

LatticeMico32 Timer

The LatticeMico32 timer is a highly configurable countdown timer with a WISHBONE-compliant slave interface compatible with the LatticeMico32 microprocessor.

Version

This document describes the 3.0 version (formerly the 7.0 SP2 version) of the LatticeMico32 timer.

Features

The LatticeMico32 timer includes the following features:

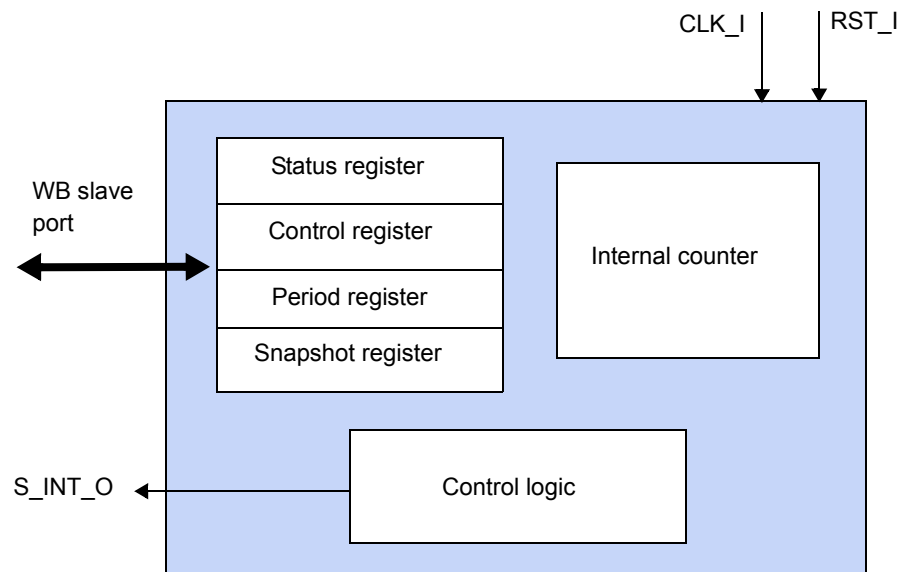
- ◆ WISHBONE B.3 interface
- ◆ RTL-configurable period-counter width up to 32 bits
- ◆ RTL-configurable period-counter access modes (read and write)
- ◆ RTL-configurable start and stop controls for software-controlled start and stop
- ◆ Maskable interrupt generation on countdown reaching zero
- ◆ Software programmable single-shot and continuous countdown modes of operation

For additional details about the WISHBONE bus, refer to the *LatticeMico32 Processor Reference Manual*.

Functional Description

The LatticeMico32 timer uses the WISHBONE clock to drive its counter. The timer is implemented using a count-down register and can be programmed to generate an interrupt request when the count reaches zero. The timer is shown in Figure 1.

Figure 1: LatticeMico32 Timer



Details on each of these registers are given in “Register Definitions” on page 5.

Configuration

The following sections describe the graphical user interface (UI) parameters, the hardware description language (HDL) parameters, and the I/O ports that you can use to configure and operate the LatticeMico32 timer.

UI Parameters

Table 1 shows the UI parameters available for configuring the LatticeMico32 timer through the Mico System Builder (MSB) interface.

Table 1: Timer UI Parameters

Dialog Box Option	Description	Allowable Values	Default Values
Instance Name	Specifies the name of the timer instance.	Alphanumeric and underscores	timer
Base Address	Specifies the base address for configuring the timer device. The minimum boundary alignment is 0X80.	0X00000000–0XFFFFFFF	0X00000000
Options			
Writeable Tick Count ¹	Determines whether the period register is writable.	1 – true 0 – false	1
Readable Tick Count	Determines whether the snapshot register is readable.	1 – true 0 – false	1
Start Stop Control	Determines whether the START and STOP bits are controllable.	1 – true 0 – false	1
Settings			
Default Reload Ticks	Specifies the initial countdown value. The period number is the reload value used to initialize the counter. The value should be in the range that can be represented by the period width.	0X00000000–0XFFFFFFF	20 (0X14)
Counter Width	Controls the period register and internal counter's width. The period width is the width of the adder-subtractor used to implement the counter.	1-32	32

¹ A tick is a clock cycle.

HDL Parameters

Table 2 lists the parameters that appear in the HDL.

