Reducing the time taken to test your next embedded system

Michael J. Pont

TTE Systems Ltd

This short paper provides an introduction to “time triggered” system architectures and explains how the use of such architectures can help to reduce the time taken to test designs. The paper is based on a presentation made by the author at the UK Embedded Masterclass events held in Cambridge and Reading in May 2010.

Introduction

There are two main architectures used to develop software for modern embedded systems: these can be labelled as “event triggered” or “time triggered”.

For many developers, event-triggered (or “ET”) architectures are more familiar. ET designs involve creating systems which handle multiple interrupts. For example, interrupts may arise from periodic timer overflows, the arrival of messages on a CAN bus, the pressing of a switch, the completion of an analogue-to-digital conversion and so on. To create such systems, the developer may write code to handle the various events directly: this will typically involve creating an “interrupt service routine” to deal with each event. Alternatively, the developer may employ a conventional real-time operating system (RTOS) to support the event handling. Whether an RTOS is used or not, the end result is the same: the system must be designed in such a way that events – which may occur at “random” points in time, and in various combinations – can always be handled correctly.

The alternative to an event-triggered architecture is a time-triggered (“TT”) architecture. Although TT architectures are widely used in industries such as aerospace and medical systems, they are less familiar to developers of mainstream embedded systems. When implementing TT systems, the key thing we need to remember is the “one interrupt per CPU” rule. That is, **TT designs only have one interrupt enabled**. This single interrupt is usually linked to a timer “tick”, which might occur (for example) every millisecond. In a TT design, all other inputs are polled.

Learning to design with “just one interrupt” may sound like a significant challenge, but it’s much easier than you may imagine and – as we will seek to demonstrate in this brief paper – use of a TT architecture can have very significant benefits for developers, not least a reduction in testing times.

In the remainder of this paper, we first provide a summary of some of the key features of both static and dynamic TT systems. We then explain how use of TT architectures can help to reduce testing effort, and describe how you can determine whether use of this approach will be effective in **your** next embedded system.

Static TT architectures: Some examples

When we say that a computer system has a **static TT architecture**, we mean that we can determine in advance — before the system begins executing — *exactly* what it will do at every moment of time in which it is running. As we will discuss later in this paper, this level of predictability has a number of significant benefits, both for developers and eventual system users, not least the fact that system testing is very straightforward.

The simplest practical implementation a static TT architecture is illustrated schematically in Figure 1. This figure assumes that a system timer has been confirmed to overflow periodically (in this case every millisecond). This “system tick” is used in turn to trigger an interrupt service routine (ISR), represented in the figure as a function called Update(). When not executing the Update() function, the system is assumed to "go to sleep" (that is, it will be in a low-power idle state).

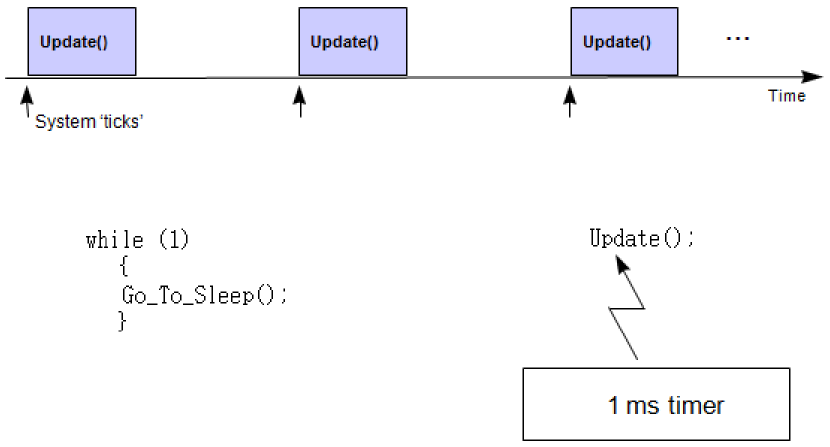


Figure 1: A simple implementation of a static TT design, based on periodic system "ticks" (in this case every 1 ms). Please note that in the design summarised in Figure 1 (as with all TT designs), we need to follow the "one interrupt per CPU" rule: that is, apart from the timer interrupt, no other interrupts can be enabled in this system.

