

Unit testing in embedded environments with Parasoft C++test

■

Agenda

- Introduction
 - What are the embedded systems
 - What is Unit Testing
- Unit Testing With C++test
 - General flow
 - Test cases generation
- Unit Testing in Embedded Environments
 - General flow
 - Runtime requirements
 - Building test executable
 - Pros, Cons, limitations
- Questions



We make software work.

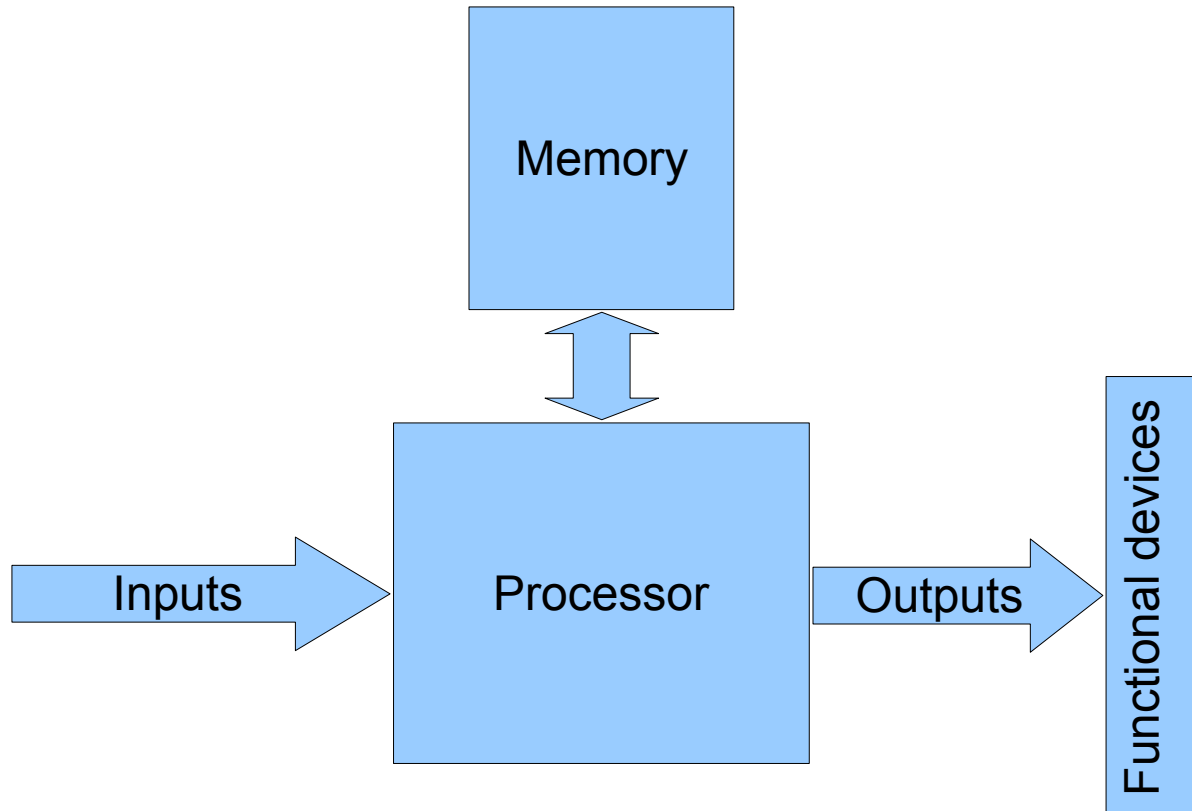
Introduction

What is an embedded system ?

An **embedded system** is a combination of computer hardware and the software, and perhaps additional mechanical or other parts, designed to perform a specific function.

(Michael Barr - Programming Embedded Systems)

A generic embedded system



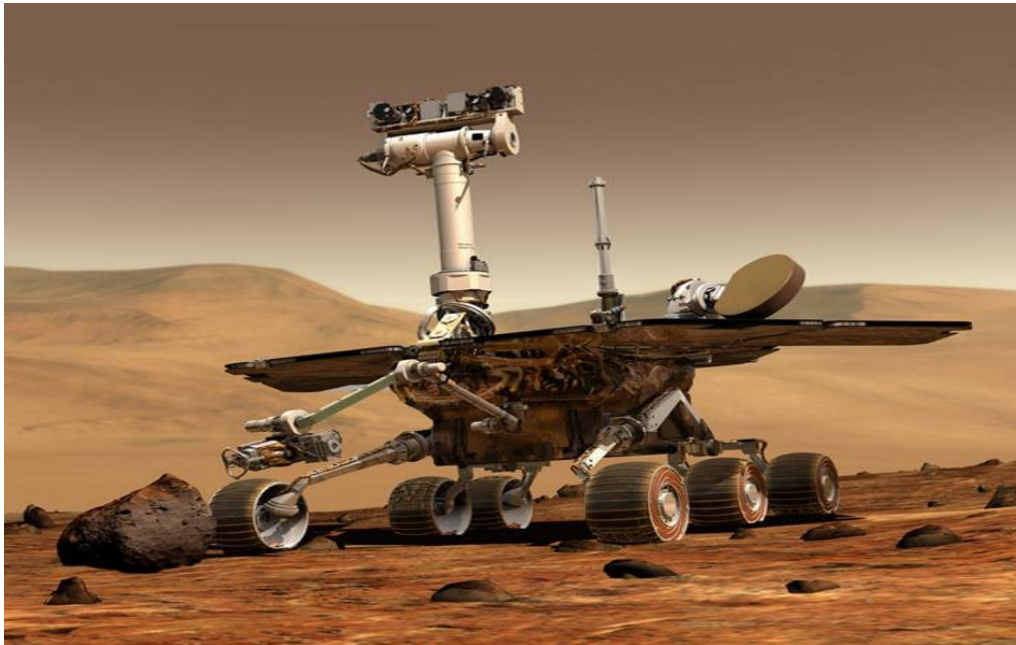
Example 1 - VCR



VCR can be made of:

- 8 bit processor
- ROM and RAM
- Mechanical components
- Analog modules for signal processing
- Platform targeted operating system

Example 2 - Opportunity



Opportunity:

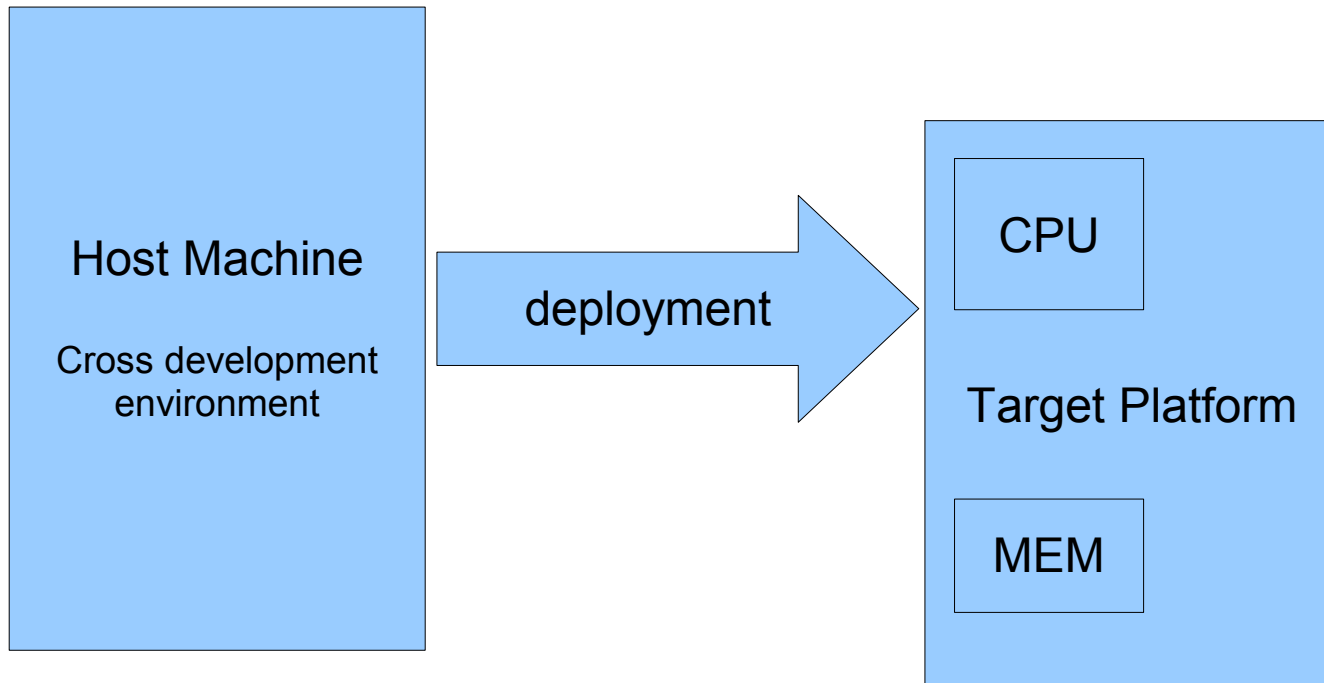
- CPU RAD6000 (RISC)
- 128 MB DRAM
- 3 MB EEPROM
- 256 MB FLASH
- Running on VxWORKS (WindRiver) operating system

Example 3 - Pendolino



Composed of many sub systems running on different hardware platforms and different operating systems

Development process



Unit Testing

In computer programming, a **unit test** is a procedure used to validate that a particular module of source code is working properly. The procedure is to write test cases for all functions and methods so that whenever a change causes a regression, it can be quickly identified and fixed.

(www.wikipedia.org)

Unit testing with C++test

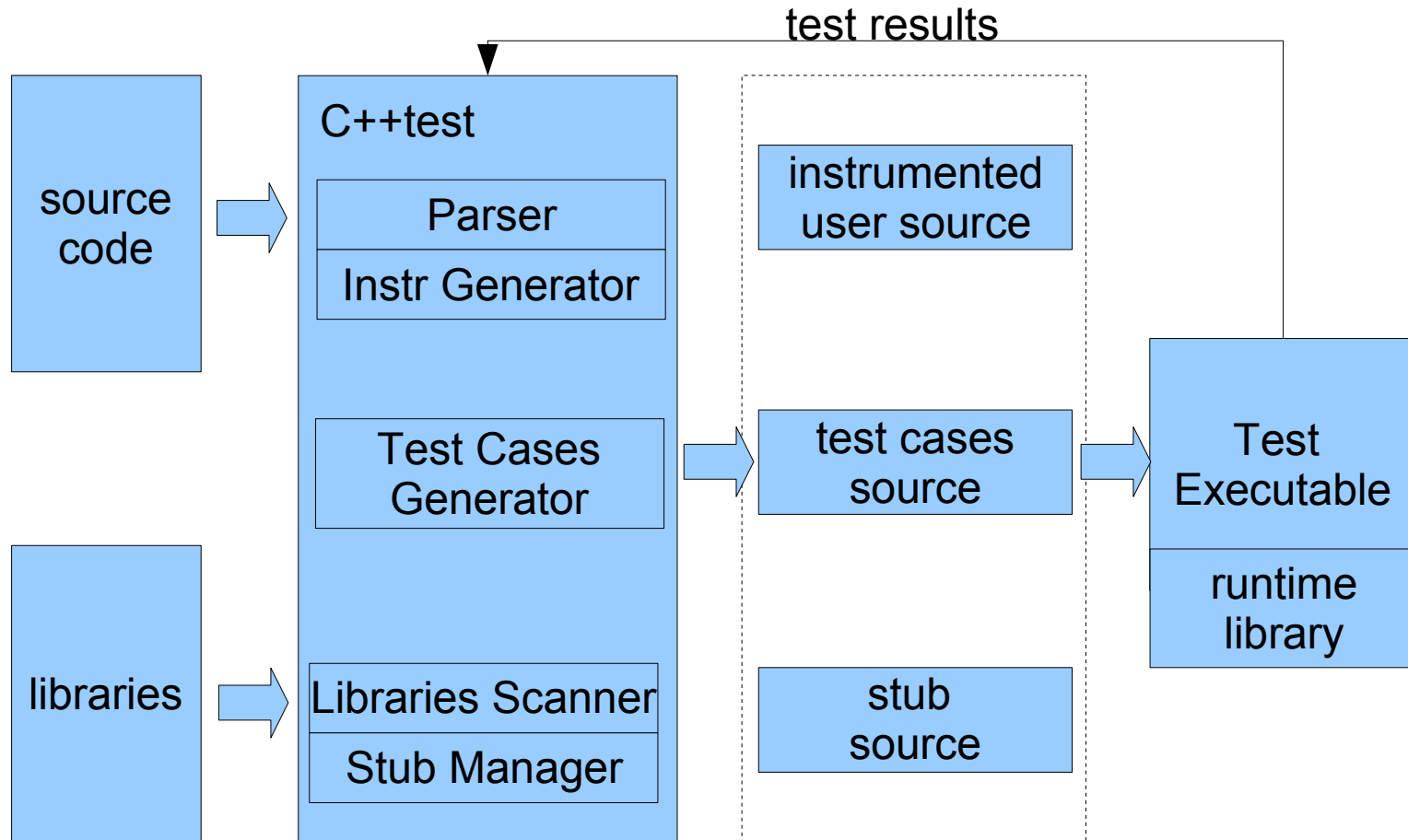
- test flow
- test harness architecture
- test cases
- regression testing

