MOT-SFS-IAPI-SAS-001S-ZFR11
2.0
April 14, 2006

I.API Software Architecture Specification

APPROVED:

Jean Casteres	**Gregory Meunier**	

REVISION HISTORY

Version	Date	Name	Reason
0.01	January 1999	Jean Casteres	Initial Frame maker document
0.02	May 2003	Jean Casteres	Updated with PCS documents and Rainbow family product inputs
0.03	March 2003	Gregory Meunier	Add new function (iapi_ConnectChannelScript) and modification of the channel descriptor structure.
0.04	April 2004	Gregory Meunier	Update Buffer Descriptor structure to mention Last buffer descriptor bit.
0.05	October 2004	Nicuşor Penişoara	Aligned to I.API label 02.02
0.06	October 2004	Gregory Meunier	Add more details on Wrap/Cont bit
1.0	October 2004	Gregory Meunier	Published version for the SDMA_SR01
1.1	December 2004	Nicuşor Penişoara	Added details about OS support in I.API, new initialization function, control codes and inclusion of the IRQ handler inside I.API.
1.2	January 2005	Gregory Meunier	I.API only supports extended buffer descriptors. Iapi_Inflate/deflate functions are removed.
1.3	1 Dec 2005	Jean-Pascal Blondin	New config_data structure passed to the iapi_Init function to initialize all SDMA usefull CONFIG register bits
2.0	14 April 2006	Jean-Pascal Blondin	Update for merchant market customers

CONTRIBUTORS

Ownership and maintenance of this document belongs to the SPS Softsim Team. The primary contact for this document is Jean Casteres.

The following engineers have contributed to the creation of this document through authoring, editing, answering questions, participating in information gathering meetings, and/or reviewing documentation. All input into the creation of this document is greatly appreciated.

Jean Casteres, European System and Architecture Center, Software System Simulation E3S (Softsim Team).

Gregory Meunier, EMEA WMSG, Toulouse

Nicuşor Penişoara, EMEA WMSG, Bucharest

Jean-Pascal Blondin, EMEA WMSG, Toulouse

TABLE OF CONTENTS

List of figures

List of Tables

1 Introduction

1.1 Audience Description

This document is intended for architect engineer in software and hardware and for software engineers working in architecture, design, implementation and test.

1.2 Purpose Statement

The purpose of this document is to present the Inter-Processor Communication Module Application Programming Interface (I.API). The Software Architecture Specification presents the architecture of the API, the hierarchy between the function of the API, the data structures used by the API and all functions presented at a high level.

This document is a Software Architecture Specification; it is preceded by the Software Requirement Document and followed by the Software Design Document. Please refer to [8] for New Product Introduction specifics and relation to [12].

1.3 References

[1] "Inter-Processor Communication Module: Programming model Specification", Brian Lucas and Sheila Rader, Corporate Software Center, October 16, 1998.
[2] “Efficient Interrupt Processing on the M.Core architecture” White Paper, John Vaglica, M.Core Technology Center, 1998
[3] “IPCM Application Programming Interface”, Jean Casteres, Revision 0.1, June 1, 1999
[4] “Rainbow IC Specification, Freescale Complete Multi-Mode Baseband Integration Development Platform”, Revision 1.8, December 2000.
[5] “IPCM Core Software Requirements Specification”, Robert Adair, Version 2.0, October 2001.
[6] “IPCM Core Interface Specification”, Robert Adair, Version 2.0, October 2001.
[7] “I,API Software Architecture Specification”, Jean Casteres, Version 0.2, May 2003
[8] “I.API Software Design Specification”, Jean Casteres, Version 0.2, May 2003
[9] “I.API Software Quality Specification”, Jean Casteres, Version 0.2, June 2003.
[10] “I.API Software Requirements Specification”, Jean Casteres, Version 0.2, May 2003.
[11] “I.API Software Test Specification”, Jean Casteres, Version 0.2, June 2003.
[12] “New Product Introduction Software Development Process Manual”, document number 68MSA10102D, issue B

1.4 Conventions

Bold indicates emphasis abbreviation or acronym defined in the section below.

Underline indicates a paragraph to come later in the document, which is presented or defined at this point.

References in brackets are hyperlink that brings back to the reference section of this document.

1.5 Definitions, Acronyms, and Abbreviations

AP Application Processor: in charge of the application layer software and interaction with the user.

API Application Programming Interface

Dedicated Dedicated processor is the micro-processor connected to the SDMA that is intended to perform a particular, or dedicated, class of software (i.e.: DSP or applications depending on the platform).

E3S European Systems and Architecture Center (ESAC) Software Systems Simulation team

GSM Groupe Special Mobile

I.API Inter Processor Communication Module Application Programming Interface

MOOSE Freeescale Object Oriented Simulation Environment

MP Modem Processor in charge of the wireless telephony modem software (engine) to provide wireless network access

MSB Most significant byte

msb most significant bit

rainbow Silicon first generation base-band System on a Chip (SoC) for third generation wireless protocol 3G products.

SDMA Smart Direct Memory Access module

Sfs Softsim 3-letters abbreviation

ZATS **Zeus Argon Tortola SCM-A11. The SDMA is included in these 4 ICs.**

1.6 Overview

This document is the architecture document for the Inter-Processor Communication Module (IPCM) Application Programming Interface: I.API. The goal of the API is to propose a set of calls that ease the access and use of the smart direct memory access SDMA (formerly IPCM) virtual DMA channels from both the host processor and dedicated core perspective.

The library is written in C and compiled on the host processor and dedicated core using their respective compilers in order to guarantee functionality and integrity in the usage of SDMA virtual DMA channels.

The Inter-Processor Communication Module has been introduced by Brian Lucas and Sheila Rader's teams in [1]. The first simulator was developed by Brian Lucas' team in 'C' with an M.Core-SDMA to M.Core data transfer. This architecture has been maintained and modified by the SDMA team and Softsim team in the Toulouse Wireless Design Center (TWDC).

The M.Core interrupt handler methods used in this document directly derives from [2]. This document is a full re-work of the original I.API and the PCS API detailed in [5].

This architecture document first introduces the external programming model for the SDMA visible to the programmer, continues with a presentation of possible synchronization techniques for the virtual DMA channels, to finish with a presentation of the library architecture of the application programming interface.

1.7 Document Location

http://compass.freescale.net/doc/151006401/IAPI-SAS.doc

1.8 Problem Reporting Instructions

Problems or corrections to this document should be reported by email to the Author.

2 SDMA Programming Model presentation

The programming model exposes the interface visible to the programmer that enables to program and use the SDMA DMA channels. The terminology used with the SDMA is introduced first, to continue with the presentation of the data structures defined and used by the SDMA on the host processor and the dedicated processor sides. The onboard SDMA memory layout and channel ownership and activation are presented last.

2.1 Terminology

Channel: is a unidirectional communication path established by the SDMA between a source and a destination (or sink). It is composed of a script that is triggered by one or more hardware events received by the SDMA. There is a maximum of 32 hardware events wired to the SDMA and there is a maximum of 32 channels operating simultaneously on the SDMA at any one time (v100 - v300 SDMA families).

Context switch: is a change of flow triggered by an external event. The active channel (or from IDLE mode) is replaced by a higher priority channel after saving SDMA registers and current channel information to an area of its internal private SRam memory space.

Dedicated: Dedicated processor is the micro-processor connected to the SDMA that is intended to perform a particular, or dedicated, class of software. In the case of Rainbow family platforms, the dedicated processor is the digital signal processor DSP (Star*Core sc140), whereas in the Argon family platforms, the companion application processor is the dedicated processor and the Star*Core P2002 the host processor.

Event: asynchronous external hardware signal that can cause a change of the SDMA active channel, therefore triggering a context switch. The hardware signal is wired to one of the 32 SDMA event entries.

Host: the M.Core for Rainbow, ARM for ZATS. Because the library source code is compiled for both these cores, there is no difference in the way the SDMA is communicating to them at the API level.
The MCU (i.e.: ARM or M.Core) is called the master host or “the host” because, on Rainbow, the SDMA external programming model on the host processor side is an extended one: it provides the host processor programmer with the capability to configure the channel configuration properties (see below).
This is platform dependant since for future platforms, the host is the P2002 platform and the client processor is the application processor platform (i.e.: “Zeus” Arm 1136 based).

Scripts: the SDMA is programmed in assembler using syntax similar to that of a reduced M.Core instruction set. The programs that the SDMA runs are called scripts.

Local memory: refers to the memory attached to the core currently referenced. SDMA local memory is the private on-chip memory attached to the SDMA core.

2.2 SDMA Communication scheme

2.2.1 Introduction to SDMA transfers

In the Rainbow architecture, the SDMA core communicates with:

- Host processor eDRam
- Dedicated processor on-chip SRam.
- SDMA own peripherals

A simplified figure shows these basic communication paths

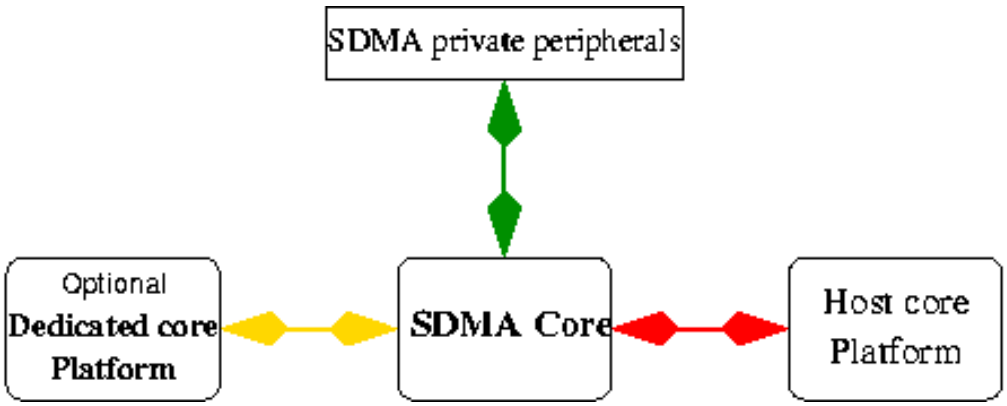


Figure 1: *SDMA simple communication paths.*

The communication of the SDMA with the host and dedicated cores involves going to or from the ***memory*** of the host and/or dedicated cores. The communication, when established, acts like a virtual DMA channel and is supported with special memory structures defined for the virtual DMA channel in the host processor eDRam and the dedicated core SRam respectively. The SDMA on board memory is used for the needs of the scripts. At the time of writing, there are 4 kilobytes of ROM and 8 kilobytes of SRAM. These sizes are given as an example based on the SCM-A11 architecture. Memory sizes are platform dependant.
For V300 and on versions of the SDMA a 'C' compiler is also available in order to compile or port additional programs such as device drivers for the peripherals.

The communication to the peripherals is done with the SDMA data memory and peripherals DM Bus. The communication with each peripheral is detailed in the peripheral programming models.

The data transfer utilizing the SDMA to and from the host and dedicated core is supported by buffer memory structures on both sides. Communicating with the SDMA is about reading and writing buffers in the dedicated processor on-chip SRam and host eDRam. This is achieved by using the supporting memory structures explained in details in the following paragraphs.

2.3 Memory buffer structures for connected processors

For each channel, the supporting memory structures in the dedicated processor are defined by:

- The Channel Control Blocks (CCB): one for each channel.
- The Buffer Descriptor array (BD): a buffer descriptor points to the "real" data buffer of the data to be transferred from source to sink and defines its properties and state.
- The SDMA supports of channel memory structures on each side through a reference to two pointers: the host Channel Zero Pointer HostC0Ptr and the dedicated processorC0Ptr respectively. Both point to the first channel control blocks that is the channel zero control blocks for the respective sides.

The figure below presents a graphical view of the supporting memory structure on the host eDram side and the dedicated processor on-chip SRam side respectively. The real data to transfer is contained in the buffer pointed to by the buffer descriptor: the size of this buffer can vary and is reflected by the count field in the buffer descriptor.

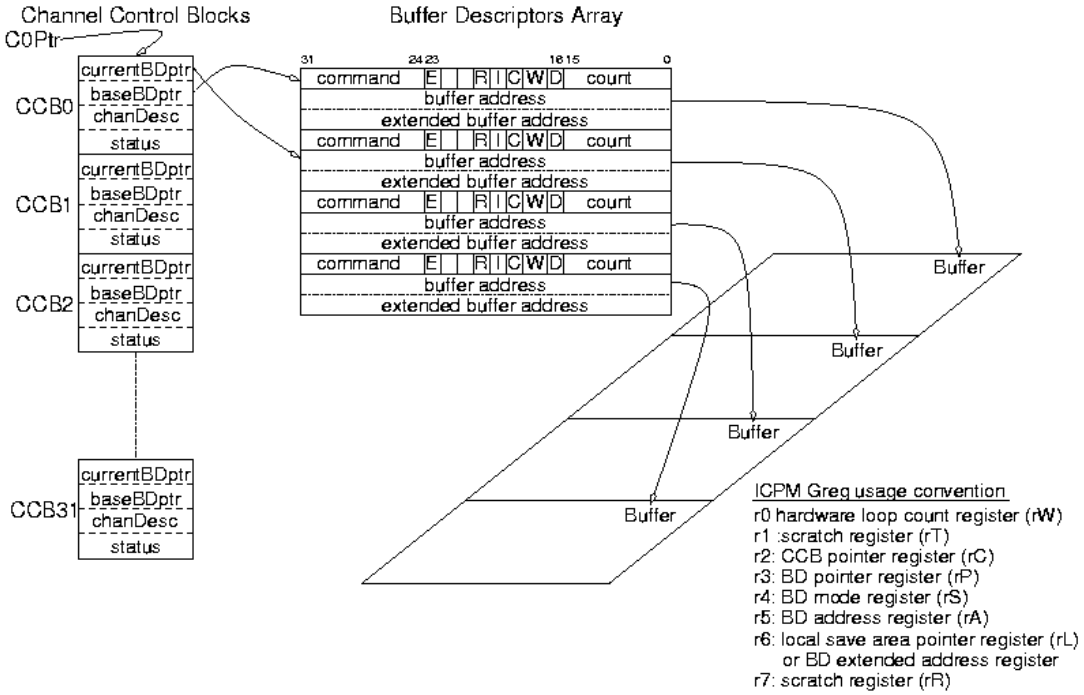


Figure 2: *Supporting Memory structure layout.*

It is to note that at the design time of the SDMA and its supporting memory structures the channel control block zero pointer (C0Ptr) and the channel control block table are meant to remain STATIC for the duration of the SDMA operation.

