Electronica 2006 – Embedded Conference

Embedded Linux Quality Assurance: Unit Testing with Open Source

Author: Roland Stigge <stigge@philosys.de>

Abstract

In the Open Source world, there are well established tool suites in use for common build environments, e.g. *make*, *autotools*, which are increasingly utilized for quality assurance tasks. The goal is an integration of unit tests and more general testing with certain properties.

Testing must always be automated to the highest possible degree. Otherwise, developers won't employ these in their daily work. Unit testing doesn't necessarily mean exact obedience to Extreme Programming rules as its origin would suggest. Nevertheless, there must not be an unnecessary overhead of writing additional tests for new test cases, and actually running the tests must be easy enough for the developer to regard the process as a gain higher than the loss of time caused by it.

Therefore, fast and convenient *make test* runs should be available as it is also becoming more and more common in Open Source packages. Further, implementing test code should be an integral part of the overall development in a certain project, package or general programming effort. Therefore, it should be possible to work on test code alongside the general development on a project. Ideally, it is maintained in the same code repository for every developer (and other involved persons) to access and execute.

This way, it is even possible to run test suites fully automatically (e.g. a "nightly test" as known as from "nightly builds").

For every programming language, there exists at least one established and freely available unit testing framework, e.g. *JUnit* for Java, *CUnit* and *autounit* for C and *PyUnit* for Python. The common functionality here is the collection of test cases with test code provided by the respective developer and a standardized way of building up and tearing down a test harness.

The application of tests is not restricted to functions and methods but can be expanded to higher level functionality by implementing tests with scripting languages like Shell, Perl and Python that can directly utilize the development environment's features. This way, an automated test suite can cover several levels of functionality, from functions and methods to library and user interfaces.

For embedded development, especially in a Linux environment, special care must be taken for successfully applying the above mentioned methods. In general, they can be applied to embedded development as known in the common development case, but with some additional features and properties, like cross compilation and host-target connection (setup and running remote tests).

This paper describes a method for integrating tests in embedded software development projects, after introducing common unit test frameworks, the GNU/Linux build environment and the integration of both. A way of setting up a test harness in a host-target configuration will be presented.

As for general Quality Assurance, the goal is to automatically detect problems and bugs in the embedded software to be developed, therefore increasing quality by minimizing cost of testing. By integrating a high degree of software tests into the process early on, later manual tests can be minimized.

This presentation explicitly **does not cover** automated test code generation as known from some proprietary software testing tools.

Introduction

Quality Assurance in software development, especially in the Open Source world, is commonly under-represented in today's projects. This is typical for the whole business except for projects necessarily bound to the integration of Quality Assurance and testing into the process early on (e.g. aviation, medical instruments, nuclear power plants).

This paper concentrates on Unit Testing which enables the developer to supplement the source code of a project with a test suite that can be run in the highest automated way possible, thus minimizing time and cost of running tests, lowering the barrier of actually repeating tests everywhere in the development process and making the project schedule more predictable. Instead of developing fast and spending most of the time with bug fixing (especially in late phases of the project) and having a long and unpredictable debugging period, we can invest more time in defining functionality which can be tested automatically from C-function level (even assembler) upwards (actually, up to an impressive high level).

Other important Quality Assurance measures which can be of great help to most projects (but can't be discussed in detail here) are the following:

- Bug tracking systems (request tracker, ticket system, issue tracker), e.g. Request Tracker, Debbugs, OTRS
- Memory Monitor (memory profiler, leak tester), e.g. Valgrind, Electric Fence
- Software Code checkers, e.g. lint-like tools
- Compiler Warnings, e.g., gcc -Wall -Wextra (!)
- Black box testing frameworks, e.g. Dejagnu

In the following, we will discuss a certain scenario of developing software packages in the GNU / Open Source style (but not being bound to this paradigm license-wise). Some preconditions for this are:

- The target machine is powerful enough and has got enough resources to run a certain overhead of test software in it. This includes memory and connectivity (network or some kind of terminal access, e.g. serial connection)

- Most test cases for functions (in our case, C language functions) can be created with a harness in an automated way (e.g. necessary input sequences can be triggered in some way)

1. Tools and Frameworks

Traditional tool suites employed in Unit Testing are derived versions of the well known *JUnit* testing framework by Kent Beck and Erich Gamma, instantiated for the respective programming languages. Although relatively easily to implement in a new language, this kind of tool is very powerful and commonly respected which is reflected in the availability of the framework for virtually every language.

