CODE TIME TECHNOLOGIES

# Optimized Interrupt Handling

Whitepaper

**Copyright Information**
This document is copyright Code Time Technologies Inc. ©2012. All rights reserved. Code Time Technologies Inc. may have patents or pending applications covering the subject matter in this document. The furnishing of this document does not give you any license to these patents.

**Disclaimer**
ARM and Cortex are registered trademarks of ARM Limited. All other trademarks are the property of their respective owners.

## Introduction

In embedded computing, an interrupt is an asynchronous signal indicating the need for attention, and is a mechanism used to avoid wasting the processor's valuable time in polling loops, waiting for external events.  Modern microcontrollers typically support dozens or even hundreds of interrupt sources.  As such, it is critical that interrupt handling occur in a timely, and memory and processor efficient manner.

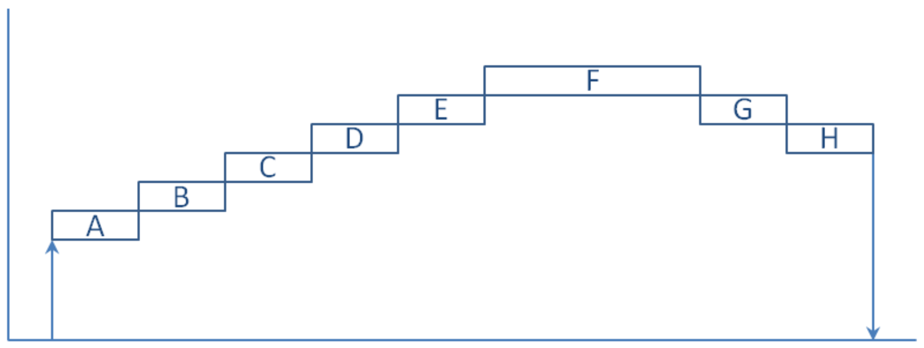
This paper examines the problem of handling interrupts in a RTOS based application in more depth, and details the available remedies, including those unique to the Abassi real-time kernel.

## Interrupt Latency

In real-time systems, interrupt latency is the time between the validation of an interrupt signal and the servicing of that interrupt by the interrupt handler.  However, what exactly constitutes this duration varies between real-time operating system vendors, and frequently ignores or deemphasizes their weaknesses.

Truly, there is a single, complete definition for interrupt latency, which contains multiple components.  Some of these components are present in all kernels, and others are present in only a subset.  However, to intelligently compare different real-time kernels, all components need to be understood and analyzed.

The best way to analyze interrupt latency is to consider all aspects of entering an interrupt handler:



A = Interrupt controller reaction time
B = Maximum time RTOS disables interrupts
C = Processor interrupt setup
D = RTOS Interrupt dispatcher
E = RTOS Interrupt prologue
F = User interrupt handler
G = RTOS Interrupt epilog
H = Processor interrupt return

**Figure 1. Interrupt latency components**

For all real-time kernels on a given platform, Time "A" is the same.  This is because it is a function of the underlying interrupt controller hardware, and is not affected by the RTOS.

Most real-time kernels disable interrupts during critical sections (Time "B"), such as updating their internal structures. If this occurs during the time that an interrupt arrives, the interrupt will be delayed until the interrupts are re-enabled. This can range from a few tens to many hundreds of CPU cycles, depending on the RTOS architecture.

On processors that automatically push (Time "C") and pop (Time "H") registers at interrupt entry and exit, there will be an overhead that is a function of the processor itself, and is equal for all real-time kernels.

Some real-time operating systems utilize an interrupt dispatcher (Time "D") for forwarding interrupts to the appropriate interrupt handler. Using a dispatcher eliminates the need for each interrupt handler to instantiate interrupt prologue and epilogue code, used to interact with the RTOS, resulting in significant program memory savings. A dispatcher can also eliminate restrictions on the RTOS services available within an interrupt handler, and can allow interrupt handlers to be standard 'C' functions, for improved code portability.

Real-time kernels that do not employ an interrupt dispatcher usually require the addition of interrupt prologue (Time "E") and epilog code (Time "G") to all interrupt handlers, to facilitate the use of RTOS services within an interrupt handler. If such code is not required, then strict limitations usually exist on which kernel services are available within an interrupt handler, and such services are usually accessed through a unique API.

