DMA linked list modes

32-bit AURIX™ microcontrollers

About this document

Scope and purpose

The Direct Memory Access (DMA) module in the Infineon AURIX™ microcontroller family supports linked list modes. Compared to setting up a normal DMA transaction a DMA linked list can be slightly more complex to program and so this document aims to provide some guidance. We provide an overview of the DMA linked list modes in the TC3xx AURIX™ microcontrollers and provide some programming examples.

Intended audience

This document is intended for software programmers who are already familiar with programming the DMA module in the AURIX™ microcontroller, and wish to make use of the DMA linked list mode.

***Note**: The example code follows Tasking compiler syntax where relevant.*

Table of contents

1 DMA transactions basics

The DMA is used to perform data transfers without CPU intervention. For example the DMA can transfer data from Memory to Memory, or Memory to Peripheral, or vice-versa. In the DMA there are multiple channels (128 channels in TC39x for example) which can be independently programmed to perform different data transfers. Such data transfers can be broken down into a sequence of reads and writes on the buses (SRI, SPB) in the AURIX™ microcontroller.

One such read and write operation is known as a DMA Move. A set of such moves constitute a DMA Transfer.

The DMA in the AURIX™ microcontroller is programmed to perform DMA Transactions. Each DMA Transaction consists of a set of DMA transfers, which in turn consists of a certain number of DMA moves. Figure 1 explains this relationship between DMA Moves, Transfers and Transactions.

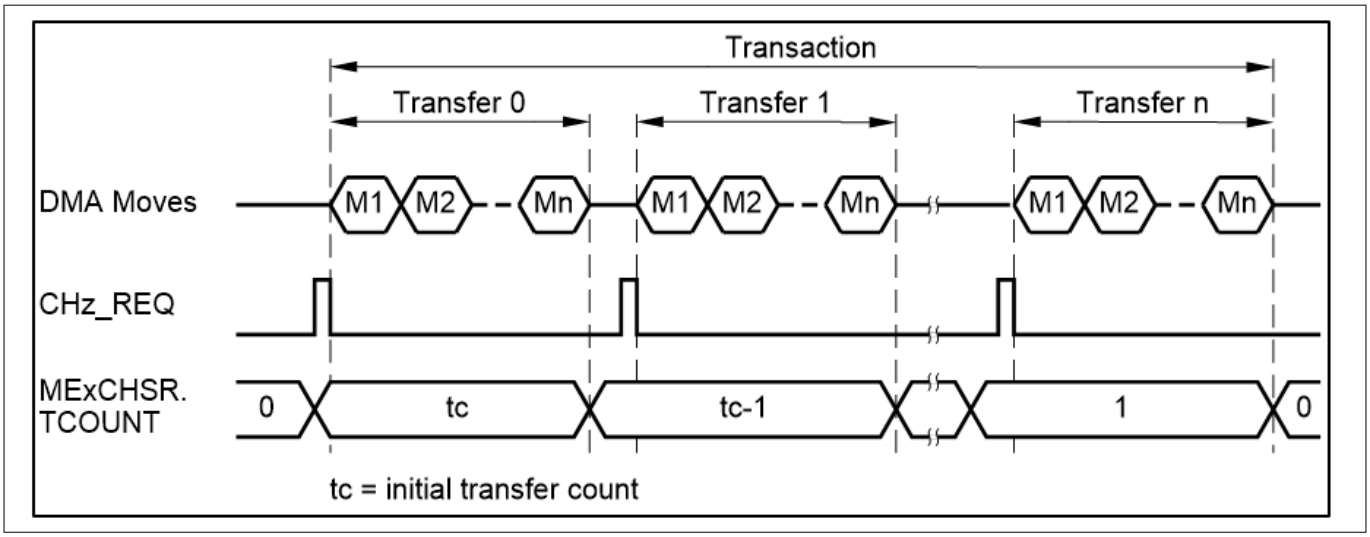

Figure 1 DMA Moves, Transfers and Transactions

Depending on the configuration of the RROAT field of the CHCFGRz register, DMA transfers may be requested by a software or a hardware trigger.

DMA hardware requests are triggered by specific request lines from the Interrupt Router (IR) or from other DMA channels. Typically the parallel occurrence of DMA requests and interrupts requests for DMA channels is possible.

***Note**: The DMA arbitration happens after each transfer, rather than each complete transaction. The arbitration sequence results in the highest number DMA channel with a pending request executing a transfer.*

The DMA controller primarily consists of DMA channels, sub-blocks (Move Engines) and a bus switch. Once configured, the DMA sub-blocks are able to act as a master on the SPB Bus and on the SRI Bus, and execute the DMA Transactions programmed in various DMA channels.

1.1 Transaction Control Set (TCS)

In order to program a DMA transaction in a DMA channel, a set of 8 x 32-bit registers have to be configured for the channel. This set of 8 registers is known as the Transaction Control Set (TCS). One such set of registers exist for each implemented DMA channel in the device.

Figure 2 shows the 8 registers in a channel's transaction control set.

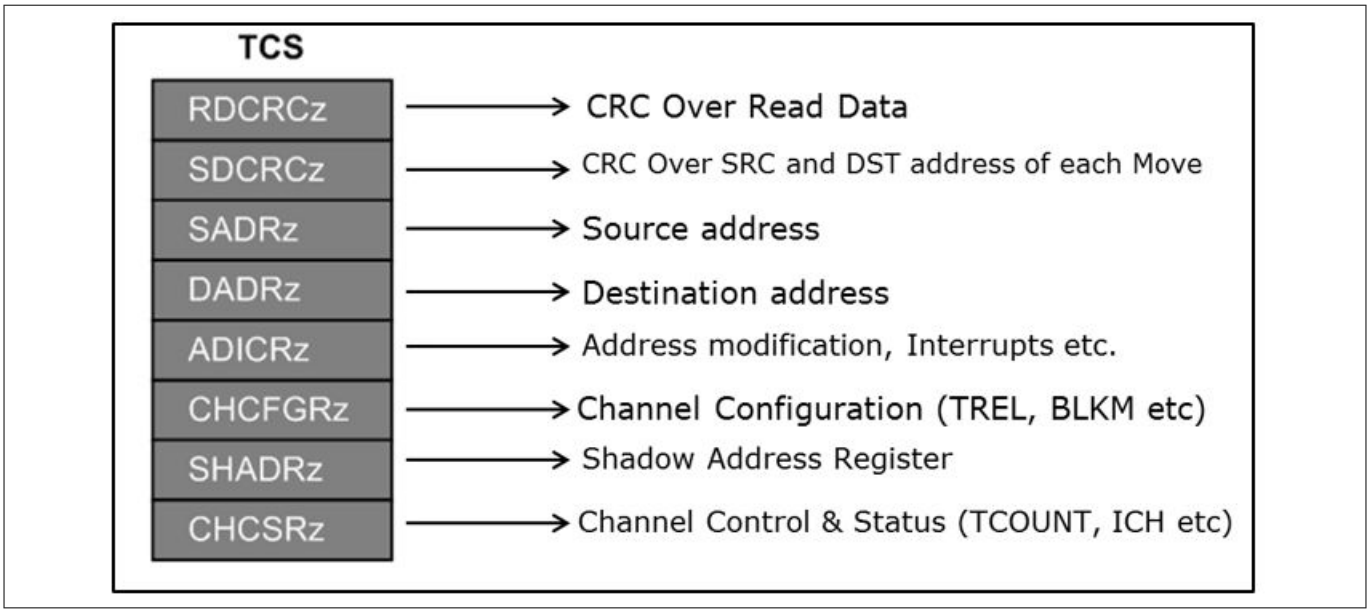

Figure 2 DMA Transaction Control Set

If you do not know how to program a basic DMA transaction, please refer to the user manual and iLLD examples. In this document we will focus only on how to program the various DMA linked list modes.

