Application Report
SPNA138A–November 2011

# Execution Time Measurement for Hercules™ ARM® Safety MCUs

*Rainer Troppmann*

**ABSTRACT**

This application report describes methods for measuring code execution time of TMS470 and TMS570 microcontrollers. Besides the typical pin toggle approach, where the duration between two pin toggles is measured, e.g., with an oscilloscope, Hercules MCUs support cycle count methods to measure the code execution time in terms of clock cycles.

The real-time-interrupt (RTI) module hardware counters can be used to count the clock cycles required to execute the code. In case of a Cortex-R4F-based TMS570 microcontroller, a third measurement option can be chosen. The performance monitoring unit (PMU) integrated into the Cortex-R4F CPU can be configured to count the CPU clock cycles or other CPU events of interest.

Project collateral and source code discussed in this application report can be downloaded from the following URL: http://www.ti.com/lit/zip/spna138.

**Contents**



## 1 Code Execution Time Measurement Methods

Hercules MCUs support multiple methods to measure the code execution time. The next sections provide the software implementation details for the three measurement methods: pin toggle, RTI cycle count, and the performance measurement unit (PMU) cycle count by providing code examples for the related methods. The included code snippets, which are based on the requirements of the Herstellerinitiative Software (HIS), show the code relevant to the topics in this application report. If needed, the code can also be re-written according to other application-/customer-specific naming conventions. Texas Instruments provides a hardware abstraction layer code generation tool (HALCoGen) with a graphical user interface to ease the code generation process (see: http://www.ti.com/tool/halcogen). In addition, a minimum required device setup is shown. For more details, see *Recommended Initializations for TMS570LS20x/10x Microcontrollers* (SPNA119) and *Initialization of Hercules™ ARM® Cortex™-R4F Microcontrollers* (SPNA106).

In principle, the code for which the code execution time is measured is enclosed between an action that starts the measurement process, i.e., starts resp. reads the reference counter or toggles a device pin, and an action that stops the measurement process, i.e., stops resp. reads the reference counter or toggles a device pin. Due to the Hercules MCUs system architecture, these measurement actions take some time to become effective. This measurement execution time should be taken into account and subtracted from the total measured time (cycle counts). To allow an easy identification of the measurement execution time, the start and stop action is executed a second time, right after each other without having the application code in between. This gives the time (number of cycles) for the adjustment to be made.

For some use cases, the application code, for which the code execution time is measured, might need to be run more than once to get the worst case cycle count, e.g., because some of the variables controlling the program flow need to get to their required values. In such a case, the code might need to be executed upfront; in some cases, even multiple times, to the code of interest, but without the need to measure the execution time. Alternatively, a preparation code that sets up the required flow conditions can be run instead.

## 1.1 Hercules MCUs Device Startup

The device startup for the three measurement methods does not require any special treatment. But, it is important to know the settings made for the clocks the measurement methods are based on: system clock (HCLK), peripheral clock (VCLK), and the clock for the real-time-interrupt module (RTICLK). The number of cycles counted and the execution time is directly related to the prescaler settings of these clocks. The code snippets of the *system.c* code example show the configuration for a HCLK of 180 MHz, a VCLK of 90 MHz and a RTICLK of 90 MHz when a 16 MHz crystal is used as the oscillator clock source. The dedicated setup for the Hercules modules, PMU, RTI, and DIO, used for the measurements is done in the *main.c*, but requires that the CPU runs still in privilege mode. Therefore, the operating mode must not be switched to user mode as shown in the *sys_startup.c* code example. In case of a Cortex-R4F CPU device, the calculation of the absolute runtime can take advantage of the vector floating point (VFP) unit; e.g., the floating-point divisions like 'time_PMU_code = cycles_PMU_code / (f_HCLK);' used at the end of *main.c* can take advantage of the hardware divider. Therefore, the VFP setup function *_coreEnableVfp()* gets called at the beginning of the *sys_startup.c* code example.

***sys_startup.c*** *(excerpt)*

```
…

void _c_int00()
{
    /* Enable VFP Unit */
    _coreEnableVfp();

…

    /* Initialize System */
    systemInit();

…

    /* switch to user mode */
    // PMU requires privilege mode
//    asm(" mov   r0,        #0x03D0");
//    asm(" msr   cpsr,      r0");

    /* call the application */
    main();
    exit();
}
```

