

# Improve your Team's Productivity: Setting up Continuous Integration on Symbian OS

Penrillian

Published by the Symbian Developer Network

Version: 1.0 – May 2008

<b>1</b>	<b>INTRODUCTION .....</b>	<b>2</b>
<b>2</b>	<b>VERSION CONTROL.....</b>	<b>2</b>
<b>3</b>	<b>WHERE TO FIND THE TOOLS AND SCRIPTS.....</b>	<b>2</b>
<b>4</b>	<b>THE BUILD PROCESS.....</b>	<b>3</b>
	4.1 THE BUILD.PROPERTIES FILE.....	3
	4.2 ANT TARGETS .....	4
	4.3 BUILDING FOR S60 AND UIQ SIMULTANEOUSLY .....	4
	4.4 DETECTING ERRORS IN THE BUILD .....	5
<b>5</b>	<b>DESCRIPTION OF EACH MAIN TARGET.....</b>	<b>5</b>
	5.1 THE CLEAN TARGET .....	5
	5.2 THE TESTS-HARDWARE-DEBUG-S60 TARGET .....	6
	5.3 THE BUILD-RUN-TESTS-EMULATOR TARGET .....	8
<b>6</b>	<b>USING CRUISECONTROL.....</b>	<b>9</b>
	6.1 THE BUILD ANT SCRIPT .....	10
<b>7</b>	<b>CONCLUSION .....</b>	<b>11</b>
<b>8</b>	<b>REFERENCES .....</b>	<b>11</b>

## 1 Introduction

At Penrillian<sup>[1]</sup> we find our most valuable asset is the ability to respond quickly to the changing needs of our customers. One of the ways we do this is to use Test Driven Development, where the majority of a piece of software is automatically tested almost every time it's built; this makes the software very robust, so we can accept changing requirements as 'business as usual.'

To help us do this, we've extended our automated test environment so that the tests run automatically every time a change is submitted to our version control system. We're immediately notified of any issues that occur, enabling us to spot any problem at the earliest opportunity and minimize waste. This gives a significant improvement in team productivity.

You too can benefit from this improved automated test environment. In this article we'll explain how our continuous integration<sup>[2]</sup> process automatically builds and tests S60 and UIQ versions of a sample Symbian OS application. To do this we use *CruiseControl*<sup>[3][4]</sup> to control the build process (including testing), *Apache Ant*<sup>[5][6]</sup> to perform the build process itself, and *Subversion*<sup>[7][8]</sup> for version control or Source Code Management (SCM)<sup>[9]</sup>. *CruiseControl*, *Ant*, and *Subversion* are all open source software tools.

Setting up continuous integration is straightforward – you can probably install everything and get a machine set up with the *SymbianOSUnit*<sup>[10]</sup> sample project in under a day.

## 2 Version Control

These days, it would be a rare for a project not to have all of its resources under some form of version control. To perform continuous integration, use of version control is a prerequisite, because it is the commit of an update to the SCM tool that triggers the build process. In addition, should a problem be identified with a build, the use of version control allows developers to rollback to a working version rapidly and with ease.

While we are using *Subversion* here, it is only one small, albeit critical, step in the continuous integration script, and so it can easily be substituted by another SCM tool, if necessary.

Note that the location of the SCM repository is not tied to the location of the servers running the build process. In the example presented here, our *Subversion* repository resides on *SourceForge*<sup>[11]</sup>, yet the build process runs on a machine within our office.

## 3 Where to find the Tools and Scripts

All these scripts are available with the *SymbianOSUnit* distribution<sup>[12]</sup>. The particular locations within the package of the scripts we'll be discussing are as follows:

```
tutorial/buid.xml
CruiseControl/SOSUCruiseBuild.xml
CruiseControl/SOSUCruiseConfig.xml
CruiseControl/buid.properties
```

We'll also discuss revised versions of a couple of Symbian scripts, which can be found at:

```
Tools/abld.pl
Tools/abld_UIQ.pl
```

You can download the Open Source tools you'll need from the sites listed in the References section of this article – see<sup>[1]</sup>, etc. The versions of the software we used in the configuration described in this article are as follows:

- CruiseControl 2.6
- Apache Ant 1.6.5
- Subversion 1.4.2
- Perl 5.6.1
- Series 60 Third Edition FP1 SDK
- UIQ 3.0 SDK.

