every month. Also, the licensing, GNU General Public License version 2+, allows almost any kind of project to use Mercurial.

5 BUILD FRAMEWORK SELECTION

5.1 Introduction and Requirements

Build framework is responsible for compiling the software (14, p.3). The build framework should also produce needed target packages. Other requirements for the framework include configurability, speed, robustness, repeatability and communication with other systems. Also, as all open source projects, it needs an active community to support in problem cases and an adequate license to be used in commercial software development. If the software effort uses open source software components, the build framework also needs to be able to support a FOSS process - at least by collecting the license info from the open source modules.

Compilation requirements depend on the software, but in projects for wireless devices the software environment typically includes multiple different programming languages; for example Android software may contain both Java and C code (1). The build framework needs to be flexible enough to support different programming languages in different software modules.

Target packages are generally images when working for a wireless device but they might also be update packages or installable files. Depending on the project, target packages might consist of one or many different configurations. If there are multiple target packages, a good build framework should be able to provide all needed configurations, either by a build time variation or by a package generation time variation. This also requires a source code support for multiple different configurations.

Build framework configurability is essential in complex software projects as different modules of code might vary a lot at different development stages. A good framework can be configured relatively easily according to different requirements, for both a source code compilation and a target package generation. For example, adding a temporary compilation flag to one software module should be a simple and fast task to do. Also adding or removing software modules should not be a hard task.

The speed of the framework is essential in many different ways. If the target generation is fast, the developers have more time to focus on the development instead of waiting for a testable package to generate. Also, the compilation time both per a software module and per a whole codebase helps focusing to the development as target packages for larger scale tests are available sooner and test results might be available during the same day instead of days later. The most commonly used feature to speed up build framework is using concurrency, but also the build framework itself needs to be lightweight enough not to take up too much processing time.

The robustness of the build framework is an important feature. The larger the codebase grows, the longer it will take to compile and the longer the time penalty is when the compilation fails due to build a framework issue. Even with fast computers and a fast build framework the wireless device projects' build time is typically from one to many hours, so any random error might cause even a day's delay in development or in client delivery. Many build framework speedup methods such as parallelization or skipping some sanity tests generally decrease robustness.

Repeatability is closely related to robustness and means that any build done can be repeated with the same results (not including timestamps, etc.). For example build framework changes, knowledge transfer situations or some legal issues might require creating the build again.

Communication with other systems is important as the build framework is usually part of a larger process. Build framework usually gets its source code from the configuration management, and the resulting target package needs to be published to test automation, directory share, etc. Also, the initiation of the build might come from an automation system which needs an input from the build framework also during the build.

5.2 Bitbake

Bitbake is a build framework done with Python and designed for an OpenEmbedded project and is currently maintained by a Yocto project, but it can be used in any other software project, too (3). Bitbake can use Git as a default source file location but can also use any SCM system that has a command-line interface, or even packed files on a hard drive. It also describes the dependencies between software modules thus it is possible to compile any software module with its specific dependencies. Bitbake also includes an emulator for testing purposes.

Target package generation is flexible during build time - each recipe's installation part dictates what to take to the target package - but after the build, selecting only certain files to the target package is not possible. This means that target package variation can only be done by separate builds or external tools.

Thanks to its recipes, Bitbake is well configurable. Minimal recipe is presented in Figure 8. Another way to configure software is Bitbake configuration files that can define functions, variables, etc. common for the build environment. Bitbake has a lot of ready-made functions and more can be defined. Also, the component specific recipes can temporarily overwrite any functions affecting that module's handling. Configuring included packages can be done for example by defining a main recipe and setting all wanted packages as its dependencies.

```
DESCRIPTION = "test recipe"
 HOMEPAGE = ""
 LICENSE = "GPLv2"
 DEPENDS = "anotherTest"
BRANCH="master"
SRCREV="HEAD"
SRC_URI = "git://git@myurl.com/projects/test.git;protocol=ssh;branch=${BRANCH}"

 S = "${WORKDIR}/test-v${PV}"
 do_configure () {
   ./configure --prefix=${prefix} --without-snapshot
 }
 do_compile () {
   make
 }
 do_install () {
   DESTDIR=${D} oe_runmake install
}
```

FIGURE 8. Minimal Bitbake recipe.

