Advantages of the Cortex-M3

Author:
Brian Nagel, Micrium

Synopsis:
ARM introduced its new line of Cortex architectures in 2004, tackling the embedded marketplace with a three-pronged solution. Firstly, the Cortex-A family provides high-performance applications processors. Next, the Cortex-R family targets real-time, deeply-embedded solutions. Finally, the Cortex-M family are low-cost embedded microcontrollers.

The Cortex-M3, a member of the Cortex family, has already been integrated into MCU lines from several silicon vendors, with several more intending to follow. It is often marketed as a successor to the popular ARM7TDMI-S; though the architectures are much different, most generic comparisons (as listed in Table 1) demonstrate that the Cortex-M3 is superior. In general, the improvements it incorporates help establish its suitability for embedded systems in cost-sensitive applications that require deterministic system behavior.

Characteristic	ARM7TDMI-S	Cortex-M3
Interrupt Architecture	FIQ + IRQ; IRQ nesting may be possible	NMI + up to 240 physical interrupts; nesting always possible Interrupt
Latency	24-42 cycles	12 cycles
Interrupt Controller	External peripheral	Integrated peripheral (NVIC)
Hardware Stacks	FIQ, IRQ, SVC, USR, ABT, UND	Process + Main
OS Timer	External peripheral	Integrated peripheral (SysTick)
Context Switch (µC/OS-II)	50-73 instructions	29 instructions

Table 1: Comparison of ARM7TDMI-S and Cortex-M3 Characteristics.

Although low-level characteristics of an embedded processor, such as interrupt prioritization, stack handling and instruction set, are generally invisible to the C- or C++-developer, they can affect end-product performance significantly. The absence of interrupt nesting capabilities, for instance, may unacceptably increase the latency of a high-priority source with strict service requirements. Since the Cortex-M3 improves the ARM7 in most qualitative estimates—simpler stack architecture, better interrupt controller, enhanced debug capabilities, higher-performance instruction set—it is fitting to detail the differences.

Stacking and Interrupts

The Cortex-M3 reduces the overhead (and complexity) of ARM7TDMI-S stack management by incorporating only two stacks (as shown in Figue 1 on next page). Tasks execute in Thread mode, using the *process stack*; interrupts execute in Handler mode, using the *main stack*. The task context is automatically saved on the process stack when an exception occurs, whereupon the processor moves to Handler mode, making the main stack active. On return from the exception, the task context is restored and Thread mode re-instated.

This assumes that the interrupted task remains the active task at interrupt return. If, however, a new task is to be scheduled, the context switch must take place. By having already saved the task context, the Cortex-M3 hardware reduces context switch times. The software must only save registers `R4-R11` on the old task's stack, pop `R4-R11` from the new task's stack, adjust the stack pointer and exception return value and return. Not only is this substantially more straightforward than the procedure required by a ARM7/9 processor, it also consumes fifty percent fewer processor cycles.

NVIC and SysTick

The Cortex-M3 includes not only the core CPU (ALU, control logic, data interface, instruction decoding, etc.), but also several integrated peripherals. Most important is the Nested Vectored Interrupt Controller (NVIC), designed for low latency, efficiency and configurability. The NVIC:

- Saves half the processor registers automatically upon interrupt, restoring them upon exit, allowing for efficient interrupt handling. Moreover, back-to-back interrupts are handled without saving/restoring registers (since that is unnecessary);
- Offers 8 to 256 priority levels, with dynamic priority assignment;
- Handles between 1 and 240 external sources; and

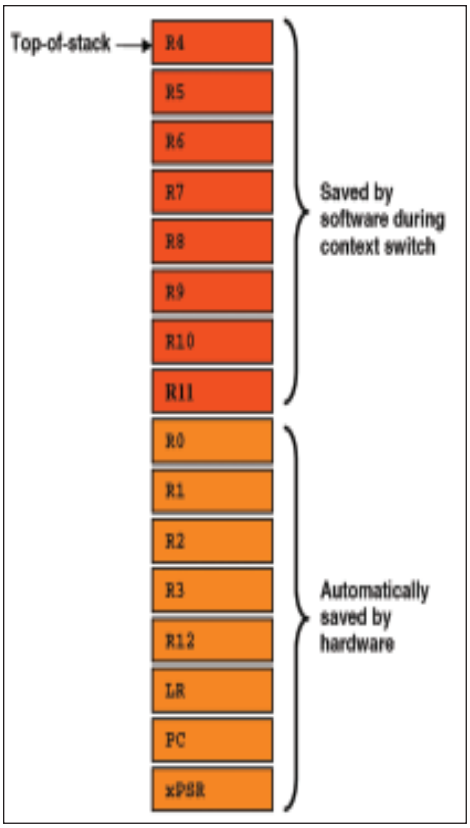

Figure 1: Cortex-M3 Task Stacking.

