SPC58x Ethernet FIFO access and IMA error injection

Introduction

This document aims at showing an example of how to program the embedded Ethernet controller available on the spc58x automotive microcontroller (MCU) to access its own internal FIFO by using the slave debug port and, in specific, of how to inject an error through the indirect memory access (IMA) unit checking the error from the memory error management unit (MEMU).

The example code reported in this document has been tested on the SPC584Cx/SPC58ECx MCU.

1 Ethernet controller overview

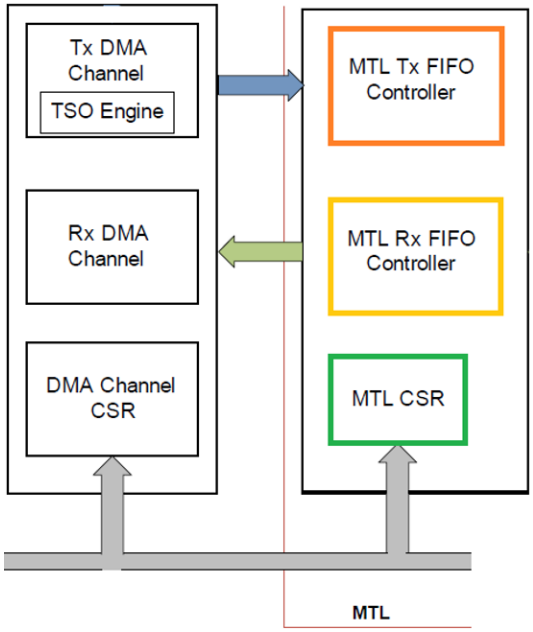
SPC58x automotive microcontrollers feature a quality-of-service 10/100 Mbit/s Ethernet peripheral that implements the medium access control (MAC) layer plus several internal modules for standard and advanced network features that can be summarized into the following categories:

- MAC core and MAC transaction layer (MTL)
- DMA engine
- SMA interface
- MAC management counters
- Power management block (PMT)

1.1 MAC transaction layer

The MAC transaction layer (MTL) provides the FIFO memory interface to buffer and regulate the packets between the application system memory and the MAC. The MTL block consists of Tx FIFO and Rx FIFO managed by the block and not exposed to the application software. As a result, a software driver cannot access them directly unless for implementing a dedicated procedure as detailed in the next chapter.

Figure 1. Controller block diagram for MTL and FIFO



For further details about the Ethernet controller, for instance FIFOs size, programming and configuration, refer to the device reference manual.

1.2 Slave mode access

The Ethernet module supports the debug access to the FIFO memory, and the Tx and Rx FIFO queues memory is accessible through the internal slave port as described in the reference manual.

By enabling this mode the user can program the entire selected Tx memory to Tx FIFO queue 0 and the Rx memory to Rx FIFO queue 0.

The FIFO will work in store-and-forward mode, for example the MTL pops out the packet towards the MAC only when the complete packet is stored in the queue.

The offload features such as checksum offload engine and IEEE timestamp should not be enabled with this feature.

The following sections will describe how to write and read data through a slave interface on a FIFO.

1.2.1 Writing to the Tx FIFO

The following code configures and writes a pattern in the Tx FIFO in slave debug mode.

```
/* Configuration */
reg->MTL_DBG_CTL.B.DBGMOD = 1;
reg->MTL_DBG_CTL.B.FDBGEN = 1;
reg->MTL_DBG_CTL.B.RSTALL = 1;
reg->MTL_DBG_CTL.B.BYTEEN = 3;
reg->MTL_DBG_CTL.B.PKTSTATE = 0;

/* Write a pattern (fill 4KiB FIFO) */
for (i=0; i < FIFO_ELEMENT_N; i++) {
       limit = 10;
       /* Complete data word in 64-bit mode. */
       for (j = 0; j < 2; j++) {
              /* Optionally it could be useful to look at the LOCR bit
                 for monitoring the FIFO */

              /* To better monitor the FIFO elements, store a pattern */
              reg->MTL_FIFO_DEBUG_DATA.B.FDBGDATA = FIFO_PATTERN+count;
              count++;

              /* Enable the Writing access to the TX FIFO */
              reg->MTL_DBG_CTL.B.FIFOSEL = 0; // 0 TX FIFO, 3 RX_FIFO
              reg->MTL_DBG_CTL.B.FIFORDEN = 0;
              reg->MTL_DBG_CTL.B.FIFOWREN = 1;

              /* Wait for FIFOBUSY in the status register */
              while ((reg->MTL_DBG_STS.B.FIFOBUSY == 1) && limit)
                    limit--;
        }
}
```

