Test-Driven Development For

Embedded C++ Programmers

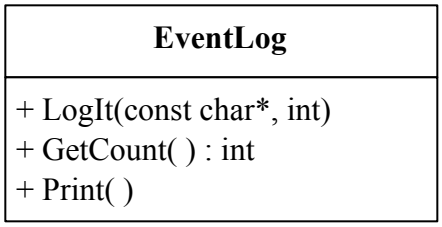
By James Grenning

Test-Driven Development is a technique for programming. Initially, it requires a lot of discipline. Over time it is addictive. TDD is a practice developed by Ward Cunningham and Kent Beck, and it is a key practice of Extreme Programming[BECK]. This technique can be used very effectively for developing embedded software. If you need your software to work and want to have high test coverage, you should try TDD.

Instead of talking a lot about TDD, I am going to show it to you TDD instead. I'll show you TDD in C++ and use CppUnitLite[OM] to write unit tests. I will also write code that conforms to the Embedded C++ draft standard to the best of my ability to understand the standard. If you are a C programmer, TDD can be used to develop C programs as well[KOSS], so please read on.

Our scenario is that we need an event logging class to use in our embedded application. The event logger will allow a string and an integer to be logged as a pair. Strings do not come from the heap, so I don't need to manage them. The logger will keep track of only the last N events, where N is determined at runtime. Once the logger fills up, the oldest log entry is lost. The log contents can be printed from a console port on our system. The log can also be queried to find out how many entries have been put into it. For this scenario, we won't worry about the log count reaching the capacity of an int.

These requirements imply a class that looks like this:



If this is the class you want, what would you do next? You might think about how to store the log entries. You might consider using a circular array of log entries. You're and engineer, you would be thinking of how to solve the problem. Once you had the solution in mind you would start coding the solution. Once you are done coding you would start to consider how to test your solution.

As the name implies, test-driven development has tests that drive the development of the code. Tests are a forethought, not an afterthought. This does not mean that all the tests are written, then all the target code. Rather, a single test is written, followed by writing

james@wingman-sw.com

the code needed to satisfy the failing test. This process continues until the code has enough functionality to be useful in the application.

The TDD cycle looks like this:

1. Think of some behavior that needs to be added
2. Write a test that expresses the missing behavior
3. Compile the test, watch it fail
4. Add what is necessary to get the code to compile
5. Compile the code and
6. Run the test, watch it fail
7. Add the code needed to make the test pass
8. Look at your work and decide if design improvements (Refactoring[FOWLER]) are needed
9. Repeat until needed behavior is implemented

This is a feedback loop. The loop is tight. It may only take a few minutes to complete a single cycle. A small amount of functionality is added each cycle, maybe only a few lines of code each time through. The programmer gets to find out if the program is behaving as they expect it to.

That's enough talking, let's create the EventLog class. First we have to think of an initial test. The current state of the system is that there is no EventLog class at all. Each application class will have a test class. The test file, by convention, is named EventLogTest. I am using a free unit testing tool called CppUnitLite. I will be running these tests on the development machine. It is a good idea to also run the tests on the target machine, but you just do not need to do it as often.

Here is the first test:

```
//EventLogTest.cpp
#include "TestHarness.h"
#include "EventLog.h"

TEST(EventLog, Create)
{
  EventLog* log = new EventLog();
```

Let me explain what we're looking at. TestHarness.h defines macros and classes needed to create tests and install the tests into the test harness. The TEST macro defines a test. The parameters to TEST are used to create the name of a test class. By convention, the first parameter is the name of the class being tested. The second parameter is the name of the test. Between the curly braces is the definition of the test. The names are used under the hood by CppUnitLite to name a test class.

james@wingman-sw.com

This does not seem like much of a test, but it is a start. I am working in small steps, getting feedback from the compiler about the code. This test confirms that the name EventLog is available. If this compiled without errors, maybe there is a name clash in your code base (or maybe your co-worker pulled an all-nighter wrote the code for you). Next I define the class.

```
//EventLog.h - excerpt

class EventLog
{
  public:
    EventLog();
    ~EventLog();

  private:
    EventLog(const EventLog&); //Hide the copy constructor
    EventLog& operator=(const EventLog&);//Hide assignment
};
```

This initial test and piece of code have a fair amount of overhead, so getting this to compile is a step not to be underestimated. Once this compiles we get a link error, at which point I add the constructor and destructor for the class in the cpp file. Notice I hid the copy constructor and assignment operator. I am preventing the compiler from generating those member functions. If I decide I need them, I'll write a test for them and probably write them my self.