The interrupt handler itself (Time "F") is also affected by the RTOS architecture and efficiency. Whenever an interrupt handler utilizes a RTOS service, this lengthens the time the system remains in an interrupt context, and delays the processing of additional interrupts. The more efficient the RTOS implementation, the less time the interrupt handler requires, and the higher the rate of interrupts that can be processed; and more time available for the application itself.

## Interrupt Stack Usage

When an interrupt occurs, the system halts whatever processing is happening such that the interrupt can be serviced. Since, after the interrupt handler has executed, the system must return to its original state, it is necessary to save and restore the processor registers modified during interrupt handling. And, if the interrupt handler utilizes a RTOS service that causes a task switch to occur, the entire register context must be saved and replaced with that of the new task. Depending on the processor, this can require significant data memory and processing time.

In a preemptive multitasking system, two possibilities exist for where the registers used during interrupt handling can be temporarily preserved: the current task stack; or, a dedicated interrupt stack.

If the current task stack is used, then all tasks must be dimensioned to reserve enough room to handle the worst-case interrupt handler memory usage. This becomes much more of an issue if interrupt nesting can occur. Nesting allows higher priority interrupts to interrupt lower priority ones, and increases the cumulative per task memory requirements.

For example, consider a system with 8 tasks, on a platform that supports nested interrupts. Further assume that the individual interrupt handlers require 100 words of memory each, and that up to 4 levels of interrupt nesting can occur. This necessitates allocating 4*100 words of additional stack to each of the 8 tasks, requiring a total of 3,200 words of RAM.

By using a dedicated interrupt stack, only a single 400 word interrupt stack is required, saving 2,800 words of RAM.

Even more important is the safety and testability improvements provided by the dedicated interrupt stack. By using a dedicated interrupt stack, it is much easier to prove that it has been properly dimensioned, versus needing to validate every possible interrupt/task interaction. The dedicated interrupt stack helps eliminate the random failures and crashes seen when a task stack is incorrectly dimensioned and "overflows" due to an interrupt.

## Abassi Advantages

Continuing the practice of high performance without sacrificing memory and processing overhead, the Abassi kernel implements a hybrid interrupt stack, and highly optimized kernel service handling, to minimize the duration spent in an interrupt context. This greatly improves interrupt handling response time, while keeping stack memory usage to a minimum.

### Hybrid Interrupt Stack

Abassi improves upon the traditional dedicated interrupt stack by preserving the first level interrupt registers on the current task stack, and then switching to the use of the dedicated interrupt stack. The reasoning behind this is that most meaningful interrupt handlers access some kernel service, which may trigger a task switch. If the dedicated interrupt stack was used to preserve the first level interrupt registers also, and a task switch was triggered, this would necessitate the copying of the interrupt registers to the pre-empted tasks stack. By utilizing a hybrid interrupt stack, and requiring the additional of only a trivial amount of space on each task stack, task switch time is optimized.

### Optimized Kernel Service Handling

In order to minimize the time spent in an interrupt context, and maximize the rate interrupts can be processed and the time available for the application itself, Abassi optimizes access to its kernel services. By deferring handling until the interrupt context is exited, additional pending interrupts are not blocked, and task context switches are kept to a strict minimum, without any application modifications. Abassi is also architected such that the kernel does not disable the interrupts (Time "B" is zero), and does not require an interrupt prologue (Time "E") or interrupt epilog (Time "G").

Consider two cases: a single interrupt being handled; and, a second interrupt arriving during the time another is being handled (non-nested). In both cases, the interrupt handlers access kernel services that would result in a task context switch.