1.2.2 Reading from a FIFO

The following code reads from the Rx FIFO and saves the content in the user buffer User_FIF0_data.

```
/* READING FROM FIFO IN DEBUG MODE */
reg->MTL_DBG_CTL.B.DBGMOD = 1;
reg->MTL_DBG_CTL.B.FDBGEN = 1;

for (i=0; i < FIFO_ELEMENT_N; i++) {
       limit = 10;
       for (j = 0; j < 2; j++) {                // 64bit
              reg->MTL_DBG_CTL.B.FIFOSEL = 0; // 0  TX FIFO, 3 RX_FIFO
              reg->MTL_DBG_CTL.B.FIFORDEN = 1;
              reg->MTL_DBG_CTL.B.FIFOWREN = 0;

              while ((reg->MTL_DBG_STS.B.FIFOBUSY == 1) && limit)
                    limit--;

              /* Store the FIFO elements in a user array */
              if (limit)
                    User_FIF0_data[count] =
                          reg->MTL_FIFO_DEBUG_DATA.B.FDBGDATA;
              count++;
       }
}
```

2 IMA overview

Indirect memory access (IMA) refers to the activity of accessing any volatile memory for the purpose of reading or modifying data, or both, including ECC check bits.

This capability is mandatory to verify the ECC functionality.

The SPC584Cx/SPC58ECx MCU provides two mechanisms for IMA:

- The built-in CPU E2E_ECC test capability provides a path for accessing data and ECC check bits from some SRAM arrays.
- An IMA controller module provides registers for selecting, reading and writing memory array including both data and relevant ECC bits.

Despite other peripherals where the application can access the shared memory, in case of the Ethernet controller, where the FIFO cannot be accessed by the software, a dedicated function has been developed to verify this feature.

2.1 Read and write functions

This chapter describes how to perform accesses through the IMA.

2.1.1 IMA read access

In order to perform an IMA read access the user must follow the sequence below:

- Unlock IMA read access
- Based on the memory address to be accessed, write the IMA_SLCT.ARRAY_SLCT and IMA_SLCT.ROW_SLCT fields
- Set the IMA_CTRL.READ bit
- Set IMA_ENABLE.EN bit to initiate read access. This bit will be self cleared after two clocks.
- Wait an appropriate number of clocks depending on the speed of the selected memory
- Read data from IMA_READ_DATA registers

The following code implements the operations described above.

```
static void IMA_Read(unsigned int slct, unsigned int row)
{
    IMA.READ_UNLOCK.B.REA_KEY = 0xF06AB5BC;
    IMA.READ_UNLOCK.B.READ_KEY = 0x14081B56;
    if (IMA.STATUS.B.READ_LOCK == 0) {
        IMA.SLCT.B.ARRAY_SLCT = slct;
        IMA.SLCT.B.ROW_SLCT = row;
        IMA.CTRL.B.READ = 1;
        IMA.ENABLE.B.EN = 1;
        while (IMA.ENABLE.B.EN);
        IMA_Read_Data_Buffer[0] = IMA.READ_DATA_0.R;
        IMA_Read_Data_Buffer[1] = IMA.READ_DATA_1.R;
        IMA_Read_Data_Buffer[2] = IMA.READ_DATA_2.R;
        IMA_Read_Data_Buffer[3] = IMA.READ_DATA_3.R;
        IMA_Read_Data_Buffer[4] = IMA.READ_DATA_4.R;
    }
    IMA.SLCT.B.ARRAY_SLCT = 0;
    IMA.SLCT.B.ROW_SLCT = 0;
    IMA.READ_UNLOCK.B.READ_KEY = 0x00000000;
}
```