When I run the test I get this output:

```
1 tests ran with 0 checks executed and no test failures
Detected memory leaks!
```

Wow, we ran a test that does nothing but create an object and CppUnitLite already found an error. I created an EventLog instance on the heap[i] and never deleted it. I'll fix the test, getting rid of the error.

Up to this point we have really only been getting ready to do some useful work. The Create test should check the initial state of the EventLog.

```
TEST(EventLog, Create)
{
  EventLog* log = new EventLog();
  LONGS_EQUAL(0, log->GetCount());
  delete log;
}
```

One initial condition I expect is that the log will be empty when I create it. I test that assertion using the LONGS_EQUAL[ii] macro. The first parameter is the expected value.

james@wingman-sw.com

The second parameter is checked for numeric equality with the first value. When this is compiled, an error is generated because there is no GetCount method. I'll add the method to the class initially with an implementation that fails, so that I am confident that the test is hooked into the test harness.

```
//EventLog.h - excerpt

class EventLog
{
  public:
    . . .
    int GetCount();
    . . .

};
```

```
//EventLog.cpp – excerpt

int EventLog::GetCount()
{
    return -1;
}
```

The resulting output is:

```
Failure: "expected 0 but was: -1"
        line 7 in C:\PROJECTS\EventLog\EventLogTest.cpp

1 tests ran with 1 checks executed and 1 failures
Detected memory leaks!
```

This result demonstrates that my new test is being called. It failed just as was expected. I also was notified of a memory leak. This memory leak is due to the early termination of the test. When LONGS_EQUAL macro fails the test is terminated and the `delete log` statement is never executed resulting in a memory leak.

Next I want to make this test pass. I make the test pass by adding a member variable to count the items logged, and initialize the count to zero in the constructor.

```
1 tests ran with 1 checks executed and no test failures
Press any key to continue
```

Let's log something. With this test we will evolve the logging interface and check that a single item is logged. Later, we will log multiple items and eventually log an item that causes the log to be over-filled, causing the oldest item to be discarded. I really need to print out the log to see what it contains. It is tempting to just implement EventLog and test it later, but I will resist that.

james@wingman-sw.com

How can progress be made without jumping into the implementation details? One thing I can do is to use GetCount to see if the log entry is made. That is an easy test. It's not thorough, but it is easy. Thorough will come. I write the test.

```
TEST(EventLog, LogOne)
{
  EventLog* log = new EventLog();
  log->LogIt("The answer is", 42);
  LONGS_EQUAL(1, log->GetCount());
  delete log;
}
```

To make this test pass I just have LogIt increment the counter and return. That is not the whole implementation, but it is enough to get the test to pass. The first time I run this test I comment out `logCount++` so that my test fails. Then the code is restored and the test passes.

```
//EventLog.cpp - excerpt

void EventLog::LogIt(const char* str, int value)
{
  logCount++;
}
```

This test needs to be improved. The way the problem is stated, there is a need to output the log. I'd rather not deal with printing the log yet. Without printing, there is no visibility into the internals of the EventLog. That is a good thing, because I'd like to encapsulate those details. But the hidden implementation is making it hard for me to devise the next test. To make life right now a little easier I'll add two accessor methods to the EventLog interface. The accessors will tell me what is stored in the log at some specific location.

As the user of the log I should not care how the internals are organized, but as the designer I am free to add methods to aid in testing. I'd prefer to test the log through the public interface, so I might remove these accessors later when I have a fully functional EventLog class.

james@wingman-sw.com

```
TEST(EventLog, LogOne)
{
  EventLog* log = new EventLog();
  log->LogIt("The answer is", 42);
  LONGS_EQUAL(1, log->GetCount());
  STRCMP_EQUAL("The answer is", log->getLogString(0));
  LONGS_EQUAL(42, log->getLogValue(0));

  delete log;
```

To get this test to pass I can rely on Kent Beck's advice to "Fake it till you make it"[BECK2]. In the following excerpt, LogIt does not really log anything, and the query methods return constant values. I'm rigging the logger to pass the tests. Again, I make sure my first run fails by first having GetLogValue return a 43, then *fixing* it to return a 42.

```
//EventLog.cpp – excerpt

int EventLog::GetLogValue(int index) const
{
    return 42;
}

const char*  EventLog::GetLogString(int index) const
{
    return "The answer is";
}
```

Run the test.

```
2 tests ran with 4 checks executed and no test failures
Press any key to continue
```