The SDMA General register (Greg) usage convention presents in the schema the usage convention for the SDMA general purpose registers. This convention is important for the understanding of the SDMA scripts and library calls presented in the following paragraphs.

2.3.1 Channel control block:

Channel control blocks are four (4) longs (32-bit) big in size. All of the 32 channel control blocks are meant to be defined in memory even if the corresponding channel will not be used.

The Channel Control Block (CCB) is an array of 64 pointers having two pointers per SDMA channel. There can be more than one buffer descriptor for a given application: the channel control block holds the base buffer descriptor and the currently used one. If an application has only one buffer descriptor, the current and base buffer descriptor will always be the same.

The meaning of each long in the channel control block structure is:

Status: the I.API uses this register to store status information about the channel, see the SDMA API variable table.

Base buffer descriptor pointer: holds an address pointing to the first buffer descriptor for the channel.

Current buffer descriptor pointer: holds an address pointing to the current buffer descriptor (usually also store in the rP register of the SDMA).

Channel Descriptor Pointer: holds the address pointing to the channel's associated descriptor structure. The channel descriptor data structure holds the properties and characteristic parameters of the channel (i.e.: number of buffer descriptors, buffer sizes, etc…).

Channel control blocks are defined contiguously in memory, this is important since the script onboard the SDMA awaits such continuity when accessing channel control blocks (ROM script accesses CCBs with increments of a fixed size).

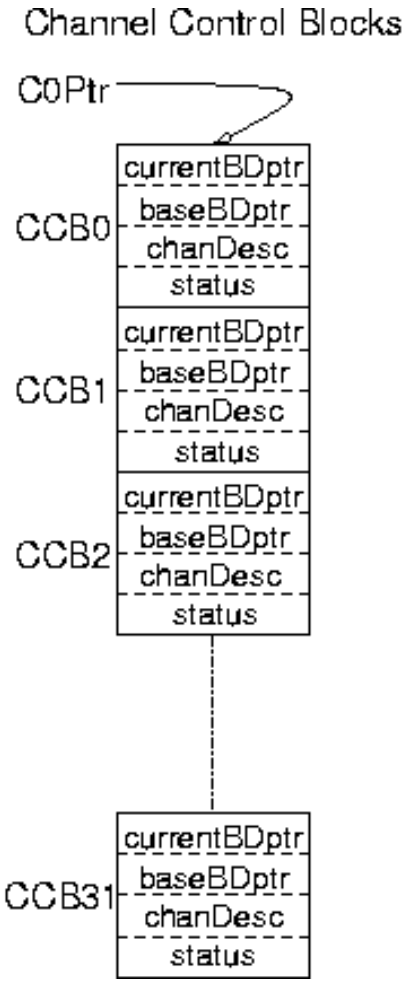


Figure 3: *Channel control blocks*

Consequently the structure used for the API is:

```
typedef struct iapi_ChannelControlBlock {
               bufferDescriptor *    currentBDptr;
               bufferDescriptor *    baseBDptr;
               channelDescriptor *   channelDescriptor;
```

```
                channelStatus *       status;
} channelControlBlock;
```

2.3.2 Buffer descriptor and channel zero commands

Buffer descriptor are three (3) longs (32-bit) big in size. A buffer descriptor describes the properties of the data buffer it points to.

The host and dedicated processor software use the same buffer descriptor structure. The overall field structure of the buffer descriptor should be kept and is defined in include files for both host dedicated processor sides; but application can use these fields to indicate status that reflect the state of the particular transfer transaction that is being worked on.

The buffer descriptors can be used for linear or circular data buffers in the host or dedicated processor memory. Hence the use of current and base buffer descriptor pointers

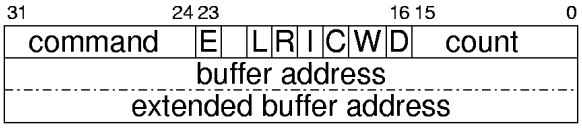


Figure 4: *Buffer Descriptor*

The properties are classified in three different fields in the first long of the buffer descriptor, this first field is usually named “mode”. The second long is the address pointer to the data buffer. The meaning of each field of the buffer descriptor structure is:

Command: the command field is used to differentiate operations performed within a script when the script accesses this particular buffer descriptor. The use of this field can be defined on a per channel basis.

For the channel zero which is reserved for host processor to SDMA communications there is a set of pre-defined commands, handled by the corresponding script on the SDMA:

- C0_SETPM : 0x04, C0_SETDM: 0x01: load the buffer descriptor data in the SDMA local memory at the address pointed to by the “extended buffer address” field. The counter in the buffer descriptor is assumed to be in bytes so it is divided by 4 as 32-bit long word accesses are performed. This command can be used to download scripts to the SDMA memory.
- C0_GETPM: 0x08, C0_GETDM: 0x02: write to the buffer descriptor’s data buffer the content of the SDMA local memory from the address pointed to by the “extended buffer address” field for the length defined by the count in the buffer descriptor. The distinction between PM and DM is made due to the SDMA addressing architecture: PM addresses 16-bit instructions while DM addresses 32-bit data words. Therefore the count is expressed in “longs” for the DM and in “shorts” for PM.
- C0_SETCTX: 0x07: load a context into the SDMA context page area. The handling script decodes the channel number from the first long of the buffer descriptor; using the channel number the script computes the offset of the context data pointer for the channel relative to the context page base. The local save area pointer register is updated with the newly computed value. Then the set command explained above is invoked, the counter indicates the size of the context switch structure.
- C0_GETCTX: 0x03: write to the buffer descriptor’s data buffer the content of the SDMA context page area. The handling script decodes the channel number from the first long of the buffer descriptor; using the channel number the script computes the offset of the context data pointer for the channel relative to the context page base. The local save area

pointer register is updated with the newly computed value. Then the get command explained above is invoked, the counter indicates the size of the context switch.

Note that the original script design in would call "get" commands "dump" and "set" commands load. An "address" command would load an address into the save area pointer which s no longer necessary.

NOTE: encoding of the channel number in the status/mode first long of the buffer descriptor is done in the following way:

buff_desc.mode.command = (C0_SETCTX | (channel_number << 3))

Status-bits or mode-bits: define E, L, I, C, W, R, D flags:

- E – "Extended": bit 23: direct buffer memory transfer. When this bit is set, the extended buffer address of the buffer descriptor is used: the buffer address is the source originating point of the direct memory access data transfer to the extended buffer address that points to the destination address in memory.
  Note: The IAPI only supports extended buffer descriptor. If extended address field is not needed by the SDMA script, a dummy value must be written in this field.
- L – "Last Buffer Descriptor": bit 21: this bit is set in SDMA IPC scripts to indicate to the receiving Core that the transfer should end. This is the case when the DSP prepares a single buffer descriptor with count equals to 25 and MCU prepares a single buffer descriptor with count equals 100. When 25 bytes have been transferred from DSP to MCU, the DSP buffer descriptor is normally closed while the MCU buffer descriptor will have the L bit set and the byte count updated to 25. This is in the case where the receiving side has a buffer bigger than the data being sent. The 'L' bit is set and the interrupt (whatever the value of the 'I' bit) sent to allow the receiving side to handle the data at this point.
- R – "erroR|": bit 20: indicates an error occurred on the channel's buffer descriptor requested command. The count filed has been updated at the instant the error occurred, therefore indicating the number of bytes successfully processed. The command field is overridden with and error code detailing further the source of the error.
- I – "Interrupt": bit 19: when SDMA is done processing data attached to this buffer descriptor, send an interrupt to the core (host or dedicated processor) the buffer descriptor belongs to.
- C – "Continuous": bit 18: this buffer is allowed to receive multiple transmit buffers or is allowed to transmit to multiple receive buffers.The Continous bit is decoded at the end of the processing of a BD to determine if the SDMA script must open a new BD to potentially continue the data transfer.
- W – "Wrap": bit 17: if set, this buffer descriptor is the last one for the channel control block. When encountering this bit set, the SDMA scripts updates the CurrentBD pointer to point to the first Buffer Descriptor of the array. This bit is set if the MCU or DSP wants to organize the array of BD in a circular way (like a ring). When all BD have been processed and if Wrap bit and CONtinuous bit are set in the last BD, the SDMA script will wrap around and it will try to re-open the first BD.
- D - "Done": bit 16: indicates the "ownership" of the buffer descriptor. When D=0 the host (host or dedicated processor) owns the buffer descriptor; when D=1 SDMA owns the buffer descriptor. In the case of a transfer from the host to the SDMA D=0 indicates the SDMA has not yet processed this buffer, D=1 indicates the SDMA has processed this buffer.

Count: the count field indicates the length in bytes of the data buffer pointed by the buffer descriptor, except for the situations mentioned below.

The SDMA memory structure is different for program memory (16-bit shorts) and data

memory (32-bit long). To accommodate this, when the *command* field is used (by channel 0) with the following commands, the count is expressed correspondingly:

C0_SET_PM: *count* is the buffer length in 16-bit shorts

C0_GET_PM: *count* is the buffer length in 16-bit shorts

C0_SET_DM: *count* is the buffer length in 32-bit long

C0_GET_DM: *count* is the buffer length in 32-bit long

C0_SETCTX: *count* is the buffer length in 32-bit long

C0_GETCTX: *count* is the buffer length in 32-bit long

Buffer address: the entire long (32-bit) is an address pointer to the "real" data buffer.

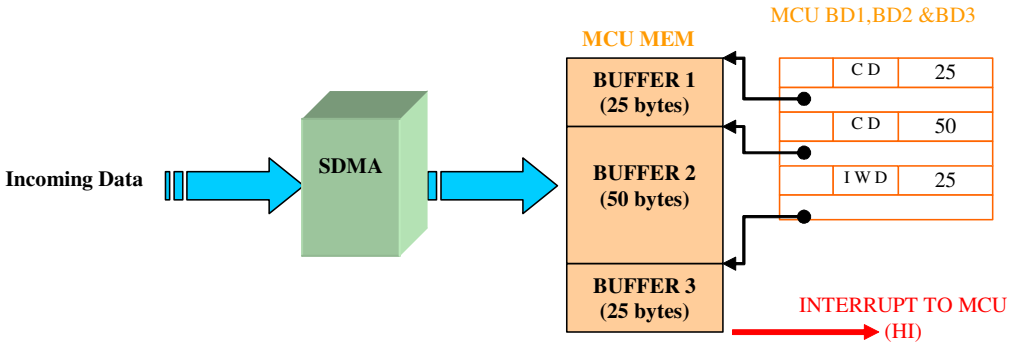
Extended buffer address: the "buffer address" field is the source originating point of the direct memory access data transfer to the "extended buffer address" field that points to the destination address in memory.

The buffer descriptors form an array of programmable size, the last buffer descriptor is marked by the Wrap flag-bit W=1. The array of buffer descriptor is treated as a ring, with some logically continuous portion owned by the host D=0, and the remainder owned by the SDMA D=1. The count field of the buffer descriptor indicates how much data has been transmitted.

If MCU (or DSP) has prepared 3 buffers to be filled by the SDMA script, it has also prepared 3 BD: one for each buffer. The Cont and Wrap bits are used to organize the buffers in a circular way. If CONTinous bit is set to 1 in the 2 first BDs and Wrap is set to the 3rd BD, the SDMA script open and process BD#1, because CONTinous bit is set for this BD, the SDMA will open the second BD and it will process it. Each time a BD is process, its Done bit is reset by the SDMA. After the 3rd BD, if CONTinous is not set but if Wrap is set, the SDMA script stops here and the next time the channel will be triggered, the script will open the BD pointed by the currentBDptr pointer of the CCB and it will correspond to the first buffer descriptor. Indeed Wrap bit is used in setBD (to close BD) to wite baseBDptr address in the currentBDptr of the CCB.

If CONTinous bit and Wrap bit is set in the 3rd BD, the script will close it and it will try to open the first BD but here an error maybe raised. Indeed, the BD#1 has already been processed, its Done bit is 0, and SDMA script does not tolerate a BD with a Done bit to 0, it means the BD has to be processed by the processor(MCU or DSP). This is why the CONTinous should not be set for the last BD if Wrap is set, and the Interrupt flag is set only for the last BD. It will warn the owner of the BD (MCU or DSP) that all the BDs have been treated and it has to re-set to 1 the Done bit of all the BD after processing the BDs if it desires the SDMA to fill them again.

Basically, if the MCU or DSP expects the SDMA to process the buffers in a circular fashion, then it should take care that each BD processed by the SDMA is treated by the MCU or DSP until the SDMA will try to open it again. This can be done by setting the Interrupt flag for every BD (thus being signalled after the SDMA has finished with every BD, one at a time) and processing the BDs fast enough that the SDMA will not catch up and try to open a BD that is owned by the MCU or DSP, and has the DONE bit set to 0. In this case, the last BD in the array must have the WRAP and CONT bits set to one (so that the SDMA will try to open the first BD after processing the last one), as well as the Interrupt flag Another sensitive point in this approach is the size of the buffers being processed by the SDMA - if the buffers are processed too fast, the DSP or MCU can be flooded with interrupts, so care must be taken if this way of using the I.API is chosen.



When the incoming data were stored in the first buffer descriptor of 25 bytes, the SDMA script opens the second one because the CONTinuous bit was set. Then next incoming data are put in the second buffer. After receiving 50 bytes, the second buffer descriptor is also closed: the DONE bit is reset and the third one is opened. After 25 bytes, the third buffer is full and an interrupt is sent to the MCU because the Interrupt flag is set in the 3rd BD. The CONTinuous flag is not present so it means the transfer is over but the next time the script will be triggered, the BD to be opened will be the first buffer descriptor. Indeed the Wrap flag was set in the 3rd BD. Again, it is the MCU (or DSP) responsibility to set again the DONE bit of all the BD if it wants to use the same buffers.

Consequently the structure used for the API is:

```
typedef struct iapi_ModeCount {
                unsigned long         command             : 8;
                unsigned long         status              : 8;
                unsigned long         count               : 16;
} modeCount;

typedef struct iapi_BufferDescriptor {
                ModeCount             mode;
                void *                bufferAddr;
                void *                extBufferAddr;
} bufferDescriptor;
```

2.3.3 Handling Errors

There are four main errors for the SDMA: Illegal instruction error, Channel control block access errors (CCB), buffer descriptor (BD) access errors and data transfer errors.

Illegal instruction causes the SDMA to trap and go into an infinite loop, the host watchdog should the take the decision to reset (see [4]).

The activation of the watchdog is essential in preventing situations in which the system can be blocked. If the I.API polling mode is used, and SDMA enters the infinite loop, then the only way to recover from the tight loop the system is in is by watchdog reset.

