AN12912

Using the Code Watchdog Timer

Rev. 1 — 11/2020 Application Note

Contents

1 Introduction....1
1.1 Secure Counter....1
1.2 Instruction Timer....1
1.3 State....2
2 Usage....2
2.1 Secured read....3
2.2 Secured write....3
2.3 Secured function....4
2.4 Secured loop....6
2.5 Secured branch....8
2.6 Secured decision control....9
2.7 Conclusion....9
2.8 Revision history....9
3 Revision history....10

1 Introduction

The Code Watchdog Timer (CWT) is available on all LPC55S1x/LPC551x and LPC55S0x/LPC550x devices. This application note introduces the basic concepts and usage of the CWT.

The CWT provides two primary mechanisms (Secure Counter and Instruction Timer) for detecting code flow and data integrity checking.

Secure Counter (SEC_CNT): Detects altered software in the execution flow.

- This counter (that is, an accumulator) is loaded with an initial value and then the runtime software issues ADD and SUB commands to increment/decrement the counter.
- Periodically, a secure counter value check is initiated by passing the expected value to the CWT using the STOP and RESTART commands.
- If a mismatch is detected between the Secure Counter and the value passed to it, it indicates that the execution flow has been altered by a side channel attack or another suspicious activity.

Instruction Timer (INST_TIMER): Places an upper-limit on the interval between checks of the secure counter.

- The START command loads the internal decremental counter. Before the counter generates an underflow (reaches 0), a STOP or RESTART command must be executed to force a secure counter check.

For more information, see *LPC55S1x/LPC551x User Manual* (document UM11295) and *LPC55S0x/LPC550x User Manual* (document UM11424).

NOTE

This AN only introduces the usage of CWT, and does not involve system security.

1.1 Secure Counter

The Secure Counter is a 32-bit accumulating register that holds a dynamically changing value. This value can be periodically evaluated to determine if a program is executing in an expected manner. If a mismatch is detected, a FAULT is generated.

The Secure Counter is used to protect the code execution flow and also to ensure the integrity of critical data.

1.2 Instruction Timer

The Instruction Timer is a 32-bit count-down timer that is used by an application to set the number of instructions that are expected to be executed. The Instruction Timer can therefore detect cases where the counter is reset (set to 0), which may be an indication that unauthorized instructions are being executed. The application programmer pre-loads the Instruction Timer with slightly more than the number of instructions in the next execution sequence. The Instruction Timer counts the instructions as they are executed (clocks), and if the number is exhausted before the CWT is serviced, a FAULT is generated. The Instruction Timer can be programmed to either pause, or keep running, while the interrupt service routines are executing.

The Instruction Timer can be used to confirm that the corresponding code is completed within the specified execution time.

1.3 State

The STATE registers perform the following functions:

• The CWT has two legal states: IDLE and ACTIVE.

• Internally, the two states are encoded by 4 bits, with only two of the possible 16 combinations permissible.

• After any reset, including a reset generated by the CWT itself, the module is in IDLE state.

• A correct sequence of CPU writes to the CONTROL group, followed by a START command, which changes the state to ACTIVE.

• Once ACTIVE, a correctly formulated STOP COMMAND changes the state back to IDLE.