***system.c** (excerpt)*

```
…

void systemInit(void)
{
…

    /**   Initialize Flash Wrapper: */

    /**   - Setup pll control register 1:
    *     - Setup reset on oscillator slip
    *     - Setup bypass on pll slip
    *     - Setup Pll output clock divider
    *     - Setup reset on oscillator fail
    *     - Setup reference clock divider
    *     - Setup Pll multiplier
    */
    systemREG1->PLLCTL1 =  0x00000000U
                        |  0x20000000U
                        | (0U << 24U)
                        |  0x00000000U
                        | (5U << 16U)
                        | (134U << 8U)
…

    /** - Setup synchronous peripheral clock dividers for VCLK1 and VCLK2 */
    systemREG1->VCLKR     = 1U;
    systemREG1->VCLK2R    = 1U;
    systemREG1->PENA      = 1U;

    /** - Setup RTICLK1 and RTICLK2 clocks */
    systemREG1->RCLKSRC = (0U << 24U)
                        | (SYS_VCLK << 16U)
                        | (0U << 8U)
                        |  SYS_VCLK;
…
}
```

## 2 Pin Toggle

The pin toggle method provides one or more device pins where the change of the pin state can be observed by an external measurement tool like an oscilloscope. With this measurement method, the duration between two pin toggles is measured and then divided by the clock period to calculate the number of clock cycles.

### *2.1 DIO Setup*

The base setup for the DIO module (DIO is the naming for the general-purpose I/O (GPIO) applying HIS requirements) used for the pin toggle method and the required DIO functions are described in the *DIO_lld.c* and *DIO_lld.h* code examples. The two functions, *OCDIO_InitSync()* and *OCDIO_SetSync()*, provide the needed DIO functionality. In the following code example, GIOA[0] is set up as device output. Forcing GIOA[0] to ‘High’ during the measurement phase provides the capability to measure the execution time.

***DIO_lld.c** (excerpt)*

```
…

IO_ErrorType OCDIO_InitSync(OCDIO_ConfigType config)
{
    /* take DIO module out of reset state */
    OCDIO->GCR0 = 1U;

    /* initalise port 0 */
    OCDIO0->DOUT   = 0x00000000U;
    OCDIO0->DIR    = 0x00000001U;
    OCDIO0->PDR    = 0x00000000U;
    OCDIO0->PULDIS = 0x000000FFU;
    OCDIO0->PSL    = 0x00000000U;

    /* initalise interrupt registers */
    OCDIO->LVLCLR    = 0xFFFFFFFFU;
    OCDIO->LVLSET    = 0x00000000U;
    OCDIO->INTENACLR = 0xFFFFFFFFU;
    OCDIO->FLG       = 0xFFFFFFFFU;
    OCDIO->POL       = 0x000000FFU;
    OCDIO->INTDET    = 0x00000000U;

    return IO_E_OK;
}

…

void OCDIO_SetSync(IO_ChannelType channel, OCDIO_ValueType value)
{
    if (value)
    {
        OCDIO0->DSET = 0x01U << channel;
    }
    else
    {
        OCDIO0->DCLR = 0x01U << channel;
    }
}

…
```

***DIO_lld.h** (excerpt)*

```
…

/*-----------------------------------------------------------------------------*/
/* OCDIO configuration                                                         */

#define OCDIO_MAX_PORT         0x00000001U
#define OCDIO_MAX_CHANNEL      0x00000107U

/*-----------------------------------------------------------------------------*/
/* Definitions                                                                 */

/* Port 0 */
#define OCDIO_PORT_0 0U

#define OCDIO_0 0x00000000U

…

typedef       T_U32  OCDIO_ValueType;
typedef const void * OCDIO_ConfigType;

…

/*-----------------------------------------------------------------------------*/
/* DIO register definition                                                     */

typedef volatile struct
{
    T_U32 GCR0;      /* 0x0000 */
    T_U32 PWDN;      /* 0x0004 */
    T_U32 INTDET;    /* 0x0008 */
    T_U32 POL;       /* 0x000C */
    T_U32 INTENASET; /* 0x0010 */
    T_U32 INTENACLR; /* 0x0014 */
    T_U32 LVLSET;    /* 0x0018 */
    T_U32 LVLCLR;    /* 0x001C */
    T_U32 FLG;       /* 0x0020 */
    T_U32 OFFSET0;   /* 0x0024 */
    T_U32 OFFSET1;   /* 0x0028 */
    T_U32 EMUA;      /* 0x002C */
    T_U32 EMUB;      /* 0x0030 */
} T_OCDIO_REG;

#define OCDIO ((T_OCDIO_REG  *)0xFFF7BC00U)
typedef volatile struct
{
    T_U32 DIR;    /* 0x0000 */
    T_U32 DIN;    /* 0x0004 */
    T_U32 DOUT;   /* 0x0008 */
    T_U32 DSET;   /* 0x000C */
    T_U32 DCLR;   /* 0x0010 */
    T_U32 PDR;    /* 0x0014 */
    T_U32 PULDIS; /* 0x0018 */
    T_U32 PSL;    /* 0x001C */
} T_OCDIO_PORT;

#define OCDIO0 ((T_OCDIO_PORT *)0xFFF7BC34U)

/*-----------------------------------------------------------------------------*/
/* Global Variables                                                            */

extern T_OCDIO_PORT * const g_OCDIO_base[];
```

```
…

/*-----------------------------------------------------------------------------*/
/* DIO interface functions                                                     */

IO_ErrorType OCDIO_InitSync(OCDIO_ConfigType config);
…
void OCDIO_SetSync(IO_ChannelType channel, OCDIO_ValueType value);

…
```

## 3 Real-Time-Interrupt (RTI) Cycle Count

The RTI cycle count method uses the Hercules real-time interrupt (RTI) module to count RTICLK cycles during the time the measured code is running. The code execution time can then be calculated by multiplying the counted RTI clock cycles with the clock period.