Speed of the Bitbake framework is good; parsing the recipes takes a while - about 2 minutes with 1600 recipes - but after that the software modules can be built in parallel. Another speedup option is to compile only the changed and depended modules, but Bitbake relies on the recipe version number with this, therefore all changes should increment also the related recipe's version. This feature is called a shared state cache. While building, Bitbake uses about 950 MB of memory for a 120 MB codebase when compiling with 4 cores. The processor load from the framework is minimal.

Both robustness and repeatability are at a good level. During the testing the amount of randomly failed builds was under 1 ‰ of total builds, and all the failed cases were due to undocumented software dependencies that came visible with enough parallelization. All the builds repeated produced an expected result, except the ones that failed because of a software dependency error.

Most communications with other systems require a command-line interface or a special function for Bitbake recipes. Bitbake comes with an integrated Git support but it is restricted to downloading the source code. Bitbake itself is a command line tool which enables it to communicate with related external tools such as a code review and a build automation.

Bitbake does not have an active community as such but it has some support via Yocto and OpenEmbedded projects. Also, the licensing, GNU General Public License (GPL) and MIT/X Consortium License, allows any kind of project to use Bitbake. Alternatively, Bitbake has a built-in support for collecting software components' license files and making a summary of them.

Bitbake has some problems, and most of them are caused by users. Recipes' flexibility may cause problems when working with a large project as it enables abusing many kinds of hacks, patches, etc. Bitbake's features include selecting a certain - or the newest - version of each software module automatically, which can sometimes cause issues. Also, the dependency handling relies on recipe versions; if a module is updated but its version is not and the module is not cleaned before a dependent module compilation, the changed module will not be compiled.

5.3 Yocto

Yocto provides an infrastructure and tools for Linux distributions for any hardware architecture, but it can also be used in other software projects, too (27). Yocto can use Git as a default source file location but it can also use any SCM system that has a command-line interface, or even packed files on a hard drive. It also describes the dependencies between software modules therefore it is possible to compile any software module with its specific dependencies.

Yocto uses Poky as a development and build system. Poky uses Bitbake as the build framework, thus most of its results are similar to Bitbake. Yocto has one major difference compared to Bitbake alone: Whereas Bitbake is a build framework, Yocto provides a full system, including a basic functionality for an end product, for developing a Linux based system. Also, the documentation has been vastly improved.

Target package generation is flexible during the build time - each recipe's installation part dictates what to take to the target package - but after the build, selecting only certain files to the target package is not possible. This means that target package variation can only be done by separate builds or external tools.

Yocto uses recipes due to its Bitbake usage and is therefore well configurable. A minimal recipe is presented in Figure 8. Yocto project has a lot of ready-made functions and more can be defined. Also, the component specific recipes can temporarily overwrite any functions affecting that module's handling. Configuring included packages can be done for example by defining a main recipe and setting all wanted packages as its dependencies.

The speed of the framework is good; parsing the recipes takes a while - about 2 minutes with 1600 recipes - but after that the software modules can be built parallel. Another speedup option is to compile only the changed and depended modules with a shared state cache, but Yocto, as well as Bitbake, relies on the recipe version number in this, thus all changes also should increment the related recipe's version. While building, Bitbake uses about 950 MB of memory for a 120 MB codebase when compiling with 4 cores. The processor load from the framework is minimal.

Both robustness and repeatability are at a good level, and no build errors were seen during the testing. All builds repeated produced an expected result.

Most communications with other systems require a command-line interface or a special function for the recipes. Yocto comes with an integrated Git support but it is restricted to downloading a source code. Yocto itself is a collection of command line tools and libraries which enables it to communicate with related external tools such as a code review and a build automation.

Yocto has an active community and it is working closely with OpenEmbedded and Poky projects. Also the licensing, which depends on which tools from the infrastructure are used, allows any kind of project to use Yocto. Alternatively, Yocto has a built-in support for collecting software components' license files via Bitbake and making a summary of them.

5.4 Apache ANT

Apache ANT is a very flexible build automation system written with Java and based on xml files. It is originally designed for Apache Tomcat project, but it has been a standalone build automation tool since 2000 (2).

