Chapter 1

General Purpose DMA

DRAFT (full of errors)

The General Purpose Direct Memory Access Controller (GPDMA) can transfer blocks of data between memory mapped locations. It can do this in parallel to the CPU code execution, and also in parallel to Ethernet DMA operations on the second AHB bus.

There are two channels in the GPDMA. You can think of them as two block transfer tasks, which can be activated at the same time. Both channels offer equivalent functionality.

The following transfer types are supported by each channel:

- Memory-to-Memory transfers
- Memory-to-Peripheral transfers
- Peripheral-to-Memory transfers
- Peripheral-to-Peripheral transfers

As the GPDMA is located at AHB1 bus, its scope is everything an AHB1 master can access. This includes the AHB1 RAM block starting at address *0x7FD00000*, all external memory via the EMC, and of course the APB peripherals through the AHB1-APB bridge. Memory on the AHB2 bus (the Ethernet segment) and the CPU's local bus memory (flash and system SRAM) are not accessible by the GPDMA.

Not all APB peripherals are DMA ready, but a selection of them:

- SSP0 (Synchronous Serial Interface)
- SSP1 (Synchronous Serial Interface)
- I^{2}S (Audio Interface)
- MCI (SD Card/MMC Memory Card Interface)

1.1 DMA Basics

While memory-to-memory transfers are carried out immediately, as soon as the AHB1 bus is available, any DMA transfer involving a peripheral is initiated by the peripheral through request lines.

Standard signals are defined for DMA capable peripheral units. These signals are divided into *single request* and *burst request* lines. They are further subdivided into a regular request line and a last-request line. For both single and burst request lines, there's an acknowledge signal from the DMA.

....

1.2 Transfer Width

Transfer width is the data width used to transfer a single data item from source or to destination. Source and destination can have different transfer widths. You can choose transfer width to be 8 bits (byte), 16 bits (half-word) or 32 bits (word).

For memory transfers, you should choose the largest possible width for your memory area. For internal memory, this is 32 bits. For external memory with smaller data bus, you want to reduce the transfer width accordingly. Source or destination buffers in memory must have a size which is a multiple of the transfer width.

For peripheral transfers, transfer width is usually determined by the way you configured the peripheral. As an example, if you configure the SSP interface for a frame width of 8 bits, the transfer width must be 8 bits as well. This is because each transfer corresponds to one FIFO location, i.e. one SSP frame.

1.3 Burst Size

Data items can be transferred from source or to destination in bursts, for improved efficiency. Bursts can reduce the overhead of the DMAC having to acquire and release the bus repeatedly. However, bursts which are too long, may increase latency for other bus masters waiting for access.

Burst size is in multiples of transfer width. With a transfer width of 32 bits, a burst size of 4 means that a burst involves 4 * 32 bit = 16 bytes.

Several constraints limit your choice of burst size:

- *Memory as source or destination*
  Optimum burst size is the one that fills the internal DMA FIFO with one burst. As the GPDMA is configured with a four-word FIFO, you may set the burst size as high as four. Larger burst sizes do not make sense.
- *Peripheral transfer*
  Burst size is determined by the peripheral's FIFO. Burst requests are made by the peripheral, if its receive FIFO is filled up to 50%, or if its transmit FIFO dropped down to 50% fill level. In both cases, a burst transfer may now receive/send half the FIFO size. In case of the SSP, which has a FIFO size of eight frames, burst size must be set to four.

There may be reasons for selecting a burst size which doesn't follow the rules above. Such exceptions are described below when various usage scenarios are discussed.

1.4 Flow Control

Flow control is the concept of controlling when a block data transfer is over. Either the DMAC or the peripheral can be in charge of it. Flow control is not to be mixed up with the request/acknowledge handshaking between DMAC and peripherals. Transfers from DMAC to peripheral and vice versa are only done upon request of the peripheral. The flow controller is just the one who knows when all the chunks of data have been exchanged.

Whether DMAC or peripheral should be assigned to be the flow controller, depends on the nature of the peripheral and/or the way you use it. We will discuss the possible options and show the pros and cons.

1.4.1 Using DMAC as the flow controller

When DMAC is selected as the flow controller, you need to tell it the number of *source* transfers. Once it completes this number of transfers, DMA operation is stopped, and the *Terminal Count* interrupt is fired, if enabled.