Table 2: Timer HDL Parameters

Parameter Name	Description	Allowable Values
BASE_ADDRESS	Specifies the base address for configuring the timer device.	0X00000000–0XFFFFFFF
WRITEABLE_PERIOD	Determines whether the period register is writable.	1 (true), 0 (false)
READABLE_SNAPSHOT	Determines whether the snapshot register is readable.	1 (true), 0 (false)
START_STOP_CONTROL	Determines whether the start and stop bits are controllable.	1 (true), 0 (false)
PERIOD_NUM	Specifies the initial countdown value, in decimal. The period number is the reload value used to initialize the counter. The value should be in the range that can be represented by the period width.	0X00000000–0XFFFFFFF
PERIOD_WIDTH	Controls the period register and internal counter's width. The period width is the width of the adder-subtractor used to implement the counter.	1-32

I/O Ports

Table 3 describes the input and output ports of the LatticeMico32 timer.

Table 3: Timer I/O Ports

I/O Port	Active	Direction	Initial State	Description
System Clock and Reset				
CLK_I	HIGH		X	Input clock signal
RST_I	HIGH		X	Reset signal
WISHBONE Slave Interface				
S_ADR_I	XX		X	Slave address bus
S_DAT_I	XX		X	Slave data input bus
S_WE_I	HIGH		X	Slave write enable signal
S_STB_I	HIGH		X	Slave strobe signal
S_CYC_I	HIGH		X	Slave cycle signal
S_CTI_I	HIGH		X	Slave cycle type indicator
S_BTE_I	HIGH		X	Slave burst type
S_LOCK_I	HIGH		X	Slave bus locked

Table 3: Timer I/O Ports

I/O Port	Active	Direction	Initial State	Description
S_SEL_I	HIGH		X	Slave select signal
S_DAT_O	XX		0	Slave output data bus
S_ACK_O	HIGH		0	Slave acknowledge signal
S_RTY_O	HIGH		0	Slave retry
S_ERR_O	HIGH		0	Slave error
Other Auto-Connected Internal Signals				
S_INT_O	HIGH		0	Slave interrupt signal to master (CPU)

User Impact of Initial State

On power-up or reset, the timer stops, the period counter is loaded with the Default Reload Ticks value, and the interrupt is disabled.

Register Definitions

The LatticeMico32 timer includes the registers shown in Table 4.

Table 4: Register Map

Register Name	Offset	31-4	3	2	1	0
Status	0x00	Reserved			RUN	TO
Control	0x04	Reserved	STOP	START	CONT	ITO
Period	0x08	Period register				
Snapshot	0x0c	Snapshot register				

Note

Offsets in the register table must be listed in C hexadecimal and use a minimum of two hexadecimal digits with zero padding.

Table 5 through Table 8 provide details about each register in the LatticeMico32 timer.

Table 5: Status Register Bit Definition

Register Name	Bit	Default	Access Mode	Description
TO	0	0	Read/write	When the internal counter reaches zero, the timeout (TO) bit is set to 1. After it has been set, the TO bit remains set until it is cleared by a master component. You can clear the TO bit by writing zero to the status register.
RUN	1	0	Read only	When the internal counter is running, the RUN bit is read as 1. When the internal counter is not running, it is read as 0. A write operation to the status register does not change the RUN bit.
Reserved	31:2	0	Read/write	Reserved

Table 6: Control Register Bit Definition

Register Name	Bit	Default	Access Mode	Description
ITO	0	0	Read/write	Specifies the interrupt enable signal. It is active high. The default is 0. Write 1 to enable interrupt requests and 0 to disable them.
CONT	1	0	Read/write	The continuous (CONT) bit determines how the internal counter behaves when it reaches zero. If the CONT bit is 1, the counter will keep running until it is stopped by the STOP bit. If the CONT bit is 0, the counter will stop when it reaches zero. When it reaches zero, the counter reloads with the 32-bit value stored in the period register, regardless of the CONT bit. When START_STOP_CONTROL is turned off, the timer keeps running and is not affected by the value of this bit.
START	2	0	Read/write	The START bit enables the counter when a write operation is performed. Writing a 1 to the START bit causes the internal counter to begin counting down. If the timer is stopped before reaching zero, writing a 1 to the START bit will cause the timer to restart counting from the number currently held in its counter. If the timer is already running, writing a value to START will have no effect.
STOP	3	0	Read/write	The STOP bit disables the counter when a write operation is performed. Writing a 1 to the STOP bit causes the internal counter to stop. The STOP bit has no effect if the timer is already stopped, if a 0 is written to the STOP bit, or if the timer hardware has been configured with the START_STOP_CONTROL bit option turned off. An undefined result is produced when a 1 is written to both the START and STOP bits simultaneously.
Reserved	4:31	0	Read/write	Reserved

