A Flat Chart Technique for Embedded OS Testing[1]

V.V. Nikiforov<nik@ iias.spb.su>
S.N. Baranov<snbaranov@ iias.spb.su>
St. Petersburg Institute for Informatics and Automation
of the Russian Academy of Sciences,
39, 14 liniya, St. Petersburg, 199178, Russia

Abstract. Modern automatic devices are more and more equipped with microcontroller units. The logic of work of the automatic equipment is supported by a number of various embedded software applications, which run under an embedded real-time operating system (OS). The OS reliability is extremely important for correct functionality of the whole automatic system. Therefore, the embedded OS should be tested thoroughly with an appropriate automated test suite. Such test suite for testing of an embedded OS is usually organized as a set of multi-task test applications to be executed in a data-driven manner. The paper features a special language to define the respective testing task logic and the concept of flat charts to efficiently perform an embedded OS execution-based testing. To avoid heavy interpreting of text strings during the test run, the respective test presentation is pre-processed in order to convert the initial string form into a regular array form and thus to increase its efficiency.

Keywords: Embedded Applications; Operating Systems; Software Testing; Real-Time Systems.

DOI:10.15514/ISPRAS-2017-29(5)-5

For citation: Nikiforov V.V., Baranov S.N. A Flat Chart Technique for Embedded OS Testing. Trudy ISP RAN/Proc. ISP RAS, vol. 29, issue 5, 2017, pp. 75-92. DOI: 10.15514/ISPRAS-2017-29(5)-5

1. Introduction

Software applications, which control various automatic devices, are usually built as a set of two kinds of sequential executing threads: *tasks* and *interrupt service routines* (*ISRs*). Coordination of execution of these threads is realized by the kernel of the embedded real-time operating system (OS). OS reliability is extremely important for correct functioning of the automatic technical device under software

[1] This paper is an extended version of a presentation at the Industrial track poster session of the 29th IFIP International Conference on Testing Software and Systems (ICTSS-2017), St. Petersburg, Russia, October 9-11, 2017.

control.

The variety of requirements for such an OS grows along with the variety of technical devices, for which embedded systems are designed, especially for devices built on the basis of *microcontroller units* (MCUs). Each OS for an MCU should be tested thoroughly to avoid a crash of an embedded application. Verification of embedded real-time software is a well-known problem [1], [2]. Thorough execution-based testing [3] of an embedded OS requires significant effort along two axes: full-bodied *test suite design* and *test suite execution*.

Effort reduction for test execution may be achieved by designing a highly automated test suite. Effort reduction for design of such a test suite may be achieved through efficient testing techniques, languages, and tools.

The paper describes a special language to define the testing task logic based on the concept of flat charts to efficiently run embedded OS execution-based testing.

2. Approach to Testing an Embedded OS

Usually, an embedded OS provides static and dynamic services for applications to run on top of this OS. *Static services* are used to specify static configuration features of the application: the set of its tasks and ISRs, the subset of the used OS functions, basic task properties (e.g., task priorities), static resource distribution among the application tasks (allocation of memory, stacks and other special structures). *Dynamic services* may be further split into basic and additional ones.

Basic dynamic services ensure:

- run-time distribution of resources among the threads (memory, special structures, processor time);
- exchange of data and signals among tasks;
- passing data and signals from ISRs to tasks;
- error (fault, exception) handling which provides data on an abnormal situation in the application.

Additional dynamic services support specific functions:

- run-time generation of threads, tasks, and ISRs;
- run-time updating of the basic task properties (e.g., the task priority);
- run-time stack reallocation;
- mathematical calculations, string processing, etc.

The problem of basic dynamic services testing will be considered in this paper from two points of view:

- functional testing – checking the correctness of the basic OS directives execution logic; and
- timing testing – measurement of time intervals required for execution of basic OS directives.

Functional testing is aimed at checking the correctness of the OS behavior through

finding defects in:

- execution of basic OS directives invoked from application tasks and ISRs;
- processor switching among threads;
- data and signal transactions;
- error handling routines.

Timing testing is aimed at obtaining the following timing data on OS execution:

- execution time of a particular OS directive (local time measurement);
- total execution time of the whole application (global time measurement);
- time interval between the moments when the interrupt occurred and when a respective ISR started this interrupt processing (latency measurement).

The described flat chart technique is aimed at both kinds of testing of embedded OS basic dynamic services, functional and timing, through a unified approach.

2.1 Testing Rules

The following generally established testing rules [4], [5] are usually observed for embedded OS testing:

- *focus* – each test should check only one OS feature under particular conditions with only two possible outcomes: pass or fail;
- *repeatability* – the test behavior should be the same at each execution;
- *non-interference* – the test should not intrude into OS functioning (no direct access to OS variables, command lines, or structures), the test uses the OS services as a regular application;
- *black-box approach* – each test should be developed with no knowledge or assumptions about the OS inner structures, with information at the user's level only.