2.1.2 IMA write access

In order to perform an IMA write access the user must follow the sequence below:

- Unlock IMA write access
- Based on the memory address to be accessed, write the IMA_SLCT.ARRAY_SLCT and IMA_SLCT.ROW_SLCT fields
- Write the data in IMA_WRITE_DATA registers
- Clear the IMA_CTRL.READ bit
- Set IMA_ENABLE.EN bit to initiate write access. This bit will be self cleared after two clocks.
- Wait an appropriate number of clocks depending on the speed of the selected memory
- Access by a master (e.g. CPU) to the selected memory location to detect the ECC error
- Check the relevant MEMU register to verify the ECC error is reported

The following code implements the operations described above and it will be invoked by IMA_Test function to inject an error.

```
static void IMA_Write_Err(unsigned int slct, unsigned int row,
                                  unsigned int mask_0, unsigned int mask_1,
                                  unsigned int mask_2)
          {
                IMA.WRITE_UNLOCK.B.WRITE_KEY = 0x04A43F95;
                IMA.WRITE_UNLOCK.B.WRITE_KEY = 0xE4A9EBF7;

                if (IMA.STATUS.B.WRITE_LOCK == 0) {
                    IMA.SLCT.B.ARRAY_SLCT = slct;
                    IMA.SLCT.B.ROW_SLCT = row;
                    IMA_Write_Data_Buffer[0] = IMA_Read_Data_Buffer[0]^mask_0;
                    IMA_Write_Data_Buffer[1] = IMA_Read_Data_Buffer[1]^mask_1;
                    IMA_Write_Data_Buffer[2] = IMA_Read_Data_Buffer[2]^mask_2;
                    IMA_Write_Data_Buffer[3] = IMA_Read_Data_Buffer[3];
                    IMA_Write_Data_Buffer[4] = IMA_Read_Data_Buffer[4];
                    IMA.WRITE_DATA_0.R = IMA_Write_Data_Buffer[0];
                    IMA.WRITE_DATA_1.R = IMA_Write_Data_Buffer[1];
                    IMA.WRITE_DATA_2.R = IMA_Write_Data_Buffer[2];
                    IMA.WRITE_DATA_3.R = IMA_Write_Data_Buffer[3];
                    IMA.WRITE_DATA_4.R = IMA_Write_Data_Buffer[4];
                    IMA.CTRL.B.READ = 0;
                    IMA.ENABLE.B.EN = 1;
                    while (IMA.ENABLE.B.EN) {};
                }
                IMA.SLCT.B.ARRAY_SLCT = 0;
                IMA.SLCT.B.ROW_SLCT = 0;
                IMA.WRITE_UNLOCK.B.WRITE_KEY = 0x00000000;
}
```

2.2 Error injection

This section reports the defined commands and the IMA_Test function invoked in the application.

The FIFO has been filled with a pattern sequence and the IMA_Test function reads the data from the memory region that maps the internal FIFO and then injects a single-bit error in the 62nd position of the 2nd data word. Note that IMA performs a 64-bit access.

```
static void IMA_test(void)
{
    IMA_Read(IMA_ARRAY_ETH_TX_E, 0);
    IMA_Write_Err(IMA_ARRAY_ETH_TX_E, 0, 0x00000000, 0x40000000, 0x00000000);
    IMA_Read(IMA_ARRAY_ETH_TX_O, 0);
    IMA_Write_Err(IMA_ARRAY_ETH_TX_O, 0, 0x00000000, 0x40000000, 0x00000000);
    IMA_Read(IMA_ARRAY_ETH_TX_E, 1);
    IMA_Read(IMA_ARRAY_ETH_TX_O, 1);
    IMA_Read(IMA_ARRAY_ETH_TX_E, 2);
    /*
     * Further elements could be read, and other errors introduced...
     * IMA_Write_Err(IMA_ARRAY_ETH_TX_E, 2, 0x00000000,
     * 0x40000000, 0x00000000);
     */
}
```