- Integrates a 24-bit down-counting timer (the SysTick) intended for RTOS use.

The NVIC and SysTick peripherals ease the migration between Cortex-M3 processors; this is particularly true for RTOSs. The µC/OS-II port to a ARM7TDMI-S processor required

- The generic ARM port (which is the same for all processors);
- A port to the processor's interrupt controller; and
- A port to a hardware timer. (in addition to low-level initialization of system clocks and GPIO). Generally, this involves writing a set of functions as shown in Figure 2(b-d).

```
static void App_TaskStart (void *p_arg)
{
       BSP_Init();
       BSP_TmrTick_Init();
#if OS_TASK_STAT_EN > 0
       OSStatInit();
#endif
       .
       .
       .
}
```
(a)

```
void    BSP_TmrTick_Init (void)
{
     /* ... Initialize uC/OS-II tick timer ... */ }
```
(b)

```
void    OS_CPU_ExceptHndlr (INT32U except_type)
{
     CPU_FNCT_VOID pfnct;

     if (except_type == OS_CPU_ARM_EXCEPT_IRQ) {
     /* ... Handle IRQ ... */
     } else if (except_type == OS_CPU_ARM_EXCEPT_FIQ) {
     /* ... Handle FIQ ... */
     } else {

     }
}
```
(c)

```
void   BSP_IntEn (CPU_INT08U src)
{
     /* Enable interrupt (IRQ)            ... */
}

void   BSP_IntDis (CPU_INT08U src)
{
     /* Disable interrupt (IRQ)           ... */
}

void   BSP_IntVectSet (CPU_INT08U    src,
                       CPU_FNCT_VOID handler)
{
     /* Assign interrupt (IRQ) handler    ... */
}

void   BSP_IntPrioSet (CPU_INT08U src,
                       CPU_INT08U prio)
{
     /* Set interrupt (IRQ) priority      ... */
}
```
(d)

Figure 2: µC/OS-II ARM7TDMI-S Port.
(a) The start task calls `BSP_Init()` ***to perform low-level processor initialization and (b)*** `BSP_TmrTick_Init()` ***to initialize the processor timer that will generate the OS timer tick.***

(c) The generic ARM port (used for all ARM7/9 processors) includes assembly language to move all exceptions to a single handler in C, `OS_CPU_ExceptHandler()`***. That function must call the handler for the IRQ or FIQ that generated the request.***

(d) Functions to enable and disable individual IRQ source, as well as to set handlers and priorities of sources, must be written (or this functionality should be 'inlined' where appropriate).

In contrast, a Cortex-M3 port must only provide a function that returns the clock frequency on which the SysTick timer is based (in addition to normal low-level initialization of system clocks and GPIO), as shown in Figure 3(b). The interrupt functionality (Figure 2(d)), which must be written for each ARM7TDMI-S port, is provided for all Cortex-M3's in the µC/CPU port's `CPU_IntSrc...()` functions:

- `CPU_IntSrcEn()` Enable interrupt.
- `CPU_IntSrcDis()` Disable interrupt.
- `CPU_IntSrcPrioSet()` Set interrupt priority.
- `CPU_IntSrcPrioGet()` Get interrupt priority.

```
static void App_TaskStart (void *p_arg)
{
       BSP_Init();
       OS_CPU_SysTickInit();

#if OS_TASK_STAT_EN > 0
       OSStatInit();
#endif
       .
       .
       .
}                                          (a)
```

```
INT32U     OS_CPU_SysTickClkFreq (void)
{
     /* Calc & rtn SysTick clk freq ... */
}
                                           (b)
```