The above rules for focus and repeatability impose structural constraints for tests because with these rules each single test should be a multi-task application which starts from a known inital state. The most reliable way to bring the system under test into this state is system restart with re-initialization. Therefore, the size of each test for an embedded OS is that of a multi-task application, and the test execution time includes the time required for system initialization.

The repeatability rule requires special solutions to ensure it. Regular real-time applications running under an embedded OS usually lack repeatability: their tasks and ISRs work *asynchronously* without any pre-defined order. Test applications should be built in such a way as to avoid such indetermination.

2.2 Repeatability of Testing

The test scheme in Fig. 1 shows how variations of the test behavior may occur. The test *DelayCoEnd* below is related to the most basic service of an embedded OS – the *delay service.* The operator *Delay*(*N*) holds up the task execution for *N* ticks where

'tick' is an atomic time interval, usually part of a millisecond. The test *DelayCoEnd* checks the correctness of OS behavior when delay intervals of two tasks come to a completion simultaneously. The scheme uses a C-like notation. A digit in the name of the task starting point corresponds to the task priority. The task that starts at the point *Task_1* has higher priority than the task labeled *Task_2*.

```
// -----  "DelayCoEnd" test application  ------
  Task_1:                          |      Task_2:
/* Step_01 */   Delay(50);         | /* Step_02 */ for(i = 0;  i == WAIT_CONST;  i++ );
/* Step_05 */   GlobFlag = 1;     | /* Step_03 */       Delay(30);
/* Step_06 */   TaskEnd( );       | /* Step_07 */       GlobFlag = 1;
                                   | /* Step_08 */     TaskEnd( );
  Task_3:
/* Step_04 */   for(GlobFlag = 0; GlobFlag == 0 ; );
/* Step_09 */   End_of_Test( );
```

Fig. 1. Variations of the test behavior

Step numbers shown in comments indicate the expected order of execution. At *Step_01* execution of *Task_1* is held up for 50 tics, the processor switches to *Task_2*, and the 'for'-operator of *Task_2* (*Step_02*) starts. The value of *WAIT_CONST* should ensure a simultaneous completion of the two delay intervals. While both intervals have not been completed, the 'for'-operator of *Task_3* (*Step_04*) is executed. The idea of the whole scheme is in selection of a *WAIT_CONST* value which ensures simultaneous completion of both delay intervals, so that if OS correctly handles this, then the actual sequence of steps follows that of the step numbers (additional steps may appear between them).

For automatic registration of the sequence of executed steps, the *Trace*(*i*) operator should be substituted for each comment *Step_i*. The procedure *Trace*(*i*) checks whether its parameter *i* corresponds to the current step in the expected step sequence and signals an error otherwise. This trace operator should be inserted at each point where the execution sequence should be checked.

At looking at these three tasks, one may decide that the only issue for checking the correctness of the OS delay function is an appropriate selection of the *WAIT_CONST* value which ensures simultaneous completion of the two delay intervals and therefore, *DelayCoEnd* repeatability. However, this is not true at a closer consideration.

Non-repeatability of such test execution is caused by the fact, that *Step_01* may start either at the beginning of an atomic tic interval or closer to its end and the required value of *WAIT_CONST* is different for these two situations. To make the test consistently correct, the operator *Step_01* should be shifted to the end of an atomic tic by inserting an additional delay operator before *Step_01*.

The requirement for repeatability is specific for a test application, which differs

from the real one with asynchronous execution of application tasks where the order of operators from parallel tasks may vary from one execution to another.

Test applications require special efforts for strict task synchronization. As a result, the sequence of operations from parallel tasks in test execution becomes strictly determined as if it were from a sequential process.

In the listing in Fig. 2 the *flag synchronization* method with *WaitFlag*( ) and *SetFlag*() procedures ensures test repeatability:

```
void WaitFlag ( )                          {
   int i;
   for(GlobFlag = 0; GlobFlag == 0 ; )
       if (i++ > LONG_WAIT) break;     }

void  SetFlag ( )   {
     GlobFlag = 1;  }
```

Fig. 2. Flag synchronization method ensures test repeatability

Here *GlobFlag* is a global variable and *LONG_WAIT* is a constant, which limits the time of waiting to avoid an infinite execution of the loop. A simple procedure in Fig. 3 prevents any task to gain access to the processor during the time interval specified with the *CycleNum* value.

```
void HoldTime (int CycleNum)     {
   int i;
   for(i = 0;   i++;   i <  CycleNum); }
```

Fig 3. A simple procedure to prevent gaining access to the processor

3. Flat Charts

The straightforward multi-task test description presented above has a weak point. When a test designer follows the test logic step by step, his attention jumps from one task to another across the text description. In spite of a clear execution order, it is too difficult to recognize the test logic even for very short and simple tests. And this is much more difficult for tests of the length of dozens of operators and more. A more suitable form for test description, the *flat chart form*, was developed by the authors and later was improved with creation of a number of real test suites for various embedded OSs.