2 DMA linked list modes

Linked lists are an extension of the DMA channel functionality which executes a single transaction. Linked lists consist of a series of DMA transactions executed by the same DMA channel z. Each DMA transaction has a unique Transaction Control Set (TCS). The source and destination areas do not have to exist in contiguous areas of memory. The new DMA transaction can issue a DMA software request and auto-start the DMA transaction bypassing the Interrupt Router and so reducing the cumulative latency over a number of DMA transactions.

Linked list advantages

The advantages of using a linked list to perform a series of different DMA transactions are:

- There is no reprogramming required at the end of each DMA transaction. If the DMA transactions are executed independently, the software would have to reprogram the channel at the end of each one, for the next. This reduces overall CPU load and may improve the performance of the application, as well as reduce programming complexity.
- Number of DMA channels used is reduced
  - When there are a series of DMA transactions to be executed (Memory- Memory transactions for example), they could be programmed to different DMA channels, but this increases the complexity. Adding them as a linked list makes it much simpler.
- Autostart may reduce the cumulative latency
  - In linked list mode it is possible to automatically start a DMA transaction. This reduces the cumulative latency as explicit software triggers or interrupts through the IR are not required.

2.1 Normal linked list

The DMA linked list mode is chosen by setting ADICRz.SHCT = 0xC. Figure 3 shows the principle of a DMA linked list.

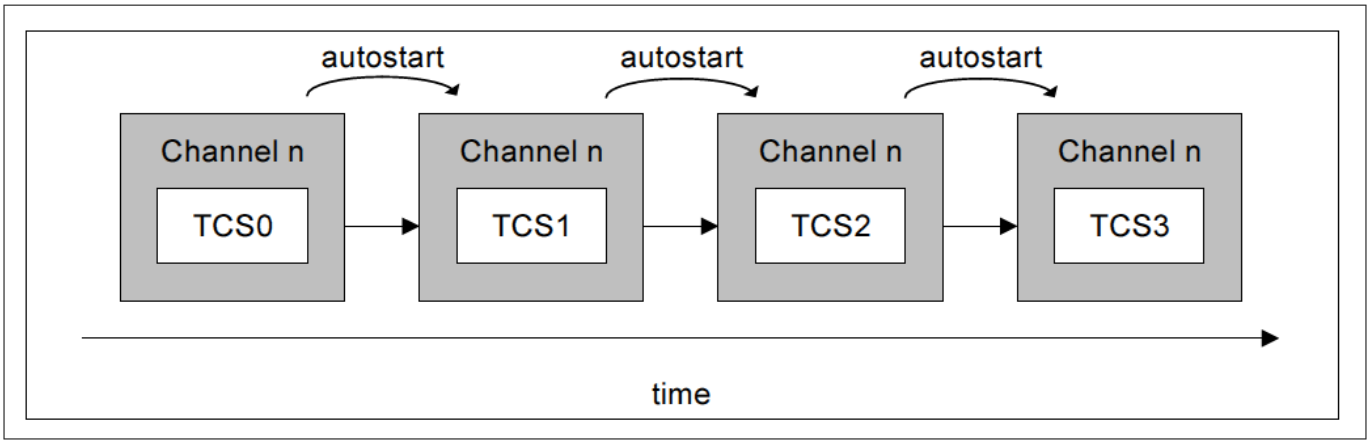

Figure 3 DMA linked list operation

Here a DMA channel (n) is initially programmed for a DMA transaction. At the end of the DMA transaction the DMA *automatically* loads a new TCS from external memory (such as DSPR, LMU or PFLASH) and overwrites the channel TCS, and can start to execute it. At the end of this second DMA transaction, a third TCS is newly loaded and executed and so on.

The DMA transactions in the linked list can be stored in any memory on the SRI bus, such as DSPR, PSPR, PFLASH, or LMU for example.

How this works is that the current DMA transaction uses the shadow address register (SHADR) as an address pointer to the next TCS. The address pointer to the next TCS must be mapped to a 32-byte aligned address.

That is, all DMA linked list TCSs stored in the memory shall be aligned to a 32-byte word. This is automatically guaranteed if they are in a contiguous memory area, and the first transaction is aligned to 32-bytes (since the size of a transaction control set is also 32-bytes).

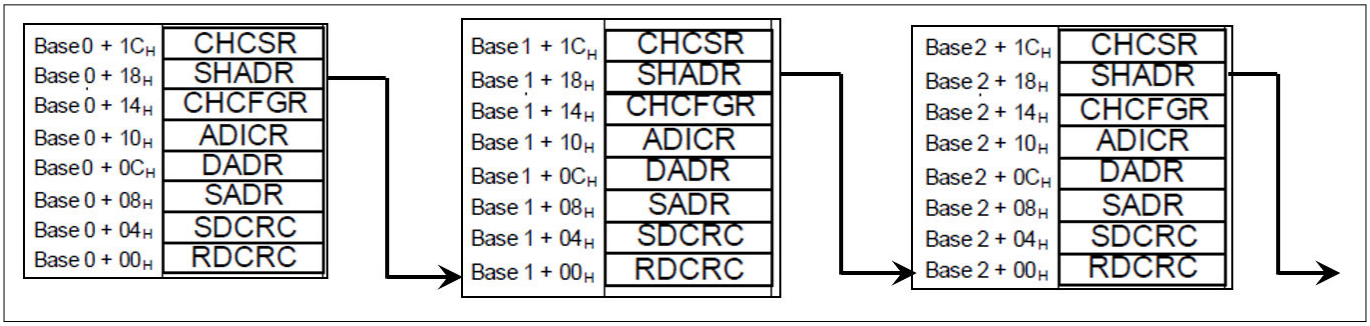

Figure 4 DMA linked list operation

2.2 Accumulated linked list

The DMA accumulated linked list mode is chosen by setting ADICRz.SHCT = 0xD. It is exactly the same as a normal DMA linked list, except that the SDCRC and RDCRC checksums are accumulated over all the DMA transactions in the linked list.

2.3 Safe linked list

The DMA safe linked list mode is chosen by setting ADICRz.SHCT = 0xE. It is similar to the DMA accumulated linked list in the sense that the SDCRC and RDCRC are accumulated across DMA transactions.

However you are required to load the SDCRC word with an expected DMA Address Checksum value. As soon as the next TCS is read from memory, the DMA compares the DMA Address Checksum calculated by the current DMA transaction against the expected DMA Address Checksum stored in the next TCS. If the checksums match, then the DMA proceeds with the execution of the next TCS. If the checksums do not match then the Move Engine records a SAFLL DMA Address Checksum Error and triggers a DMA RP Error Interrupt Service Request. The DMA stops the execution of the linked list operation. Assuming all the DMA Address Checksum values match during the sequence of linked list operations then the DMA Address Checksum is calculated across all DMA transactions. The RDCRC is simply accumulated over all the transactions.