Sometimes, there are even multiple famous independent implementations for a single language available, depending on the kind of development environment. Common examples for this framework are:

- *JUnit* for Java
- *CUnit* and *autounit* for C
- *PyUnit/unittest* for Python
- *Test::Unit* and *TAP* for Perl

This is just an incomplete list, but you will find the respective version for your favourite language quite easily. Although unit testing is employed to a high degree in Extreme Programming, this testing strategy is not necessarily connected to XP in general. It can be used in many programming styles and adapted to it (e.g. frequency, time and number of test runs, evaluation of test protocols etc).

The tools used throughout this presentation comprise standard tools of the GNU world, where possible. This includes well-known frameworks like the *autotools* (*autoconf*, *automake*, *libtool*, *aclocal*, *autoheader*) and the additional tools called GNU *autounit*, *make*, *gcc*. They are well established and widely used in the Open Source world. Here, they are only described shortly, please refer to the man and info pages for details:

- ***make***:
  The standard build system under UNIX and GNU/Linux. Although alternatives exist, it is still the most widely used build framework driven by *Makefiles* which describe dependencies for building objects, executables and libraries. Running *make* automatically resolves dependencies.

- ***gcc***:
  The GNU Compiler Collection, often also referred to as "GNU C Compiler", although *gcc* includes compilers for Fortran, Java, C++, ObjectiveC, and more. It is used not only under GNU but also with BSD, Solaris etc.

- ***autoconf***:
  Configuring a portable program (usually written in C/C++) for one of several supported platforms is a difficult task. *autoconf* can automatically generate a *configure* script as common for GNU/Linux software packages and generate symbols which can just be included and used in programs. Besides that, a *configure* run creates the necessary *Makefiles* that are adapted to the build system.

- ***autoscan***:
  Simple tool to generate an initial *configure.ac* to start working with. Useful for first projects that already comprise a certain complexity of actual source code. The sources will be scanned and handled in *configure.ac* with a heuristic approach.

- ***automake***:
  autoconf uses *Makefile* templates (*Makefile.in*) as input. Those have long and complicated contents; *automake* can help generating those files with a set of simple definitions in *Makefile.am* files.

- ***aclocal***:
  autoconf input (*configure.ac*) is written in the M4 macro language. Usually, only some standard macros are used. Some of the *automake* related ones are included in *aclocal.m4* which can easily be generated with *aclocal*.

- ***autoheader***:
  autoconf creates a list of macro definitions that can be used by C programs (*config.h*). However, for certain cases it is useful to have a template list for these macros (*config.h.in*). This is created by *autoheader*. Ideally, *autoheader* wouldn't be necessary, but in practice, it helps with debugging.

- ***libtool***:
  A tool that helps creating static and shared libraries, automatically, and abstracting from these different concepts. Together with *autoconf*, this can adapt the software package to the actual target system.

- ***autoreconf***:
  This is a simple tool that calls programs like *autoconf*, *autoheader*, *aclocal*, *automake* and *libtoolize* and more in the right order for you.

- ***autounit***:
  A C library that is a derivative of the original Unit Testing framework by Kent Beck and Erich Gamma. It is well integrated with GNU *autotools*.

2. Applying the methods in an embedded environment

Setting up a project / package with GNU *autotools* is easily done everywhere (not to confuse with "well understood"!). To learn more about this in detail, you should read the documentation of the respective packages. A simple example for our specific field is given in the next section.

For a start, *autoscan* can produce a *configure.ac* (rename it from *configure.scan*) to start with, and a *Makefile.am* can look as simple as:

```
------------------------------------------------------------------------
bin_PROGRAMS = helloworld
------------------------------------------------------------------------
```

Assuming we already set up *configure.ac* and some *Makefile.am*'s, the next step would be writing tests for our actual code. Ideally, tests would be written before actual code. You can try to achieve this as hard as feasible, but in reality, development approaches often include some kind of evolutionary dynamics that will prevent you from writing all tests before the first "real" code line. Also, Extreme Programming (an origin of radical unit testing) applies to writing test cases for smaller sections of code.