However, we'd recommend using the latest version of *CruiseControl*, *Ant*, and *Subversion*, since the scripts don't use any unusual or out of date features of these applications.

## 4 The Build Process

Regardless of using continuous integration, we first need a script to build and test our application. Since it is the script that we later present to *CruiseControl*, we shall need to keep in mind its eventual purpose.

We use *Ant* to automate the build and testing processes. *Ant*'s scripts are written in XML, and the default build script is named `build.xml`.

The first thing you notice about `build.xml` is that it comprises a group of properties and a number of targets.

*Ant* properties are basically populated strings. For example:

```
<property name="tests.report.dir" location=".. /logs" />
```

They are provided with a little extra functionality, however, in that they can reference other populated strings (via the syntax, `${property_name}`). It is common practice to extract from `build.xml` those properties that will require local customization and place them in a file named `build.properties`. Properties defined in this file are added to `build.xml` via the line:

```
<property file=".. /CruiseControl /build.properties" />
```

One especial advantage of this is that we can reference this property file from other parts of the build, particularly the *CruiseControl* configuration, as we'll see later in this article.

### 4.1 The build.properties file

Our own properties file contains the locations of the main project configuration files, plus versions of the standard EPOC environment variables for both UIQ and S60. The final settings tell *CruiseControl* how to send email notifications when the build fails.

```
ProjectName=SymbianOSUnit
SOSUFolder=c:\projects\SymbianOSUnit
SOSUBuildFile=SOSUCruiseBuild.xml

S60_EPOCROOT=\\Symbian\9.2\S60_3rd_FP1\
S60_EPOCPLAT=Series 60 v3.1
S60_EPOCPATH=C:\Program Files\CSL Arm
Toolchain\bin; C:\Symbian\9.2\S60_3rd_FP1\EP0C32\gcc\bin; C:\Symbian\9.2\S60_3rd_FP1\Epoc32\Tools; C:\Program Files\Common
```

```

Files\Symbian\tools;
UIQ_EPOCROOT=\\Symbian\UIQ3SDK\
UIQ_EPOCPLAT=UIQ_v3
UIQ_EPOCPATH=C:\Program Files\CSL Arm
Toolchain\bin; C:\Symbian\UIQ3SDK\epoc32\gcc\bin; C:\Symbian\UIQ3SDK
\epoc32\tools\; C:\Program Files\Common Files\Symbian\tools;
CruiseControlHome=C:\Program Files\CruiseControl

MailHost=smtp.gmail.com
MailPort=465
MailUser=<USER_NAME>@gmail.com
MailPassword=<PASSWORD>
BuildResultsURL=<URL>
UseSSL=true
EmailTo=<EMAIL TO RECEIVE CRUISE CONTROL REPORTS>

```

## 4.2 Ant Targets

*Ant* targets are discrete functions that can be both dependencies of other targets and also called by those targets. Here's an example *Ant* target:

```

<target name="clean" description="Removes all build artefacts" >
    target content...
</target>

```

Each target can be an entry point for an *Ant* run. Because we are using *CruiseControl* to run the build process, we can enter the script multiple times to complete our task; this keeps our script loosely coupled, making it easier to test and provide more flexible functionality.

We can also 'call' *Ant* targets, as if they were procedure calls, using the `<ant_call>` operation. We've used this feature to simplify some of the scripts.

## 4.3 Building for S60 and UIQ simultaneously

To build and test both S60 and UIQ versions of the same application using *Ant* and its `build.xml`, we run *Ant* four times using the following sequence of targets:

1. `clean`
2. `tests-hardware-debug-s60`
3. `tests-hardware-debug-uiq`
4. `build-run-tests-emulator`.

The only dependencies between these steps are that `clean` must run first, and `build-run-tests-emulator` must run last.

