

uDMA: User Manual

September 2016   
Revision 1.1  
Antonio Pullini (pullinia@iis.ee.ethz.ch)

Integrated Systems Laboratory  
ETH Zürich, Switzerland

Copyright (C) 2015-2016, Integrated Systems Laboratory, ETH Zürich, Switzerland

This document is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.

Document Revisions

| **Rev.** | **Date** | **Author** | **Description** |
| --- | --- | --- | --- |
| 0.1 | 31/05/16 | Antonio Pullini | First Draft |
| 0.2 | 23/09/16 | Antonio Pullini | Added new features present in GAP8 |

Table of Contents

[1 INTRODUCTION 6](#_Toc444245332)

[2 ARCHITECTURE 6](#_Toc444245333)

[2.1 RX Channels 8](#_Toc444245334)

[2.2 TX Channels 9](#_Toc444245335)

[2.3 L2 memory interface 10](#_Toc444245336)

[2.4 Configuration Interface 10](#_Toc444245337)

[3 API 10](#_Toc444245338)

[3.1 Overall register map 10](#_Toc444245339)

[3.2 uDMA channel register 12](#_Toc444245340)

[3.3 SPI Master 13](#_Toc444245341)

[3.4 UART 17](#_Toc444245342)

[3.5 I2S 18](#_Toc444245343)

[3.6 I2C 20](#_Toc444245344)

# INTRODUCTION

Today IoT(Internet Of Things) devices works most of the time in energy constrained environments in which small batteries has to last long times. The most widely used technique to achieve this is duty cycling in which computation is alternated to long deep sleep states. Often CPU is turned on not to do processing of data but just to handle I/O to/from peripherals. To reduce the CPU time used to control I/O subsystem we propose the uDMA, a smart, lightweight and completely autonomous DMA capable of handling complex I/O schemes.

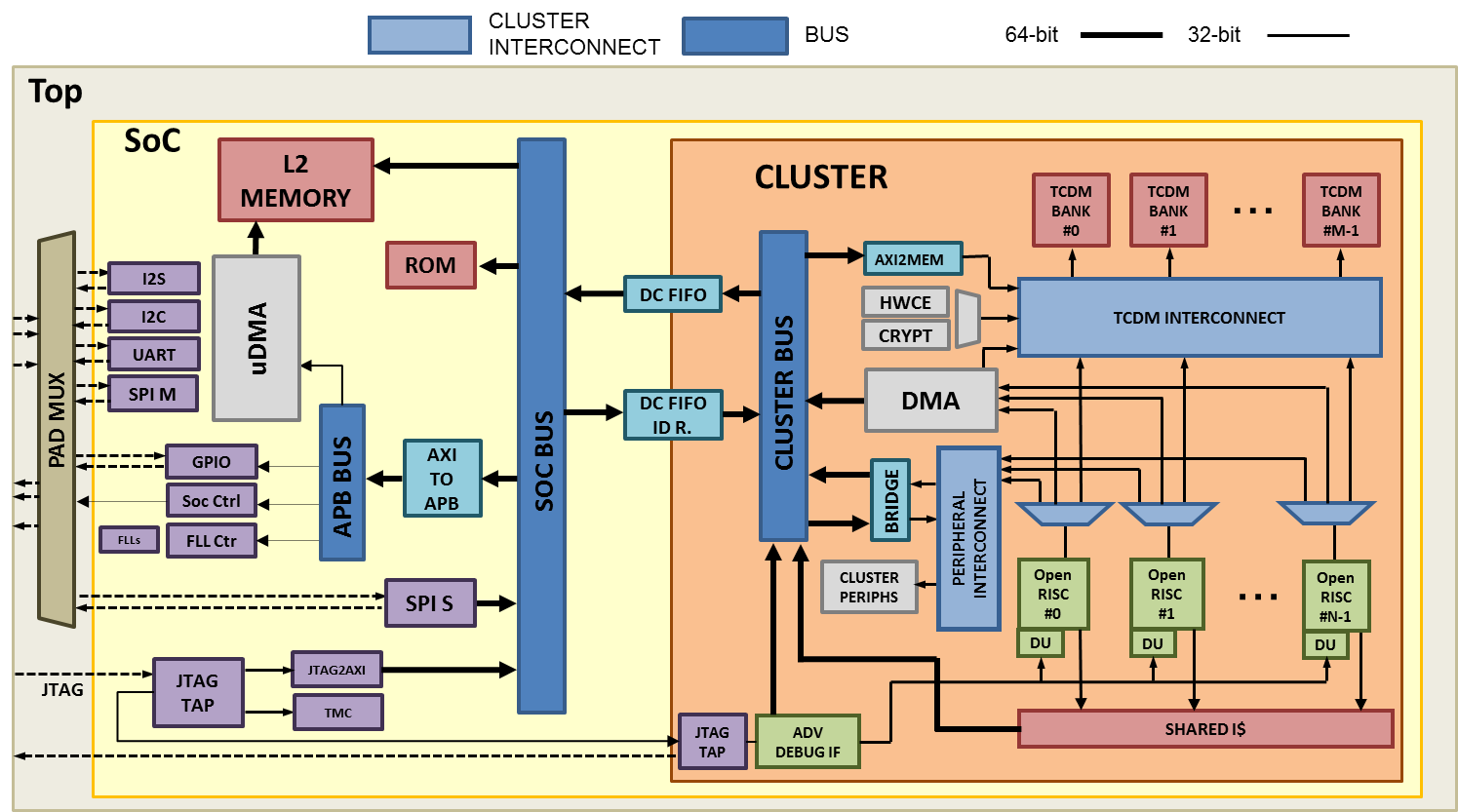


Figure : SoC Architecture

# ARCHITECTURE

In this chapter, we will show how the uDMA is implemented and how it handles data transfer from/to the peripherals. In Figure 2, we can see high-level block diagram of the uDMA. The main components are the tx channels, the rx channels, the L2 interface and the configuration block. Each peripheral may have one or more tx or rx channels depending on the need. As an example, SPI will require one TX and one RX channel while I2S will need only RX channel. The interface with the peripherals implements a simple valid/ready protocol. The peripherals ride the valid signal of the rx channels when having data available and rise the ready signal of a tx channel when requesting data from the memory. Those channels are now the only way to control data flow between peripherals and uDMA. All other controls of the peripherals are done by accessing directly the configuration interface of each peripheral.