The RTI free-running counter (FRC) in combination with the up-counter (UC) of the RTI module is used as the timebase for the measurements. In the given code example, RTI channel 0 is configured to count VCLK cycles by selecting VCLK as the source for the RTICLK. The finest granularity can be achieved by setting the compare value for the compare up-counter (CPUC) to '1', which results in a counter step width of two VCLK cycles; this is equivalent to four HCLK cycles with the given device setup (see clock prescaler settings applied in *system.c*; $f_{RTICLK} = f_{VCLK} = f_{HCLK} / 2$). Therefore, the RTI cycle count needs to be multiplied by a factor of two to get the number of VCLK cycles, or multiplied by a factor of four to get the number of HCLK cycles.

### 3.1 RTI Setup

The base setup for the RTI module and the required RTI functions are described in the *TIM_lld.c* and *TIM_lld.h* code examples (TIM is the naming for the RTI timer/counter applying HIS requirements). The functions *OCTIM_InitSync()*, *OCTIM_StartSync()*, *OCTIM_StopSync()* and *OCTIM_GetSync()* provide the needed RTI functionality. *OCTIM_InitSync()* does the initialization of the RTI module. *OCTIM_StartSync()* is used to start the RTI counter, and *OCTIM_StopSync()* is used to stop it. The value of the free-running counter is read with the *OCTIM_GetSync()* function.

***TIM_lld.c*** *(excerpt)*

```
…

IO_ErrorType OCTIM_InitSync(const void * address)
{
    /* initalise RTI module 0 */
    OCRTI0->GCTRL         = 0x00000000U;
    OCRTI0->TBCTRL        = 0x00000000U;
    OCRTI0->COMPCTRL      = 0x00000010U;
    OCRTI0->CNT[0U].UCx   = 0x00000000U;
    OCRTI0->CNT[0U].FRCx  = 0x00000000U;
    OCRTI0->CNT[0U].CPUCx = 0x00000001U;
    OCRTI0->CMP[0U].COMPx = 0xFFFFFFFFU;
    OCRTI0->CMP[0U].UDCPx = 0x00000000U;
    OCRTI0->INTFLAG       = 0x0007000FU;
    OCRTI0->CLEARINT      = 0x00070F0FU;

    return IO_E_OK;
}

…

OCTIM_ValueType OCTIM_GetSync(IO_ChannelType channel)
{
    OCTIM_ValueType value;

//orig//    value = OCRTI0->CMP[channel].COMPx - OCRTI0->CNT[channel].FRCx;
```

```
    value = OCRTI0->CNT[channel].FRCx;

    return value;
}

…

void OCTIM_StartSync(IO_ChannelType channel)
{
    OCRTI0->GCTRL |= (1U << channel);
}

…

void OCTIM_StopSync(IO_ChannelType channel)
{
    OCRTI0->GCTRL &= ~(1U << channel);
}

…
```

***TIM_lld.h*** *(excerpt)*

```
…

/*-----------------------------------------------------------------------------*/
/* OCTIM configuration                                                         */

#define OCTIM_MAX_PORT    0U
#define OCTIM_MAX_CHANNEL 2U

/*-----------------------------------------------------------------------------*/
/* Definitions                                                                 */

#define OCTIM_E_NOT_PRIVILEDGE_MODE 100U
#define OCTIM_0 0U

/*-----------------------------------------------------------------------------*/
/* Data Types                                                                  */

typedef T_U32 OCTIM_ValueType;

/*-----------------------------------------------------------------------------*/
/* RTI register definition                                                     */

typedef volatile struct
{
    T_U32 GCTRL;
    T_U32 TBCTRL;
    T_U32 CAPCTRL;
    T_U32 COMPCTRL;
    struct
    {
        T_U32 FRCx;
        T_U32 UCx;
        T_U32 CPUCx;
        T_U32 : 32;
        T_U32 CAFRCx;
        T_U32 CAUCx;
        T_U32 : 32;
        T_U32 : 32;
    } CNT[2U];
    struct
    {
        T_U32 COMPx;
```

```
        T_U32 UDCPx;
    } CMP[4U];
    T_U32 TBLCOMP;
    T_U32 TBHCOMP;
    T_U32 : 32;
    T_U32 : 32;
    T_U32 SETINT;
    T_U32 CLEARINT;
    T_U32 INTFLAG;
    T_U32 : 32;
    T_U32 DWDCTRL;
    T_U32 DWDPRLD;
    T_U32 WDSTATUS;
    T_U32 WDKEY;
    T_U32 WDCNTR;
} T_OCRTI;

#define OCRTI0 ((T_OCRTI *)0xFFFFFC00U)

…

/*-----------------------------------------------------------------------------*/
/* TIM interface functions                                                     */

IO_ErrorType OCTIM_InitSync(const void * address);
…
OCTIM_ValueType OCTIM_GetSync(IO_ChannelType channel);
…
void OCTIM_StartSync(IO_ChannelType channel);
void OCTIM_StopSync(IO_ChannelType channel);

…
```

