TriCore® Family

AP32171

CRC Computation using PCP

Application Note

V1.0 2010-09

Microcontrollers

Edition 2010-09

Published by
Infineon Technologies AG
81726 Munich, Germany

LEGAL DISCLAIMER

THE INFORMATION GIVEN IN THIS APPLICATION NOTE IS GIVEN AS A HINT FOR THE IMPLEMENTATION OF THE INFINEON TECHNOLOGIES COMPONENT ONLY AND SHALL NOT BE REGARDED AS ANY DESCRIPTION OR WARRANTY OF A CERTAIN FUNCTIONALITY, CONDITION OR QUALITY OF THE INFINEON TECHNOLOGIES COMPONENT. THE RECIPIENT OF THIS APPLICATION NOTE MUST VERIFY ANY FUNCTION DESCRIBED HEREIN IN THE REAL APPLICATION. INFINEON TECHNOLOGIES HEREBY DISCLAIMS ANY AND ALL WARRANTIES AND LIABILITIES OF ANY KIND (INCLUDING WITHOUT LIMITATION WARRANTIES OF NON-INFRINGEMENT OF INTELLECTUAL PROPERTY RIGHTS OF ANY THIRD PARTY) WITH RESPECT TO ANY AND ALL INFORMATION GIVEN IN THIS APPLICATION NOTE.

Information

For further information on technology, delivery terms and conditions and prices, please contact the nearest Infineon Technologies Office (**www.infineon.com**).

Warnings

Due to technical requirements, components may contain dangerous substances. For information on the types in question, please contact the nearest Infineon Technologies Office.

Infineon Technologies components may be used in life-support devices or systems only with the express written approval of Infineon Technologies, if a failure of such components can reasonably be expected to cause the failure of that life-support device or system or to affect the safety or effectiveness of that device or system. Life support devices or systems are intended to be implanted in the human body or to support and/or maintain and sustain and/or protect human life. If they fail, it is reasonable to assume that the health of the user or other persons may be endangered.

Revision History: V1.0, 2010-09

Previous Version: none

Page	Subjects (major changes since last revision)

We Listen to Your Comments

Is there any information in this document that you feel is wrong, unclear or missing?
Your feedback will help us to continuously improve the quality of this document.
Please send your proposal (including a reference to this document) to:

mcdocu.comments@infineon.com

Table of Contents

1 Preface

The importance of a CRC (Cyclic Redundancy Check) in data storage and transmission is well known, and it is also a fundamental feature in safety and high reliability embedded systems.

The TriCore® Audo-NG and Audo-Future family do not actually include a CRC engine, but they do have a MISR (Multiple Input Shift Register) Checksum engine that uses an Ethernet Polynomial.

The MISR algorithm is suitable for protecting a large quantity of data but lacks the following property: for a given polynomial and a given data length, CRC detects all possible combinations of a number of failure bits (= Hamming Distance -1)[1]. For this reason the most common transmission protocols and data protection schemes use the CRC algorithm to protect data.

In Audo-NG and Audo-Future devices the Peripheral Co-Processor (PCP) is present. The PCP performs tasks that would normally be performed by the combination of a DMA controller and it's supporting CPU Interrupt Service Routines in a traditional computer system. It could be considered as the host processor's first line of defence as an interrupt-handling engine. The PCP can unload the CPU from having to service time-critical interrupts. This provides many benefits, including:

- Avoiding large interrupt-driven task context-switching latencies in the host processor.
- Reducing the cost of interrupts in terms of processor register and memory overhead.
- Improving the responsiveness of Interrupt Service Routines to data-capture and data transfer operations.
- Easing the implementation of multitasking operating systems.

Therefore it is quite natural to check or generate the CRC checksum of a data packet inside the PCP.

In this application note an efficient and re-usable PCP implementation of a configurable CRC algorithm is described. The application note is provided together with a *Tasking VX-toolset for TriCore®* and a PCP example for the TC1797 device. This example is implemented in a way that allows for easy configurability for 8, 16 and 32-bit CRC, and it can also be adapted very easily to different applications such as validation of incoming data from a peripheral, or memory to memory data transfer with CRC validation, and so on.

