Implementing a distributed TDL-EMachine for the RTLinux Platform

Klemens Winkler

University of Salzburg
Institute for Computer Science
5020 Salzburg, Austria

July 22, 2004

Contents

Overview

Introduction

The main goal of my diploma thesis is implementing a TDL-EMachine to aid the development of distributed, real-time systems for the RTLinux Platform. RTLinux provides the necessary real-time functionality needed by most applications. The main steps will be:

1. Extending the TDL Compiler to generate RTLinux specific Code.
2. Implementing a full functional TDL EMachine on RTLinux.
3. Implementing a global data pool, to share variables among different nodes over the network.

The next sections will describe above steps in more detail.

Extending the TDL Compiler

In order to understand the need of extending the TDL compiler we need to take a closer look at the TDL Tool Chain. First of all the user needs to provide a program describing the timing behavior of the system, called TDL-Program. This program is the input for the TDL-Compiler, which generates platform independent ECode (binary code) and platform dependent Glue Code (needed as a link between Functionality Code and EMachine during system execution). Nothing magical about the next steps. The Glue Code and Functionality Code are compiled and linked together. On the RTLinux Platform the resulting object file is a Linux-Kernel Module. The Kernel Module with the corresponding ECode file are called Partition. As you might have guessed, we need to extend the TDL Compiler to produce the RTLinux specific Glue Code in order to get a Kernel Module.