2.4 Conditional linked list

The DMA conditional linked list mode is chosen by setting ADICRz.SHCT = 0xF. It is an interesting feature because it is similar to executing an “if-else” condition. In this mode, in addition to the normal address pointer in the SHADR register, the SDCRC register acts as an address pointer, instead of storing the checksum.

The decision on which of these 2 pointers to fetch the next DMA transaction from, is made based on a pattern match on the read value.

During each DMA move the pattern detection logic determines the selection of the address pointer:

- If PAT00 masked by PAT02 matches the DMA move data stored in MEm0R RD00 then as soon as the DMA move completes the ME shall load the DMA channel with the next TCS read from an address stored in SHADR.
- If PAT01 masked by PAT03 matches the DMA move data stored in MEm0R RD00 then as soon as the DMA move completes the ME shall load the DMA channel with the next TCS read from an address stored in SDCRCR.

- If both PAT00 masked by PAT02 and PAT01 masked by PAT03 match the DMA move data stored in MEm0R RD00 then as soon as the ME shall load the DMA channel with the next TCS read from an address stored in SHADR.
- If there is no pattern match then the ME continues with the DMA transaction. If a pattern match is detected then:
  - The DMA transaction ends with the current DMA move.
  - The DMA channel TSR.CH will be cleared when the DMA write move has completed.

Table 1 Summary for linked list modes selection

SHCT	Linked list mode
0xC	Normal linked list (DMALL)
0xD	Accumulated linked list (ACCLL)
0xE	Safe linked list (SAFLL)
0xF	Conditional linked list (CONLL)

3 Programming examples

In this section some coding examples are provided for the different DMA linked list modes. Note that in these programming examples, software triggering (Autostart) configuration is used. If hardware triggering is to be used for linked-list modes, the same code may be used but after setting the CHCSR.SCH = 0; CHCFGR.CHMODE = 1; and TSR.HTRE = 1. Additionally for a single transfer per hardware interrupt, the CHCFGR.RROAT shall be set to 0.

3.1 Linked list

In the simplest example of a linked list, a sequence of transactions is executed, as shown in Figure 5.

Here a DMA channel executes 3 different DMA transactions, programmed through TCSs *tcs0, tcs1* and *tcs2*. These 3 DMA transactions have different configurations, such as different source/destination addresses, data width, transfer length, or move counts for example. The data lengths and the source and destination buffers of each transaction are shown in Figure 5.

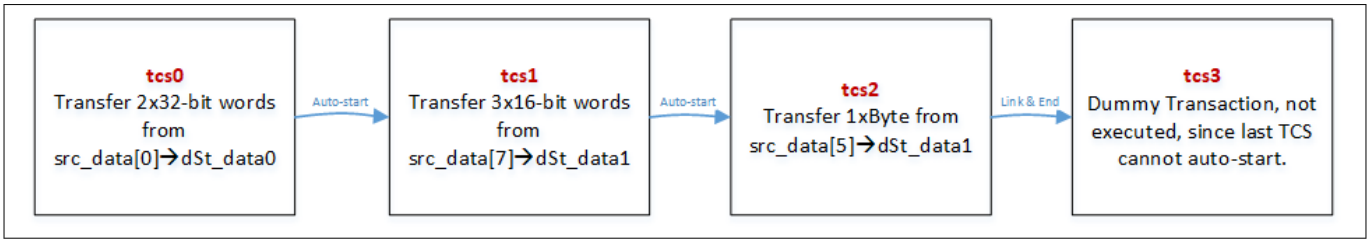

Figure 5 DMA linked list example

***Note**: The last transaction (tcs3) is simply a dummy non-linked list transaction. This is required since a non-linked list transaction is never auto-started. Hence the last executable tcs (tcs2 in this case) has to be in linked-list mode. The dummy transaction could be replaced by tcs0 if the latter is configured to not auto-start (CHCSR.SCH = 0).*

3.1.1 Allocation of the TCSs

The various global buffers and variables used in the example code sequences are shown in Figure 6. As can be seen here the TCSs are allocated as arrays with 8x32-bit words, and aligned to 32-bytes.

```
/* Allocate TCS, aligned to 32-bytes*/
__align((32)) static uint32 __tcs0[8], __tcs1[8], __tcs2[8], __tcs3[8];

/* Destination buffers for the 3 transactions*/
static uint32 __dst_data0[10], __dst_data1[10], __dst_data2[10];

/* Source data buffer*/
static uint32 __src_data[10] = {0x043C1882, 0x7F6FEF14, 0xC0CAC01A, 0xDEADBEEF,
0x05F80630, 0x0037F054, 0x0FB7FC68, 0x4096F202, 0x0C08F037, 0xBF02F074};

/* Read-data and Source-Destination CRC32 results*/
static volatile uint32 sdcrc, rdcrc;
```

Figure 6 Global variable declarations

3.1.2 Initialization of the TCSs

The TCSs are initialized according to the configurations shown in Figure 5. This initialization is shown in Figure 7 in the function ***configure_dma_ll_auto***.

```
void configure_dma_ll_auto()
{
	__tcs0[0] = 0x0; //RDCRC
	__tcs0[1] = 0x0; //SDCRC
	__tcs0[2] = (uint32)__src_data; //SADR
	__tcs0[3] = (uint32)__dst_data0; //DADR
	__tcs0[4] = 0x000C0088; //ADICR: SHCT=0xC; INCS,INCD=1
	__tcs0[5] = 0x480002; //CHCFGR: CHDW=0x2; RROAT=0x1; BLKM=0x0; TREL=0x2
	__tcs0[6] = (uint32)(__tcs1); //SHADR: Start address of TCS1
	__tcs0[7] = 0x80000000; //CHCSR: SCH=0x1

	__tcs1[0] = 0x0; //RDCRC
	__tcs1[1] = 0x0; //SDCRC
	__tcs1[2] = (uint32)&__src_data[7]; //SADR
	__tcs1[3] = (uint32)&__dst_data1[1]; //DADR
	__tcs1[4] = 0x000C0088; //ADICR: SHCT=0xC; INCS,INCD=1
	__tcs1[5] = 0x280003; //CHCFGR: CHDW=0x1; RROAT=0x1; BLKM=0x0; TREL=0x3
	__tcs1[6] = (uint32)(__tcs2); //SHADR: Start address of TCS1
	__tcs1[7] = 0x80000000; //CHCSR: SCH=0x1

	__tcs2[0] = 0x0; //RDCRC
	__tcs2[1] = 0x0; //SDCRC
	__tcs2[2] = (uint32)&__src_data[9]; //SADR
	__tcs2[3] = (uint32)__dst_data2; //DADR
	__tcs2[4] = 0x000C0088; //ADICR: SHCT=0xC; INCS,INCD=1
	__tcs2[5] = 0x080001; //CHCFGR: CHDW=0x0; RROAT=0x1; BLKM=0x0; TREL=0x1
	__tcs2[6] = (uint32)(__tcs3); //SHADR: Start address of TCS1
	__tcs2[7] = 0x80000000; //CHCSR: SCH=0x1

	//Non-linked-list transaction - does not autostart, hence loaded to DMARAM, but
not executed (dummy)
	__tcs3[0] = 0x0; //Dummy values
	__tcs3[1] = 0;
	__tcs3[2] = 0xDEADBEEF;
	__tcs3[3] = 0xA1A2A3A4;
	__tcs3[4] = 0x00000;
	__tcs3[5] = 0x0;
	__tcs3[6] = 0;
	__tcs3[7] = 0x80000000; //CHCSR:SCH=1, but TCS won't be executed.
}
```

Figure 7 DMA linked-list initialization of the TCSs

3.1.3 Starting the linked list

The linked-list is started by simply copying the TCS0 to any DMA channel. Since the *CHCSR.SCH* bit is set in all the TCSs, each DMA transaction will automatically start and execute its DMA moves, and will terminate after loading TCS3, which is not automatically started, since it is a non-linked list TCS.