Figure 3: µC/OS-II Cortex-M3 Port.
(a) The start task calls `BSP_Init()` ***to perform low-level processor initialization and*** `OS_CPU_SysTickInit()` ***to initialize the SysTick timer.***

(b) `OS_CPU_SysTickClkFreq()` ***must return the clock frequency on which the SysTick timer is based.***

The sleep mode feature of the Cortex-M3 can be used to conserve power when the target application is idle. In µC/OS-II, the idle task executes when no higher priority (i.e., application) task needs to run. This task calls an application-level hook, `App_TaskIdleHook()`, as shown in Figure 4. By invoking the `WFI` (wait for interrupt) instruction, this hook causes the processor to enter sleep mode until the next interrupt is received.

```
#if OS_VERSION >= 251
void App_TaskIdleHook (void)
{
     CPU_WaitForInt();
}
#endif
```

Figure 4: µgC/OS-II Idle Hook.
An RTOS can use the integrated sleep mode of the Cortex-M3 to conserve power in the idle task. The application idle task hook should invoke the `WFI` ***instruction (which is done here by the assembly-language function*** `CPU_WaitForInt()`.

Memory Map and MPU

internal SRAM, peripherals and external memories and devices occupy set areas of the first 3.5-GB of the 4-GB space. The next 256-kB is devoted to internal core peripherals, with register locations common to all implementations. The *System Control Space* (SCS) within this region includes the NVIC (which includes the SysTick) as well as other important control registers. The lowest 1-MB of the SRAM and peripheral spaces are bit-banding regions; each bit in these regions can be set or cleared by writing 1 or 0 to a word in the respective alias region. For example, to clear bit 5 of the byte at `0x20001000`, the value 0 would be written to the word at address

```
bit_word_addr = bit_band_base + (byte_offset x 32) + (bit_nbr x 4)
              = 0x22000000    + (0x00001000  x 32) + (5        x 4)
  = 0x22020014
```

Unlike traditional bit set and clear operations, bit-banding offers an atomic procedure requiring no read-modify-write cycle. If part of a processor's internal RAM extends outside the bit band region or external RAM is used for data, important variables (e.g., core RTOS structures) can still be guaranteed to lie within the SRAM bit-band region. Under most development toolchains, the read/write data (variables) for a compile unit (typically, a code file) can be assigned to a specific memory region defined in a linker file or memory map.

A Memory Protection Unit (MPU) is an optional Cortex-M3 component. As part of the core, the MPU (if present) is the same for every implementation, all served by the same RTOS port (as with the NVIC and SysTick support). Up to eight regions, each with eight sub-regions that can be individually disabled, can be constructed and assigned access permissions. The MPU can be leveraged to protect a system against unauthorized accesses to system memory with µC/OS-MPU, an extension of µC/OS-II. The system is divided into MPU contexts (processes); each process contains one or more tasks (threads). A process has its individual read, write and execution rights. Exchanging data between threads can be done in the same manner as µC/OS-II, but handling across different processes is done by the core operating system.

Instruction Set

With its v4T architectures, ARM introduced the Thumb Instruction Set Architecture (ISA), a 16-bit ISA that allowed improved