■

Unit Testing with C++test

- Automatic test cases generation
- Test cases in a form of source code
- Coverage metrics
- Post-Conditions recording for regression testing
- Automatic Stubs configuration
- Automatic test flow management
- Command line mode
- Convenient reporting system

From source code to test results

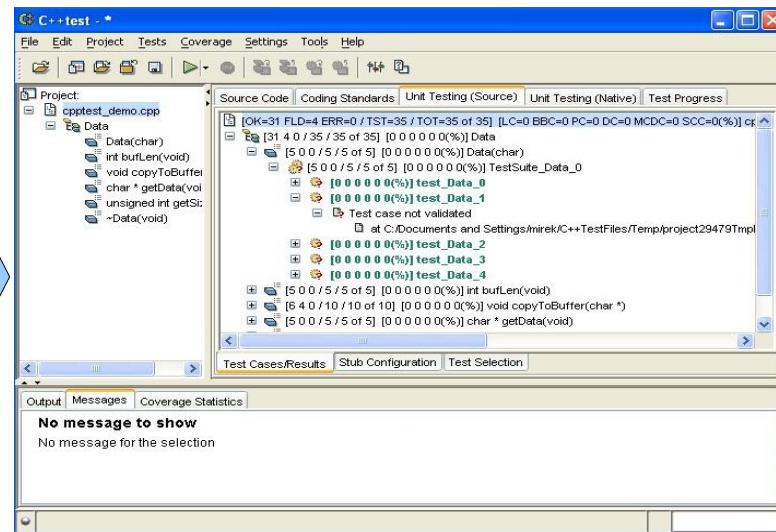


Tests architecture - example

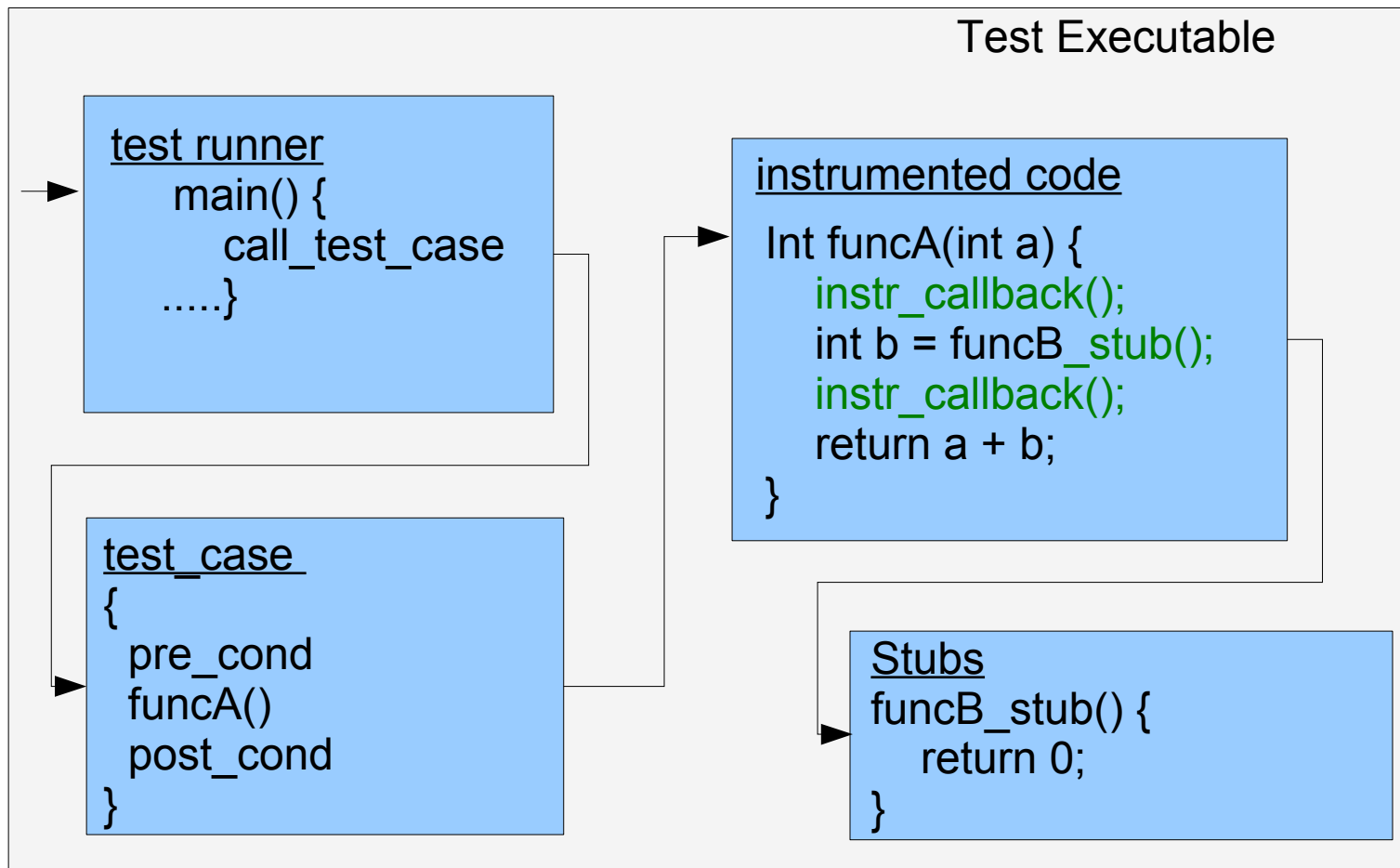
```
#include <header>

int funcB();

int funcA(int b)
{
    int a = funcB();
    return a + b;
}
```