Many current products – in sectors ranging from automotive to household consumer goods - employ the simple approach outlined in Figure 1. However, there are various other options when we wish to create static TT designs. For example, in the aerospace sector some form of "time line" scheduler of the form illustrated in Figure 2 is common.

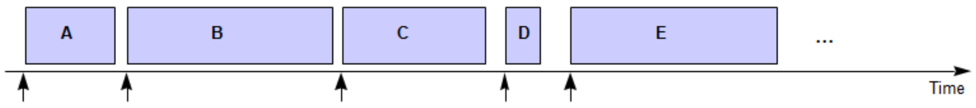


Figure 2: An example of a "timeline scheduler". In this TT architecture, the tick interval varies to match the worst-case execution time of the individual tasks which are being executed in the system.

The solutions illustrated in Figure 1 and Figure 2 are called a "time-triggered, co-operative" (TTC) architectures. They are referred to as time-triggered because – through the use of the timer ISR and the "one interrupt per CPU" rule – we know that <u>**all**</u> of the system operations are driven by the passage of time. The scheduling employed is called co-operative (or "non-pre-emptive") because the tasks cannot be pre-empted (that is, each tasks runs to completion, and then returns control to the scheduler or RTOS).

Use of co-operative tasks simplifies the scheduler / RTOS design, and can simplify the implementation. However, use of a TT approach does not restrict the developer to the use of co-operative tasks. For example, we can add a single, time-triggered, pre-emptive task to a TTC architecture, to create a "time-triggered hybrid" (TTH) architecture, as illustrated in Figure 3.

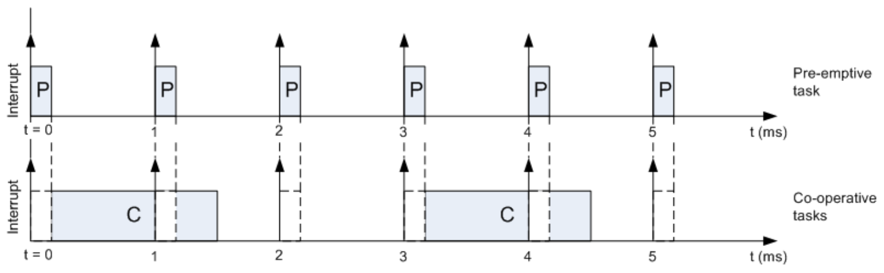


Figure 3: A simple "time-triggered hybrid" architecture with limited support for task pre-emption.

Use of a TTH architecture allows the system designer to create a static schedule made up of (i) a collection of tasks which operate co-operatively and (ii) a single – short - pre-emptive task. In many designs, the pre-emptive task will be used for periodic data acquisition, typically through an analogue-to-digital converter or similar device: such a requirement is common in, for example, a wide range of control systems.

Where a TTH design does not match your requirements, a full "TTP" (time-triggered pre-emptive) RTOS can be employed (see Figure 4). This form of RTOS allows the design to set task priorities and run standard scheduling algorithms (e.g. "rate monotonic" and "deadline monotonic"). However, it still enforces a "one interrupt per CPU" rule and provides very predictable behaviour.

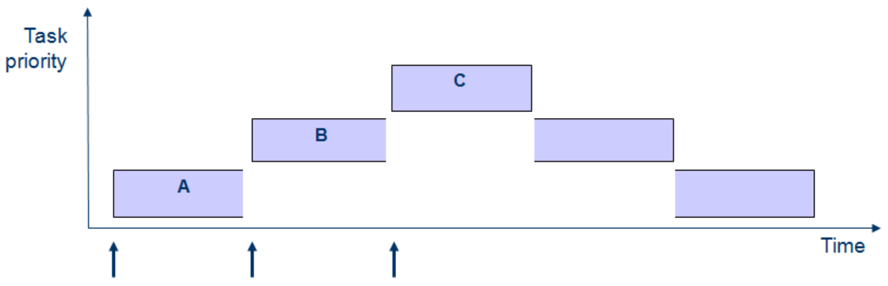


Figure 4: A schematic representation of a TTP RTOS in operation. This form of RTOS provides full support for task priorities and task pre-emption while still providing very predictable timing behaviour.