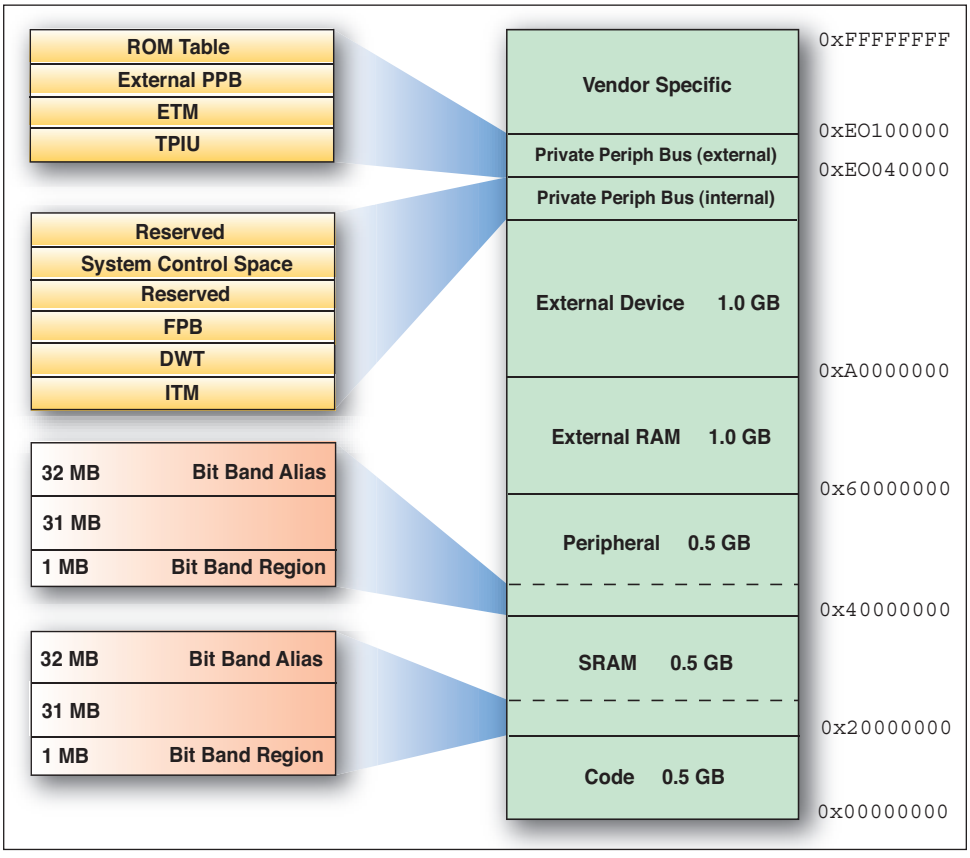

Figure 5: Cortex-M3 Memory Map.

code density (generally, by around 30%). The Cortex-M3 implements ARMv7-M, using the Thumb-2 ISA, a superset of the original Thumb, including new 16- and 32-bit instructions. These additions remove the need for the original ARM ISA, so Cortex-M3s always execute in a single mode (Thumb-2), unlike ARM7s that needed to switch between ARM/Thumb modes.

The Cortex-M3 includes 36 instructions not available on the ARM7TDMI-S; the more interesting of these include

- `CLZ` (count leading zeros)
- `RBIT` (reverse bits in word)
- `MOVT` (move 16-bit immediate to top of word) and MOVW (move 16-bit immediate)
- `BFC` (bit field clear) and BFI (file field insert)
- `SBFX` (signed bit field extract) and UBFX (unsigned bit field extract)

The `CLZ` instruction offers one very relevant example optimization: kernel scheduling algorithms. Most kernels, like µC/OS-II, track ready tasks using bit-fields. µC/OS-II uses a two-level hierarchy of fields. As shown in Figure 6, `OSRdyTbl[]` contains one bit for each task; if that bit is 1, then the corresponding task is ready; if that bit is 0, then the task is not ready. `OSRdyGrp` contains one bit for each row in `OSRdyTbl[]`; if any task in that row is ready, then the bit in `OSRdyGrp` is 1; if no task is ready, then the bit is `OSRdyGrp` is 0. When a task is suspended, its bit in `OSRdyTbl[]` must be cleared and, if it had been the only ready task in its row, the row's bit in `OSRdyGrp` must also be cleared.

Since µC/OS-II is designed to be hardware-independent, it relies on no `CLZ` instruction to determine the highest-priority ready task. However, the original C code (as shown in Figure 7, on next page, slightly modified for clarity) can easily be re-written in assembly to take advantage of `CLZ`. The result is shown in Figure 6, right. (Note: The `RBIT` instructions are necessary since the number of trailing zeros, rather than the number of leading zeros, should be counted.) Unlike Figure 6 (which illustrates a 64-task configuration), the C code and assembly are intended for a 256-task configuration, in which `OSRdyTbl[]` is an array of sixteen 16-bit integers and `OSRdyGrp` is a 16-bit integer.

Debug and Trace

One of the most exciting features of the Cortex-M3 is its debug controller. Devices are accessed through a Debug Port, either a *Serial Wire JTAG Debug Port* (SWJ-DP) or *Serial Wire Debug Port* (SW-DP), depending on implementation. SWD offers a low pin-count debug solution (two pins versus five pins for JTAG), with the same functionality and speed of JTAG.

