PIC32MX Timer Library Documentation

Revision 1.0

Luka Gacnik

April 2023

Table of contents

Table of figures

Acronyms

MCU Microcontroller Unit

TMR Timer

SFR Special Function Register

PPS Peripheral Pin Select

ISR Interrupt Service Routine

1 Library description

The library consists of basic functions that control the Timer modules of an PIC32MX series of the Microchip microcontrollers.

1.1 About the timer

A timer is a circuit which uses can use a clock (or an external) signal to increment a register. Such a timer can be used for various purposes such as generic scheduling, timeout, manually triggered delayed interrupt, and more. A special type of a timer is called a Watchdog timer and is not provided by this library. Additionally, a clock could be implemented using timers, however usually inside microcontrollers there is a Real-time clock and calendar module for this purpose.

The library is based on the PIC32MX MCU series. Some of the key features of its Timer modules are:

- 16-bit synchronous/asynchronous timer/counter with gate
- 16-bit or 32-bit synchronous timer/counter with gate and Special Event Trigger

All timers include the following common features:

- 16-bit timer/counter
- Software-selectable internal or external clock source
- Programmable interrupt generation and priority
- Gated external pulse counter

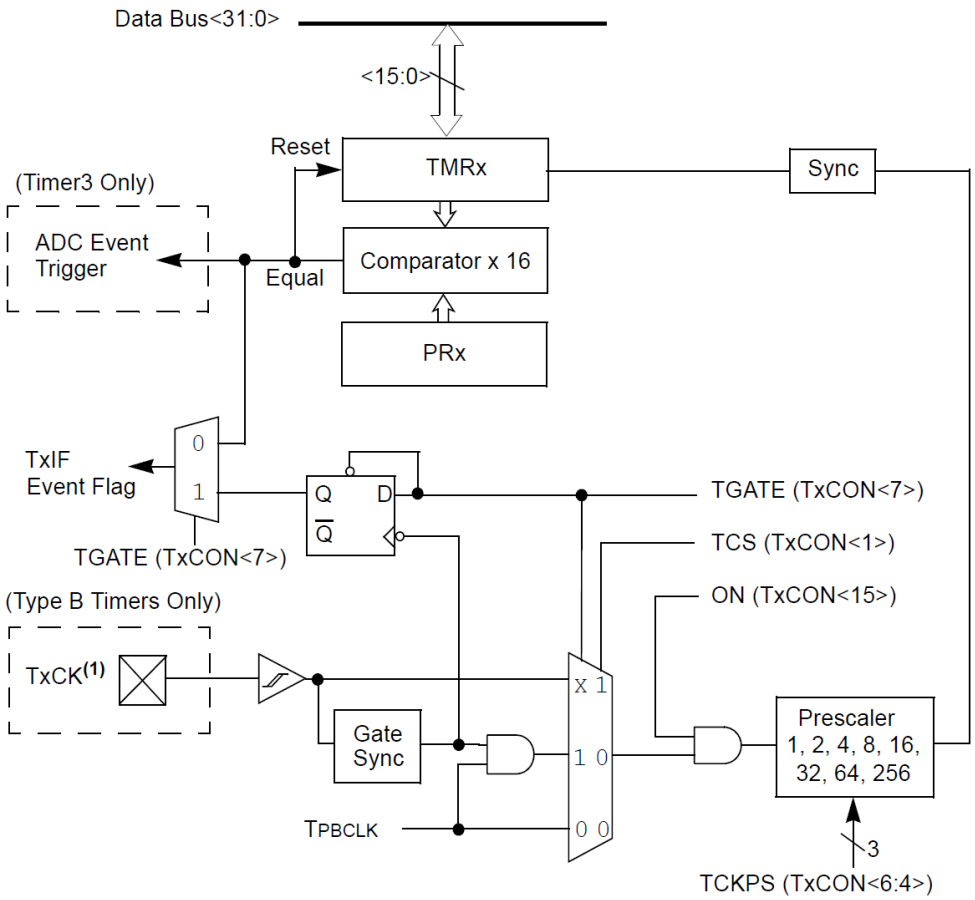

Figure 1.1: Type B timer of PIC32MX series

1.2 Library features

Currently, the Timer driver is capable of performing the following operations:

- Generating a polling-based delay
- Generating an interrupt-based delay (timeout mode)
- Measuring the presence of an external signal (gated mode) with optional falling-edge triggered event

For the listed features all five timer modules are available in both modes.

