Freescale Semiconductor
Application Note

AN2604
Rev. 1, 08/2005

Introduction to BestComm

by: Davide Santo

1 BestComm

BestComm refers to the integration of a specific DMA engine, a set of communication-oriented peripherals, a local bus and different FIFOs embedded within the MPC5200 processor.

The BestComm DMA engine belongs to the wider family of the SmartDMA engines and, in fact, embeds only a limited set of features of the latter. It has been designed to best integrate within the MPC5200 architecture to achieve the maximum possible performance.

BestComm is a powerful and flexible DMA engine, enabling transparent data movement between different peripherals and the SDRAM with minimal CPU intervention.

Even if the BestComm main task remains data movement, it can also provide, if needed, *minimal* data handling, such as buffer descriptor support, limited CRC algorithms, endian byte swap, etc.

Table of Contents

1 BestComm 1
1.1 BestComm Features 2
2 BestComm Architecture 8
2.1 BestComm Hardware 8
2.2 BestComm Microcode 14
2.3 The BestComm API and Task Builder 21
3 BestComm Debugging 27
3.1 BestComm, The Core and the Data Cache . 29
4 BestComm Limitations 31
5 Performance Analysis 32
5.1 Initial Latency 33
5.2 BestComm Efficiency 34
5.3 XLB Bandwidth Usage 34
6 Conclusion 35

This application note is intended for those curious and bold designers (both hardware and software), who would like an introduction into this area of the MPC5200 processor.

NOTE

For more detailed information, please refer to the MPC5200 Users Guide (MPC5200UG).

1.1 BestComm Features

BestComm is a high-performance user-programmable DMA engine that uses c-like for-loops descriptors to move and manipulate data.

MPC5200 implementation comprises a generic logic unit inclusive of a small checksum unit, called Logic Unit with Redundancy (LURC), and 16 kB of internal SRAM. It does ***not*** have any Multiplier and Accumulate (MAC) units.

BestComm decouples code parsing from data operation, thus realizing the above mentioned high performance data movement. This implies an evident hardware acceleration because the CPU is off-loaded from these tasks.

In order to achieve this goal, every BestComm supported peripheral, with the exception of I^2C, is equipped with at least one FIFO (and, in many cases, two to allow ‘full-duplex’ data movement) and its associated FIFO controller. The function of the FIFO is exactly that of decoupling the peripheral’s speed, specific signalling and timings from those of the internal core bus (called XLB**.**) *See Note below.*

NOTE: MPC5200 Bus Nomenclature

The MPC5200 processor has three main internal and two external buses.

The core (a derivative of the PowerPC® 603e) bus is called XLB (from 60x Local Bus). This bus is a 64-bit bus with a specific interface protocol to it. All the possible internal or external bus masters (the core, USB, PCI, and BestComm) can access other peripherals (especially the local memory) via this bus. Its maximum speed is set at boot via the Power On Configuration Word and can reach 132 MHz over the entire temperature range.

The Internal Peripheral (IP) bus connects the core to all the peripherals, including those supported by BestComm and the internal SRAM. This is a 32-bit bus. Its frequency is configurable to be the same or half that of the XLB.

The Communication Bus (CommBus) is the specific bus used by BestComm only to communicate directly with the peripherals that are supported by the DMA engine. It is a 32-bit bus, and its frequency is always set to the same as that of the IP bus.

The PCI bus is shared externally by the ATA, Local Plus and PCI peripherals. It is a 32-bit bus, and its maximum frequency is 66 MHz.

The last external bus is the Dynamic Memory bus used to interface to SDRAM or DDR memories. This is a 32-bit bus running at a maximum of 132 MHz.

Within the MPC5200, BestComm supports up to 16 simultaneous tasks, with up to 32 "requestors."

A task is a single program executable by BestComm. Only one task can execute at a time, so task switching (including all associated operations such as arbitration, context saving and restoring) is an important feature of BestComm.

A task is composed by two basic instruction types: a Loop Control Descriptor (LCD) and a Data Routing Descriptor (DRD). These are 32-bit words, whose bits are parsed by the BestComm engine to determine which operation must be executed. On an LCD, indexes are initialized, while, on a DRD, they are used to point at different source and destinations to effectively move the data.

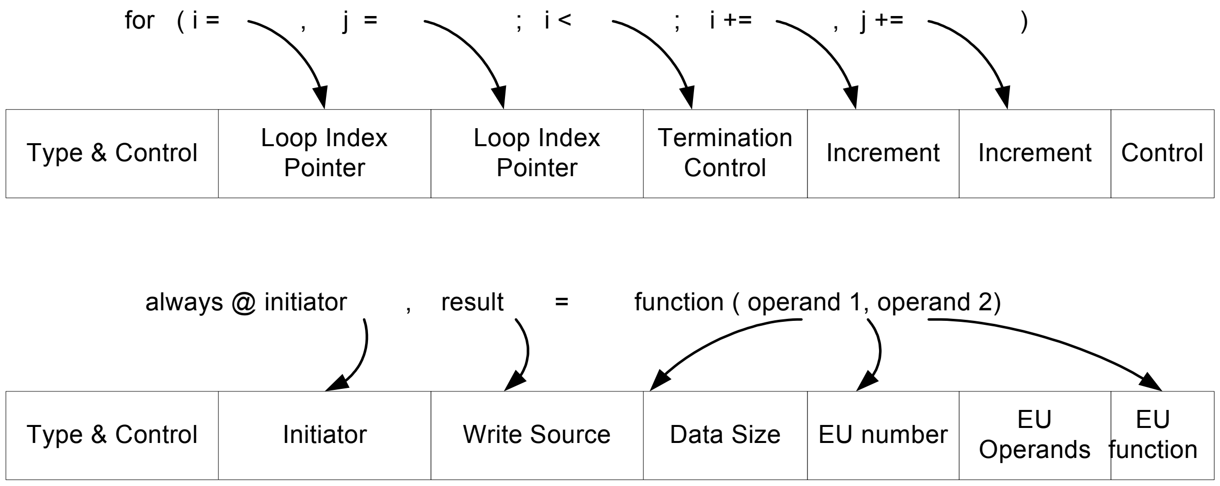

Figure 1. LCD and DRD

A "requestor," also called "initiator," is a hard wire inside the MPC5200, connecting a peripheral FIFO to the central DMA engine (i.e., BestComm itself). Its function is simple: when asserted, it requests (or initiates) a DMA transfer.

This hard-wired line is neither under direct software control, nor is externally accessible; therefore, no external DMA request input pin is available on the MPC5200. The requestors are instead handled (asserted / de-asserted) automatically by a FIFO controller as a function of two internal watermarks (called Alarm and Granularity). The user can, via software control, set the position of these watermarks and fix the priority of the requestors.

BestComm supports *simultaneous* operations on up to 12 sources or 11 sources and 1 destination. All the internal operations are single-clock design, so that any BestComm task would theoretically need no wait states when it is executed.

Reality, on the contrary, is not always so bright; as overhead, some clocks are always needed for the parsing of the tasks, the arbitration and eventually the save and restore phases. Once a task is running, each data movement or operation happens within the engine in one clock; but wait states might be inserted via the XLB bus by the addressed slave, indicating its temporary unavailability for data transfer (e.g., the DRAM controller cannot write or read data in a single clock beat).