The simplest flat chart form is based on the following assumptions:

- repeatability of the test execution order is maintained;
- test utilities in the test application are simple and small in number;
- tests contain invocations of only OS services and test utilities;
- all tasks are generated statically;
- task priorities are static;
- no two tasks have the same priority.

A sequence of actions performed by the test application consists of two kinds of

operations: *OS service operations* (for *DelayCoEnd* these are *Delay()* and *TaskEnd()* operators) and *special testing utility operations* (like *HoldTime()*, *SetFlag()*, *WaitFlag()*, and *End_of_Test()* operators). This structure is typical for any embedded OS test suite. Each OS service and testing utility has a limited number of parameters. With such assumptions, information about each test step may be described with the data structure shown in Fig 4.

```
typedef struct Test_Step   {
        int TaskId;
        void (* UtilServ) ( );
        StepArg Arg_1;
        StepArg Arg_2;        } TestStep;
```

```
union StepArg                {
        int IntArg;
        char* StringArg;
        void (* FunArg) ( );
          ...                };
```

Fig. 4. Information about each test step

where *TaskId* identifies the task that performs the operation. The field *UtilServ* stores a pointer to a procedure which either performs some actions with the test application variables (a procedure from the test utility library), or performs an OS service call. The fields *Arg_1* and *Arg_2* are used to represent the procedural parameters of the test utility or the OS service. As the type of arguments may vary from one operator to another, a union type *StepArg* in this C-like notation is defined, where ... denotes other types of parameters used in service or utility calls.

Now the operator sequence of the test *DelayCoEnd* steps from subsection 2.2 may be described as an array of the *TestStep* type (Fig. 5; steps 01, 05, and 07 were added to ensure repeatability of the test as explained above):

```
TestStep DelayCoEnd [ ] =       {
      1,&CallDelay, 5, 0 ,                    // Step_01
            2,&WaitFlag,  0,  0,              // Step_02
      1,&SetFlag,  0,  0,                     // Step_03
      1, &CallDelay, 50,  0,                  // Step_04
            2,&HoldTime,  0,  0,              // Step_05
            2,&CallDelay, 30,  0,             // Step_06
                  3,&WaitFlag,  0,  0,        // Step_07
      1,&SetFlag,  0,  0,                     // Step_08
      1, &CallTaskEnd,  0,  0,                // Step_09
            2,&SetFlag,  0,  0,               // Step_10
            2, &CallTaskEnd,  0,  0,          // Step_11
                  3,&End_of_Test,  0,  0,     // Step_12
 0,&End_of_scheme,  0, 0        };
```

Fig. 5. Operator sequence of the test DelayCoEnd steps

where the item <0,&*End_of_scheme*,0,0 > terminates the test description. There is no task with the number 0; therefore, a zero in the *Task_Id* field means that this item does not describe an application step, but is an auxiliary one.

Representing a test scheme in form of a single entity (flat chart) is convenient for visual analysis of the test logic as well as for realization of the flat chart interpreter because this simplifies control over the correctness of the order, which steps of the application tasks are executed in.

This *DelayCoEnd* array provides a complete specification of the test application logic. Two important features in this form of test presentation are worth mentioning. First, the order of the *TestStep* structure items is that, which they should be executed in. There's no need to specify step numbers as they are determined by ordering of the *DelayCoEnd* array elements.

The second feature relates to the starting position of each line in the *DelayCoEnd* array description. The test description may be regarded as a table of 12 rows and 3 columns. Columns correspond to tasks. If the line describes an operation to be performed by *Task_i*, then its description shall start in the *i*-th column. Thus, the column order reflects the task priorities and the order of rows reflects the execution sequence.

The authors' experience with test suites design proves the efficiency of this table form called a *flat chart* for describing multi-task test applications. It turned out to be an effective tool for test logic design, understanding, and updating. Moreover, it allows for automation of test suite development for testing embedded real-time operating systems.

4. Data-Driven Test Applications

Automatic processing of flat charts is performed through a corresponding interpreter. An instance of the interpreter is initialized for each task and all such instances run concurrently. Each interpreter instance scans the flat chart specification line by line. Suppose, that each instance has its own variable

```
TestStep* CurrentStep;
```

which points to the flat chart element being analyzed or interpreted.

The *i*-th instance of the interpreter executes only those lines of the flat chart which correspond to the *i*-th task. All others lines are skipped as they are executed by other interpreter instances. When the next line for execution is found, its number is checked – it should be the first line number in the whole flat chart not yet executed by any interpreter instance.

The number of interpreter instances equals to the total number of tasks, which use the same interpreter body parameterized with the task number at the respective interpreter instance initialization.