The function *copy_tcs0_to_dma* in Figure 8 copies TCS0 to DMA Channel-0, so starting the linked list.

```
void copy_tcs0_to_dma()
{
	DMA_RDCRCR000.U	= __tcs0[0];
	DMA_SDCRCR000.U	= __tcs0[1];
	DMA_SADR000.U	= __tcs0[2];
	DMA_DADR000.U	= __tcs0[3];
	DMA_ADICR000.U	= __tcs0[4];
	DMA_CHCFGR000.U	= __tcs0[5];
	DMA_SHADR000.U	= __tcs0[6];
	DMA_CHCSR000.U	= __tcs0[7]; //SCH=1 in tcs0. Hence the channel will start with
tcs0 automatically.

}
```

Figure 8 Starting the linked list; Copy TCS0 to DMA channel

In summary, to execute the linked-list:

- Define the TCSs in memory, aligned to 32-bytes, and also the source and destination buffers.
- Initialize the TCSs
  - call the function configure_dma_ll_auto()
- Trigger the linked-list
  - call the function copy_tcs0_to_dma()

3.1.4 Results

After the execution of the DMA linked list, the destination buffers look as follows:

```
__dst_data2 = {0x74, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0}
__dst_data1 = {0x0, 0x4096F202, 0xF037, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0}
__dst_data0 = {0x043C1882, 0x7F6FEF14, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0}

__src_data = {0x043C1882, 0x7F6FEF14, 0xC0CAC01A, 0xDEADBEEF, 0x05F80630, 0x0037F054, 0x0FB7FC68,
0x4096F202, 0x0C08F037, 0xBF02F074}
```

Figure 9 Destination buffer values after DMALL execution

The DMA registers after the DMA linked list execution looks as shown in Figure 10. Note that the dmmy TCS (tcs3) has been loaded, but not executed.

Also note that the SDCRC & RDCRC values are 0; i.e. they are overwritten on a TCS load.

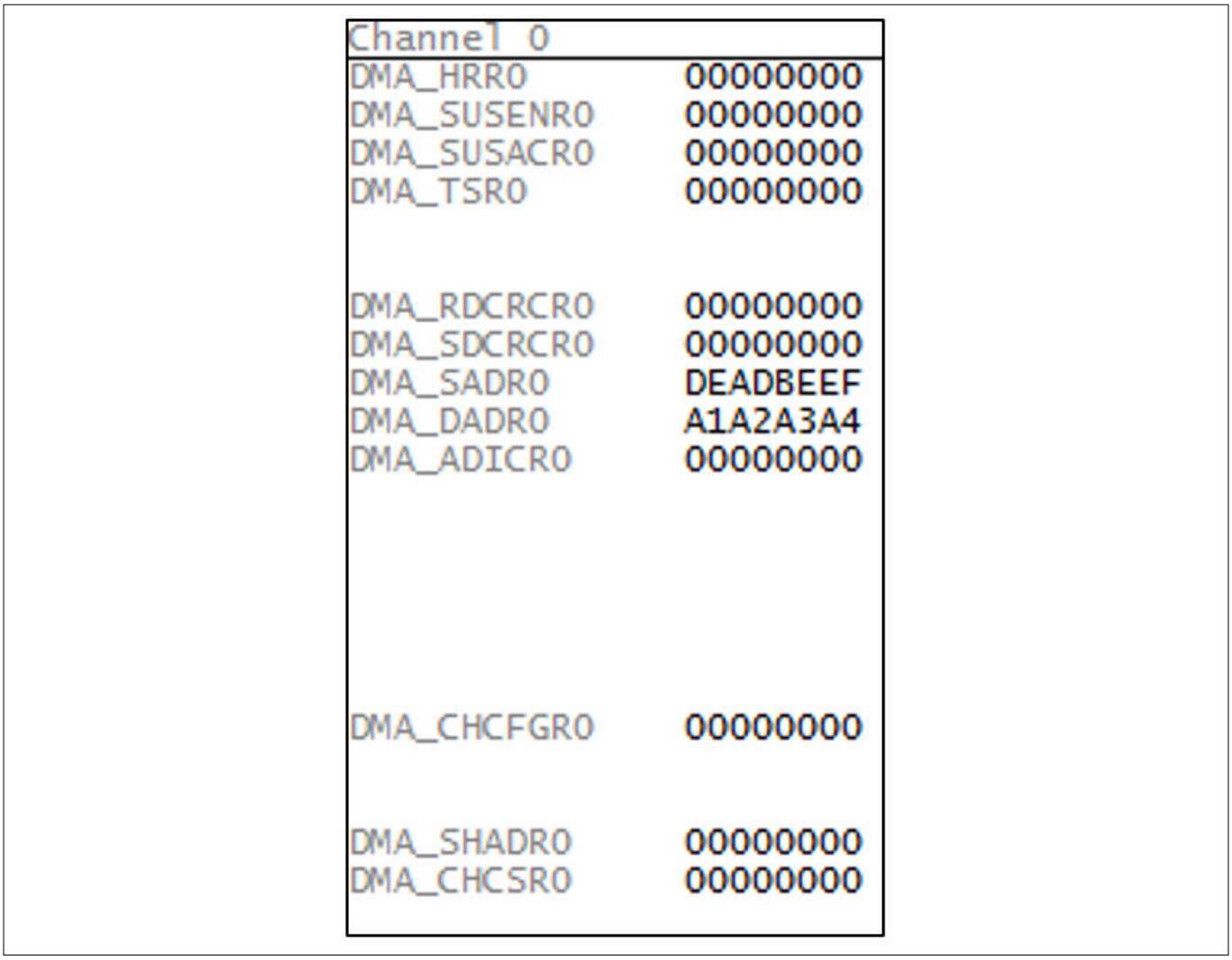

Figure 10 DMA channel register values after DMALL execution

3.2 Accumulated linked list

The accumulated linked list is similar to a normal linked-list as described in the previous section, except that the CRC values are accumulated across multiple TCSs and are not overwritten. This allows the software to check for safe operation.

The function ***accumulate_crc*** shows an example of CRC computation for the DMA SDCRC and RDCRC. It takes as arguments, the DMA parameters such as source, destination addresses, transfer and move count for example, and calculates the RDCRC and SDCRC checksums. It is also possible to accumulate the computation across multiple calls (hence multiple transaction configurations). This function has been checked for data widths up to 32-bits.