Dynamic TT architectures: Some examples

In the brief review in the previous section, we looked at the **static TT architecture:** in such designs, we determine when the "ticks" will occur at design time (for example, we determine that the system will be driven by periodic ticks which occur once per millisecond). The second main design option for developers is the **dynamic TT architecture**. In a dynamic TT architecture, we don't know when all of the future ticks will occur but we do know when the next tick will occur. As we will see shortly, even knowing when the next tick will occur can have significant benefits when your system requires real-time performance.

One example of a dynamic TT design is shown in Figure 5. This figure illustrates a situation where we require the system to operate in different states with different task sets in each state. More specifically, the example shown in Figure 5 illustrates an aerospace system in which different tasks are required during take off, level flight and landing manoeuvres.

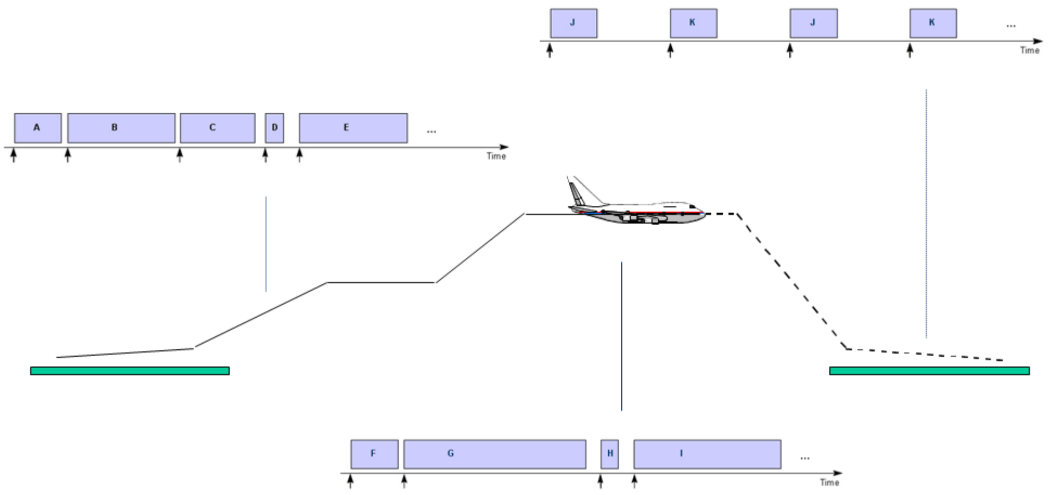


Figure 5: An example of a dynamic TT architecture in which different tasks sets (and different ticks intervals) are required in different system states.

The need for this type of dynamic TT architecture is not restricted to the aerospace industry: for example, a washing machine may require different sets of tasks to run (with different timing behaviour) depending on the wash cycle selected by the user.

Another example of a dynamic TT architecture which can be used (for example) for engine management or for control of a brushless DC motor is shown in Figure 6. In this type of design, the tick rate will depend on environmental factors (for example, the speed of the petrol engine or electric motor). This provides very flexible behaviour.

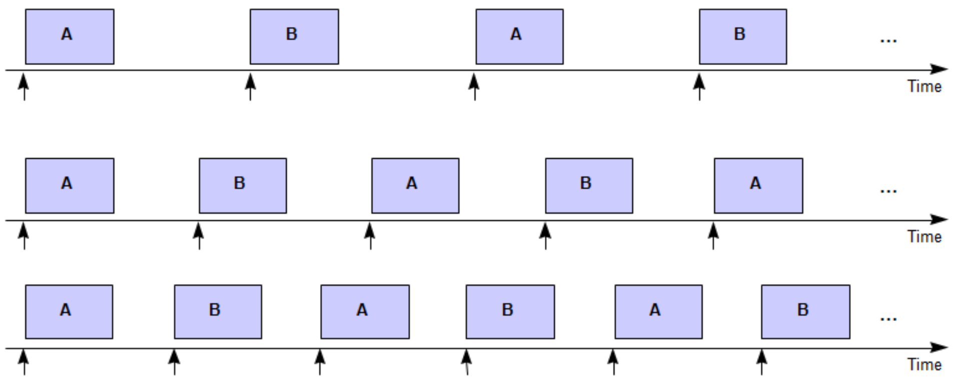


Figure 6: A schematic example of a dynamic TT architecture which can be used (for example) for engine management or for control of a brushless DC motor. In this type of design, the system tick rate will depend on environmental factors (for example, the speed of the petrol engine). Unlike static TT designs, we don't know when all future ticks will occur, but we always know when the next tick will occur.

