WHITE PAPER

Software Design Considerations for a successful ARM Cortex-M based Microcontroller project

Microcontroller devices and applications based upon ARM's Cortex™-M processor family are becoming the de facto choice for modern embedded systems. The complexity and functionality of these modern designs is increasing at an incredible rate, with an expanding portfolio of Cortex-M based microcontrollers that are rapidly making the traditional 8/16/32-bit processor classification obsolete.

Today, there is increasing pressure on development teams to achieve a high quality design that meets the performance/price requirements and on time. This reinforces the importance of making the correct choices on technology early in the development cycle.

The traditional method of selecting a processor that fits the immediate performance requirements and then choosing the development tools and software in isolation invariably introduces issues related to the integration, validation and reusability of the design. These issues can place additional pressure on an already tight development time-line.

This paper will highlight design considerations. It will focus on how carefully researching and selecting key technology components, at the design start, will enable the development team to achieve its goals, and enable organisation to provide a platform for future designs and developments.

This paper will look at the process from a new design requirement perspective.

MCU design - Typical decision process

A project manager about to start a system development using a Cortex-M class processor will need to select a range of key technologies to enable the delivery of a product.

Microcontroller selection

Choosing the Microcontroller is ordinarily the first decision made, followed by either the development tools or the embedded software technology.
There are many considerations concerning the selection of an appropriate microcontroller including:

- What performance is required for the target application?
- What on-chip peripherals are supported and what will need to be implemented at board level to support the required application communications and functionality?
- Commercial considerations
    - What is the price?
    - Is it single source?

After making an informed choice on the viable microcontroller for the design, the project manager will look at the available development tools. These tools will typically include an IDE, compiler, simulator, debugger and any performance/test analysis tools.

When choosing the development environment and tools, there are a number of factors that should be considered:

- Do they provide the best performance in terms of speed and code size?
- Are the tools intuitive, easy to use and well documented?
- Is the tools vendor aligned with the microcontroller manufacturer's roadmap?
    - Will the compiler and debugger support future extension and enhancements to the microcontroller?
- What is the price?

The final major technology decision to be made relates to the embedded software and middleware choice. The middleware components may include Real-time

Operating System (RTOS), Networking stacks, Communications stacks, device drivers and application protocols for example.

When selecting the embedded software components there are again some requirements that need to be accessed.

- Do they provide the functionality and performance required to meet the application requirements?
- Have the components been optimized for the chosen microcontroller?
    - Do the software components support the on-chip peripherals?
- Do the individual components work seamlessly together and are they tightly integrated with the development tools?
- Is the middleware provider aligned with the tools and microcontrollers suppliers roadmap?
- What is the price?

Very often the choice of embedded software is made prior to or at the same time as the selection of the development tools.

Let's expand on the area of microcontroller selection and more specifically look at the range of devices available in the market based upon ARM's Cortex-M core technology and why the adoption in the market has been so significant.

ARM currently has four Cortex-M microcontroller cores and these have been widely licensed in the market by major semiconductor companies including NXP, ST Microelectronics, TI, and Freescale to name a few.

The adoption of these core technologies by the silicon companies has been very rapid, as their end customers, demand more features, higher performance, more connectivity and lower power, and of course, a lower price.

