TDL4Gumstix: Extending the TDL Tool Chain to Support a new Platform

Gerd Dauenhauer and Patricia Derler

***Abstract*—TDL4Gumstix is a case study showing how to extend the TDL Tool Chain to support a new platform. The platform chosen is Gumstix, a Linux-based computer system of the size of a gum stick. On this platform, real-time tasks designed with the Timing Definition Language together with MATLAB Simulink are executed. A simple Java client simulates Sensors and Actuators that are used by the tasks running on the new platform. The communication between the client and the platform is done via UDP.**

***Index Terms*—timing definition language, Gumstix, real-time programming**

I. Motivation

The *Timing Definition Language* (TDL) allows one to specify the timing behavior of a hard real-time control application in a descriptive way and separates the timing aspect of such applications from the functionality [1]. TDL enables the definition of components that behave identically on any platform and with any distribution schema. TDL generates code for Tasks and Modules that have to run on the platform which means that the code generators need to have a minimum amount of knowledge of the platform. In this work, we show that there are only a few steps necessary when porting TDL for a new platform by providing TDL support for the Gumstix platform.

II. Hardware Setup

We used the Gumstix hardware as a platform for porting TDL. Gumstix, inc. is the maker of full function miniature computers(FFMC) in gum stick size (20mm x 80mm x 8mm) having a Linux motherboard [2]. We used the Gumstix 400xm with a 400 Mhz Intel XScale CPU and a 16 MB Flash memory together with an extension board to enable communication over Ethernet (Dual ethernet board). A buildroot environment provided for the Gumstix platform which is publicly available enables comfortable compilation

email: gerd.dauenhauer@cs.uni-salzburg.at
email: patricia.derler@cs.uni-salzburg.at

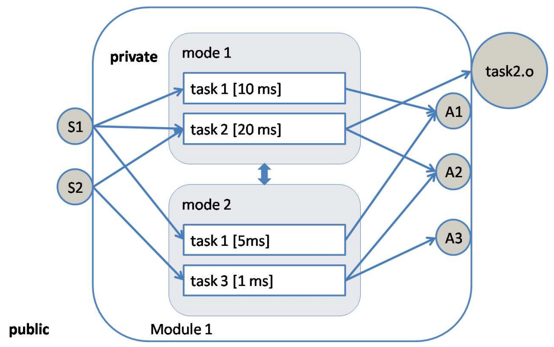


Fig. 1. TDL - An Example

III. TDL

The Timing Definition Language is a high-level textual notation for defining the timing behavior of real-time applications. TDL is based on Giotto [3] abstractions and concepts like logical timing, the separation of timing and functionality, modes and mode switches. Furthermore, TDL is extended by a component architecture, it allows the *transparent* distribution over multiple nodes and provides a new syntax. The main goal of TDL is the separation of timing from implementation.

Basically, real-time software involves a set of *tasks* that get input from a set of *sensors* and calculate output for a set of *actuators*. A *mode* is a collection of tasks representing a particular state of operation of a control application. Modules as units of parallel composition and units of distribution consist of multiple modes, sensors and actuators. Tasks, sensors and actuators can either be public or private for a module. Figure 1 shows the basic concepts of TDL.

The same example can be written in TDL Syntax which is shown below. For further information about the TDL Syntax please look at [1].

```
module Module {

  sensor    double s1 uses getS1;
  sensor    double s2 uses getS2;
  actuator double a1 := 1 uses setA1;
  actuator double a2 := 2uses setA2;
  actuator double a3 := 3 uses setA3;
  task t1 {
    input  double port1;
    output double port2;
    uses t1Impl(port1,port2);
  }

  task t2 {
    input  double port1;
           double port2;
    output double port3;
    uses t2Impl(port1,port2,port3);
  }

  task t3 {
    input  double port1;
    output double port2;
           double port3;
    uses t3Impl(port1,port2,port3);
  }

  start mode mode1 [period=10 ms] {
    task
      [freq=10] task1 {port1 := s1;}
      [freq=20] task2 {port2 := s2;
                       port1 := s1;}
    actuator
      [freq=10] a2 := task2.port3;
      [freq=10] a1 := task1.port2;
    mode
       [freq=10] if s1 = 100
                         then mode2;
  }

  mode mode2 [period=1 ms] {
    task
      [freq=5] task3 {port1 := s2;}
      [freq=1] task1 {port1 := s1;}
    actuator
      [freq=5] a2 := task3.port2;
      [freq=5] a3 := task3.port3;
      [freq=1] a1 := task1.port2;
  }
}
```