2 Dependencies

The library consists of the following source and header files:

- `Tmr.c`
- `Tmr.h`
- `Tmr_sfr.h`

All the external user-accessible function prototypes are provided in the `Tmr.h` file. Their definitions and other local functions like ISR handlers are contained in the `Tmr.c` file. The file `Tmr_sfr.h` contains all the macro definitions for register bit-fields and SFR memory layout type.

The library depends on the following other libraries:

- `Pio.h`
- `Ic.h`
- `Osc.h`

The library `Pio.h` provides the functions configuring a `TxCK` pin with Peripheral Pin Select (PPS) module. Interrupts are configured using the `Ic.h` library. Lastly, the library `Osc.h` is used for reading peripheral clock frequency and determining clock source. As a side note, for the PIC32MX MCUs the PPS module must be configured for a proper SPI operation within the scope of the `Pio.h` library. Such an action requires (un)locking a specific set of registers which is provided by the `Cfg.h` library and is already contained within the `Pio.h` library.

Additionally, there needs to be one of the standard Microchip libraries included (since the compiler XC32 is the made by the Microchip), that is the `sys\attrib.h`

library. It is used for its ISR macro definition that is needed for ISR function attribute definition. Lastly, its important to provide the fact that this library was meant to be used with a specific MCU - the PIC32MX170Bxxx series made by the Microchip. As previously mentioned the XC32 compiler was used to compile the library files, meaning there are some specifics in the library which are unique to the XC32 compiler (such as ISR function attribute). In case any other MCU of the PIC32 family would be used, some peripheral libraries (used by this driver) might need modifications (like different register sets, etc.).

3 Notes and warnings

Some code related facts must be known prior to using the SPI library:

a) **Multi-module operation**

The code from this library is structured in a way which allows multiple timer modules to be used simultaneously. Only in timeout mode a structure of 5 members is required to store relevant data in case of simultaneous usage of multiple timers.

b) **Interrupt priority of a Timer module**

Prior to any of the interrupt-based functions is used, the (sub)priority level of the corresponding Timer module's interrupt vectors must be defined. This is done via providing a constant value using macros `TMRx_ISR_IPL`, `TMRx_ICX_IPL`, and `TMRx_ICX_ISL` (where lowercase `x` can be in the range of 1 to 5).

c) **System clock frequency macro**

Prior to polling-based delay function use one must set the system clock frequency macro `TMR_DELAY_SYSCLK` to a valid value. If not set by the user, default value of `8000000` is used.

d) **Paired modules for 32-bit mode**

The PIC32MX series comes with 5 Timer modules. All are 16-bit timers. If a 32-bit timer is desired then two modules are used. If the `TIMER_2` is set for 32-bit operation then also the `TIMER_3` is configured and used (same goes for `TIMER_4` and `TIMER_5`). Both `TIMER_2` and `TIMER_4` module base address must be used when configuring a timer for the 32-bit operation The `TIMER_1` is only capable of 16-bit mode and cannot be paired with any other module.

e) **No Reset Allowed for Core Timer**

Core timer is used by the delay functions and an ISR, triggered on a periodical 1 ms time interval basis. The latter may be used by specific processes, required by different libraries (the amount of callbacks is fixed and limited). Therefore, the

Core timer shall not be reset, ever. A wrap-up (overflow) of count register doesn't result in faulty timeout behavior if the condition (6.1) is satisfied.

4 Future ideas

None.

5 Custom types

The library comes with a set of custom made structure or enumeration types. For the sake of convenience these types are explained throughout this chapter. Note that individual enumeration types are not covered here because they (and their set of values) follow self-explanatory naming conventions.