`build.xml` provides the following functionality:

- supports a rigorous clean-up process
- invokes `blmake` and `abld` – with appropriate parameters
- creates the signed SIS file
- runs the emulator test programs.

The interesting functionality in these targets is the lack of reliance on machine-based environment variables – specifically `EPOCROOT`, `EPOCPLAT`, and `EPOCPATH`. We cannot use these environment variables because we will be building on a remote machine whose environment we cannot guarantee, and changing the environment on the build machine to suit ourselves has the potential

to break the system for other users. Similarly, we cannot depend on the kit devices' setting that allows you to switch development configurations between Symbian OS devices on installed SDKs.

The *Path* environment variable is constructed from the system's path and EPOCPATH (see target `init_s60`), so again it is not necessary, nor desirable, to include Symbian paths in the build machine's path; let the *Ant* script do it for you.

The `createsis` target performs the essential post build functions of making and signing the `.sis` file.

## 4.4 Detecting Errors in the build

As experienced Symbian OS programmers will know well, spotting errors in the Symbian `abld` build can be tricky. *CruiseControl* needs to have automatic error detection, so it's essential that `abld` generates an error when it fails. Unfortunately the standard versions of `abld` do not do that. To solve the problem, we've created different versions of the `abld.pl` script, which correctly detect failures; these are called from the `abld_s60` and `abld_uiq` targets, and can be found in the Tools directory of the *SymbianOSUnit* release<sup>[1]</sup>.

So, for example, here's the `abld_s60` target, which takes two parameters, `{dir}` for the directory where it's invoked, and `{command}` with is the `abld` command line:

```
<target name="abld_s60" >
  <exec executable="perl" failonerror="{fail}" dir="{dir}">
    <env key="EPOCROOT" value="{S60_EPOCROOT}" />
    <env key="Path" path="{S60_EPOCPATH}:{env.Path}" />
    <arg line="-I${S60_EPOCROOT}/Epoc32/tools {tools.dir}/abld.pl {dir}/
{command}" />
  </exec>
</target>
```

## 5 Description of each main target

The following sections examine each target in turn and explain what they do. It may be helpful to look at the actual code too to see the whole picture.

### 5.1 The Clean target

The `clean` target removes all build artefacts present in the current build path.

```
<target name="clean" description="Removes all build artefacts" depends="clean_s60,
clean_uiq" >
  <echo message="Clean completed." />
</target>
```

This target simply calls a `clean` for the S60 build and one for the UIQ build. Since these two `clean` targets are essentially identical, we'll examine only the S60 `clean` target.

We start with the target identifier:

```
<target name="clean_s60"
  description="Removes all build artefacts for S60" >
```

Next, in preparation for the `abld` step, we chop off the front 'c:' from the full path name of our project's group directory.

```

<pathconvert property="tests.build.path.s60" targetos="unix">
  <map from="c:" to="" />
  <path location="${tests.group.dir.s60}" />
</pathconvert>

```

As an example, this converts:

```

c:\projects\SymbianOSUnit\Tutorial\group\S60_3RD\
into
\projects\SymbianOSUnit\Tutorial\group\S60_3RD\

```

This edited property is later used (in the `abl d` step) by appending it to the Symbian build path.

Next, we simply call the `abl d_s60` target with three parameters.

```

<antcall target="abl d_s60">
  <param name="dir" value="${tests.build.path.s60}" />
  <param name="fail" value="false" />
  <param name="command" value="test -k realclean" />
</antcall>

```

`abl d_s60` simply calls a version of `abl d` with the parameters supplied to it. In this case, we perform a `realclean`.