A powerful trace unit incorporates traditional instruction trace functionality (through the *Embedded Trace Macrocell* (ETM)) and a novel *Instrumentation Trace Macrocell* (ITM). The ITM allows an application or software module to use `printf()`-style output to trace events. Unlike traditional semihosted `printf()`, ITM output can be performed without halting the target processor. Information from both the ETM and ITM are output through a *Serial Wire Viewer* (SWV), in either Manchester-encoded or UART format through a single output pin.

Through the *AHB Access Port* (AHB-AP), the

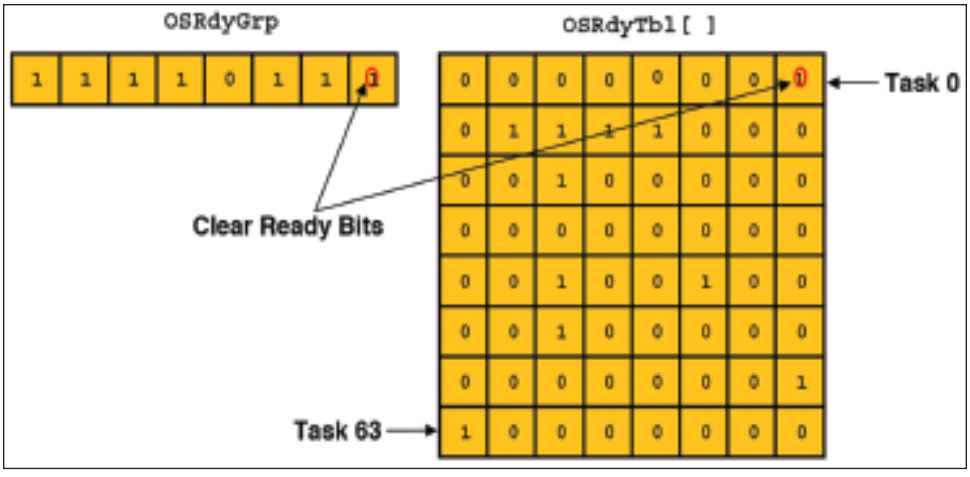

Figure 6: Managing Ready Task Information: Suspending Task 0

```
static void OS_SchedNew (void)                  OS_SchedNew:
{                                                 PUSH {R0-R4}
  INT8U x;                                        LDR R0, =OSRdyGrp
  INT8U y;
  INT16U *ptbl;

  if ((OSRdyGrp & 0xFF) != 0) {                   LDRH R2, [R0]
    y = OSUnMapTbl[OSRdyGrp & 0xFF];              RBIT R2, R2
  } else {                                        CLZ R2, R2 ; y = CLZ(OSRdyGrp);
    y = OSUnMapTbl[(OSRdyGrp >> 8) & 0xFF] + 8;
  }
                                                                ; ptbl = &OSRdyTbl[y];
  ptbl = &OSRdyTbl[y];                            LDR R0, =OSRdyTbl
                                                  ADD R1, R0, R2, LSL #1

  if ((*ptbl & 0xFF) != 0) {                      LDRH R3, [R1]
    x = OSUnMapTbl[(*ptbl & 0xFF)];               RBIT R3, R3
  } else {                                        CLZ R3, R3  ; x = CLZ(*ptbl);
    x = OSUnMapTbl[(*ptbl >> 8) & 0xFF] + 8;
  }


                                                                ; OSPrioHighRdy
  OSPrioHighRdy = (INT8U)((y << 4) + x;                         ; = x + (y << 4);
                                                  LDR R0, =OSPrioHighRdy
                                                  ADD R4, R3, R2, LSL #4
                                                  STRB R4, [R0]

  return;                                         POP {R0-R4}
}                                                 BX LR
```

Figure 7: Finding Highest-Priority Ready Task: µC/OS-II C code vs.Cortex-M3 Assembly

debug port can access all memory and registers (including processor registers) without stopping the core. This means the Cortex-M3 memory can be directly accessed by the debugger, so a real-time monitoring program such as µC/Probe operates non-intrusively (without additional code, without stopping the processor and with only a standard debug interface). In contrast, the memory on ARM7/9 processors can only be accessed without halting the processor through the DCC (Debug Communication Channel). Since the DCC offers only a single data register for processor/debugger communication, ARM7s and ARM9s require additional tar-