Where:

```
#define IMA_ARRAY_ELEMENT 5
unsigned int IMA_Read_Data_Buffer[IMA_ARRAY_ELEMENT];
unsigned int IMA_Write_Data_Buffer[IMA_ARRAY_ELEMENT];
unsigned int Tx_Fifo_data[128];
#define FIFO_PATTERN 0x12345670
#define FIFO_ELEMENT_MAX 64 /* 4KiB*/
#define FIFO_ELEMENT_N 8
#define IMA_ARRAY_ETH_TX_O 29
#define IMA_ARRAY_ETH_TX_E 31
```

The table below reports the ethernet SRAM arrays accessible via IMA, from the device reference manual:

Table 1. Ethernet IMA array select

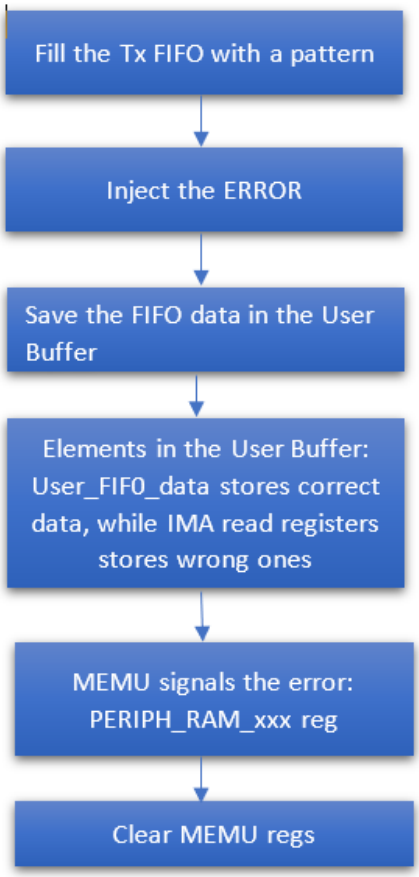
IMA array select	Block	Function	SPC584Cx/SPC58ECx		
			Words	Bits	ECC bits
28	Ethernet	Ethernet ODD RX	512	76	75-64
29		Ethernet ODD TX	256	76	75-64
30		Ethernet EVEN RX	512	76	75-64
31		Ethernet EVEN TX	256	76	75-64

3 Application

After the Ethernet is successfully initialized, the IMA test can be started. The following diagram shows the flow of the functions explained in the previous chapters.

The Tx FIFO is filled with a known pattern, then the error is injected and the results are visible in the user buffer and in the IMA read registers.

Figure 2. IMA test flow diagram



In case of a single-bit error, i.e. correctable error, in a certain memory location, the error is detected and the data is corrected by the ECC logic before it is provided to the requesting master. As a result, the memory location still holds the corrupted data, as it can be verified by performing a new read operation by the IMA on the same memory location.

Note that a new read access by a master, e.g. the CPU, on the same memory location generates a new single-bit error detection (and the correction of the data provided to the requesting master) with the reporting to the MEMU but in this case the MEMU will drop the error because the address and the syndrome are the same as the error already recorded in the MEMU reporting table. On the contrary, if the valid bit PERIPH_RAM_CERR_STS0.VLD was previously cleared, the error is recorded again.

The following code checks inside the remaining FIFO content.

```
IMA_Read(IMA_ARRAY_ETH_TX_E, 0);
IMA_Read(IMA_ARRAY_ETH_TX_O, 0);
IMA_Read(IMA_ARRAY_ETH_TX_E, 1);
IMA_Read(IMA_ARRAY_ETH_TX_O, 1);
IMA_Read(IMA_ARRAY_ETH_TX_E, 2);
```

4 Memory error management unit (MEMU)

The MEMU is responsible for collecting and reporting error events associated with the error correction code (ECC) logic which is also used on peripheral memories.