Table 7: Period Register (0x08)(W/R)

Register Name	Description
Period [<i>period_width</i> -1:0]	<p>Controls the period register and the width of the internal counter. The period width is the width of the adder-subtractor used to implement the counter.</p> <p>The internal counter is loaded with the <i>period_width</i>-bit value stored in the period register whenever a write operation to the period register occurs or the internal counter reaches zero.</p>

When START_STOP_CONTROL is turned on, writing the period register updates the internal counter, and the countdown continues. If the START_STOP_CONTROL option is turned off, writing the period register does not affect the internal counter. When the hardware is configured with the WRITEABLE_PERIOD option disabled, writing the period register causes the counter to reset to the fixed PERIOD_NUM value.

Table 8: Snapshot Register (0x0c)(R)

Register Name	Description
Snapshot [<i>period_width</i> -1:0]	<p>Specifies the snapshot value of the internal counter.</p> <p>The internal counter is loaded with the <i>period_width</i>-bit value stored in the period register whenever a write operation to the period register occurs or the internal counter reaches zero.</p>

Note

A WISHBONE master port can request a snapshot of the current *period_width*-bit internal counter by performing a master read operation to the snapshot registers. Requesting a snapshot does not change the internal counter's operation.

The structure shown in Figure 2 depicts the register map layout for the timer component. The elements are self-explanatory and are based on the register map shown in Table 4. This structure, which is defined in the MicoTimer.h header file, enables you to directly access the timer registers, if desired. It is used internally by the device driver for manipulating the timer.

Figure 2: Timer Register Map Structure

```
/* mico-timer register structure */
typedef struct st_MicoTimer{
    volatile unsigned int Status;
    volatile unsigned int Control;
    volatile unsigned int Period;
    volatile unsigned int Snapshot;
}MicoTimer_t;
```

Interrupt Behavior

When the internal counter reaches zero and the counter stops, the ITO bit of the control register is set to 1 and the timer generates an interrupt request. One of the following methods can be used to acknowledge the interrupt request:

- ◆ Clear the TO bit of the status register.
- ◆ Clear the ITO bit of the control register to disable the interrupt.

The timer keeps the IRQ asserted, even if the counter keeps counting down, until the interrupt is acknowledged by writing a 0 to the status register or until the interrupt is masked by writing a 0 to the ITO bit of the control register.

Timing Diagrams

The timing diagrams featured in Figure 3 and Figure 4 show the timing of the timer's WISHBONE and external signals.

Figure 3 shows how the timer's master ports update the data in the internal register.