Tests architecture - example



Test cases

Test cases architecture

```
TestSuite_MyClass::test_foo_0()
{
    //PreCondition initialization
    int arg_1 = 0;
    MyClass testObject();

    //Tested function call
    int ret = testObject.func(arg1);

    //PostConditions validation
    CPPTEST_ASSERT(ret == 1)
}
```

Test case is composed of three sections:

- Pre conditions setup
- Tested function call
- Post conditions validation

Automatic test cases generation

C++test's test cases generator automatically produces source elements for:

- Initializing pre-conditions like: function arguments, global variables, 'this' object for tested function
- Calling the tested function or method
- Validating post-conditions (achieved test result)

Initialization strategies

- Simple (built-in) types:
 - Integers (-1,0,1, min, max value)
 - Floating point (-1.0, 0.0, 1.0, max, min positive, max, min negative)
 - Heuristic values
- Complex types (user defined types) C++test will use available constructors and generate code necessary to initialize constructors arguments.

Initialization strategies

- Pointer types:
 - NULL pointer
 - address of a local object created on stack
 - address of an object created with "new" (on heap)
- Reference types:
 - local object created on stack
 - object created using operator "new"
- Heuristic values for simple types:
 - int
 - float
 - char *

Post conditions

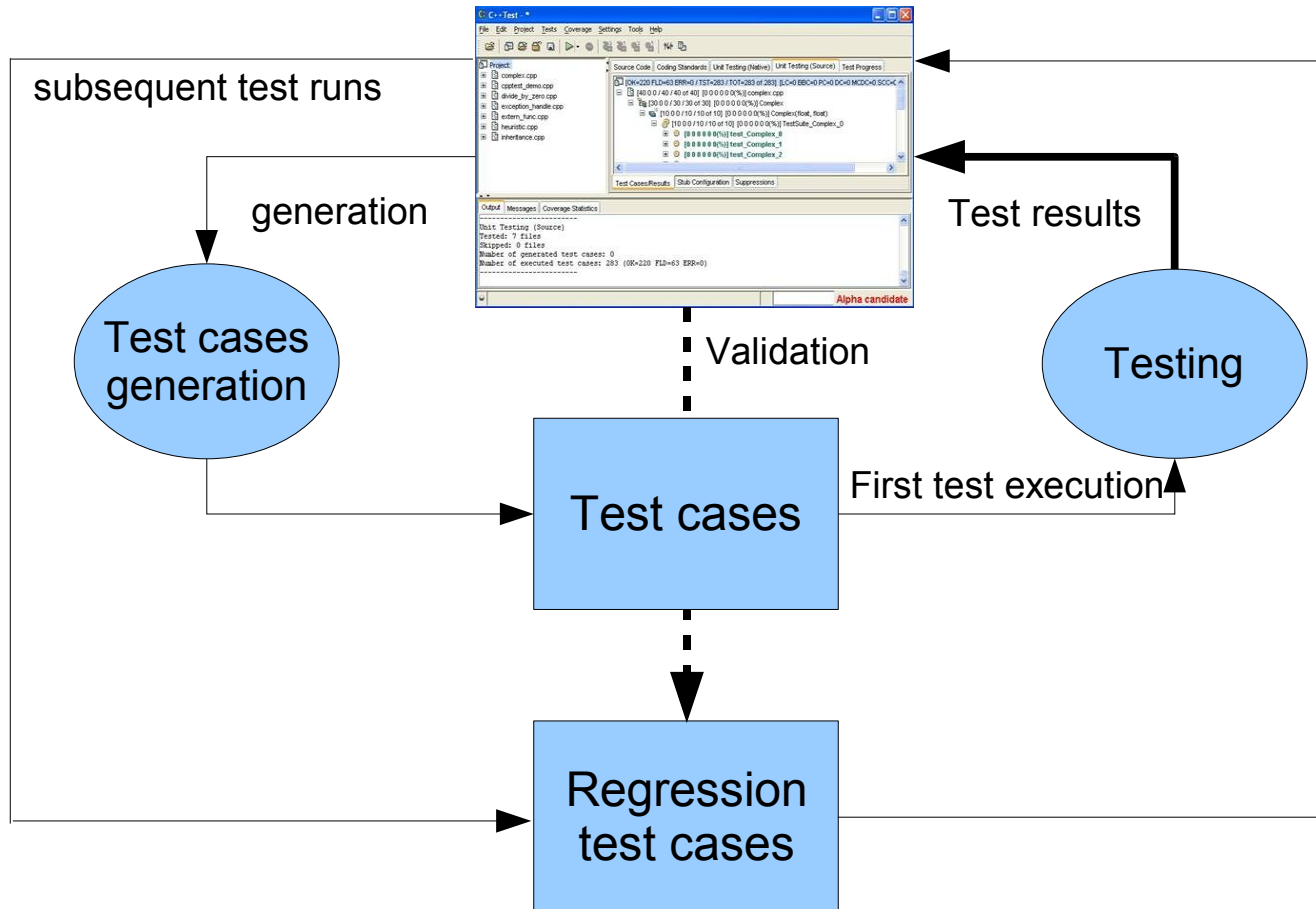
C++test provides macros and routines for controlling how source code test results are collected and validated:

- Test case post-conditions are controlled via assertion macros (similar to CppUnit)
 - `CPPTEST_ASSERT_INTEGER_EQUAL(expected,actual)`
- Assertions perform a condition validation and send results back to C++test

Test case example with post-conditions

```
/* CPPTEST_TEST_CASE_BEGIN test_bufLen_0 */  
void TestSuite_bufLen_0::test_bufLen_0()  
{  
    /* Pre-condition initialization */  
  
    /* Initializing argument 0 (this) */  
    /* Initializing constructor argument 1 (fill) */  
    char _fill_0 = '\001';  
    ::Data_cpptest_TestObject (_fill_0);  
  
    /* Tested function call */  
    int _return = _cpptest_TestObject.bufLen();  
  
    /* Post-condition check */  
    CPPTEST_ASSERT_INTEGER_EQUAL(4, ( _return ))  
    CPPTEST_ASSERT(( _cpptest_TestObject._data ) != 0)  
}
```