Apache ANT is providing a framework based on xml files that control each software modules' compilation and installation. Also JUnit testing (11) is easy therefore ANT is a good choice for a test-driven development. ANT supports any compilation tools, but unfortunately it does not have a dependency handling therefore it is not good for large projects as such unless the compilation tools can handle the dependencies. However, ANT has several extensions that support dependency handling, such as Apache Ivy.

Target package generation is flexible during the build time as a user can set on xml files what to take to the target package. After the build, selecting only certain files to the target package is also possible, but it requires well designed structures.

ANT is quite well configurable as it is based on xml files. It does not have any coding conventions or restrictions to the project layout; however, the xml specification needs to be followed. The user can also define extension modules called antlibs on xml files. An example of the ANT's build.xml is presented in Figure 9.

```
<?xml version="1.0"?>
<project name="Example" default="all">
   <path id="classpath.test">
     <pathelement location="/Projects/Java/Lib/junit.jar" />
   </path>
   <target name="compile" description="compile the example to class files">
      <mkdir dir="classes"/>
      <javac srcdir="." destdir="classes"/>
   </target>
<target name="test" depends="compile ">
   <junit>
     <classpath refid="classpath.test" />
     <test name="Test" />
   </junit>
  </target>
  <target name="all" depends="test" />
</project>
```

FIGURE 9. Apache ANT example build.xml.

The speed of the ANT framework is relatively good; parsing the xml files will not take long but parallel compilation requires separate code modules due to lack of proper dependency handling. While building, Java uses quite a lot of memory, and in large projects the Java virtual machine may need more memory allocation than it has by default. The processor load from framework is minimal.

Both robustness and repeatability are at a good level. During the testing no build errors due to the build framework were seen - except some misconfigurations by users. Also, all the reproduced builds produced similar results.

Most communications with other systems require a command-line interface. ANT itself is a command line tool which enables it to communicate with related external tools such as a code review and a build automation. Many development tools such as Eclipse and Jenkins support ANT directly.

Apache ANT has an active community. The community does not only error corrections but also implements new features, and new releases come few times a year. Also, the licensing, Apache Software License, allows any kind of project to use ANT. Collecting software components' license info is not directly supported by ANT but there are few Antlibs that can collect them, at least for Java projects.

5.5 Apache Maven

Apache Maven is a derivative of Apache ANT, and introduces some coding conventions, a file structure dependency and new features such as a dependency handling and an experimental parallel build support (15). It also provides an ability to download dependent modules and plugins over the network. Maven is one of the most used build frameworks for Java projects but it can also support other compilers.

Apache Maven is providing a framework based on xml files that control each software modules' compilation and installation. Modules are separated by a directory structure by default, and modules can include other modules. Also, JUnit testing is as easy as with ANT. Maven supports any compilation tool and has a quite good dependency handling thus it is sufficient for large projects. Maven also supports plugins.

Target package generation is flexible during the build time; a user can set on xml files what to take to the target package. After the build, selecting only certain files to the target package is also possible, but it requires well designed structures.

Maven is quite well configurable as it is based on xml files. It has stricter coding conventions and restrictions to project layout than ANT, but most of them are making configurations easier to use. The user can also use plugins written with Java with xml files. An example of the Maven's pom.xml is presented in Figure 10.

The speed of the Maven framework is relatively good; parsing the xml files will not take long. As the parallel build support is experimental, it can only be used with caution; if no issues are seen, parallelization should be used to improve the build time. While building, Java uses quite a lot of memory, and in large projects the Java virtual machine may need more memory allocation than it has by default. The processor load from the framework is minimal.