The next step in the `clean` target is similar to the last, but this time calls `clean` on `bl dmake.pl`. This is the equivalent to invoking `bl dmake clean` on the command line.

```

<exec executable="perl" dir="${tests.build.path.s60}" failonerror="false">
  <env key="Path" path="${S60_EPOCPATH}:${env.Path}" />
  <env key="EPOCROOT" value="${S60_EPOCROOT}" />
  <arg line="-I${S60_EPOCROOT}/Epoc32/tools
${S60_EPOCROOT}/Epoc32/tools/bl dmake.pl clean" />
</exec>

```

Finally, we clear up a few straggling files that aren't deleted by the Symbian OS build chain.

```

<delete quiet="true" includeEmptyDirs="true">
  <fileset dir="${tests.group.dir.s60}" includes="*.mbmdef" />
  <fileset dir="${tests.group.dir.s60}" includes="TEMPMAKSYSDECOY*" />

  <fileset dir="${tests.output.dir.s60}" includes="SOSUnit.conf" />
  <fileset dir="${tests.sis.dir.s60}" includes="udeb.*" />
  <fileset dir="${tests.sis.dir.s60}" includes="*.SIS" />
  <fileset dir="${tests.report.dir}" includes="junit*.html" />
  <fileset dir="${tests.report.dir}" includes="TEST-*.xml" />
</delete>
<echo message="S60 clean completed." />
</target>

```

## 5.2 The *tests-hardware-debug-s60* target

This build target builds the entire S60 native application, and decides automatically if any errors have been found.

This target has a dependency on `init_s60`, which sets up *Ant* variables and runs `bl dmake` on the project. Then we call `abl d_s60` with the application's build parameters.

```

<target name="tests-hardware-debug-s60" description="Builds the unit tests for
Target Debug S60E3" depends="init_s60" >
  <antcall target="abld_s60">
    <param name="dir" value="${tests.build.path.s60}" />
    <param name="fail" value="true" />
    <param name="command" value="test build gcce udeb" />
  </antcall>
  <antcall target="createsis">
    <param name="sdk" value="S60" />
    <param name="epocroot" value="${S60_EPOCROOT}" />
    <param name="appname" value="${testapp.name}" />
    <param name="pkgdir" value="${tests.sis.dir.s60}" />
  </antcall>
</target>

```

Providing the `abld_s60` build completes successfully, we next run `createsis`, which creates the `.sis`, and then signs it, creating the `.sisx`.

Since Carbide.c++ `.pkg` files contain file path macros, such as `$(EPOCROOT)`, we need to pre-process these files (as Carbide.c++ does). We use a simple Perl command-line script to do this. So the `createsis` target starts by taking a copy of the application's `.pkg` file, then by running a Perl script on it that replaces references to environment variables within the `.pkg` file with the data from the respective environment variables.

```

<target name="createsis" >
  <copy file="${pkgdir}/${appname}.pkg" tofile="${pkgdir}/udeb.pkg"
  overwrite="true" />
  <exec executable="perl" failonerror="true">
    <env key="PLATFORM" value="gcce" />
    <env key="TARGET" value="udeb" />
    <env key="EPOCROOT" value="${epocroot}" />
    <arg value="-p" />
    <arg value="-ibak" />
    <arg value="-e" />
    <arg value="s/\$\((\w*)\)/$ENV{$1}/g;" />
    <arg value="${pkgdir}/udeb.pkg" />
  </exec>

```

Next, we create the `.sis` file, and in case `makesis` fails silently – without setting a non-zero return code – we force a failure should the `.sis` file not have been created.

```

<exec executable="makesis" failonerror="true">
  <arg value="${pkgdir}/udeb.pkg" />
</exec>
<fail>
  <condition>
    <not> <available file="${pkgdir}/udeb.SIS" /> </not>
  </condition>
  Error: ${sdk} udeb.sis is missing.
</fail>

```

Finally, we sign the `.sis`, producing a `.sisx` file, and tidy up after ourselves.

```

<exec executable="signsis" failonerror="true">
  <arg value="${pkgdir}/udeb.SIS" />
  <arg value="${pkgdir}/${appname}udeb.sisx" />
  <arg value="${signing.certificate}" />
  <arg value="${signing.key}" />

```

```

</exec>
<delete quiet="false" >
  <fileset dir="{pkgdir}" includes="udeb.*" />
</delete>
<echo message="{sdk} {appname}udeb.sisx created successfully" />
</target>

```

The `tests-hardware-debug-uiq` target is identical to `tests-hardware-debug-s60` with the obvious editing of `s60` to `uiq`.