Each uDMA channel, both TX and RX, has some dedicated resources in the configuration interface and internal to the TX/RX channel logic of the uDMA itself. The hardware resources associated to each channels are:

1. 32 bit address register storing the address of the following transfer
2. 16 bit bytes left register storing the number of bytes left in the current transfer
3. 2bit data size register keeping the amount of bytes to be transferred at each uDMA transfer. Possible values are 1,2,4

The address register stored in the configuration interface is the starting address of the next transfer while the one inside the uDMA contains the current pointer to the L2 buffer. The 16 bit size register sets the transfer size in bytes for the next transfer when in the configuration interface while specifies the number of remaining bytes in the current transfer. Last, the 2 bits for the data size contain the amount of bytes to be transferred at each data transfer for the next transfer and for the current transfer. Configuration resources are write only and the current resources are read only and accessible at the same address in the uDMA configuration space. If there is no transaction, running resources are copied from the configuration interface to the internal resources as soon as the enable bit is set. If a transfer is ongoing, they are latched as soon as the current transfer is over. Those 2 sets of registers for each channel and the queueing mechanism eases the implementation of double buffering. At the first setup of the transfer the software can start the first transfer with a target address and soon after queue another transfer with a second target address. As soon as the first transfer is over the software can queue a transaction with the first target address again and process the first chunk of data while the second chunk is transferred to the second target address in the background.

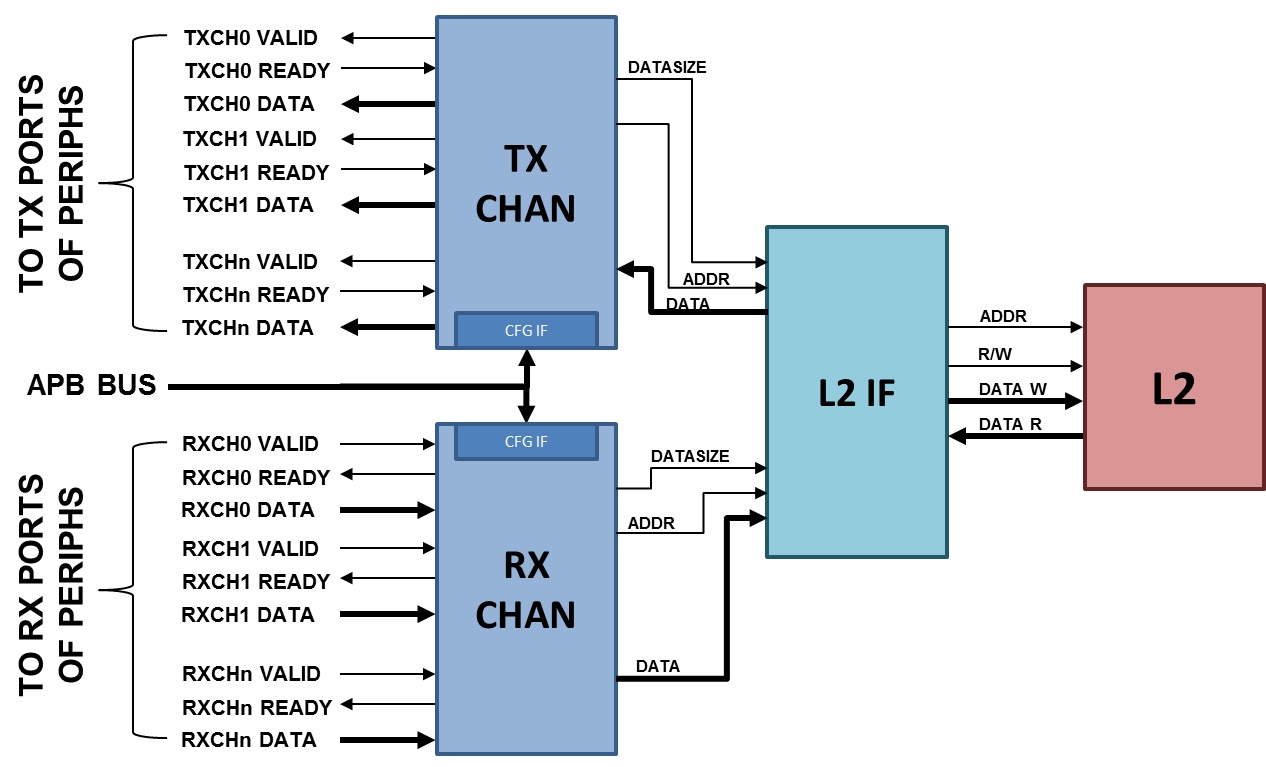


Figure : uDMA top level

## RX Channels

The RX Channels block handles the data transfer from peripherals to memory. When a peripheral has available data it raises the data valid signal of its associated channel. In the rx channels block all the valid signals, if the corresponding channels is active, are arbitrated with a fair round robin arbiter. The ID of the winning channel is used to select the data to be sent to memory and both ID and data are sampled and sent to the next pipeline stage. In the next pipeline stage the stored ID is used to select the current pointer associated with the channel, the channel data size and the number of bytes left for the transfer. Pointer, data size, and the data itself are pushed into a fifo and sent for transfer. In parallel in the same cycle the next pointer and the number of bytes left for transfers are updated. To reduce the complexity of the logic backpropagating the stalls to the peripherals, each channel, once winning an arbitration, is disabled until its data is pushed into the rx fifo. Looking at the effect on the single channels this may seems a big limitation since it cuts the bandwidth in half. In real use cases this is never an issue since all the peripherals involve some type of serial to parallel conversions and none of the available peripherals are capable of producing data at full bandwidth capable of saturating the memory bandwidth not even during peaks.