Figure 3: Write Internal Register

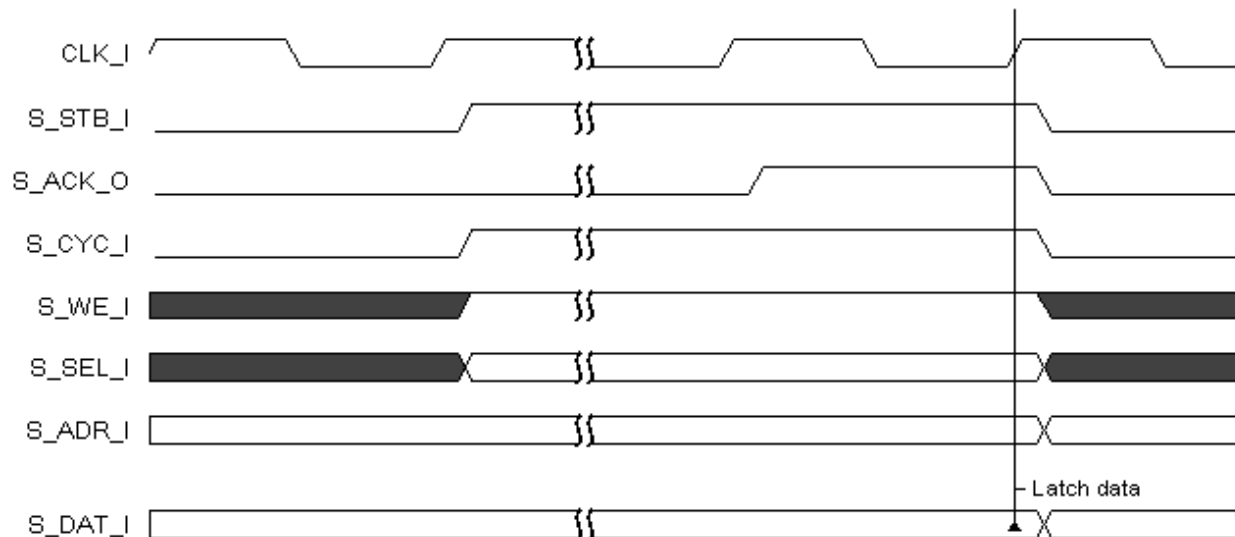


Figure 4 shows how the timer's master ports read the data in the internal register.

Figure 4: Read Internal Register

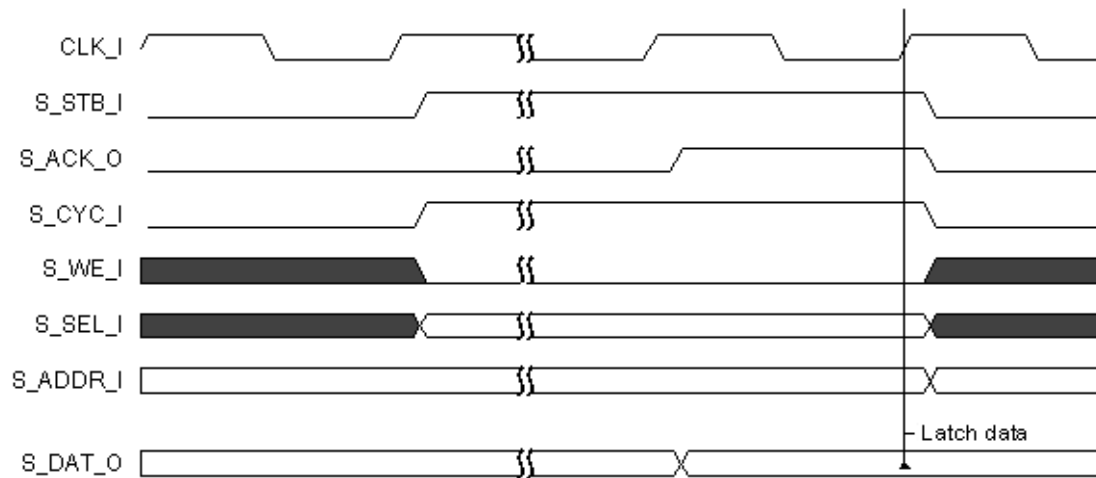
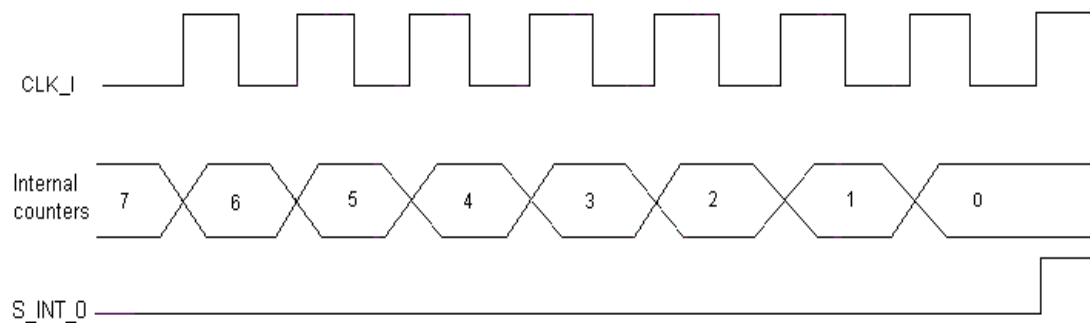


Figure 5 shows that the interrupt goes high when the timer hits 0.

Figure 5: When Interrupt Occurs



EBR Resource Utilization

The LatticeMico32 timer uses no EBRs.