Attention**: **Due to errata DMA_TC.039, the RDCRC for data widths less than 32 bits is not supported when using the TC39xA product.

```
/* sadr: Source address start (assumes incrementing)
 * dadr: Destination address start (assumes incrementing)
 * transfers: No:of transfers in the transaction as defined by TREL
 * moves: No:of moves in each transfer as defined by BLKM.
 * chdw_bytes: Channel data width in bytes, as defined by CHDW.
 * accumulate: 0: start a new computation; 1: continue accumulating from previously
computed value
 */
void accumulate_crc(uint8* sadr, uint8* dadr, uint32 transfers, uint32 moves, uint32
chdw_bytes, uint32 accumulate)
{
	int i;
	static uint32 a, bsd = 0, brd = 0;

	if(!accumulate){
		a = 0;
		bsd = 0;
		brd = 0;
		sdcrc = 0;
		rdcrc = 0;
	}


	for (i = 0; i < transfers*moves; i++){
		a = (uint32)(sadr + i*chdw_bytes);
		bsd = __crc32bw(bsd, a);

		a = (uint32)(dadr + i*chdw_bytes);
		bsd = __crc32bw(bsd, a);

		if(chdw_bytes >= 4){
			a = *(uint32*)(sadr + i*chdw_bytes);
			brd = __crc32bw(brd, a);
		}
		else if (chdw_bytes == 2){
			a = *(uint8*)(sadr + 2*i + 1);
			brd = __crc32b(brd, a);
			a = *(uint8*)(sadr + 2*i);
			brd = __crc32b(brd, a);
		}
		else if (chdw_bytes == 1){
			a = *(uint8*)(sadr + i);
			brd = __crc32b(brd, a);
		}
	}

	sdcrc = bsd;
	rdcrc = brd;

}
```

Figure 11 CRC computation over transactions

The CRCs can be pre-computed during the initialization of the TCSs. This is shown in the function ***configure_dma_acll_auto***.

```
void configure_dma_acll_auto()
{
	__tcs0[0] = 0x0; //RDCRC
	__tcs0[1] = 0x0; //SDCRC
	__tcs0[2] = (uint32)__src_data; //SADR
	__tcs0[3] = (uint32)__dst_data0; //DADR
	__tcs0[4] = 0x000D0088; //ADICR: SHCT=0xD; INCS,INCD=1
	__tcs0[5] = 0x480002; //CHCFGR: CHDW=0x2; RROAT=0x1; BLKM=0x0; TREL=0x2
	__tcs0[6] = (uint32)(__tcs1); //SHADR: Start address of TCS1
	__tcs0[7] = 0x80000000; //CHCSR: SCH=0x1

	__tcs1[0] = 0x0; //RDCRC
	__tcs1[1] = 0x0; //SDCRC
	__tcs1[2] = (uint32)&__src_data[7]; //SADR
	__tcs1[3] = (uint32)&__dst_data1[1]; //DADR
	__tcs1[4] = 0x000D0088; //ADICR: SHCT=0xD; INCS,INCD=1
	__tcs1[5] = 0x280003; //CHCFGR: CHDW=0x1; RROAT=0x1; BLKM=0x0; TREL=0x3
	__tcs1[6] = (uint32)(__tcs2); //SHADR: Start address of TCS2
	__tcs1[7] = 0x80000000; //CHCSR: SCH=0x1

	__tcs2[0] = 0x0; //RDCRC
	__tcs2[1] = 0x0; //SDCRC
	__tcs2[2] = (uint32)&__src_data[9]; //SADR
	__tcs2[3] = (uint32)__dst_data2; //DADR
	__tcs2[4] = 0x000D0088; //ADICR: SHCT=0xC; INCS,INCD=1
	__tcs2[5] = 0x080001; //CHCFGR: CHDW=0x0; RROAT=0x1; BLKM=0x0; TREL=0x1
	__tcs2[6] = (uint32)(__tcs3); //SHADR: Start address of TCS3
	__tcs2[7] = 0x80000000; //CHCSR: SCH=0x1

	/* Non-linked-list transaction, does not autostart, hence loaded to DMARAM, but not
executed (dummy) */
	__tcs3[0] = 0x0; //Dummy values
	__tcs3[1] = 0;
	__tcs3[2] = 0xDEADBEEF;
	__tcs3[3] = 0xA1A2A3A4;
	__tcs3[4] = 0x00000;
	__tcs3[5] = 0x0;
	__tcs3[6] = 0;
	__tcs3[7] = 0x80000000; //CHCSR:SCH=1, but TCS won't be executed.

	/* Accumulate the CRC values over the 3 transactions*/
	accumulate_crc(__src_data, __dst_data0, 2, 1, 4, 0);
	accumulate_crc(&__src_data[7], &__dst_data1[1], 3, 1, 2, 1);
	accumulate_crc(&__src_data[9], __dst_data2, 1, 1, 1, 1);
}
```

Figure 12 DMA Accumulated linked list configuration - with computation of CRCs.

The function call sequence to execute the accumulated linked list remains similar:

- Define the TCSs in memory, aligned to 32-bytes, and also the source and destination buffers.
- Initialize the TCSs and compute the CRC
  - call the function configure_dma_acll_auto()
- Trigger the accumulated linked-list
  - call the function copy_tcs0_to_dma()

3.2.1 Results

The data transferred with the accumulated linked list is the same as before, but what is different is the value of the CRC registers after the transaction finished. This is shown in Figure 13.