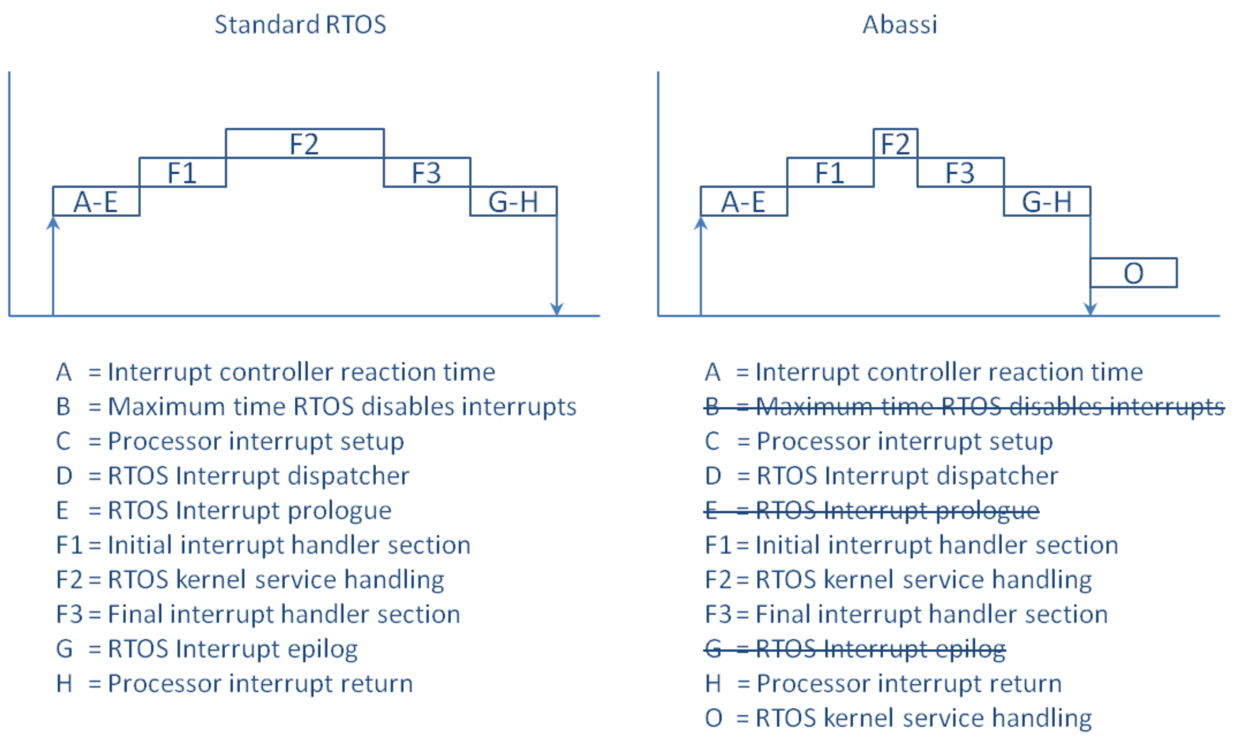
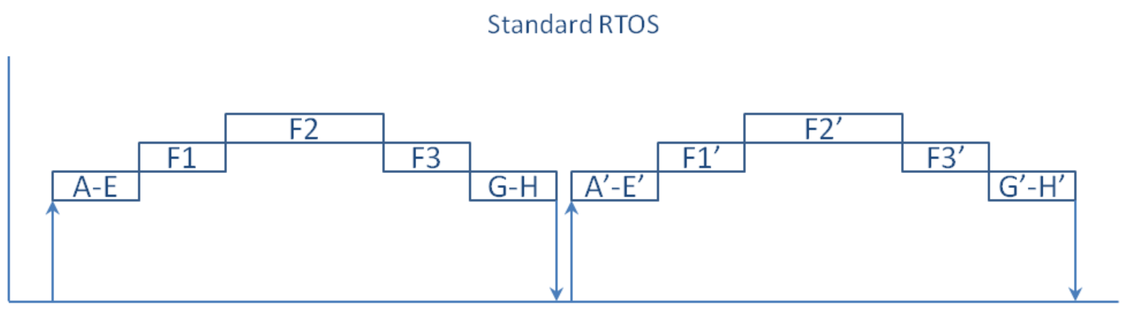

Standard RTOS
A-E
F1
F2
F3
G-H
A = Interrupt controller reaction time
B = Maximum time RTOS disables interrupts
C = Processor interrupt setup
D = RTOS Interrupt dispatcher
E = RTOS Interrupt prologue
F1 = Initial interrupt handler section
F2 = RTOS kernel service handling
F3 = Final interrupt handler section
G = RTOS Interrupt epilog
H = Processor interrupt return
Abassi
A-E
F1
F2
F3
G-H
O
A = Interrupt controller reaction time
~~B = Maximum time RTOS disables interrupts~~
C = Processor interrupt setup
D = RTOS Interrupt dispatcher
~~E = RTOS Interrupt prologue~~
F1 = Initial interrupt handler section
F2 = RTOS kernel service handling
F3 = Final interrupt handler section
~~G = RTOS Interrupt epilog~~
H = Processor interrupt return
O = RTOS kernel service handling

**Figure 2. Single interrupt handling**

In the single interrupt case, a standard RTOS will perform the entire kernel service handling (Time "F*x*") within the interrupt context, including any task context switch. However, Abassi defers the kernel service handling for outside the interrupt context (Time "O"). In this scenario, the *total* time to handle both the interrupt and the kernel service is not decreased, but the time in an *interrupt context* is greatly decreased. This improves the interrupt response time, since additional pending interrupts are not blocked for longer than necessary.