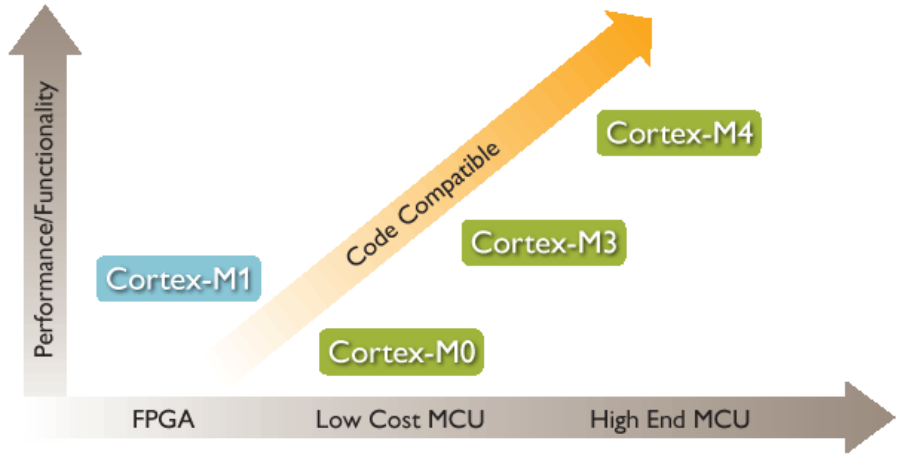

Figure 1. Cortex-M family

The Cortex-M range of microcontroller cores were developed to address increasing requirement demands from customers that could not be met by existing 8/16-bit MCU's.

The original design brief for the Cortex-M range was to implement a 32-bit architecture that could be scaled to address the extremely low power and low cost demands that were traditionally served by the proprietary 8-bit microcontrollers. Add to this the need to increase performance to enable customers to provide additional features and connectivity that was typically only available to the high-end, high cost proprietary 32-bit microcontrollers on the market.

In addition to these requirements, it was important make the family of cores both binary and tool compatible, whilst implementing revolutionary functionality to ease application development and debugging.

This resulted in the Cortex-M family of cores (shown in figure 1)

There are a rapidly expanding number of devices on the market that are based on the Cortex-M technology. These vary from M0 based ultra low power devices priced at less than $0.65 to M4 based devices delivering exceptional performance, DSP extensions and with a vast array of peripheral and communications support.

This breadth of coverage has effectively made the 8/16/32-bit classification redundant. The development manager is now focussed on design flow and how to meet the development schedule.

As the complexity of microcontroller designs increases so too does the need for advanced debug capabilities within the microcontroller.

The Cortex-M4 processor for instance, features DSP and floating point functionality that enables ever more complex and high-performance applications to be developed. Developing these complex applications requires an equally powerful debug and verification capability. Many markets are now demanding formal software verification or certification.

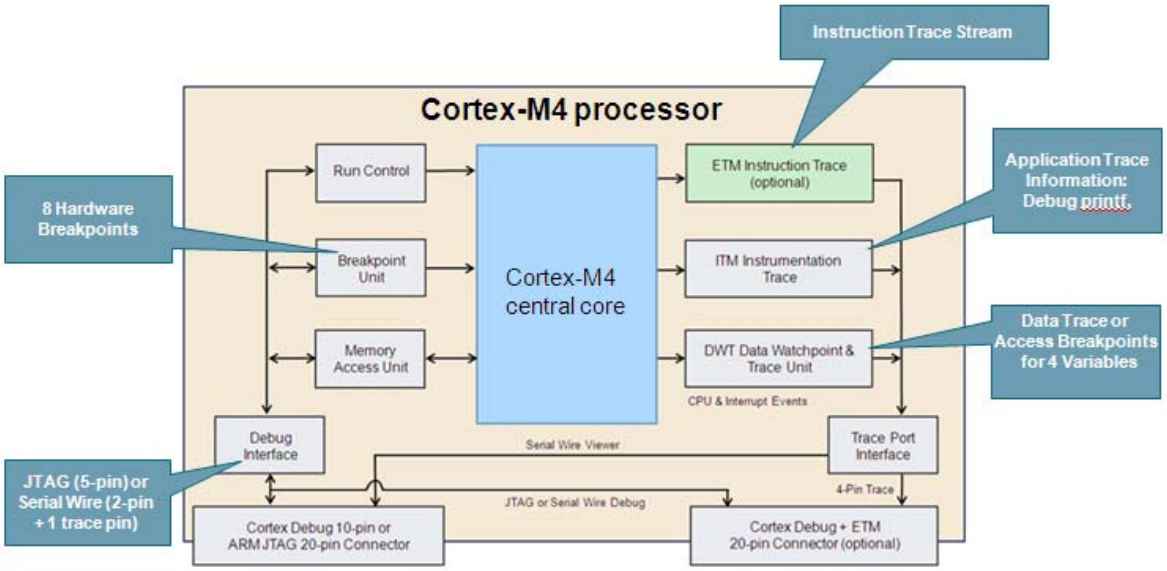

Figure 2. CoreSight™ Debug Technology

All Cortex-M processor-based devices feature the ARM CoreSight™ technology. CoreSight has introduced advanced debug and trace capabilities designed to enable the developer to analyze, optimize, and verify program execution with minimum effort and cost. CoreSight also delivers major benefits to the embedded developer, benefits which other MCU architectures do not have.