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/maven-v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <parent> <groupId>fi.jkokko.maven.example</groupId>
                  <artifactId>parent</artifactId><version>0.1-SNAPSHOT</version>
</parent>
  <artifactId>example-parent</artifactId>  <packaging>pom</packaging>
<name>Example Project</name>
  <modules>
    <module>example1</module>
    <module>example2</module>
  </modules>
  <build>
    <pluginManagement>
    <plugins>      <plugin>       <groupId>org.apache.maven.plugins</groupId>
<artifactId>maven-compiler-plugin</artifactId>       <configuration>
<source>1.5</source>         <target>1.5</target>       </configuration>      </plugin>
</plugins>
    </pluginManagement>
  </build>
  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>3.8.1</version>
      <scope>test</scope>
    </dependency>
  </dependencies>
</project>
```

FIGURE 10. Maven example pom.xml.

Both robustness and repeatability are at a good level. During the testing no build errors due to the build framework were seen - except some misconfigurations by users. Also, all reproduced builds produced similar results. Most problems seen were because of failed external downloads; Maven seems to be a bit unstable with an encrypted tunneled proxy connection.

Most communications with other systems require a command-line interface. Maven itself is a command line tool which enables it to communicate with related external tools such as a code review and a build automation. Many development tools such as Eclipse and Jenkins support Maven directly.

Apache Maven has an active community under Apache Software Foundation. The community does not only error corrections but also implements new features, and new releases come few times a year. Also the licensing, Apache Software License, allows any kind of project to use Maven. Additionally, Maven has plugins for collecting software components' license info.

6 CODE REVIEW SYSTEM SELECTION

Code review systems try to increase the code quality by forcing every change to be reviewed by someone, be that a fellow developer or an automated test environment. The best result is usually achieved if the code review systems integrate with the software configuration management and build automation system along with the human reviewer.

A good code review system initiates when a change is done to the software configuration management, runs compilation, static analysis, light test automation and if possible unit testing. After machine testing has been done, the system should also notify a human reviewer to review the change. Also, as all open source projects, it needs an active community to support in problem cases and an adequate license to be used in commercial software development.

As software changes coming to a code review are not necessarily final and may even contain confidential information by mistake, it is good if the code review system separates commits under review from the codebase. Also, in commercial usage the access control mechanism needs to be adequate so that users on remote sites can work together without traveling to a review meeting.

6.1 Gerrit

Gerrit is an easy to use Java EE servlet based code review system designed to work with Git (8). Gerrit keeps commits in its own SQL database while waiting for a review, and pushes the commits to a codebase only after the review has been successfully approved. As a web based system, it also enables code reviewing by someone located on the other side of the world, so multisite projects benefit from this.

Gerrit has a multi-layer access control model, meaning that each project can have both common settings and project's separate settings on what certain people or groups can do; also people can be part of different access groups. Access control can also be separated by what kind of commits can be done by certain users, what types of approvals can users give, etc. Gerrit also supports LDAP, OpenID and SSH so it can be arranged as a secure system even via public Internet.
Gerrit works well with Jenkins via plugin therefore build and test automation for each commit can be arranged via Jenkins. Also, REST API is supported but all features are not yet implemented to that.

Submitting a commit for the review requires only a push to Gerrit instead of a master repository, presented in Figure 11. The manual code review with commenting can be done with only few clicks, and the change is shown as a difference compared to the original code. Also, abandoning, submitting and rebasing the commit can be done easily, directly from the web user interface. Also, searching commits is possible but limited to the project, branch and hash code. Notifying the related parties can be done via e-mail.