To complete the picture, it is important to note that BestComm is one of the possible masters of the XLB, and it can burst on this bus. A burst is composed of four beats (64 bits wide each), thus moving 32 bytes per burst in either direction. (In MPC5200, a word is by definition 32 bits long). This corresponds to a PowerPC 60x cache line size.

In case the memory locations interested by the BestComm accesses are cached in copy back mode by the core, a 'Snooping' activity can be forced to the core, so that an 'address retry' request will be issued, and the Core will first flush its cache line before allowing BestComm to retry its transaction. All of these features are transparent to the task's programmer. The user is allowed to change the XLB arbiter priority and to eventually park the XLB bus on the BestComm master.

NOTE: XLB Arbiter priority

It is generally good advice to set the core and the BestComm at the same priority level right before starting to use the DMA engine. If the core is set to higher priority, a higher risk of bus starvation is run since the core is faster and, therefore, might request the bus more often.

A special feature of BestComm is the way of writing a task: BestComm uses c-alike code (not real *C* but something resembling it). Normally it is necessary to simulate the written tasks to avoid problems and to do a sanity check. Because this option is not directly available to the customer, a BestComm Graphical User Interface (BestComm Task Builder) is provided to produce tasks. The Task Builder allows the user to assemble a set of tested tasks (called tasks *image*), which can be assembled and linked into a software project.

NOTE

BestComm can run over the extended temperature range at the maximum IP bus speed, which is 132 MHz for the MPC5200. This allows the usage of BestComm in conjunction with a DDR memory subsystem.

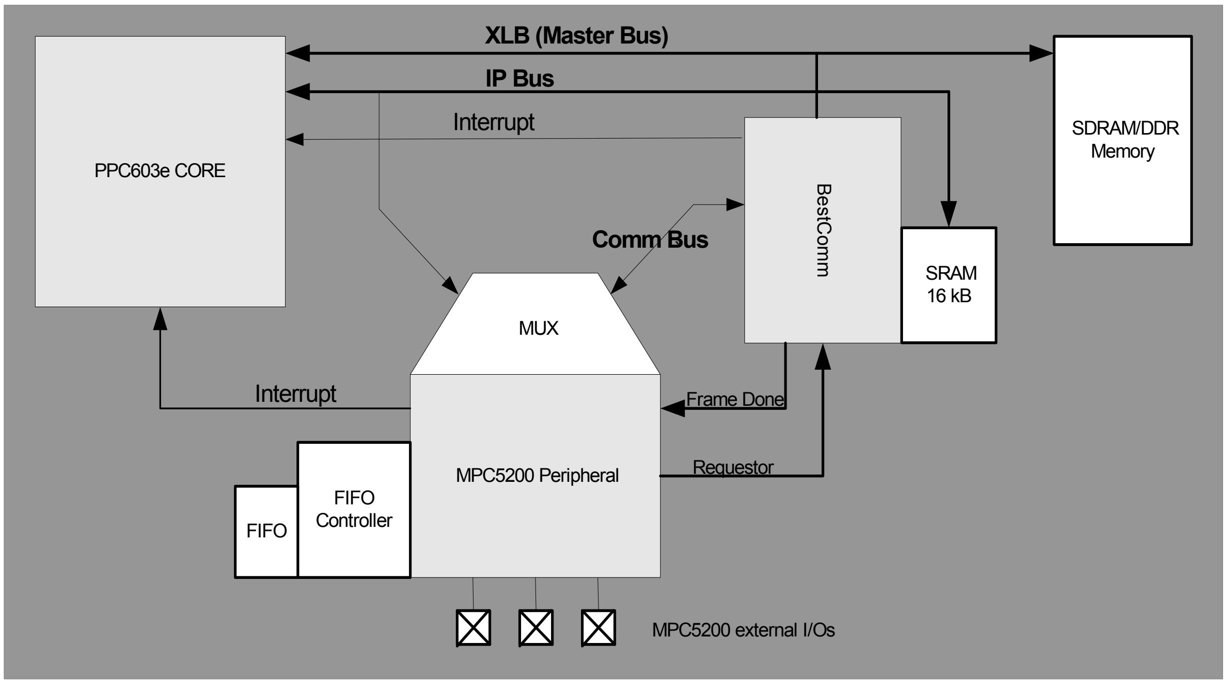

Figure 2. BestComm's Architecture

There are different hardware modules that interact during a DMA data transfer:

1. The BestComm main engine is composed of different subsystems, among which are the following:
   a.) The LURC is the only implemented execution unit responsible for the boolean operations and the CRC computation. (Specifically, it is referred to in the code as execution unit 3.)
   b.) The Master DMA En coder (PTD) is responsible for the prioritization of the requestors and of the tasks. The user has indirect control of the priorities of the initiators/tasks via the proper IP registers interface.

NOTE: Internal BestComm Modules

It is not essential to understand what each internal submodule exactly does, but their simplified descriptions have been added here for completeness and to allow the reader to picture how the BestComm engine is organized to better understand its limits and strengths.

2. The 16 kB of internal SRAM: this memory is used to store the task code, buffer descriptor table, function descriptors, local variables and context saved variables. The content of SRAM is kept alive during sleep mode as long as power supply is properly supplied. BestComm has direct access to the SRAM, and data are moved to and from it in a single clock cycle.
3. The FIFOs and FIFO controllers: each supported peripheral with the exception of I^2C has at least a FIFO (half-duplex), if not two (for full-duplex). FIFOs and their associated controllers are different for each peripheral: some are single-port ram; some are dual-port ones, and the FIFO controller might have slightly different behavior.
4. The peripherals themselves: each one has a BestComm dedicated programming section, which is different from peripheral to peripheral.

1.1.1 Bestcomm Supported Peripherals

BestComm supports 11 different functions spread over seven different peripherals. These are reported in the following table:

BestComm supported peripherals

Peripheral	Function	FIFO(s)
PCI	PCI	2 x 512 bytes (“full duplex”)
Local Plus (LP)	Memory mapped devices	1 x 512 bytes (“half duplex”)
ATA	ATA, True-IDE PC cards	1 x 512 bytes (“half duplex”)
I^2C	I^2C	No FIFO
Ethernet	10Base-T and 100Base-T Ethernet	2 x 1024 bytes (“full duplex”)
PSC	Audio Codec	2 x 512 bytes (full duplex)
	I^2S	2 x 512 bytes (full duplex)
	AC97	2 x 512 bytes (full duplex)
	SPI	2 x 512 bytes (full duplex)
	UART	2 x 512 bytes (full duplex)
IrDA (PSC6)	IrDA	2 x 512 bytes (full duplex)

Each FIFO and FIFO controller has different user-accessible registers, which are mostly useful during the debug phase of a BestComm task. Information such as FIFO overflow or underrun, alarm reached and the position of the FIFO read or write pointer are readable by software at any time. Some FIFOs can generate interrupts (namely Ethernet, PCI and LP) responding to different errors. In some cases, the FIFO controller can even block the peripheral if there is no data available in the FIFO to be transmitted. (PSC, PCI, LP and ATA work in this fashion.)