At SDMA startup, C0PTR is updated to point to the first channel control block: the host starts the command channel. The command channel script will attempt to access the channel control block and buffer descriptors using the DMA to the host core memory. It expects to find the channel control block structure for all virtual DMA channels. If the access to any one of the CCB fails, the SDMA calls the illegal instruction handler which triggers a software breakpoint for debugging.

While starting and stopping various virtual DMA channels the running scripts access buffer

descriptors with the DMA to the host core memory. If such operation fails, the buffer descriptor is not accessible and the virtual DMA channel information is not available to that channel: "BD access error" is fatal for the virtual DMA channel.

2.3.3.1 Behaviour in case of CCB access error

The error occurs on a ldf/stf instructions for a channel control block (CCB). The SDMA considers this error as "illegal" by jumping to the Illegal Instruction trap which triggers a software breakpoint for debugging.

2.3.3.2 Behaviour in case of BD access error

The error occurs on a ldf/stf instructions for a channel buffer descriptor (BD). The SDMA considers the error to be fatal for the current virtual DMA channel, the command channel script then performs the following:

- Access the channel control block for the virtual DMA channel and indicate the error by setting to one the error bit (bit 28) in the status word
- Terminate the channel (done)

The host processor checks the status long for the virtual DMA channel. If the error status is set, a buffer descriptor access for that virtual DMA channel failed. The recommendation is for the host processor to consider the virtual DMA structure for that channel corrupted, close the channel and re-open it: this operation free/re-allocate the resources (BD and buffers). The application writer should be research further in the host application program the reason for the original corruption of the structures.

These errors are reported up the API. In the case of non-blocking calls using call-back functions, the user shall use the input-output control routine to check the status.

2.3.3.3 Additional data transfer failure indication

Virtual DMA channels transfer data using the SDMA, typically this transfer will use load and store instruction to and from the host or dedicated memories. This implies to use the corresponding DMA that access the host and dedicated processor memories. An error can be detected on the DMA operation and the SDMA can indicate it to the host or dedicated buffer descriptor structure. The convention proposed and implemented in the IAPI is to update the buffer descriptor mode status bit indicating that the corresponding data buffer encountered and error; and update the channel control block status data failure error bit to inform that the corresponding virtual DMA channel encountered an error upon data transfer. This second update is meant to facilitate the error condition checking after the channel terminates.

2.4 Use of SDMA on board memory

2.4.1 Description

The SDMA on board memory is described in details in the memory space of the SDMA chapter ([4] ch: 9.6.4). From a general programmer's standpoint, the SDMA on board memory is split in two memory maps: the instruction memory map and the data memory map. The difference resides in the peripheral external programming model registers not being visible from the instruction memory map. From a script standpoint everything is accessible through the data memory map. The big categories of data found in the SDMA on board memory are:

- *Boot code and standard routines:* start-up of the SDMA and scripts known not to change (ROM).
- *Context switch routine:* this routine is detailed in the SDMA chapter and uses a set of dedicated SDMA instructions to switch from one channel to another (ROM).For reference, V100 and V200 version use a special set of instructions for this operations

while V300 performs the save restore operation in the background (see **Error! Reference source not found.**).

- *Channel contexts:* these memory structures hold the information for the SDMA to be able to restore or save a context for a given channel (RAM).
- *Data and user script routines:* the scripts and data the users attaches to the SDMA channels.
- *Peripheral registers:* see the external programming models of the respective SDMA peripherals.
- *Scheduler registers:* the memory mapped registers are the ones that the script may need to access to gain knowledge of the state of channels and SDMA core in general. This set is different from the one visible on the host or the dedicated processor external programming models.

2.4.2 Channel contexts

There are 32 channel context memory structures pointed to by the local save area pointer. These channel context memory structures are fixed. The script in the SDMA computes the memory offset for a given channel based on the structure length and channel number. The figure below shows the structure of the channel context as it is saved in the SDMA local memory:

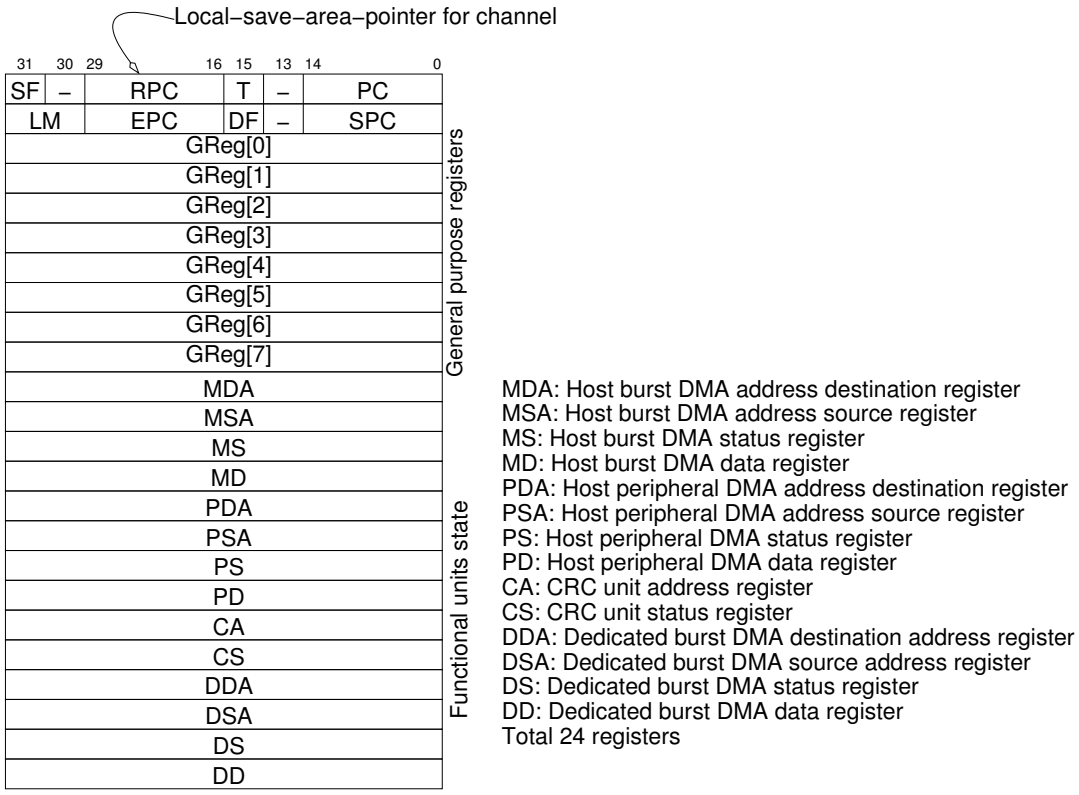
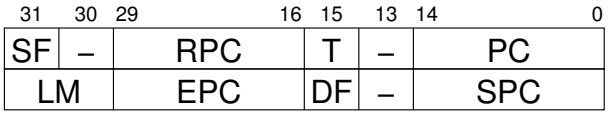


Figure 5: *Channel context memory structure.*

The structure is divided in three areas, the first one is the general purpose registers, the second the channel status registers and the third is the functional units state registers reflecting the state of the dedicated processor and host DMAs and the CRC unit respectively. The local save area pointer points to the address of the first element of the saved channel context. The details of the channel context state registers are:



SF: source fault while loading data
RPC: return program counter
T: test bit: status of arithmetic and test instructions
PC: program counter
LM: loop mode
EPC: loop end program counter
DF: destination fault while storing data
SPC: loop start program counter

Figure 6: *Processor Status in channel context memory structure*

Consequently the structure used for the API is:

```
typedef struct iapi_StateRegiters {
               unsigned int          sf                : 1;
               unsigned int          unused0           : 1;
               unsigned int          rpc               : 14;
               unsigned int          t                 : 1;
               unsigned int          unused1           : 1;
               unsigned int          pc                : 14;
               unsigned int          lm                : 1;
               unsigned int          lm_last_loop      : 1;
               unsigned int          epc               : 14;
               unsigned int          df                : 1;
               unsigned int          unused3           : 1;
               unsigned int          spc               : 14;
} stateRegiters;

typedef struct iapi_ContextData {
               stateRegiters         channelState;
               unsigned long         gReg[IPCM_NUMBER_GREGS];
               unsigned long         mda;
               unsigned long         msa;
               unsigned long         ms;
               unsigned long         md;
               unsigned long         pda;
               unsigned long         psa;
               unsigned long         ps;
               unsigned long         pd;
               unsigned long         ca;
               unsigned long         cs;
               unsigned long         dda;
               unsigned long         dsa;
               unsigned long         ds;
               unsigned long         dd;
```

```
} contextData;


typedef struct iapi_script_data {

                unsigned short load_address
                unsigned long  wml;
                unsigned long  shp_addr;
                unsigned long  event_mask1;
                unsigned long  event_mask2;
            } script_data;
```

2.4.3 Summary of memory structure: overall schema.

The paragraphs above detailed each memory structure necessary for SDMA virtual DMA channels to run. In order to use the SDMA virtual DMA channels, all of the presented memory structures are necessary together with the more classical external programming model registers.

The figure below, given as an example, summarizes the Rainbow SDMA external programming model in a condensed form to highlight the parts involved in the set up and use of an SDMA virtual DMA channel.

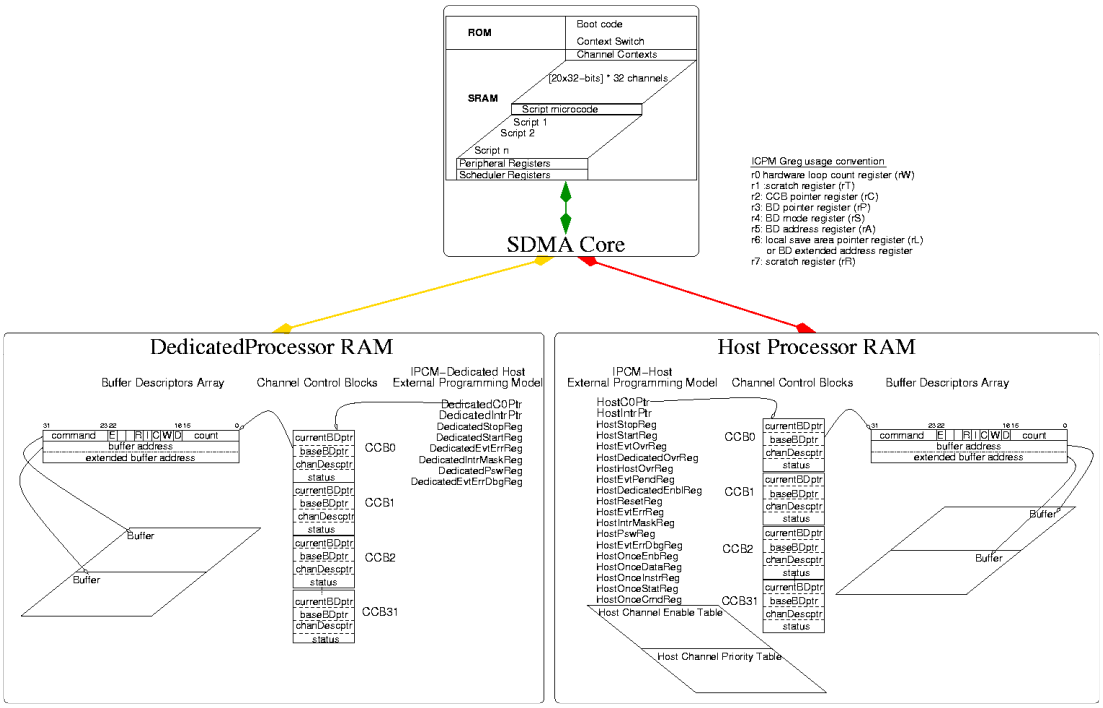


Figure 7: *Overall External Programming Model*

2.5 Memory structure programmers important remarks

Channel Control block for 32 channels must be contiguous in memory as the original example script expects them to be this way: address of next control block is computed by adding 4 32-bit longs to the current control block address.

For the buffer descriptors, dynamic allocation is supported: the number of buffer descriptors can be changed. The controlling host processor stops the channel and can allocate more or less descriptors; it can also move these buffer descriptors in its memory space. The SDMA script allows flexibility on the buffer descriptor array size and placement in memory.

This is interesting because when the buffer descriptors for a channel are fixed, a high volume of data may overflow the buffer descriptors' data buffers capacity. This would force the controlling application program to have another handling layer on top of the SDMA accesses to control the flow and volume of transfers. Dynamic buffer descriptors avoids the extra complexity for the higher level client application.

2.6 Channel enabling and override configuration properties

An SDMA virtual DMA channel is typically a unidirectional data channel between the SDMA and its external destination cores using the DMAs. The external destination cores can be either the host processor or the dedicated core. The host processor has more configuration control of the SDMA than the dedicated core: the host processor is really the host master of the Rainbow architecture and will configure the SDMA virtual DMA channels.

The virtual DMA channel properties detailed below relate to the state of a channel and when it is, or not, run able. A channel is run able if and only if:

(HE[i] | HO[i]) & (DE[i] | DO[i]) & (EP[i] | EO[i])

where i is the channel index number [0-31].

For an SDMA virtual DMA channel to be run able, HE[i] and/or DE[i] need to be set: this bits are set by the application program running on the corresponding cores and show that the memory structure have been allocated and loaded with proper content for the channel to properly function.
The EP[i], event pending register is the scheduler output which has determined which of the event channel lines was the highest priority to be run at this time.

The host processor external programming model presents a set of registers to control the following virtual DMA channel properties:

- Event Override: channel is not conditioned to the triggering of an SDMA event line.
- Host Override: host processor will not be involved in the waking up of the channel: HE[i] is not taken into account. The host processor allows for the definition of a channel in which it is not involved.
- dedicated processor Override: dedicated core will not be involved in the waking up of the channel: DE[i] is not taken into account. The host processor configures this channel to disallow the dedicated processor access to the channel.

It is the combination of those six registers that define the property and use of a channel. A summary table of channels' uses for a particular property can be presented:

HO	DO	EO	Run Condition	Typical use
0	0	0	HE & DE & EP	Memory structures need to have been set on host and dedicated processor side (HE and DE) and an event need to occur for the channel to start: this could be the case of an SDMA peripheral transmitting to both sides.
0	0	1	HE & DE	host to dedicated processor transfer
0	1	0	HE & EP	SDMA peripheral to host transfer (triggered by peripheral)

0	1	1	HE	host to SDMA peripheral transfer
1	0	0	DE & EP	SDMA peripheral to dedicated processor transfer (triggered by peripheral)
1	0	1	DE	Dedicated processor to SDMA peripheral transfer
1	1	0	EP	Overrides for both host and dedicated processor are set: no buffers are ready on either side. This property could be used for SDMA peripheral to peripheral communications
1	1	1	-	Immediate run until property is changed: not usable.