Should you consider using a TT architecture in your next embedded system?

When we are asked by organisations whether they should be considering use of TT architectures in their systems, we usually start by asking two questions:

- Does the system have hard timing constraints?
- What is the required timing resolution?

When we talk about hard timing constraints, we mean: are there deadlines which the system **must meet** in order to operate correctly. For example, if your system is required to control the speed of a motor in a machine which rotates at up to 1500 rpm, you need to be able to respond to sensor signals from this motor at the maximum operating speed or your design is not fit for purpose: this is a hard timing constraint.

The next question relates to the required timing resolution. In many cases, the required timing resolution is higher than it may appear. For example, consider the signal shown in Figure 7.

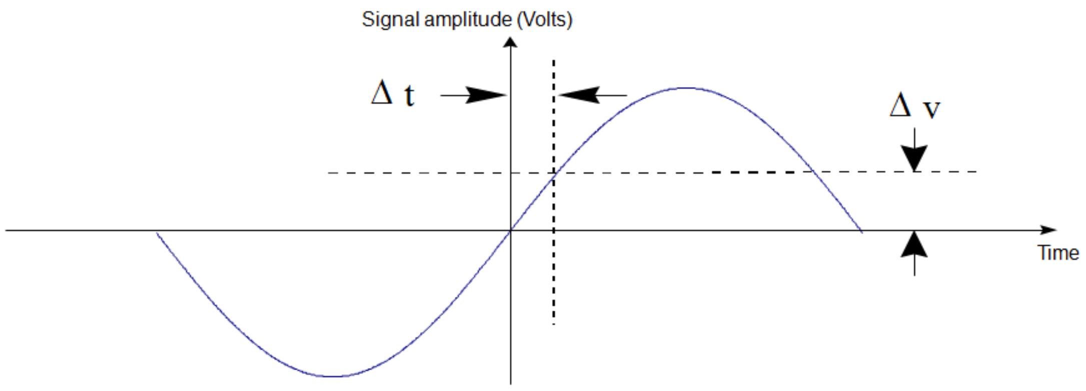


Figure 7: Sampling a signal requires full control over timing behaviour.

Many systems (for example, data acquisition systems, condition monitoring systems, control systems) involve data sampling of the form illustrated in Figure 7. In the example shown, we expect to sample the signal at time t = 0, but – instead – the sampling is delayed by a short time. This results in data corruption in the system.

Suppose (for example) that you are sampling a 500 Hz tone at a sample rate of 1000 Hz, using an 8-bit ADC (a very basic sampling task). If there is a variation (or "jitter") in your sampling time which is greater than 1 µs, you will suffer data loss in the system.

At low levels, the impact of jitter is a reduction in the signal-to-noise level in your system: the effect is the same as using a 7-bit ADC, then a 6-bit, ADC, etc, as the jitter level increases. However, there comes a point when the signal-to-noise level is so great that there is – for practical purposes – no useful information left. As a rule of thumb, the "unusable data" point is reached in a sampled-data system in which the jitter is at a level greater than 10% of the sample interval. For example, when sampling a 500 Hz signal (at 1000 Hz), the data will be unusable at jitter levels of around 100 µs: if we are sampling a 20 kHz signal (at 40 kHz), the data will be unusable if we have 2.5 µs of jitter (and we will lose data quality long before this point is reached).

It is not always appreciated that exactly the same rules apply in system which must generate data at particular points in time: for example, for video playback, speech and music playback and **all forms of control system**. This means that, in a control system (for example), you will find that the quality of the control decreases as the jitter levels increase, up to a point (at around the 10% level) where you may lose control completely.

If you are creating any system involving data sampling or data playback, you need to understand the key timing limits that your system must meet: this determines whether you are creating a 1 ms system, a 100 µs or a 1 µs system. In our experience, creation of systems with a timing resolution of 100 ms or less can be carried out by most experienced developers when using a conventional RTOS solution: when greater timing resolution is required, we suggest that developers consider a TT solution. Where timing resolution of 10 µs (or better) is required, we usually find that it is extremely difficult to meet these requirements with a conventional ET approach and that a TT solution is almost always required.