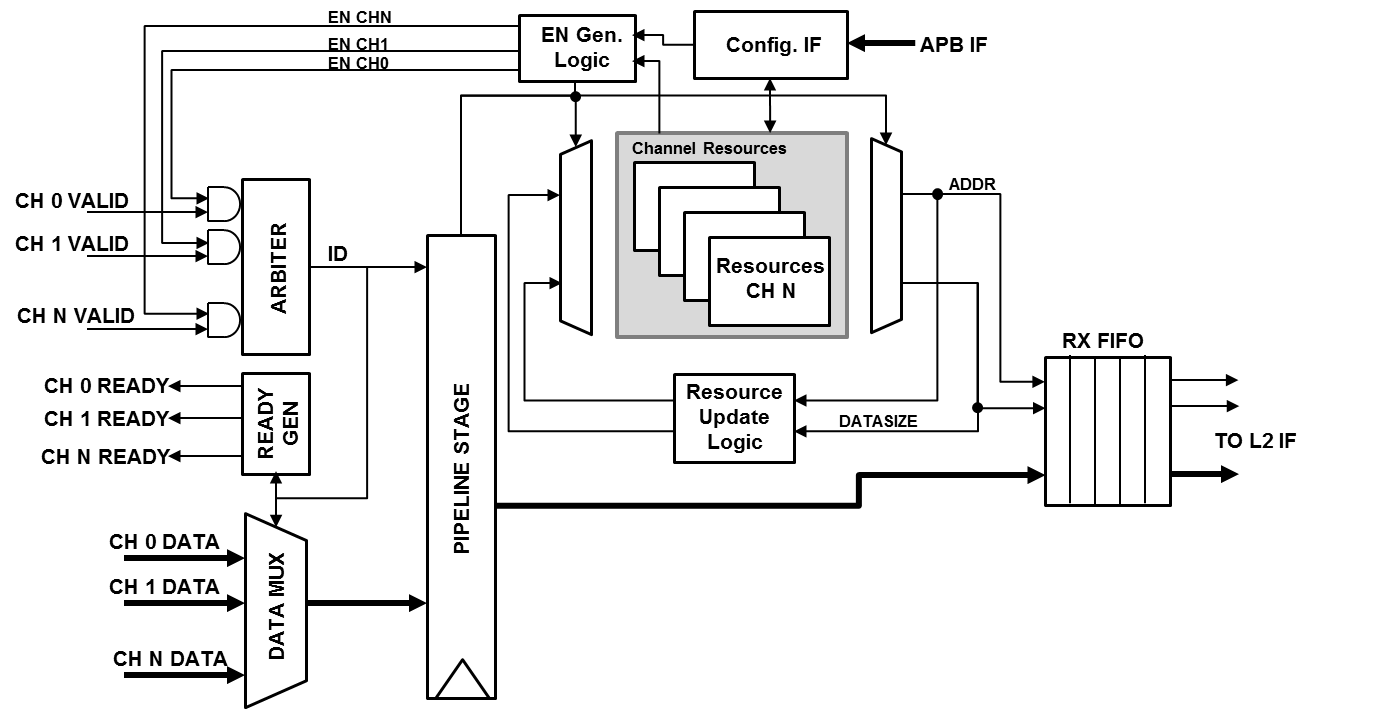


Figure : RX Channels architecture

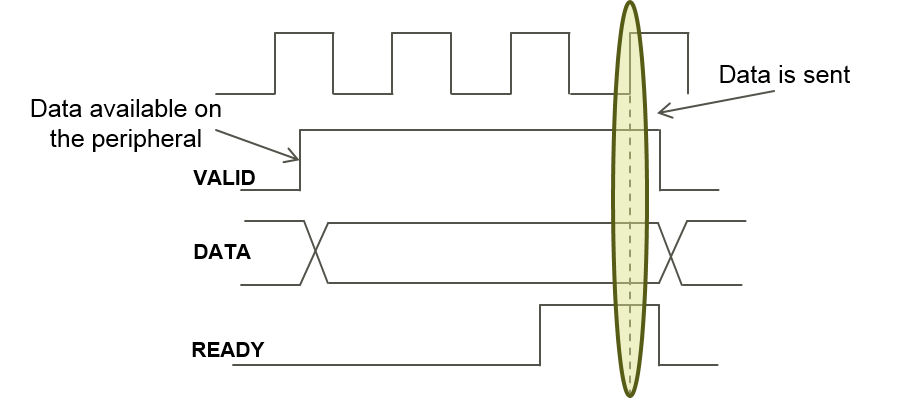


Figure : RX protocol

## TX Channels

The TX Channels block handles the data transfer in the opposite direction, from memory to peripherals. Peripherals with a TX channel, when having the pipe ready for transmission, raise the data ready signal informing the uDMA that it is now ready to accept new data. The uDMA similarly to what is done in the RX channels arbitrates the data requests coming from the active channels and latches the result of the arbitration(channel ID) to be used in the stage. During the next pipe stage the ID of the winner is used to select the corresponding resources and the pointer, datasize and channel ID are sent to the tx fifo. In parallel the new address and the number of bytes left are updated. In the tx channels also the ID is pushed in the tx fifo because the information is needed to demux the data to the proper channel on the response path. Due to the blocking nature of the simple valid/ready protocol at the uDMA boundaries each tx channel is stalled until the data is fetched from the memory. This implies that with no contention we can fetch one data every 4 cycles. This sets the maximum bandwidth of a TX channel to 1/4th of the L2 bandwidth. Once again this is not a limit in the current IO subsystem since S/P conversion absorbs the latency.