2 CRC Parameters

This document does not discuss the CRC theory and the mathematics behind the chosen implementation, but for background reading regarding this topic please refer to the 6Bibliography references (1) (4) and (5).

The Direct table algorithm implementation was chosen because it is very fast and allows easy configuration for different CRC polynomials. Unfortunately in real-life a CRC algorithm is not only defined by the polynomial as in the mathematical theory, but other parameters are required, such as the Initialization value, reflection, and so on. As there is no standard for the full definition of a CRC computation, this document is based on the "Rocksoft™ Model CRC Algorithm" parametrized model (5).

Table 1 lists the CRC model parameters together with the name used in the example source code.

Table 1 CRC Algorithm Parameters

Parameter	Description
CRC_WIDTH	CRC bits width
POLYNOMIAL	Value of CRC polynomial
INIT_VALUE	Initilization value
XOR_OUT	Value to be xored to crc checksum before to output it.
REFLECT_IN	If true the input bytes will be reflected.
REFLECT_OUT	If true CRC chekcsum is reflected before to ouput it.

[1] Refer to the bibliography chapter for a paper reporting the Hamming distance for most common CRC polynomial (4).

Note: The REFLECT_IN parameter enables an efficient CRC computation when the input bytes are reversed. Some serial communication protocols require that each byte be transmitted with the least significant bit (LSB) first and the most significant bit last. Instead of reflecting (switching from LSB first to LSB last ordering) all input bytes, by 'reflecting' the algorithm it is possible to compute the correct CRC without the overhead of byte reflection.

3 PCP Implementation

The Direct Table Algorithm has been chosen as the C implementation because it is a good compromise between speed, memory footprint and configurability. A lookup table of 256 entries was selected because of the small memory size available.

The Tasking compiler provides a PCP C compiler. The C implementation of the Direct Table algorithm is very efficient and it can be compiled in fast PCP binary code as each part of code is adapted for PCP.

The PCP assembly instruction set does have the following limitations that affect the C compiler (3):

- PRAM addressing is 32bits → PRAM can be written from FPI bus only by double-word.
- No load/store byte/word instruction from/to PRAM → all data stored in PRAM has 32-bit size.

Therefore it is not possible to just copy the C code and compile it unless some adaptations have been made.

The code described in the next section is developed in order to allow full configurability for any CRC8/16/32 polynomial and to maximize the speed.

3.1 Example Description

The example performs the following operations:

1. TriCore® project initializes two different PCP channels that implement two different CRCs.
2. The two PCP channels are activated sequentially and calculate two different CRCs of a block of 1024 bytes stored in flash.
3. The CRC and throughput is printed out.

The example is divided into three sub-projects: pcp-crc-start, pcp-crc-ch1 and pcp-crc-ch2.

3.1.1 Project: pcp-crc-start

This TriCore® project initializes the PCP channels, triggers their execution and verifies the result of the CRC computation.

The project contains the following files:

- **main.c**
  - computes the tables required by the Direct Table CRC implementation, triggers the PCP channels, checks results and prints performance results.
- **crc_init.c and crc_init.h**
  - contains the function required to generate the table required by the CRC Direct Table algorithm.
- **cstart.c and cstart.h**
  - these are Tasking toolset generated files required to initialize the C environment.
- **simio.c**
  - required by the pls debugger in order to enable simulated I/O.

3.1.2 Project: pcp-crc-ch1 and pcp-crc-ch2

These two projects compute two different CRCs of data pointed to by the `buffer_pointer`, using different CRC parameters. The two projects each contain two files:

- **pcp-crc1.c and pcp-crc2.c**
  - identical files except for the header files, which are used to configure the code. Input and output variables are shared PRAM variables.
- **crc_parameters_ch1.h and crc_parameters_ch2.h**

– two header files that store the configuration of the two CRC algorithms. They are used by the pcp-crc-start project to initialize the CRC lookup tables used in the calculation.