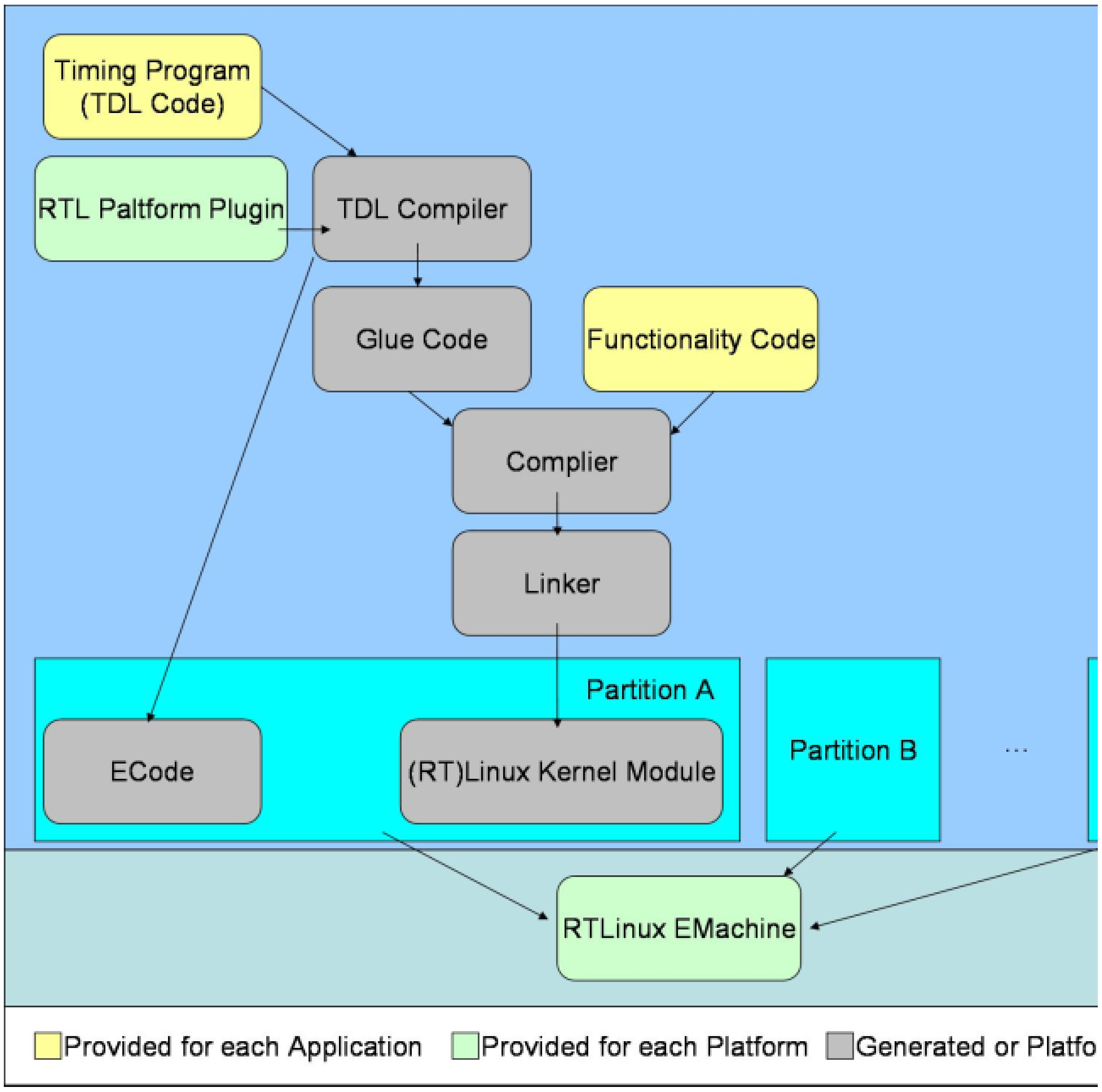

Figure 1.1: Tool Chain

Implementing a TDL EMachine for RTLinux

The TDL-EMachine is a Kernel Module itself. As you can see in Figure1.1 it gets the ECode (discribing the timing behavior) and Functionality Code as input in order to execute the TDL-Program. The TDL-EMachine is able to execute more than one partition in parallel, as long as all timing constraints can be met. This is achieved by partitioning. Each module gets as much CPU as needed in the worst case. The main components of the TDL-EMachine are:

- Registration Service

- ECode Parser
- Scheduler
- Dispatcher

The next subsections will describe above steps in more detail.

Registration Service

Each Partition registers itself at the registration service provided by the TDL-EMachine. This mechanism is needed in order to load Partitions at runtime and notify the TDL-EMachine, that it needs to take care of one more Partition. If the new Partition needs more CPU time than available, the Partition is rejected by the TDL-EMachine. The process mainly uses standard Linux concepts (system calls) and passes pointers to functionality code to the TDL-EMachine. Figure1.2 shows the described process.

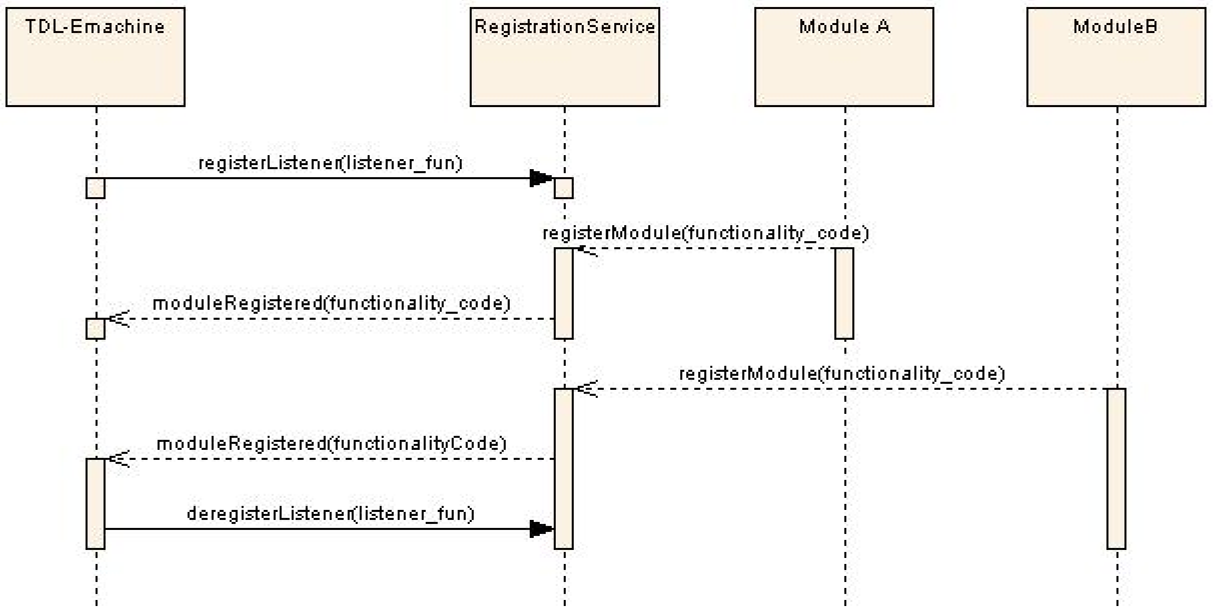
TDL-Emachine
RegistrationService
Module A
ModuleB
registerListener(listener_fun)
registerModule(functionality_code)
moduleRegistered(functionality_code)
registerModule(functionality_code)
moduleRegistered(functionalityCode)
deregisterListener(listener_fun)

Figure 1.2: Registration Service

ECode Parser

Since it is not good practice to access I/O directly from real time programs, RTLinux provides a mechanism called Real Time Fifos. A user space Program reads the ECode file from hard disc and puts it into a designated Real Time Fifo. After the whole file is read the ECode Parser, running in kernel space, parses the byte stream and builds all the internal data structures needed in order to execute the program.