Such data-driven organization of the test suite has an important advantage: the test application calls the OS under test at only one point, where the test interpreter

invokes a service procedure or a utility pointed to by the flat chart line being interpreted. This simplifies realization of local time measurements (subsection 6.1).

A flat chart description for the *DelayCoEnd* test application in section 4 is represented as an array of *TestStep* structures. Such a form may be used directly for developing a test suite in C. OS services like *CallDelay()* may look as Fig 6 shows.

```
void CallDelay ( )                                  {
        /* Global test variables */
   Delay  ( (CurrentStep->Arg_1).IntArg );      }
```

Fig. 6. OS services CallDelay()

To avoid heavy line interpreting during test runs in real-time, a Forth-like method based on threaded code [6] may be used: the test representation shall be pre-processed in order to convert the initial form into a regular array which elements store the task number, a pointer to the procedure to be called, and the procedure parameters.

5. Developing Scenario Tests with Flat Charts

In accordance with the focus rule (subsection 2.1) each functional test such as *DelayCoEnd* checks a particular OS feature under specific conditions. In this respect, functional tests are not like regular applications. A complete test suite should include also a set of *scenario tests*, which are much closer to regular applications. Each scenario test realizes a sequence of actions, which is based on some underlying idea and uses the OS in a way close enough to real functioning. The flat chart technique is suitable to describe them. The following array (Fig. 7) describes a scenario test for message passing between four threads – three tasks and an ISR.

```
// -----   Flat chart for message passing test application   ------
TestStep MsgTravel [  ] =   {
   1,&CallGetMsg, &mes1_ptr,  0,              // No msg, TASK_1 is waiting
      2,&ResumeIsr,  0,  0,                   //  Interrupt is  simulated
   -1,&CallPutMsg, TASK_3, TEST_MSG,          // Send msg to TASK_3
      2,&CallGetMsg,  &mes2_ptr,  0,          //  No msg, TASK_2 is waiting
            3,&CallGetMsg,  &mes3_ptr,  0,    //  TASK_3 received Msg
            3,&CallPutMsg,  TASK_1, &mes3_ptr,     //  Activate TASK_1
   1,&CallPutMsg,  TASK_2, &mes1_ptr,         // TASK_2 becomes ready
   1,&CallTaskEnd,   0,  0 ,                  //  Activate TASK_2
      2,&Check_Equal,  &mes2_ptr, TEST_MSG,      //  Is msg the same
      2,&End_of_Test,   0,   0,                  // as TEST_MSG?
   0,&End_of_scheme,  , 0 }
```

Fig. 7. A scenario test for message passing between four threads

A negative number in the *TaskId* field corresponds to an operator to be executed by an ISR. Its absolute value specifies the nesting level of the interrupt, which this particular line of the flat chart is interpreted at, rather than a particular ISR.

The scheme *MsgTravel* was designed to check the message exchange mechanism, which provides message pointer passing from an ISR to a task or from one task to another. Variables *mes1_ptr*, *mes2_ptr*, and *mes3_ptr* are message pointers. The value of the constant *TEST_MSG* is a pointer to some initialized message instance.

The scenario of message passing between tasks consists of the following events:

- *Task_1* tries to receive a message and becomes suspended because there's no message for it yet (*Step_01*);
- the ISR passes the message *TEST_MSG* to *Task_3* which is not ready yet to receive it (*Step_02*, *Step_03*);
- *Task_2* tries to receive a message which is absent and therefore becomes suspended (*Step_04*);
- *Task_3* receives the message *TEST_MSG* sent previously by the ISR and resends it to suspended *Task_1* waiting for it (*Step_05*, *Step_06*);
- *Task_1* resends the message to *Task_2* and frees the processor through invoking the service procedure *TaskEnd()* (*Step_07*, *Step_08*);
- upon termination of *Task_1* the message received by *Task_2* is compared to *TEST_MSG* – the two message pointers should coincide (*Step_09*).

The utility procedure *ResumeIsr()* initializes ISR invocation. The simplest way to do this is to throw a software interrupt. Each ISR scans the flat chart line after line, similar to a task. Therefore, an instance of the same common flow chart interpreter is generated for this ISR. Its configuring is performed by just a few operators executed by ISR before entering the common interpreter body. Hence, the same unified flat chart interpreter is used by tasks and by ISRs.

5.1. Loops in Flat Charts

Auxiliary items, such as the terminator *End_of_scheme* mentioned in previous sections are used to build flat charts. Two other kinds of auxiliary items are described below: a flat chart *loop delimiter* (this subsection) and an *error checking operator* (subsection 5.2).