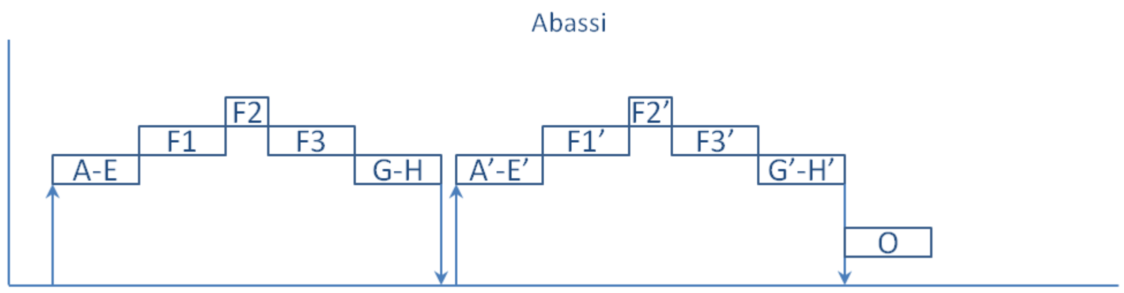
Even though the handling of kernel requests is deferred until after the interrupt handler has exited, this does not translate into a delay in task switching. Since the system is servicing an interrupt, the currently running task is already suspended, and a newly running task would only resume once the interrupt handler was exited. By deferring the task switch itself until the interrupt handler exits, the total time that the application itself is halted remains the same or improves.

Deferring the task switch until the main interrupt handler has completed is comparable to the mechanism many real-time kernels use when operating on an ARM Cortex processor. The Cortex processor provides access to a `PendSV` exception, which is serviced by a low priority exception handler. This allows the RTOS kernel access to be separated into two parts: kernel service handler; and, task switch. Abassi improves upon this by not only deferring the task switch until the main interrupt handler has exited, but by also deferring the bulk of the kernel service handling itself. And, more importantly, this optimization is available on all platforms Abassi supports, not just the Cortex family of processors.



A = Interrupt controller reaction time
B = Maximum time RTOS disables interrupts
C = Processor interrupt setup
D = RTOS Interrupt dispatcher
E = RTOS Interrupt prologue

F1 = Initial interrupt handler section
F2 = RTOS kernel service handling
F3 = Final interrupt handler section
G = RTOS Interrupt epilog
H = Processor interrupt return



A = Interrupt controller reaction time
~~B = Maximum time RTOS disables interrupts~~
C = Processor interrupt setup
D = RTOS Interrupt dispatcher
~~E = RTOS Interrupt prologue~~

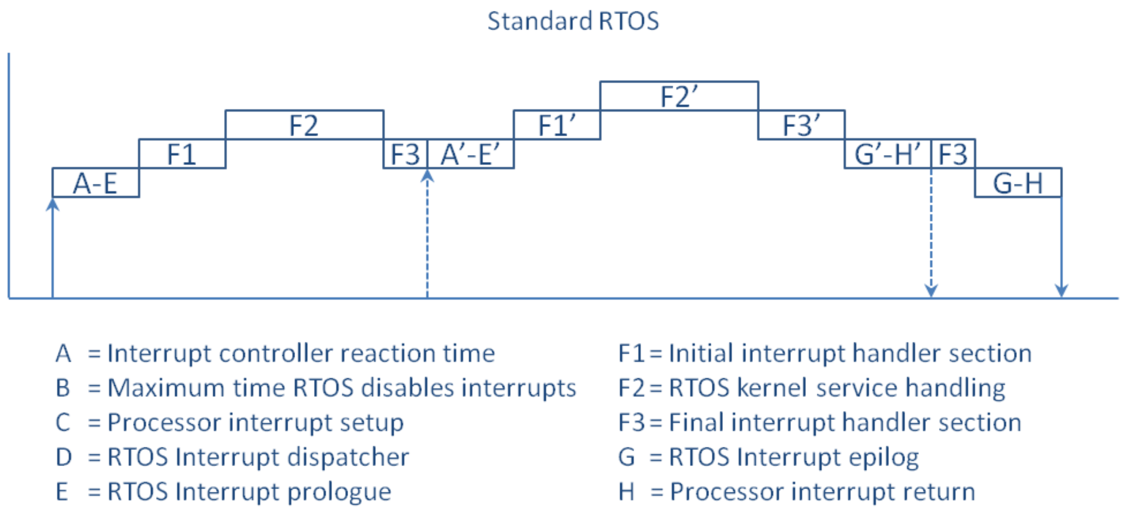
F1 = Initial interrupt handler section
F2 = RTOS kernel service handling
F3 = Final interrupt handler section
~~G = RTOS Interrupt epilog~~
H = Processor interrupt return
O = RTOS kernel service handling