Upon initialization of a DMA block transfer, the number of source transfers is written to *DMACCxControl.TransferSize*. This field counts down to zero with every completed transfer. You *might* use this field as a progress indicator while the DMAC channel is still enabled, but you mustn't rely on it to reflect the actual amount of data already transferred at the time of the reading. Only after you disable the channel will this field be accurate.

Memory-to-Peripheral. While the number of transfers left for the destination peripheral is equal to or greater than the configured burst size for the peripheral, the DMAC will only evaluate the DMABREQ (burst request) from that peripheral, while the DMASREQ (single request) signal will be ignored! Only after the number of transfers left drops below the configured burst size, will the DMAC read the DMASREQ signal, and send the remaining data as single transfers.

Peripheral-to-Memory. While the number of transfers left for the source peripheral is equal to or greater than the configured burst size for the peripheral, the DMAC will only evaluate the DMABREQ (burst request) from that peripheral, while the DMASREQ (single request) signal will be ignored! Only after the number of transfers left drops below the configured burst size, will the DMAC read the DMASREQ signal, and accept the remaining data as single transfers.

DRAFT (full of errors)

No rule without exception: The above is true, when you set the source burst size to four or above. If you set the source burst size to *one*, then the DMAC will use the DMASREQ right from the beginning, independent of the number of transfers left!

1.4.2 Using the peripheral as the flow controller

When the peripheral is selected as the flow controller, you need to set the *TransferSize* field in the *DMACCxControl* register to zero. However, in case of using *Linked List Items*, the *TransferSize* will have to be set to the size of the first list item.

Without LLI, the amount of data being transferred is under total control of the peripheral. You must only use this mode when the peripheral reliably sets an upper limit to the number of transfers. The DMAC itself will not be able to stop, because it has no idea where the memory buffer ends. In the peripheral-to-memory case, the effect of writing beyond the buffer boundary will be disastrous.

The peripheral can terminate a block transfer by asserting its DMALSREQ or DMALBREQ lines. Streaming peripherals usually do not have these lines, but block-oriented peripherals (MCI) do have them.

1.4.3 When would you choose DMAC to be the flow controller?

- *Memory-to-memory transfers.*
  There is no other option. DMAC will always be the flow controller.
- *You know the amount of data to be transferred in advance, and you send to or receive from a streaming peripheral.*
  If you need to transfer an exact amount of data to or from a peripheral, that doesn't have a data counter by itself, this is the way to do it. An example is the SSP transmit interface: It sends whatever you write into its TX FIFO. If the DMA is programmed to transfer n frames to the SSP TX channel, exactly n frames are sent via SSP.
- *The number of memory-to-peripheral transfers is not known, but has an upper limit.*
  You would then configure the *TransferSize* field to the upper limit, which is usually the size of the buffer from which the data stream is served. You would expect the average block transfer to involve less elements than the buffer contains. While the block transfer is still running, the DMAC refills the peripheral's FIFO using burst transfers. There will always be enough data to transmit, even if you decide to stop your block transfer after a number of transfers which is not a multiple of the burst size. By means of protocol (data content, handshake lines) you determine the end of the block transfer. Once this is reached, you simply disable the DMA channel. Usually, you will also have to drain the peripheral's transmit FIFO.
- *The number of peripheral-to-memory transfers is not known, but has an upper limit.*
  You would then configure the *TransferSize* field to the upper limit, which is usually the size of the buffer to which the data stream is written. You would expect the average block transfer to involve less elements than the buffer contains. You must distinguish two scenarios:
  1. *You expect an arbitrary number of transfers, which is not a multiple of half the peripheral's FIFO size.*
     In this case, set the source burst size to 1, destination burst size should be 1 as well. The destination transfer width must match the source transfer width.