The flat chart loop mechanism allows to prevent construction of a long scheme with repeated fragments. The test *MessQueue* checks the message queue mechanism: a queue of 10 messages is formed for *Task_2*, which then consumes these messages from the queue one after another.

```
//  -----   Flat chart with single loop   ------
TestStep MessQueue [ ] = {
// ----  The message queue with 10 messages is formed for Task_2
0, &LoopStart, &cycle_var,  10,
   1,&CallPutMessage, Task_2, &mes1_ptr,  // Repeat msg send
0, &LoopEnd, &cycle_var,  0,
   1, &CallTaskEnd,  0,  0,                          // Task_1 terminates
 // ---- The message queue of 10 messages is consumed by Task_2
0, &LoopStart, &cycle_var,  10,
          2,&CallGetMessage,  &mes2_ptr,   0,  // Repeat msg receive
0, &LoopEnd, &cycle_var,  0,
0,&End_of_scheme,  , 0        }
```

Fig. 8. Flat chart with single loop

The variable *cycle_var* is used as the loop control variable. Nested loops may by designed, each with its own control variable; e.g., *cycle1_var* for an outward loop and *cycle2_var* for an inner loop.

For each loop a boundary condition shall be satisfied: the task state at the *LoopEnd* delimiter shall be the same as its state when the corresponding *LoopStart* was encountered.

5.2. Testing the Error Handling Service

Test applications *DelayCoEnd* and *MessTravel* demonstrate the suitability of the flat chart technique for testing most of the OS basic services. Each one checks the order of processor switching among threads of actions. *MessTravel* checks correctness of data passing between tasks and from an ISR to a task. Allocation of memory and of special data structures may be checked in a similar way.

Flat chart forms may be further extended to cover testing of the error handling service as well. The following flat chart sample illustrates this possibility (Fig. 9).

```
//  -----   Flat chart for error service testing   ------
   TestStep MemReqErr [ ] =  {
// .......Steps from Step_01 to Step_i exhaust all memory resource
       1,&GetMemory,  30, &mem_ptr,            // Step_i+1
       0, &CheckErrData, NO_MEMORY,  0,
 // ..................... Remaining elements of the MemReqErr array
       0,&End_of_scheme,  , 0 }
```

Fig. 9. Flat chart for error service testing

Task_1 requires 30 memory blocks, which causes an error because the memory resource becomes exhausted. The proposed technique of testing the error handling

service is based on the same interrupt simulation technique as in the *ResetIsr()* utility. An error invokes a special thread of actions, which the flat scheme interpreter body enters. The interpreter finds the respective auxiliary line in the flat chart and performs the *CheckErrData()* utility assuming that the OS reports the *NO_MEMORY* error code into the error handling block.

Thus, auxiliary items extend the flat chart technique and allow to build tests for checking the OS error handling service.

6. Automated Test-Run Sessions

A test suite for an embedded OS shall include automated means for building a test application, for loading it into the target device, for test run, and for producing test-run reports with analysis of the test-run session. Automation tools are intended to organize a specified test session. The test session specification describes an action list for building, loading, running test applications, and analyzing the results.

Scalability is one of the most important requirements for an embedded OS testing application. The user may configure its options to achieve the needed level of efficiency in terms of speed, memory usage, and the needed inventory of services to be used. The number of such OS clones grows exponentially with the number of options. A dozen of binary options correspond to a thousand and more of different OS clones to be tested. A wide set of tests should be built, loaded, run, and analyzed for each such clone, their total number may be a million and more. This results in the need for automation of test sessions with tools to specify them.

Beyond OS scalability there are at least two more reasons for test session automation: OS *projected enhancements* and OS *porting* to other MCUs.

When an OS is ported to a different MCU, the test suite should be ported as well and such porting should take much less effort than initial development. The flat chart technique and automated test sessions allow to save porting efforts .

6.1. Local Time Measurement

There are three basic points in the flat chart interpreter body executed by every thread in the test application. They are:

- the main interpreter loop start point;
- the main interpreter loop end point;
- points of invocation of an OS service or utility.

These points split the body of the interpreter *FlatChartInterpreter()* into the following three sections:

- Section 1 – thread configuring to prepare the thread to enter the main interpreter loop: initialize local variables, determine the *TaskId* or the ISR nested level, set *CurrentStep* to point at the top of the *TestStep* array, and specify the start point of the main interpreter loop;

- Section 2 – organize interpretation of the flat chart through a search of the appropriate item in the *TestStep* array: set *CurrentStep* at the appropriate value, check correctness of the operation sequence, and perform actions prior to invocation of a respective service or utility;
- Section 3 – perform a call of the *UtilServ* utility: (*A*) for local time measurement read the timer register, (*B*) call the *UtilServ* utility, (*C*) for local time measurement read the timer register again, (*D*) perform log operations.

The flat chart technique simplifies realization of local time measurements. Calling an OS service or a utility in the point *B* is performed through indirect addressing of the called procedure. All time measurement actions are around the point *B* (in points *A* and *C*). Storing the result of time measurement is performed in point *D*.

In case of a context switch resulting from performing operator *D*, operator *A* may be performed in a thread other than that, which operator *C* was performed in.