3.2 Implementation Details

The PRAM variables used by the PCP channels are declared as follows:

```
//Pointer to the message to be translated in FPI space.
unsigned char __far __mau8 * volatile buffer_pointer;
volatile unsigned long num_bytes;   //Num of bytes to be transferred
volatile unsigned long init_value;  //Initial value of CRC
volatile unsigned long xor_out;     //Value to be xored before returning the CRC
volatile unsigned long crc_out;     //CRC result
```

Attention: The pointer to the buffer located on the FPI bus is declared using __far __mau8 in order to have a data size of 8 bit (see (2)).

Attention: You can use the type qualifier __mau8 only on objects that have the __far qualifier, because only objects located in the FPI space can have byte access. PRAM has only double word access (see (3)).

For TriCore®, the variable counterparts of the PCP channel global variables are as follows:

```
/*
 * These are the external declarations required by Tricore compare in order to
 * access the PCP channel global variables.
 */
#if __CTC__
#define ADD_GLOBAL_SYMBOL_PREFIX(A) CH1_##A

extern volatile unsigned long __pram ADD_GLOBAL_SYMBOL_PREFIX(buffer_pointer);
extern volatile unsigned long __pram ADD_GLOBAL_SYMBOL_PREFIX(num_bytes);
extern volatile unsigned long __pram ADD_GLOBAL_SYMBOL_PREFIX(init_value);
extern volatile unsigned long __pram ADD_GLOBAL_SYMBOL_PREFIX(xor_out);
extern volatile unsigned long __pram ADD_GLOBAL_SYMBOL_PREFIX(crc_out);
extern unsigned long __pram ADD_GLOBAL_SYMBOL_PREFIX(crc_table)[256];
#endif //__CTC__
```

The **__pram** keyword tells the TriCore® compiler to use symbol names prefixed by `_lc_s_` (2); i.e. it declares the variables as shared global variables of PCP.

The prefix of the shared symbols (in this example CH1) is set by an option in the PCP projects (see **Figure 1**). This enables only one source code for all CRC algorithms because the specific labelling is performed automatically by the PCP C compiler.