Table 1: **Channel configuration properties**

The override registers and their interactions with the DE, HE, EP registers respectively defines the extra capability that the host processor has over the dedicated core side. The remaining registers defined in the host processor external programming model that are unique to the host side are the ONcE registers, the ability to read the event pending and the dedicated processor channel enable register (HostEvtPendReg Hostdedicated processorEnblReg) and the Reset register respectively, these do not change the SDMA virtual DMA channel properties.

2.7 SDMA architecture note

The description in the SDMA specification document ([4] chapters: 9.2.2.1, 9.2.2.2) shows that instruction and data access the SRAM. Conflicts are resolved by the system bus when data accesses and instruction accesses conflict. Because the system bus is 32- bits wide and instructions are 16-bit, the even address fetches the most significant part of the 32-bit accessed value and the odd address fetches the least significant part of the 32-bit word accessed from either ROM or SRAM.

3 Methods for Virtual DMA synchronization

3.1 Introduction

There are two methods:

- Blocking calls to the library (support from the operating system is required)
- Use a call-back function attached to the channel (non-blocking calls to the library)

These two methods are provided to accommodate the various software architecture designed at the upper layers.

3.2 I.API Call-back ISR

The I.API proposes to use the call-back functions to handle exceptions triggered by the virtual DMA channel. The call-back function mechanism is often used in graphical user interfaces (GUI) to handle human interaction in a graphic environment (X-Windows). The call-back function is a function pointer that is attached to an event at initialisation time. When the event occurs, the event handler calls the function, the call-back function hence its name.

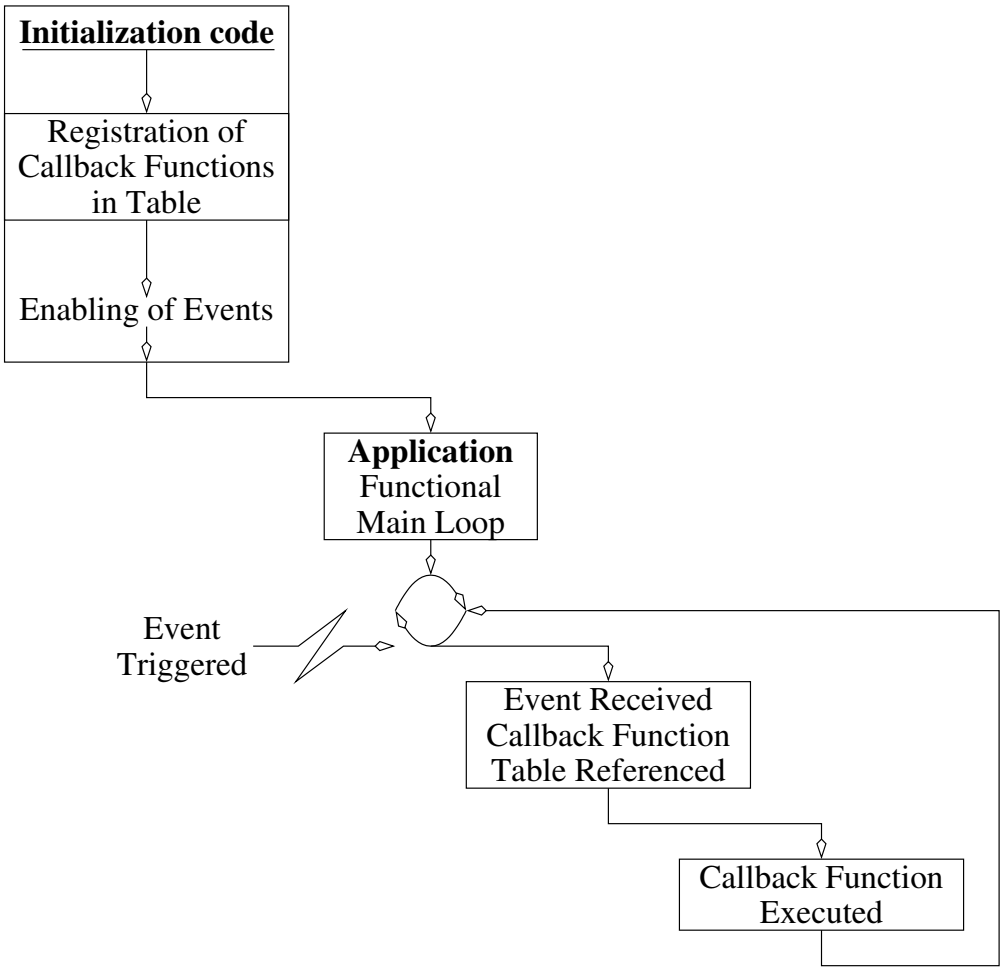


Figure 8: *Generic call-back function mechanism*

The I.API idea is to have the exception handler of the core jump to the call-back function's address. The action or processing to be executed upon such an exception is inside the call-back function.

The initialisation of the call-back function table occurs at the initialisation of the SDMA performed by the I.API. The initialisation fills the function pointer in the channel descriptor table associated with the channel. This initialisation is performed in order to keep control of the processing upon reception of an event once events are enabled. Because hardware peripherals can send events in an asynchronous manner, attaching a call back function will guarantee handling of the event. To that intent a "default" call back function could be attached at initialisation (Figure 9:), that for example, merely indicates what event was triggered for debug purposes, and later the user can use the I.API attach to another function of choice.

Also, the user of I.API has the option of registering a parameter to be passed to the call-back routine, helping the user pass data to the call-back function in an elegant manner.

3.2.1 Initialise, change and remove an action

The fact that the initialisation code must define to the system the function's address to be called upon reception of an interrupt event, is called attachment of the call-back-ISR. The I.API defines the attach action as an atomic action to be provided by the system supporting library. In the same manner the detach atomic action is the fact of de-registering the call- back function's address from the system call-back-ISR table. The detach atomic action must also be

part of the system support library.

Simultaneous support for attach and detach function allows the system to also change the call-back ISR while up and running. System support for attach/detach includes the enabling/disabling of interrupt events during the change of the call-back-ISR address in the system's table.

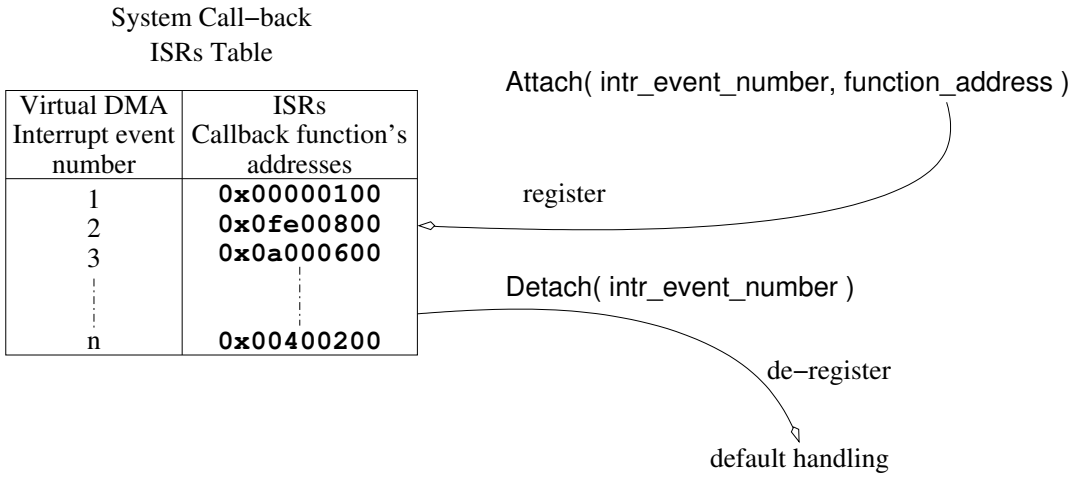


Figure 9: *Attach and detach for call-back Interrupt Service Routines*

In order to maintain system integrity, a default interrupt service routine must be provided, that accounts for the time where no call-back ISR is provided or during the time where the application wants to change the existing call-back ISR. This default ISR can also be a user provided call-back ISR.

3.2.2 Hardware independence

The I.API can be applicable in many hardware configurations. The latest third generation architecture embeds multiple different processor cores each of which have a different way to handle interrupts or exceptions more generally. The virtual DMA device can therefore "talk".

The way the two cores have to take an interrupt into account is different on the host and the dedicated processor side. The I.API call-back ISR can accommodate these differences. The I.API uses the schema of interrupt priority determination provided in the core architecture (fast or normal interrupt), and the service provided by the interrupt controller which allows the first level handler to determine the interrupt number. The algorithm is purely software, relying on the information provided by the interrupt controller and reacting as quickly as the hardware can establish priority and jump through the exception vector table.

The three next figures detail the flow using the I.API with a different hardware support: one with autovectored interrupt where the interrupt from the virtual DMA device is mixed with other devices and all virtual DMA channel interrupt events are OR-ed together, and a second case with vectored interrupts which highlights the removal of the decision stage which determines the interrupt source's device. Finally a case where all virtual DMA channels are wired on the processor's interrupt lines.

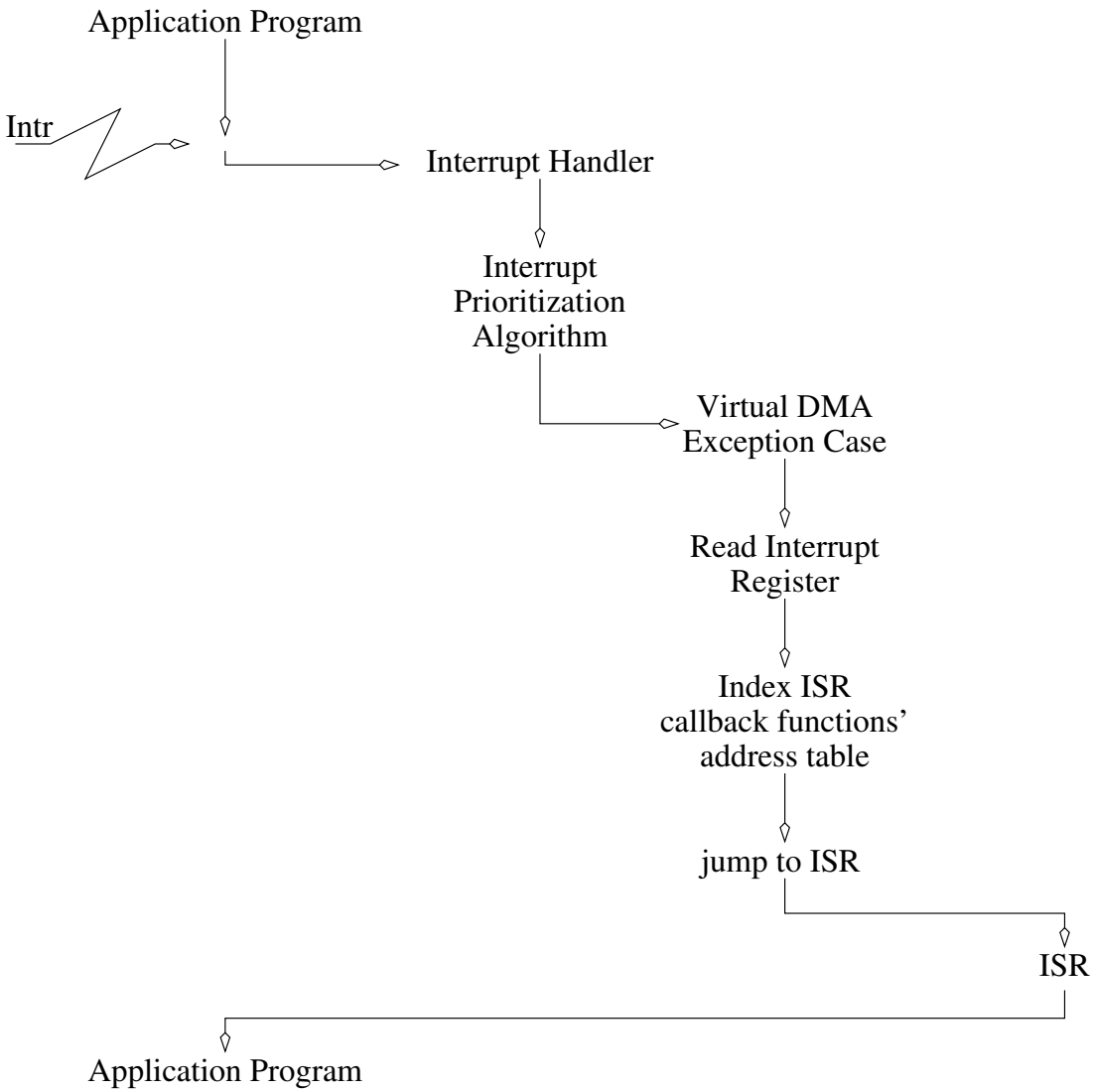


Figure 10: *Autovector with multiple device sources and OR-ed channels*

The steps can be detailed as follow:

- application running on the processing core (host or dedicated) starts,
- interrupt event is triggered by the SDMA and received by the processing core,
- interrupt handler is called and starts executing,
- interrupt prioritisation algorithm determines it is an virtual DMA exception case that need to be handled among all devices connected to the same interrupt line,
- the interrupt register of the SDMA is access-ed to get the interrupt number: the number correspond to the virtual DMA channel number on which the interrupt event occurred,
- the interrupt event number is used to index the call-back functions' table in order to get the address of the call-back function to be called: using the channel number as the index of the system's call-back ISR table allows to get to the jump sooner,
- the jump to the call-back function's address is performed giving control to the call-back Interrupt Service Routine,

- After the ISR execution control is returned to the application.