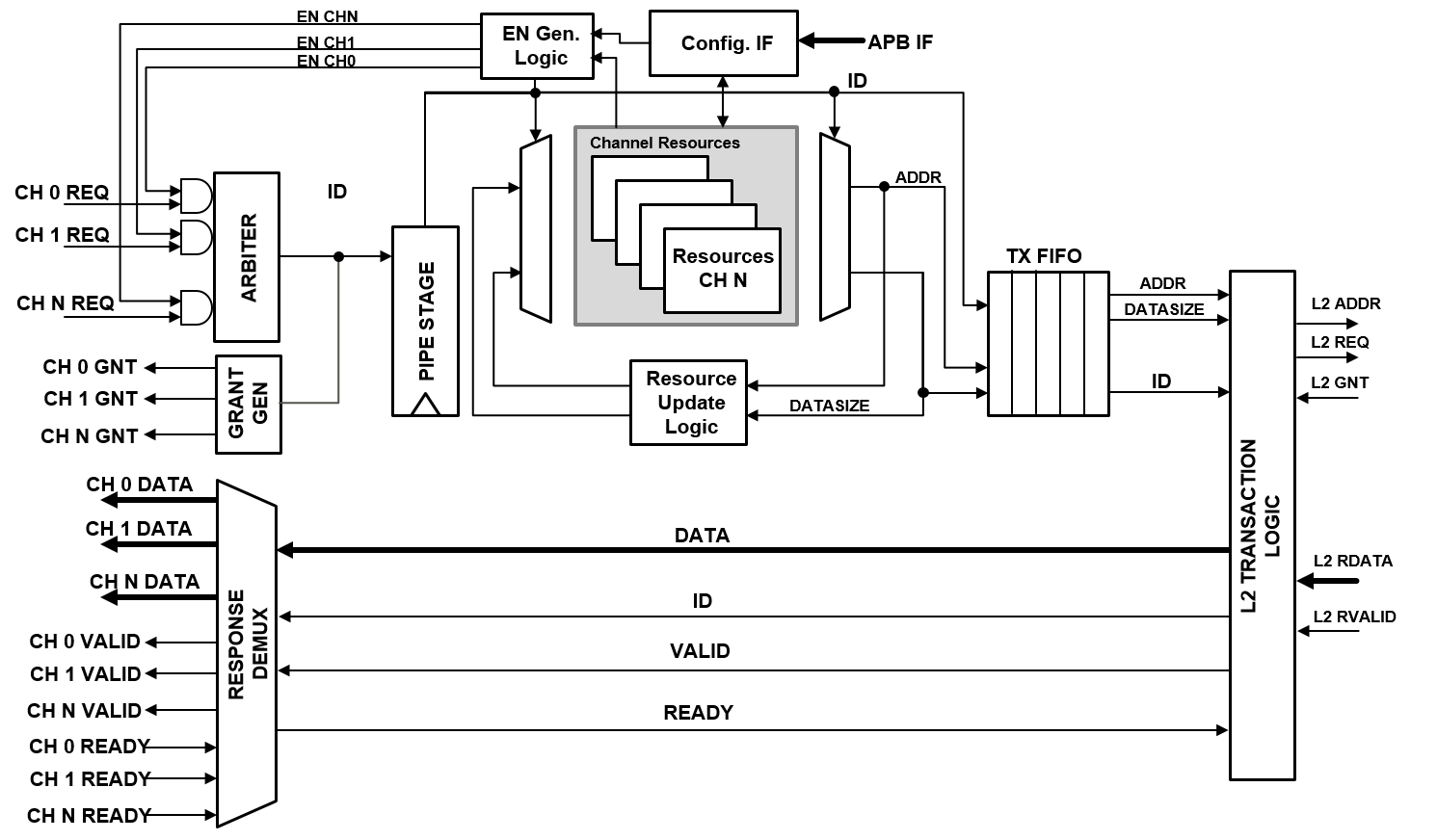


Figure : TX Channels Architecture

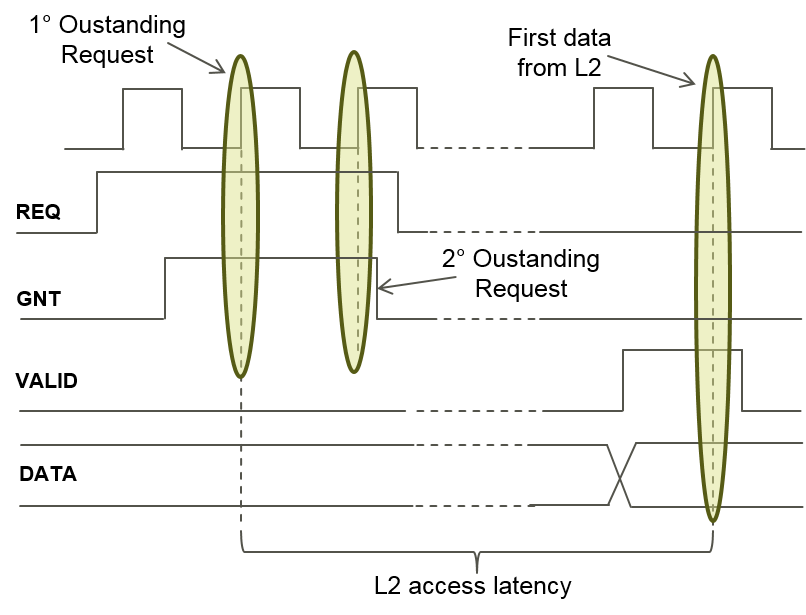


Figure : TX Protocol

## L2 memory interface

The L2 memory interface(L2if) arbitrates between the access requests originated from the TX or RX channels. Among with the arbitration the L2if generates the proper byte enables and puts the data to the proper lanes when accessing the L2. An incoming write request from the RX channels is sent to the L2 only if there is no pending read request from the TX channels or if the priority is assigned to the RX channels. The priority is flipped from TX to RX or vice versa when there is a conflict between read or write requests

## Configuration Interface

Configuration interface is connected to the system bus with an APB plug. The uDMA subsystem allows, in the current version, to have up to 15 peripherals. Each peripheral can have up to a maximum of 32 32bit configuration registers. This includes the registers needed to configure the uDMA channels dedicated to the peripheral. The uDMA address space consists in 2KB. The first 128\*15 bytes are dedicated to the 15 peripherals while the last 32 configuration registers are dedicated to the uDMA configuration

# API

In this chapter, we present the detailed description of all the registers available in the uDMA. Section presents the whole register map, Section the registers useful for configuring a uDMA transfer. Later sections present the peripheral specific registers.