A special case is the local dynamic memory. This particular slave periphery is directly interfaced to the internal XLB. The BestComm, as with every other master, can therefore require access (read or write, in Burst or single beat) to the memory. It can eventually perform memory-to-memory data movement adding, if needed, simple operations such as CRC, Endian swap, masking and so on.

1.1.2 Initiators

There can be up to 32 initiators (also called requestors) active at any given point in time. Each task’s descriptor (specifically the DRD) must be conditioned by a requestor. In other words, the initiator acts as a semaphore: when activated, the associated descriptors can be executed (if the task is currently in the “run” mode). If the requestor is de-asserted, the task’s context will be saved, and it will enter an idle period, giving up its usage of the BestComm engine. The exception to the rule is the so-called *always* requestor (requestor number 0), which is always active. By default, a descriptor not specifically assigned to a requestor will be conditioned by the *always* requestor.

The pool of requestors is divided into two sections: 16 requestors are fixed, assigned to a specific peripheral. The second group of 16 can be selected via software between two sets, one of which is made of only *always* requestors. Each initiator has a fixed number, which will be encoded in the DRD. To set a requestor's priority, the Initiator Priority Register (IPR) bearing the same number must be used (highest priority is 7, lowest is 0).

It is possible to switch from a requestor-driven priority scheme to a task-position priority scheme by setting bit 16 of the MPC5200's internal register at MBAR + 0x1210. In this case, the same registers used to select the priority for a given requestor (identified by its unique number) will do the same for a given task (whose number is associated with its position within the tasks' image).

The following table contains a complete list of requestors. The upper 16 are the defaults.

Table 1. Requestor List

Requestor Name	Requestor Number
Always Requestor	0
Reserved	1
Reserved	2
FEC RX (Fast Ethernet Controller)	3
FEC TX	4
ATA RX	5
ATA TX	6
PCI RX	7
PCI TX	8
PSC3 RX	9
PSC3 TX	10
PSC2 RX	11
PSC2 TX	12
PSC1 RX	13
PSC1 TX	14
Reserved	15
Local Plus	16
PSC5 Rx	17
PSC5 TX	18
PSC4 RX	19
PSC4 TX	20
I^2C-2 RX	21
I^2C-2 TX	22
I^2C-1 RX	23
I^2C-1 TX	24
IrDA (PSC6) RX	25
IrDA (PSC6) TX	26
Reserved	27

Table 1. Requestor List (continued)

Requestor Name	Requestor Number
Reserved	28
Reserved	29
Reserved	30
Reserved	31

NOTE: Initiator and DRD

It is possible in the DRD to spot the number of the initiator at least in the readable comment appended to the binary word by the assembler, e.g.,

.long 0x011ec798 /* DRD1A: *idx1 = *idx3; FN=0 init=8 WS=3 RS=3 */

where the *init* = *8* signals that the relative DRD is conditioned by the PCI TX requestor.

2 BestComm Architecture

The complexity of the BestComm system and its interaction with the supported peripherals cannot be included in a few lines. The scope of this section is to review the basic elements, which make it possible to transfer data using BestComm, focusing on those aspects that are more likely to cause confusion or potential errors. No detailed description of the BestComm core or structure is provided even if some hardware details are at times treated.

2.1 BestComm Hardware

A BestComm task is executed by cooperation of different hardware modules:

1. BestComm's DMA engine
2. SRAM internal memory
3. FIFOs and the FIFO controllers
4. Supported peripherals and dynamic memory

Each component plays an important role and can be a challenge for the user during the debugging process. Many times a programming error or misuse may lead to an incorrectly functioning task.

2.1.1 The BestComm Core

The BestComm core is responsible for the following:

1. Interpreting the task code. The code is fetched from the local SRAM memory. First the task descriptor table is parsed to detect where each task's code and associated variables are located. Then the task code itself (LCDs and DRDs) are parsed to the point where all the necessary information to execute a DRD is gathered.

2. Execution of the task. When the proper initiator becomes active for the currently loaded task, the associated DRD is executed. This implies some kind of data movement or a specific operation on a given datum. Every operation might have a different execution time since it depends on the associated peripheral and the state of the XLB (busy or free).
3. Arbitrating between competing tasks. When more than one task has been enabled, the BestComm will check either the current requestors priority or the task priority to decide which task will be the next one to enter the execution phase. It is important to understand that incorrectly setting the requestor priority can greatly influence the behavior of the system. Associated to the arbitration phase, there is often the save and restore phase, where the context of a suspended task is saved or restored from the SRAM.

The BestComm core accesses the peripherals' FIFO via the CommBus (32-bit access); on this bus, a prefetch buffer on a read operation is provided to improve performance on the bus.

Some peripherals (e.g., Ethernet or ATA) are very timing-sensitive; the prefetch buffer might create issues and can always be disabled via software control.

Readline buffers and *Write combined buffers* (eight 32-bit wide each) are provided to allow bursting to and from the XLB (read/write operation). Both buffers can be disabled.

BestComm can also do what is technically defined as speculative reading from the XLB bus. This feature requires that, during a read operation from any XLB bus address, the successive address location is automatically read into a temporary buffer, speculating that it will be more likely to be used during the DMA transfer. This feature can be disabled in the same way as for the readline buffer.

BestComm can read and write the SRAM directly, and the BestComm control registers are accessible by the core via the IP bus.

BestComm's core acts as the Master of the XLB, which is not accessible or controllable directly by the user, but its role is essential since all the data which is written to or read from the dynamic memory pass through it.

2.1.2 The SRAM

The SRAM (16 KB) is an internal memory always accessible from BestComm. At run time, it is used to store the task image, task variables, task functions and the task context save areas. This covers about 25-33% of the total SRAM. The rest is available for the user. It is recommended that when using buffer descriptor based tasks, the buffer descriptors table should be located in this memory. BestComm can read and write the SRAM in a single clock cycle.

It is possible at all times to read or write the SRAM using the core.

It is often necessary to check directly in SRAM whether the variables passed by the upper layer of the software are correctly written.

NOTE: SRAM Usage

The internal SRAM can be accessed by the core via the IP bus. Its primary role is to act as storage space for BestComm, but it can be used for all kinds of storage. Execution from SRAM is possible but discouraged.

The Buffer Descriptor Table could also be theoretically located in SDRAM (or in other storage capable area), but this setup is strongly discouraged.

2.1.3 FIFO and FIFO Controller

Each peripheral supported by BestComm has one or two FIFOs with associated FIFO controller. Not all the FIFO controllers behave the same, and not all FIFOs are the same. Size, for instance, may change. The function of the FIFO is to decouple the external peripheral, its operating frequency and specific handshake or protocol signals from the BestComm core and any other internal peripheral, such as the memory. The FIFOs are fixed in size, and they are directly accessible from the core, but every read or write access is destructive.

The FIFOs' size is normally much smaller than the amount of data which is requested to be transferred through it. In a video task, a 512-byte-wide FIFO will handle the complete frame. This works because BestComm runs at a faster or at least as fast a frequency as the controlled peripheral.