     In this mode, DMAC will respond to every kind of DMA request, burst and single, immediately. Hence in general, most of the transfers will be done as single transfers, because the DMAC will usually be handling the DMASREQ before the RX FIFO in the peripheral has a chance to fill up to the DMABREQ level. Once you decide to terminate a block transfer, you can drain the DMAC's FIFO by setting its *DMACCxConfiguration.H* bit, then waiting for the FIFO to be drained. Once the *DMACCxConfiguration.A* bit goes low, all data will have been drained to memory. No data is left in the source peripheral's FIFO, as the DMAC has previously fetched all data through single transfers.
  2. *You can tolerate that only multiples of half the peripheral's FIFO size are transferred.*
     In this case, set the source burst width to match half the peripheral's FIFO size. For the SSP channel, the FIFO size is 8, hence a burst size of 4 must be selected.

     In this mode, DMAC will only look for burst requests, while the number of transfers left is at least the burst size. Only once the numbers of transfers left drops below the burst size, will

DRAFT (full of errors)

DMAC respond to single transfer requests! If your communication block involves a number of transfers which is not a multiple of the selected burst size, you may end up with the last few characters left in the peripheral's RX FIFO. In this case, use the previous setting with burst size one.

While the block transfer is still running, the DMAC refills the peripheral's FIFO using burst transfers from the source peripheral. There will be no single transfers, as long as the number of remaining transfers is equal to or greater than the burst size. Only for the last transfers (less than burst size) will the DMAC look at the peripheral's DMASREQ line to do single transfers.
Note: If the number of transfers in a block is not a multiple of the burst size, and if it is well below the limit set by *TransferSize*, the DMAC will not do the last few transfers, but leave the data in the source peripheral. After disabling the DMAC, you would then have to fetch the remaining data manually. This is not a recommended operating mode.

1.4.4 When would you choose the peripheral to be the flow controller?

- *You know the amount of data to be transferred in advance, and you work with a block-oriented peripheral.*
  If you need to transfer an exact amount of data to or from a peripheral, that has its own control over the amount of data being transferred, this is the way to do it. This works for the MCI peripheral (SD card and MMC controller). The MCI is instructed to read or write a certain amount of data, usually multiples of the sector size. It will signal the end of the transfer to the DMAC using its DMALSREQ/DMALBREQ lines. The *TransferSize* field must be set to 0 in this case. Of course the source or destination buffer in memory must be large enough for the amount of data to be transferred.

- *The number of transfers is unlimited.*
  This mode requires the extensive use of *Linked List Items*. Data comes from or goes to memory in chunks. The CPU's task is to process new data, once the DMA has consumed an LLI.
  You can use this mode for streaming data continuously. A good example is the I^{2}S interface, when sending audio samples coming from an MP3 decoder.

1.5 Halt a DMA transfer temporarily

Sometimes it may be necessary to halt and later resume a DMA block transfer, which has not yet finished. The DMAC channels offer the *DMACCxConfiguration.H* bit to do this. When you set this bit, the DMAC channel will no longer do any transfers on its source side, after the currently active transfer is over. Any data in the DMAC channel's FIFO will still be drained to the destination. As soon as the channel FIFO is empty, the *DMACCxConfiguration.A* bit is cleared. Now you can be sure, that the DMAC channel will not perform any source or destination transfer anymore. After a while you clear the *DMACCxConfiguration.H* bit to resume DMA operation.

Note that the FIFO can only be drained, if the destination requests more data. If the destination is memory, there is no problem, because data can be written to memory almost immediately. However, if the destination is a peripheral whose transmit FIFO is full, data can only be drained, once the peripheral had a chance to send out data, and then request some more from the DMAC. If the peripheral is not able to send data right now (for instance if it's an SSP is slave mode), *DMACCxConfiguration.A* may well never get cleared. There is also no way to prevent the destination transfers from happening somewhere in the future, even if the *DMACCxConfiguration.H* bit is set.

If in such a case you need to have a guarantee, that no future transfers (source and destination) are done by the DMAC, the only way is to disable such a channel completely. When you disable a channel by clearing its *DMACCxConfiguration.E* bit, the DMAC channel is either disabled immediately, or, if any AHB transfer is ongoing, right after that AHB transfer is complete. Once the channel is disabled, the *DMACCxControl.TransferSize* field indicates how much data was already sent to the destination peripheral. When later on you initialize the channel to resume from where it just stopped, this value determines the position from where to start. It is necessary to work with the same transfer size for both memory and peripheral in this case, and burst sizes for memory and the peripheral must be 1.