A. *The TDL Tool Chain*

An extensive tool chain enables designing TDL applications graphically and generating TDL and platform specific code. Figure 2 shows the main tools and how they interact.

With MATLAB Simulink [4], tasks and guards are designed graphically. The *TDL:VisualCreator* enables the graphical design of an application according to TDL semantics. The Real-Time Workshop of Simulink generates C-Code for task and guard implementations. The TDL:VisualCreator generates the TDL code. The *TDL:VisualDistributor* assigns modules to nodes and generates C-Code for the execution of those modules on a specific platform. In our case, code for the Gumstix hardware needs to be generated. Furthermore, wrapper code is generated for the code generated by Simulink. The *E-Machine* is a virtual machine for executing the TDL tasks and modes and provides the IO functionality.

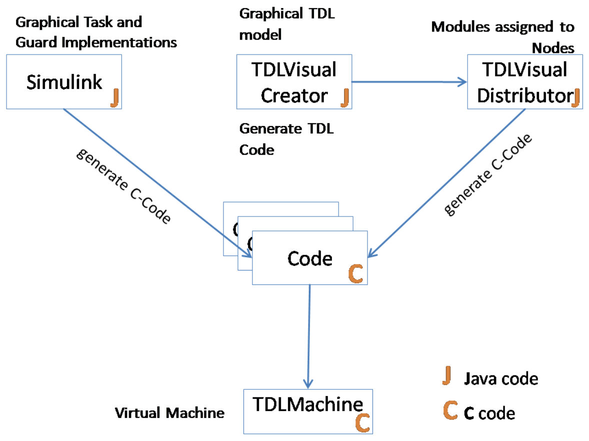


Fig. 2. TDL Tool Chain

IV. Extending TDL for the Gumstix Platform

In this section, we describe the necessary steps to port TDL for a new platform. Two main extensions of the existing tool chain have to be performed:

- Provide the new Platform in the TDL:VisualDistributor.
- Create an E-machine for the new platform.

A. *TDL:VisualDistributor Plug-In*

The Gumstix platform is implemented as a plug-in for the TDL:VisualDistributor. As such its main class GumstixPlatform is derived from the abstract base class EmbeddedCPlatform for code generation aspects and the interface NodePlatform for GUI integration aspects.

A.1 GumstixNodeProperties

The node attributes are presented in a typical properties view; properties are used to specify some settings for the code generation shared between all mod-

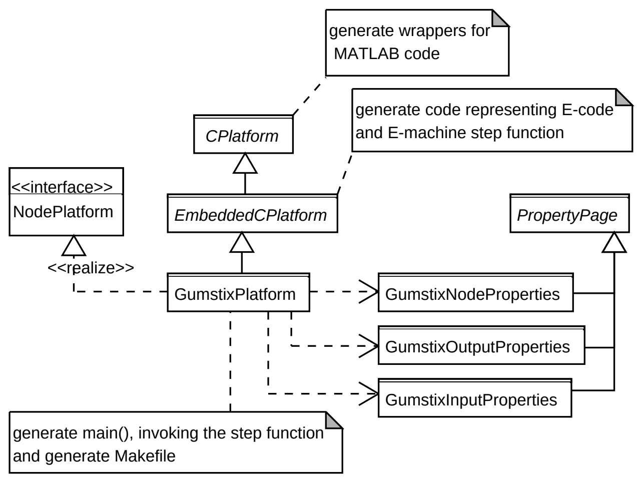

generate wrappers for MATLAB code
generate code representing E-code and E-machine step function
CPlatform
<<interface>>
NodePlatform
EmbeddedCPlatform
PropertyPage
<<realize>>
GumstixPlatform
GumstixNodeProperties
GumstixOutputProperties
GumstixInputProperties
generate main(), invoking the step function and generate Makefile