If hardware has support for vectored interrupts, the source device determination step is no longer necessary:

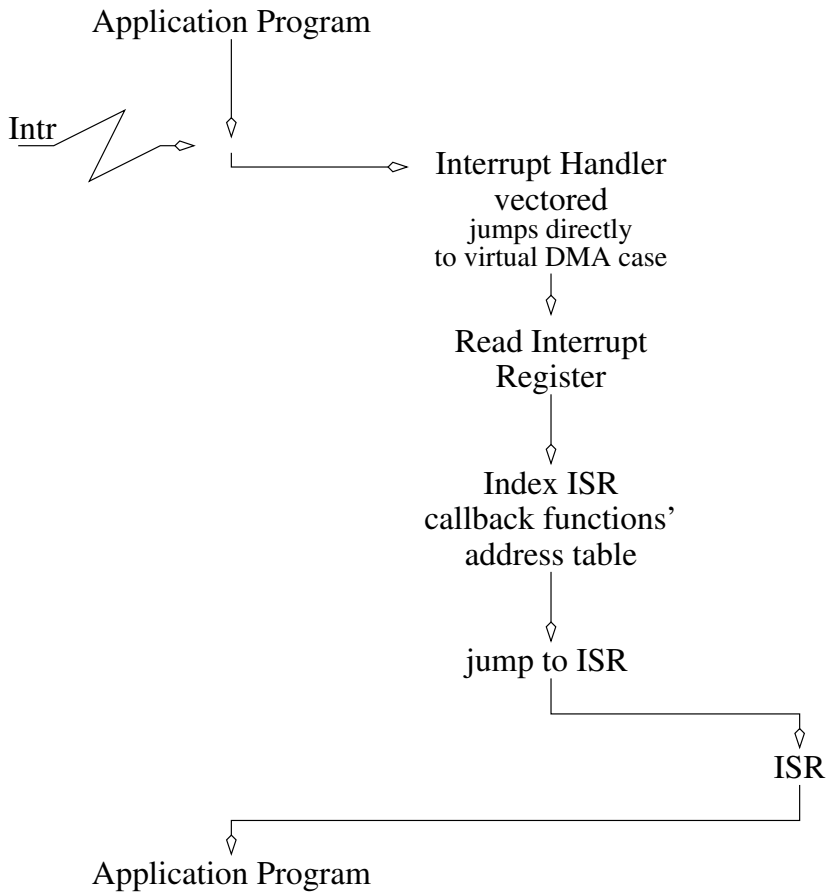


Figure 11: *Vectored with OR-ed channels*

The attach/detach routines maintains the call-back ISR functions table the same way it did in the first case.

Ultimately, if all the virtual DMA events can be wired on their private interrupt lines and the core supports vectored interrupts the read access to the interrupt register of the virtual DMA device can also be removed:

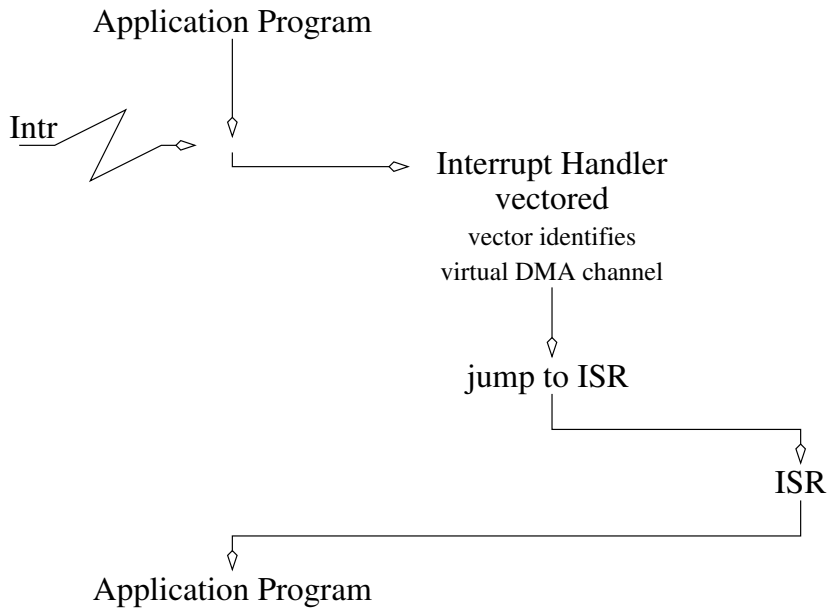


Figure 12: *Vectored with virtual DMA channels private interrupt lines*

The attach routine at this point need to write the call-back ISR function's address in the interrupt vector table. This architecture choice is the fastest, but also may require too many interrupt lines since each virtual DMA channel requires its own.

3.2.3 Synchronization: possible paradigm shift.

Because a call-back function can be attached to an event, the I.API provides a mechanism of synchronization at the application level: as soon the interrupt is called the application effectively calls the action function.

The I.API proposes an efficient way to get to this point because the address of the function to jump to, that is the call-back ISR, is recorded in the call-back table at core initialisation time. During application run time the delay is only the jump instruction that the hardware will need to perform.

Complex application can be viewed as state machines representing the system. In architecture where interrupt service time was not efficient enough, events such as the virtual DMA events make the internal change state but the action was not yet taken, it was scheduled: the ISR merely records the event occurred.

The call-back ISR reduces the time it takes to get to the action to be performed. A gain in the number of cycles spent in exception handling enhances the ability of the entire system to react to an event. This enables the software architecture to better schedule and split its tasks, resulting in a cycle gain at the system level.

3.2.4 I.API provided interrupt handler

I.API provides a default interrupt handler for the user of the library. If the user is satisfied with the default processing of the SDMA interrupt, it only has to take care about the registering of the function to the system using the I.API.

The default ISR is named *IRQ_Handler*. The function disables the interrupts, then checks the appropriate register of the SDMA (it depends on the core receiving the interrupt) and for every channel of the SDMA signalling an interrupt it wakes up the channel (see 3.3) calling *iapi_WakeUp*, then, if a callback function is registered on that channel, executes the function.

Prior to executing the callback function, the channel is released using the *iapi_ReleaseChannel* function (see 4.1)

After all channels signalling an interrupt are treated, interrupts are enabled and the IRQ_Handler returns.

In case that the user writes its own IRQ handler, this code can be seen as a starting point and recommendation for what the IRQ handler needs to do for the proper functioning of the library.

3.3 Blocking calls to I.API

This synchronization method is provided for the users of I.API wanting to wait until the operation started by a Read/Write call is finished. In order for this feature of the I.API to be efficient, operating system support is necessary. This support is accessed by 2 function entry points provided by the I.API, which must be mapped by the user of the I.API to appropriate routines:

iapi_InitSleep(int): this function is called at the opening of a channel, and receives as a parameter the number of the channel being opened. It must be mapped to a initialization function for a wait object in the OS used (for example, it can initialize a wait queue allocated for the opened channel with *init_waitqueue_head(&wait_queue_array[channel_number])* in Linux)

iapi_GotoSleep(int): this function is called after the channel setup is finished for the Read/Write operation, immediately after the channel has been started. It receives as a parameter the number of the channel being called for. It must be mapped to a function that puts to sleep the calling process, on the wait object initialized at the channel opening (for example, it can sleep on a wait queue with *interruptible_sleep_on(&wait_queue_array[channel_number])* in Linux)

iapi_WakeUp(int): this function is called in the default SDMA IRQ handler. It receives as a parameter the number of the channel being called for. It must be mapped to a function that wakes up the process put to sleep on the channel number received as a parameter.

The process being put to sleep by the iapi_GotoSleep() call must be able to execute when the started operation has finished. This will be accomplished in the Interrupt Service Routine (ISR) for the SDMA when the channel signals an interrupt, by calling iapi_WakeUp (to continue our Linux example, this should be mapped to a function like *wake_up(&wait_queue_array[channel_number])*).

If the I.API is used in an environment with no operating system, very simple implementations for these functions must be provided (for example, performing polling on a variable updated in the ISR is a quick and functional implementation).

4 Architecture of the API.

The API is aimed at providing routines to easily handle the SDMA channels. The channels are most often used as pipes between point A and B. The supporting concept of the API is to allow SDMA channels' access in an OPEN, READ, WRITE, CLOSE, IOCTL fashion.

The library is written in C and compiled on the host processor and dedicated core using their respective compiler tool chains in order to guarantee functionality and integrity in the usage of SDMA virtual DMA channels.

The library is layered in high, middle and low level routines, the open-read-write-close functions being at the highest level of the API. Lower level functions can be called directly. These functions address configuration properties and more direct control of the devices through their respective external programming models.

In order to accommodate the usage of I.API under different operating systems, I.API provides entry points for functions dealing with synchronization, memory access and mutual exclusion for channel access.

4.1 I.API operating system support

Using the I.API under a OS, some problems must be addressed: memory access function (allocation, deletion, copy), virtual and physical addresses, resource access, synchronization.

I.API provides function entry points to deal with the aspects presented above.

Regarding memory access, I.API provides the following function entry points:

iapi_Malloc (size_t size)

*iapi_Free (void * ptr)*

*iapi_memcpy(void *dest, const void *src, size_t count)*

*iapi_memset(void *dest, int c, size_t count)*

These functions are used in the library to allocate, free, copy and fill memory. The user must map this entry points to appropriate OS provided functions.

When memory is allocated on platforms with the Memory Management Unit present, one does not get the physical address, but a virtual address. The SDMA, in turns, needs the physical address to access the data in memory, so fields of the I.API data structures accessed by the SDMA must contain the physical address, not the virtual one obtained at allocation. In consequence, two I.API entry points must be mapped to appropriate functions to perform translation of address from virtual to physical and from physical to virtual:

*iapi_Virt2Phys (void * ptr)*

*iapi_Phys2Virt (void * ptr)*

Resource access is another point in which the I.API needs the OS support. By resource we understand channels, as operations on a channel (open/read/write/ioctl) can only be performed by a single thread at a time. To accommodate this, whenever I.API performs an operation on a channel, it tries first to acquire the ownership of the channel, and after it has finished, it releases the ownership. This is done by calling the following functions, passing as a parameter the channel number:

iapi_GetChannel(int)

iapi_ReleaseChannel(int)

The functions must return 0 on success and -1 on failure. When using blocking calls to perform read/write using I.API, the channel is released by the I.API, when the operation has finished, prior to returning control to the calling thread. When call-back synchronization is used and the default ISR handling offered by the SDMA is not used, the user of the I.API must release the used channel when the operation has finished (it can be done in the ISR or in the call-back routine, depending on the complexity of the operation performed). Not releasing the channel will cause the I.API to return an error when another operation is requested on the channel (read/write/ioctl/close).

For synchronization purposes, the I.API provides the following entry points:

iapi_InitSleep(int)

iapi_GotoSleep(int)

iapi_WakeUp(int)

Please see the section **3.3 Blocking calls to I.API** for more details.

In order to correctly perform the call-back routine operations (attaching, detaching and replacing), I.API needs to have interrupts to the processor disabled. It must also enable the interrupts after. Since these operations are usually OS dependant, I.API provides the following two entry points:

iapi_EnableInterrupts(void)

iapi_DisableInterrupts(void)

All above presented entry points must be set prior to opening a channel with the I.API, or the opening will fail.

4.2 SDMA initialisation

When starting and using an SDMA virtual DMA channel the following operations need to take place:

1. channel control blocks (CCBs) are allocated
2. buffer descriptor (BDs) are allocated
3. Channel context structures are allocated AND filled with proper data for channels to be used. A pre-defined constant need to hold the address in the SDMA memory space of the channel context structures.
4. channel buffer descriptor are filled with proper data:
    a. Command: if handling script handles some commands for this channel, this is the place to put the code for them. For the channel zero, which is reserved for host processor to SDMA communications, there is a set of pre-defined commands: C0_SETPM, C0_GETPM, C0_SETDM, C0_GETDM, C0_SETCTX, C0_GETCTX.
    b. Status: the status of the buffer descriptor, this sets one of the status bits according to the buffer descriptor: E, I, C, W, R, D.
    c. count: the size of the transfer in bytes
    d. Buffer address: the address of the available buffer for the transfer (of minimum size = count) allocated in the host or dedicated core memory.
5. iapi_ChannelConfig then performs the setting of the channel properties: see the channel configuration properties table above. Interrupt handling policy method is set at this step (see below).
6. iapi_InitChannelTable sends the priority table and the channel enable RAM table to the SDMA using the available external programming model registers.
7. Enable interruption of the communicating core.
8. Write Channel Control Block address to the host processor channel control block register HostC0Ptr, and to the dedicated core channel control block register DedicC0Ptr. This signifies that the structures are allocated and ready on both sides. Though this does not make the channel run able: the run able condition needs to be verified.
9. Channel zero is started: because of the set up performed on channel zero control blocks and buffer descriptor, the context for subsequent channels is loaded at this time. This is also the point where depending on the control blocks defined for channel zero, scripts for other channels can be down loaded to the SDMA local memory.

10. Host processor waits for channel zero: SDMA will interrupt the host when completed. Synchronization is interrupt based at this point and depends on the policy chosen by the user for the channel, see below for details.
11. Other channels can now be started, stopped etc....

All operations related to the initialisation of the SDMA are performed by an *iapi_Init()* function. Besides initialisation, the function offers the opportunity to initialise the useful bits of the SDMA CONFIG register (an iapi_ConfigDefaults structure is available to set defaults values), to download a RAM image for SDMA, thus easing the use of the I.API. If the user wants to perform the download at this step, the interrupts must be enabled prior to the calling of this function.

Any of the numbered bullets can be rearranged in its order: this is a one possible algorithm used in the I.API. The main goal is to have the necessary structures in place before the SDMA starts running the requested channel.

4.3 SDMA transfer flow description

4.3.1 Operation description

The Inter-Processor Communication Module (SDMA) is a design intended to facilitate data transfer on large System On a Chip (SoC) designs. The SDMA therefore moves data from a source to a destination. The SDMA is also capable of performing basic operations while transferring the data because it is driven by a micro-risk architecture core. These properties put the SDMA in the class of so called smart direct memory access devices (Smart DMA).

These data moves across the SoC architecture are performed using the virtual DMA channels controlled by the SDMA script; the script executes assembly instructions that allow the SDMA to pipe data from source to destination. The SDMA relies on data structures present in both the host and dedicated processor memories. These structures are the channel control block and the buffer descriptor presented above, and those are common to the host and dedicated processors.

Host and dedicated processor allocate these structures and indicate to the SDMA that these are ready by writing the address of the first required channel control block to the command channel pointer register (C0PTR).

Source and destination address for the transfer are indicated in the buffer descriptors. In case of host to dedicated processor transfers, there can be two counts, one each for host and dedicated processor. The SDMA script will manage the possible buffer size difference between the source and destination utilizing one of the status bit described in the buffer descriptor (CONT). This allows the client application not to be concerned by buffer size differences between the host and the dedicated processor sides.

In the typical transfer, the SDMA script reads the C0PTR register to get the address of the channel control block and indexes into the using the current channel register (CCR) to get the buffer descriptor pointer. The buffer descriptor contains the status information for this channel's buffer descriptor that provides the mode word, the data count and the data buffer pointer to the verbatim data buffer.

The same processing takes place on both the host and the dedicated processor sides and the script enters the transfer loop which forwards data from one side to the other. At this point the script manages the possible buffer size differences based on the mode bits.

