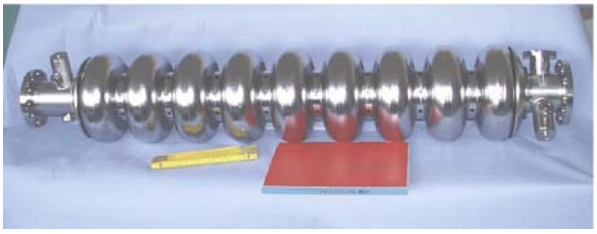
EU contract number RII3-CT-2003-506395 CARE-Conf-06-077-SRF

SRF



TIMING BASED PROCESS EXECUTION IN LINUX ENVIRONMENT

M. BORZ, ECKI, B. S WIERCZ, A. NAPIERALSKI

TECHNICAL UNIVERSITY OF ŁÓDZ, POLAND

Abstract

The complex design process of real-time control systems can be significantly simplified by use of high level abstractions. In particular, the separation of platform-dependent implementation from platform-independent definition helps the design process by providing a predictable, deterministic execution model with well defined timing constraints.Basic block of *TDL* is a stateful module, composed of modes. Each mode is characterised by period, task invocation frequency, output update sequence and possible mode transitions. This paper describes the use of *Timing Definition Language* for the purpose of automated generation of execution environment, which mimics the expected module behavior. Using mode description appropriate priorities for Linux's Round-Robin scheduler can determined, additionally the task execution can be monitored for compliance with defined timing constraints. The logging facilities can be used for debugging of the timing aspect aspects of software execution

Contribution to the MIXDES, Gdynia (Poland)

Work supported by the European Community-Research Infrastructure Activity under the FP6 "Structuring the European Research Area" programme (CARE, contract number RII3-CT-2003-506395)

TIMING BASED PROCESS EXECUTION IN LINUX ENVIRONMENT

M. BORZĘCKI, B. ŚWIERCZ, A. NAPIERALSKI
TECHNICAL UNIVERSITY OF ŁÓDŹ, POLAND

KEYWORDS: execution control, timing abstraction, timing constraints, TDL, Round-Robin scheduling

ABSTRACT: The complex design process of real-time control systems can be significantly simplified by use of high level abstractions. In particular, the separation of platform-dependent implementation from platform-independent definition helps the design process by providing a predictable, deterministic execution model with well defined timing constraints. Basic block of *TDL* is a stateful module, composed of modes. Each mode is characterised by period, task invocation frequency, output update sequence and possible mode transitions. This paper describes the use of *Timing Definition Language* for the purpose of automated generation of execution environment, which mimics the expected module behavior. Using mode description appropriate priorities for Linux's Round-Robin scheduler can determined, additionally the task execution can be monitored for compliance with defined timing constraints. The logging facilities can be used for debugging of the timing aspect aspects of software execution.

INTRODUCTION

The design process of real-time systems is mostly focused on the timing considerations of the runtime environment. A real world system consists commonly of actions taking place periodically with specific frequency. Direct implementation of system's model in low level language may appear troublesome and error prone. Introducing a layer of abstraction and providing a clear description of expected behaviour may significantly simplify the design and implementation processes [3] resulting in more effective solution. In past there have been several attempts to provide the designers with possibilities of describing the system in high level language, with *Giotto* [1] and *Timing Definition Language* [2] as examples.

TIMING DEFINITION LANGUAGE

TDL is conceptually based on *Giotto*. It assumes that the system's actions can be enclosed inside a separable elements called modules. However, unlike *Giotto*, *TDL* assumes that that modules can interoperate and provide each other with well defined interfaces, hence laying foundations for distributed environment (Fig. 1). Each module is composed of smaller elements, which model the functionality of a real world system. These are *actuators*, *sensors*, *tasks* and *modes*. All these elements are equivalent to their *Giotto* counterparts and are well described in *TDL* language specification [2]. Use of *TDL* for the purpose of modeling, is based on providing a high level of abstraction, separating the actual implementation issues in low level languages and hardware complexity from the designer's view (Fig. 2). *TDL* module description can easily interact with lower level languages (C++/C/Java) using well defined data types: *int*, *long*, *float*, *char* and providing means for declaration of user defined types and data structures (what requires the actual definition to be provided by user)