BestComm will transfer the data in clusters whose maximum size is limited by the FIFO size itself. To help better control the data movement, two *watermarks* are available called *alarm* and *granularity,* and their value is directly under user control. This is the only way for the user to influence the FIFO's behavior.

Within each peripheral's programming model, there are other registers related strictly to the FIFO, and they are useful during the debugging phase. Especially interesting are those registers that reflect potential FIFO errors (such as overflow or underrun) and those that indicate the current position of the FIFO-read and FIFO-write pointers. It is very common that when a BestComm task gets stuck, looking at these registers indicates what kind of livelock has been created. For instance, the BestComm is not moving data, or the problem lies within the peripheral itself.

Some peripherals (PCI, LP, Ethernet) can invoke an interrupt if a FIFO error is detected.

2.1.3.1 Alarm and Granularity

The alarm threshold is the level at which a FIFO needs to be serviced by BestComm. When a FIFO alarm is reached, the corresponding FIFO initiator is asserted. It can be set anywhere from 0 to the full FIFO size (in bytes).

When the alarm is reached, another task may currently be using the BestComm core. This active task must then first release the DMA engine, allowing the engine to rearbitrate. At arbitration time, if there is no other pending request with higher priority, the FIFO will be serviced.

It is obvious that the latency at which BestComm answers a request is strongly dependent on the current application and on the internal resource usage. It is almost impossible to predict when BestComm will start moving data to or from the FIFO; nonetheless, this is not influenced by all the peripheral that continues, eventually causing overflow or underrun. It is then up to the user to balance the task's priority and to experiment with the alarm level to avoid a single task from monopolizing the BestComm.

The granularity represents the level at which BestComm will stop servicing the FIFO. The granularity is a fixed number between 0 and 7. It is suggested to set the granularity to its maximum value.

It is interesting to point out the different interpretations that can be given to the alarm and granularity for a TX FIFO versus a RX FIFO.

The alarm level of a TX FIFO is expressed as the number of bytes remaining in the FIFO. At the beginning of operation, when the TX FIFO is empty, the FIFO's requestor is always active, and BestComm will move data into it. When the FIFO is almost full, at the granularity level, BestComm will halt. In the meantime, the peripheral will have started emptying the FIFO. When the FIFO has been read, such that a number of bytes equal to the alarm are left, BestComm will be requested to refill it.

The alarm level of a RX FIFO represents the number of free bytes available in a FIFO.

Beginning a read, the RX-FIFO is empty, and no request is active from the FIFO side. The peripheral fills it until the number of free bytes remaining in the FIFO is equal to the alarm (FIFO almost full). At this stage, the requestor goes active, and BestComm can empty the FIFO until the granularity level (FIFO almost empty) is again reached.

NOTE: Granularity and Alarm

The granularity number is measured in words, while the alarm is expressed in bytes. BestComm will halt one word before reaching either one of the two conditions.

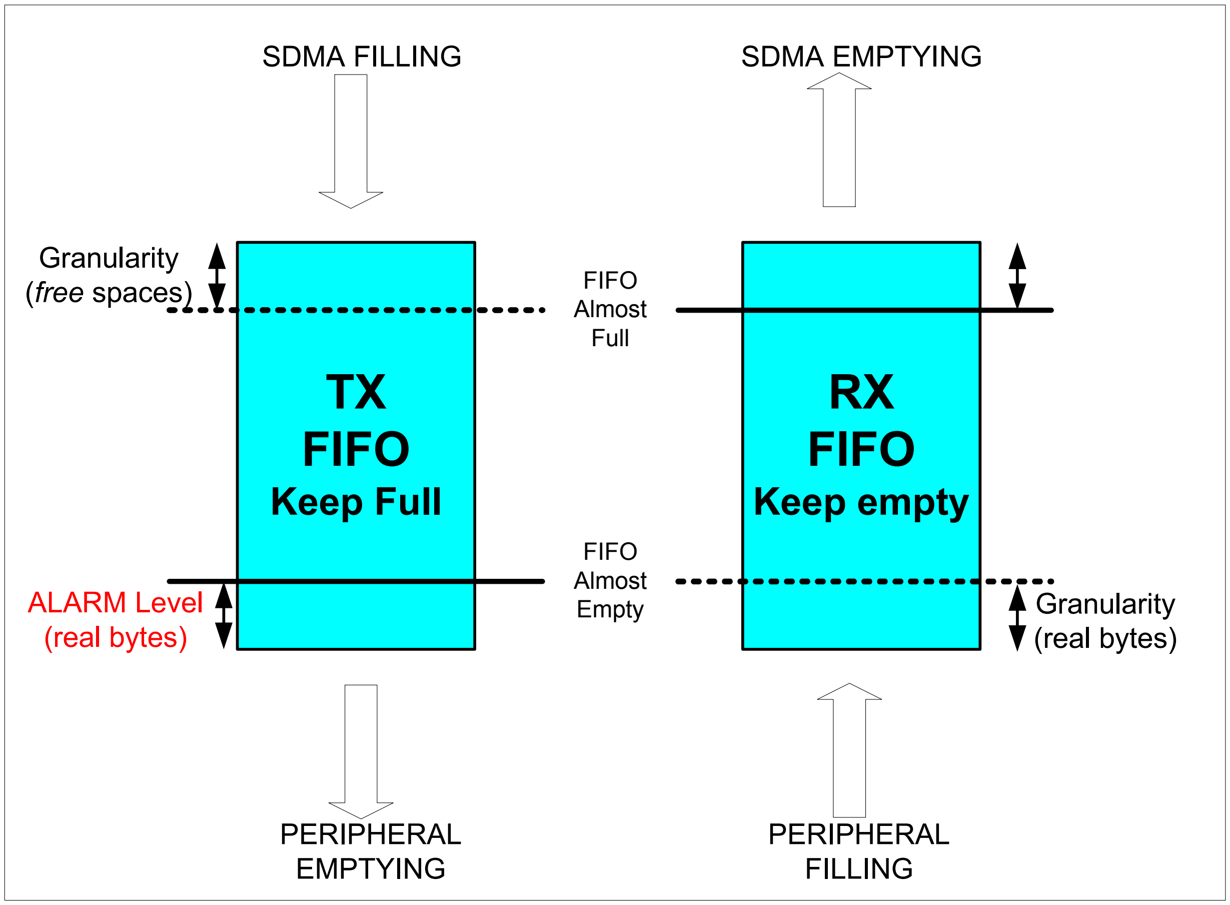

Figure 3. Alarm and Granularity

A hysteresis can be visualized between the two thresholds. BestComm will move blocks of data, whose size equals the distance between the alarm and granularity.

A special case appears just for the receiving FIFO when the task is moving the last pieces of data. In such a case if the granularity level is reached (so that BestComm halts) and the remaining data filled in by the peripheral do not reach the alarm level, stale data will remain in the FIFO. To avoid this problem (specifically for the PCI, ATA and LP peripherals), a *flush* bit is provided in the programmer's model. By setting this bit, the BestComm core will be forced to ignore the assertion of the granularity when the number of bytes left to transfer is less than the alarm level, thus avoiding this issue.