5.1 TmrTimeoutConfig_t

This structure-based type contains the properties for configuration of the selected Timer module when it operates in interrupt-based timeout mode. The type consists of the following properties:

- **`bitMode`** (*`TmrBitMode_t`*) - a bit mode of a timer (a.k.a. increment register bit-width) which limits timer's counting range
- **`clkDiv`** (*`TmrClkDiv_t`*) - a division factor of a clock source
- **`clkSrc`** (*`TmrClkSource_t`*) - a selection of a clock source that is used to increment the timer's register
- **`timeUnit`** (*`TmrTimeUnit_t`*) - a unit in which a period of a timer is set (using one of the available functions)

5.2 TmrGatedConfig_t

This structure-based type contains the properties for configuration of the selected Timer module when it operates in interrupt-based gated mode. The type consists of the following properties:

- **bitMode** (*TmrBitMode_t*) - a bit mode of a timer (a.k.a. increment register bit-width) which limits timer's counting range
- **clkDiv** (*TmrClkDiv_t*) - a division factor of a system clock signal which is used as incrementing source
- **tckPin** (*uint32_t*) - an input pin that is used as a gate for a timer
- **isGateCont** (*Bool_t*) - defines whether measuring of external signal continues after falling-edge or not

5.3 TmrSfr_t

This structure-based type contains the properties for modification of Timer related SFRs. Throughout this library it is mostly used as a pointer to a base address of any Timer module. MCU memory contents starting at such a specific base address are modified when changing the value of any member that is contained by this structured type. The way of each member is positioned in the structure is strictly defined and shall not be modified by the user. Member layout defines the exact position of Timer SFRs, relative to any of Timer base address.

6 Function description

The library includes the following functions:

- `TMR_ConfigTimeoutModeSfr()`
- `TMR_ConfigGatedModeSfr()`
- `TMR_SetCallback()`
- `TMR_SetCoreTimerCallback()`
- `TMR_SetTimeoutPeriod()`
- `TMR_StartTimer()`
- `TMR_StopTimer()`
- `TMR_DelayUs()`
- `TMR_DelayMs()`
- `TMR_ReadTimer()`
- `TMR_ReadTimerPeriod()`

The following sub-sections describe the use and operationality of these functions.

6.1 TMR_ConfigTimeoutModeSfr()

PROTOTYPE

```
TMR_ConfigTimeoutModeSfr(tmrSfr, tmrConfig)
```

DESCRIPTION

Configures Timer SFRs for timeout operation.

PARAMETERS

- **tmrSfr** (*TmrSfr_t* *) - a pointer to the base address of either of the available Timer modules
- **tmrConfig** (*TmrTimeoutConfig_t*) - contains the settings which determine operation parameters of the desired timeout operation

RETURNS

Returns `false` when either of the following is true:

- Timer base module address is incorrect
- The `TIMER_1` module is set to be configured with secondary oscillator but the latter isn't set as a current oscillator
- Configured timer module is desired to operate in 32-bit mode but the base address passed to this function doesn't match the base address of either `TIMER_2` or `TIMER_4` module

Otherwise it returns `true`.

OPERATION

Initially, two basic safety checks are done before the function continues. Then all Timer registers for the selected module are reset. Related Timer SFRs are modified afterwards. Additionally, two private structures are modified - one holds two ISR flags per timer and other one holds parameters for calculating timeout period using the `TMR_SetTimerPeriod()` function. Appropriate member is modified based on

the module base address used. Lastly, interrupt SFRs are configured and interrupts enabled.

Note that timeout is an interrupt-based delay where an ISR is triggered at the end of a timeout period. Which function executes in such an ISR can be defined by the `TMR_SetCallback()` function. If not defined then a default empty handler is called in an ISR.

NOTES

- Only the `TIMER_1` can be used with the secondary oscillator (sleep mode related)
- Either the `TIMER_2` or `TIMER_4` module base address can be used when configuring a timer for the 32-bit operation

LIMITATIONS

None.