Traditionally, debugging on MCU devices has been based on run-stop technology; However, this has always presented the developer with serious disadvantages. To set breakpoints or access memory and variables the processor must be halted. This

obviously changes the status and behaviour of the system meaning that detailed timing analysis cannot be performed.
This method also creates real practical problems; communications interfaces, such as USB, can timeout due to the loss of handshaking, and when used to debug a motor control system, it could make the motor stop in a high current state potentially destroying the motor or other parts of the hardware system.

CoreSight delivers simple solutions to these challenges. The CoreSight debug blocks (shown in figure 2), work tightly with the Cortex-M4 CPU core, allowing you to control the CPU and single-step lines of source or assembler code. Additionally, you are able to set up to 8 breakpoints and read/write memory and peripheral registers while the processor is running at full-speed. As the processor does not need to be stopped actual system behaviour can be analysed and debugged.

As microcontroller hardware designs increase in functionality we see an exponential increase in the software complexity and costs (see figure 3.).

Issues that drive software costs include the need for increasing product functionality and requirements, which are typically implemented in software. This is exacerbated by the fact that hardware problems tend to be compensated by software.

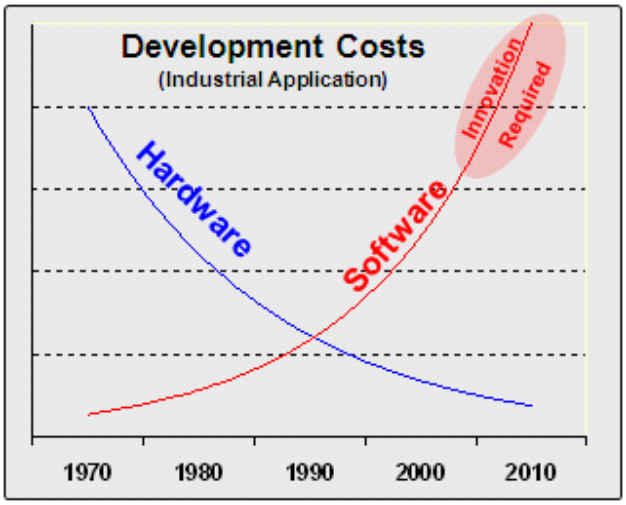

Figure 3. Hardware vs Software development costs.

Many microcontroller architectures have a long history, with the majority of 8 and 16-bit architectures being invented more than 20 years ago. Over the years, these architectures have been re-shaped several times to meet the requirements of today's demanding applications.

Since no peripheral and interface standards exist, programmers must invent solutions over and over again for the same basic problems, whilst adapting existing software algorithms to new hardware. In such environments object-oriented programming can rarely be used. Generic software components that are common in the PC world are not available, and the lack of programming standards limits software reuse. Instead, silicon vendors must provide free software frameworks for new devices that are tailored towards specific applications. This slows down the introduction of new devices and significantly increases the development costs.

On the other hand hardware components within a design have been relatively easily exchanged from design to design. Until now software components and tool-chains have not been easily exchangeable.

Introduction to Cortex Microcontroller Software Interface Standard (CMSIS

The **Cortex Microcontroller Software Interface Standard (CMSIS)** addresses the challenges faced when various software components are deployed to the actual physical processor. The CMSIS will be expanded to include future Cortex-M processor cores. It is defined in close co-operation with various silicon vendors. For wide acceptance in the industry software vendors such as IAR, Keil, Micrium, Segger, and Tasking are also involved. This collaboration has resulted in an easy-to-use and easy-to-learn programming interface. CMSIS provides a common approach for interfacing to peripherals, RTOSs, and middleware components. The CMSIS is compatible with several compiler implementations, including GCC, and provides the following two software layers that are (shown in orange in figure 4.):

- **Peripheral Access Layer (CMSIS-PAL):** contains name definitions, address definitions and helper functions to access processor core registers and device peripherals. It introduces a consistent way to access core peripherals. This also defines a device independent interface for an RTOS kernel and data trace

channels for RTOS kernel-awareness in debuggers and simple *printf*-style debugging.