**Figure 3. Multiple interrupt handling**

In a multiple interrupt scenario, a standard RTOS simply repeats the same process it does for the single interrupt case. That is, it will perform the entire kernel service handling (Time "F*x*") within the interrupt context, including any task context switches. This can result in "wasted" task switches, if the second interrupt causes a task switch which supersedes the one initiated by the first interrupt. In addition, entering the kernel for each interrupt multiplies the overhead of reading and updating the kernel variables, task scheduling, etc.

By deferring the kernel service handling for outside the interrupt context, Abassi optimizes performance by only accessing the kernel and its variables once, and minimizes scheduling overhead by only performing a single task switch once all kernel requests are handled. This minimizes both the time in an interrupt context, and the *total* time handling the interrupt and kernel access, resulting in improved overall system responsiveness and efficiency. And, again, Abassi is architected such that the kernel does not disable the interrupts (Time “B” is zero), and does not require an interrupt prologue (Time “E”) or interrupt epilog (Time “G”).

The same applies in the case of nested interrupts. A standard RTOS simply repeats the same process it does for the single interrupt case, potentially resulting in wasted task switches. However, Abassi minimizes both the time in an interrupt context, and the total time handling the interrupt and kernel access, by deferring the kernel service handling.


Standard RTOS
F2’
F2
F1’
F3’
F1
F3
A’-E’
G’-H’
F3
A-E
G-H
A = Interrupt controller reaction time
B = Maximum time RTOS disables interrupts
C = Processor interrupt setup
D = RTOS Interrupt dispatcher
E = RTOS Interrupt prologue
F1 = Initial interrupt handler section
F2 = RTOS kernel service handling
F3 = Final interrupt handler section
G = RTOS Interrupt epilog
H = Processor interrupt return