6.2 TMR_ConfigGatedModeSfr()

PROTOTYPE

```
TMR_ConfigGatedModeSfr(tmrSfr, tmrConfig)
```

DESCRIPTION

Configures Timer SFRs for gated operation.

PARAMETERS

- **tmrSfr** (*TmrSfr_t* *) - a pointer to the base address of either of the available Timer modules
- **tmrConfig** (*TmrGatedConfig_t*) - contains the settings which determine operation parameters of the desired gated operation

RETURNS

Returns `false` when either of the following is true:

- Timer base module address is incorrect
- Pin code for the member `tckPin` of the `TmrGatedConfig` structure is not set
- Configured timer module is desired to operate in 32-bit mode but the base address passed to this function doesn't match the base address of either `TIMER_2` or `TIMER_4` module

Otherwise it returns `true`.

OPERATION

Initially, one safety check is done before the function continues. Then all Timer registers for the selected module are reset. Related Timer SFRs are modified afterwards. The user-defined `TxCK` pin is configured using PPS module. Additionally, a private structure is modified which holds two ISR flags per timer. Appropriate member is modified based on the module base address used. Lastly, interrupt SFRs are configured and interrupts enabled.

Note that gated timer operation is a situation where system clock signal increments timer's register only when the state of `TxCK` pin is high, otherwise register isn't incremented. In such a case an ISR is triggered at falling-edge event on `TxCK` pin and its user-defined handler to be executed is defined by the `TMR_SetCallback()` function.

NOTES

- Source of timer incrementing (which is gated by an external signal on the `TxCK` pin) is always the system clock signal
- Either the `TIMER_2` or `TIMER_4` module base address can be used when configuring a timer for the 32-bit operation
- Call this function after a Timer module is configured
- The gated mode could also be used only for detection of the falling-edge of an external signal

LIMITATIONS

None.

6.3 TMR_SetCallback()

PROTOTYPE

`TMR_SetCallback(tmrSfr, isrHandler)`

DESCRIPTION

Sets a handler to execute in an ISR when operating in timeout or gated mode.

PARAMETERS

- **`tmrSfr`** (*`TmrSfr_t *`*) - a pointer to the base address of either of the available Timer modules
- **`isrHandler`** (*`void (*fPtr) ()`*) - pointer to the user-defined function

RETURNS

Returns `false` if a Timer base module address is incorrect, otherwise returns `true`.

OPERATION

Directly sets one of the five (five timers, five ISRs) ISR handlers. If a timer number `x` is configured for the 32-bit mode then a timer `x+1` is initially configured for interrupt operation and interrupt vector that points to ISR location corresponds to this Timer module.

NOTES

- User-defined function which executes in the scope of an ISR should be as short as possible - only most crucial tasks to be performed during ISR execution
- This function doesn't configure interrupts and their priority, therefore call this function after a Timer module is configured

LIMITATIONS

None.

6.4 TMR_SetCoreTimerCallback()

PROTOTYPE

`TMR_SetCoreTimerCallback(isrHandler)`

DESCRIPTION

Sets a handler to execute in the Core timer ISR.

PARAMETERS

- **`isrHandler`** (*`void (*fPtr) ()`*) - pointer to the user-defined function

RETURNS

Returns `true` if callback was successfully set, otherwise if there are no more available callbacks in the Core timer ISR the functions returns `false`.

OPERATION

Sets one of available function pointers to the user-defined function whose address is passed to this function. Additionally, on the first function call after reset the Core timer interrupt (sub)priority is configured using the `CT_ICX_IPL` and `CT_ICX_ISL` macros.

NOTES

- User-defined function which executes in the scope of an ISR should be as short as possible - only most crucial tasks to be performed during ISR execution
- If a user-defined function, which was previously (some time after device reset) already set to some of the available callbacks, is to be set again, the function returns `true` without setting any additional callback
- Core timer ISR is executed on a periodical 1 ms time basis. The next timeout is not delayed for a period that is consumed by all the user-defined functions (callbacks) within the Core timer ISR

LIMITATIONS

At the time of writing this library, the amount of Core timer ISR callbacks was limited to 9 callbacks. If more callbacks are required, then user needs to manually add a callback inside the Core timer ISR, put its declaration on top of `Tmr.c` file and let is point to the `IsrDefaultHandler()` default empty callback. Additionally, newly added function pointer to a callback needs to be added to the `TMR_SetCoreTimerCallback()` function.