- **Middleware Access Layer (CMSIS-MAL):** provides common methods to access peripherals for the software industry. The Middleware Access Layer is adapted by the silicon vendor for the device specific peripherals used by more complex middleware components such as communication stacks.

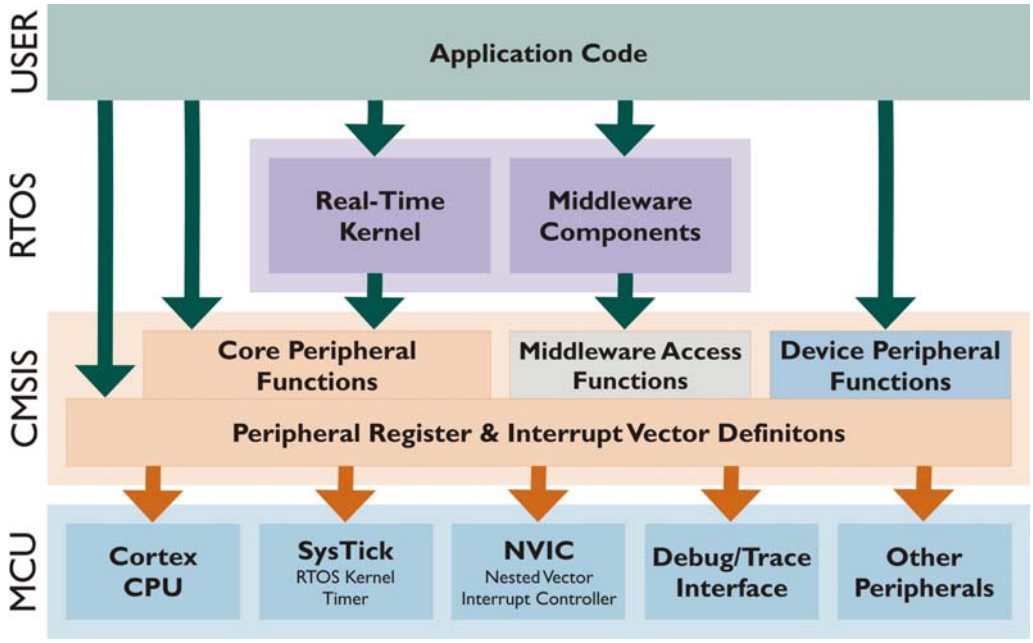

Figure 4. Structure of CMSIS

Significant advancements have been made by ARM to structure a microcontroller interface software standard that gives the performance and functionality to service today's MCU requirements, and at the same time provides the framework and structure for customers to meet future needs.

Selecting the development tools

Having a tool-chain that can provide the best-in-class performance and support the full range of selected microcontrollers and their feature set is vital.

To meet the ever reducing development timeline the developer must have a verification environment that enables them to simulate, debug, test and optimise the application and peripherals throughout the development phase.

The Coresight technology is extremely powerful and having a software development environment that can utilize it, is key. The tight coupling of Coresight and verification environment enables the developer to carry out detailed analysis of the design including code coverage, execution profiling, and performance analysis. This helps the developer to quickly identify and fix bottlenecks within the design.

A powerful compiler, simulator, and debugger is of limited use here, if it is too difficult to use and poorly documented.

Keil's Microcontroller Development Kit for ARM (MDK-ARM) provides this functionality in one package, and includes the µVision4 IDE (as shown in figure 5.) that integrates these key components, provides project management, target configuration tools, source code editor and flash programming capabilities into a single intuitive graphical design environment.