The faked out the implementation passes the test. This is worth doing to keep me moving forward at a steady pace. The fake-outs are cheap to write and the next test, log two events, will reveal their weakness.

james@wingman-sw.com

```
TEST(EventLog, LogTwo)
{
  EventLog* log = new EventLog();
  log->LogIt("The answer is", 42);
  log->LogIt("Elapsed time", 523);
  LONGS_EQUAL(2, log->GetCount());
  LONGS_EQUAL(42, log->GetLogValue(0));
  STRCMP_EQUAL("The answer is", log->GetLogString(0));
  LONGS_EQUAL(523, log->GetLogValue(1));
  STRCMP_EQUAL("Elapsed time", log->GetLogString(1));
  delete log;
}
```

Run the test and watch it fail.

```
Failure: "expected 1 but was: 2"
   line 26 in C:\PROJECTS\EventLog\EventLogTest.cpp

3 tests ran with 5 checks executed and 1 failures
Detected memory leaks!
```

How should the events be stored in the log? A simple thing to do is to create a data class that holds the string and the int and then make an array of them. I have to consider how big of a step this is. If adding this struct and array will take more than a few minutes, I should come up with an intermediate step. It seems pretty straight forward, but I think I'll start with a single instance of the LogEntry struct rather than an array.

I follow these steps:

1. Comment out the new test
2. Compile and test. Tests pass.
3. Add the LogEntry struct to EventLog.h
4. Compile
5. Add a LogEntry instance to class EventLog
6. Compile
7. Modify LogIt to store the string and value into the LogEntry
8. Compile
9. Modify GetLogValue and GetLogString to use the data stored in the log entry
10. Compile and test

That was more work than I thought. I'm glad I started with a single instance. Now we can un-comment the LogTwo test. The test fails. To make the tests pass I follow these steps:

1. Change the LogEntry instance to a pointer to a LogEntry. This will point to a LogEntry array.

james@wingman-sw.com

2. Compile, let the failures lead you to the invalid usages of the now Obsolete LogEntry instance variable. Fix LogIt, GetLogValue and GetLogString so they use the data stored in the log entry array.
3. Compile
4. Allocate an array in the constructor
5. How big should the log be? Make the log size a constructor parameter. Add the size parameter to the other EventLog creations.
6. Compile
7. Store the string and the value into the logCount array index
8. Compile and test

Ah! I get a memory leak failure. Oh, I forgot to delete the LogEntry array. Delete it in the destructor and now the tests pass.

```
3 tests ran with 9 checks executed and no test failures
Press any key to continue
```

There have been a lot of changes, so let's take a look at the current code.

```
//EventLog.h - excerpt
struct LogEntry
{
  const char* string;
  int value;
};

class EventLog
{
  public:
    EventLog(int capacity);
    virtual ~EventLog();

    int GetCount() const;
    void LogIt(const char* str, int value);

    //methods used for testing
    int GetLogValue(int index) const;
    const char* GetLogString(int index) const;

  private:

    int logCount;
    LogEntry* entries;
    EventLog(const EventLog&);
    EventLog& operator=(const EventLog&);
};
```


james@wingman-sw.com

```
//EventLog.cpp – excerpt

EventLog::EventLog(int capacity)
: logCount(0)
, entries(new LogEntry[capacity])
{
}

EventLog::~EventLog()
{
  delete [] entries;
}

int EventLog::GetCount() const
{
    return logCount;
}

void EventLog::LogIt(const char* str, int value)
{
  entries[logCount].string = str;
  entries[logCount].value = value;
  logCount++;
}

int EventLog::GetLogValue(int index) const
{
    return entries[index].value;
}

const char*  EventLog::GetLogString(int index) const
{
    return entries[index].string;
}
```

This EventLog implementation will work fine as long as we don't fill up the log. Finally the moment we've been waiting for. What do we do when we overfill the log? Our requirements are to discard the oldest log entry. A circular array would be an effective way to implement this. Our EventLog class has to pass this test:

james@wingman-sw.com

```
TEST(EventLog, LogWrapAround)
{
  EventLog* log = new EventLog(5);
  for (int i = 0; i < 5; i++)
    log->LogIt("The answer might be", 42 + i);

  log->LogIt("Overwrite 0", 199);
  LONGS_EQUAL(6, log->GetCount());
  LONGS_EQUAL(199, log->GetLogValue(0));
  STRCMP_EQUAL("Overwrite 0", log->GetLogString(0));
  LONGS_EQUAL(42, log->GetLogValue(1));
  delete log;
}
```