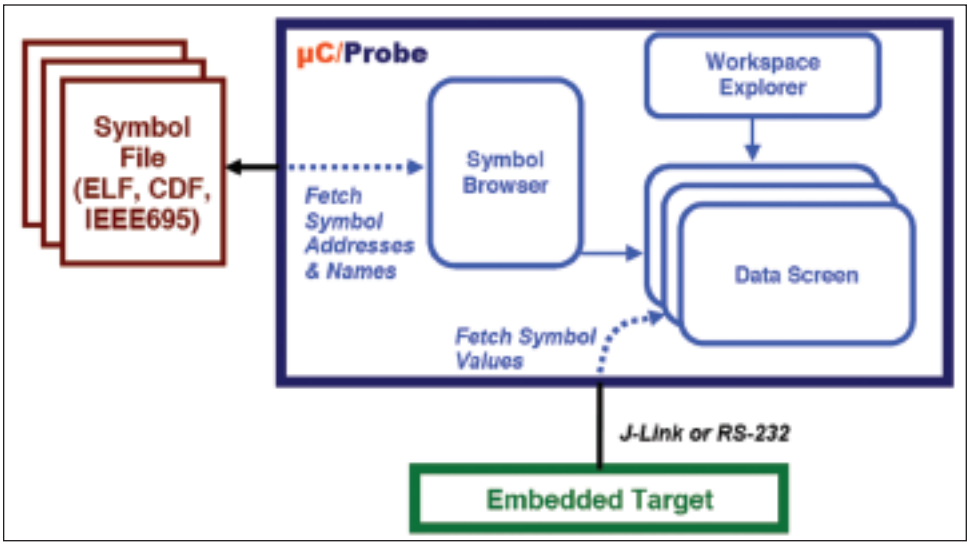

Figure 8: Basic µC/Probe Architecture

get code to perform memory reads and writes.

Micrium's real-time monitor, µC/Probe, can retrieve target memory data with an emulator (such as a J-Link) via a Cortex-M3's SWD or JTAG interface. µC/Probe is a Windows application that allows a user to display the value (at run-time) of virtually any variable or memory location on a connected embedded target. The user simply populates the graphical environment with gauges, tables, graphs and other components, and associates each of these with a variable. Variable names and addresses are read from an ELF file produced by the user's compiler/ linker. Once communication has been configured (baud rate, port, etc.), data collection can start. The components will then update with variable values read from the target. The basic architecture is illustrated in Figure 8.

Figure 9 shows two screens from the µC/OS-II workspace, populated with task information reminiscent of a typical IDE's kernel awareness. Through the Cortex-M3's debug interface, via a J-Link, µC/Probe can gather the values of about 900 symbols per second (about 4000 bytes per second), which is comparable with transfer rates achieved with RS-232 at 115200 baud. However, the J-Link access requires no target code and does not consume processor cycles or stop the processor.

Summary

The Cortex-M3 processor, based on the ARMv7-M architecture, is intended for cost-sensitive embedded applications. This MCU incorporates an efficient interrupt controller (NVIC), including a RTOS timer (the SysTick), and a MPU (optionally). In contrast to most previous ARM architectures, it also has a set memory map. These shared features make it easier to migrate from one controller to another or to port an RTOS to a new platform.

Many features aim to make software on Cortex-M3s more efficient. The simple stack architecture, with the NVIC's PendSV exception, reduces context switch time. The instruction set of the Cortex-M3 includes helpful new instructions, such as a count-leading-zeros instruction, that can improve assembly for common algorithms. When the processor is idle, it can enter sleep mode, to be awoken when an interrupt occurs.

Finally, the debug controller makes developing and testing software easier. Powerful trace features allow greater visibility into application operation. Basic debug accesses, such as reading and writing memory locations, require no processor halt, so that a real-time viewer such as µC/Probe can provide truly real-time data.

Micrium's µC/OS-II was the first RTOS ported to the Cortex-M3 (and later architectures, including the Cortex-M1), and other software modules, such as µC/TCP-IP (network protocol stack) and µC/USB (USB device and host stacks) have subsequently been supported. Throughout the process of porting and benchmarking this software, Cortex-M3s proved amenable to the requirements of modern embedded software. Not only is the architecture sensible, stable and efficient, but its designers were evidently keen to provide a developer-friendly platform. The sophisticated debug system and six flash breakpoints are immensely helpful during testing and development.