Fig. 3. Plug-In classes

ules on that node. The emitted code - precisely the Makefile - contains references to directories that must be defined by the user:

- Build Root specifies the installation directory of the Gumstix cross compilation system. E.g. the C compiler referenced throughout the Makefile is found in that directory.
- E-Machine specifies the directory containing the source code for the Gumstix specific E-machine implementation including I/O device drivers.
- MATLAB specifies the directory containing some header files referenced in the source code generated by MATLAB Real-Time Workshop.

A.2 GumstixInputProperties and GumstixOutputProperties

Input and output ports in TDL must be assigned to a specific I/O port on the Gumstix hardware - in our case, they are just emulated through a TCP/IP connection. The property views present a list of TDL ports; for each TDL port, the corresponding I/O port is assigned from a selection box. Usually one would implement some intelligence in the property views like

- restrict assignment to compatible types only, i.e. boolean TDL ports to digital I/O ports and float or integer TDL ports to analog I/O ports or
- make selectable only those I/O ports that are not already assigned.

A.3 GumstixPlatform

The Gumstix platform support is implemented by extending the functionality already found in the code for typical embedded hardware. The base class already emits files containing the E-code instructions encoded as C-code.

The base class also emits a C file containing the step function to be invoked at each tick of the hyper period of all TDL modules. Since each embedded hardware's operating system has different mechanisms for invoking a function repeatedly, this detail is left open for subclasses; our implementation simply emits additional C code that e.g. implements a main function with some initialisation code and an infinite loop with sleep and invoke instructions[1]:

```
int main(int argc, char **argv) {
  nextTimestamp = getCurrentTimestamp();
  InitIO();
  initTDLMachine();
  fflush(stdout);
  for (;;) {
    tdl_machine_step();
    nextTimestamp += 100000;
    while (sleepUntilTimestamp(
             nextTimestamp) > 01);
  }
  return 0;
}
```

Other functionality the Gumstix platform class has to provide is a mapping mechanism from the "abstract" I/O port functions defined in the E-machine to our implementation

The majority of code lines in the Gumstix platform class is, however, responsible for generating the Makefile:

- The C (cross-)compiler to be used is defined by the node property "Build Root".
- The compiler's include directories argument is constructed from the node properties "E-Machine" and "MATLAB"
- Fially, all the build targets for all modules, tasks and guards are emitted.

The result of the plug-in invocation is a directory structure containing all source and support files necessary to cross-compile the TDL application, resulting in a single executable that can be copied to the Gumstix computer.

B. *Gumstix E-Machine*

The implementation of the E-machine for the Gumstix platform is based on the implementation for the DeComSys[2] FlexRay[3] Evaluation system

[1] The sleep duration 100000 is the computed hyper period of this node's modules in micro seconds
[2] http://www.decomsys.com
[3] http://www.flexray.com

NODE<RENESAS>. Our implementation, however, is much simpler; we do not provide support for multi-node configurations, and as such, we do not need to implement the TDL communication layer. We were able to reuse the code by just *removing* the communication and I/O parts.

C. E-Machine - I/O

The I/O implementation of the TDL E-machine handles the communication between the Gumstix hardware and sensors and actuators. Information is sent to and from the Gumstix hardware via UDP packets. There are two types of messages which can be sent and received by the Gumstix hardware:

- Sensor Message. A sensor message contains information about the new value of one sensor. The format of a sensor message is:

  `<SType>:<SNumber>:<SValue>`

  The sensor type can either be *a* for analog sensor or *d* for digital sensor. Sensors are consecutively numbered and the sensor number relates to exactly one sensor of type digital or analog. The sensor value is the actual value of the sensor.