To illustrate why TT architectures are preferred for many real-time embedded systems where there are precise timing constraints, we can consider some of the complexities which arise when working with a conventional ET design.

Suppose that (as illustrated in Figure 8) we are creating a system with five tasks (Task A, Task B ... Task E) which share some resources (in this case, an analogue-to-digital convertor, a UART, a block of memory containing shared data). We'll further suppose that Task A is a high-priority task and that it must have access to one channel of the ADC every 30 µs to take a data sample. If we build this system using an ET architecture, it can be very difficult to determine whether we will meet this timing constraint, **even if we give Task A the highest priority**.

To understand the difficulty, suppose that Task C (a low-priority task) is accessing the ADC when Task A needs to take the next data sample. Various standard protocols have been developed (such as the Priority Ceiling Protocol) to deal with this situation: these operate by increasing the priority of a task temporarily while it accesses a given resource. This "priority promotion" avoids deadlock situations and ensures that all tasks use the resource "as quickly as possible": however, the timing behaviour which results from the use of these algorithms is very difficult to predict. This means that if we required predictable timing behaviour from an ET design based on Figure 8, we need to be sure that we had tested (and measured the timing) for **all** of the possible combinations of resource conflicts. **This testing can take a great deal of time.** What is more, even small changes to the system (modifying a task or adding a new task) can involve a significant period of re-testing.

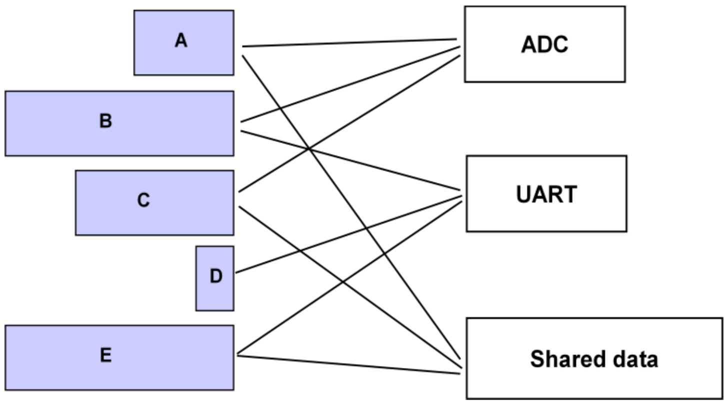


Figure 8: Running multiple tasks in an embedded system where there are "shared resources" (for example, UARTs, ADCs or large arrays of data which must be accessed by more than one task).

If – instead of using an ET architecture - we opted to build the system shown in Figure 8 using a TT architecture, we would observe very different behaviour.

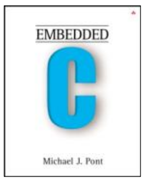
For example, if we used a TTC design, there would be no conflicts over shared resources at any time: this means that all tests can be tested completely in isolation and that changing tasks (or adding additional tasks to a system) does not have unforeseen side effects.

If, instead, we opted to use a TTH or TTP architecture to build this system, we would still have a significant advantage over an ET design in that **we always know when the next "tick" will occur**. What this means is that that we can avoid having low-priority tasks block access to resources. For example, in Figure 8, before Task C accessed the ADC it would check with the task scheduler to see if there was time for it to use this resource before Task A or Task B would next require it: if there is time to access the resource, Task C will do so, otherwise Task C will wait. The end result is that Task A will always have access to the ADC resource (and any other resources it requires) at precisely the time it requires this access. **In short: TT designs don't suffer from priority-inversion problems, and are able to guarantee timing behaviour for high-priority tasks. This ensures high performance (and greatly simplifies testing).**

Conclusions

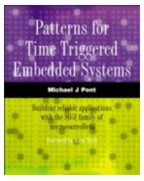
This brief paper has summarised some of the key features of TT embedded systems and explained why use of this system architecture can help to reduce testing times in deeply-embedded real-time systems.

Further reading



"Embedded C" provides an introduction to the creation of simple embedded systems using a TTC architecture.