## Overall register map(MR. Wolf)

| Peripheral | Offset | Description |
| --- | --- | --- |
| SPIM0 | 0x00 | SPI Master 0 |
| L2L2 | 0x80 | Memcopy device |
| HYPER | 0x100 | HyperRAM/Flash controller |
| UART | 0x180 | Serial controller |
| I2C0 | 0x200 | I2C controller |
| I2C1 | 0x280 | I2C controller |
| I2S | 0x300 | Audio controller |
| CAM | 0x380 | Camera Interface |

Table : uDMA register map

## uDMA channel register

Each uDMA channel has 4 register dedicated to its configuration. The configuration is done in the same way if the channel is either TX or RX. Each uDMA transfer is configured by setting:

1. start address (see )
2. 16 bit transfer length in bytes (see )
3. transfer type, if continuous or single and transfer size (see Table 4)

When a read is performed at the start address register the value of the current pointer is returned while a 0 if the transfer is over. Reading back from the transfer length register gives back the remaining number of bytes in the current transfer and 0 when the transfer is over.

| Bit # | R/W | Description |
| --- | --- | --- |
| 31:0 | R/W | **Address of associated buffer**  W sets the pointer for the buffer  R returns the status of the pointer. Returns 0 when transfer is over |

Table : uDMA start reg

| Bit # | R/W | Description |
| --- | --- | --- |
| 15:0 | R/W | **Size of buffer**  W sets the size of buffer in bytes(max 64KB)  R returns the number of bytes transfered. Returns 0 when transfer is over |

Table : uDMA size reg

| Bit # | R/W | Description |
| --- | --- | --- |
| 6 | R | **Transfer Pending**  When 1 there is a pending transfer in queue. |
| 5 | W | **Channel Clear**  When set to 1 stops the current transfer for the channel and disables it. |
| 4 | R/W | **Channel Enable**  When set to 1 enables the channel and starts the transfer. This signal is used also to queue a transfer if one is already ongoing. |
| 2:1 | R/W | **Channel transfer size**  Sets the channels transfer size when size is not hardwired  00: 8bits  01: 16bits  10: 32bits |
| 1 | R/W | **Continuous Mode**  When set to 1 enables the continuous mode. At the end of the buffer the uDMA reloads the address and size and starts a new transfer. |

Table : uDMA config register

## SPI Master

The Standar Peripheral Interface bus(SPI) is a synchronous serial communication interface specification used for short distance communications. The interface has been developed first by Motorola and now has become a de facto standard. The lack of a formal standard is reflected in a wide variety of protocol options. Different word sizes are common. Every device defines its own protocol, including whether it supports commands at all. Some devices are transmit-only; others are receive-only. Chip selects are sometimes active-high rather than active-low. Some protocols send the least significant bit first. The SPI master described here has some limitations to the supported variants of the SPI protocol. The major limitation is the lack of support for the full duplex transfers. The IP supports also the new QPI(Quad SPI) mode that has in the last years become a standard for the SPI memories enabling higher bandwidths required by the more and more powerful embedded devices. In Figure 7and Figure 8 there are examples of typical writes and reads to external memories using the standard 4-wire SPI protocol. In Figure 9 we may see an example transfer in QPI mode. All 4 datalines are bidirectional and the communication is always half duplex. Figure 10 shows an example of a complex SPI transfer in which an external SPI flash connected to CS1 is first configured to enable Quad SPI transfer and then the 2560bits of the whole flash ID are read using Quad SPI. Figure 11 shows an example of a repeated transfer in which 14bits are transferred from an external ADC connected to CS2.

C7

C6

C1

C0

A31

A30

A1

A0

DUMMY

Dn

Dn-1

D1

D0

nCS

SCK

SDIO0

SDIO1

Figure : Simple Write Transfer

Figure : Simple Read Transfer

C7

C6

C1

C0

A31

A30

A1

A0

DUMMY

Dn

Dn-1

D1

D0

nCS

SCK

SDIO0

SDIO1

C4

C0

A28

A24

A4

A0

Dn-3

Dn-7

D4

D0

C5

C1

A29

A25

A5

A1

Dn-2

Dn-6

D5

D1

C6

C2

A30

A26

A6

A2

Dn-1

Dn-5

D6

D2

C7

C3

A31

A27

A7

A3

Dn

Dn-4

D7

D3

nCS

SCK

SDIO0

SDIO1

SDIO2

SDIO3

Figure : Quad SPI Transfer

| Bit # | Description |
| --- | --- |
| 31:28 | **SPI\_CMD\_CFG**  4 bits selecting the command |
| 27:10 | **Reserved**  Reserved for future use |
| 9 | **CPOL**  Selects the clock polarity (see Figure 10) |
| 8 | **CPHA**  Selects the clock phase (see Figure 10) |
| 7:0 | **Clock Divider**  Sets the clock divider value. If the value is 0 the frequency is NOT divided. If clock divider > 0 then the frequency is divided by 2xClock Divider |

Table : SPI\_CMD\_CFG

| Bit # | Description |
| --- | --- |
| 31:28 | **SPI\_CMD\_SOT**  4 bits selecting the command |
| 27:2 | **Reserved**  Reserved for future use |
| 1:0 | **Chip Select**  Sets which CS signal to put to 0 |