After the script transferred all data controlled by a buffer descriptor, the control bits in the mode word and also the buffer descriptor pointer. Processing of the next buffer descriptor can proceed.

4.3.2 Command channel operation support

The role of the command channel (channel zero) during a normal data transfer operation is described to highlight the support provided by the data structures and command channel script presented above.

The command channel script is activated when the channel zero SDMA condition is set. Command channel script the accesses the channel control block for channel zero –like any other script- and executes the requested commands. These commands have been set and transfers prepared by the host processor when constructing the command channel data structures (CCB and BDs). Once performed command channel is done and HE[0] cleared reflect this status. Command channel can be presented in more detailed extracted from [5].

First, the appropriate channel CB is accessed by the subroutine getcb(). In the normal case, this routine will read the MC0PTR register to retrieve the CB base address, and add the appropriate channel offset into the CB by using the current channel number found in the CCR register. Once the channel's CB address is calculated, it is accessed to retrieve the current buffer descriptor address for the BD to process. The Test (T) flag is used as a return value, and is set if the MC0PTR or current BD address is zero.

Next the channel's BD array is accessed by the subroutine getbd(). Using the current BD address retrieved in getcb(), the BD's mode (status/count) word is read. The mode contains several control bits that determine further actions. If the D bit is set, the address word is read: indicating that the buffer descriptor needs to be processed by the SDMA and that the script is going to perform the transfer. The address field contains the host RAM address for the transfer. If the E bit is set, the extended address word is read. The command channel Script uses the extended address to hold the SDMA RAM address for the transfer. The T flag is used as a return value, and is set if the D bit in the status field is set.

Once all of the CB and BD control information is loaded into the registers, the command channel script checks the T flag for valid subroutine results and invokes the commanded memory transfer amongst: SET_DM, GET_DM, SET_PM, GET_PM, SET_CTX and GET_CTX.The script then executes the memory transfer. The address field contains the address of the DATA buffer in host memory and the extended address field contains the SDMA DM/PM address. During the memory transfer loop, the SDMA's SF and DF flags are monitored to determine if a transfer error occurs. If so, the error bit in the status field will be set.

After the memory transfer completes, the channel's CB and BD array are updated by the subroutine setbd(). Using the current BD address retrieved in getcb(), the BD's status/count word is updated after the D-done bit is cleared. Then the address of the next BD to process is determined. If the wrap status bit is set the address wraps back to the beginning of the BD array using the base buffer descriptor address. Otherwise, the appropriate offset is added to the current BD address. The address is updated in the current BD field and kept in register for continued operation.

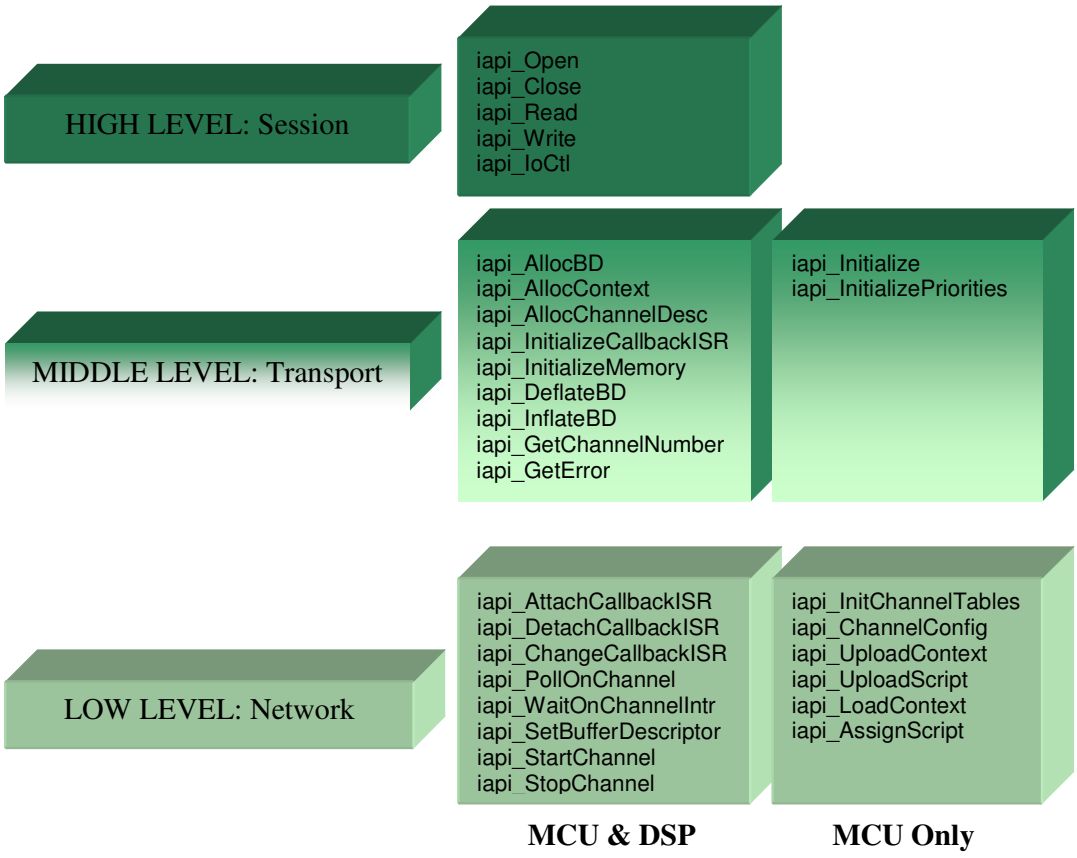
The status filed in the BD mode is checked to see if an interrupt needs to be issued to the processor and if an error occurred. If either of these two bits is set an interrupt is triggered to the host. The setbd() routine returns the updated current BD address in the register set. The T flag is used as a return value, and is set if the continue bit in the status field is set.

If operation should continue onto the next BD using the updated current BD value in the general purpose register, the command channel script jumps to the beginning of the script where the getbd() routine is called. There is no need to call the getcb() routine for continued operation as the register values are already set. The getcb() routine is called only at script activation. This is to provide the ability for the host to change the channel CB values during the inactive state. If operation is not to continue, the script sleeps by clearing the HE[0] bit.

4.4 Library Structure

Given the above presented sequence of events and the external programming models, the figure below presents the different library routines ordering them by abstract level from highest (top) to lowest (down).

The concept model used for the library is the open, read/write, close model; in order to present the SDMA Virtual DMA channels' users with a familiar model that resembles the UNIX/LINUX concepts.



The next table details each function of the 3 layers.

High Level: Session	iapi_Open	*Opens an SDMA channel to be used by the library.*
	iapi_Read	*Read from the channel into user's data buffer.*
	iapi_Write	*Write user buffer to the channel.*
	iapi_IoCtl	*Perform I/O on the channel parameters.*
	iapi_Close	*Terminates a channel*
Middle Level: Transport	iapi_AllocBD	*Allocates one Buffer Descriptor structure using information present in the channel descriptor*
	iapi_AllocContext	*Allocate one channel context data structure*
	iapi_AllocChannelDesc	*Allocates channel description and fill in with default values.*
	iapi_ChangeChannelDesc	*Changes channel description information after performing sanity checks.*
	iapi_InitializeCallbackISR	*Initialize a table of function pointers that contain the interrupt Service Routine callback pointers for the IPCM channels with a default value.*
	iapi_InitializeMemory	*For the specified channel control block, attach the array of buffer descriptors, the channel description structure and initialize channel's status using information in the channel descriptor.*

	iapi_Initialize	*This is the I.API own intialization routine.*
	iapi_InitializePriorities	*Master host loads the channel priority tables onto the SDMA local memory.*
Low Level: Network	iapi_AttachCallbackISR	*Records an ISR callback function pointer into the ISR callback function table*
	iapi_DetachCallbackISR	*Detaches (removes) an ISR callback function pointer from the ISR callback function table*
	iapi_ChangeCallbackISR	*Updates an ISR callback function pointer into the ISR callback function table.*
	iapi_lowSynchChannel	*Go to sleep on the channel.*
	iapi_WaitOnChannelIntr	*This is waiting on the bit in the stop register to show up.*
	iapi_SetBufferDescriptor	*Fill the buffer descriptor with the values given in parameter.*
	iapi_Channel0Command	*Send a command on SDMA's channel zero.ARM only.*
	iapi_lowStartChannel	*Starts the channel (core specific)*
	iapi_lowStopChannel	*Stops the channel (core specific)*
	iapi_ChannelConfig	*Configure channel ownership.*
	iapi_lowGetContext	*Load the context data of a channel from SDMA.ARM only.*
	iapi_lowGetScript	*Load PM words from SDMA PM memory.ARM only.*
	iapi_lowSetScript	*Load a SDMA script to SDMA. ARM only.*
	iapi_lowSetChannelEventMapping	*Set the channels to be triggered by an event. ARM only.*
	iapi_lowSetContext	*Load the context for a channel to SDMA. ARM only.*
	iapi_lowAssignScript	*Associate specified channel with the script starting at the specified address.ARM only.*

Figure 13: *I.API Library Hierarchy layout*

The High Level interface consists in the Open/Read/Write/Close/Ioctl mechanism that is convenient to SDMA. However, some functions of the Low Level are visible at the High level (wraper) and a Init and a MemCopy functions are also proposed. But again the Open/Read/Write/Close interface is the prefered and recommended one.

Additional Functions available at High Level	iapi_MemCopy	*Perform a memory copy operation, in the memory of the same processor.*
	iapi_Init	*Initialization of the SDMA - opening of channel 0, download RAM image.*
	iapi_StartChannel	*High layer interface for starting a channel*
	iapi_StopChannel	*High layer interface for stopping a channel*
	iapi_SynchChannel	*High layer interface for synchronising a channel*
	iapi_GetChannelNumber	*Return the channel number from the channel descriptor*
	iapi_GetError	*Return the error bit from the current BD of the channel*
	iapi_GetCount	*Return the count from the current BD of the channel*
	iapi_GetCountAll	*Return the sum of counts for all the BD's owned by the processor*
	iapi_GetScript	*High layer interface for getting program memory data from SDMA. ARM only.*
	iapi_GetContext	*High layer interface for getting data memory from SDMA. ARM only.*
	iapi_SetScript	*High layer interface for set program memory data to SDMA - e.g. scripts. ARM only.*
	iapi_SetContext	*High layer interface for set data memory to SDMA - e.g. contexts. ARM only.*
	iapi_AssignScript	*High layer interface used to associate specified channel with a script. ARM only.*
	iapi_SetChannelEventMapping	*High level interface to set the channels to be triggered by an event. ARM only.*

Figure 14: *Additional Functions available at High level*

The library was divided into 3 levels in an attempt to clarify its functionality and use:

- High level: is the session level. The session level groups the set of call used by the application programmer user in regular operation of the SDMA virtual DMA channels. The session level imposes a structure on the "conversation" that applications may have, take turns in sending data, establishing synchronization points for completed tasks. To allow for easy integration of the I.API in the SDMA drivers for different OS's, in the high level were added functions from the lower levels, allowing complete control of the SDMA using only high level function. So while a Init/Open/Read/Write/Close approach is still feasible, a better control of the SDMA operation is allowed.
- Middle level: is the transport level, this level contains the mechanism provided by the I.API to guarantee that data is delivered via the virtual DMA channel, complete and in sequence. The transport level includes the data structures that define the way SDMA channels are viewed from the cores perspective. The memory reservation and generic configuration action are taken at this level. It is also at this level that occurs the assignment between a channel and a script.
- Low level: is the network level, which is the closest access to the SDMA functions, the calls presented here are utilities called from higher levels of the library. The network level defines and changes the properties of the virtual DMA channels at the lowest level: SDMA scripts to be called and priorities which can modify the order and the end points that a channel connects to.

4.5 SDMA virtual DMA channels direction

Virtual DMA SDMA channels are unidirectional; one of the properties of the channel is the direction in which the transfer is to happen. From the core establishing the dialogue with the SDMA virtual DMA channels, a read is the action of receiving data from the SDMA and write the action of sending data via the SDMA. The meaning of an interrupt received from the SDMA is therefore dependent on the direction of the transfer on this particular virtual DMA channel. In the case of a "read", receiving an interrupt means that the data has been copied from the SDMA to the host's data buffers. Data is ready to be passed on upper layers of the application. In the case of "write", receiving an interrupt means that the data has been delivered to the SDMA handling script.

Remark on the "D" done bit: when the processor (DSP or ARM) processes a BD, the "D" bit must be 0, and it is set to "1" just before sending the BD to the SDMA. When the SDMA processes data that reside in the data buffers of the dedicated processor on-chip SRam or host eDRam, the SDMA "D" done bit is "1", and it sets the bit to "0" to indicate that the SDMA has finished is job on the particular buffer descriptor. This "job" can be either read or write depending on the direction of the transfer. To conclude, "D" = 0 means that the SDMA owns the BD, while "D" = 1 means that the BD is owned by the processor in who's memory space is (ARM or DSP).

4.6 Details of the I.API session level

Following the data structure explained in the previous sections and paragraphs, it may be useful to keep in mind the following notes:

- The channel control blocks array must be statically defined.
- Buffer descriptors are assumed to be allocated dynamically in memory.

int
iapi_Open(channelDesc * cd_p, unsigned char channelNumber)

Opens an SDMA channel to be used by the library. The memory structure are allocated during the call and subsequently freed by close.
If the channel has been successfully opened, zero is returned by the function call and a pointer to the channelDescriptor structure is placed in the channel descriptor parameter. In case of failure, a negative one (-1) is returned by the function call and a NULL is returned in the channel descriptor parameter.

Open a channel performs the following:

- try to acquire channel performing a call to iapi_GetChannel(channelNumber). If the return value from this function is not 0, the iapi_Open returns an error.
- verify SDMA has been initialised: if a null pointer is passed in channelDesc * cd_p->ccb_ptr and the channelNumber is 0 the SDMA is not initialised. The initialisation needs to take place: allocate the CCB structure for all 32 channels (contiguous in memory).initialise the SDMA registers. Note the initialise function on the host handles supplementary registers.
- verify that the channel has not yet been allocated by looking at the channel control block base buffer descriptor pointer. If channel has already allocated memory resources, it should be closed first: existence of non Null pointer from the channel's control block is checked:
    - If channel already has resource allocated, try to diagnose further by examining the "D" bits of the buffer descriptor status field. Determine what has been processed by which core in order to build a more precise error message: detailing what is the state of buffers.
    - Release the channel, report the message and return the error code.