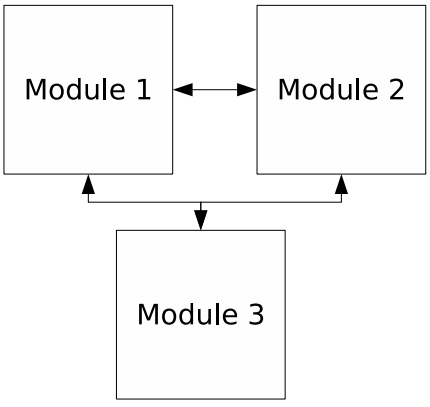


Fig. 1. TDL *modules*

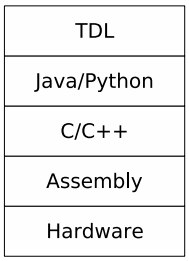


Fig. 2. TDL *abstraction layer*

IMPLEMENTATION

The module definition in *TDL* is the entry point for timing based process execution described in this paper. Language specification [2] suggests use of underlying virtual

machine for interpreting *TDL* code. However, during initial analysis stage it was found that our own implementation of virtual machine would significantly increase complexity of the project as well as the execution overhead, hence it was decided to use scheme as shown in Fig. 3. The final design will support two platforms: Linux and

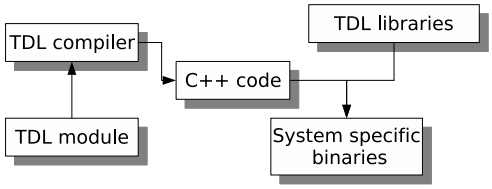


Fig. 3. System implementation

sCore [4] but it would be possible to run *TDL* environment on all POSIX compliant platform. Because of experimental and scientific character of sCore kernel [5], modifications to compiler (*tdlc*) were required as well as a separate set of libraries. It was decided to separate the *TDL* system into two distinct parts:

- static - compiler, header files
- runtime - platform specific libraries: *libtdl-linux* and *libtdl-score*

The compiler design, as shown in Fig. 4 proved to be effective. Use of generic data structure layer, allowes addi-

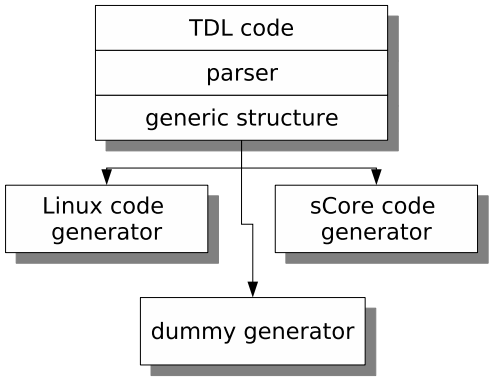


Fig. 4. tdlc - TDL *compiler*

tion of code generators for more platforms and languages (ex. C, Java, Python) to be realised with little effort, as most of usability required for handling the data structures is already provided by base classes. The *TDL* module description is fed into the code generator object, which produces code suitable for the target platform.
The *TDL* library (*libtdl-linux*) includes implementation of such generic elements as ports (*actuators*, *sensors*, *task ports*), tasks, logging objects, each element is implemented having in mind the concurrent nature of final program. *TDL* module uses a variety of external calls: write and read from output/input ports (setters/getters), guards, tasks implementations. User is required to provide the implementation of these calls in specific source files:

- *<module>_<task>_TDLTask.cpp* - task implementation
- *extern_calls.cpp* - other calls (guards, setters, getters)

The *Makefile* can be easily modified and project can be linked with user's own files instead (user provided implementation must be compliant with declarations found in *extern_calls.h*). Each external type has to be defined (or supplied by including appropriate header) in special file *extern_types.h*, which is included in the generated code where needed. Summing up, the process shown in Fig. 5 is used for generating and building C++ code corresponding to *TDL* module description.

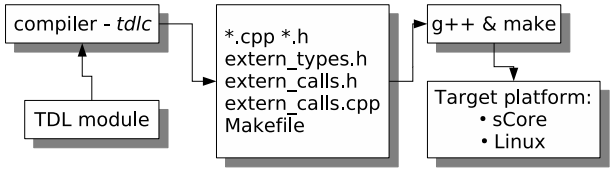


Fig. 5. Code generation process

TDL mode describes a state in which specified actions are performed. Modes are implemented as state machines, with port updates and task invocations taking place as appropriate. The tasks invocations take place sequentially (one after another), however implementations are run concurrently, only the calls which trigger the task implementation to be run are executed sequentially. For details see *TDL* modes definition in Listing 1 and generated C++ code in Listing 2. The resulting C++ code is a *switch* clause, which can be easily optimised by compiler to a jump table. The following mechanism is used by compiler to determine the states:

- find least common multiple (LCM) of frequencies of each of the actions defined in a mode, thus finding the number of states
- divide mode period/LCM yielding state transition time (used by *usleep(3)* call)

Listing 1. TDL *modes definition*

```
start mode main [10 ms]
{
  task
    [1] t1();
    [2] if t2guard(s1) .....
          actuator
    [1] a1 := t1.o;
    [5] if actguard(s1, t1.o) ....
          mode
    [2] if fail(s1, t2.o)
          then stop();
}
mode stop [100ms]
{
  mode [1] if restarted()
          then main();
}
mode freeze [200ms] { ... }
```

Listing 2. Generated C++ code

```
switch(__mode)
{
  case FREEZE:
  {
    ...
  }
  case MAIN:
  {
    switch(__state)
    {
      case 0: {     // 0us
        /* switch to mode STOP
           start with state 0
           if fail == true
        */
        if (fail(Test.s1,
                 Test.t2.o)
        {
          __mode = STOP;
          __state = 0;
          break;
        }
        //call task t1
        Test.t1.exposeOutput();
        Test.t1.i.updateValue(
          Test.s1.getValue()
        );
        Test.t1.i2.updateValue(
          Test.s2.getValue()
        );
        // call task t2
        if (t2guard(Test.s1) ) {
          ...
          Test.t2.run();
        }
        break;
      }
      case 1: {     // 2000us
        ...
      }
      case 2: {     // 4000us
        ...
      }
      ...
    }
  }
}
```

The concurrency is provided by Linux system facilities, *i.e.* standard POSIX threads mechanism [7]. The library *libtdl-linux* guarantees that there will be only one instance of each task implementation running, hence whenever the currently running instance has not finished at the time of subsequent invocation error is generated. This information can be used for realtime analysis of running environment. By adjusting the scheduling policy to Round-Robin [8] and assigning appropriate scheduling priorities satisfactory results can be obtained. Moreover the overhead related to thread creation has been minimised by use of thread pools.

The priority adjustment mechanism is based on well known solutions described in [9]. A mix both of *rate-monotonic* and *deadline driven* priority assignments is used with several assumptions:

- requests for tasks are periodic (due to mode period and task invocation frequency)
- task processing is finished before the subsequent request (controlled by synchronisation mechanism)
- run time for each task is constant (holds under low system load)

Task invocation frequency is used as main factor for determining priority, following *rate monotonic* approach tasks with highest frequency are given highest priority. Remaining tasks are adjusted by use of *deadline driven* algorithm. It has to be noted that although priorities are assigned by use of methods described above, the actual scheduling mechanism is governed by Linux system, particularly fixed priority *Round-Robin* scheduler. The execution of module, i.e. task invocations (start and stop times), port updates are logged, allowing offline analysis with simple Python script and carrying out performance evaluation.

PERFORMANCE

Simple performance test has been conducted with module as shown in Fig. 6. Two tasks are used:

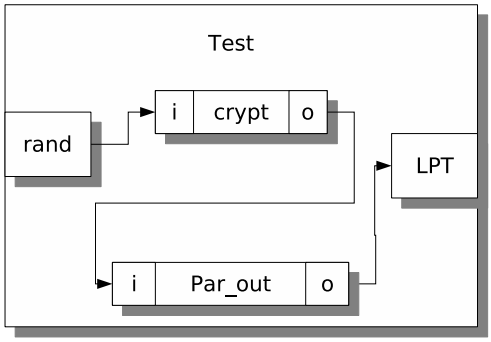


Fig. 6. Module used for testing

- crypt - uses *crypt*(*3*) call, task input *i* provides salt, encrypted key is exposed at output *o*
- Par_out - outputs least significant byte of input *i* on output *o*

Moreover, the module is composed of the following modes:

- main - period 10ms, runs crypt once and Par_out twice per period
- freeze - period 200ms, waits for parallel port to become available
- stop - period 1000ms, stops the execution due to port failure

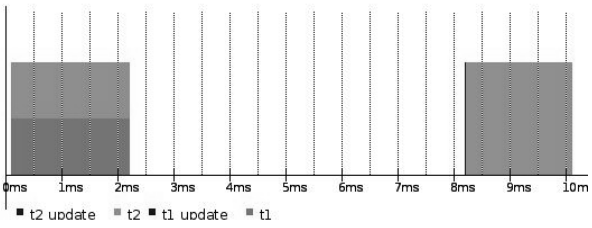


Fig. 7. Performance graph

Although the tasks are simple with short execution times, the actual performance may be disturbed by other user-land processes working in the background. It can be seen in Fig. 7 that the second task invocation missed it's expected time (starts at 8ms instead of 5ms). This may be caused by low accuracy of timing measurements, which will be modified in future version.