# 4 Performance Measurement Unit (PMU) Cycle Count (Cortex-R4F only)

The PMU cycle count method uses the Cortex-R4F performance measurement unit (PMU) to count CPUCLK cycles during the time the measured code is running. The code execution time can then be calculated by multiplying the counted CPU clock cycles with the clock period.

The PMU consists of four counters: one only counting clock cycles and three for counting selectable events. The example code use event counter 0 to count the clock cycles; so the PMU configuration can be easily changed to count any other event type (see 'enum pmuEvent' in the *sys_pmu.h* source code). Since the CPU runs with system frequency (CPU clock MCLK, $f_{MCLK} = f_{HCLK}$), HCLK is used as the timebase for the measurements. In the given code example, event counter 0 of the PMU is set up to count CPU clock cycles (Event = PMU_CYCLE_COUNT).

## 4.1 PMU Setup

The PMU functionality is done in assembly code due to the CP15 accesses needed. The following PMU function calls are use to establish the measurement function:

| | |
|---|---|
| `_pmuInit();` | Initialize the PMU |
| `_pmuSetCountEvent();` | Select CPU cycles (Event = 0x11) to be counted |
| `_pmuEnableCountersGlobal();` | Enable all PMU counter |
| `_pmuResetCounters();` | Reset the selected PMU counter |
| `_pmuStartCounters();` | Start the selected PMU counter |
| `_pmuStopCounters();` | Stop the selected PMU counter |
| `_pmuGetEventCount();` | Get the current value of the selected PMU counter |

***sys_pmu.asm*** *(excerpt)*

```
    .text
    .arm

;-------------------------------------------------------------------------------
; Initialize Pmu
; Note: It will reset all counters

    .def     _pmuInit
    .asmfunc

_pmuInit

        stmfd sp!, {r0}
        ; set control register
        mrc   p15, #0, r0, c9, c12, #0
        orr   r0,  r0, #(1 << 4) + 6 + 1
        mcr   p15, #0, r0, c9, c12, #0
        ; clear flags
        mov   r0,  #0
        mcr   p15, #0, r0, c9, c12, #3
        ; select counter 0 event
        mcr   p15, #0, r0, c9, c12, #5 ; select counter
        mov   r0,  #0x11 ;
        mcr   p15, #0, r0, c9, c13, #1 ; select event
        ; select counter 1 event
        mov   r0,  #1
        mcr   p15, #0, r0, c9, c12, #5 ; select counter
        mov   r0,  #0x11
        mcr   p15, #0, r0, c9, c13, #1 ; select event
        ; select counter 2 event
        mov   r0,  #2
        mcr   p15, #0, r0, c9, c12, #5 ; select counter
        mov   r0,  #0x11
        mcr   p15, #0, r0, c9, c13, #1 ; select event
        ldmfd sp!, {r0}
        bx    lr

    .endasmfunc

;-------------------------------------------------------------------------------
; Enable Counters Global [Cycle, Event [0..2]]
; Note: It will reset all counters

    .def     _pmuEnableCountersGlobal
    .asmfunc

_pmuEnableCountersGlobal

        mrc   p15, #0, r0, c9, c12, #0
        orr   r0,  r0, #7
        mcr   p15, #0, r0, c9, c12, #0
        bx    lr

    .endasmfunc

;-------------------------------------------------------------------------------
; Reset Cycle Counter and Event Counters [0..2]

    .def     _pmuResetCounters
    .asmfunc

_pmuResetCounters
```

```
        mrc   p15, #0, r0, c9, c12, #0
        orr   r0,  r0, #6
        mcr   p15, #0, r0, c9, c12, #0
        bx    lr

    .endasmfunc

;-------------------------------------------------------------------------------
; Start Counters [Cycle, 0..2]

    .def     _pmuStartCounters
    .asmfunc

_pmuStartCounters

        mcr   p15, #0, r0, c9, c12, #1
        bx    lr

    .endasmfunc

;-------------------------------------------------------------------------------
; Stop Counters [Cycle, 0..2]

    .def     _pmuStopCounters
    .asmfunc

_pmuStopCounters

        mcr   p15, #0, r0, c9, c12, #2
        bx    lr

    .endasmfunc

;-------------------------------------------------------------------------------
; Set Count event

    .def     _pmuSetCountEvent
    .asmfunc

_pmuSetCountEvent

        lsr   r0,  r0, #1
        mcr   p15, #0, r0, c9, c12, #5 ; select counter
        mcr   p15, #0, r1, c9, c13, #1 ; select event
        bx    lr

    .endasmfunc

;-------------------------------------------------------------------------------
; Get Event Counter Count Value

    .def     _pmuGetEventCount
    .asmfunc

_pmuGetEventCount

        lsr   r0,  r0, #1
        mcr   p15, #0, r0, c9, c12, #5 ; select counter
        mrc   p15, #0, r0, c9, c13, #2 ; read event counter
        bx    lr

    .endasmfunc
```