For convenience, *automake* already provides a means for integrating tests into our code base. Listing executable files in the variable *TESTS* in *Makefile.am* will make *automake* aware of the tests located in the respective directory; *automake* will create the necessary rules for us to simply issue "make check" at build time. It is possible to integrate the tests along with the actual source code of our project (i.e., having test code in the same directory), or maintaining test suites in a separate directory. We strongly recommend the latter. This way, test code is well separated from the actual code, helping us to keep out pure test code from deployment, and still providing the ease of writing new test code.

All this usually doesn't apply to the Embedded case where we want to run an executable on another machine than the build machine. Although GNU *autotools* are aware of cross compiling, a*utomake* doesn't know how to execute programs on a remote host. Maybe this is related to the fact that Embedded development is too diverse, including to many completely different approaches to compiling, host-target setup etc. But the a*utomake* authors provide a useful variable to set up a host-target connection for tests, among other things: *TESTS_ENVIRONMENT*. Including setting up variables, it specifies the command to execute for invoking the test programs (instead of */bin/sh*). This makes it possible for us to do a target connection via SSH with:

```
TESTS_ENVIRONMENT=ssh username@targethost
```

Please note that this is *Makefile* syntax for variable assignment. The actual contents of this variable will be a bit different in reality (e.g., for getting the right environment on the target hosts), but the basic functionality is shown above. The important thing here is that the command to execute will be supplied as an argument so that SSH can execute the respective test program on the target. See the next section for actual construction.

Besides SSH (requiring a network connection), it would be possible to just have a serial connection to the target. The only things necessary are automatically deploying just compiled test code to the target, invoking the programs there and evaluating the return value and *stdout*. While in our case this is easy with NFS and SSH, it's relatively easy to implement a single client-server infrastructure to provide this functionality via different connection setups. E.g., another approach would be networking via serial lines or USB. There are many possibilities. For now, let's assume that this connection is done.

For the case of NFS exported volumes, the easiest case (as discussed in the following) is the presence of the same directory hierarchy under the software package directory on both development host and target. Then, the only thing left to do on the target is to change to the same directory as on the host and executing the test executable as would be done on the host.

3. An Example Test Setup

In the following, we will discuss our example in detail. The package (*hellotest-XX.XX.tar.gz*) can be downloaded from http://www.philosys.de/news/... It is a regular GNU style package, and besides setting up a cross compiler on your development host, you can use it as such. (See the file README.)

3.1. Basic Package Infrastructure

As you can see in the package's root *Makefile.am*:

```
---------------------------------------------------------------------------
SUBDIRS = src lib tests
test: check
---------------------------------------------------------------------------
```

The test code *(tests/ directory*) is separated from the actual code (*src/* and *lib/*). Normally, the GNU *autotools* target for tests is *check*, for convenience we also define *test* as an alias.

The contents of our *configure.ac* look like:

```
---------------------------------------------------------------------------
AC_PREREQ(2.59e)
AC_INIT(hellotest, 1.0, stigge@philosys.de)
AM_INIT_AUTOMAKE
AC_CONFIG_SRCDIR([src/main.c])
AC_CONFIG_HEADER([config.h])

# Checks for programs.
AC_PROG_CC
AC_PROG_LIBTOOL

# Checks for libraries.
AM_PATH_GLIB

# Checks for header files.
AC_HEADER_STDC
AC_CHECK_HEADERS([fcntl.h stdlib.h string.h termios.h unistd.h])

# Checks for typedefs, structures, and compiler characteristics.
AC_HEADER_TIME

AC_MSG_CHECKING(to see if we can add '-Wall -W' to CFLAGS)
if test x$GCC != x ; then
  CFLAGS="$CFLAGS -D_U_=\"__attribute__((unused))\" -Wall -W -D_GNU_SOURCE"
  AC_MSG_RESULT(yes)
else
  CFLAGS="-D_U_=\"\" $CFLAGS"
  AC_MSG_RESULT(no)
fi

# Checks for library functions.
AC_FUNC_SELECT_ARGTYPES
AC_CHECK_FUNCS([memset select strstr])

AC_OUTPUT([Makefile
           src/Makefile
           lib/Makefile
           tests/Makefile])
----------------------------------------------------------------------
```

Most of the lines *in configure.ac* are present to make sure that everything is in place (and actually checking, where exactly). This information will be used later in *Makefiles* to find the respective header files, dependent libraries etc. The last definition of *AC_OUTPUT* specifies, which *Makefiles* should be generated.

src/Makefile.am:

```
---------------------------------------------------------------------------
bin_PROGRAMS = hellotest

hellotest_SOURCES = hello.c main.c
noinst_HEADERS = hello.h
---------------------------------------------------------------------------
```

And *lib/Makefile.am*:

```
---------------------------------------------------------------------------
lib_LTLIBRARIES = libhellotest.la

libhellotest_la_SOURCES = hellotest.c
include_HEADERS = hellotest.h
---------------------------------------------------------------------------
```

As we can see, defining the sources and target file of a library is as easy as defining a program. Up to here, we have only defined the program and library under test. In the following, we will discuss test infrastructure and code of our example.