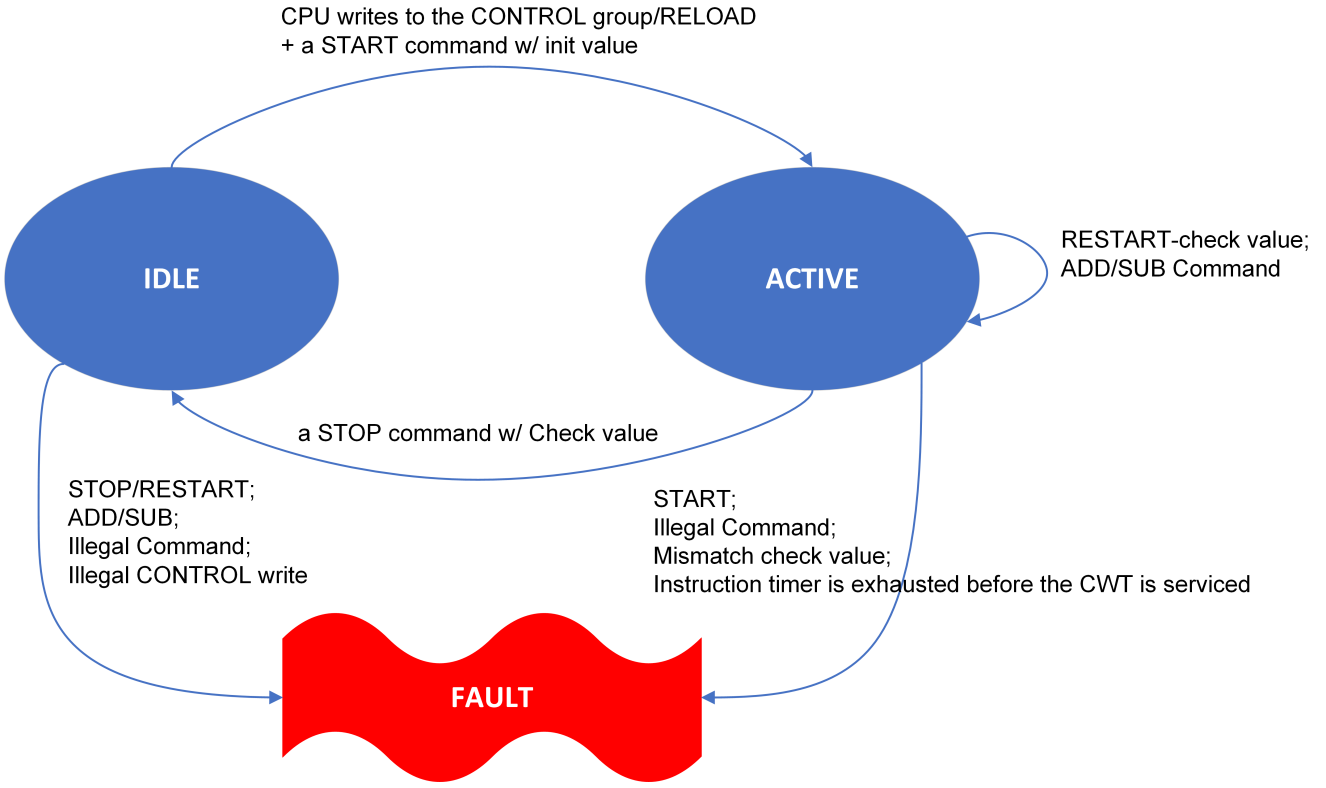

Figure 1. Code watchdog timer state

2 Usage

In this section, C-Style pseudocode is used to explain the usage and programming model of the CWT.

Table 1. C-style pseudocode

Pseudocode Function	Description
`CWT_Start(uint32_t reload, uint32_t start)`	Sets `start` to Secure Counter and `reload` to Instruction Timer values.
`CWT_Check(uint32_t check)`	This function compares the `check` value with the Secure Counter value by writing to the RESTART register. If the comparison fails, a fault is triggered.

Table continues on the next page...

Table 1. C-style pseudocode (continued)

Pseudocode Function	Description
`CWT_Stop(uint32_t stop)`	This function stops the Instruction Timer and Secure Counter. `stop` is the expected value which is compared with the value of the Secure Counter. If the comparison fails, a fault is triggered.
`CWT_Add(uint32_t add)`	This function adds a specified value to the Secure Counter. `add` denotes the value to be added.
`CWT_Sub(uint32_t sub)`	This function subtracts a specified value from the Secure Counter. `sub` is the value to be subtracted.

NOTE

To maximize the security, the reload value of Instruction Timer must be as small as possible.

2.1 Secured read

We must ensure that the values read from Flash, OTP Fuse, and memory are not maliciously modified by external attacks while the code is executing. CWT can be used to protect these critical data reads.

The pseudocode is as follows.

```
/* Secured Read */

/* Start Code Watchdog Timer */
CWT_Start(inst_timer_reload, sec_cnt_start);

/* Add the value from hardware memory(Flash,OTP fuse, memoey, etc.) to secure counter */
CWT_Add(hw_memory);

/* Read the data into a variable */
value = hw_memory;

//Using the value as needed

/* Check if secure counter is as expected */
CWT_Check(sec_cnt_start + value);

/* optional */
CWT_Sub(value);

CWT_Stop(sec_cnt_start);
```

Figure 2. Secured read

Multiple values can be protected using this method. If needed, a volatile keyword can be used to prevent compiler optimization.

2.2 Secured write

A secured write is similar to a secured read. It is important to operate on security/critical related registers or write security-related data. Ensure that the value written is the expected value.

The pseudocode is as follows.

```
/* Secured Write */

/* Start Code Watchdog Timer */
CWT_Start(inst_timer_reload, sec_cnt_start);

/* Add the value will be written to hardware register to secure counter */
CWT_Add(value);

/* Write hardware register */
hw_reg = value;

/* Check if secure counter is as expected */
CWT_Check(sec_cnt_start + hw_reg);

/* optional */
CWT_Sub(hw_reg);

CWT_Stop(sec_cnt_start);
```

Figure 3. Secured write

2.3 Secured function

This section describes how to write and call a function protected by CWT.