Running the tests we get the expected failure. Wrap-around has not been implemented yet, so 42 is still in slot zero.

```
Failure: "expected 199 but was: 42"
        line 43 in C:\PROJECTS\EventLog\EventLogTest.cpp

4 tests ran with 11 checks executed and 1 failures
Detected memory leaks!
```

To implement the wrap-around I need another index. I could reuse the logCount member variable, but my requirements call for being able to report the total number of items ever logged. I add a logIndex instance variable to the class definition and modify LogIt as follows.

```
//EventLog.cpp - excerpt

void EventLog::LogIt(const char* str, int value)
{
  logCount++;
  entries[logIndex].string = str;
  entries[logIndex].value = value;
  if (++logIndex < capacity)
    logIndex = 0;
}
```

Much to my surprise I get these results:

james@wingman-sw.com

```
Failure: "expected 43 but was: -842150451"
   line 45 in c:\projects\eventlog\eventlogtest.cpp

Failure: "expected 42 but was: 523"
   line 28 in c:\projects\eventlog\eventlogtest.cpp

4 tests ran with 10 checks executed and 2 failures
Detected memory leaks!
```

Like usual I make a small mistake with my conditional logic. I change the "<" to ">=" compile and test.

```
Failure: "expected 42 but was: 43"
   line 45 in c:\projects\eventlog\eventlogtest.cpp

4 tests ran with 13 checks executed and 1 failures
Detected memory leaks!
```

Wait a second, another failure! Look closely, this failure is due a cut and paste error in the `LogWrapAround` test. The value logged in slot one is 43 not 42. I change the 42 to a 43 compile and test and I am back in business.

Do tests have to be tested? That's an interesting question. In test driven development how do the test get tested? Part of the answer is that the tests are tested by the code (completeness testing is another issue). Working code found a bug in a broken test. Both these programming errors were very easy to find, mainly due to the fact that I just created the problem only seconds before. If I had not a day ago or a week ago it would be much harder to find the problems. Especially if the logger was integrated into some application that itself could have other problems.

The only thing left to do is to get the EventLog to support a print method. Testing this will be a bit tricky. If my requirement is to print to the screen, how can I automate the test for it? If I have to visually inspect the output this test will not be executed very often. A value of the tests that I have created so far is that they are installed into my test framework and whenever I change anything, I can re-run my tests, testing everything. A manual test won't get done often enough and defects can creep in.

So, a manual test is not an option. In C++ text output is done by writing to an output stream called cout. C++ and the embedded C++ standard support cout and a stream called a strstream[iii]. These two streams implement a common interface called ostream. Because they share the same interface I can code my print function to print to an ostream. At runtime cout is used and during testing strstream is used. The relationship between printf and sprintf is like the relationship between cout and strstream. cout writes to standard output, as does printf. strstream writes to a text buffer, as sprintf writes to a text buffer.

james@wingman-sw.com

Let's start simple. What will the print look like for an empty log? The output should show that there are no items logged, and print a separator. Here is the test.

```
//EventLogTest.cpp - excerpt
TEST(EventLog, LogPrintEmptyLog)
{
  EventLog* log = new EventLog(5);

  strstream os;
  log->Print(os);
  os << '\0';  //terminate the stored string

  char * expected =
    "EventLog Items logged 0\n"\
    "----\n";

  char* output = os.str();
  STRCMP_EQUAL(expected, output);

  delete output;
  delete log;
```

This test creates a strstream[iv] to capture the output. It calls Print, passing in the stream. The character string is copied from the stream and compared to the expected results. When I first wrote this test I thought I could use os.str() directly, but it turned out it is not null terminated producing a test failure. It took a bit of experimenting to get this test to pass. Additionally the os.str() operation returns a heap object that has to be freed. The memory leak detector found this one.