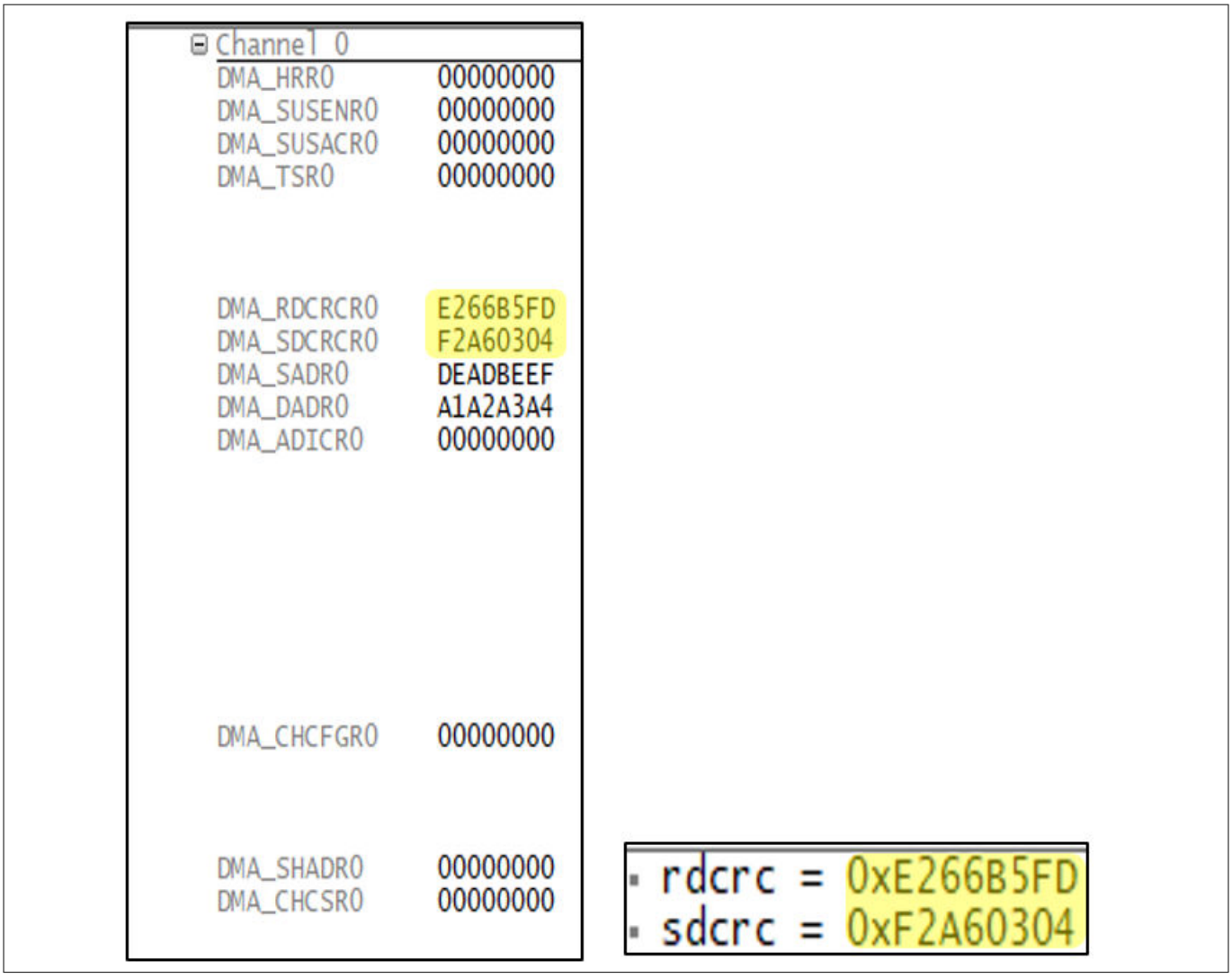

Figure 13 DMA channel registers after ACCLL execution and the SW computed CRCs

The CRC computed by calling the ***accumulate_crc*** function is also shown. As can be seen, the **sdcrc** and the **rdcrc** computed by software over all the transactions match the DMA result.

3.3 Safe linked list

A DMA safe linked list follows an accumulated linked list; i.e. the CRCs are accumulated over all the transactions in the linked list. The difference is that a new TCS is loaded only if the calculated SDCRCR matches the value stored in the next TCS.

The ***configure_dma_safll_auto*** configures a safe linked list and also gives the possibility to inject an error and test it, by setting the *inject_error* argument to 1.

```
void configure_dma_safll_auto(int inject_error)
{
	__tcs0[0] = 0x0; //RDCRC
	__tcs0[1] = 0x0; //SDCRC
	__tcs0[2] = (uint32)__src_data; //SADR
	__tcs0[3] = (uint32)__dst_data0; //DADR
	__tcs0[4] = 0x000E0088; //ADICR: SHCT=0xE; INCS,INCD=1
	__tcs0[5] = 0x480002; //CHCFGR: CHDW=0x2; RROAT=0x1; BLKM=0x0; TREL=0x2
	__tcs0[6] = (uint32)(__tcs1); //SHADR: Start address of TCS1
	__tcs0[7] = 0x80000000; //CHCSR: SCH=0x1

	__tcs1[0] = 0x0; //RDCRC
	__tcs1[1] = 0x0; //SDCRC
	__tcs1[2] = (uint32)&__src_data[7]; //SADR
	__tcs1[3] = (uint32)&__dst_data1[1]; //DADR
	__tcs1[4] = 0x000E0088; //ADICR: SHCT=0xE; INCS,INCD=1
	__tcs1[5] = 0x280003; //CHCFGR: CHDW=0x1; RROAT=0x1; BLKM=0x0; TREL=0x3
	__tcs1[6] = (uint32)(__tcs2); //SHADR: Start address of TCS2
	__tcs1[7] = 0x80000000; //CHCSR: SCH=0x1

	__tcs2[0] = 0x0; //RDCRC
	__tcs2[1] = 0x0; //SDCRC
	__tcs2[2] = (uint32)&__src_data[9]; //SADR
	__tcs2[3] = (uint32)__dst_data2; //DADR
	__tcs2[4] = 0x000E0088; //ADICR: SHCT=0xE; INCS,INCD=1
	__tcs2[5] = 0x080001; //CHCFGR: CHDW=0x0; RROAT=0x1; BLKM=0x0; TREL=0x1
	__tcs2[6] = (uint32)(__tcs3); //SHADR: Start address of TCS3
	__tcs2[7] = 0x80000000; //CHCSR: SCH=0x1

	/* Non-linked-list transaction - does not autostart, hence loaded to DMARAM,
but not executed (dummy) */
	__tcs3[0] = 0x0; //Dummy values
	__tcs3[1] = 0;
	__tcs3[2] = 0xDEADBEEF;
	__tcs3[3] = 0xA1A2A3A4;
	__tcs3[4] = 0x00000;
	__tcs3[5] = 0x0;
	__tcs3[6] = 0;
	__tcs3[7] = 0x80000000; //CHCSR:SCH=1, but TCS won't be executed.


	accumulate_crc(__src_data, __dst_data0, 2, 1, 4, 0);
	__tcs1[1] = sdcrc; //SDCRC @ end of TCS0


	accumulate_crc(&__src_data[7], &__dst_data1[1], 3, 1, 2, 1);
	__tcs2[1] = sdcrc; //SDCRC @ end of TCS1

	if(inject_error)
		__tcs2[1]++; //Inject an error in the SDCRC


	accumulate_crc(&__src_data[9], __dst_data2, 1, 1, 1, 1);
	__tcs3[1] = sdcrc; //SDCRC @ end of TCS2
}
```

Figure 14 Safe linked list configuration example

The function call sequence to execute the safe linked list remains similar:

- Define the TCSs in memory, aligned to 32-bytes, and also the source and destination buffers.

- Initialize the TCSs and compute the CRC
  - call the function **configure_dma_safll_auto(<argument>)** with 0 as argument for no error, and 1 to inject an error.
- Trigger the safe linked-list
  - call the function **copy_tcs0_to_dma()**

3.3.1 Results

Without any error ***(configure_dma_safll_auto(0))*** then the results are similar to the accumulated linked list. Please refer to Figure 13.