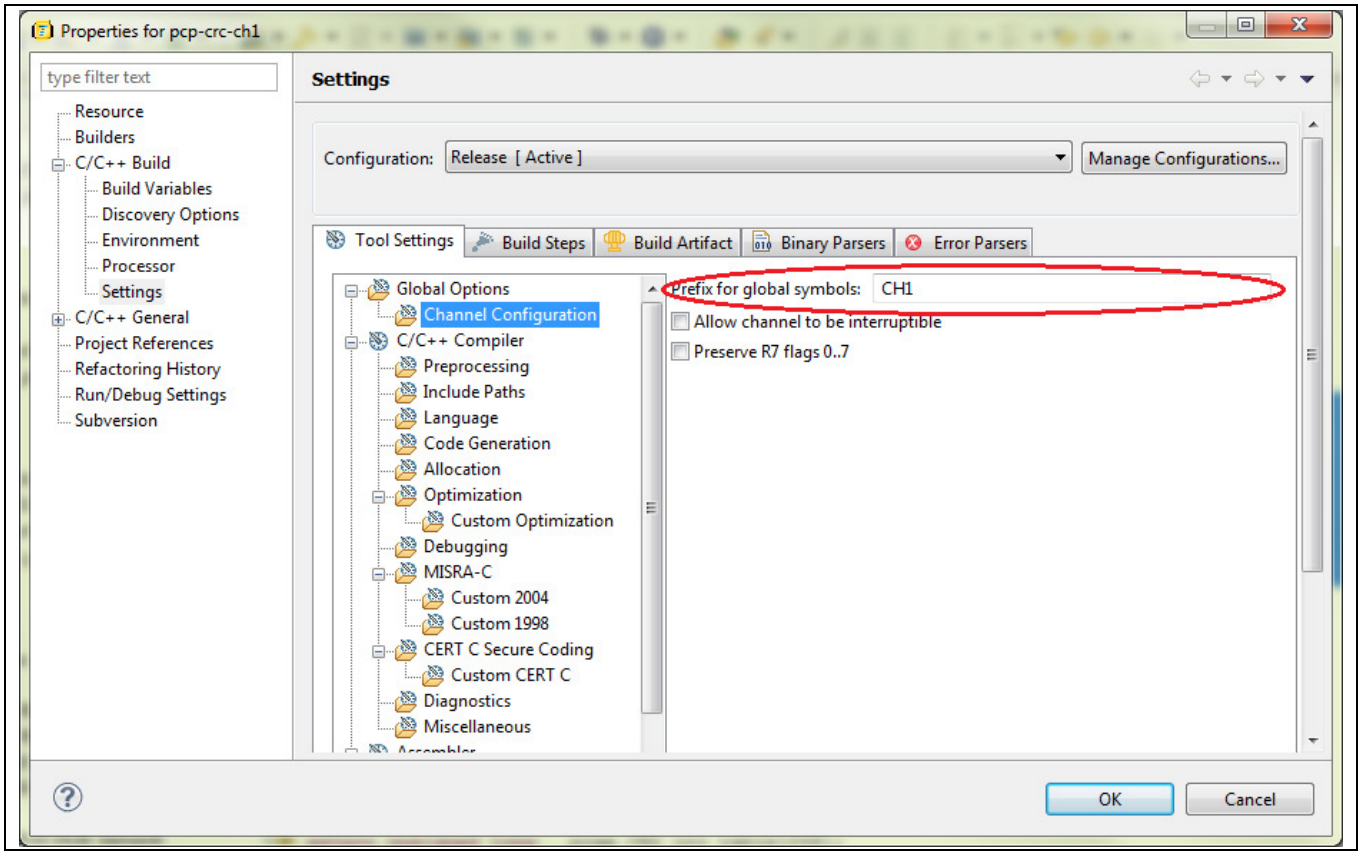

Figure 1 How to Set Prefix for Global Symbol

The PCP channel variables are initialized using macros stored in the channel specific header file **crc_parameters_chx.h**.

The following shows how initialization is performed for channel 1.

```
//Initialize crc parameters of ch1

crc_table_init(CH1_crc_table,REFLECT_IN_CH1,CRC_WIDTH_CH1,POLYNOMIAL_CH1,CRC_TABLE
_ALWAYS_32BITS_CH1);

CH1_buffer_pointer=(unsigned int)test_pattern;

CH1_num_bytes=BUFFER_SIZE;

CH1_init_value=INIT_VALUE_CH1;

CH1_xor_out=XOR_OUT_CH1;
```

Note that the function `crc_table_init` is used to set the CRC lookup table, so it is possible to change some CRC algorithm parameters at runtime.

Attention: REFLECT_IN and CRC_WIDTH cannot be changed at runtime because those are used to configure the code for efficiency reasons. In fact if these are used as runtime parameters, additional conditional instructions are required inside the loop that computes the CRC, and so performance is impacted.

The PCP limitation described at the beginning of this chapter has an impact on the performance of CRC8 and CRC16. It is only possible to address the CRC lookup table as a double-word because it is located in the PRAM. One solution is to place this array in DPRAM or PMU and access it through the FPI, but this is not desirable when the FPI bus is intensively used.

The solution to maximize speed is to use the double-word data type for 8 and 16 bit CRC. This will cost 4 or 2 times the table size, but the performance will be not impacted.

Where there are tight PRAM constraints it is possible to use the regular size CRC lookup table, but some extra code is required to set and read table values. This has a big impact on performance. For example, an 8-bit CRC is ~2 times slower than a 32-bit CRC computation. By using the `CRC_TABLE_ALWAYS_32BITS_CH2` macro it is possible to switch between the two implementations.

When `CRC_TABLE_ALWAYS_32BITS_CH2` is set to 0, the lookup table has the expected size (crc_size_in_bytes x 256) but extra code is required to access it, to get the correct lookup table value.