This code passes this first test:

```
//EventLog.cpp - excerpt, initial implementation

void EventLog::Print(ostream& os) const
{
    os << "EventLog Items logged 0\n";
    os << "----\n";
}
```

I have a test strategy, an intial test and an initial implementation. Now it is time to print a log that contains something. Fill the log and print it. Don't over-fill the log yet.

james@wingman-sw.com

```
//EventLogTest.cpp - excerpt

TEST(EventLog, LogPrintFullLog)
{
  EventLog* log = new EventLog(5);

  for (int i = 0; i < 5; i++)
    log->LogIt("Fill it up", 100+i);

  strstream os;
  log->Print(os);
  os << '\0';  //terminate the stored string

  char * expected =
    "EventLog Items logged 5\n"\
    "1 Fill it up 100\n"\
    "2 Fill it up 101\n"\
    "3 Fill it up 102\n"\
    "4 Fill it up 103\n"\
    "5 Fill it up 104\n"\
    "----\n";

  char* output = os.str();

  STRCMP_EQUAL(expected, output);

  delete output;
  delete log;
```

Here is the initial implementation:

```
//EventLog.cpp - excerpt

void EventLog::Print(ostream& os) const
{
    os << "EventLog Items logged " << logCount << "\n";
    for (int i = 0; i < capacity; i++)
      os << i + 1 << " "
         << entries[i].string << " "
         << entries[i].value << "\n";
    os << "----\n";
}
```

When I run the tests this crashes. I introduced another simple mistake. When the log is not full, using the capacity member variable as the loop limit causes access to uninitialized log entries. Accessing uninitialized data results in undefined behavior (its defined in this case, the system crashes). For now I just change from capacity to logCount. This is not the final version, but it makes this test pass.

james@wingman-sw.com

The following test will over-fill the log. We should witness that the oldest log entry is discarded and the entries are listed from oldest to newest.

```
//EventLogTest.cpp – excerpt

TEST(EventLog, LogPrintOverFullLog)
{
  EventLog* log = new EventLog(2);

  log->LogIt("This should not print at all", 100);
  log->LogIt("This should print first", 101);
  log->LogIt("This should print last", 999);

  strstream os;
  log->Print(os);
  os << '\0';  //terminate the stored string

  char * expected =
    "EventLog Items logged 3\n"\
    "2 This should print first 101\n"\
    "3 This should print last 999\n"\
    "----\n";

  char* output = os.str();
  STRCMP_EQUAL(expected, output);

  delete output;
  delete log;
```

This test also crashes because of the naive implementation of Print's loop. The log is to be printed from oldest to newest. It's time to think for a minute. If the log is full, logIndex refers to the oldest entry in the log. If the log is not full, slot zero is the oldest log entry. The number of items to print for a full log is the log's capacity. To print a log that is not full, we print up logCount entries. That sounds reasonable enough.

After a few tries, I got this code to pass the test. I am embarrassed to list all the ways I messed this code up. Eventually the tests pass.

james@wingman-sw.com

```
//EventLog.cpp – excerpt

void EventLog::Print(ostream& os) const
{
    os << "EventLog Items logged " << logCount << "\n";
    int index = 0;
    int loopCount = logCount;
    int oldestLogCount = 1;

    if (logCount >= capacity)
    {
      index = logIndex;
      loopCount = capacity;
      oldestLogCount = logCount - capacity + 1;
    }

    for (int i = 0; i < loopCount; i++)
    {

      os << oldestLogCount + i << " "
         << entries[index].string << " "
         << entries[index].value << "\n";

      if (++index >= capacity)
        index = 0;
    }

    os << "----\n";
```

This code works but I am not very happy with it. There is a lot going on in this function: printing the header and footer, looping through and printing all items. This method certainly does not a have single responsibility. Now that the code works and is fully tested I'll refactor it into a better design. (Earlier in the paper I said we might want to remove the public accessor methods added to help in testing. I don't see that they are doing any harm, so I'll leave them.)

Code After Refactoring

During Refactoring I extracted helper functions for the Print method. After each code modification, the tests demonstrated that the behavior of the program did not change.

james@wingman-sw.com

```
//EventLog.h - excerpt

class EventLog
{
  public:
    EventLog(int capacity);

    virtual ~EventLog();

    int GetCount() const;
    void LogIt(const char* str, int value);
    void Print(ostream& output) const;

    //methods used for testing
    int GetLogValue(int index) const;
    const char* GetLogString(int index) const;

  private:

    int logCount;
    int logIndex;
    int capacity;
    LogEntry* entries;

    void PrintHeader(ostream& output) const;
    void PrintFooter(ostream& output) const;
    void PrintLogItems(ostream& output) const;
    void PrintLogItem(ostream& output,
                int logNumber, int index) const;
    int NextIndex(int index) const;

    EventLog(const EventLog&);
    EventLog& operator=(const EventLog&);
```