"EC" is available from a range of bookshops, including Amazon.com.

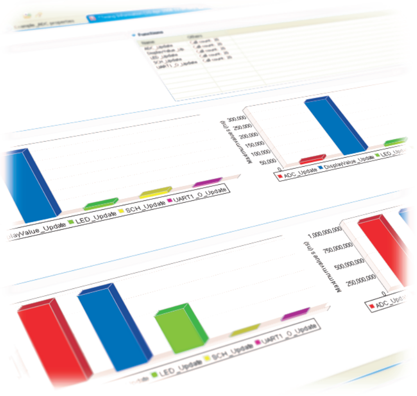


"Patterns for time-triggered embedded systems" demonstrates how embedded systems using TTC and TTH architectures can be created evening using target hardware with very limited resources. This book also describes how to create distributed embedded systems using a TT approach (building on standard communication protocols such as CAN and RS-485).

You can down "PTTES" (free of charge) from the following WWW site: http://www.tte-systems.com/books/pttes

RapidiTTy® development tools

RapidiTTy® development tools deliver "**TT in a box**": they provide everything you need to create reliable **time-triggered** embedded systems. The tool sets include **compilers**, substantial **code libraries**, the InfiniTTy™ operating system (**royalty free** for most customers) and full support for **detailed timing analysis**.



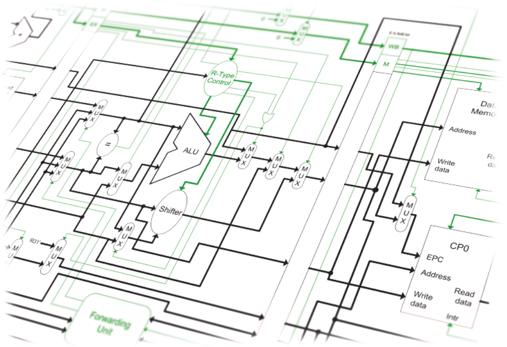
RapidiTTy® development tools help to ensure that even new developers can produce **reliable** systems **very quickly**, and help to **maximise the efficiency**, and **reduce testing costs** for an experienced development team.

The various members of the RapidiTTy® family allow developers to target **microcontrollers** (MCUs), field-programmable gate arrays (**FPGAs**) and 'embedded PC' (**x86**) hardware from a wide range of hardware manufacturers.

If you choose to work with RapidiTTy® Pro, you can migrate designs between targets very easily: for example, you can prototype on an x86 platform and migrate the final design to a microcontroller — or FPGA-based product — within a few minutes. Alternatively, microcontroller-based designs which have "run out of steam" can be "ported up" to an Intel® Atom platform with equal ease.

TTE®32 high-reliability processor cores & microcontrollers and related software toolsets

TTE®32 processor cores and related microcontrollers are aimed at sectors - including medical, industrial, automotive and aerospace - where their combination of high performance and very predictable behaviour greatly simplifies design, test and certification activities.



TTE®32 processor cores are based on a 32-bit design with 32-registers and a five-stage pipeline. They have a Harvard architecture and support precise exceptions. As they have been designed as a matched hardware platform for high-reliability systems which employ a time-triggered software architecture, the cores support one active interrupt. They have a constant interrupt overhead (even during multi-cycle operations) and provide guaranteed memory-access and instruction-execution times.

TTE®32 processor cores are compatible with the MIPS®-1 Instruction Set Architecture (ISA).

We also develop custom processors and microcontrollers for our customers. Custom designs may be based on an existing ISA, or a more specialised solution. Compilers and related toolsets can also be provided to match your target.