6.5 TMR_SetTimeoutPeriod()

PROTOTYPE

`TMR_SetTimeoutPeriod(tmrSfr, period)`

DESCRIPTION

Sets a timeout period for a specific Timer module. Requires a timer to be pre-configured for the timeout mode.

PARAMETERS

- **`tmrSfr`** (*`TmrSfr_t *`*) - a pointer to the base address of either of the available Timer modules
- **`period`** (*`uint32_t`*) - a timeout period value

RETURNS

Returns `false` when either of the following is true:

- Timer base module address is incorrect
- A timeout period is zero
- A Timer module is not configured for timeout mode

Otherwise it returns `true`.

OPERATION

Modifies a compare SFR of Timer module. Value that is written to it is calculated based on the information from a private structure which is set during either of Timer modules.

NOTES

The unit of a period value passed into this function is determined by the configuration structure of the type *`TmrTimeoutConfig_t`*, namely by the member `timeUnit`.

LIMITATIONS

The maximum period value of the timeout mode operation is determined by the bit mode that was initially configured - the 16-bit (maximum period of 65535) or the 32-bit (maximum period of 4294967295) mode.

6.6 TMR_StartTimer()

PROTOTYPE

`TMR_StartTimer(tmrSfr)`

DESCRIPTION

Starts a previously configured Timer module.

PARAMETERS

- **`tmrSfr`** (*`TmrSfr_t *`*) - a pointer to the base address of either of the available Timer modules

RETURNS

None.

OPERATION

Clears timer incrementing register and starts a timer by setting a bit in a configuration SFR.

NOTES

This function has no safety check for a valid base address. This is because starting a timer should be as quick as possible. For the same reason is this function always inlined.

LIMITATIONS

None.

6.7 TMR_StopTimer()

PROTOTYPE

`TMR_StopTimer(tmrSfr)`

DESCRIPTION

Stops a previously started Timer module.

PARAMETERS

- **`tmrSfr`** (*`TmrSfr_t *`*) - a pointer to the base address of either of the available Timer modules

RETURNS

None.

OPERATION

Clears a bit in a configuration SFR.

NOTES

This function has no safety check for a valid base address. This is because stopping a timer should be as quick as possible. For the same reason is this function always inlined.

LIMITATIONS

None.

6.8 TMR_DelayUs()

PROTOTYPE

`TMR_DelayUs(period)`

DESCRIPTION

A polling-based delay that counts microseconds.

PARAMETERS

- **`period`** (*`uint32_t`*) - a delay period in microseconds

RETURNS

None.

OPERATION

Uses the Core timer (updates every other system clock cycle). Delay value is set using input parameter and macro `TMR_DELAY_SYSCLK` and a current count of the Core timer. Core timer is not reset.

NOTES

- Always define the `TMR_DELAY_SYSCLK` macro before using this function
- It is advisable to have peripheral clock frequency at least 10 MHz to achieve acceptable delay results for very low `period` values
- Core timer is not reset when timeout count starts however a valid timeout will be carried out even the Core timer count register overflows, however user-defined delay is limited and the condition requirement, shown by (6.1) needs to be met

LIMITATIONS

Since the core timer is a 32-bit timer, the following inequality must be true:

$$delay \cdot \frac{TMR\_DELAY\_SYSCLK}{500000} \leq 4294967295 \tag{6.1}$$

6.9 TMR_DelayMs()

PROTOTYPE

`TMR_DelayMs(period)`

DESCRIPTION

A polling-based delay that counts milliseconds.

PARAMETERS

- **`period`** (*`uint32_t`*) - a delay period in milliseconds

RETURNS

None.