Table : SPI\_CMD\_SOT

| Bit # | Description |
| --- | --- |
| 31:28 | **SPI\_CMD\_SEND\_CMD**  4 bits selecting the command |
| 27 | **QPI**  Sends the command using QuadSPI |
| 26:20 | **Reserved**  Reserved for future use |
| 19:16 | **Command size**  Size in bits of the command to send. SPI Master will send command size bits + 1 |
| 15:0 | **Command to send**  Sets the command to send. MSB must always be at bit15 also if cmd size is lower than 16 |

Table : SPI\_CMD\_SEND\_CMD

| Bit # | Description |
| --- | --- |
| 31:28 | **SPI\_CMD\_SEND\_ADDR**  4 bits selecting the command |
| 27 | **QPI**  Sends the command using QuadSPI |
| 26:21 | **Reserved**  Reserved for future use |
| 20:16 | **Address size**  Size in bits of the address to send. SPI Master will send address size bits + 1 |
| 15:0 | **Reserved**  Reserved for future use |

Table : SPI\_CMD\_SEND\_ADDR

| Bit # | Description |
| --- | --- |
| 31:28 | **SPI\_CMD\_DUMMY**  4 bits selecting the command |
| 27:21 | **Reserved**  Reserved for future use |
| 20:16 | **Dummy size**  Number of dummy cycles to perform |
| 15:0 | **Reserved**  Reserved for future use |

Table : SPI\_CMD\_DUMMY

| Bit # | Description |
| --- | --- |
| 31:28 | **SPI\_CMD\_WAIT**  4 bits selecting the command |
| 27:0 | **Reserved**  Reserved for future use |
| 1:0 | **Event Sel**  Selects which of the 4 available events to use during the wait |

Table : SPI\_CMD\_WAIT

| Bit # | Description |
| --- | --- |
| 31:28 | **SPI\_CMD\_TX\_DATA**  4 bits selecting the command |
| 27 | **QPI**  Sends the command using QuadSPI |
| 26 | **Disable Byte Alignment**  When Set disables the byte alignment and start shifting data into the shift register starting from bit0 |
| 25:16 | **Reserved**  Reserved for future use |
| 15:0 | **Data Size**  Number of bits to send(Max 64Kbits) |

Table : SPI\_CMD\_TX\_DATA

| Bit # | Description |
| --- | --- |
| 31:28 | **SPI\_CMD\_RX\_DATA**  4 bits selecting the command |
| 27 | **QPI**  Sends the command using QuadSPI |
| 26 | **Disable Byte Alignment**  When Set disables the byte alignment and start shifting data into the shift register starting from bit0 |
| 25:16 | **Reserved**  Reserved for future use |
| 15:0 | **Data Size**  Number of bits to receive(Max 64Kbits) |

Table : SPI\_CMD\_RX\_DATA

| Bit # | Description |
| --- | --- |
| 31:28 | **SPI\_CMD\_RX\_DATA\_CHECK**  4 bits selecting the command |
| 27 | **QPI**  Sends the command using QuadSPI |
| 26 | **Disable Byte Alignment**  When Set disables the byte alignment and start shifting data into the shift register starting from bit0 |
| 25:24 | **Check Type**  Selects the check to perform  00:Compares bit a bit  01:Compares only Ones  10:Compares only Zeros |
| 20 | **Reserved**  Reserved for future use |
| 19:16 | **Status Size**  Size of the word to read - 1 |
| 15:0 | **Data to Compare**  Maximum 16bits of data to compare |

Table : SPI\_CMD\_RX\_DATA\_CHECK

| Bit # | Description |
| --- | --- |
| 31:28 | **SPI\_CMD\_RPT**  4 bits selecting the command |
| 27:16 | **Reserved**  Reserved for future use |
| 15:0 | **Repeat Number**  Number of transfers to repeat(Max 64K) |

Table : SPI\_CMD\_RPT

| Bit # | Description |
| --- | --- |
| 31:28 | **SPI\_CMD\_EOT**  4 bits selecting the command |
| 27:1 | **Reserved**  Reserved for future use |
| 0 | **Generate EVENT**  When set to 1 generates EOT event |

Table : SPI\_CMD\_EOT

| Bit # | R/W | Description |
| --- | --- | --- |
| 6 | R | **DATA\_RX**  Set to 1 when the SPI Master is receiving data |
| 5 | R | **DATA\_TX**  Set to 1 when the SPI Master is sending data |
| 4 | R | **DUMMY**  Set to 1 when the SPI Master is receiving dummies. This state is useful when some SPI devices need some time to provide useful data |
| 3 | R | **MODE**  Set to 1 when transmitting MODE. Not implemented yet. |
| 2 | R | **ADDR**  Set to 1 when SPI Master is sending Address |
| 1 | R | **CMD**  Set to 1 when SPI Master is sending Command |
| 0 | R | **IDLE**  Set to 1 when no transfer in progress |

Table : REG\_SPIM\_STATUS read

D0

D1

D2

D3

D4

D0

D1

D2

D3

D4

CPOL=0

CPOL=1

CPHA=0

CPHA=1

| Encoding | Name | Description |
| --- | --- | --- |
| 0000 | SPI\_CMD\_CFG | Sets the configuration for the SPI Master IP |
| 0001 | SPI\_CMD\_SOT | Sets the CS |
| 0010 | SPI\_CMD\_SEND\_CMD | Transmits a configurable size command |
| 0011 | SPI\_CMD\_SEND\_ADDR | Transmits a configurable size address |
| 0100 | SPI\_CMD\_DUMMY | Receives a number of dummy bits(not sent to the rx interface) |
| 0101 | SPI\_CMD\_WAIT | Waits an external trigger to move to the next instruction |
| 0110 | SPI\_CMD\_TX\_DATA | Sends data(max 64Kbits) |
| 0111 | SPI\_CMD\_RX\_DATA | Receives data(max 64Kbits) |
| 1000 | SPI\_CMD\_RPT | Repeat the next transfer N times |
| 1001 | SPI\_CMD\_EOT | Clears the CS |
| 1010 | SPI\_CMD\_RPT\_END | Ends the repeat command |
| 1011 | SPI\_CMD\_RX\_CHECK | Checks up to 16bits of data against an expected value |
| 1100 | SPI\_CMD\_FULL\_DUPLEX | Does a full duplex transfer |



Figure : Example of multiple SPI transfers in a single command queue



Figure :Example repeated transfer

## UART

The UART(Universal Asynchronous Receiver/Transmitter) IP implements the asynchronous serial transmitter and receiver. It supports the most standard options for UART TX/RX, it has a configurable number of data bits from 5 to 8, a configurable number of stop bits(1 or 2) and can add a parity bit at the end of data bits in case of a transmission or check the parity bit in case of a reception.