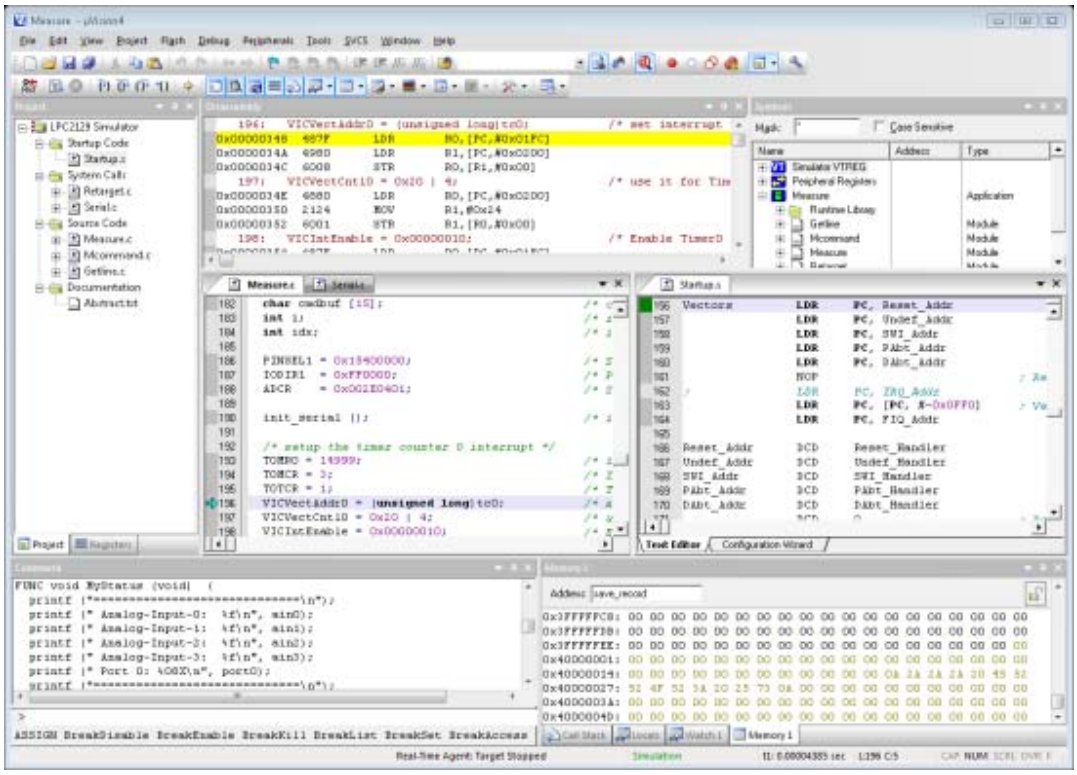

Figure 5. Keil's µVision4 IDE

Selecting the middleware

Most microcontroller-based embedded systems are real-time systems that have to respond within predefined time limits to external events. This is a key constraint in the both the design of the hardware and software of these systems.

Thanks to the performance and functionality of 32-bit microcontrollers, real-time embedded systems often have a single microcontroller device. This enables low unit cost, but creates challenges for software developers, as a single application must respond timely to a larger number of increasingly complex peripherals.

Today's embedded systems are expected to handle many tasks simultaneously whilst monitoring various interrupts with varying priority and be able to accept a wide variety of interface standards.

This presents software developers with real challenges; if they are to deliver the system on time, with the expected performance and the ability to expand with future market requirements and performance demands. These challenges increase when the system is also required to communicate efficiently with other systems, remote applications or via the internet. Software developers have typically taken one of two approaches to implementing their embedded software applications; the “Super-Loop” and the RTOS.

Simple embedded systems typically use a Super-Loop concept where the application executes each function in a fixed order. Interrupt Service Routines (ISR) are used for time-critical program portions and communication is via global variables; not data communication protocol.

- There are some significant disadvantages of the super-loop approach:
- Time-critical operations must be processed within ISRs and this means that ISR functions get complex and require long execution times, also ISR nesting may create unpredictable execution time & stack loads.
- Super-Loop can be synchronized with the System timer, but, if system requires several different cycle times, it is hard to implement.
- Split of time-consuming functions can exceed Super-Loop cycle and creates software overhead making the application hard to understand.
- Super-Loop applications can easily become complex, hard to extend and a simple change can have unpredictable side effects. This makes re-use of application code and migration to another processor difficult.
- Applications also become difficult and time consuming to analyze due to lack of analyzing features.