```
git push origin # pushes changes to default remote
git push origin HEAD:refs/heads/sandbox/mynewbranch # pushes changes to
sandbox/mynewbranch, creates it if it doesn't exist
git push origin HEAD:refs/for/sandbox/mynewbranch # pushes changes to
sandbox/mynewbranch via Gerrit, creates it if it doesn't exist
```

FIGURE 11. Git push operations.

Gerrit has a quite active community that makes quite a few releases per year, containing both new features and error corrections. Also the licensing, Apache License 2.0, allows almost any kind of project to use Gerrit.

6.2 Review Board

Review Board is a feature-packed code review system written with Python and a Django framework (20). It keeps the changes in its own SQL database, but does not submit those into the version control like Gerrit. Review Board supports multiple SCM systems including Git, SVN and Mercurial, and it can even work with Github and similar services. One of the features is that Review Board can use cloud services such as Amazon S3 for storing the data. As a web based system, it also enables code reviewing by someone located on the other side of the world, therefore multisite projects benefit from this.

Review Board has a multi-layer access control model, meaning that each project can have both common settings and project's separate settings on what certain people or groups can do; also people can be part of different access groups. Access control can also be separated by what kind of commits can be done by certain users, what types of approvals can users give, etc. Review Board also supports LDAP, Active directory and NIS and SSH so it can be arranged as a secure system even via public internet.

Thanks to Python and Django, Review Board is extensible with almost any kind of module. Also, REST API is fully supported. During the usage, an extension enabling automatic change submits to the SCM system was not found but it would be possible to implement.

Submitting changes can be done either via uploading patches from the user interface or connecting review tool into a special branch or repository on the SCM system. The manual code review with commenting can be done quite easily, and even different lines of code can be commented along with attaching more files such as documentation. The change is shown as a difference compared to the original code. Also, abandoning and updating the patch can be done easily, directly from the web user interface, but submitting the approved change to the master repository requires manual work. Also, searching commits is possible and well supported. Notifying the related parties can be done via e-mail.

Review Board has a relatively active community that does a few releases per year, containing both new features and error corrections. Also, the licensing, MIT license, allows almost any kind of project to use Review Board.

7 SELECTION OF THE SUPPORTING TOOLS

7.1 Introduction and Requirements

Build and release management needs a lot of supporting tools for various activities. Most important ones relate to build automation and code review. Other often used tools are test automation and error management related tools.

7.2 Build Automation

Build automation is almost always needed in order to support different kinds of builds. Good build automation system integrates with the SCM system and the build framework as well as with the error management and test automation. Build automation should also support multiple builds done in parallel and gather some metrics of the results. Also, easy configuration is useful if the software project requirements for builds vary a lot. Also, as all open source projects, it needs an active community to support in problem cases and an adequate license to be used in commercial software development.

There are two good open source build automation systems available; Jenkins and CruiseControl. They both manage their main task, which is doing builds automatically, extremely well. They both are written with Java and both can be set up to do builds periodically, continuously or by detecting changes on SCM system (12, 6). They both are also quite easy and fast to use, though Jenkins' web user interface for configuring builds is easier and faster. They both also have free licensing; Jenkins is licensed under an MIT License and CruiseControl under a BSD-style license. The biggest difference between these two is that CruiseControl has stopped support in September 2010 whereas Jenkins project releases even several times a week.

7.3 Test Automation

Test automation availability depends on the type of project but it should have at least a two-way interface to communicate with build automation and code review systems. Test automation system should report the test results in a both machine and human readable format so that the results can be used in decision making, and if something fails, the report tells what and why. For quality purposes, also a code coverage and unit testing should be run. From the build management point of view the test automation system needs a two-way communication - what to test and how the test went - and possibly a provided build support.