FUTURE WORK

The future work will be mainly concentrated on further analysis of adequate priority assignment algorithms. Specifically aimed at reducing latency and jitter. Additionally a need to implement faster synchronisation elements than POSIX semaphores may appear. It is expected that significant improvement may be obtained by refactoring the state machine for each *TDL* mode, which currently uses mostly *usleep(3)* call. New implementation may be based on *setitimer(2)* instead for more accurate timings. The *TDL* compiler tools and libraries described in this article will be proposed to use for the purpose of CARE project in which the authors' department takes part. *TDL* module definitions will be used for automated code generation with specific timing requirements. The nature of the project focuses on design of a radiation tolerant system working in error prone environment (specifically under influence of radiation induced errors) [5][6]. Using *TDL* one may define which tasks should be run and the desired frequency. The library provided mechanisms allow for real time execution control, hence radiation triggered memory corruption which may lead to system instability can be detected and appropriate action can be taken.

CONCLUSIONS

The *Linux* implementation shows that it is relatively easy to generate code which will try to follow the design assumptions. However, the main caveats are due to the non-realtime properties of unmodified *Linux* system. Use of real-time extensions available in *RTLinux* [10] or other operating system may help to improve obtained results, especially to reflect real-time properties of *TDL* modules. Another factor that may significantly influence the perfomance is appropriate priority assignment. Assigning high priority with use of *Round-Robin* scheduling policy may help to reduce the latency and favour *TDL* process even under heavy load. The only significant inconvenience is the need to supply the *worst case execution time* of each task, as it cannot be directly estimated and requires prior measurements. The attempt to create timing based execution environment is actually a proof-of-concept showing that it is possible to schedule tasks meeting specific constraints with acceptable accuracy even on unmodified standard Linux system. The range of possible applications spans from remote device polling to system load monitoring. It is expected that porting *TDL* to system providing real-time extensions such as sCore may provide even better results.

ACKNOWLEDGEMENTS

We acknowledge the support of the European Community-Research Infrastructure Activity under the FP6 "Structuring the European Research Area" program (CARE, contract number RII3-CT-2003-506395), and Polish National Science Council Grant "138/E-370/SPB/6.PR UE-DIE 354/2004-2007".

REFERENCES

[1] Thomas A. Henzinger Benjamin Horowitz Christoph M. Kirsch, "Giotto: A Time-triggered Language for Embedded Programming", Proceedings of the IEEE, Vol. 91, No. 1, January 2003.

[2] Josef Templ, "TDL Specification and Report", University of Salzburg, Computer Science Department, 2004.

[3] Wolfgang Pree, "Trends in Embedded Software Engineering", University of Salzburg, Department of Computer Science, 2005.

[4] B. Świercz, "Mikrojądro systemu czasu rzeczywistego" (Polish), Master thesis, 2004.

[5] B. Świercz, D. Makowski, A. Napieralski, "The sCore – Operating System for Research of Fault-Tolerant Computing", 12^{th} Mixed Design of Integrated Circuits and Systems, Mixdes 2005

[6] B. Świercz, D. Makowski, A. Napieralski, "The IAradSim - IA32 architecture under a high radiation environment simulator", NSTI Nanotechnology Conference and Trade Show, Smart Sensors and Systems, Nanotech 2005

[7] Information technology - Portable Operating System Interface (POSIX), International Standard ISO/IEC 9945-1: 1996 (E), IEEE Std 1003.1, 1996 Edition

[8] D. Abbott, "Linux for Embedded and Real-time Applications", Elsevier Science, 2003.

[9] C. L. Liu, J. W. Layland, "Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment", Journal of the Association of Computing Machinery, Vol. 20, No. 1, January 1973.

[10] V. Yodaiken, "FSMLabs Lean POSIX for RTLinux", Finite State Machine Labs 1999, 2000.