UART IP has only on configuration register to configure both the transmit and receive blocks. TX and RX channels are fully decoupled and support full duplex communication.

Baud rate is not changed when changing the clock frequency and needs to be configured by sw at each frequency change.

shows the configuration register with its options. In the current version of IP no error handling is implemented.

| Bit # | R/W | Description |
| --- | --- | --- |
| 31:16 | R/W | **CLOCK DIVIDER**  Sets the clock divider ratio for the baud rate generator. |
| 8 | R/W | **TX ENABLE**  When set to 1 enables the TX logic |
| 7 | R/W | **RX ENABLE**  When set to 1 enables the RX logic |
| 3 | R/W | **STOP BITS**  When set to 1 generates to stop bits. 1 stop bit only when 0 |
| 2:1 | R/W | **BITS NUMBER**  00: send 5 bits  01: send 6 bits  10: send 7 bits  11: send 8 bits |
| 0 | R/W | **PARITY ENABLE**  When set enables the parity generation and check. |

Table : REG\_UART\_SETUP

## I2S

I²S, also known as Inter-IC Sound, Integrated Interchip Sound, or IIS, is an electrical serial bus interface standard used for connecting digital audio devices together. It is used to communicate PCM audio data between integrated circuits in an electronic device. The I²S bus separates clock and serial data signals, resulting in a lower jitter than is typical of communications systems that recover the clock from the data stream. Despite the name, it is unrelated to the bidirectional I²C bus.

R1

R0

Ln

Ln-1

L1

L0

Rn

SCK

WS

SD

Figure : Standard I2S Transfer

The protocol uses 3 wires SCK(clock), WS(word select) and SD(data). Data is always sampled on the rising edge of the clock. WS tells to which channel the following data is part of. While SD flow always from master to slave SCK and WS can be generated both by master or both by slave. The current I²S IP supports both modes as well as other 2 non standard transfer modes.

D0

D1

D2

D3

D4

D5

D6

D7

D8

D9

D10

D11

SCK

SD

Figure : DDR Transfer

| Bit # | R/W | Description |
| --- | --- | --- |
| 7:6 | R/W | **CH3 mode select**  00: Use clkgen0 (clock and WS generated by clkgen)  01: Use clkgen1 (clock and WS generated by clkgen)  10: External clock but internally generated WS  11: External clock and external WS |
| 5:4 | R/W | **CH2 mode select**  selects the mode of ch2. See ch3 for info. |
| 3:2 | R/W | **CH1 mode select**  selects the mode of ch2. See ch3 for info. |
| 1:0 | R/W | **CH0 mode select**  selects the mode of ch2. See ch3 for info. |

Table : REG\_I2S\_CHMODE

| Bit # | R/W | Description |
| --- | --- | --- |
| 19 | R/W | **LSB First CH3**  Enables LSB shifting for CH3 |
| 18 | R/W | **LSB First CH2**  Enables LSB shifting for CH2 |
| 17 | R/W | **LSB First CH1**  Enables LSB shifting for CH1 |
| 16 | R/W | **LSB First CH1**  Enables LSB shifting for CH1 |
| 3 | R/W | **DDR Enable CH3**  Enables Double Data Rate(Sampling on both edges) for CH3. |
| 2 | R/W | **DDR Enable CH2**  Enables Double Data Rate(Sampling on both edges) for CH2. |
| 1 | R/W | **DDR Enable CH1**  Enables Double Data Rate(Sampling on both edges) for CH1. |
| 0 | R/W | **DDR Enable CH0**  Enables Double Data Rate(Sampling on both edges) for CH0. |

Table : REG\_I2S\_USEDDR

| Bit # | R/W | Description |
| --- | --- | --- |
| 19 | R/W | **SNAP Enable CH3**  Enables SNAP CAM mode for CH3 |
| 18 | R/W | **SNAP Enable CH2**  Enables SNAP CAM mode for CH2 |
| 17 | R/W | **SNAP Enable CH1**  Enables SNAP CAM mode for CH1 |
| 16 | R/W | **SNAP Enable CH0**  Enables SNAP CAM mode for CH0 |
| 4:0 | R/W | **EXT WS**  When using EXT Clock and internal WS selects after how many bits the WS is toggled. The value written here is num bits – 1. |

Table : REG\_I2S\_EXT\_SETUP

| Bit # | R/W | Description |
| --- | --- | --- |
| 31:16 | R/W | **CLKGEN0 CLKDIV**  Sets the clock divider ratio for clockgen0 |
| 5:4 | R/W | **CLKGEN0 ENABLE**  When set enables the clkgen0 clock and ws generator. |
| 4:0 | R/W | **CLKGEN0 WS**  Selects after how many bits the WS is toggled. The value written here is num bits – 1. |

Table : REG\_I2S\_CFG0\_SETUP

| Bit # | R/W | Description |
| --- | --- | --- |
| 31:16 | R/W | **CLKGEN1 CLKDIV**  Sets the clock divider ratio for clockgen1 |
| 5:4 | R/W | **CLKGEN1 ENABLE**  When set enables the clkgen1 clock and ws generator. |
| 4:0 | R/W | **CLKGEN1 WS**  Selects after how many bits the WS is toggled. The value written here is num bits – 1. |

Table : REG\_I2S\_CFG1\_SETUP

## I2C

I²C (Inter-Integrated Circuit), is a multi-master, multi-slave, single-ended, serial computer bus invented by Philips Semiconductor (now NXP Semiconductors). It is typically used for attaching lower-speed peripheral ICs to processors and microcontrollers.