Due to the complex nature of the wireless devices, no open source test framework exists that can be directly used with it, even though Robot Framework might be possible to expand enough thanks to its modular structure and free license, Apache License 2.0 (21). For example a conformance testing with TTCN is not possible with open source tools, commercial ones such as OpenTTCN (17) need to be used. There are some tools that can be used to help getting better software, for example Splint (24) is a good choice for unit testing any C++ source code and JUnit for unit testing Java source code. Coverity (5) is a great tool for static analysis of a C++ source code but it is free only when used with an open source code.

7.4 Error and Feature Management Related Tools

Error and feature management tools are often used separately and manually, but it is possible to get for example static analysis tools to report errors automatically. This will improve project tracking as errors and their corrections are often a major part of the development effort. Manual usage requires a fast and logical user interface.

There are quite a few possibilities to choose from, but most popular ones that have either a command line or a web interface for automation purposes are Bugzilla (4), Jira (13) and Trac (26). Of these three Jira is the most advanced one, supporting multiple types of issues such as change requests and features, but it is free only for open source projects. Bugzilla and Trac rely on issues that can have special states such as an enhancement. From the build management point of view any of these are well usable, therefore the selection needs to be done based on other requirements.

7.5 Delivery Methods

All software projects focus on delivering a product to a customer. In large projects this usually requires an appointed release manager that ensures that all needed data is delivered. The delivery usually consists of the compiled binaries and related documentation, sometimes including also the source code. Usually there is not a single tool to publish the delivery fully, but instead multiple tools are required.

In commercial open source projects there are plenty of tools to be used for publishing - for example Sourceforge is a popular publishing channel for project binaries, source code and documentation (23). Another popular publishing tool is GitHub but it focuses almost purely on source code sharing (10).

On projects working with complex wireless devices, the open source channels are usually forbidden, and each delivery has a specific customer. The delivery content usually consists of the product binaries and a release note document, but it may also contain other data such as usage instructions and separate test reports. In almost all cases the documents are created with Microsoft products, even on projects focused on Linux systems.

The delivery transfer needs to be done securely; quite common ways are to use SSH protected connections to a secure server or to use an internal dedicated tool, but it is not unheard of to do the delivery via portable media by hand. When the delivery is done internally, also the version control or even a network share can be used if the network itself is protected.

The software effort's selected software development model affects a lot to the delivery. Usually, a waterfall based development model means rare deliveries with a vast documentation whereas an agile model means frequent deliveries with a lighter documentation, and occasionally with a more specific documentation.

8 CONCLUSIONS

Each complex wireless device project has its specific needs but these guidelines will help selecting the correct tools. In some areas there is only one good option, in some other areas almost any of the presented tools is at least sufficient.

Hardware selection for each tool should be done based on the tool requirements and using the already available hardware when possible. Most of the presented applications and all recommended ones are designed to work on Linux, so that it should be used as an operating system if possible.

Both Git and Mercurial are good choices for project's SCM system - SVN may be a bit lightweight for a large project. If the selected development model is close to waterfall, Mercurial would be a better candidate as it is easier to use than Git, but if the project has iterative cycles and branching is needed then Git is the best selection. If the codebase branching needs are not known at the startup phase of the project, then Git is the safer option even it requires a bit more training than Mercurial.

Build framework selection depends on the target device of the project. For most cases, the Bitbake is the fastest and most flexible build framework of the presented ones. If the target device is based on embedded Linux, then the Yocto framework is a better choice as it already includes the basic software and a lot of tools.

Code review system selection is a bit two-fold; if Git is used as a SCM system then Gerrit is a better option due to its tight integration with Git, even if the Review Board has a lot of features that are not included in Git and they would be somewhat beneficial for the project. If any other SCM system is used, then the Review Board is the only option.

Supporting tools, excluding the build automation tool, are almost completely dictated by project's needs unrelated to build and release management: The delivery methods are dictated by the project customer; error and feature management related tools are dictated by the project's issue tracking needs. The build automation should rely on Jenkins as it has an active community.