When `CRC_TABLE_ALWAYS_32BITS_CH2` is set to 1, the lookup table is always 4 x 256 bytes, regardless of the CRC size and only a small portion of table entries are used to store the values needed for the computation. So retrieving values is very fast at a cost of PRAM size.

Below is the macro that retrieves the lookup table values:

```
#if CRC_TABLE_ALWAYS_32BITS
#define GET_TABLE_ITEM(A) crc_table[A]
#else

#define GET_TABLE_ITEM(A) get_table_item(A)
/*
 * Return crc table value at position defined by index.
 */
static inline unsigned long get_table_item(unsigned long index) {
    unsigned long index_by_32bit,bits_offset,value;
    index_by_32bit=index/(4/(CRC_WIDTH/8));
    value=crc_table[index_by_32bit];
    bits_offset=(index*CRC_WIDTH) % 32;
    value=(value >> bits_offset) & CRC_MASK;
    return value;
}
#endif //CRC_TABLE_ALWAYS_32BITS
```

3.3 Memory Footprint and Performance

The memory footprint depends on the CRC configuration:

- Code footprint ranges from 216 to 78 bytes (max speed optimizations).
- Data footprint ranges from 308 to 1024 bytes.

Minimum measured throughput is ~1.5 MB/sec (CRC8) while maximum measured throughput is ~3.5 MB/sec (CRC32).

When `CRC_TABLE_ALWAYS_32BITS_CH2` is set to 0 all configurations give almost the same throughput; ~3.5 MB/sec.

3.4 How to Create a New PCP CRC Channel

The following steps describe how to use the PCP library and create a new CRC PCP channel.

1. Create a new PCP project (e.g. by copying an existing one and renaming pcp-crc-ch3).
2. Copy **pcp-crc1.c** and **crc_parameters_ch1.h**.
3. Change the file names to **pcp-crc3.c** and **crc_parameters_ch3.h**
4. Update the included header filename in **pcp-crc3.c**
5. Change all _CH1 to _CH3.

6. Update the CRC parameters as required.
7. Update in the PCP project the prefix of global symbols to CH3 (see **Figure 1**).
8. Switch to the TriCore® project and include the **crc_parameters_ch3**.
9. Add code to initialize the PCP channel as shown in **main.c**, with the precaution of changing the prefix CH1 to CH3.
10. Activate the channel.

Note: The PCP CRC channel does not overwrite shared global variables (except `crc_out`*), so it is not necessary to re-initialize the variables before every new channel execution.*

4 Conclusion

This application note presents a configurable library code to compute CRC inside the PCP.

This very efficient implementation can be configured for any 8, 16 or 32-bit CRC using the parameters defined by the "Rocksoft™ Model CRC Algorithm".

This implementation can be very easily adapted for different applications as a peripheral to memory data transfer and CRC verification, memory to peripheral and CRC computation, and memory to memory safe transfer.

5 Appendix A: Project Tool Requirements

The following tools are required to compile and execute the example:

Software

- Tasking VX-toolset for TriCore® and PCP version 3.4r1 (**http://www.tasking.com/)**
- PLS UDE Desktop (**http://www.pls-mc.com**/)

Hardware

- Triboard TC1797 v4.1
- PLS Universal Access Device 2

6 Bibliography

1. Cyclic Redundancy Check (CRC). *Wikipedia.* [Online] http://en.wikipedia.org/wiki/Cyclic_redundancy_check.
2. TASKING VX-toolset for PCP User Guide. s.l. : Altium.
3. **Infineon AG.** TC1797 User's Manual (TC1797_UM_v1.1.pdf). Infineon Web Site. [http://www.infineon.com]
4. 32-Bit Cyclic Redundancy Codes for Internet Applications. **Koopman, Philip.** 2002. The International Conference on Dependable Systems and Networks (DSN) 2002. 10.1109/DSN.2002.1028931 .
5. **Williams, Dr Ross.** CRC: A Paper On CRCs. [Online] http://www.ross.net/crc/crcpaper.html

w w w . i n f i n e o n . c o m

Published by Infineon Technologies AG