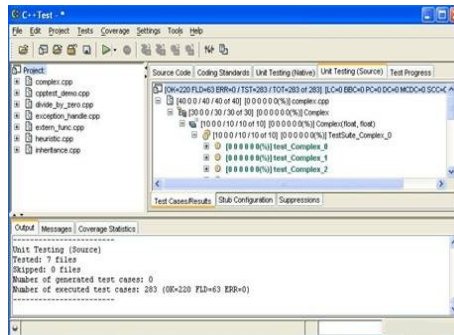
Regression test cases generation



Stubs

Stubs – handling missing definitions

Libraries
&
object files



Instrumented
source code

Auto generated
stubs

```
int func2()
{
    return 0;
}
```

User defined
stubs

```
extern int func2();
```

```
int func1(int b)
{
    int a = func2();
    return a + b;
}
```



Stubs in C++test

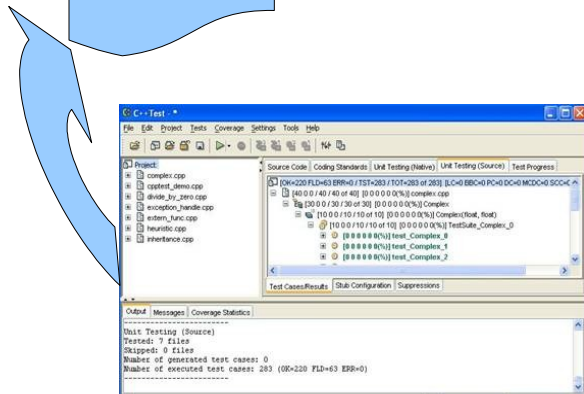
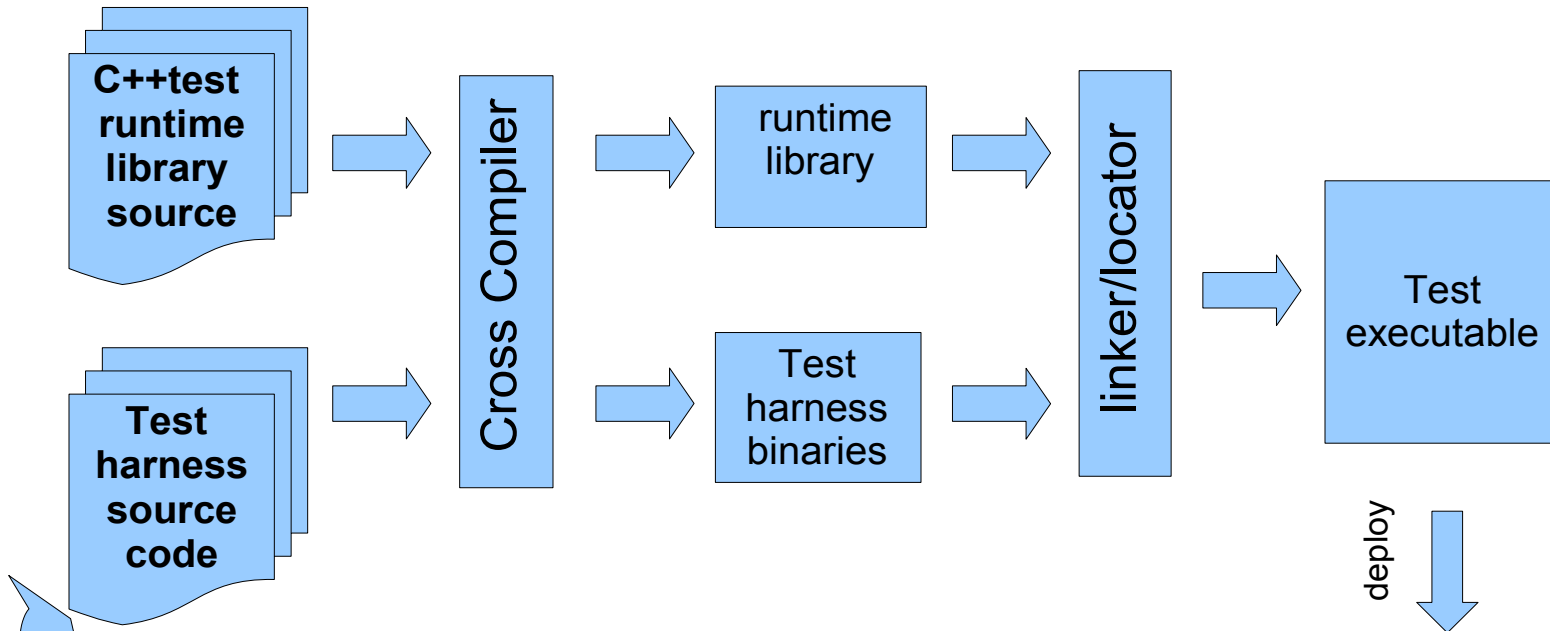
- Stubs are used:
 - when testing in separation from other modules
 - when application module is not yet ready
 - when hardware module is not yet ready
- C++test provides three modes for stubbed function:
 - original
 - user-defined
 - auto-generated
- Stubs configuration is managed from C++test GUI and persisted in C++test project

Unit Testing in embedded environments

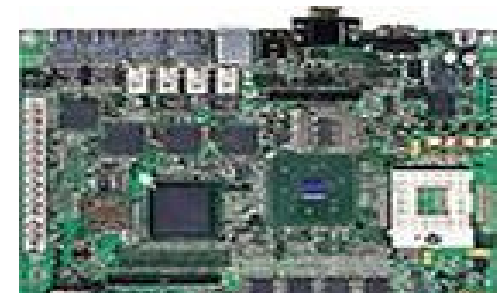
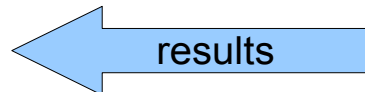
- test flow
- runtime library
- problems & limitations
- why to unit test on target ?