Some working examples are provided below to stress how important the correct settings can be for the alarm and granularity levels.

Let's assume that a fast receiving PCI task has been enabled (for a video application for distance). The task reads at a 66-MHz-long burst of words. BestComm has to move the data into the local DDR memory. Let's assume BestComm's bus frequency is set to 132 MHz.

At the same time, the alarm level is set to 32 (bytes) for the receiving FIFO. This means that when in the PCI RX FIFO 476 bytes (512 - 32) are written by the PCI module, the internal alarm level is asserted, requesting BestComm to empty the FIFO.

Let's assume the Core (PPC 603e running at high speed, 396 MHz) is accessing the XLB to fetch data it needs for its internal operations.

Given these conditions, it is highly probable that an overflow will be detected at the FIFO. In fact, as the request for BestComm intervention is detected, some clocks are needed by the BestComm's core to restore the previously suspended task and to enter the run phase (not to mention the case where another task is active). In the meantime, the PCI module keeps on filling the FIFO (getting close to the FIFO upper limit).

If, moreover, the XLB is not immediately available because the core is using it, BestComm access to the bus will be delayed. The FIFO can easily overflow.

A simple solution for this case is to increase the FIFO alarm, so that there is more time allowed for BestComm to start emptying the FIFO.

Another area to keep under control would be allowing the XLB arbiter to prioritize BestComm versus the core. (Because the core is faster, it tends to overload the XLB.)

A second example refers to those peripherals like PCI (or LP), where the number of words (or Bytes) per burst can be selected by the user. (In PCI, for example, up to eight words per burst can be handled, for the LP, up to eight bytes.)

When transmitting, these peripherals need to have enough bytes in their TX FIFO to perform a complete burst (as defined by their "bytes per beat" setting). In case there are fewer bytes available, the controller will wait for more data to be filled in.

Let's assume that the PCI burst size has been set to 32 bytes (eight words), while the alarm level for the TX FIFO has been set to 20 (bytes).

At the start of operation, the BestComm fills in the FIFO until it reaches granularity. The PCI controller meanwhile slowly empties it. As the FIFO empties, a level is reached where less than 32 bytes are left inside it. This level, unfortunately, is greater than the programmed alarm level. What happens is that both the BestComm and PCI will be idle; a livelock situation is reached.

An easy way to avoid this is to increase the alarm level to more than 32. Another way around it would it be to reduce the bytes per burst transferred by the PCI, but this will impact the performances overall.

2.1.3.2 Ethernet, A Framed Peripheral

Ethernet represents a special case. Together with the SPI TX mode of the PSC, it supports the concept of *frame*.

An Ethernet packet's size is unknown *a priori* (payload per frame can be 64 bytes to 1518 bytes). During transmission, the Fast Ethernet Controller (FEC) will be asserting the TX-enable line to the external PHY together with the first valid nibble (or bit in the 10 MB/s case) and de-assert it just as the last nibble is transmitted. In a receiving case, the PHY will drive the RX-Data Valid signal in a similar fashion to indicate valid data.

The data either written or read is stored temporarily in a 1024-bytes-deep FIFO (one for each direction, TX and RX). At transmission time, BestComm will fill the TX FIFO. As the last data is written to it, it is tagged in the FIFO with a special bit indicating that datum represents the end of the frame. When the FEC reads the datum and its associated special tags later on, it will de-assert the TX-ENABLE as it transmits it out to the PHY. A special bit in the task's code, called Transmit Frame Done (TFD), indicates that the associated DRD will tag the 'last' data as described above.

Similarly, if a received data is followed by the immediate deassertion of the RX-DV signal, it is tagged by the FEC with a bit indicating a frame has been completed. As BestComm empties the FIFO, it reaches this piece of data and automatically forces the DMA engine to terminate the current executing DRD. Exiting the current nested level of execution, it thus retrieves a complete frame as it was originally transmitted.

2.1.4 The Peripherals

Last but not least, the peripherals themselves play a vital role in any BestComm transfer. Each one has a set of registers which must be correctly programmed to permit the transfer to occur successfully.

Every single peripheral has some different but proper characteristics; we can classify the supported peripherals according to the different criteria:

1. Half Duplex Peripherals
   — Local Plus and ATA: the user is required to program the data direction before starting the task
2. Full Duplex Peripherals
   — PCI, Ethernet, PSC and IrDa: they all have two FIFOs
3. Framed Peripherals
   — Ethernet and PSC-SPI (TX only) can use the frame signal which is a internal line between the peripheral and the BestComm to indicate that a frame has been transmitted (or received in the Ethernet case)
4. Peripherals that support the *flush* mechanism
   — PCI, LP and ATA. The PSC does not need it as the FIFO controller 'asserts' the requestor until the FIFO is emptied while Ethernet uses the frame signal to indicate end of received data
5. Peripherals that can be used by BestComm but do not have FIFOs

— The dynamic memory (SDRAM/DDR) and the two I^2C

It is the user, who must take care of properly programming the peripherals before enabling the BestComm's task that will use them. Typical error in this area is a wrong setup of the buffer descriptor table and of the registers, which control the peripheral, so that either BestComm or the peripheral itself will eventually stall, waiting for the other to move data. In general, every peripheral has registers used to reset the FIFO and FIFO controller, to start a data movement and to intercept errors and invoke interrupts.

2.2 BestComm Microcode

It is useful to analyze the microcode that is written to run the hardware. Let's start first with a few definitions:

- **Task**: refers to the microcode executed by BestComm to perform a desired function. A generic task can be categorized as follows:
  - Half duplex versus full duplex tasks: data are simply moved from Source (FIFO or memory) to Destination (memory or FIFO), or both data movements are executed within the same loop, 'emulating' a full duplex data movement. (In reality, only one transfer in one direction is executed at a time.)
  - Buffer descriptor versus Not-buffer descriptor tasks: where the task parses a buffer descriptor's table (a ring of buffer descriptors) to determine where the data must be transferred and how many bytes must be moved (in either direction)
- **Data Routing Descriptor (DRD)** and **Loop Control Descriptor (LCD)**: the basic elements of a task
- **Index**, **Increment** and **Variable**: a task uses indexes to point at those memory locations that will be interested by the data movement. Indexes are neither directly under user control nor be read at run time by software. A variable is used to pass a parameter to a task, often to initialize an index or as an intermediate temporary placeholder. They can be read (or written) at run time; an increment is used in a task loop control to add or subtract a fixed amount from an index at every turnaround of the task
- BestComm microcode **image**: refers to the collection of all the tasks loaded at run time in the SRAM, which can be singularly executed
- **Task Table** (or **Entry Table**): a table of pointers to the location of the DMA tasks. It could be located anywhere (SRAM, FLASH, SDRAM), but it achieves better performance if it is located with the tasks in SRAM
- **Task Descriptor Table**: a table for each task comprising the tasks's DRDs and LCDs
- **Task Variable Table**: an area allocated to each task where the DMA engine fetches the initial values for its variables (up to 24) and increments (up to eight). A task's variable table is composed of 32 words
- **Function Descriptor table**: the table, including the functions (e.g., AND, CRC, endian swap, etc.), available for each task. These tables are divided into four sections, of which only the last is pertinent to BestComm (in fact, of the four possible execution units of the SmartDMA family,

only EU3 is used). The last eight bits of the word of the Table Entry, pointing at the Function Table, do not represent any address, but assume special meanings, such as enabling the readline buffers, the integer/floating mode of the BestComm and others
- **Context Save Area**: a 20-word table used by BestComm to store and restore the context of a task while at run time a task swap or a task idle phase is encountered

It is possible to analyze an image from the user's point of view. The process of reverse engineering, at times a bit cumbersome and complicated, can nonetheless be very helpful when trying to debug a task.