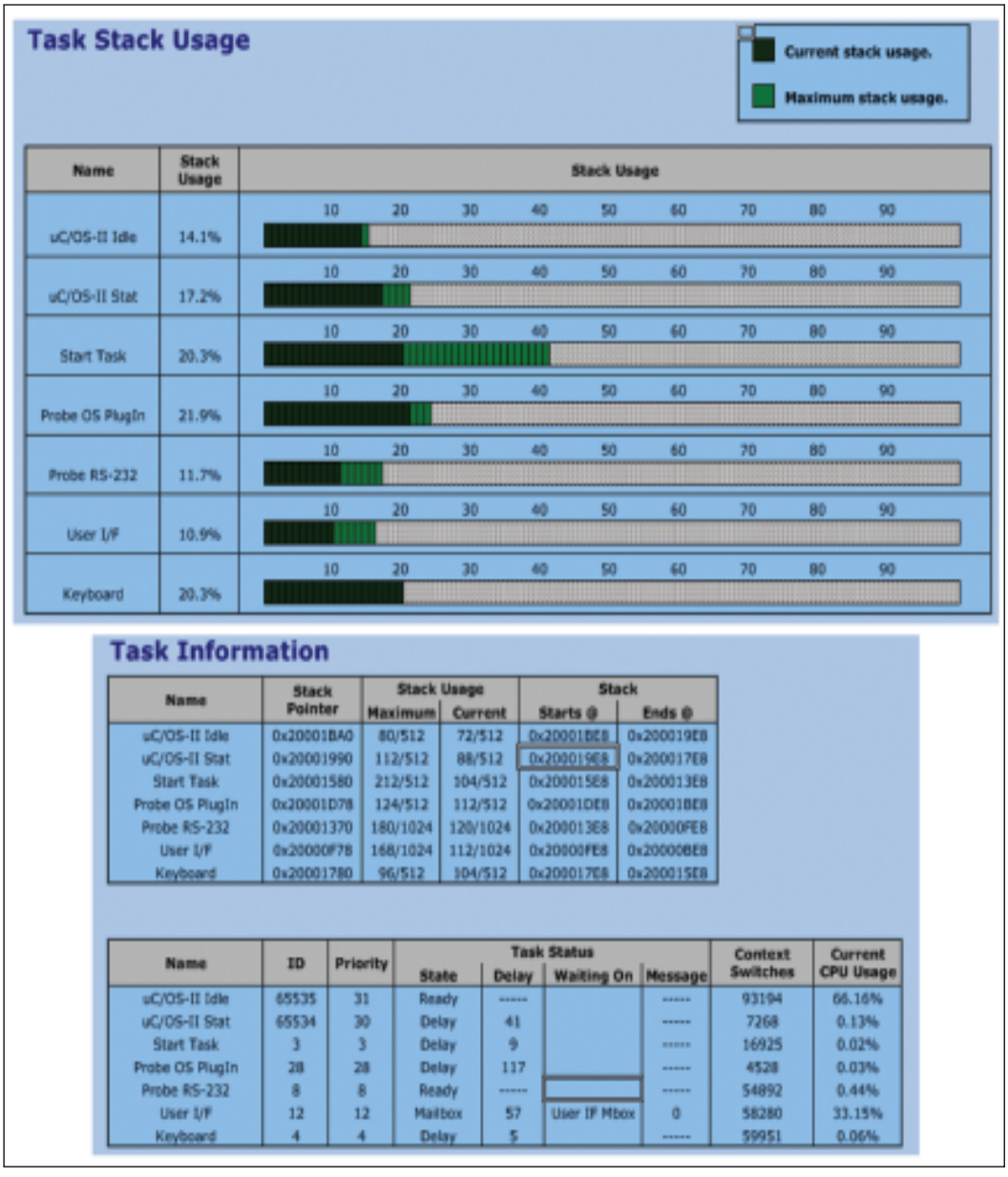

Figure 9: µC/Probe Workspace for µC/OS-II: Task Stack Usage (top) and Task Information (bottom)