### 5.3 The *build-run-tests-emulator* target

This is the most interesting of the targets; it builds the emulator versions for both S60 and UIQ and runs the tests on both – detecting errors in both the build and the test run.

Before running the tests, we first build the emulator executables. This suggests the target's dependencies, as follows:

```

<target name="build-run-tests-emulator"
  description="Builds, runs and analyses all unit tests"
  depends="build-tests-emulator-debug, run-tests-emulator_S60, run-tests-
  emulator_UIQ" />

```

The first dependency target, `build-tests-emulator-debug`, runs `blmake` (via `init_s60`) and `abld` (via `abld_s60`) with the command `'test build wincw udeb'` for both S60 and UIQ SDKs.

With the emulator executables built, we can run the tests.

```

<target name="run-tests-emulator_S60"
  description="Runs and analyses all unit tests" >
  <exec executable="{tests.executable.s60}" dir="{tests.output.dir.s60}">
    <env key="Path" path="{S60_EPOCPATH}:{env.Path}" />
    <arg value="x" />
    <arg value="x" />
    <arg value="tail" />
  </exec>

```

The odd-looking arguments above require some explanation. The emulator has not been designed to support applications receiving parameters when the application is invoked from the MS Windows command line. For historical reasons, however, the framework will pass information from the command line to the application, as long as it is in a suitable format. The format we've used is:

- `x x tail`

The first two parameters are ignored by *SymbianOSUnit*, while the `tail` text is received by the application. Any non-null value of the `tail` -part causes the tests to run automatically.<sup>1</sup>

---

<sup>1</sup> If you're using the normal Carbide.c++ development environment, this also allows you to run emulator tests automatically without waiting for user input. You'll need to enter the same string `'x x tail'` in the 'Program Arguments' box in the 'Arguments' tab of the Carbide.c++ Debug Configurations Dialog. You'll also need to remove the contents of the 'Emulator' field in the 'Main' tab of the same dialog, to run the program without needing the emulator menu.

The next step collects the `.xml` files created by the test run and reformats them into a file called `TESTS-TestSuites.xml`, then places it in `${tests.report.dir}`. The report element generates a report based on `TESTS-TestSuites.xml` (called `junitt-noframes.html`) that can be opened in a browser.

```
<mkdir dir="${tests.report.dir}" />
<junitreport todir="${tests.report.dir}" >
  <fileset dir="${tests.output.dir.s60}">
    <include name="*.xml" />
  </fileset>
  <report format="noframes" todir="${tests.report.dir}" />
</junitreport>
```

Finally we must check the output from the test runs to see whether any tests have failed. The best way to do this is to ensure that the log from the tests contains the text `'testsuite errors="0" failures="0"`, and abort if it doesn't.

```
<condition property="tests.all.passed" >
  <isfileselected file="${tests.output.file.s60}" >
    <contains text='testsuite errors="0" failures="0' />
  </isfileselected>
</condition>
<fail message="Tests failed" unless="tests.all.passed" />
<echo message="All S60 tests passed." />
</target>
```

## 6 Using CruiseControl

The easiest way to use *CruiseControl* is to let it install itself as a service. Thereafter, you simply point its `config.xml` (found in the installation directory) at your project's `config.xml` (for *SymbianOSUnit* we called this file `SOSUCruiseConfig.xml`), which in turn points to your project's `build.xml`. For example, our simple *CruiseControl* `config.xml` is:

```
<cruisecontrol >
  <include.projects file="SOSUCruiseConfig.xml" />
</cruisecontrol >
```

Have a look at the `SOSUCruiseConfig.xml` to see how it works. It has several sections of interest.

*CruiseControl* detects changes to a project by retrieving changes to the version control system. The `modificationset` determines whether an update has taken place by examining the local source of the project, which itself contains meta data about the *Subversion* repository, which in turn enables it to detect an update to the mainline.

```
<modificationset quietperiod="0">
  <svn localworkingcopy="${SOSUFolder}" />
</modificationset>
```

*CruiseControl* supports most other mainstream version control systems, such as Perforce and CVS. Consult the *CruiseControl* documentation to see how to set it up for these – it's pretty straightforward.