Software Support

This section describes the software support provided for the LatticeMico32 timer. After discussing the usage model for the timer, it describes the timer device driver that directly interacts with a timer instance, then goes on to describe the timer services that manage multiple timer instances and any related collective service. Code examples are provided at the end of this section that show the typical usage of the software.

The supporting routines are meant for use in a single-threaded environment. If they are used in a multi-tasking environment, you must provide re-entrance protection.

Usage Model

The LatticeMico32 timer is a countdown timer. When you activate it by writing to the control register, it starts with the period count value and counts down to zero. You can operate it in continuous mode so that when it reaches zero, the period count is reloaded and the countdown continues. Alternatively, you can operate it in a single-shot mode so that when it reaches zero, it stops automatically. In either mode of operation, you can explicitly stop the timer countdown by writing an appropriate value to the control register. You can reload the period count at run time if it is configured for write access as part of the timer RTL configuration. You can read the countdown value as the timer counts down if the period count is configured for read access as part of the timer RTL configuration.

The actual usage of the timer depends on the end application. Some typical usage scenarios are the following:

- ◆ Operating the timer in a continuous mode to generate periodic timer interrupt requests for performing periodic software activities such as generating a system tick for an operating system
- ◆ Operating the timer in a single-shot mode and using the start and stop control in conjunction with the ability to read the countdown value for performing timing measurements

Device Driver

The timer device driver directly interacts with a timer instance. This section describes the type definitions for the timer device context structure.

This structure, shown in Figure 6, contains timer component-specific information and is dynamically generated in the DDStructs.h header file. This information is largely filled in by the MSB managed build process, which extracts the timer component-specific information from the platform definition

file. The members should not be manipulated directly, because this structure is for exclusive use by the device driver.

Figure 6: Timer Device Context Structure

```
typedef struct st_MicoTimerCtx_t {
    const char* name;
    unsigned int base;
    unsigned int intrLevel;
    DeviceReg_t lookupReg;
    void * userCtx;
    void * callback;
    void * prev;
    void * next;
} MicoTimerCtx_t;
```

Table 9 describes the parameters of the timer device context structure shown in Figure 6.

Table 9: Timer Device Context Structure Parameters

Parameter	Type	Description
name	const char *	Is a pointer to the timer instance name
base	unsigned int	Specifies the base address of the timer instance
intrLevel	unsigned int	Is the CPU interrupt to which the timer instance's interrupt line is connected
lookupReg	DeviceReg_t	Used by the device driver to register the timer component instance with the LatticeMico32 lookup service. Refer to the <i>LatticeMico32 Software Developer User Guide</i> for a description of the DeviceReg_t data type.
userCtx	void *	Used by the device driver to store the data pointer that you provided when starting the timer. The data pointer is provided as a argument to the callback function.
callback	void *	Used by the device driver to store the user-provided timer callback function pointer.
prev	void *	Used by the device driver service to keep track of registered timer instances
next	void *	Used by the device driver service to keep track of registered timer instances

Functions

This section describes the implemented device-driver-specific functions.

MicoTimerInit Function

```
void MicoTimerInit(MicoTimerCtx_t *ctx)
```

This function initializes a timer instance. It is called as part of the platform initialization for managed builds for each instance of the timer. It sets the timer in a known stopped state for future use and registers this timer for device lookup service.

Table 10 describes the parameter in the MicoTimerInit function syntax.

Table 10: MicoTimerInit Function Parameter

Parameter	Description	Notes
MicoTimerCtx_t *	Pointer to a timer context	For a managed build, the structure referenced is located in the DDStructs.c file.

MicoTimerStart Function

```
mico_status MicoTimerStart(MicoTimerCtx_t *ctx, TimerCallback_t
callback, void *priv, unsigned int timerCount, int periodic)
```

This function programs a timer for countdown, using the parameters that you supply. Because the timer callback routine is activated as part of the timer interrupt service routine (ISR), the interrupt requests are disabled. You must not enable interrupt requests as part of the callback routine, and you must keep the processing to a minimum to avoid large interrupt latencies. The timer ISR, which is invisible to you, acknowledges the timer interrupt.

Table 11 describes the parameters in the MicoTimerStart function syntax.