3.2. Tests

As discussed before, we can list actual tests in the *TESTS* variable in *Makefile.am*'s. Consider the example *Makefile.am* for our *tests/* directory:

```
---------------------------------------------------------------------------
TARGET_LOGIN=root@targetdev
TESTS_ENVIRONMENT=ssh $(TARGET_LOGIN) "cd `pwd` ; "

OUR_SCRIPT_TESTS = test-test
OUR_BINARY_TESTS = test-functions test-static test-library

OUR_TESTS = $(OUR_SCRIPT_TESTS) $(OUR_BINARY_TESTS)

TESTS = $(OUR_TESTS)

check_PROGRAMS = $(OUR_BINARY_TESTS)

test_functions_SOURCES = test-functions.c
test_functions_LDADD = ../src/hello.o
test_functions_LDFLAGS = -lau-c-unit $(GLIB_LIBS)
test_functions_CFLAGS = $(GLIB_CFLAGS)

test_static_SOURCES = test-static.c
test_static_LDFLAGS = -lau-c-unit $(GLIB_LIBS)
test_static_CFLAGS = $(GLIB_CFLAGS)

test_library_SOURCES = test-library.c
test_library_LDADD = ../lib/libhellotest.la
test_library_LDFLAGS = -lau-c-unit $(GLIB_LIBS)
test_library_CFLAGS = $(GLIB_CFLAGS) -I../lib/
```

```
check_SCRIPTS = $(OUR_SCRIPT_TESTS)
noinst_SCRIPTS = $(OUR_SCRIPT_TESTS)
EXTRA_DIST = $(OUR_SCRIPT_TESTS)
---------------------------------------------------------------------------
```

TESTS_ENVIRONMENT contains a command (*cd `pwd`*) to go to the respective directory on the target before running the actual executable. The test programs are divided into binary tests (our own variable *OUR_BINARY_TESTS*) and script tests (our own variable *OUR_SCRIPT_TESTS*). This is necessary to tell *automake* how to generate the necessary build rules. We specify these lists in the *check_PROGRAMS* and *check_SCRIPTS automake* variables to support the creation of the *chec*k rule. The different sections regarding function tests, library tests and tests of static C functions are discussed in the following subsections.

In the *automake/autounit* case, tests are divided into the tests listed in the *Makefile.am*. These programs can have multiple test case implementations each of which are run inside test harness code.

3.2.1. Function Tests

To test functions that are global to an object (non-static), we can just link our test code with the object from the actual source directory (as seen above in the *Makefile.am*) and use the functions:

```
---------------------------------------------------------------------------
#include <stdio.h>
#include <string.h>
#include <unistd.h>

#include <glib.h>

#include <autounit/autounit.h>

#include "../src/hello.h"

#define TTY_DEVICE "/dev/ttyX1"

gint
test_functions_tty_open(autounit_test_t *t)
{
  int fd;

  fd = dev_open(TTY_DEVICE);
  au_assert(t, fd >= 0, "dev_open didn't return valid file decriptor");

  if (fd >= 0) {
    dev_close(fd);
  }

  return TRUE;
}

[...]

static autounit_test_group_t tests[] = {
  {"test_functions_tty_open", test_functions_tty_open, TRUE, TRUE},
  {"test_functions_tty_open_badfile", test_functions_tty_open_badfile,
TRUE, TRUE},
  {"test_functions_tty_close", test_functions_tty_close, TRUE, TRUE},
  {"test_functions_tty_init", test_functions_tty_init, TRUE, TRUE},
  {"test_functions_tty_ok", test_functions_tty_ok, TRUE, TRUE},
  {0, 0, FALSE, FALSE}
};
```

```
int main(int argc _U_, char* argv[] _U_) {
  autounit_suite_t *test_suite;
  gint result;

  test_suite = au_new_suite(g_string_new("Function Tests"), 0, 0);
  au_add_test_group(test_suite, tests);
  result = au_run_suite(test_suite);

  au_delete_suite(test_suite);

  return result;
}
---------------------------------------------------------------------------
```

As we can see here, multiple tests can be specified in a group and be run together in an initialized test suite (the *au_* prefix refers to *autounit*). In the above example, we just set the last two arguments of *au_new_suite()* to zero, which can be actual function pointers to setup and teardown functions which would be called before and after every test function call. In our example, the small setup code is contained in the test functions. Besides, the setup and teardown code can only be reused and set via *au_new_suite* if it is equal for all functions in the suite which is not the case here.