The `schedule` section tells it how often to make the check, and how to build if there are changes. We want to use the *Ant* script we discussed earlier in this article.

```
<schedule interval="30">
  <ant anthome="${env.ANT_HOME}" />
</schedule>
```

```

    buildfile="${SOSUBBuildfile}"
    target="build_all"
    uselogger="true"
    usedebug="false" />
</schedule>

```

Listeners start before a build; you can have multiple listeners. Currentbuildstatuslistener provides *CruiseControl* with a mechanism to determine whether it is currently building a project, so that it doesn't start a build when one is already running.

```

<listeners>
  <currentbuildstatuslistener file="logs/${project.name}/status.txt" />
</listeners>

```

Finally the publishers section instructs *CruiseControl* how to notify us of failures.

```

<publishers>
  <onfailure>
    <html email mailhost="${MailHost}"
      mailport="${MailPort}"
      username="${MailUser}"
      password="${MailPassword}"
      usessl="${UseSSL}"
      returnaddress="${EmailTo}"
      buildresulturl="http://${BuildResultURL}/"
      skipusers="true" spamwhilebroken="true"
    ...

```

## 6.1 The build Ant script

After all of the above, the main *CruiseControl* build script is a simple affair. It has a rule to check out all the code.

```

<target name="update" description="check out the latest files" >
  <exec executable="svn" dir="${SOSUFolder}" failonerror="true" >
    <arg line="update" />
  </exec>
</target>

```

And then it just invokes the appropriate clean and build targets in the main *Ant* script.

```

<target name="build" depends="update">
  <ant antfile="${SOSUFolder}\Tutorial\build.xml"
    dir="${SOSUFolder}/Tutorial"
    target="clean" >
  </ant>
  <ant antfile="${SOSUFolder}\Tutorial\build.xml"
    dir="${SOSUFolder}/Tutorial"
    target="tests-hardware-debug-s60" >
  </ant>
  ... etc.

```

## 7 Conclusion

Once *CruiseControl* has been installed and configured, as described above, it can sit there, *ad infinitum*, ensuring that your project mainline is always in a pristine state. When problems occur, *CruiseControl* will notify you immediately that there is something wrong and provide diagnostics enabling you to begin your analysis.

We at Penrillian find *CruiseControl* invaluable with our Symbian OS projects. You can easily modify the scripts described here to suit your own development process. We hope you find them as useful as we do.

## 8 References

- Penrillian's web site  
[www.penrillian.com](http://www.penrillian.com)
- Wikipedia article on Continuous Integration  
[en.wikipedia.org/wiki/Continuous\\_Integration](http://en.wikipedia.org/wiki/Continuous_Integration)
- Wikipedia article on revision control  
[en.wikipedia.org/wiki/Revision\\_control](http://en.wikipedia.org/wiki/Revision_control)
- Wikipedia article on Cruise Control  
[en.wikipedia.org/wiki/CruiseControl](http://en.wikipedia.org/wiki/CruiseControl)
- CruiseControl download  
[cruisecontrol.sourceforge.net/](http://cruisecontrol.sourceforge.net/)
- Wikipedia article on Apache Ant  
[en.wikipedia.org/wiki/Apache\\_Ant](http://en.wikipedia.org/wiki/Apache_Ant)
- Ant download  
[ant.apache.org/](http://ant.apache.org/)
- Wikipedia article on Subversion  
[en.wikipedia.org/wiki/Subversion\\_%28software%29](http://en.wikipedia.org/wiki/Subversion_%28software%29)
- Subversion download  
[subversion.tigris.org/](http://subversion.tigris.org/)
- Forrester report on the popularity of Subversion  
[www.collab.net/forrester\\_wave\\_report/index.html](http://www.collab.net/forrester_wave_report/index.html)
- SymbianOSUnit code repository  
[symbianosunit.svn.sourceforge.net/svnroot/symbianosunit/trunk](http://symbianosunit.svn.sourceforge.net/svnroot/symbianosunit/trunk)
- SymbianOSUnit home page.  
[www.symbianosunit.co.uk/](http://www.symbianosunit.co.uk/)