2.3.1 Writing a secured function

```
/* Secured function */

#define CWT_SECURED_FUNC_CONST (0x5AA5)
uint32_t CWT_secured_func(uint32_t param)
{
    uint32_t result = 0xAFFE;

    /* Check the parameter passed in */
    CWT_Check(sec_cnt_start + param);

    CWT_Sub(param);

    //Do something
    CWT_Add(1);                                    // some checkpoint

    //Do something
    CWT_Add(1);                                    // another checkpoint

    //Do something
    CWT_Add(1);                                    // one more checkpoint

    //Do something
    CWT_Add(1);                                    // checkpoint!

    //Do something
    CWT_Add(1);                                    // checkpoint!

    CWT_Check(sec_cnt_start + 5);                  // CWT check

    CWT_Add(result + CWT_SECURED_FUNC_CONST - 5);

    return result;
}
```

Figure 4. Secured Function

2.3.2 Calling a secured function

```
/* Secured Function Call */

/* Start Code Watchdog Timer */
CWT_Start(inst_timer_reload, sec_cnt_start);

CWT_Add(param);

result = CWT_secured_func(param);

CWT_Check(sec_cnt_start + CWT_SECURED_FUNC_CONST + result);

/* optional */
CWT_Sub(result);

CWT_Stop(sec_cnt_start + CWT_SECURED_FUNC_CONST);
```

Figure 5. Secured Function Call

2.4 Secured loop

We must ensure that the loop executes a certain number of times as expected. This section describes how to use CWT to implement this feature.

```
/* Secured Loop */

/* Start Code Watchdog Timer */
CWT_Start(inst_timer_reload, sec_cnt_start);

CWT_Add(loop_count);

CWT_Check(sec_cnt_start + loop_count); // CWT checkpoint

for (int i = 0; i < loop_count; i)
{
    //Do something
    //loop body remains unprotected in this example
    CWT_ADD(1);
}

CWT_Check(sec_cnt_start + (2 * loop_count)); // CWT checkpoint

/* optional */
CWT_Sub(2 * loop_count);

CWT_Stop(sec_cnt_start);
```

Figure 6. Secured Loop

NOTE

`loop_count` must be initialized in a secured way (for example, secured read) if needed.

2.5 Secured branch

```
/* Secured Branch */

/* Start Code Watchdog Timer */
CWT_Start(inst_timer_reload, sec_cnt_start);

CWT_Add(value);

switch (value)
{
case 0xF00D:
    CWT_Check(sec_cnt_start + 0xF00D); // CWT checkpoint
    CWT_Sub(0xF00D);
    //Do something
    break;
case 0xFACE:
    CWT_Check(sec_cnt_start + 0xFACE); // CWT checkpoint
    CWT_Sub(0xFACE);
    //Do something
    break;
case 0xCAFE:
    CWT_Check(sec_cnt_start + 0xCAFE); // CWT checkpoint
    CWT_Sub(0xCAFE);
    //Do something
    break;
default:
    CWT_Stop(0xBADBEEF);
    // it is highly recommended to do something here, otherwise it might stay
    // undetected if value was glitched to zero
    break;
}

CWT_Stop(sec_cnt_start);
```

Figure 7. Secured Branch

2.6 Secured decision control

```
/* Secured Decision Control */

/* Use 32-bit unique pattern to indicate true or false */
#define SECURE_TRUE  (0x5AA5)
#define SECURE_FALSE (0x55AA)

/* Start Code Watchdog Timer */
CWT_Start(inst_timer_reload, sec_cnt_start);

/* Add flag variable as counter increment irrespective of true or false */;
CWT_Add(critical_flag);

if (critical_flag == SECURE_TRUE) //if this instruction is glitched the CWT_Check() will catch it
{
    CWT_Sub(SECURE_TRUE);
    CWT_Check(sec_cnt_start);

    //Do some critical actions
}
else
{
    CWT_Sub(SECURE_FALSE);
}

CWT_Stop(sec_cnt_start);
```

Figure 8. Secured decision control

2.7 Conclusion

The security counter can be used to protect function calls, loops, if-else constructs, switch-case constructs, pointers, and so on.

Multiple checks can be combined to save code size.

As more updates are made, more security is added (maximum is one Secure Counter update per basic block).

Each Secure Counter update can be seen as a checkpoint that a certain code location was passed. It does not guarantee that the code around has not been altered by an attack. The checking before performing the critical operation is important.

To maximize the security, the reload value of Instruction Timer shall always be as small as possible.

NOTE

The function of this module is closely related to the rational use of the application, considering there is some performance loss.

2.8 Revision history

Table 2. Revision history

Rev	Date	Description
0	July 2020	Initial version

3 Revision history

Table 3. Revision history

Rev.	Date	Description
0	07/2020	Initial release
1	11/2020	Added description for LPC55S0x/LPC550x in Introduction

How To Reach Us

Home Page:

nxp.com

Web Support:

nxp.com/support

For more information, please visit: http://www.nxp.com

For sales office addresses, please send an email to: salesaddresses@nxp.com

Date of release: 11/2020

Document identifier: AN12912

arm