Lets have a look at an example: Assume you have a $buffer$ of 512 bytes, you need a transfer width of 8 bits, and you transfer from memory to an SSP peripheral, which is slave to an external SPI master. At some time you decide to halt the channel temporarily. When you start the DMA, it will immediately

DRAFT (full of errors)

see a burst request from the SSP. In response to that, it will do just one transfer, because burst size is 1! Right after that, there will be another burst request, and so on. The last burst request will be generated, when there are four positions left free in the SSP TX FIFO. In response to that, the DMAC will do a final transfer, and fill up the SSP TX FIFO to five frames.

- Start DMA operation with the channel source set to $buffer$. Set the *DMACCxControl.TransferSize* field to 512.
- Enable the DMA channel. Then wait until you like to halt the DMA channel operation.
- Clear the *DMACCxConfiguration.E* bit.
- Wait until *DMACCxConfiguration.A* $== 0$.
- Determine the amount of data which was already sent to the peripheral: $transfers\_done = 512 - DMACCxControl.TransferSize$
- In order to continue, do a complete reinitialization of the channel. This time, set the start pointer to $buffer + transfers\_done$, and set *DMACCxControl.TransferSize* to $512 - transfers\_done$.

Like some more confusion? As a variaton of the above theme, you may like to know the amount of frames sent via the SPI interface. What *DMACCxControl.TransferSize* tells you, is the amount of transfers done between DMAC and SSP. In general, this is different from the number of SPI frames, because you don't know, whether the five frames in the SSP TX FIFO have been sent or not! If you need to know, you must make the $buffer$ six bytes (better: transfers) larger than necessary, in our example this would be 518 bytes. Change the above example as follows:

- Start DMA operation with the channel source set to $buffer$. Set the *DMACCxControl.TransferSize* field to 512+6.
- Enable the DMA channel. Then wait until you like to halt the DMA channel operation.
- Clear the *DMACCxConfiguration.E* bit.
- Wait until *DMACCxConfiguration.A* $== 0$.
- Determine the amount of data which was already sent to the peripheral: $transfers\_done = 512 + 6 - DMACCxControl.TransferSize$
- In order to continue, do a complete reinitialization of the channel. This time, set the start pointer to $buffer + transfers\_done$, and set *DMACCxControl.TransferSize* to $512 + 6 - transfers\_done$.
- After the block transfer is over, and the DMA channel has been disabled, the followin equation yields the number of frames sent via SPI: $spi\_frames\_sent = 512+6-DMACCxControl.TransferSize-5$ This accounts for the five frames left in the TX FIFO.

1.6 Use cases

You may now want to go through a few simple examples, which will help you better understand how the GPDMA works.

1.6.1 Transmit and receive channels for SSP master

Here we will use the two DMA channels to support transmitting and receiving over one of the SSP channels. The SSP is configured as master with 8–bit frames. The simple task is now to transmit some data to the external device, then switch to receive mode, and receive a block of data.

Transmit data is assumed to be in three separate small blocks of memory. The first block contains 7 bytes (= frames) at address 0x7FD00000, the second block contains 10 bytes at address 0x7FD00200, and the last block contains 5 bytes at address 0x7FD00300. Hence we will use gather DMA with linked lists.

DRAFT (full of errors)

Transmit. As the transmit data consists of three blocks of memory, we need two linked list items. We only need two of them, because the description of the first data block is written directly into the DMA registers. Once the first data block has been serviced, the DMA will follow the list pointer to the first LLI in RAM, which holds description for the second block of data.

DRAFT (full of errors)

We create two linked list items (at addresses 0x7FD01000 and 0x7FD01010):

LLI1 (second data block)	source address	0x7FD00200
	destination address	0xE0068008 (SSP0DR)
	pointer to next LLI	0x7FD01010
	control word	0x00000000
		= 10
		+ SOURCE_BURST_4
		+ DESTINATION_BURST_4
		+ SOURCE_WIDTH_8
		+ DESTINATION_WIDTH_8
		+ SOURCE_INCREMENT
LLI2 (second data block)	source address	0x7FD00300
	destination address	0xE0068008 (SSP0DR)
	pointer to next LLI	0 (end of list)
	control word	0x00000000