OPERATION

Calls the function `TMR_DelayUs()` and multiplies the `delay` parameter by 1000.

NOTES

- Always define the `TMR_DELAY_SYSCLK` macro before using this function
- It is advisable to have peripheral clock frequency at least 1 MHz to achieve acceptable delay results for very low `period` values
- Core timer is not reset when timeout count starts however a valid timeout will be carried out even the Core timer count register overflows, however user-defined delay is limited and the condition requirement, shown by (6.2) needs to be met

LIMITATIONS

Since the core timer is a 32-bit timer, the following inequality must be true:

$$delay \cdot \frac{TMR\_DELAY\_SYSCLK}{500} \leq 4294967295 \tag{6.2}$$

6.10 TMR_ReadTimer()

PROTOTYPE

`TMR_ReadTimer(tmrSfr)`

DESCRIPTION

Reads Timer count register of a given Timer module base address.

PARAMETERS

- **`tmrSfr`** (*`TmrSfr_t *`*) - a pointer to the base address of either of the available Timer modules

RETURNS

Returns a 32-bit (16-bit if counter configured for 16-bit mode) counter register's value of the type *`uint32_t`*.

OPERATION

Direct read of a Timer module's count register.

NOTES

None.

LIMITATIONS

None.

6.11 TMR_ReadTimerPeriod()

PROTOTYPE

`TMR_ReadTimerPeriod(tmrSfr)`

DESCRIPTION

Calculates time period in seconds (timeout or gate mode).

PARAMETERS

- **`tmrSfr`** (*`TmrSfr_t *`*) - a pointer to the base address of either of the available Timer modules

RETURNS

Returns the calculated value of the type *`double`*.

OPERATION

Obtains timer clock source divide factor, timer register value, and the peripheral clock frequency to calculate a period given in seconds.

NOTES

None.

LIMITATIONS

None.

7 Examples

A set of examples is provided to demonstrate a simple operation of a Timer module. Examples provide only the main operation related code. Other processes such as the start-up of an MCU, programming of the device configuration register, setting up other peripheral modules, and similar is not provided here.

7.1 Timeout operation

This example provides the code for setting up a Timer module for a timeout operation.

Initially, the `TIMER_2` module has its SFRs configured, an ISR handler set, and timeout period set as well. Then optional pin labeled as `RPB2` pin is put into a high state and timer incrementing starts. This pin can be measured to verify a valid timeout operation and timeout period. For a real-life application there might be some code after timer starts and at some point the timer is could be reset to prevent the timeout function `TestFunct()` from executing. This operation is somewhat similar to the Watchdog functionality.

```
1  /** Custom libs **/
2  #include "Tmr.h"
3
4  /** Test prototype **/
5  static void TestFunct(void);
6
7  int main(int argc, char** argv)
8  {
9     /* Configuration structure for timeout mode */
10    TmrTimeoutConfig_t tmrTimeoutConfig = {
11       .bitMode = TMR_BITMODE_32BIT,
12       .clkDiv = TMR_CLK_DIV_1,
13       .clkSrc = TMR_CLK_SRC_PBCLK,
14       .timeUnit = TMR_TIME_UNIT_US
15    };
```

```
16
17      /* Configure indication pin */
18      PIO_ConfigGpioPin(GPIO_RPB2, PIO_TYPE_DIGITAL, PIO_DIR_OUTPUT);
19
20      /* Configure TMR module for timeout mode */
21      TMR_ConfigSfrTimeoutMode(&TMR2_MODULE, tmrTimeoutConfig);
22
23      /* Configure a function pointer */
24      TMR_SetCallback(&TMR2_MODULE, TestFunct);
25
26      /* Set a timeout period */
27      TMR_SetTimerPeriod(&TMR2_MODULE, 100000);
28
29      /* Set pin (clear it after timeout) */
30      PIO_ClearPin(GPIO_RPB2);
31      PIO_SetPin(GPIO_RPB2);
32
33      /* Start a timeout */
34      TMR_StartTimer(&TMR2_MODULE);
35
36      while(1)
37      {
38          /* Main program execution */
39          // Code...
40
41          /* Stop timeout if this is reached before "TestFunct()" */
42          TMR_StopTimer(&TMR2_MODULE);
43          PIO_ClearPin(GPIO_RPB2);
44      }
45
46      return 0;
47  }
48
49
50  /** Test function **/
51  static void TestFunct(void)
52  {
53      PIO_TogglePin(GPIO_RPB2);
54  }
```

7.2 Gated timer operation

This example provides the code for setting up a Timer module for a gated operation.