Let's work it through an example. We shall refer to a standard PCI task case (both TX and RX).

2.2.1 BestComm Task Image

Code Example 1. Initial part of an image: global variables and Table Entry

```
.globl taskTableBytes
taskTableBytes:
.long 0x00001300 /* Number of bytes in image */

.globl taskTableTasks
taskTableTasks:
.long 0x00000010 /* Number of tasks in image */

.globl offsetEntry
offsetEntry:
.long 0x00000000 /* Offset to Entry section */

.globl taskTable
taskTable:

/* SmartDMA image contains 4864 bytes (411 bytes unused) */

/* Task0(TASK_PCI_TX): Start of Entry -> 0xef008000 */
.long 0x00000200 /* Task 0 Descriptor Table */
.long 0x0000022c
.long 0x00000500 /* Task 0 Variable Table */
.long 0x00000d07/* Task 0 Function Descriptor Table & Flags */
```

```
.long 0x00000000
.long 0x00000000
.long 0x00000e00 /* Task 0 context save space */
.long 0x00000000
/* Task1(TASK_PCI_RX): Start of Entry -> 0xef008020 */
.long 0x00000230 /* Task 0 Descriptor Table */
.long 0x00000250
.long 0x00000580 /* Task 0 Variable Table */
.long 0xXXXXXX07/* No FDT */
.long 0x00000000
.long 0x00000000
.long 0x00000e50 /* Task 0 context save space */
.............
```

Above is a part of a *dma_image.reloc.s* file as it might be released to a customer. This assembler file will have to be compiled and linked into the user's project. At run time a specific C function (a task loader, also provided to the customer) will load the desired image (resident in a repository either in SDRAM or Flash) into the SRAM.

In order to perform such an operation some global variables are provided directly in the file, namely the *taskTableBytes*, *taskTableTasks*, *offsetEntry* and *taskTable*. These four variables indicate the number of bytes which compose the total image, the number of tasks which are present in the image, eventually an offset for the position of the *Entry Table* with respect to the start of the image.

The taskTable variable is initialized by the linker and will indicate the source location from which to copy the image into SRAM.

Analyzing the above image (see Code Example 1. Initial part of an image: global variables and Table Entry) we see that the image is 4864 (0x1300) bytes long, it contain 16 (0x10) tasks, the *Entry Table* is located on top of the image (the first two entry of this table are also visible in the snippet).

The user must be careful while copying the image. In fact, the Table Entry is nothing else that a group of pointers to the location of the task descriptor tables. At loading time it is important to ensure that each of this pointer will point to the new location of the Task Descriptor table. Let's investigate this point in detail.

A typical task loader will look like the C function (see Code Example 2. Task Loader example). In such a function first the whole image will be copied byte per byte from the current location (as indicated by taskTable) to a destination location indicated by:

(sdma->taskBar) - MBarPhyOffsetGlobal.

The *taskBar* is a register of the MPC5200 (located at MBAR + 0x1200), which points to the physical location of the SRAM. The MBarPhyOffsetGlobal is used together with an Operating System (such as QNX or Linux) when virtual address and physical address must be different.

Code Example 2. Task Loader example

```
void TasksLoadImage (sdma_regs *sdma)

{

uint32 i;

SCTDT_T *tt;


/* copy task table from source to destination */

memcpy ((void *)((uint8 *)(sdma->taskBar) - MBarPhysOffsetGlobal), (void *)
&taskTable, (unsigned long) taskTableBytes);

/* adjust addresses in task table */

for (i=0; i < (uint32) taskTableTasks; i++) {

        tt = (SCTDT_T *) (((uint8 *)(sdma->taskBar) - MBarPhysOffsetGlobal) + (uint32)
offsetEntry + (i * sizeof (SCTDT_T)));

        tt->start                                          += sdma->taskBar;

        tt->stop                                           += sdma->taskBar;

        tt->var                                            += sdma->taskBar;

        tt->fdt                        = (sdma->taskBar & 0xFFFFFF00) + tt->fdt;

        tt->context                                        += sdma->taskBar;

}

/* initialize task variable pointers */

init_dma_image ((uint8 *)(sdma->taskBar) - MBarPhysOffsetGlobal);}
```

Just as an example, the PCI RX Task is located 0x230 bytes after the start of the image. Let’s assume that the image is loaded in SRAM, starting at address 0xef008000 (i.e., MBAR is currently set to 0xef000000).

Then the first pointer of the PCI RX *Entry Table* must point to 0xef008230. This relocation function is exactly what the second part of the task loader performs.

Let’s proceed further with the task’s image, specifically the Task Descriptor table.

Code Example 3. Task Descriptor Tables for the PCI Tx and Rx tasks

```
    .....

    /* Task0(TASK_PCI_TX): Start of TDT -> 0xef008200 */
```

```
.long 0xc080601b /* :  LCDEXT: idx0 = var1, idx1 = var0; ; idx0 += inc3, idx1 +=
inc3 */
.long 0x82190292 /* :  LCD: idx2 = var4; idx2 >= var10; idx2 += inc2 */
.long 0x1004c018 /* :    DRD1A: *idx0 = var3; FN=0 MORE init=0 WS=2 RS=0 */
.long 0x8381a288 /* :    LCD: idx3 = var7, idx4 = var3; idx4 > var10; idx3 += inc1,
idx4 += inc0 */
.long 0x011ec798 /* :      DRD1A: *idx1 = *idx3; FN=0 init=8 WS=3 RS=3 */
.long 0x10001f18 /* :    DRD1A: var7 = idx3; FN=0 MORE init=0 WS=0 RS=0 */
.long 0x850102db /* :    LCD: idx3 = var10, idx4 =var2; idx3 != var11; idx3 +=
inc3, idx4 += inc3 */
.long 0x60080002 /* :      DRD2A: EU0=0 EU1=0 EU2=0 EU3=2 EXT init=0 WS=0 RS=1 */
.long 0x08ccfd0b /* :      DRD2B1: idx3 = EU3(); EU3(*idx4,var11)  */
.long 0x0002d058 /* :    DRD1A: *idx4 = var11; FN=0 init=0 WS=1 RS=0 */
.long 0x80180024 /* :  LCD: idx0 = var0; idx0 once var0; idx0 += inc4 */
.long 0x040001f8 /* :    DRD1A: FN=0 INT init=0 WS=0 RS=0 */


/* Task1(TASK_PCI_RX): Start of TDT -> 0xef008230 */
.long 0xc000e01b /* :  LCDEXT: idx0 = var0, idx1 = var1; ; idx0 += inc3, idx1 +=
inc3 */
.long 0x81990212 /* :  LCD: idx2 = var3; idx2 >= var8; idx2 += inc2 */
.long 0x1004c010 /* :    DRD1A: *idx0 = var2; FN=0 MORE init=0 WS=2 RS=0 */
.long 0x83012208 /* :    LCD: idx3 = var6, idx4 = var2; idx4 > var8; idx3 += inc1,
idx4 += inc0 */
.long 0x00fecf88 /* :      DRD1A: *idx3 = *idx1; FN=0 init=7 WS=3 RS=3 */
.long 0x10001b18 /* :    DRD1A: var6 = idx3; FN=0 MORE init=0 WS=0 RS=0 */
.long 0x040001f8 /* :    DRD1A: FN=0 INT init=0 WS=0 RS=0 */
.long 0x8018001b /* :  LCD: idx0 = var0; idx0 once var0; idx0 += inc3 */
.long 0x040001f8 /* :    DRD1A: FN=0 INT init=0 WS=0 RS=0 */
......
```