Within the actual tests, several assertions can be defined which must be fulfilled at the respective point in the code. Otherwise, the whole test will be counted as *FAILED*. The result of all tests will be accumulated and displayed in the end, as we will see later.

3.2.2. Library Tests

As the *Makefile.am* suggested, we can just link our test code together with the library (**.la* abstraction for static and shared libraries) under *test*. With *libtool* created libraries it is even not yet decided if it's a static or shared library. The respective actions will be taken at configure time.

Here is an example of library testing code:

```
---------------------------------------------------------------------------
#include <stdio.h>
#include <string.h>
#include <unistd.h>

#include <glib.h>
#include <autounit/autounit.h>

#include <hellotest.h>

gint
test_library_say_hello(autounit_test_t *t)
{
  char *s;

  s = say_hello();
  au_assert(t, s != NULL, "say_hello() returned NULL");
  au_assert(t, strlen(s) == 6, "say_hello() returned string of wrong
length");
  au_assert(t, strcmp(s, "Hello!") == 0, "say_hello() returned wrong
string");

  return TRUE;
}

[...]

int main(int argc _U_, char* argv[] _U_) {
  autounit_suite_t *test_suite = NULL;
  autounit_suite_t *stresstest_suite = NULL;
  gint result;

  test_suite = au_new_suite(g_string_new("Library Tests"), 0, 0);
  au_add_test_group(test_suite, tests);
  result = au_run_suite(test_suite);

  au_delete_suite(test_suite);

  if (!result) {
    stresstest_suite = au_new_suite(g_string_new("Stress Tests"), 0, 0);
    au_add_test_group(stresstest_suite, tests);
    result = au_run_stress_suite(stresstest_suite, 100, 5);

    au_delete_suite(stresstest_suite);
  }

  return result;
}
---------------------------------------------------------------------------
```

Thus, library test code is quite similar to normal function tests. In addition, the above example contains stress test code for the library (run in the successful normal test case), which in principle is not specific for the library case and will be discussed in one of the following subsections.

3.2.3. Static Function Tests

Testing static functions in normal code is not really easy to do gracefully without interfering with the actual code to test. Since we don't want to change the deployment code for testing, we need to employ one of several possible workarounds here. We just include the C-source file in the test code:

```
---------------------------------------------------------------------------
#include "../src/hello.c"
---------------------------------------------------------------------------
```

Right, this construction is not the common use of the C pre-processor. But in our case, it works fine. We just need to make sure that the included C-file doesn't include another *main()* function and that other included files work well with this construction.

3.2.4. Stress Tests

As seen in the library test example above, the *autounit* library provides a means to do stress test suites instead of normal tests. The respective tests are just run n times, with n specified with *au_run_stress_suite()*.

3.2.5. Script Tests

Since our source code doesn't consist completely of C code, we can run test programs that are implemented as shell scripts and other languages. As defined in the *Makefile.am* above, the respective files just need to be listed in the *TESTS* variable. Then, they will be run automatically together with the other tests. When implementing, we just need to take into account that the program will be run on the embedded target.

The example package contains a script test called *test-test* which just writes *running on `hostname`...* to *stdout*.

3.3. Target Connection

As stated in the *Makefile.am*, our *TESTS_ENVIRONMENT* variable defines an SSH approach (IP networking) to target connection. The *TARGET_LOGIN* variable can be used (actually, practically be specified on the command line on *make check*) as a login on the target. To prevent *make check* from asking for a password on target login, we need to enable public key authentication on the target. A detailed step by step description is contained in the *README* file in the example tarball.