6.2. Global Time Measurement

The OS time service directives are not appropriate for local time measurements. Their precision is not adequate and a direct access to the hardware time register is needed. In contrast, global time measurements are less precise; therefore, the OS time service may be used for them. The structure of a flat chart for global measurements may look as Fig. 10 shows.

```
TestStep HighPriorTaskSwitching [ ] =    {
  1,&CallSysTime, &start_time,  0,          // Store the start time
  0, &LoopStart, &cycle_var,  1000,         // Initialize the loop
//  ------ The set of operations for measurements  ------
  1,&CallGetMessage, &mes1_ptr,  0,         //  Suspend Task_1
     2,&CallPutMessage, &mes1_ptr,  0,      // Send msg to Task_1
  0, &LoopEnd, &cycle_var,  0,              // Terminate loop operations
  1,&CallSysTime, &finish_time,  0,         // Store the end time
  0,&LogGlobalTime,  0,  0,                 // Store the result
  0,&End_of_scheme,  , 0                }   // End of scheme
```

Fig. 10. Structure of a flat chart for global measurements

The number N of cycles required for measurement depends on the relation between the precision ΔT_m of the *SysTime()* mechanism and the duration T_c of one application cycle. Another factor is the duration T_p of the *LoopStart()* and *LoopEnd()* operations (assuming they are equal). The larger the value of $N \times (T_c/(\Delta T_m + T_p))$, the more precise measurement results will be obtained.

Global measurements provide an answer the question: "Does the time of context switching depend on the task priority?" To answer this question, compare the result

of the *HighPriorTaskSwitching* test with the result of the test shown in Fig. 11.

```
TestStep LowPriorTaskSwitching [ ] = {
//  ..... Suspend the set of 100 tasks with high priorities
      101,&CallSysTime, &start_time,  0,        // Store the start time
      0, &LoopStart, &cycle_var,  1000,         // Initialize the loop
// ------ The set of operations for measurements ------
      101,&CallGetMessage, &mes1_ptr,  0,       // Suspend Task_101
        102,&CallPutMessage, &mes1_ptr,  0,     // Msg to Task_101
      0, &LoopEnd, &cycle_var,  0,              // Terminate the loop
      101,&CallSysTime, &finish_time,  0,       // Store the end time
      0,&LogGlobalTime,  0  0,                  // Store the result
      0,&End_of_scheme,  , 0          }         // End of scheme
```

Fig. 11. The test, in which a set of high priority tasks is suspended prior to entering the flat chart loop

The only difference between flat chart loops in the tests *HighPriorTaskSwitching* and *LowPriorTaskSwitching* is in the task priority. In the second test, a set of high priority tasks was suspended prior to entering the flat chart loop. If the OS context switching is performed at the same time for tasks with different priorities, then the measurement results will be the same for both tests.

The considered two tests are a particular case of a round-robin processor switching among tasks. Such a scheme may include an arbitrary number of tasks with different priorities. Changing the number of operating tasks allows to establish the dependency between OS performance and its load while changing the task priorities may impact the speed of task scheduling.

6.3. Latency Testing

For a multi-threaded application executed on a single processor, the tasks and ISRs operations are executed in a quasi-parallel mode. Flat charts are convenient for specifying such quasi-asynchronous processes. From the OS point of view, all threads are asynchronous, but the test logical structure guarantees strong synchronization of all operations in different threads.

However, a true asynchronous mode of operation is needed for measuring the application latency w.r.t. external interrupts. The simplest statistical way of latency measurement assumes simultaneous execution of two logically isolated components:

- a benchmark application with a set of interacting tasks;
- a special measurement ISR to calculate time difference between the moment of the measurement interrupt and the moment when its processing started.

The benchmark application determines conditions for measuring the latency value.

It is built in form of a flat chart within a loop with a large number of iterations, which ensures the repeatability of the conditions of latency measurement.

With this approach, a single result L_m of measuring the latency value will be less than or equal to the maximal possible latency value L_r: $L_r \leq L_m$. The difference $d=L_r-L_m$ represents the inaccuracy a single latency measurement. Let the acceptable inaccuracy Δt of the final result of measurement and the time interval T of time measurement interrupts be greater than the duration of one iteration of the benchmark application. Then the probability P that the required accuracy of measurement is achieved ($d<\Delta t$) is greater than or equal to $\Delta t/T$: $P \geq \Delta t/T$. To achieve higher accuracy of the latency measurements, single measurements are performed n times and the maximum of the values L_m is considered as the final result. The required accuracy of the final result is achieved with the probability P not less than $1-(1-\Delta t/T)^n$: $P \geq 1-(1-\Delta t/T)^n$.