However when ***configure_dma_safll_auto(1)*** is called (so the error injection is enabled), then the output is as follows:

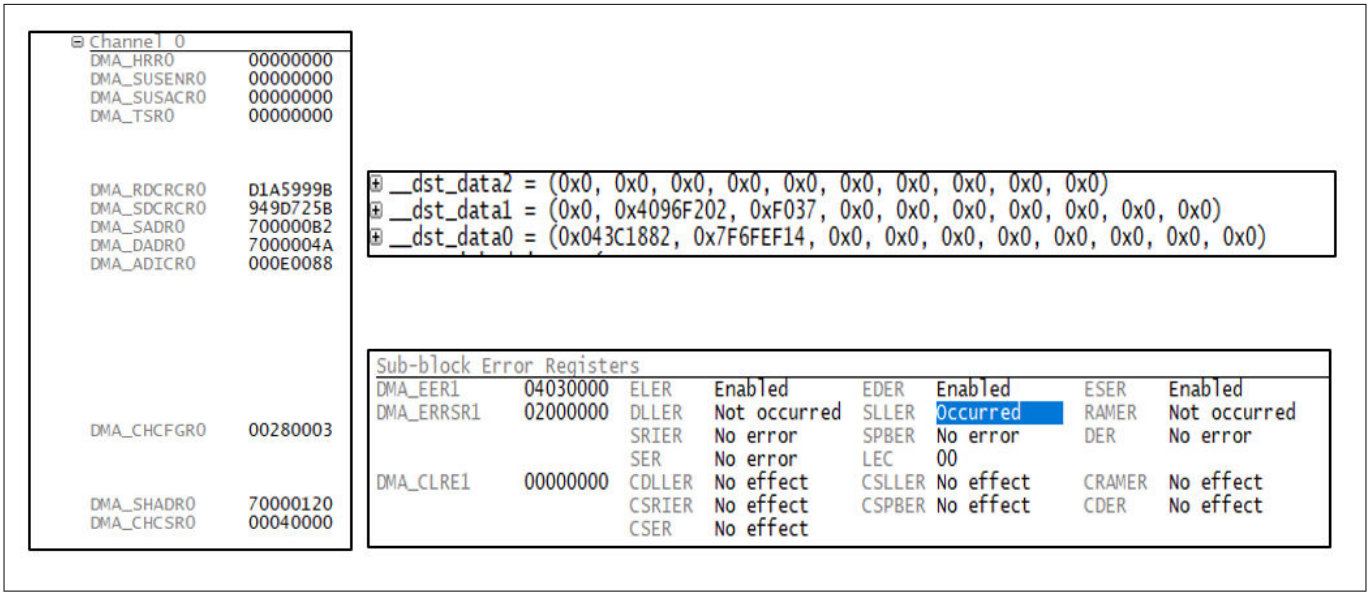

Figure 15 DMA Channel registers and Destination buffers after SAFLL error.

As can be seen in Figure 15, the DMA did not execute TCS2, since the SDCRCR did not match the expected value. Correspondingly, the SLLER bit in the ERRSR1 register is set, indicating that a DMA safe linked list error occurred.

3.4 Conditional linked list

A simple way to imagine the DMA conditional linked list is as an ‘if-else’ condition based on a transferred byte value. In the example shown in Figure 16, TCS0 initially transfers 16 bytes. Whenever a transferred byte matches 0x3X (i.e. the MSB nibble in the byte = 0x3, LSB nibble is don’t care), then TCS0 is finished after the DMA move, and the next DMA transaction is loaded.

When the pattern in question (here 0x3X) is matched in PAT[0] masked by PAT[2], then the next TCS is loaded from the address pointed to by SHADR. If the matched pattern is programmed in PAT[1] masked by PAT[3], then the next TCS is loaded from the address pointed to by SDCRCR.