Table 11: MicoTimerStart Function Parameters

Parameter	Description	Notes
MicoTimerCtx_t *	Pointer to a timer context	Pointer to the desired timer instance's context information
TimerCallback_t	Pointer to the user-provided timer callback function with the following prototype: <pre>typedef void(*TimerCallback_t) (void *);</pre>	The timer callback routine is called by the ISR timer when the timer count expires and generates an interrupt request.
void *	Pointer to user-supplied data	Pointer provided as an argument in the timer callback routine

Table 11: MicoTimerStart Function Parameters (Continued)

Parameter	Description	Notes (Continued)
unsigned int	Timer count	Used for programming the timer count. You must ensure that the value specified corresponds to the appropriate timer count bit width.
int	Periodicity	If set to non-zero, the timer is programmed to run continuously. Otherwise, the timer is set to run once.

Table 12 shows the values returned by the MicoTimerStart function.

Table 12: Values Returned by MicoTimerStart Function

Return Value	Description
MICO_STATUS_E_INVALID_PARAM	Returned if: <ul style="list-style-type: none"> ◆ Context is null ◆ Callback is null ◆ Timer count is zero
0	Success

MicoTimerStop Function

```
mico_status MicoTimerStop(MicoTimerCtx_t *ctx)
```

This function stops a timer. It is possible for the timer to expire while this routine is being processed, triggering the ISR.

Table 13 describes the parameter in the MicoTimerStop function syntax.

Table 13: MicoTimerStop Function Parameter

Parameter	Description
MicoTimerCtx_t *	Pointer to a timer context

Table 14 shows the values returned by the MicoTimerStop function.

Table 14: Values Returned by MicoTimerStop

Return Value	Description
MICO_STATUS_E_INVALID_PARAM	Returned if the pointer is null.
0	Success

MicoTimerSnapshot Function

```
unsigned int MicoTimerSnapshot(MicoTimerCtx_t *ctx)
```

This function reads the timer snapshot register.

Table 15 describes the parameter in the MicoTimerSnapshot function syntax.

Table 15: MicoTimerSnapshot Function Parameter

Parameter	Description	Notes
MicoTimerCtx_t *	Pointer to valid timer context structure	The timer operates in countdown mode, so the returned value must be subtracted from the initial count to extract the elapsed count.

Table 16 shows the value returned by the MicoTimerSnapshot function.

Table 16: Value Returned by MicoTimerSnapshot Function

Return Value	Description
unsigned int	Timer snapshot value

Services

The timer device driver registers timer instances with the LatticeMico32 lookup service by using their instance names for device names and “TimerDevice” as the device type.

For information on the LatticeMico32 lookup service, refer to the *LatticeMico32 Software Developer User's Guide*.

Timer services also implement system-specific timer functionality. This section describes the type definitions, structures, and functions of the timer services.

Type Definitions and Structures

This section explains the timer services type definitions for the LatticeMico32 timer component.

System Timer Callback Type

```
typedef void (* MicoSysTimerActivity_t) (void *);
```

This structure declares the type of system timer callback routine required when you register a system tick activity.

Functions

This section describes the application programming interface (API) of the timer service function.

MicoTimerServicesInit Function

```
void MicoTimerServicesInit(void)
```

This function initializes timer services and is called internally when a timer instance is registered. You are not expected to call this routine.

RegisterSystemTimer Function

```
MicoTimerCtx_t* RegisterSystemTimer(MicoTimerCtx_t *, unsigned int)
```

This function enables registering a timer instance as the system timer. There can be only one system timer. When a system timer is registered, it cannot be unregistered.

Table 17 shows the parameters in the MicoTimerCtx_t* RegisterSystemTimer function syntax.

Table 17: MicoTimerCtx_t* RegisterSystemTimer Function Parameters

Parameter	Description
MicoTimerCtx_t *	Pointer to a valid timer instance context structure
unsigned int	Time between system ticks, in milliseconds

Table 18 shows the value returned by the MicoTimerCtx_t* RegisterSystemTimer function.

Table 18: Value Returned by MicoTimerCtx_t* RegisterSystemTimer Function

Return Value	Description
MicoTimerCtx_t *	Pointer to the current system timer's context information. If none exists, the value is 0.

MicoGetCPUTicks Function

```
void MicoGetCPUTicks(unsigned long long int *ticks)
```

This function returns the elapsed CPU ticks (not system ticks) as a 64-bit value, beginning from the registration of a system timer. If no system timer is registered, it returns 0. This process is described in detail in the *LatticeMico32 Software Developer User Guide*.