The comments (enclosed in the C-style /* */ delimiters) relate interesting information. As an example, let's analyze the PCI TX task.

To add insight to this example, the reader should know that the task's operations are conceptually divided in the following phases:

1. Initialize an index, which will keep track of the number of iterations the task performs. Variable var10 will be passed by an upper software layer API function as a task's parameter to indicate exactly the number of desired loops. In such a way, an upper-bound limit on the maximum number of bytes transferable via a peripheral (such as in the PCI case, limited to 65536) can be worked around (increased, as a matter of fact).

2. Write the *TX Packet Size* register of the PCI Controller located at 0xef003800 with the number of bytes to be transmitted.
3. Transfer all data from a DDR memory location, whose memory location address has already been passed to the task as a parameter, to the PCI TX FIFO (at MBAR + 0x3840).
4. After having moved a number of bytes equal to var3 (another initial parameter for the task), the task will proceed to read the PCI TX Status Register located at offset 0x381C to poll the normal termination bit (bit 7 precisely). If this is found to be set high, meaning a successful transmission termination has been reached, the next iteration will start.
5. At the end of the last iteration transfer, an interrupt to the core will be generated to indicate the completion of the whole intended transfer.

The task code starts with a an outer LCD, which is described in the first two lines: the indexes idx0, idx1 and idx2 are initialized by the variables var1, var0 and var4, while their increments will be respectively inc3, inc3 again and inc2. The loop is executed as long as the boolean condition *idx2 >= var10* is evaluated as true as in typical C for-loop.

The increments and variables are evaluated at run time, so reading them ‘statically’ from the image may not give insight. It is a good idea during debug to print out the state of the variable table right before enabling the task. If this is done, the following would most likely be the result:

- var0 = any address of the DDR memory. This is the location of the data to be transferred (ex: 0x50000);
- var1 = 0xef003800, the address of the PCI TX data FIFO;
- var3 = number of bytes to transfer per iteration (maximum 65536);
- var4 = total number of iteration loops to be executed minus one as the iteration index start counting from 0. Example: 3 when four total iterations are required;
- var10 = Initial value of the iteration index, set automatically to 0 by the task’s code itself;
- inc3 = 0;
- incr2 = -1;

The first DRD of the outer loop reads as follows:

```
DRD1A: *idx0 = var3; FN=0 MORE init=0 WS=2 RS=0 */
```

That is, variable var 3 is read as a 32-bit word and then written it as a 16-bit short at the location pointed by idx0.

The size of the read and write is indicated by the RS and WS parameters, where 0 indicates 32-bit word, 1 a single byte, 2 a 16-bit short and 3 the fact that the size is dynamically set at run time by software.

The initiator indicated by the DRD (init = 0) in the code is actually initiator 0, the *always* initiator. At run time, the initiator of a task can be dynamically updated by the upper layer software API as requested. In such a way, the same task’s code can be applied to different peripherals, such as the PSC, for instance.

No special function is used for this DRD as indicated by FN = 0.

This DRD achieves phases (1) and (2) as described above. All the information readable inside the delimiters is, in fact, encoded in the DRD. There are different types of DRD to cover multiple functionalities and modes of operations.

The whole outer loop could be described in a C-like form as follows:

Code Example 4. a C-language equivalence of Code Example 3. Task Descriptor Tables for the PCI Tx and Rx tasks

```
for (idx0 = 0xef003800, idx1 = 0x50000, idx2 = 3; idx2<=0; idx0+=0, idx1+=0,
idx2-= -1)

{

    *(uint16*)(idx0) = (uint16) (0x0000FA00) /* 0xFA00 = d'64000 bytes to be TX-ed
*/

    ....... /* here is the rest of the task's code */

}
```

It is interesting to note that the PCI TX Packet Size register is indeed a 32-bit register, which must be written in the upper half short (the lower is reserved). A BestComm variable can only be 32-bits wide.

Therefore, it is important to indicate within BestComm which is the dominant data type to be assumed while performing a read or write. In our case, we are working with addresses, which are *unsigned integer* (a 32-bit word). The *INTEGER* mode of the BestComm must then be used. This is enabled by setting bit 27 of the pointer to the task descriptor table. This mode will force a 16-bit short to be aligned to the *left* within a word addressing space.

This operation can be done dynamically under software control, and, therefore, this bit appears as *not set* in the Code Example 1. Initial part of an image: global variables and Table Entry. The last byte of the fourth entry of the Table Entry reads 0x07, meaning that the readline and write combined buffers are enabled and that speculative reading is performed: these are normally enabled for most tasks.

Proceeding with the rest of the code, it is then easy to identify the sections of the Code Example 1. Initial part of an image: global variables and Table Entry which cover the points (3) through (5).

It is interesting to note that the data movement (phase (3)) is performed in an inner loop, by the DRD:

DRD1A: *idx1 = *idx3; FN=0 init=8 WS=3 RS=3 */

This DRD is conditioned by the PCI TX initiator (init = 8). The size of the data movement is set at run time, using the Transfer Size registers, and even the address of the PCI TX FIFO (0xef008340) is passed at run time by an upper layer API function. This indicates the DMA engine's flexibility and, of course, stresses the need for its proper initialization.

After exiting the first inner loop, a second inner loop is executed where phase (4) is executed. In this case, an if-clause is implemented via a for-loop. The PCI Status register is polled continuously, *and*-ed with a predefined mask, precisely 0x01, to check if the normal termination flag has been asserted. The function used (a binary *and*) is encoded in the LCD as EU3 = 2, indicating that Execution Unit 3 function number 2 will be used. This operation is conditioned again by the always (init =0) initiator.