6.4. Measuring Code Coverage

A straightforward technique to measure code coverage of the OS under test by a given test suite is based on direct tracing of the OS code control flow supported with designated software-hardware means. It's hardware component should have a mechanism of trace interrupts with a designated vector (TRAP-interrupts). This software component is composed by a handler of step-wise interrupts which performs the role of the tracing program. Execution of each OS instruction is preceded by an interrupt on the TRAP-vector, which results in the next activation of the tracing program.

This technique of direct tracing matches the rule for non-interference (subsection 2.1). However, it may be inapplicable for embedded systems because an embedded application under test may work much slower when running in parallel with the tracing program. Some operators covered in a real run may be unreachable in the mode of coverage measuring.

A more appropriate technique of measuring code coverage is based on using codes of prohibited TRAP instructions. This mechanism is realized with another designated vector of TRAP-interrupts. In this case, the respective interrupt handler plays the role of the tracing program and the coverage measurement process consists of the following steps:

- the contents of the memory area with the OS body (its code) is saved in a special array and then is filled with the codes of TRAP instructions;
- execution of the test application is started and a software TRAP-interrupt occurs when any OS service is invoked;
- the tracing program is invoked as the interrupt handler, it restores the original OS instruction from the special array and passes control to it;
- the restored original OS instruction is executed;

- if the next instruction to be executed is from the OS body, then it may be either restored through previous executions or still replaced with a TRAP instruction and then another TRAP-interrupt occurs which restores the original OS instruction so that more and more OS instructions are restored.

Upon termination of the test application all OS instructions needed for this application will be restored and their number equals to the number of invocations of the tracing program.

This technique of code coverage measurement with TRAP instructions decreases the time of the test application execution if compared to technique with direct tracing. Each OS instruction corresponds to at most one invocation of the tracing program and therefore the overall execution pace becomes close to that of a regular execution without tracing. A complete match of these two paces is achieved when only one OS instruction, which we'd like to find whether it's covered or not is replaced:

- this one OS instruction is saved and replaced with a TRAP instruction;
- the test application runs to termination and if the instruction is not restored then it was not covered.

This technique with single instruction replacing requires much more processor time because complete measurement of code coverage assumes iterative runs of the test application as many times as there are instructions in the OS body.

6.5. Enhancements of the Flat Chart Technique

As noted in subsection 6.3, the flat chart technique allows to describe a quasi-asynchronous order of test application runs only. To represent true asynchronous threads of actions (as required for latency measurements), methods beyond the flat chart scheme should be used.

The quasi-asynchronous order fits well for testing OS kernel services. However, for testing services related to peripheral devices an extension of the flat chart technique is needed which allows to specify real asynchronous action flows. This may be done through introducing new forms, which specify alternatives in the action flow similar to loop forms in subsection 5.1.

The flat chart technique may be further extended to distributed OS testing. In this case, a test application is a program with true parallelism and if quasi-asynchronous execution turns out to be suitable for particular testing, then the only extension needed is refinement of action flows naming. Otherwise, a separate flat chart should be developed for each physical processor with additional means for cross-referencing among elements of these flow charts.

Flat charts form representations considered above are suitable for usage in C-programs. Similar syntax forms, which require no any special pre-processing, may be developed for other programming languages. However, when moving from one language to another flat charts should be completely reworked which is effort

consuming as the total size of flat charts in a test suite may reach hundreds of thousands lines. Thus, it is reasonable to develop a language independent unified syntax for flat chart forms. Then porting a test suite to another platform requires only to develop a pre-processor of several hundred lines of code. Development of a universal syntax forms for test representation opens the opportunity to build standardized test suites for embedded OS testing. A universal language for OS test applications could be a step forward in development of an automatic test generator [7], [8].

7. Results of Experiments

Experimental data provided below come from authors' experience in developing and testing a particular software product – a compact embedded OS for real-time applications with specific features requested by the customer. The overall approach to developing this OS follows the classical one [9] initially designed for 16-bit single board controllers manufactured by DEC since early 1980-ies. To emphasize the compactness and specifics of such OSs they are usually named "*kernals*" or "*executives*". The usual size of such an OS developed within this approach is about several thousand lines of code in C plus several hundred lines in assembler.

The MCUexec (MicroController Unit EXECutive) product, which development the authors participated in, supported execution of software applications on microcontrollers HC-11 and HC-12 originally manufactured by Motorola, Inc. and since 2015 by NXP Semiconductors. To test the MCUexec functional features, 9 groups of flat charts were developed with the described technique.

For integration testing of MCUexec additional 234 flat charts split in 17 groups were developed, the total number of the developed flat charts being 378. Running all these test suites resulted in 8 detected defects in different versions of MCUexec, each of about 5 KLOCs in assembler. The overall effort for developing these flat charts, running the test suites, and analyzing test run results was 6 staff-months.