***sys_pmu.h** (excerpt)*

```
…
/** @def pmuCOUNTER0
*   @brief pmu event counter 0
*
*   Alias for pmu event counter 0
*/
#define pmuCOUNTER0 0x00000001U

/** @enum pmuEvent
*   @brief pmu event
*
*   Alias for pmu event counter increment source
*/
enum pmuEvent
{
    PMU_INST_CACHE_MISS                     = 0x01,
    PMU_DATA_CACHE_MISS                     = 0x03,
    PMU_DATA_CACHE_ACCESS                   = 0x04,
    PMU_DATA_READ_ARCH_EXECUTED             = 0x06,
    PMU_DATA_WRITE_ARCH_EXECUTED            = 0x07,
    PMU_INST_ARCH_EXECUTED                  = 0x08,
    PMU_EXCEPTION_TAKEN                     = 0x09,
    PMU_EXCEPTION_RETURN_ARCH_EXECUTED      = 0x0A,
    PMU_CHANGE_TO_CONTEXT_ID_EXECUTED       = 0x0B,
    PMU_SW_CHANGE_OF_PC_ARCH_EXECUTED       = 0x0C,
    PMU_BRANCH_IMM_INST_ARCH_EXECUTED       = 0x0D,
    PMU_PROC_RETURN_ARCH_EXECUTED           = 0x0E,
    PMU_UNALIGNED_ACCESS_ARCH_EXECUTED      = 0x0F,
    PMU_BRANCH_MISSPREDICTED                = 0x10,
    PMU_CYCLE_COUNT                         = 0x11,
    PMU_PREDICTABLE_BRANCHES                = 0x12,
    PMU_INST_BUFFER_STALL                   = 0x40,
    PMU_DATA_DEPENDENCY_INST_STALL          = 0x41,
    PMU_DATA_CACHE_WRITE_BACK               = 0x42,
    PMU_EXT_MEMORY_REQUEST                  = 0x43,
    PMU_LSU_BUSY_STALL                      = 0x44,
    PMU_FORCED_DRAIN_OFSTORE_BUFFER         = 0x45,
    PMU_FIQ_DISABLED_CYCLE_COUNT            = 0x46,
    PMU_IRQ_DISABLED_CYCLE_COUNT            = 0x47,
    PMU_ETMEXTOUT_0                         = 0x48,
    PMU_ETMEXTOUT_1                         = 0x49,
    PMU_INST_CACHE_TAG_ECC_ERROR            = 0x4A,
    PMU_INST_CACHE_DATA_ECC_ERROR           = 0x4B,
    PMU_DATA_CACHE_TAG_ECC_ERROR            = 0x4C,
    PMU_DATA_CACHE_DATA_ECC_ERROR           = 0x4D,
    PMU_TCM_FATAL_ECC_ERROR_PREFETCH        = 0x4E,
    PMU_TCM_FATAL_ECC_ERROR_LOAD_STORE      = 0x4F,
    PMU_STORE_BUFFER_MERGE                  = 0x50,
    PMU_LSU_STALL_STORE_BUFFER_FULL         = 0x51,
    PMU_LSU_STALL_STORE_QUEUE_FULL          = 0x52,
    PMU_INTEGER_DIV_EXECUTED                = 0x53,
    PMU_STALL_INTEGER_DIV                   = 0x54,
    PMU_PLD_INST_LINE_FILL                  = 0x55,
    PMU_PLD_INST_NO_LINE_FILL               = 0x56,
    PMU_NON_CACHEABLE_ACCESS_AXI_MASTER     = 0x57,
    PMU_INST_CACHE_ACCESS                   = 0x58,
    PMU_DOUBLE_DATA_CACHE_ISSUE             = 0x59,
    PMU_DUAL_ISSUE_CASE_A                   = 0x5A,
    PMU_DUAL_ISSUE_CASE_B1_B2_F2_F2D        = 0x5B,
    PMU_DUAL_ISSUE_OTHER                    = 0x5C,
    PMU_DP_FLOAT_INST_EXCECUTED             = 0x5D,
    PMU_DUAL_ISSUED_PAIR_INST_ARCH_EXECUTED = 0x5E,
    PMU_DATA_CACHE_DATA_FATAL_ECC_ERROR     = 0x60,
```

```
    PMU_DATA_CACHE_TAG_FATAL_ECC_ERROR      = 0x61,
    PMU_PROCESSOR_LIVE_LOCK                 = 0x62,
    PMU_ATCM_MULTI_BIT_ECC_ERROR            = 0x64,
    PMU_B0TCM_MULTI_BIT_ECC_ERROR           = 0x65,
    PMU_B1TCM_MULTI_BIT_ECC_ERROR           = 0x66,
    PMU_ATCM_SINGLE_BIT_ECC_ERROR           = 0x67,
    PMU_B0TCM_SINGLE_BIT_ECC_ERROR          = 0x68,
    PMU_B1TCM_SINGLE_BIT_ECC_ERROR          = 0x69,
    PMU_TCM_COR_ECC_ERROR_LOAD_STORE        = 0x6A,
    PMU_TCM_COR_ECC_ERROR_PREFETCH          = 0x6B,
    PMU_TCM_FATAL_ECC_ERROR_AXI_SLAVE       = 0x6C,
    PMU_TCM_COR_ECC_ERROR_AXI_SLAVE         = 0x6D
};

/** @fn void _pmuInit_(void)
*   @brief Initialize Perfprmance Monitor Unit
*/
void _pmuInit(void);

/** @fn void _pmuEnableCountersGlobal_(void)
*   @brief Enable and reset cycle counter and all 3 event counters
*/
void _pmuEnableCountersGlobal(void);

…

/** @fn void _pmuResetCounters_(void)
*   @brief Reset cycle counter and event counters 0-2
*/
void _pmuResetCounters(void);

/** @fn void _pmuStartCounters_(unsigned counters)
*   @brief Starts selected counters
*   @param[in] counters - Counter mask
*/
void _pmuStartCounters(unsigned counters);

/** @fn void _pmuStopCounters_(unsigned counters)
*   @brief Stops selected counters
*   @param[in] counters - Counter mask
*/
void _pmuStopCounters(unsigned counters);

/** @fn void _pmuSetCountEvent_(unsigned counter, unsigned event)
*   @brief Set event counter count event
*   @param[in] counter - Counter select 0..2
*   @param[in] event   - Count event
*/
void _pmuSetCountEvent(unsigned counter, unsigned event);

/** @fn unsigned _pmuGetEventCount_(unsigned counter)
*   @brief Returns current event counter value
*   @param[in] counter - Counter select 0..2
*
*   @return event counter count.
*/
unsigned _pmuGetEventCount(unsigned counter);

…
```