The super-loop approach is well suited to small systems but has significant limitations for more complex applications.

While it is possible to implement an embedded program without using an RTOS, a proven RTOS like Keil RTX enables developers to save time, produce a reliable, expandable system and makes software development easier.

Using a RTOS allows developers to concentrate on application development rather than managing system resources, scheduling tasks and other general system 'house keeping', all of which are becoming increasing difficult when viewed in a system as a whole.
An RTOS provides an abstraction layer between the application software and the hardware, allowing for future hardware upgrades and redesigns with minimum software redesign. Using an RTOS also enables a logical partitioning of the application and hence more easily supports development across multiple developers.

Having a powerful and flexible RTOS that can meet the design requirement is a must, as is having it tightly integrated with the tool-chain.

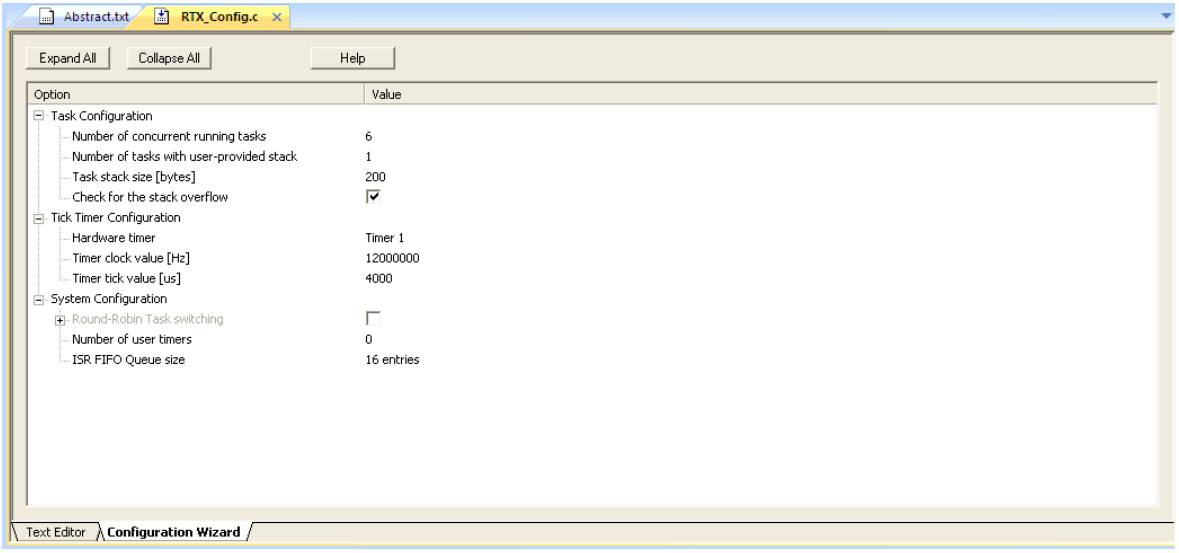

Figure 6. RTX Configuration wizard

RTX is closely integrated within the Keil MDK-ARM and includes graphical configuration wizards (see figure 6.) for configuring all main parameters.

Tight integration between RTOS and debugger gives the developer detailed information on the activities running within the RTOS and present it for further analysis. This functionality is often referred to as Kernel (RTOS) aware debugging or task aware debugging. Kernel awareness enables the developer to analyze the tasks and events running within the RTOS and provide detailed information on the resource loading and the stack usage (see figure 7.), thus helping the developer to identify defects in the code, bottlenecks in the design, and optimize the application for performance.