Scheduler

After a partition is loaded (Functionality Code is registered and ECode File is parsed) a EDF schedule is calculated for each partition. This schedule is needed by the dispatcher.

Dispatcher

The dispatcher is responsible for time partitioning and meeting deadlines calculated by the scheduler. These calculations need to be done dynamically since we do not know how many partitions will get loaded during program execution.

Software Bus

A challenging part will be designing and implementing a protocol, which supports global data sharing among all nodes in the system. This protocol is called Software Bus and is under design right now. In order to meet real-time constraints the Software Bus needs to rely on a predictable and reliable Protocol (in time and value domain) it is based on. I have chosen a TT-Ethernet implementation written by Walter Egger.

Example with two nodes

The following example shows the hierarchy of the components and the communication path through them. For the developer of a real-time application the communication between RTThreads (Application Threads) is transparent. The system has to guarantee, that the correct value is available at the right time. Whereas the physical communication path depicts the actual method messages are transmitted.

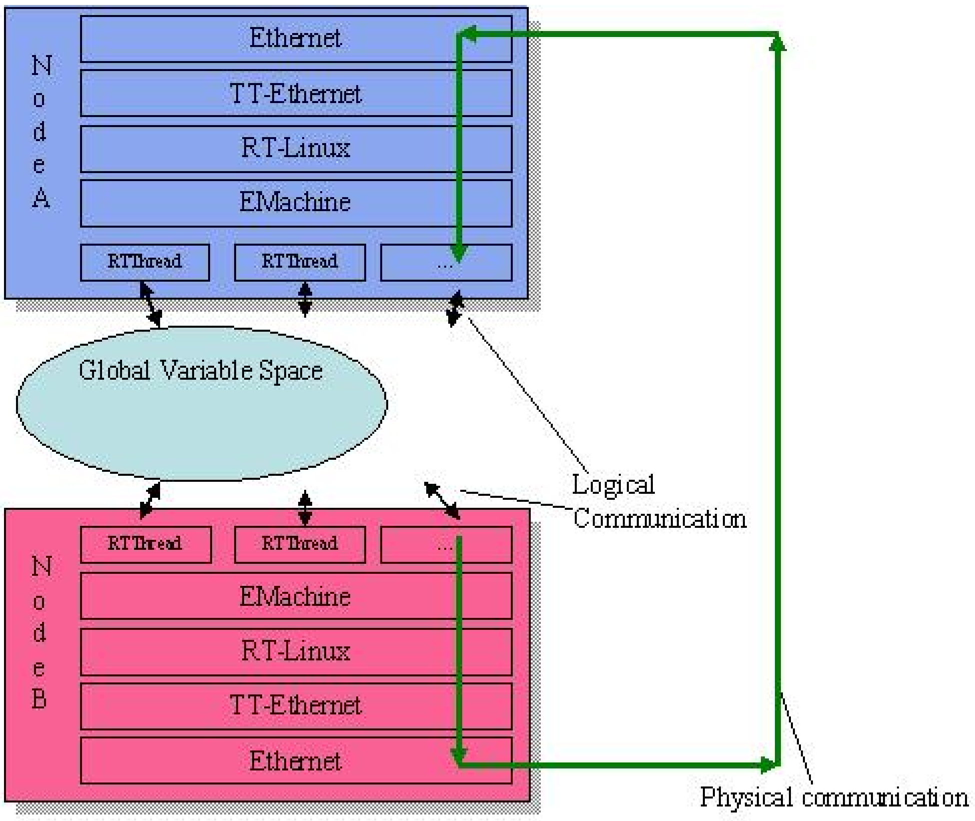

Figure 1.3: SWBus

Expected Completion Date

October 2004

About this document ...

Implementing a distributed TDL-EMachine for the RTLinux Platform

---

Klemens Matthias Winkler 2004-07-22