■

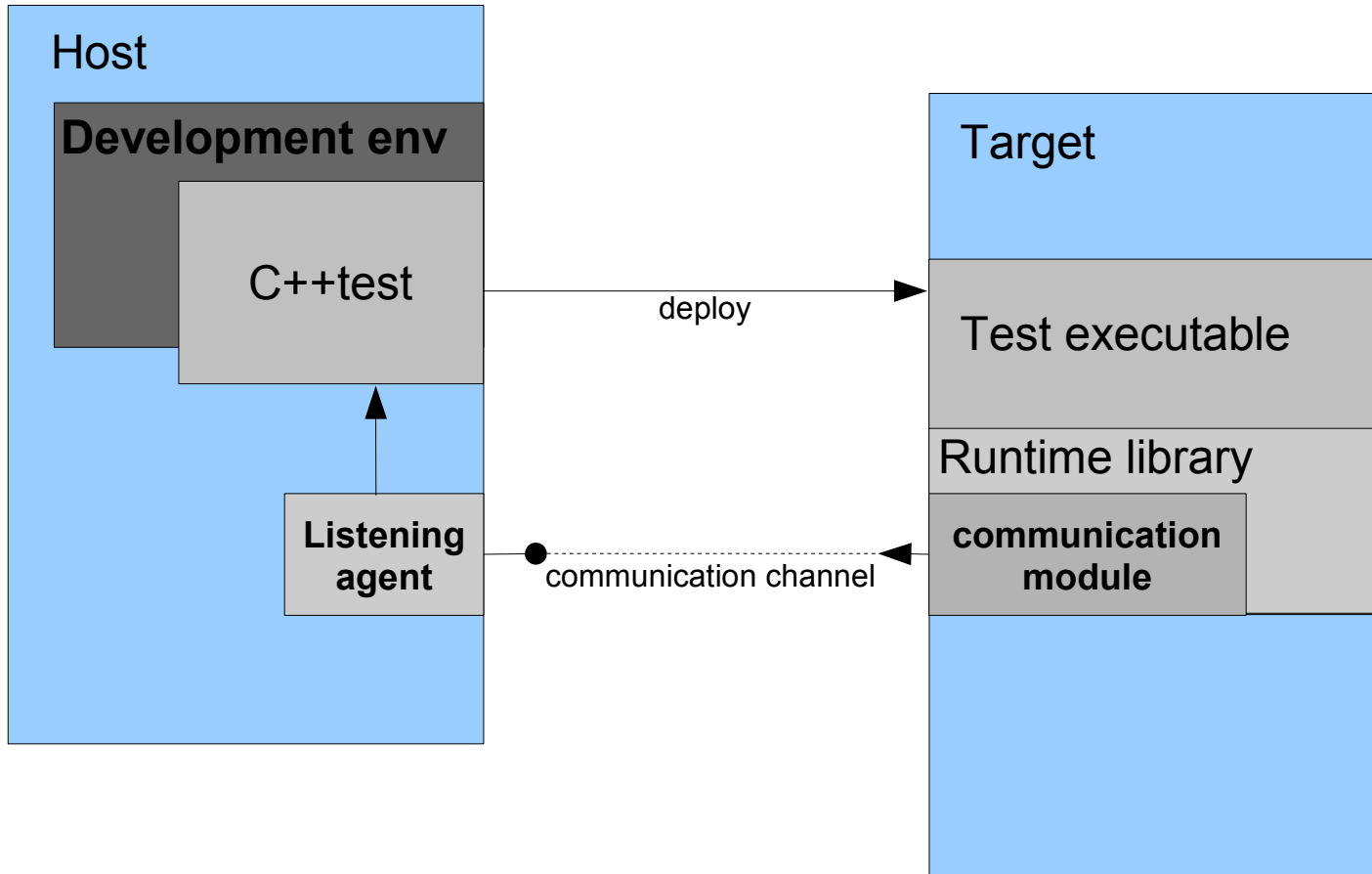
Unit testing in embedded environment



host - target communication



Unit testing in embedded environment





We make software work.

C++test runtime library

C++test runtime library

- Provided in form of source code
- Implemented in plain C
- Needs to be cross-compiled for specific target platform
- Contains definition of communication channel
Provided implementations:
 - file
 - TCP/IP sockets
- Designed to be easily customizable



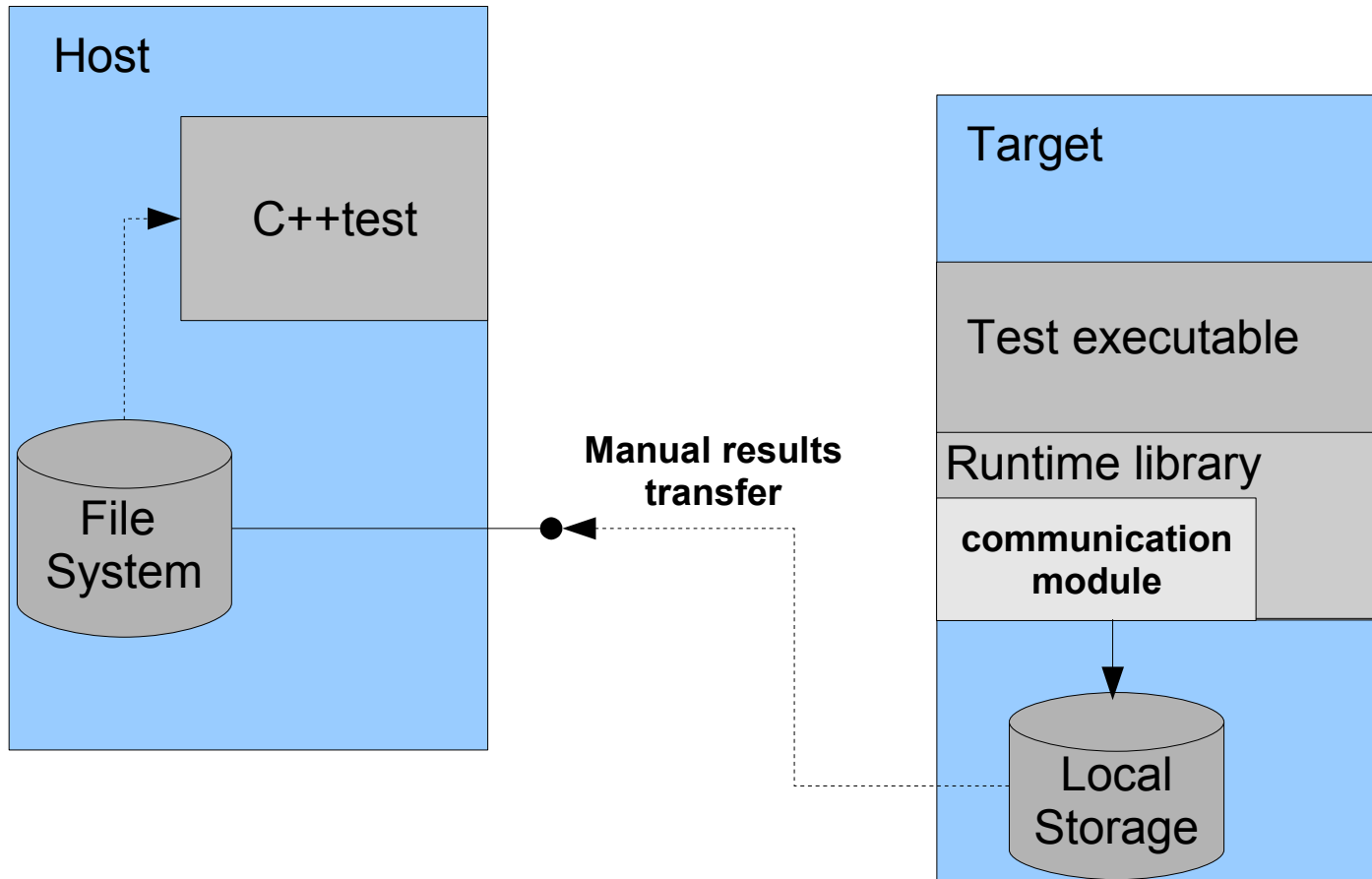
We make software work.