# 5 Measurement

The Hercules MCUs supports multiple methods to count clock cycles, e.g., used to measure the code execution time. The example code *main.c* does cover all of them, allowing the user to switch them on/off before the code compilation is done.

## 5.1 Measurement Flow

The individual method is selected by adding the #define statement for the related measurement method to beginning of the *main.c*, for example,

```
#define PMU_Cycle
```

```
#define RTI_Cycle
```

```
#define GIO_Toggle
```

The source files for the unused measurement methods can be removed from the Code Composer Studio™ project.

The measurement flow consists of the following five steps:

```
// -- Measurement Initialization --
// -- Measurement Preparation --
// *** Benchmark Code ***
// -- Measurement Time Compensation --
// -- Code Cycles / Run Time Calculation --
```

The *Measurement Initialization* step configures the module base functionality using the functions defined in the module related code files shown in the sections 2, 3 and 4 of this application report. The *Measurement Preparation* step is an optional step that allows getting the application code into an environment where it runs with the worst case cycle count. The *loop_count_prep_max* constant can be used to specify the number of loops the preparation code needs to be run upfront to the code execution time measurement (this can be the application code or any other special code) .

The third step *Benchmark Code* runs the code to be analyzed, by either doing the function call *code_to_be_measured()* , or in case the time for the call needs to be avoided, by replacing the function call with the application code it self.

Since the measurement actions take some time to get effective, the measurement execution time gets identified in the *Measurement Time Compensation* step by executing the measurement start and the stop action a second time right after each other, without having the application code in between. This gives the time (number of cycles) for the compensation to be made before the net execution time gets calculated in the last step *Code Cycles / Run Time Calculation*. The calculation of the absolute time is based on the definitions made for the HCLK and the RTICLK frequency, which should be in line with the clock settings made in the device setup.

The three measurement methods are wrapped around each other to achieve a better compensation for the measurement execution time in case more than one measurement method is chosen to be active.