I²C uses only two bidirectional open-drain lines, Serial Data Line (SDA) and Serial Clock Line (SCL), pulled up with resistors.

The I²C reference design has a 7-bit or a 10-bit (depending on the device used) address space. Common I²C bus speeds are the 100 kbit/s standard mode and the 10 kbit/s low-speed mode, but arbitrarily low clock frequencies are also allowed. Recent revisions of I²C can host more nodes and run at faster speeds (400 kbit/s Fast mode, 1 Mbit/s Fast mode plus or Fm+, and 3.4 Mbit/s High Speed mode). In our current implementation we implement the 100Khz,400KHz and 1Mbit/s Fast mode.

The maximum number of nodes is limited by the address space, and also by the total bus capacitance of 400 pF, which restricts practical communication distances to a few meters.

I²C defines basic types of messages, each of which begins with a START and ends with a STOP:

* Single message where a master writes data to a slave;
* Single message where a master reads data from a slave;
* Combined messages, where a master issues at least two reads and/or writes to one or more slaves.

All I²C transfers could be splitted in a reduced number of bus accesses types, those are:

* Start Bit
* Send Byte and get acknowledge
* Get Byte and send acknowledge
* Get Byte and send not acknowledge
* Stop Bit

With different combinations of the above, we can create any type of I²C transfer. Under those conditions we decided to change the interface of the I²C IP and have it fetch command from L2 memory instead of only data. In this way we can recreate complex I²C transfer fully autonomously and without any intervention of the CPU. Figure 10 shows an example of a more complex I²C transfer that generates a write on a 24LC1024 EEProm. A list of the available commands and their encoding is shown in Table 19. The command sequence starts by generating a start bit on the bus followed by a byte write and waiting for the slave acknoledge. The first byte, following the I²C standard sends the 7bit address with the last bit coding the access type(0 for write 1 for read) so in this case 0x52 is the address and access is a write. The following two writes are the internal address of the EEProm (0x0000). The following instructions tell the I²C IP to repeat the next instructions 16 times. The instruction to be repeated is the write and the data for each write instruction is queued. Here we do write 16 bytes 0x00, 0x01…0x0F. The I2C\_CMD\_STOP generates the stop bits and ends the transfer. I2C\_CMD\_WAIT waits some I2C cycles (in this case 16) and the following I2C\_CMD\_START restart a new I²C transfer. The start is followed by the address of the peripheral and then by the internal address. I2C\_CMD\_START generates a restart condition needed by the EEPRom and then sends the 7bit address but this time with a read flag(0xA5). The next command says to read 15 bytes and sends acknowledge at each byte and then read the last byte followed by a not acknoledge to inform the slave that we are done with the transfer. A stop bit then finalizes the transfer. All the commands are read through the TX port while each read pushes data to the RX channel.



Figure : Example of a complex I2C command string

| Bit # | R/W | Description |
| --- | --- | --- |
| 31:16 | R/W | **I2C CLOCK DIVIDER**  Sets the clock divider ratio for the I2C |
| 8 | W | **I2C ENABLE**  Enables the I2C peripheral. |

Table : REG\_I2C\_SETUP

| Encoding | Name | Description |
| --- | --- | --- |
| 00000000 | I2C\_CMD\_START | Does a start bit on the I2C bus |
| 00000001 | I2C\_CMD\_STOP | Does a stop bit on the I2C bus |
| 00000010 | I2C\_CMD\_RD\_ACK | Receives 1 byte and sends an acknowledge |
| 00000011 | I2C\_CMD\_RD\_NACK | Receives 1 byte and sends a not acknowledge |
| 00000100 | I2C\_CMD\_WR | Sends one byte and waits for the acknowledge |
| 00000101 | I2C\_CMD\_WAIT | The following byte tells how many I2C cycles to wait |
| 00000110 | I2C\_CMD\_RPT | The following byte tells how many time to repeat the next instruction |

Table : I2C commands

## CAMERA

Parallel camera interface is widely used among low power, low resolution sensors. The camera interface included in the uDMA supports a wide set of input formats.

| Register Name | Offset | Description |
| --- | --- | --- |
| REG\_CAM\_SADDR | 0x00 | Address of CAM IP RX buffer |
| REG\_CAM\_SIZE | 0x04 | Size in bytes of CAM IP RX buffer |
| REG\_CAM\_CFG | 0x08 | Configuration register CAM IP RX channel |
| REG\_RX\_INTCFG | 0x0C | Reserved |
| REG\_CAM\_CFG\_GLOB | 0x20 | Globl settings |
| REG\_CAM\_CFG\_LL | 0x24 | Settings of lower left corner for image croping |
| REG\_CAM\_CFG\_UR | 0x28 | Settings of upper right corner for image croping |
| REG\_CAM\_CFG\_SIZE | 0x2C | Size of image |
| REG\_CAM\_CFG\_FILTER | 0x30 | Linear Filter configuration |

Table Camera IP Register Map

| Bit # | R/W | Description |
| --- | --- | --- |
| 31 | R/W | **Camera Enable**  When set enables frame reception |
| 30:16 | R/W | **Reserved** |
| 15:14 | R/W | **Pixel Size**  00: 8bits  01: 16bits |
| 13:10 | R/W | **Shift**  Performs a right shift on the result of the linear composition |
| 9:8 | R/W | **Input format**  00: RGB565  01: RGB555  10: RGB444 |
| 7 | R/W | **Frame Slice En**  Enables cropping of frame. |
| 6:1 | R/W | **Frame Drop Value**  Selects how many frame to drop when frame drop enabled |
| 0 | R/W | **Frame Drop EN**  Enables frame dropping |

Table Register REG\_CAM\_CFG\_GLOB