  Else channel is free, the opening process can continue.
- Allocate buffer descriptor and buffers.
- Establish channel's configuration properties according to the definitions detailed in Table 1: channel properties. This is performed by writing the override registers. The interrupt handling policy is then set accordingly.
- Channel control block structure is updated for the corresponding channel to point to the new buffer descriptor.
- Enable the channel and set priority to 1.
- Release the channel by a call to iapi_ReleaseChannel(channelNumber) and return IAPI_SUCCESS.

Parameters needed that are defined in the iapi_ChannelDefaults structure:

- channelNumber = 0
- bufferDescNumber = 1
- bufferSize = 8
- blocking = 0
- callbackSynch = DEFAULT_POLL
- ownership = DONT_OWN_CHANNEL (EVT + MCU + DSP)
- priority = 1
- trust = TRUE

- useDataSize = 0
- dataSize = 0
- forceClose = 0
- scriptId = 0
- watermarkLevel = 0
- eventMask1 = 0
- eventMask2 = 0
- peripheralAddr = NULL
- callbackISR_ptr = NULL
- iapi_channelControlBlock = NULL

The channel control block and the channel descriptor, both fully define the virtual DMA channels' information.

The status for the channel is stored in the channel control block structure status field and reflects if the channel is opened or closed (initialized or not), the channel direction from the SDMA standpoint and if the channel is has been activated. (O,S,X).

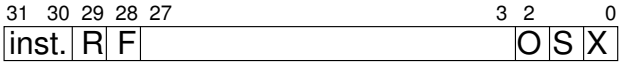


Figure 15: *Channel Status*

Buffer descriptor access error and data transfer failure use the 'R' and 'F' bits respectively to indicate errors (please refer to 2.3.3 for a detailed explanation).

Channel status-bits or mode-bits: define O,S,X flags:

- "O" : bit 2: When this bit is set, the channel is opened and initialized.
- "S" : bit 1: When this bit is set, the core is reading data from SDMA. When it is not set, the core it is writing to SDMA.
- "X": bit 0: When this bit is set, it signals that some operation is in progress on that channel .

The channel structure is defined as:

```
typedef struct iapi_ChannelStatus {
               unsigned long        ipcmInstance       : 2;
               unsigned long        unused             : 27;
               unsigned long        openedInit         : 1;
               unsigned long        stateDirection     : 1;
               unsigned long        execute            : 1;
} channelStatus;
```

The open function accesses the channelDescriptor structure defined by the user: it checks if

the value provided could be used and attempts to open the channel.

The library provides a default initialisation of the channelDescriptor structure that can be modified by the session layer call of the I.API: see iapi_ioctl().

If any value cannot be used as given or the attempt to open the channel fails, open returns -1. Otherwise the pointer to the channel descriptor structure is returned and the iapi_status bit shows a'1' confirming the open was successful. The status indicates if the channel has been successfully opened: this is for later accesses by other library calls and is homogenous with the status return by the iapi_Open call.

Figure 16: *Channel Descriptors*

The channel descriptors are defined in the following way:

- channelNumber: [0:31]
- bufferDescNumber: number of buffer descriptors attached to this channel control block.
- bufferSize: size of the buffers attached to each buffer descriptors; all buffer descriptors are assumed to have the same size. Size is an unsigned int that will be encoded in the buffer descriptor first long (see buffer descriptor structure described above. It could be decided to build a linked list of buffer descriptors and buffer descriptor non-constant sizes from here.
- blocking: defines if the Read/Write calls block until the number of bytes requested have been serviced. This feature is not yet available.
- callbackSynch: Interrupt policy chosen for this channel: callback Interrupt Service Routine (ISR) or default handling for polling mechanism.
- ownership: defines the channel property for its use and direction of transfer: see channel configuration properties above Table 1:.

- priority: priority for the channel, it reflects the channel priority on the SDMA [0-7] and allows the user to change the channel priority using input output control (ioctl).
- trust: defines if the buffer descriptor pointers that are handed over to the I.API should be trusted or not. Refer to the memory to memory transfer and input output control iapi_ioctl() call below.
- callbackISR_ptr: Function pointer to the callback function to be called upon receiving an interrupt on this channel.
- useDateSize: is the data size information used in transfers on this channel.
- dataSize: the transfer data size, coded on 2 bits.
- forceClose: should the channel be closed even if the buffer descriptor DONE bit indicate an operation in progress on the channel.
- ScriptID: name of the SDMA script that is attached to this channel.
- WatermarkLevel: Threshold of the FIFO peripheral (when involved in the transfer) that triggers an interrupt when it is over passed.
- eventMask1: DMA request number that triggers this channel. Must be in a one-hot code style. Applicable only for channel triggered by a peripheral.
- eventMask2
- ccb_ptr: channel control block pointer: NULL if the channel control block has not yet been initialized, pointer to the channel control block otherwise. It acts like a flag.

NOTE 1 : Choose of fixed Buffer Descriptor was done because full dynamic link list capability would move the real-time constraint from the application interface to the OS, inside prioritisation of the I.API drivers versus other tasks. Inside the RTOS we may have few priority levels.

Consequently the structure used for the channel description is:

```
typedef struct iapi_ChannelDescriptor {
                unsigned char          channelNumber;
                unsigned char          bufferDescNumber;
                unsigned short         bufferSize;
                unsigned long          blocking            : 3;
                unsigned long          callbackSynch       : 1;
                unsigned long          imageIntrReg        : 1;
                unsigned long          ownership           : 3;
                unsigned long          priority            : 3;
                unsigned long          trust               : 1;
                unsigned long          forceClose          : 1;
                unsigned long          scriptId            : 10;
                unsigned long          watermarkLevel      : 10;
                unsigned long          eventMask1;
                unsigned long          eventMask2;

                void (* callbackISR_ptr)( unsigned char *);
                channelControlBlock *  ccb_ptr;
} channelDescriptor;
```

The structure can therefore be viewed from the SDMA channel control block view:

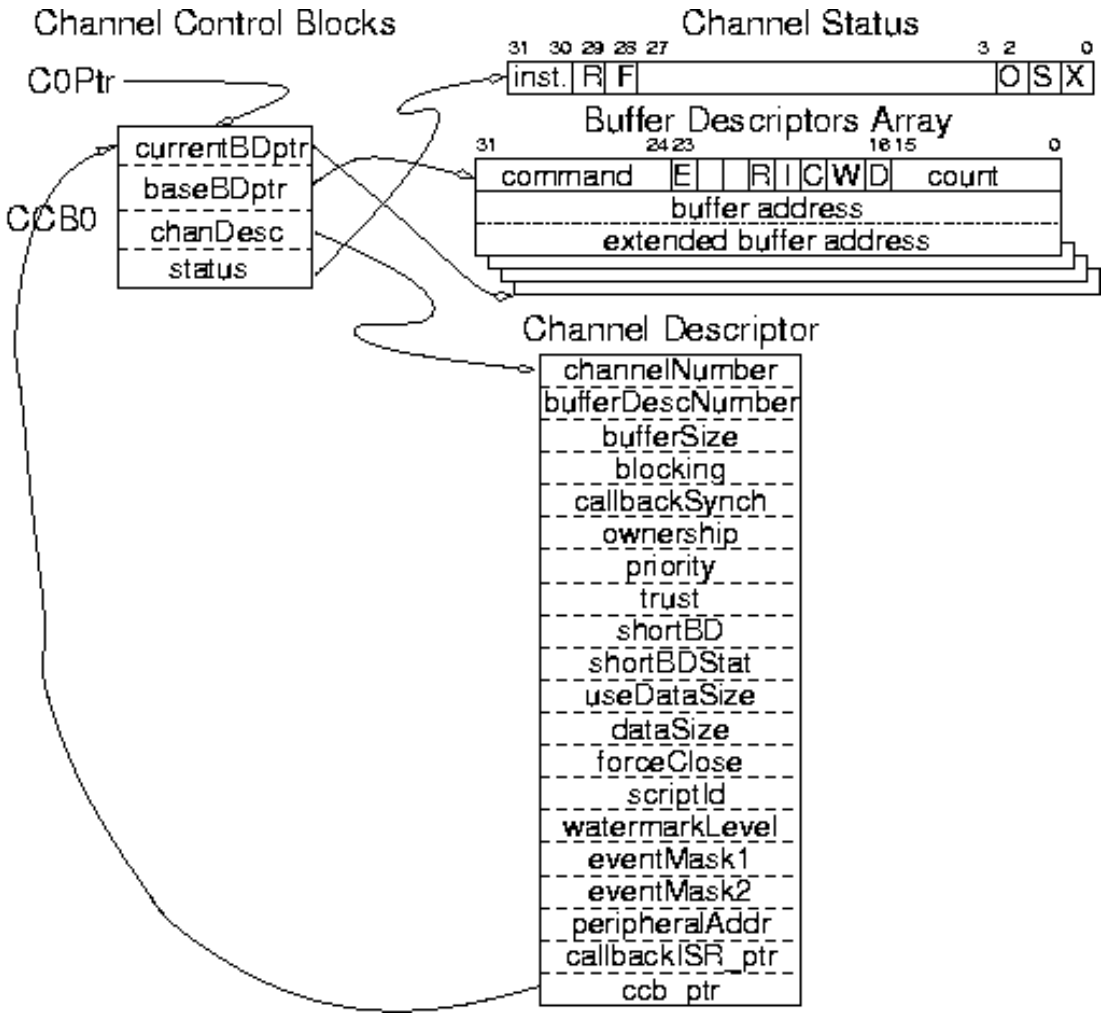


Figure 17: *CCB to Channel Descriptor structure*

But also from the channel descriptor structure view which is the pointer to the device structure passed to all API calls.

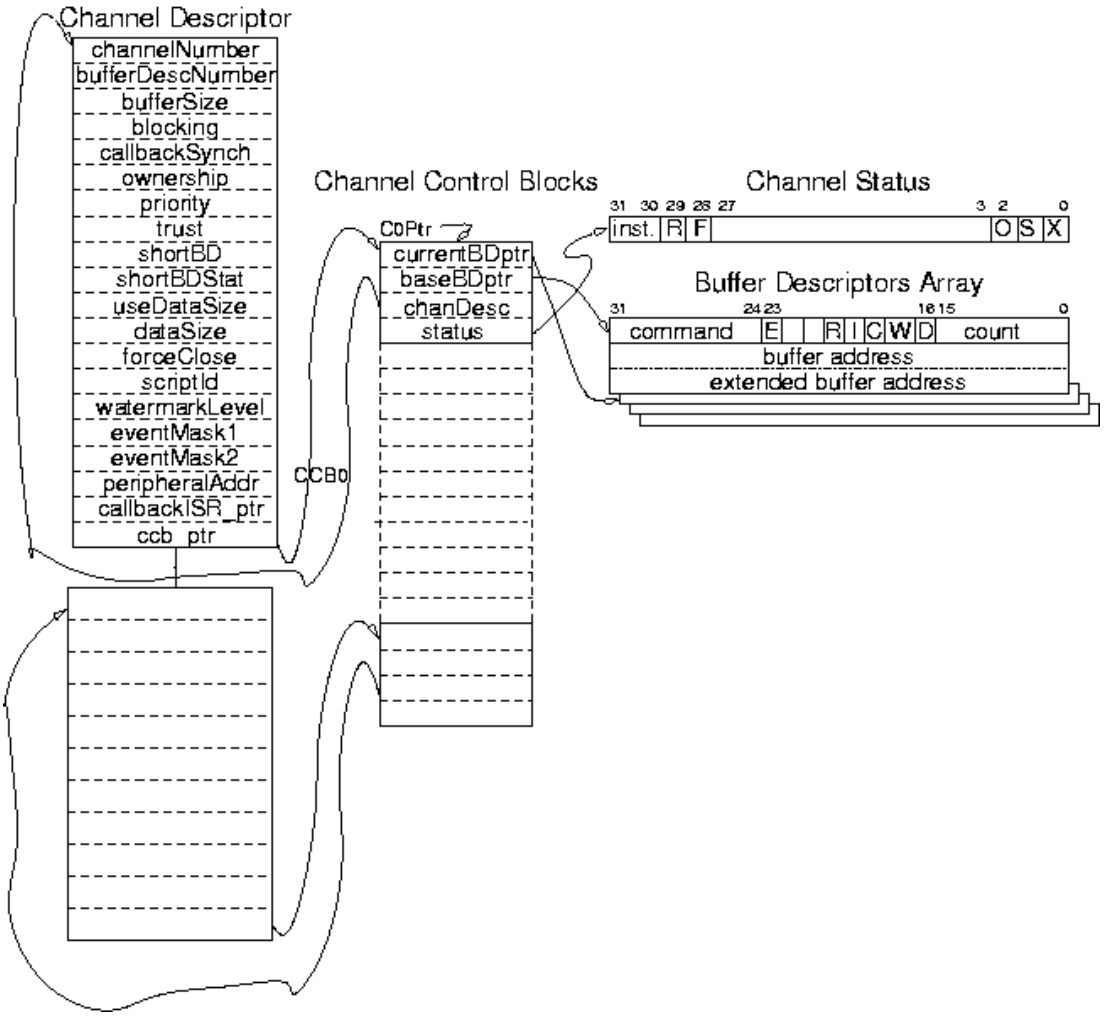


Figure 18: *Channel Descriptors to CCB structure*

Because of the double pointers that allow channel control blocks to point to the channel descriptor and reciprocally the channel descriptor to point to its channel control block counter part, both representations are possible.

int
iapi_Read(channelDesc * cd_p, void * buf, unsigned short nbyte)

The read function attempts to read *nbyte* from the data buffer descriptors associated with the channel, *channelNumber*, into the user's data buffer pointed to by *buf*.

The channel data buffers are pointed to by the channel control blocks and the buffer descriptor structures. The read call copies the data to the buffer address passed as an argument, *buf*.

Read returns the value of **bytes** read.

The data is read until the "D" status bit shows that we processed all buffers for the associated channel OR the number of bytes *nbyte* read, whichever happens first. If the "W" wrap bit is encountered set, the read call goes back to the first buffer and checks the "D" done bit again to establish if the buffer descriptor need to be handled as a ring.

Virtual DMA SDMA channels are unidirectional, an iapi_Read authorized on a channel means that we are expecting to receive from the SDMA. The meaning of an interrupt received from the SDMA is therefore that the data has been copied from the SDMA to the host's data

buffers and is ready to be passed on upper layers of the application.

Nbytes are:	Callback ISR	Default ISR: Poll
present	nbytes are copied from buffer descriptor's data buffers to the buffer pointed to by buf.	nbytes are copied from buffer descriptor's data buffers to the buffer pointed to by buf. Wait for interrupt to be received: poll iapi_IPCMIntr.
not present	bytes available are copied and the call blocks. It waits on the "D" bit to show the SDMA is progressing. It returns nbytes when nbytes have been read.	available data is copied immediately, then iapi_read call polls on iapi_IPCMIntr. If the number of bytes available (and read), after the interrupt was received was not the number of bytes requested nbytes, an error is returned otherwise nbytes is returned.