Initially, the `TIMER_2` module has its SFRs configured and optional an ISR handler set. The latter is optional since the gated mode provides two functions - measuring the length of an external signal high state and responding to falling-edge with an ISR handler. The output pin `RPA2` is used for simulating an external signal which gates the timer's register incrementing. The timer starts and afterwards the pin `RPA2` is triggered three times - twice as the timers is enabled and once as the timer is disabled. At the end of the code the function `TMR_ReadTimerPeriod()` is called to provide with a measured period in seconds.

This example yields the result of approximately 0.002 seconds. If the member `isGateCont` is set to `false` then the result changes to the value of approximately 0.001 seconds since the timer is disabled after the first falling-edge.

If the member `isGateCont` is set to `true` then the gated operation can be used only to detect a falling-edge on an external signal while optionally ignoring measurement of an external signal. For the PIC32MX devices this might be beneficial to the existing Change notice (Input change) module that triggers an ISR for both the rising and the falling-edge of an external signal.

```
1  /** Custom libs **/
2  #include "Tmr.h"
3
4  /** Test prototype **/
5  static void TestFunct(void);
6
7  int main(int argc, char** argv)
8  {
9      /* Configuration structure for gated mode */
10     TmrGatedConfig_t tmrGatedConfig = {
11         .bitMode = TMR_BITMODE_32BIT,
12         .clkDiv = TMR_CLK_DIV_1,
13         .tckPin = T2CK_RPB4,
14         .isGateCont = true
15     };
```

```
16
17      /* Configure signal generator */
18      PIO_ConfigGpioPin(GPIO_RPA2, PIO_TYPE_DIGITAL, PIO_DIR_OUTPUT);
19
20      /* Configure TMR module for gated mode */
21      TMR_ConfigSfrGatedMode(&TMR2_MODULE, tmrGatedConfig);
22
23      /* Configure a function pointer (optional) */
24      TMR_SetCallback(&TMR2_MODULE, TestFunct);
25
26      /* Initial state of signal */
27      PIO_ClearPin(GPIO_RPA2);
28
29      /* Start measuring a signal */
30      TMR_StartTimer(&TMR2_MODULE);
31
32      /* Simulate a signal */
33      PIO_SetPin(GPIO_RPA2);
34      TMR_DelayUs(1000);
35      PIO_ClearPin(GPIO_RPA2);
36
37      /* Simulate absence of a signal */
38      TMR_DelayUs(1000);
39
40      /* Signal reappears */
41      PIO_SetPin(GPIO_RPA2);
42      TMR_DelayUs(1000);
43      PIO_ClearPin(GPIO_RPA2);
44
45      /* Simulate absence of a signal */
46      TMR_DelayUs(1000);
47
48      /* Stop measuring a signal */
49      TMR_StopTimer(&TMR2_MODULE);
50
51      /* Signal reappears (but doesn't get measured) */
52      PIO_SetPin(GPIO_RPA2);
53      TMR_DelayUs(1000);
54      PIO_ClearPin(GPIO_RPA2);
55
56      /* Read result of measurement */
57      double tmrPeriod = TMR_ReadTimerPeriod(&TMR2_MODULE);
```

```
58
59      return 0;
60  }
61
62
63  /** Test function **/
64  static void TestFunct(void)
65  {
66      // Code...
67  }
```

A Revision history

Revision 1.0 (Nov 2022)

This is the initial released version of this document.

Revision 1.1 (April 2023)

Added `TMR_SetCoreTimerCallback()` and some minor library fixes.