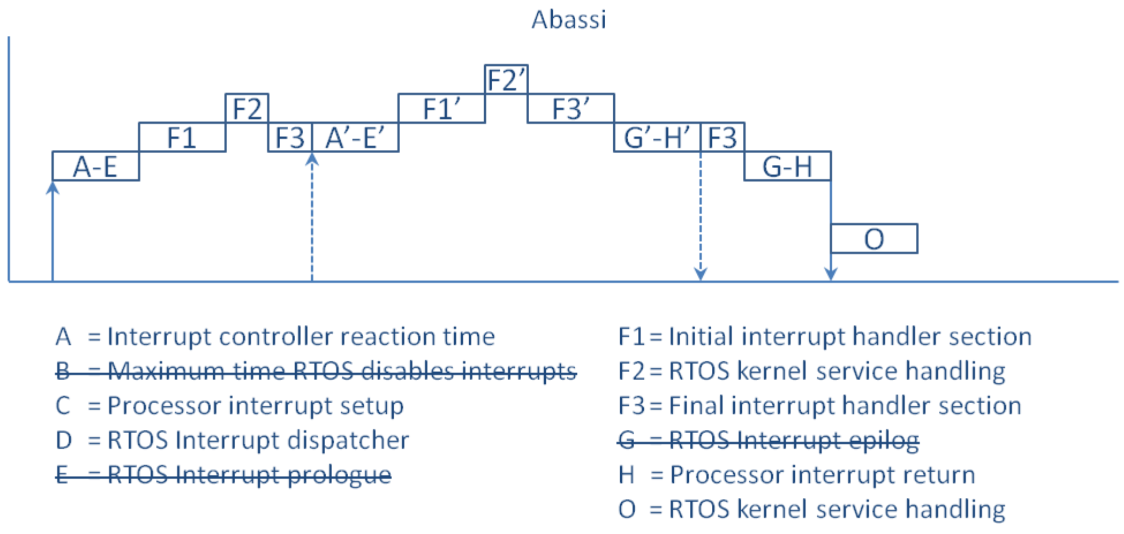

Abassi
F2’
F2
F1’
F3’
F1
F3
A’-E’
G’-H’
F3
A-E
G-H
O
A = Interrupt controller reaction time
~~B = Maximum time RTOS disables interrupts~~
C = Processor interrupt setup
D = RTOS Interrupt dispatcher
~~E = RTOS Interrupt prologue~~
F1 = Initial interrupt handler section
F2 = RTOS kernel service handling
F3 = Final interrupt handler section
~~G = RTOS Interrupt epilog~~
H = Processor interrupt return
O = RTOS kernel service handling

**Figure 4. Nested interrupt handling**

## Interrupt Dispatcher

The Abassi interrupt dispatcher forwards interrupts to their appropriate interrupt handler. Using a dispatcher eliminates the need for each interrupt handler to instantiate interrupt prologue and epilogue code, used to interact with the RTOS, resulting in significant program memory savings. It also eliminates restrictions on the RTOS services available within an interrupt handler, and allows interrupt handlers to be standard 'C' functions, for improved code portability. Furthermore, API functions exist to easily install or uninstall interrupt handlers, eliminating the need to manually modify linker and assembler files to update the interrupt vector table.

Consider the case of an application that contains 3 interrupt sources, and compare the simplicity and program memory savings afforded by the use of an interrupt dispatcher.

Standard RTOS

```
main.c:
#include "OS_Interrupts.h"
…
__interrupt void ISR_Handler1(void)
{
   OS_EnterInterrupt();
   … /* Interrupt handler */
   OS_ExitInterrupt();
}

__interrupt void ISR_Handler2(void)
{
   OS_EnterInterrupt();
   … /* Interrupt handler */
   OS_ExitInterrupt();
}

__interrupt void ISR_Handler3(void)
{
   OS_EnterInterrupt();
   … /* Interrupt handler */
   OS_ExitInterrupt();
}

int main(void) {
   …
}

vector.s:
   …
   EXTERN  ISR_Handler1
   EXTERN  ISR_Handler2
   EXTERN  ISR_Handler3
   …
   DC32   ISR_Handler1
   DC32   ISR_Handler2
   DC32   ISR_Handler3
   …
```

Abassi

```
main.c:
#include "Abassi.h"
…
void ISR_Handler1(void) {
   … /* Interrupt handler */
}

void ISR_Handler2(void) {
   … /* Interrupt handler */
}

void ISR_Handler3(void) {
   … /* Interrupt handler */
}

int main(void) {
   OSisrInstall(1, &ISR_Hander1);
   OSisrInstall(2, &ISR_Hander2);
   OSisrInstall(3, &ISR_Hander3);
   …
}
```

Abassi eliminates the need to manually modify the interrupt vector table, allows the use of standard, portable ‘C’ function for the interrupt handlers, and saves significant program memory by eliminating the need to duplicate interrupt entry/exit code.

### Fast Interrupts

Interrupts that do not access RTOS services can bypass the interrupt dispatcher, and be implemented as fast interrupts instead. Fast interrupts safely co-exist with dispatched interrupts, but do not benefit from the ease-of-use enhancements the dispatcher provides. They require the use of non-portable interrupt definitions (`interrupt void func() { }` or `__attribute__((__interrupt__))`), and require the manual update of linker and assembler files to install the interrupt in the vector table, instead of the in-built interrupt handler installer.

By using the interrupt dispatcher, instead of hard-coding fast interrupts, improved code portability is achieved, with virtually no added overhead. It also eliminates future issues if interrupt handlers are updated to include RTOS service calls, and gives access to simplified API functions to install or uninstall interrupt handlers.

## Conclusion

By employing a greenfield design approach, Code Time Technologies has been able to create a next generation real-time kernel that vastly improves upon any available today. The Abassi real-time kernel outperforms all existing platforms by combining code size and CPU efficiency with an unrivalled set of features and usability enhancements.