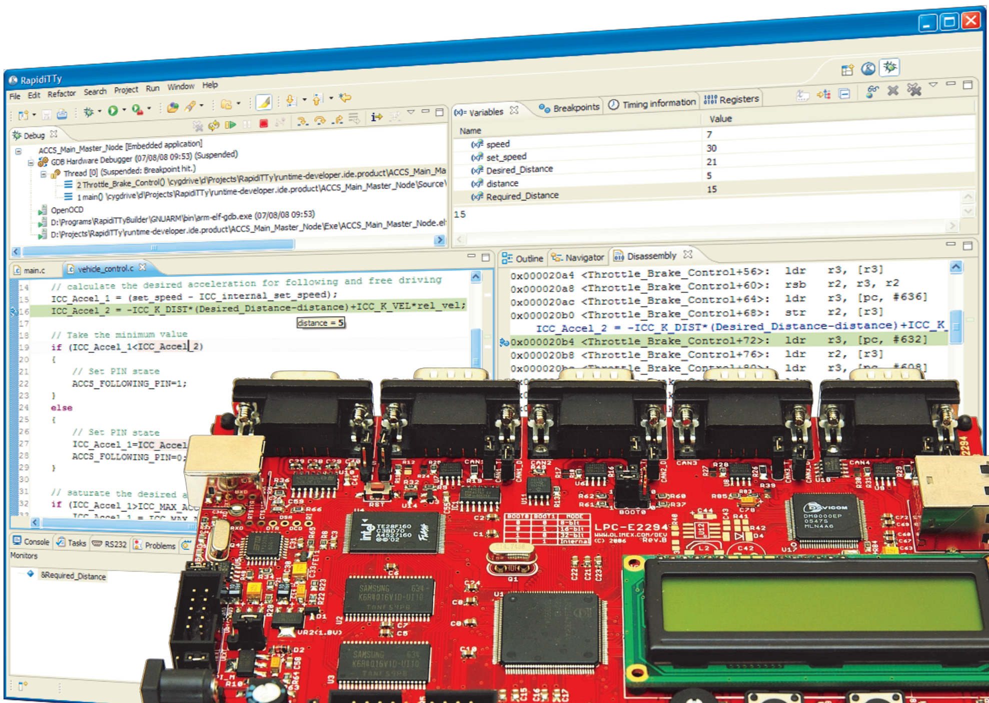
TTE Systems

TTE Systems Ltd
106 New Walk
Leicester
UK

Tel: +44 (0)116 223 1684
Fax: +44 (0)116 223 1651

www.tte-systems.com
info@tte-systems.com

Company Number: 05058157
VAT Number: 913 4859 12



```
ACCS_Main_Master_Node [Embedded application]
  GDB Hardware Debugger (07/08/08 09:53) (Suspended)
    Thread [0] (Suspended: Breakpoint hit.)
      2 Throttle_Brake_Control() \cygdrive\d\Projects\RapidiTTy\runtime-developer.ide.product\ACCS_Main_Ma
      1 main() \cygdrive\d\Projects\RapidiTTy\runtime-developer.ide.product\ACCS_Main_Master_Node\Source\
  OpenOCD
  D:\Programs\RapidiTTyBuilder\GNUARM\bin\arm-elf-gdb.exe (07/08/08 09:53)
  D:\Projects\RapidiTTy\runtime-developer.ide.product\ACCS_Main_Master_Node\Exe\ACCS_Main_Master_Node.el
```

Name	Value
speed	7
set_speed	30
Desired_Distance	21
distance	5
Required_Distance	15

```
// calculate the desired acceleration for following and free driving
ICC_Accel_1 = (set_speed - ICC_internal_set_speed);
ICC_Accel_2 = -ICC_K_DIST*(Desired_Distance-distance)+ICC_K_VEL*rel_vel;
                                        distance = 5

// Take the minimum value
if (ICC_Accel_1<ICC_Accel_2)
{
    // Set PIN state
    ACCS_FOLLOWING_PIN=1;
}
else
{
    // Set PIN state
    ICC_Accel_1=ICC_Accel
    ACCS_FOLLOWING_PIN=0;
}

// saturate the desired a
if (ICC_Accel_1>ICC_MAX_AC
```

```
0x000020a4 <Throttle_Brake_Control+56>:  ldr   r3, [r3]
0x000020a8 <Throttle_Brake_Control+60>:  rsb   r2, r3, r2
0x000020ac <Throttle_Brake_Control+64>:  ldr   r3, [pc, #636]
0x000020b0 <Throttle_Brake_Control+68>:  str   r2, [r3]
  ICC_Accel_2 = -ICC_K_DIST*(Desired_Distance-distance)+ICC_K_
0x000020b4 <Throttle_Brake_Control+72>:  ldr   r3, [pc, #632]
0x000020b8 <Throttle_Brake_Control+76>:  ldr   r2, [r3]
```