***main.c*** *(excerpt)*

```
#include "sys_types.h"
#include "sys_common.h"

// Measurement Method used - please specify !!!
#define PMU_Cycle
#define RTI_Cycle
#define GIO_Toggle

#ifdef PMU_Cycle
#include "sys_pmu.h"
#endif //PMU_Cycle

#ifdef RTI_Cycle
#include "lld_TIM.h"
/*
```

```
RTI Clock Source Register (RCLKSRC), offset = 0x
If the RTIx clock source is chosen to be anything other than the default VCLK,
then the RTI clock needs to be at least three times slower than the VCLK.
bit9..8 RTI1DIV[1:0]
bit3..0 RTI1SRC[3:0], clock source x or VCLK (1000b..1111b, default = 1001b)
*/
#define f_RTICLK (float)  90.0 // f in [MHz]; RTICLK = VCLK = HCLK / 2 (depends on device setup)
#define RTI_FRC0 0xfffffc10 // Counter0
#endif //RTI_Cycle

#ifdef GIO_Toggle
#include "lld_DIO.h"
#endif //GIO_Toggle

#define f_HCLK   (float) 180.0 // f in [MHz]; HCLK (depends on device setup)

extern code_to_be_measured();

volatile unsigned int loop_count_prep, loop_count_prep_max=1000;
volatile unsigned int loop_count, loop_count_max=1000;


#ifdef PMU_Cycle
volatile unsigned long cycles_PMU_start, cycles_PMU_end, cycles_PMU_measure, cycles_PMU_comp,
cycles_PMU_code;
volatile float time_PMU_code;
#endif //PMU_Cycle

#ifdef RTI_Cycle
volatile unsigned long i;
volatile unsigned long cycles_RTI_start, cycles_RTI_end, cycles_RTI_measure, cycles_RTI_comp,
cycles_RTI_code, cycles_RTI;
volatile float time_RTI_code;
#endif //RTI_Cycle

#ifdef GIO_Toggle
void OCDIO_Notification(IO_ChannelType channel, OCDIO_ValueType notifType)
{
}
#endif //GIO_Toggle


void main()
{

// -- Measurement Initialization --
#ifdef PMU_Cycle
      _pmuInit();
      _pmuEnableCountersGlobal();
      _pmuSetCountEvent(pmuCOUNTER0, PMU_CYCLE_COUNT); // PMU_INST_ARCH_EXECUTED
#endif //PMU_Cycle

#ifdef RTI_Cycle
      OCTIM_InitSync(OCTIM_0);
#endif //RTI_Cycle

#ifdef GIO_Toggle
      OCDIO_InitSync(OCDIO_0);
#endif //GIO_Toggle


//  -- Measurement Preparation --
      for (loop_count_prep=0;loop_count_prep<loop_count_prep_max;++loop_count_prep)
      {
            // run benchmark code - function call or code sequence
            code_to_be_measured();  // or run preparation code
```

```
      }


//  -- Measurement Execution --
#ifdef PMU_Cycle
      _pmuResetCounters();
      _pmuStartCounters(pmuCOUNTER0);
      cycles_PMU_start = _pmuGetEventCount(pmuCOUNTER0);
#endif //PMU_Cycle

#ifdef RTI_Cycle
      OCTIM_StartSync(0);
      cycles_RTI_start = OCTIM_GetSync(0);
//    cycles_RTI_start = (unsigned long) *((volatile unsigned long*) ((unsigned long)
RTI_FRC0));
#endif //RTI_Cycle

#ifdef GIO_Toggle
      OCDIO_SetSync(0, 1);
//    OCDIO0->DSET = 0x01U;
#endif //GIO_Toggle

//  *** Benchmark Code ***
//    for (loop_count=0;loop_count<loop_count_max;++loop_count) // in case multiple loops are
needed
      {
            // run benchmark code - function call or code sequence
            code_to_be_measured();
      }

#ifdef GIO_Toggle
      OCDIO_SetSync(0, 0);
//    OCDIO0->DCLR = 0x01U;
#endif //GIO_Toggle

#ifdef RTI_Cycle
      OCTIM_StopSync(0);
      cycles_RTI_end = OCTIM_GetSync(0);
//    cycles_RTI_end = (unsigned long) *((volatile unsigned long*) ((unsigned long) RTI_FRC0));
      cycles_RTI_measure = cycles_RTI_end - cycles_RTI_start;
#endif //RTI_Cycle

#ifdef PMU_Cycle
      _pmuStopCounters(pmuCOUNTER0);
      cycles_PMU_end = _pmuGetEventCount(pmuCOUNTER0);
      cycles_PMU_measure = cycles_PMU_end - cycles_PMU_start;
#endif //PMU_Cycle


//  -- Measurement Time Compensation --
#ifdef PMU_Cycle
      _pmuResetCounters();
      _pmuStartCounters(pmuCOUNTER0);
      cycles_PMU_start = _pmuGetEventCount(pmuCOUNTER0);
#endif //PMU_Cycle

#ifdef RTI_Cycle
      OCTIM_StartSync(0);
      cycles_RTI_start = OCTIM_GetSync(0);
//    cycles_RTI_start = (unsigned long) *((volatile unsigned long*) ((unsigned long)
RTI_FRC0));
#endif //RTI_Cycle

#ifdef GIO_Toggle
      OCDIO_SetSync(0, 1);
//    OCDIO0->DSET = 0x01U;
```

```
      OCDIO_SetSync(0, 0);
//    OCDIO0->DCLR = 0x01U;
#endif //GIO_Toggle

#ifdef RTI_Cycle
      OCTIM_StopSync(0);
      cycles_RTI_end = OCTIM_GetSync(0);
//    cycles_RTI_end = (unsigned long) *((volatile unsigned long*) ((unsigned long) RTI_FRC0));
      cycles_RTI_comp = cycles_RTI_end - cycles_RTI_start;
#endif //RTI_Cycle

#ifdef PMU_Cycle
      _pmuStopCounters(pmuCOUNTER0);
      cycles_PMU_end = _pmuGetEventCount(pmuCOUNTER0);
      cycles_PMU_comp = cycles_PMU_end - cycles_PMU_start;
#endif //PMU_Cycle


//  -- Code Cycles / Run Time Calculation --
#ifdef PMU_Cycle
      cycles_PMU_code = cycles_PMU_measure - cycles_PMU_comp;
      time_PMU_code = cycles_PMU_code / (f_HCLK); // time_code [us], f_HCLK [MHz]
      //time_PMU_code = cycles_PMU_code / (f_HCLK * loop_Count_max); //
#endif //PMU_Cycle

#ifdef RTI_Cycle
      cycles_RTI = (cycles_RTI_measure - cycles_RTI_comp);
      /*
        RTI Compare Up Counter 0 Register (RTICPUC0), offset = 0x18:
        CPUC0  = 0 ==> RTICLK / 2^32
        CPUC0 /= 0 ==> RTICLK / (n+1)
        CPUC0  = 1 ==> RTICLK / 2     (smallest counter step width = 2 --> factor 2)
      */
      /*
        RTICLK = VCLK = HCLK / 2      (factor 2)
      */
      cycles_RTI_code = cycles_RTI * 4; /* factor 2*2 to compensate counting of every 2nd VCLK
clock pulse */
      time_RTI_code = cycles_RTI_code / (f_HCLK); // time_code [us], f_HCLK [MHz]
#endif //RTI_Cycle

#ifdef GIO_Toggle
      // to be measured with oscilloscope
#endif //GIO_Toggle


while (1);

}
```