C++test runtime library Communication module

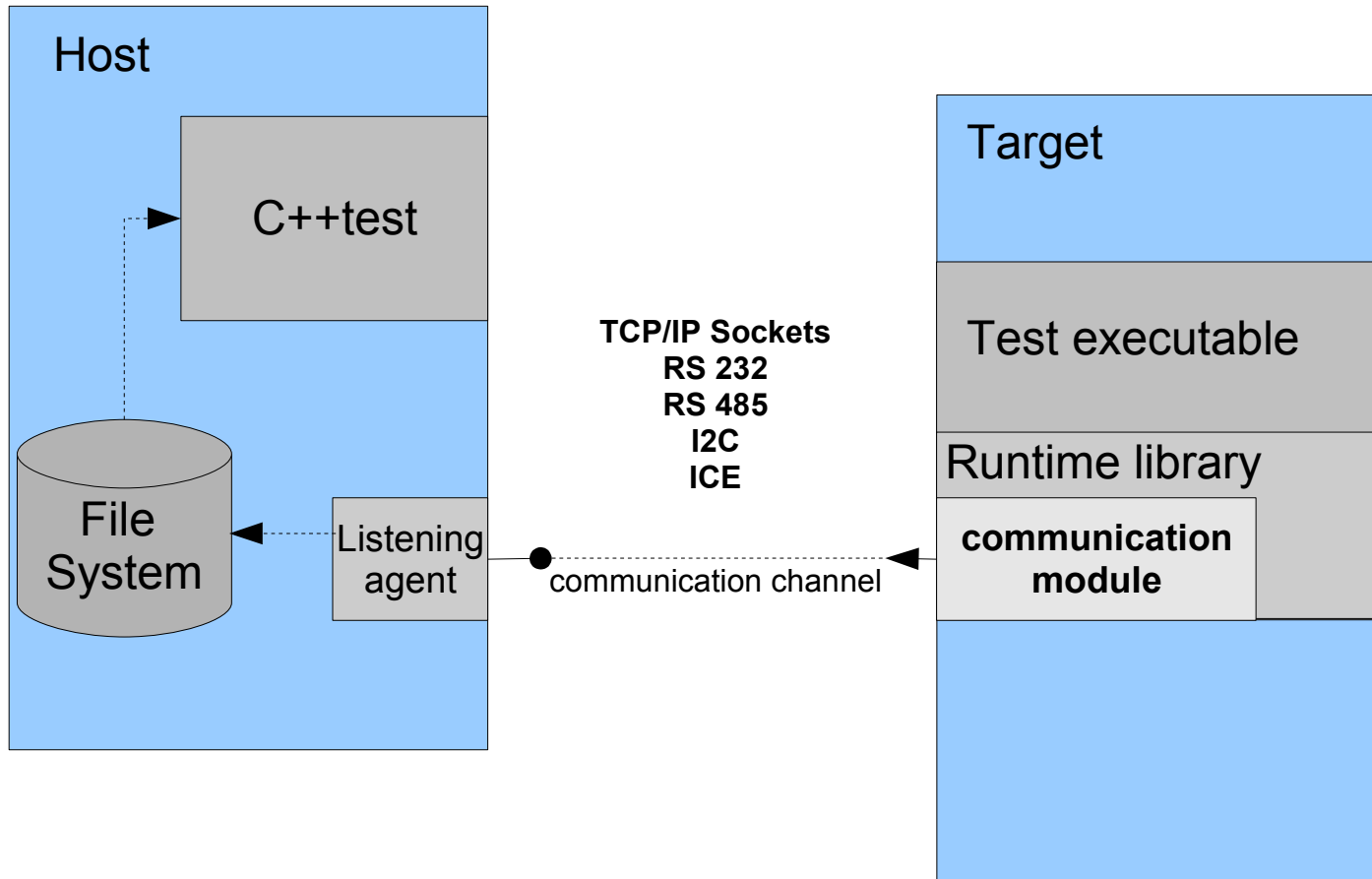
Target-to-host communication module

- Custom implementation of communication channel can be plugged in
- Defining communication channel requires implementing four methods:
 - `cpptestInitializeCommunication(...);`
 - `cpptestFinalizeCommunication(...);`
 - `cpptestSendData(const char* data, unsigned size);`
- All data transfer operations, performed inside C++-test runtime library are implemented in terms of above interface