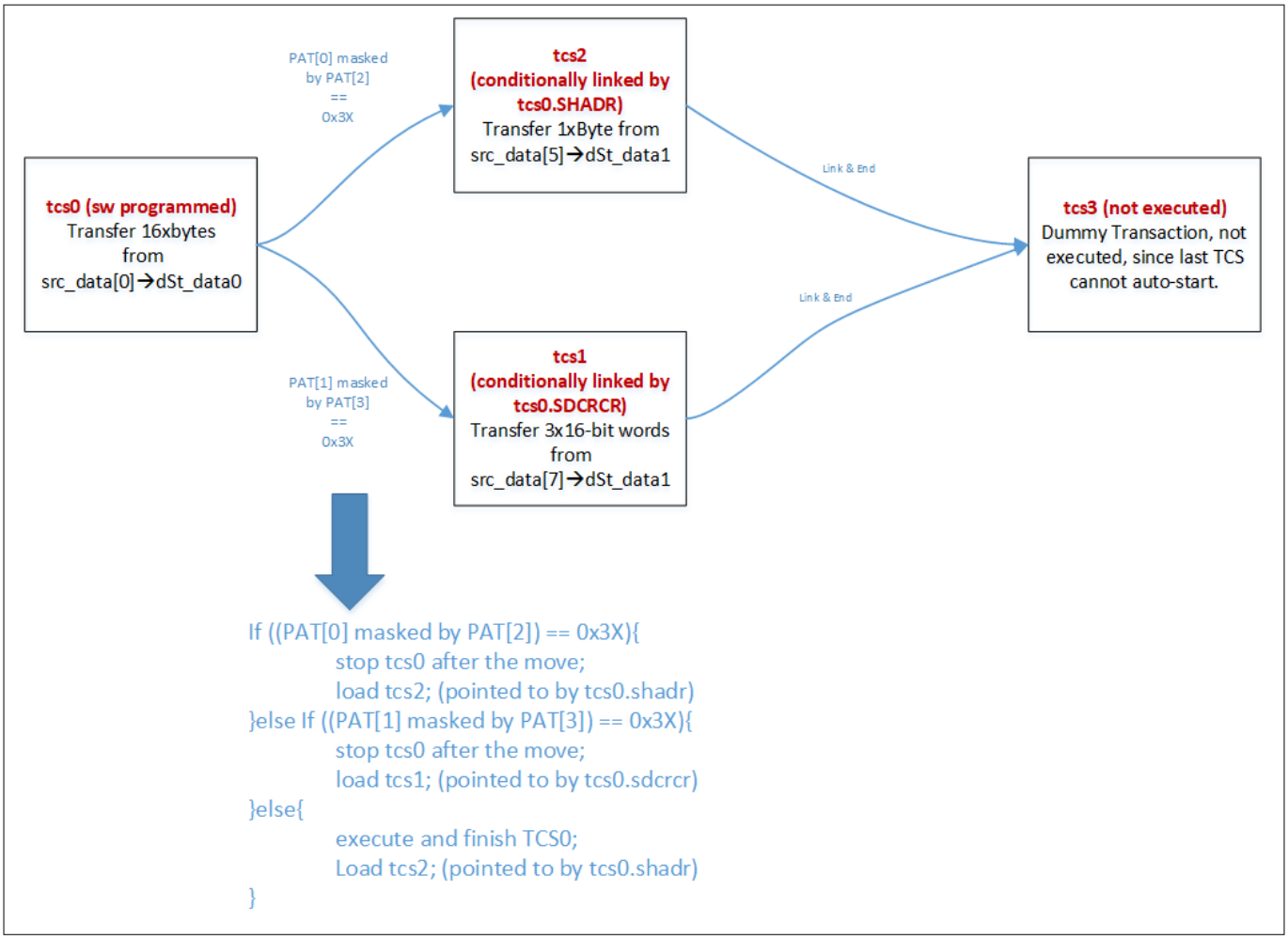

Figure 16 Conditional linked list example

Figure 17 shows the implementation of ***configure_dma_conll*** function and configures the example shown in Figure 16. The argument of the function can be set to 0 or 1 to choose whether the pattern match is programmed in PAT[0] or PAT[1] (i.e. whether tcs2 or tcs1 is loaded on the condition match) respectively.

```
void configure_dma_conll(int prr)
{
	__tcs0[0] = 0x0; //RDCRC
	__tcs0[1] = (uint32)(__tcs1); //SDCRC: Start address of TCS1
	__tcs0[2] = (uint32)__src_data; //SADR
	__tcs0[3] = (uint32)__dst_data0; //DADR
	__tcs0[4] = 0x000F0088; //ADICR: SHCT=0xF; INCS,INCD=1
	__tcs0[5] = 0x80010; //CHCFGR: CHDW=0x0; RROAT=0x1; BLKM=0x0; TREL=0x10
	__tcs0[6] = (uint32)(__tcs2); //SHADR: Start address of TCS2
	__tcs0[7] = 0x80000000; //CHCSR: SCH=0x1

	__tcs1[0] = 0x0; //RDCRC
	__tcs1[1] = 0x0; //SDCRC
	__tcs1[2] = (uint32)&__src_data[7]; //SADR
	__tcs1[3] = (uint32)&__dst_data1[1]; //DADR
	__tcs1[4] = 0x000C0088; //ADICR: SHCT=0xC; INCS,INCD=1
	__tcs1[5] = 0x280003; //CHCFGR: CHDW=0x0; RROAT=0x1; BLKM=0x0; TREL=0x3
	__tcs1[6] = (uint32)(__tcs3); //SHADR: Start address of TCS3
	__tcs1[7] = 0x80000000; //CHCSR: SCH=0x1

	__tcs2[0] = 0x0; //RDCRC
	__tcs2[1] = 0x0; //SDCRC
	__tcs2[2] = (uint32)&__src_data[9]; //SADR
	__tcs2[3] = (uint32)__dst_data2; //DADR
	__tcs2[4] = 0x000C0088; //ADICR: SHCT=0xC; INCS,INCD=1
	__tcs2[5] = 0x080001; //CHCFGR: CHDW=0x0; RROAT=0x1; BLKM=0x0; TREL=0x1
	__tcs2[6] = (uint32)(__tcs3); //SHADR: Start address of TCS3
	__tcs2[7] = 0x80000000; //CHCSR: SCH=0x1

	//Non-linked-list transaction - does not autostart, hence loaded to DMARAM, but not
executed (dummy)
	__tcs3[0] = 0x0; //Dummy values
	__tcs3[1] = 0;
	__tcs3[2] = 0xDEADBEEF;
	__tcs3[3] = 0xA1A2A3A4;
	__tcs3[4] = 0x00000;
	__tcs3[5] = 0x0;
	__tcs3[6] = 0;
	__tcs3[7] = 0x80000000; //CHCSR:SCH=1, but TCS won't be executed.

.
```

Figure 17 Configuring conditional linked list- part1

```
.
.
.
        /* Setup the Pattern match:
         * In this example, the pattern match is setup to check a byte 0x3*. i.e. - MSB
nibble in the byte = 0x3 & LSB nibble is don't care.
         *
         * if prr = 0 when calling this function, then PAT0 is used and TCS from SHADR
pointer is fetched on a match.
         * if prr = 1 when calling this function, then PAT1 is used and TCS from SDCRCR
pointer is fetched on a match.
         *
         */
        DMA_PRR0.U = 0;

        if(!prr){
                DMA_PRR0.B.PAT00 = 0x30;
                DMA_PRR0.B.PAT02 = 0x0F;

                DMA_PRR0.B.PAT01 = 0x0;
                DMA_PRR0.B.PAT03 = 0x0;


        }else{
                DMA_PRR0.B.PAT00 = 0x0;
                DMA_PRR0.B.PAT02 = 0x0;

                DMA_PRR0.B.PAT01 = 0x30;
                DMA_PRR0.B.PAT03 = 0x0F;

        }

}
```

Figure 18 Configuring conditional linked list- Part2

The function call sequence to execute the conditional linked list remains similar:

- Define the TCSs in memory, aligned to 32-bytes, and also the source and destination buffers.
- Initialize the TCSs and pattern match registers
  - call the function **configure_dma_conll (<argument>)**, with 0 as argument to set the pattern match using PAT[0]/PAT[2], and 1 to set the pattern match using PAT[1]/PAT[3].
- Trigger the conditional linked-list
  - call the function **copy_tcs0_to_dma()**

3.4.1 Results

First, the DMA conditional linked list is executed by setting the pattern match condition in PAT[0] and PAT[2], so the function ***configure_dma_conll*** is called with argument 0.

The results are as shown in Figure 19. Note from the values in __dst_data0 buffer that the tcs0 stopped executing after transferring the byte 0x3C; since the pattern match condition 0x3X matches. Hence tcs2 is next executed and finished.

```
__dst_data2 = {0x74, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0}
__dst_data1 = {0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0}
__dst_data0 = {0x003C1882, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0}

__src_data = {0x043C1882, 0x7F6FEF14, 0xC0CAC01A, 0xDEADBEEF, 0x05F80630, 0x0037F054, 0x0FB7FC68,
0x4096F202, 0x0C08F037, 0xBF02F074}
```

Figure 19 CONLL execution result with *configure_dma_conll(0)*

Second, the conditional linked list is executed by setting the pattern match condition in PAT[1] and PAT[3] (i.e. function ***configure_dma_conll*** is called with argument 1).

The results are as shown in Figure 20. Note from the values in __dst_data0 buffer that the tcs0 stopped executing after transferring the byte 0x3C; since the pattern match condition 0x3X matches. Hence tcs1 is next executed and finished.

```
__dst_data2 = {0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0}
__dst_data1 = {0x0, 0x4096F202, 0xF037, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0}
__dst_data0 = {0x003C1882, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0}

__src_data = {0x043C1882, 0x7F6FEF14, 0xC0CAC01A, 0xDEADBEEF, 0x05F80630, 0x0037F054, 0x0FB7FC68,
0x4096F202, 0x0C08F037, 0xBF02F074}
```

Figure 20 CONLL execution result with *configure_dma_conll(1)*

4 Revision history

Document revision	Date	Description of change
1.0	January 2019	First release
V1.1	2024-04-17	Template update Removed Note from the Results section

Trademarks

All referenced product or service names and trademarks are the property of their respective owners.

Edition 2024-08-14

Published by

Infineon Technologies AG

81726 Munich, Germany

Do you have a question about any aspect of this document?

Email: erratum@infineon.com

Document reference
IFX-vod1711277868674

Important notice

The information given in this document shall in no event be regarded as a guarantee of conditions or characteristics ("Beschaffenheitsgarantie").

With respect to any examples, hints or any typical values stated herein and/or any information regarding the application of the product, 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.

In addition, any information given in this document is subject to customer's compliance with its obligations stated in this document and any applicable legal requirements, norms and standards concerning customer's products and any use of the product of Infineon Technologies in customer's applications.

The data contained in this document is exclusively intended for technically trained staff. It is the responsibility of customer's technical departments to evaluate the suitability of the product for the intended application and the completeness of the product information given in this document with respect to such application.

Warnings

Due to technical requirements products may contain dangerous substances. For information on the types in question please contact your nearest Infineon Technologies office.

Except as otherwise explicitly approved by Infineon Technologies in a written document signed by authorized representatives of Infineon Technologies, Infineon Technologies' products may not be used in any applications where a failure of the product or any consequences of the use thereof can reasonably be expected to result in personal injury.