Table 19 describes the parameter in the MicoGetCPUTicks function syntax.

Table 19: MicoGetCPUTicks Function Parameter

Parameter	Description
unsigned long long int*	Pointer to a 64-bit value that contains the elapsed CPU ticks

MicoRegisterActivity Function

```
void MicoRegisterActivity(MicoSysTimerActivity_t activity, void *ctx)
```

This function allows the registration of a function that is activated on each system tick. It is activated through a timer ISR, so it runs at interrupt level. Interrupt requests must not be enabled within this activity function. There can be only one system timer activity function.

Table 20 describes the parameters in the MicoRegisterActivity function syntax.

Table 20: MicoRegisterActivity Function Parameters

Parameter	Description	Notes
MicoSysTimerActivity *	Pointer to a system timer activity function	
void *	Pointer to user-specific data	Pointer passed as an argument in the system timer activity function

Software Usage Examples

This section provides two examples of typical LatticeMico32 timer software usage.

The example code in Figure 7 demonstrates the following timer driver usage:

- ◆ Starting the timer, including registering a callback routine
- ◆ Stopping the timer
- ◆ Reading the snapshot timer value

Figure 7: Directly Accessing the Timer

```
void TimerInterrupt(void *ctx)
{
    return;
}
```


Figure 7: Directly Accessing the Timer

```

main() {

    MicoTimerCtx_t *pTestTimer;
    Const char *TEST_TIMER = "timer";
    unsigned int ElapsedTicks;

    /* fetch timer-context by name. */
    pTestTimer = (MicoTimerCtx_t *)MicoGetDevice(TEST_TIMER);

    /* if the device could not be found, don't perform the tests
    */
    if(pTestTimer == (MicoTimerCtx_t *)0){
        printf("failed to find timer: %s\n", TEST_TIMER);
    }

    /* Start the timer for a value of 2 seconds */
    MicoTimerStart(pTestTimer, TimerInterrupt, NULL,
    MILLISECONDS_TO_TICKS(2000), 1);

    /* wait for 1 second through s/w loop */
    MicoSleepMillisecs(1000);

    /* read timer-count snapshot */
    ElapsedTicks = MILLISECONDS_TO_TICKS(2000) -
    MicoTimerSnapshot(pTestTimer);

    /* stop timer */
    MicoTimerStop(pTestTimer);
}

```

The example code in Figure 8 demonstrates the following timer services usage:

- ◆ Registering a timer instance as the system timer
- ◆ Fetching CPU ticks elapsed since registration

Figure 8: Using the System Timer

```

Const char *TEST_TIMER = "timer";
unsigned int ElapsedTicks;

/* fetch timer-context by name. */
pTestTimer = (MicoTimerCtx_t *)MicoGetDevice(TEST_TIMER);

```

Figure 8: Using the System Timer (Continued) (Continued)

```

/* if the device could not be found, don't perform the tests
*/
if(pTestTimer == (MicoTimerCtx_t *)0){
    printf("failed to find timer: %s\n", TEST_TIMER);
}

/* Start the timer for a value of 2 seconds */
MicoTimerStart(pTestTimer, TimerInterrupt, NULL,
MILLISECONDS_TO_TICKS(2000), 1);

/* wait for 1 second through s/w loop */
MicoSleepMillisecs(1000);

/* read timer-count snapshot */
ElapsedTicks = MicoTimerSnapshot(pTestTimer);

/* stop timer */
MicoTimerStop(pTestTimer);
}
const char *const TEST_TIMER = "timer";
#define SYSTEM_TIMER_TIME_TO_WAIT_MS    (5)
unsigned long long int begin;
unsigned long long int end;

/* fetch timer-context by name. */
pTestTimer = (MicoTimerCtx_t *)MicoGetDevice(TEST_TIMER);

/* if the device could not be found, don't perform the tests
*/
if(pTestTimer == (MicoTimerCtx_t *)0){
    printf("failed to find timer: %s\n", TEST_TIMER);
    while(1){};
}

end = begin;
RegisterSystemTimer(pTestTimer, 2000);
MicoGetCPUTicks(&begin);
MicoSleepMillisecs(SYSTEM_TIMER_TIME_TO_WAIT_MS);
MicoGetCPUTicks(&end);
end = end - begin;

```