3.4. Running the tests

When everything is set up correctly, a *make check* run should result in something like this:

```
---------------------------------------------------------------------------
[...]
make  check-TESTS
make[2]: Entering directory `/home/stigge/electronica/hellotest/tests'
Test running on targetdev...
PASS: test-test
Function Tests
.....
OK 5 succeeded 10.655778 s (655778 us) total elapsed time
PASS: test-functions
Static Tests
..
OK 2 succeeded 2.210588 s (210588 us) total elapsed time
PASS: test-static
Library Tests
.....
OK 5 succeeded 0.006428 s (6428 us) total elapsed time
Stress Tests (100 iterations)
(1).....(6).....(11).....(16).....(21).....(26).....(31).....(36).....(41).
....(46).....(51).....(56).....(61).....(66).....(71).....(76).....(81)....
.(86).....(91).....(96).....
OK 5 succeeded 0.878457 s (878457 us) total elapsed time
PASS: test-library
==================
All 4 tests passed
==================
make[2]: Leaving directory `/home/stigge/electronica/hellotest/tests'
[...]
---------------------------------------------------------------------------
```

Important to mention here is the fact that the individual output from the test actually comes from the target (except the *PASS:* lines that are generated on the host). The statistics (*All 4 tests passed*) originates from the development host.

With an error in between, the output looks like:

```
---------------------------------------------------------------------------
[...]
make  check-TESTS
make[2]: Entering directory `/home/stigge/electronica/hellotest/tests'
Test running on targetdev...
PASS: test-test
Function Tests
...!.
FAIL 1 of 5 Function Tests 10.688908 s (688908 us) total elapsed time
```

```
test_functions_tty_init: elapsed time 4.248237 s
(248237 us):failed assertions (1 of 1):
test-functions.c:88:FAIL:dev_init was unsuccessful
FAIL: test-functions
Static Tests
..
OK 2 succeeded 2.219318 s (219318 us) total elapsed time
PASS: test-static
Library Tests
.....
OK 5 succeeded 0.006387 s (6387 us) total elapsed time
Stress Tests (100 iterations)
(1).....(6).....(11).....(16).....(21).....(26).....(31).....(36).....(41).
....(46).....(51).....(56).....(61).....(66).....(71).....(76).....(81)....
.(86).....(91).....(96).....
OK 5 succeeded 0.874148 s (874148 us) total elapsed time
PASS: test-library
===================================
1 of 4 tests failed
Please report to stigge@philosys.de
===================================
make[2]: *** [check-TESTS] Error 1
make[2]: Leaving directory `/home/stigge/electronica/hellotest/tests'
[...]
---------------------------------------------------------------------------
```

This output should be pretty self-explanatory, including output of our initial script test (*test-test*), the *test-functions.c*, *test-static.c* and *test-library.c* which are all included completely in our example package.

4. Conclusion

Throughout this presentation, we discussed the integration of unit tests in an embedded environment, using tools from the Open Source world and suggesting a specific approach to host-target connection, and the implementation of specific tests. Since every embedded project is quite individual, test harnesses and certain ideas will need to be modified. Creating useful test harnesses is actually one of the most difficult tasks in writing new tests. Further, we need to keep in mind that unit testing is not tied to a special implementation of a unit testing framework (like *autounit*) or a special implementation language, nor is Quality Assurance handled by just including unit testing in the development process.

One of the most important things with unit testing is: Doing it! Further, it is highly recommended to start integrating it early on in the development process. Unit testing can drastically help to make the development schedule more predictable because it can dramatically reduce debugging time, although we need to take into account the size of the necessary test code which can actually be even much bigger that the code under test.

-.-.-

About the author Roland Stigge:
Roland Stigge is a Software Engineer employed by Philosys Software in Unterschleissheim nearby Munich. He is a Debian developer and GNU maintainer, working on the most widely used Open Source operating systems since 2001 when he was working for Epigenomics, Inc. After graduating in computer science from Humboldt University in Berlin, he started working in the embedded business developing communication products with GNU/Linux operating systems. His main interests are quality assurance, Open Source software and Linux/UNIX operating systems.

About Philosys Software GmbH:
Philosys Software has been founded in 1988. Philosys is an IT-services provider with a clear focus on UNIX-/Linux systems and (technical) software development. Among our clients are well known brands, as there are BMW, Deutsche Telekom, Fujitsu-Siemens, Infineon, Siemens and Thomson. Philosys' offerings include project planning, software architecture and design con-sulting, system integration and software implementation or development. For further information please visit our website: http://www.philosys.de