Table 2: **iapi_read actions**

The number of bytes to be serviced can either be copied by the call (r/w) or not: this depends on the security architecture between the I.API the operating system and the user application. The application being trusted, allows avoiding a level of copy: the pointer passed to the I.API is trusted.

int
iapi_Write(channelDesc * cd_p , void * buf, unsigned short nbyte)

The iapi_Write function attempts to write nbyte bytes from the buffer pointed to by buf to the channel data buffers associated with the opened channel number channelNumber. The write case requires a table to detail the handling performed to start or not the corresponding channel.

The write function performs the loading of the data buffer pointed to by the channel control blocks and the buffer descriptor structures with data provided at the address passed in parameter.

The data is written until the "D" status bit shows that we wrote all buffers available at this time for the associated channel. If the "W" wrap bit is encountered set, the read call goes back to the first buffer and checks the "D" done bit again to establish if the buffer descriptor need to be handled as a ring. In all case *nbytes* indicates the number of bytes really written to the buffer descriptors' data buffers. If "W" is not set and the data to copy is larger than the underlying already allocated buffers, *nbytes* really written is returned.

Virtual DMA SDMA channels are unidirectional, an iapi_Write authorized on a channel means that we are expecting to send to the SDMA. The meaning of an interrupt received from the SDMA is therefore that the data has delivered to the SDMA (read by the script).

Write returns the value of **bytes** written.

Start channell	callback ISR	Default (Poll)
Yes	N/A: channel is not started, ISR cannot get called.	copy data from buf to buffer descriptors data buffers, start channel. Then wait for interrupt: poll iapi_IPCMIntr.
No	The interrupt is sent by script at the	copy from buf to channel's buffer

	beginning of the transfer (on first BD) iapi_write call back is triggered: copy from buf to channel's buffer descriptors, SDMA gets data until "D" bits are all flipped.	descriptors, iapi_write polls iapi_IPCMIntr until interrupt comes. Channel is to be started by another task either before or after the call to iapi_write.

Table 3: **iapi_write actions**

int
iapi_Close(channelDesc * cd_p)

The close function terminates a channel. This operation goes through multiple steps:

- Try to acquire the channel
- If the forceClose flag is not set in the channel descriptor, it checks that all buffers have been processed: all "D" done bits are zeroes reflecting that the host core has performed his job on all of them. If it is not the case, an error is returned and channel is NOT closed (remember that Open checks first if channel was closed). If the forceClose flag is set, it sets all "D" bits to zero and proceeds.
- free allocated memory structures,
- update channel control block pointer to NULL,
- Re-instantiate default interrupts handling if it was modified.
- Release the channel

int
iapi_MemCopy(channelDescriptor * cd_p, void* dest, void* src, unsigned long size)

The number of bytes specified by size parameter are copied from the address dest to address src in the memory of the same processor. The channel must be opened and associated with the appropriate script to perform this operation – MCU_2_MCU or DSP_2_DSP.

This function will set in the BD it uses both the bufferAddr (to the physical address corresponding to src) and extBufferAddr (to the physical address corresponding to dest) fields and start the channel.

The function will block the caller if the synchronization for the channel is set to DEFAULT_POLL, and return immediately if the synchronization is set to CALLBACK.

4.7 Input Output Control of SDMA channel IOCTL

The ioctl() function performs a variety of control functions on devices in the Unix environment. To facilitate the inclusion of the I.API into existing operating systems the same paradigm is taken for the high level I.API functions.

4.7.1 Definition of the call

The request argument and an optional third argument with varying type are passed to the file designated by "fildes" and are interpreted by the device driver (I.API). In the case of the I.API the "fildes" that describe the device is the channel descriptor data structure.

int
iapi_Ioctl(channelDesc * cd_p, unsigned long ctlRequest,
unsigned long params)

The fildes argument is an open file descriptor that refers to an active SDMA virtual DMA channel, in this case the channel number (data type = int). The request argument selects the control function to be performed and for some control codes it may contain, in the upper bits, the number of the buffer descriptor to apply to. The channel descriptor pointer argument is a

pointer to the device specific (virtual DMA channel) data structure used by the ioctl() call to read and modify information on the targeted channel. In the third parameter the caller can pass information relevant to the change or can receive information, case in which it should pass a pointer.

In the case that the call addresses a field from a specific buffer descriptor, the number of the buffer descriptor is encoded in the upper bits of the request. The number of offset bits is defined by the BD_NUM_OFFSET define. In such cases, the function should be called having as a ctlRequest parameter a construction like the following:

(CHANGED_BD << BD_NUM_OFFSET) | IAPI_CHANGE

Table 4: **List of ioctl() commands**

Control code	Action
IAPI_CHANGE_CHANDESC	Changes the pointer to the channel descriptor structure: the pointer to the new channel descriptor is given in the third argument of the call. Check for sanity of the information is performed then a bounce sequence on the channel is performed (close frees the old structures).
IAPI_CHANGE_BDNUM	Changes the number of buffer descriptor for the channel.
IAPI_CHANGE_BUFFSIZE	Changes the size of the data buffers pointed to by the buffer descriptor; note that all buffer descriptors are assumed to have the same size for a given buffer descriptor chain.
IAPI_CHANGE_OWNERSHIP	Changes the ownership of the channel - the new value is received in the third argument.
IAPI_CHANGE_SYNCH	Changes the sychronization method for the channel: DEFAULT_POLL or CALLBACK_ISR
IAPI_CHANGE_TRUST	Changes the trust for the memory buffers received from the user.
IAPI_CHANGE_CALLBACKFUNC	Changes the callback function pointer, when this method is used, to replace it with a new one.
IAPI_CHANGE_PRIORITY	Changes the priority of the channel.
IAPI_CHANGE_BDWRAP	Change the BD_WRAP bit on the last DB from the BD array
IAPI_CHANGE_WATERMARK	Change watermark level in CD
IAPI_CHANGE_SET_BDCONT	Set the BD_CONT bit on BD specified in param argument or all BD's from the BD array.
IAPI_CHANGE_UNSET_BDCONT	Unset the BD_CONT bit on BD specified in param argument or all BD's from the BD array.
IAPI_CHANGE_SET_BDEXTD	Set the BD_EXTD bit on BD specified in param argument or all BD's from the BD array.
IAPI_CHANGE_UNSET_BDEXTD	Unset the BD_EXTD bit on BD specified in param argument or all BD's from the BD array.
IAPI_CHANGE_EVTMASK1	Change event mask 1 in CD
IAPI_CHANGE_EVTMASK2	Change event mask 2 in CD
IAPI_CHANGE_PERIPHADDR	Change peripheral address in CD
IAPI_CHANGE_SET_BDINTR	Set the BD_INTR bit on BD specified in param argument or all BD's from the BD array.
IAPI_CHANGE_UNSET_BDINTR	Unset the BD_INTR bit on BD specified in param argument or all BD's from the BD array.
IAPI_CHANGE_SET_TRANSFER_CD	Change the transfer size indicator from the CD to the value passed in param.

IAPI_CHANGE_FORCE_CLOSE	Change the forced close indicator from the CD to the value passed in param.
IAPI_CHANGE_SET_TRANSFER	Set the value passed as the transfer size in the command field of all buffer descriptors of the channel
IAPI_CHANGE_USER_ARG	Change the user argument registered to be received in the user provided callback function
IAPI_CHANGE_SET_BUFFERADDR	Set the buffAddr field of the buffer descriptor specified in the upper bits of ctlRequest parameter to the value passed in the param. The value is translated from virtual to physical
IAPI_CHANGE_SET_EXTDBUFFERADDR	Set the extBuffAddr field of the buffer descriptor specified in the upper bits of ctlRequest parameter to the value passed in the param. The value is not translated from virtual to physical.
IAPI_CHANGE_SET_COMMAND	Set the command field of the buffer descriptor specified in the upper bits of ctlRequest parameter to the value passed in the param. The value is translated from virtual to physical
IAPI_CHANGE_SET_COUNT	Set the count field of the buffer descriptor specified in the upper bits of ctlRequest parameter to the value passed in the param. The value is translated from virtual to physical
IAPI_CHANGE_SET_STATUS	Set the status field of the buffer descriptor specified in the upper bits of ctlRequest parameter to the value passed in the param. The value is translated from virtual to physical
IAPI_CHANGE_GET_BUFFERADDR	Retrieve the buffAddr field of the buffer descriptor specified in the upper bits of ctlRequest parameter in the unsigned long value pointed by param. The value is translated from physical to virtual.
IAPI_CHANGE_GET_EXTDBUFFERADDR	Retrieve the extBuffAddr field of the buffer descriptor specified in the upper bits of ctlRequest parameter in the unsigned long value pointed by param. The value is not translated from physical to virtual.
IAPI_CHANGE_GET_COMMAND	Retrieve the command field of the buffer descriptor specified in the upper bits of ctlRequest parameter in the unsigned long value pointed by param.
IAPI_CHANGE_GET_COUNT	Retrieve the count field of the buffer descriptor specified in the upper bits of ctlRequest parameter in the unsigned long value pointed by param.
IAPI_CHANGE_GET_STATUS	Retrieve the status field of the buffer descriptor specified in the upper bits of ctlRequest parameter in the unsigned long value pointed by param.
IAPI_CHANGE_SET_ENDIANNESS	Set the endianness bit of the command field to ask for a software swapping of the data (AP/BP in a different endianness) for a transfer involving both processors.

All the control changes presented above are supported by calls to the lower levels of the I.API library, the input-output control call manages calls to these lower level routines. The ioctl() call performs a check whether or not the requested control can be performed on the channel in its current state. For example the control request that affect buffer descriptor number or size can only be performed on an initialized but not running channel.

Upon successful completion, the value returned is IAPI_SUCESS. Otherwise, the negated

value of the error code is returned .

Bounce sequence

The bounce sequence is used for the change channel descriptor input output control. It is used to make sure the channel is properly closed (stopped, de-allocated) and restarted with the new channel properties.

- Stop the channel
- Depending on the trust policy free IAPI data buffers
- Free IAPI buffer descriptors
- Clear channel descriptor structure and update with new values
- Allocate and update new buffer descriptors and data buffers (depending on trust) with new parameters
- Update channel descriptor and channel is restarted.

The bounce sequence re-starts the channel. It does not allow changing the channel SDMA script for the controlled channel. This is because the event assignment and functionality for a channel are established for a platform. The user can perform the equivalent operations to change a script using the other ioctl and managing the download, but care must be taken on the management of the SDMA program memory (please see [10] for requirement and discussion).

Error cases at the high level:

- Bad channel number: the channel number is not active our out of range.
- Input output control function was interrupted by a signal pertinent to the channel being modified: control operation has aborted.
- The channel descriptor pointer is not valid
- The information in the channel descriptor data structure is not compatible with the control operation requested
- Control operation failed on an IO to the SDMA (transmitting changes to the SDMA).

4.7.2 Pointer Trust:

The pointer trust allows the calling application to allocate its own buffers and pass the pointer to the IAPI buffer descriptor; provided it passes it through a properly defined buffer descriptor structure (preferably using existing IAPI calls).

This capability is given to take full advantage of the direct memory transfer capability of the SDMA. Attention must be paid to the software architecture for the tasks calling the IAPI:

- RTOS managed user/supervisor space
- All application under the same permissions

In the first case the tasks can dialog through shared memory segment for example. Depending on the support from the RTOS or custom shared memory segment handler, pointers given from these memory areas could be considered unsafe. In the second case, all applications could be considered either safe or unsafe.

In the case the pointers are considered unsafe, the copy is performed by the IAPI from the user buffer to the IAPI allocated buffers, since the IAPI is compiled with the RTOS (possibly loaded) the IAPI performs the move from user to supervisor. This move is done at the expense of one more copy.

On the other hand if the pointer is trusted the transfer takes place immediately, without the

extra copy, which is particularly relevant in the case the SDMA is used to perform memory to memory transfer into the same core memory space.

Additionally, one advantage of DMA transfer is to offload the main core from memory move. Therefore the transfer operates in an asynchronous manner for the requesting core and the synchronization mechanism used is the interrupt. This end of transaction interrupt takes place at any time. From the application level, the constraints are:

- If pointers are trusted, the application must maintain the pointers and data buffer area until the end of transaction is received. During this length of time, the buffers can be corrupted.
- If the pointers are not trusted, the IAPI can perform the additional copy and freeing. But one additional copy can be overly expensive for the transaction standpoint.

Because the system on chip that are using the IAPI have to perform multiple DMA transactions, the default of the IAPI is to TRUST the buffer descriptor pointers.

.

4.8 Hosts Interrupt Handler designs

This paragraph is of special interest if one chooses to write the SDMA IRQ handler and not use the one provided by the I.API.

In order to support both methods of Virtual DMA synchronisation explained in the previous chapter, the interrupt handler needs to have some facilities programmed in. The IRQ handler provided by the I.API should be used as a starting point.

The IRQ Handler must do the following in order to allow proper functioning of the I.API:

- check in the relevant SDMA register the channels requesting service from the processor (Host_Interrupt register) and for every one of these
- clear the interrupt by writing the corresponding bit in the Host_interrupt register.
- wake up the corresponding channel
- if a callback function is registered for the channel, release the channel and call the function with the proper arguments (channel descriptor pointer and user registered argument)

4.9 SDMA API Variables

The variable defined below are used by various routines of the SDMA API library to check the state of the hardware or the API itself, these are private variable to the API.

Variable Name	Description
iapi_errno	Error number for internal IAPI failures, this can be operating system dependant. If it is provided in the environment, the error should be copied to the accessible variable of the operating system.
callbackIsrTable **	Table of interrupt service routine function pointers indexed by channel number. The initialization points to the default ISR routine

	and can be updated by attach/detach.
userArgTable**	Table of user pointers registered to be passed as parameters to the callback function, indexed by channel number. At initialization the value is NULL and can be updated by IoCtl calls

Table 5: **SDMA API Variables**

In the case multiple SDMA are found on the platform, the instance bits of the channel control block status field are used. The creation of instances after the initial first one is handled with an open call followed by an ioctl to change the instance. The following IAPI calls carry, in the channel descriptor structure, the pointer to the channel descriptor that allows to identify the instance of the SDMA the operation applies to.

The variables listed here characterize an SDMA instance and can be maintained in a array in case of multiple instance as suggested for the ISR table.