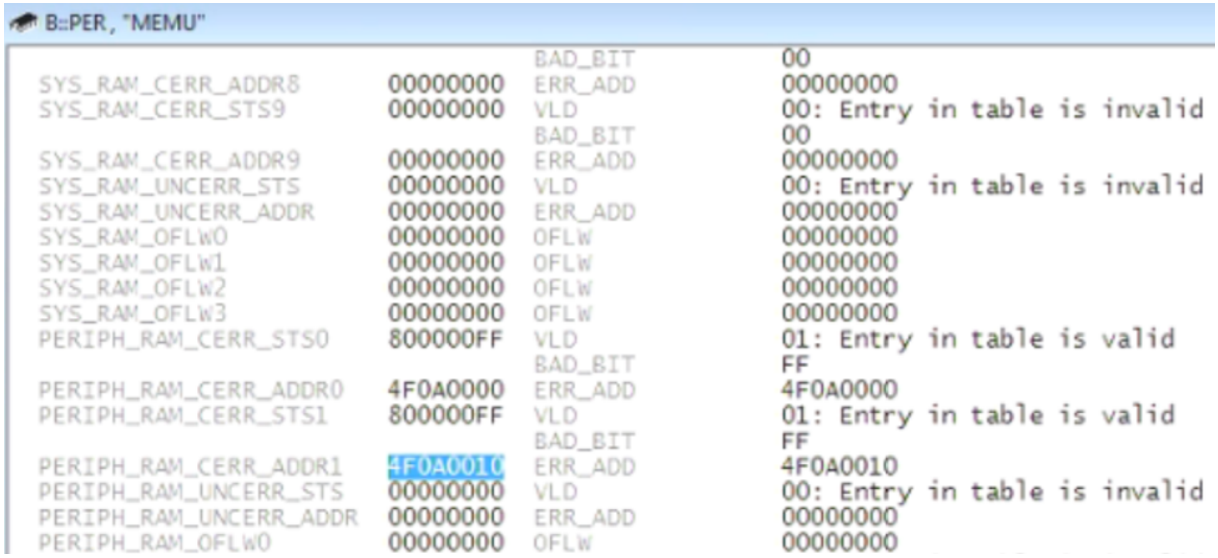
The MEMU receives an error signal which causes an event to be recorded and the corresponding error flags to be set and reported to the fault collection and control unit (FCCU).

In the example, inserting an error in the 2nd word, the MEMU will report the error after the reading of the first word and the address is the address of the double word.

This behavior is shown in Figure 3 below: error inserted in the 2nd word of the 3rd row of Tx FIFO EVEN (0x**1**2345679 changed in 0x**5**2345679).

After reading the 1st word (data = 0x12345678), MEMU reports the error with address (0x4F0A0000+0x10 where 0x10 is the offset of the 3rd double word).

Figure 3. MEMU status



The code below checks the Address fields:

```
if ((MEMU.PERIPH_RAM[0].CERR_STS.B.BAD_BIT == 0xFF)
          &  (MEMU.PERIPH_RAM[1].CERR_STS.B.BAD_BIT == 0xFF))
          ret = pPASS;

          /* Clear MEMU regs */
          MEMU.PERIPH_RAM[0].CERR_STS.R = 0;
          MEMU.PERIPH_RAM[1].CERR_STS.R = 0;
```

Appendix A Other information

A.1 Acronyms and abbreviations

Table 2. **Acronyms and abbreviation table**

Terms	Meaning
DMA	Direct memory access
FIFO	First In – First Out
IMA	Indirect memory address
MAC	Medium access control
MTL	MAC transaction layer
MEMU	Memory error management unit

A.2 Reference documents

- SPC58EHx/SPC58NHx 32-bit Power architecture microcontroller reference manual

Revision history

Table 3. **Document revision history**

Date	Version	Changes
02-Sep-2020	1	Initial release.

Contents

List of tables

List of figures