Collecting results on target



“On the fly” results transfer

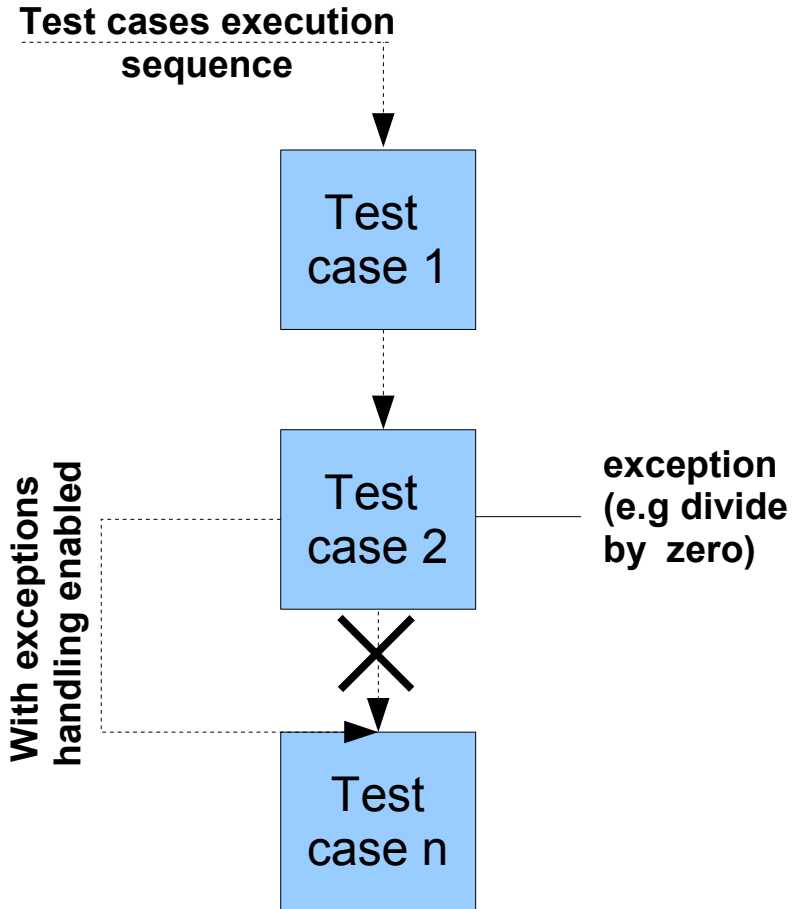




We make software work.

C++test runtime library Exceptions handling module

Exceptions handling



- C++test uses exceptions handling facilities (try, catch, setjmp, longjmp, signal) to recover if exception is thrown during test sequence
- If target platform does not provide exceptions handling, test sequence is broken on first exception and needs to be restarted



We make software work.

Test flow automation

Test flow automation

- Test sequence can be automatically performed in C++test GUI as well as in command line mode
- Test sequence consists of test harness preparation, building test executable, deploying test executable, starting it, and collecting results
- Limitations in automation of testing process can be imposed by development environment
- C++test provides easy to use XML-based format for defining test framework operations

Problems & Limitations

■

Problems & limitations

- Not enough memory on the target device to store test harness and tested code may block or limit testing capabilities

Compilation without debug information

	Original file	Min Instr	Med Instr	Max Instr
GNU GCC 3.2	22 KB	22 KB (0%)	29 KB (30%)	41 KB (80%)
MSVC++ 7.1	33 KB	33 KB (0%)	41 KB (24%)	69 KB (100%)
Tornado simpc	23 KB	23 KB (0%)	30 KB (30%)	41 KB (78%)

Problems & limitations

- Lack of communication channel may effect in weak test automation
- Missing support for exceptions handling may increase the number of runs necessary to execute test cases scheduled for test session.
- Additional amount of work required during development process



We make software work.

Why to unit test on target ?

■

Pros of unit testing on target

- All well known positives from host-based unit testing (early bugs detection, increased code robustness, ...)
- Possibility of detecting hardware problems
- Possibility of detecting software/hardware interaction problems
- Easy stubbing of missing software (also hardware) modules
- Quick and automated generation of regression suites

Unit testing in embedded environment - example

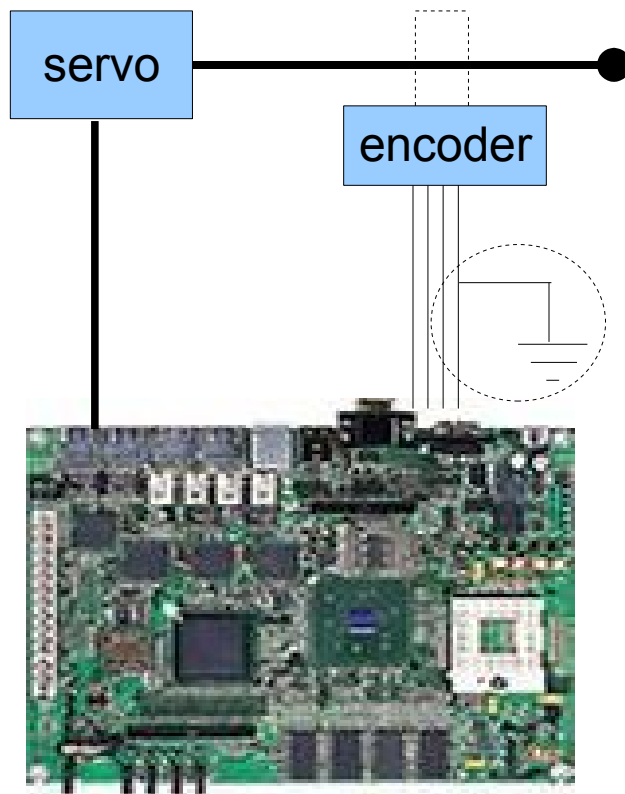
```
static int iTemperatures[2];
```

```
void interrupt vReadTemperatures(void) {  
    ITemperatures[0] = /*Read sensor 1*/  
    ITemperatures[1] = /*Read sensor 2*/  
}
```

```
bool testSensors() {  
    int iTemp0, iTemp1;  
    iTemp0 = ITemperatures[0];  
    iTemp1 = ITemperatures[1];  
    if (iTemp0 != iTemp1) { // Alarm !!!  
        return -1;  
    }  
    return 0;  
}
```

- “Critical section” related bugs are hard to detect during host-based unit testing
- Target-based unit testing highly increases probability of catching this kind of problems, however it does not provide 100% certainty of catching them.

Unit testing in embedded environment - example



- Module for driving the position of functional mechanical equipment
- Hardware contains “stuck-at-zero” problem between the encoder and board interface

```

int set_position(int angl)
{
    if (angl < MIN || angl > MAX)
    {
        return INCORRECT_ANGL;
    }
    return run_servo(angl);
}
    
```

Epilog

C++test offers a high level of automation for tasks which are usually performed manually, especially in embedded environments, therefore very often it lets us give the positive answer for a question:

“to test ? or not to test ?”