Table 1. Nine groups of flat charts for testing the MCUexec functional features

Test group identifier	Brief description	Number of flat charts
Basic	Task delay, system configuration and reconfiguration	10
TaskId	Getting the task Id	3
Task	Task suspending/resuming	12
EventU	Updating and checking of events	21
EventW	Waiting for an event to be set or cleared	30
Slice	Time-slicing features	6
Buf	Buffer manipulating	23
MesS	Message sending and receiving	20
MesR	Reply features	19
	TOTAL:	**144**

8. Conclusion

The flat chart technique gives an efficient way to develop test suites for embedded OS execution-based testing. Flat chart forms allow to build well-structured and understandable descriptions of test applications with specifications of tasks and ISRs for parallel execution. The flat chat technique is suitable for checking the correctness of implementation of basic OS mechanisms – data and signal exchange among action threads, run-time allocations of memory, special structures, and processor's time. Flat charts are efficient not only for developing functional tests but for local and global time measurements, for measuring the OS latency and code coverage. Standardized test suites for embedded OS testing may be built with the described flat chart technique.

References

[1]. Li Q., Yao C. Real-time concepts for embedded systems. CRC Press (2003).
[2]. Thane H., Hansson H. Testing distributed real-time systems. Microprocessors and Microsystems 24(9), 463–478 (2001).
[3]. Desikan S. Software testing: principles and practice. Pearson Education India (2006).
[4]. Myers G.J., Sandler C., Badgett T. The art of software testing. 3rd Edition. John Wiley & Sons, New York (2011).
[5]. Hailpern B., Santhanam P. Software debugging, testing, and verification. IBM Systems Journal 41(1), 4–12 (2002).
[6]. Brodie L. Thinking Forth. Punchy Pub (2004).
[7]. Biswal B. N. Pragyan N., Durga P. M. A novel approach for scenario-based test case generation. In: International Conference on Information Technology 2008 (ICIT'08). IEEE, (2008).
[8]. Lefticaru R., Florentin I. Automatic state-based test generation using genetic algorithms. In: International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC 2007)
[9]. Comer D. Operating System Design: The Xinu Approach, 2nd Edition. – Boca Raton: CRC Press, Taylor & Francis Group, 668 p. (2015).

Техника плоских схем для тестирования встроенных операционных систем

В.В. Никифоров <nik@ iias.spb.su>
С.Н. Баранов <snbaranov@ iias.spb.su>
Санкт-Петербургский институт информатики и автоматизации
Российской академии наук,
199178, Россия, Санкт-Петербург, 14 линия, 39

Аннотация. Современные автоматические устройства все чаще оснащаются микроконтроллерами. Логика работы автоматического оборудования поддерживается рядом различных встроенных программных приложений, которые выполняются под управлением встроенной операционной системы реального времени (ОС). Надежность

ОС чрезвычайно важна для правильной работы всей автоматической системы. Поэтому встроенную ОС следует тщательно тестировать с помощью соответствующего набора автоматических тестов. Такой набор тестов для тестирования встроенной ОС обычно организуется как набор многозадачных тестовых приложений, которые должны выполняться под управлением данных. В статье представлены специальный язык для определения соответствующей логики задачи тестирования и концепция плоских съем для эффективного выполнения тестирования встроенной ОС. Чтобы избежать интенсивной интерпретации текстовых строк во время тестового прогона, предварительно образуется специальное представление теста, в котором исходная строковая форма преобразуется в форму регулярного массива и, таким образом, повышается эффективность тестирования.

Ключевые слова: встроенные приложения; операционные системы; тестирование программного обеспечения; системы реального времени

DOI: 10.15514/ISPRAS-2017-29(5)-5

Для цитирования: Никифоров В.В., Баранов С.Н. Метод плоских схем. Труды ИСП РАН, том 29, вып. 5, 2017 г., стр. 75-92 (на английском языке). DOI: 10.15514/ISPRAS-2017-29(5)-5

Список литературы

[1]. Li Q., Yao C. Real-time concepts for embedded systems. CRC Press (2003).
[2]. Thane H., Hansson H. Testing distributed real-time systems. Microprocessors and Microsystems 24(9), 463–478 (2001).
[3]. Desikan S. Software testing: principles and practice. Pearson Education India (2006).
[4]. Myers G.J., Sandler C., Badgett T. The art of software testing. 3rd Edition. John Wiley & Sons, New York (2011).
[5]. Hailpern B., Santhanam P. Software debugging, testing, and verification. IBM Systems Journal 41(1), 4–12 (2002).
[6]. Brodie L. Thinking Forth. Punchy Pub (2004).
[7]. Biswal B. N. Pragyan N., Durga P. M. A novel approach for scenario-based test case generation. In: International Conference on Information Technology 2008 (ICIT'08). IEEE, (2008).
[8]. Lefticaru R., Florentin I. Automatic state-based test generation using genetic algorithms. In: International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC 2007)
[9]. Comer D. Operating System Design: The Xinu Approach, 2nd Edition. – Boca Raton: CRC Press, Taylor & Francis Group, 668 p. (2015).