On a complex wireless device project many kinds of different issues needs to be handled, therefore Jira would be a good candidate but also Bugzilla and Trac are manageable. Test automation needs are specific to the project but at least unit tests and static analysis should be run for each programming language used - Splint and Coverity are good choices for the C++ code.

On a project that uses agile methods and develops software for a wireless embedded Linux device, the tools selection would be simple. Git would be the selection as a SCM system, Yocto the build framework, Gerrit the code review system and Jenkins the build automation system.

REFERENCES

1. Android Developers. Date of retrieval 25.08.2013.
   http://developer.android.com

2. Ant. Date of retrieval 05.10.2013.
   http://ant.apache.org

3. Bitbake. Date of retrieval 22.09.2013.
   http://docs.openembedded.org/bitbake/html

4. Bugzilla. 2013. Date of retrieval 30.10.2013.
   http://www.bugzilla.org

5. Coverity. 2013. Date of retrieval 30.10.2013.
   http://www.coverity.com

6. CruiseControl. 2013. Date of retrieval 12.10.2013.
   http://cruisecontrol.sourceforge.net/index.html

7. Eclipse. 2013. Date of retrieval 25.08.2013.
   http://www.eclipse.org

8. Gerrit. 2013. Date of retrieval 12.10.2013.
   https://code.google.com/p/gerrit

9. Git. 2013. Date of retrieval 22.09.2013.
   http://git-scm.com/about

10. Github. 2013. Date of retrieval 22.09.2013.
    https://github.com

11. JUnit. 2013. Date of retrieval 30.10.2013.
    http://junit.org

12. Jenkins. 2013. Date of retrieval 12.10.2013.
http://jenkins-ci.org

13. Jira. 2013. Date of retrieval 30.10.2013.
https://www.atlassian.com/software/jira

14. Kokko, J.T. 2013. Software Build Management Peculiarities when Working for a Wireless Device. T863403 Seminar on Wireless Future 3 Cr. Seminar paper in autumn 2013. Oulu: Oulu University of Applied Sciences, Degree Programme in Information Technology.
http://www.oamk.fi/~karil/mit_studies/wireless_future_seminar/papers2013/final_paper_kokko_jukka.doc

15. Maven. 2013. Date of retrieval 05.10.2013.
http://maven.apache.org

16. Mercurial. 2013. Date of retrieval 22.09.2013.
http://mercurial.selenic.com

17. OpenTTCN. 2013. Date of retrieval 30.10.2013.
http://www.openttcn.com

18. Rational Clearcase. 2013. Date of retrieval 25.08.2013.
http://www-03.ibm.com/software/products/us/en/clearcase

19. Repo Tool. 2013. Date of retrieval 22.09.2013.
https://code.google.com/p/git-repo

20. Review Board. 2013. Date of retrieval 12.10.2013.
http://www.reviewboard.org

21. Robot Framework. 2013. Date of retrieval 12.10.2013.
https://code.google.com/p/robotframework

22. SVN. 2013. Date of retrieval 22.09.2013.
http://subversion.apache.org

23. Sourceforge. 2013. Date of retrieval 12.10.2013.
http://sourceforge.net

24. Splint. 2013. Date of retrieval 30.10.2013.
http://www.splint.org

25. Tomayko, J.E. 1990. Software Configuration Management. Date of retrieval 25.08.2013.
http://www.sei.cmu.edu/reports/87cm004.pdf

26. Trac. 2013. Date of retrieval 30.10.2013.
http://trac.edgewall.org

27. Yocto. 2013. Date of retrieval 06.10.2013.
https://www.yoctoproject.org

Installing Test Environment *APPENDIX 1*

Test environment was selected to be Ubuntu Linux 12.04 amd64 as a Virtualbox virtual operating system, and OpenEmbedded project's source code as the test source code. Selection was done like this as both are easily available. This attachment describes how to do the basic install.

Basic install:

Download Ubuntu 12.04 amd64 image or use existing one

Create new VM, use downloaded or existing .vdi

- Set memory usage, processor amounts, etc. according to your hardware - leave at least 2GB of memory and 1 processor for other usage

    - recommended for Dell latitude E6430: 3 cores, 4GB of memory, 3D acceleration on, shared clipboard bidirectional, shared folder with name "shared"

Increase disk size (if needed) with VBoxManage and Gparted live cd; recommendation is to have 100GB

- "C:\Program Files\Oracle\VirtualBox\VBoxManage.exe" modifyhd "c:\Users\kokkjuk\VirtualBox VMs\Ubuntu 12.04.vdi" --resize=102400

- boot virtual machine from gparted live cd (gparted.sourceforge.net/download.php)

    - add .iso to virtual machine

    - change boot order to cd first

- resize partitions

- remove the gparted image from VM configuration or change boot order to HDD first

Create admin account for yourself (if you downloaded default ready-made Ubuntu image, password is "reverse")

Change keyboard layout to correct one

Install Guest Additions (VM's Devices menu)

Restart machine

If shared clipboard, etc. are enabled but not working, reinstall:

- Cd /opt/VBoxGuestAdditions-4.2.6

- sudo sh ./uninstall.sh

- install Guest Additions (VM's Devices menu)

- restart machine

Set http proxy if needed

sudo gedit /etc/apt/sources.list

Uncomment lines:

deb http://archive.canonical.com/ubuntu precise partner

deb-src http://archive.canonical.com/ubuntu precise partner

sudo apt-get update

sudo apt-get upgrade

ssh-keygen -t rsa -C "Jukka.Kokko@company.com"

(Or copy generated ssh keys from elsewhere)

sudo apt-get install cifs-utils

mkdir ~/shared

sudo mount -t vboxsf shared ~/shared

Install needed tools for OpenEmbedded (http://www.openembedded.org):

sudo apt-get install gawk wget git-core diffstat unzip texinfo build-essential chrpath libsdl1.2-dev xterm

Download and build OpenEmbedded standalone source:

git clone git://git.openembedded.org/openembedded-core oe-core

cd oe-core

git clone git://git.openembedded.org/bitbake bitbake

source ./oe-init-build-env

bitbake core-image-minimal