james@wingman-sw.com

```
//EventLog.cpp - excerpt part 1
EventLog::EventLog(int theCapacity)
: logCount(0)
, logIndex(0)
, capacity(theCapacity)
, entries(new LogEntry[theCapacity])
{
}

EventLog::~EventLog()
{
  delete [] entries;
}

int EventLog::GetCount() const
{
    return logCount;
}

void EventLog::LogIt(const char* str, int value)
{
  logCount++;
  entries[logIndex].string = str;
  entries[logIndex].value = value;
  logIndex = NextIndex(logIndex);;
}

int EventLog::GetLogValue(int index) const
{
    return  entries[index].value;
}

const char*  EventLog::GetLogString(int index) const
{
    return entries[index].string;
}

int EventLog::NextIndex(int index) const
{
    if (++index >= capacity)
      index = 0;
    return index;
}
```


james@wingman-sw.com

```
//EventLog.cpp - excerpt part 2
void EventLog::Print(ostream& os) const
{
    PrintHeader(os);
    PrintLogItems(os);
    PrintFooter(os);
}

void EventLog::PrintHeader(ostream& os) const
{
    os << "EventLog Items logged " << logCount << "\n";
}

void EventLog::PrintFooter(ostream& os) const
{
    os << "----\n";
}

void EventLog::PrintLogItems(ostream& os) const
{
    int index = 0;
    int loopCount = logCount;
    int oldestLogCount = 1;

    if (logCount >= capacity)
    {
      index = logIndex;
      loopCount = capacity;
      oldestLogCount = logCount - capacity + 1;
    }

    for (int i = 0; i < loopCount; i++)
    {

      PrintLogItem(os, oldestLogCount+i, index);

      index = NextIndex(index);
    }
}

void EventLog::PrintLogItem(ostream& os, int logNumber, int
index) const
{
    os << logNumber << " "
       << entries[index].string << " "
       << entries[index].value << "\n";

}
```


james@wingman-sw.com

Conclusions

TDD is a predictable process. Code now, debug later programming is risky and not as predictable. Code written yesterday, last week, or last month is much harder to debug than the code that was written one minute ago. The tests have to be maintained, but they keep giving back.

I find that small things slip by often. The tests really give me the feedback to see if what I have coded actually works. Small mistakes can cause big problems when left to lay in wait.

The tests we developed serve a few purposes: to fully unit test the code, to document how to use the code, to document what the code does, and to act as a safety net when making changes. The focus on testing makes the designer pay attention first to the interface and external behavior. Implementation details are secondary. The technique encourages decoupling of software modules. Notice in this design how the print function is loosely coupled to a specific way to print. For our print function to print to standard output all we have to do is call it passing the cout standard output stream like this.

```
log->Print(cout);
```

A loosely coupled software system is easier to evolve. Automated tests make a system easier to evolve. Changes are made and the tests tell us if there are unanticipated side effects.

This technique can be effectively used for developing embedded software. It means that you have to invest time in writing code that manages the dependencies directed to the target execution environment. Ideally the modules can be decoupled from the target allowing tests like I developed in this paper to be run on your development and execution environments. This takes time and is worth it. It is paid for many times over in reduced debug time.

The EventLog class is a terminal node class in the dependency hierarchy. TDD can be used for developing classes that are in the midst of the dependency network. But that's a topic for another paper (coming soon).

[BECK] Beck, Kent, Extreme Programming Explained, Addison Wesley, 1999

[OM] CppUnitLite - http://www.objectmentor.com/resources/downloads/index

[KOSS] Koss, Dr. Robert, Langr, Jeff Test Driven Development in C/C++, C/C++ Users Journal, October 2002, http://www.cuj.com/articles/2002/0210/0210a/0210a.htm?topic=articles

[FOWLER] Fowler, Martin, Refactoring Improving the Design of Existing Code, Reading, MA, Addison Wesley,1999

james@wingman-sw.com

[i] Some of you embedded engineers may be cringing at using the heap. This is a test. It is OK here even if it is not OK in your final system.

[ii] There are other macros such as: CHECK(bool), CHECK_EQUAL, STRCMP_EQUAL DOUBLES_EQUAL, FAIL

[BECK2] Beck, Kent, Test Driven Development By Example, Addison Wesley, 2003

(Source code from this paper is available at www.objectmentor.com/resources/articles/EventLog.zip)

[iii] It is a bit unclear if strstream is supported in embedded C++. The Dinkumware implementation implies that it is. Green hills implies that a similar capability using sstream is available;

[iv] It is not completely clear that the Embedded C++ draft standard supports strstream. It appears that mainstream cross compilers, such as the Dinkumware compiler, do.

james@wingman-sw.com