  The message `<a>:<2>:<100>` could be sent to the Gumstix hardware and there be interpreted as follows: The actual value of the second analog sensor is 100.
- Actuator Message. The actuator message contains information about the new value for one actuator. The format of an actuator message is:

  `<AType>:<ANumber>:<AValue>`

  The actuator type can either be *a* for analog actuator or *d* for digital actuator. Actuators are consecutively numbered and the actuator number relates to exactly one actuator of type digital or analog. The actuator value contains the new value to which the actuator of type *AType* with ID *ANumber* should be set.

  The message `<d>:<3>:<42>` could be sent from the Gumstix hardware to the third digital actuator which would then set its value to 42.

When the system is started, the E-machine needs to initialize the I/O component. During initialization, two threads are started. One thread is used to send values to actuators and one thread is used to receive messages from sensors.

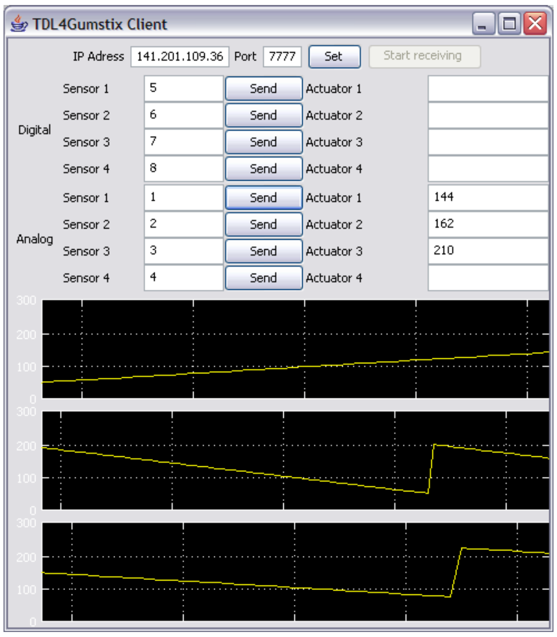
When a sensor message is sent to the platform running the TDL tasks, this message is parsed and stored in an internal array. AnalogIO_get or DigitalIO_get return the current value of a sensor which was sent some time in the past and was stored in the internal array.

AnalogIO_set and DigitalIO_set are used to set the value of an actuator. Those values are stored locally and whenever the sending thread is awake, it creates messages out of those values and sends those messages to the actuators.

V. Setting and Visualizing Sensor and Actuator Values

TDL tasks receive input from sensors and send output to actuators. In our approach we used UDP packets to transport information between sensors and actuators and the Gumstix platform.

For the sake of simplicity and to be able to concentrate on the main goal of this work, sensors and actuators are being simulated. A simple Java application simulates both, sensors and actuators. In figure 4 , a screenshot of the application shows how the application is structured. We decided to provide 4 analog and 4 digital sensors and the same amount of actuators. By pressing the send-button next to a text field for a sensor, a sensor message is created and sent to the destination IP address.


TDL4Gumstix Client

IP Adress 141.201.109.36 Port 7777

Digital	Sensor 1	5	Actuator 1	
	Sensor 2	6	Actuator 2	
	Sensor 3	7	Actuator 3	
	Sensor 4	8	Actuator 4	
Analog	Sensor 1	1	Actuator 1	144
	Sensor 2	2	Actuator 2	162
	Sensor 3	3	Actuator 3	210
	Sensor 4	4	Actuator 4	

Fig. 4. TDL4Gumstix Client

For the visualization of actuator port signals, we created a simple "Scope" GUI component in Java. The scope displays the trace of the signal in a scrolling 2D area. The appearance of the component intentionally follows that of the scope control in Simulink. The y axis is configured with the minimum and maximum