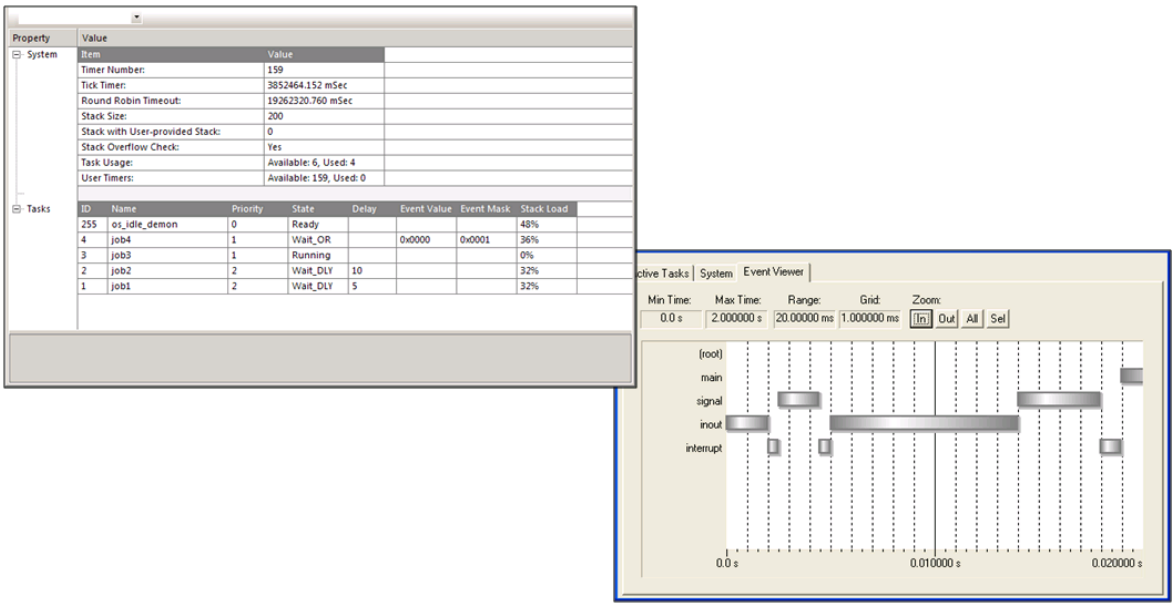

Figure 7. µVision4 Kernel Aware debug and Event Viewer windows

Almost all embedded applications are required to communicate with other systems or peripherals; many can be handled with straightforward serial interfaces such as USB and CAN. Other systems may require concurrent device and peripheral communication which cannot be handled via these types of serial interfaces, in these cases a well documented, mature, high performance interface with wide device support would be the ideal solution. The well known TCP/IP protocol meets this requirement perfectly. It is also essential if the system is to interface to a wider network or the internet.

However implementing a TCP/IP based system can be a daunting prospect for the embedded developer new to networking applications, where do they get a suitable TCP/IP stack, how do they implement it, how do they configure all of the complex peripherals, will it work with the other system components?
In addition to the implementation of the networking protocols there is typically a requirement to implement application level protocols such as HTTP, Telnet, FTP and SMTP etc.

Configuring of networking parameters and the presentation of debug information whilst the system is running is vital for the developer if they are to quickly implement and test a complex networked application.

Keil's Real-Time library (RL-ARM) for instance, includes the RTX RTOS and all of the commonly used components required to implement complex communications or networking protocols and functionality into an embedded system. Configuration wizards are included; enabling all networking parameters to be easily accessed and configured. RL-ARM is tightly integrated with the debugger and gives the developer full visibility of all system networking activities and providing all the information required to verify correct operation of the networking interface with other system software components.

Summary

The increasing demand placed on development teams to implement new functionality and increase performance of their embedded system design, whilst meeting ever decreasing time schedules and without compromising quality, means that the "old way" of selecting the key technology components in isolation can have some significant pitfalls and risks associated.

Development nearly always takes longer than you think!

- Finding and fixing problems in hardware and software, who to call for support!
- Integrating components together and bridging gaps
- Identifying defects and bottlenecks in design
- Project deadlines and QA time can be pressured

Keeping things simple is “A Good Thing”

- Less unknown variables simplifies integration
- Re-use of robust code reduces risk of errors
- Learning new skills as you build a product rarely provides optimal results
- Focus on the core of your application

The combination of an ARM Cortex-M based MCU device, Keil Microcontroller development kit (MDK-ARM) and Keil Real-Time Library (RL-ARM) provides the developer with the best in class technology, all seamlessly integrated together.

This greatly reduces the risk and time associated with integration, and allows the development team to focus on the key values of application development and product innovation.