## 6 References

- *Recommended Initializations for TMS570LS20x/10x Microcontrollers* (SPNA119)
- *Initialization of Hercules™ ARM® Cortex™-R4F Microcontrollers* (SPNA106)
- HAL Code Generator tool: http://www.ti.com/tool/halcogen
- Herstellerinitiative Software (HIS): http://www.automotive-his.de/

# IMPORTANT NOTICE AND DISCLAIMER

TI PROVIDES TECHNICAL AND RELIABILITY DATA (INCLUDING DATA SHEETS), DESIGN RESOURCES (INCLUDING REFERENCE DESIGNS), APPLICATION OR OTHER DESIGN ADVICE, WEB TOOLS, SAFETY INFORMATION, AND OTHER RESOURCES “AS IS” AND WITH ALL FAULTS, AND DISCLAIMS ALL WARRANTIES, EXPRESS AND IMPLIED, INCLUDING WITHOUT LIMITATION ANY IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE OR NON-INFRINGEMENT OF THIRD PARTY INTELLECTUAL PROPERTY RIGHTS.

These resources are intended for skilled developers designing with TI products. You are solely responsible for (1) selecting the appropriate TI products for your application, (2) designing, validating and testing your application, and (3) ensuring your application meets applicable standards, and any other safety, security, regulatory or other requirements.

These resources are subject to change without notice. TI grants you permission to use these resources only for development of an application that uses the TI products described in the resource. Other reproduction and display of these resources is prohibited. No license is granted to any other TI intellectual property right or to any third party intellectual property right. TI disclaims responsibility for, and you will fully indemnify TI and its representatives against, any claims, damages, costs, losses, and liabilities arising out of your use of these resources.

TI’s products are provided subject to TI’s Terms of Sale or other applicable terms available either on ti.com or provided in conjunction with such TI products. TI’s provision of these resources does not expand or otherwise alter TI’s applicable warranties or warranty disclaimers for TI products.

TI objects to and rejects any additional or different terms you may have proposed.

Mailing Address: Texas Instruments, Post Office Box 655303, Dallas, Texas 75265