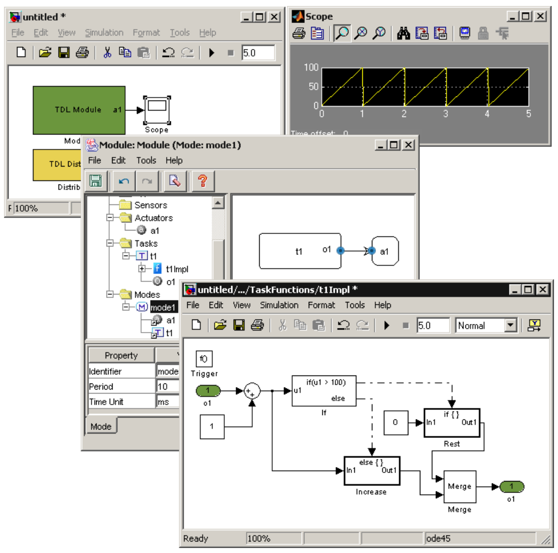

untitled *
Scope
TDL Module
Module: Module (Mode: mode1)
untitled/.../TaskFunctions/t1Impl *
Merge
Increase
Reset

Fig. 5. TDL - An Example

of the expected signal value; these values and some values in between are also used to render the labels and the grid. The grid on the x axis is automatically rendered by the component; each grid line represents one second.

The component is implemented as simple as possible, e.g. it does not follow the model-view-controller pattern, but stores the sampled signal values on its own. Data is added as a timestamp/value pair; data is stored in a circular buffer of some preconfigured size, so the scope always shows the last N sample values. Each added sample value causes the component to repaint all of its content.

Although the component was implemented in the context of our project, it may be reused in similar situations; we would recommend, however, to add some performance enhancements like block move operations for scrolling and painting only the added samples.

VI. An Example

We show how to build a TDL application in a very simple example: a ramp generator.

First, we define the modules required by TDL with the TDL:VisualCreator (see figure 5. In our example, we define one module (TDL Module). In this module, we create a task which has one actuator. The task implementation is designed with Simulink and Simulink standard blocks. Our task returns a value which it increases and resets to zero when it reached 100; a ramp is generated.

The distribution of the modules is defined in the TDL:VisualDistributor (see figure 6. Our module is placed on a node (Node1), worst case execution times of tasks are defined and actuator names in the model are mapped to real actuator names. We used a fixed list of actuators in our example as we do not have real actuators but we only visualize actuator values in an external application.

With the TDL:VisualDistributor, the code required for executing the modules on a platform is generated.

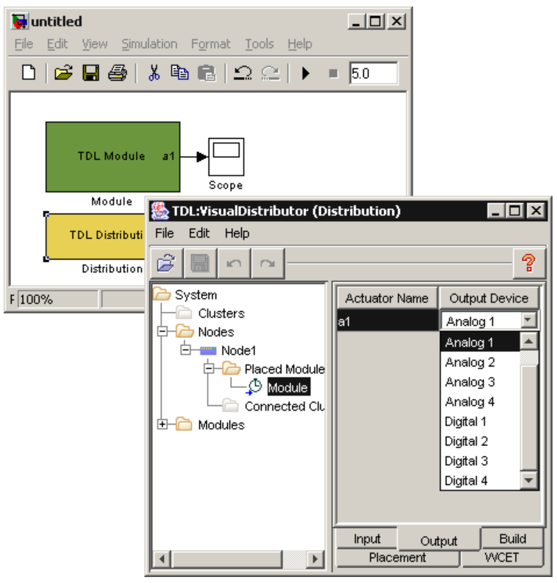


Fig. 6. Distribution of TDL Modules

VII. Outlook

The implementation shown in this work is very simple because the main goal was to show that it is possible with only a few steps to port TDL to a new platform. In a *real-world* application, real sensors and real actuators would be used. Furthermore, multiple Gumstix nodes could be supported. This would result in a difficult synchronization problem between nodes.

VIII. Contribution

Gerd Dauenhauer: E-Machine modifications, TDL:VisualDistributor Plug-in, Scope
Patricia Derler: E-Machine IO, TDL4Gumstix Java Client

References

[1] Josef Templ, TDL Timing Definition Language Specification. Technical Report, October 2006
[2] Gumstix, Gumstix way small computing `http://www.gumstix.com/`
[3] Giotto Project, `http://www-cad.eecs.berkeley.edu/~fresco/giotto/`
[4] The Mathworks. Simulink - Simulation and Model-Based Design, `http://www.mathworks.com/products/simulink/`