Code Example 5. Execution Unit 3 available functions extracted from the Function Table

```
.long 0xa0045670 /* load_acc(), EU# 3 */

.long 0x80045670 /* unload_acc(), EU# 3 */

.long 0x21800000 /* and(), EU# 3 */

.long 0x21e00000 /* or(), EU# 3 */
```

```
        .long 0x21500000 /* xor(), EU# 3 */
        .long 0x21400000 /* andn(), EU# 3 */
        .long 0x21500000 /* not(), EU# 3 */
        .long 0x20400000 /* add(), EU# 3 */
        .long 0x20500000 /* sub(), EU# 3 */
        .long 0x20800000 /* lsh(), EU# 3 */
        .long 0x20a00000 /* rsh(), EU# 3 */
        .long 0xc0170000 /* crc8(), EU# 3 */
        .long 0xc0145670 /* crc16(), EU# 3 */
        .long 0xc0345670 /* crc32(), EU# 3 */
        .long 0xa0076540 /* endian32(), EU# 3 */
        .long 0xa0000760 /* endian16(), EU# 3 */
```

In the very last DRD:

DRD1A: FN=0 INT init=0 WS=0 RS=0 */

the request of an interrupt (INT) is visible. This DRD is in the outer for-loop, counting the iterations; the interrupt will be generated at the end, after the last iteration's last piece of data has been written to the FIFO.

The same procedure can be used to analyze any given task. The reader can always reverse engineer the task and create an equivalent 'C' code. For the interested reader, the PCI RX task in the Code Example 3. Task Descriptor Tables for the PCI Tx and Rx tasks performs all the phases of the TX task but phase (4).

It is also possible to find loops, which are executed just once, therefore, called *once-loops*. Their use can be explained at times by the need to work around existing limitations of the BestComm's DMA engine or just simply to perform operations such as implementing a conditional branch in a task.

NOTE: CommBus Prefetch Bug

The release of the BestComm's engine implemented in the MPC5200 version 1.0, version 1.1 and version 1.2 contained a significant bug on the Comm prefetch. The buffer is not flushed properly as the task terminates. A coding workaround, based on a once-loop, has been used to force flushing of the buffer to avoid having stale data the next time it is used.

2.3 The BestComm API and Task Builder

To ease the work of the software programmer, a set of API functions and a graphical task builder interface are currently in development.

The first will permit easy interfacing of the lower layer, the Firmware of the BestComm, with complex systems such as Operating Systems. Operations such as loading an image, setting up the task parameters, modifying at run time the initiator number of a DRD, the data size or small customizations of the tasks will be handled directly by these functions.

The API also performs all those small but vital operations, such as starting a task, setting up the BestComm interrupt mask and eventually checking received interrupts.

The task builder instead will allow the user to build his own set of tasks based on pre-existing generic task templates. For instance, based on a general single pointer buffer descriptor task, an ATA task can be generated just by selecting the ATA initiator and eventually fixing the request for an interrupt at the end of the task.

For more details on both the API and the graphical user interface, the reader should refer to their relative documentation.

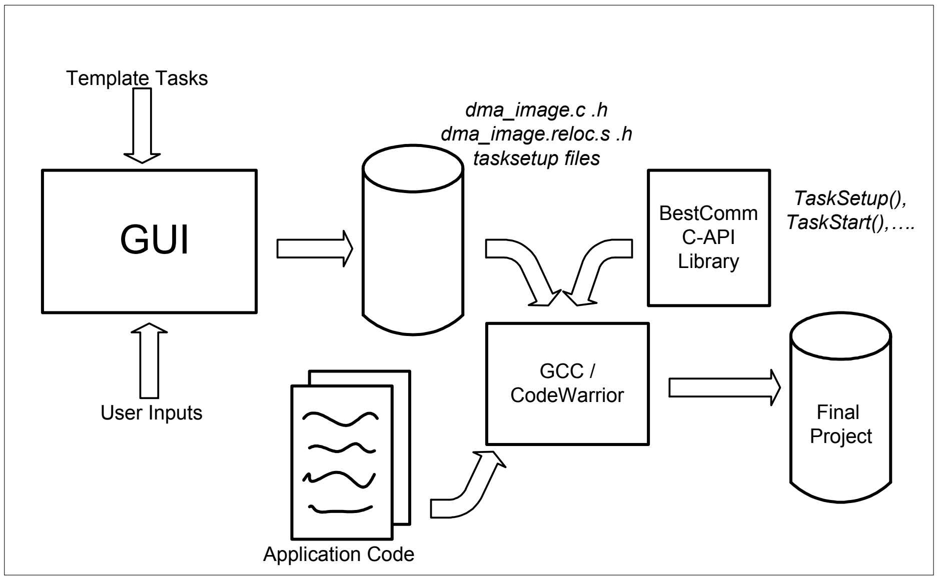

Figure 4. BestComm's development Flow

2.3.1 Ethernet: a Buffer Descriptor Task

Ethernet represents a very special case: as already noted in <st-blue>Section 2.1.3.2 Ethernet, A Framed Peripheral13, this peripheral is quite different since it uses the frame control mechanism, and its task operates based on buffer descriptors.

The MPC5200 does not implement any specific buffer descriptor mechanism in hardware; it is implemented via software.

Buffer descriptor is a technique which implements a ring of buffers to handle data transfer. A BestComm buffer descriptor task will parse through a table, called a buffer descriptor table, to detect how many bytes will be transmitted or received and to read the pointers to the source and to the destination location for the data movement. A flag can also be set to indicate on which buffer an interrupt is desired and when the frame termination signal must be asserted.

To better explore these special tasks, the Ethernet TX task will be described more in detail.

The task buffer descriptor table located in SRAM looks like the following:

BDTableBase+0x00 bd[0].Status/Length

BDTableBase+0x04 bd[0].SourcePtr
BDTableBase+0x08 bd[0].DestinationPtr
BDTableBase+0x0C bd[1].Status/Length
BDTableBase+0x10 bd[1].SourcePtr
BDTableBase+0x14 bd[1].DestinationPtr
....
BDTableLast+0x00 bd[n].Status/Length
BDTableLast+0x04 bd[n].SourcePtr
BDTableLast+0x08 bd[n].DestinationPtr

The Status/Length field is a 32-bit field whose lower 26 bits can be used to define the length in bytes of the packet transmitted or received, while bits 26, 27 and 30 represent, respectively, a request for an Interrupt, a transmit-frame-done flag and a data-ready flag, as will be evident later on.

The task is called by passing some parameters to it, among which the most important are the pointer to the buffer descriptor table first address, the pointer to the last entry and the pointer to the specific DRD, which will issue the INT and TFD flags. In such a way, an interrupt can be signalled on any buffer in the table.

In order to function properly, the table must be prepared by the user, so that the length field is always entered with the number of bytes the task is supposed to transmit (the same is true for the RX task) and with the correct pointer to the location in SDRAM, where the buffers are located. The specific Ethernet task itself will take care of initializing the destination pointer to the address of the FEC TX FIFO. The receiving case is the dual case of the transmitting one; the user will have to set up the receiving pointer, and the task will take care of the source pointer.

The basic operations performed by the task can be divided into five stages:

1. Clear the INT and TFD flag on a specified DRD as a preparation phase
2. Poll continuously the Status/Length field to check if there are bytes to be transferred and, if the data ready flag is asserted, to be sure the buffer is available for transmission. As long as the data ready flag is not set or the length field is zero, the task will hang on this DRD
3. Set, as requested by the user, the TFD and INT flag on the proper DRD
4. Move the whole amount of data (1 Buffer), and at the end of an eventual Frame (which could be made up of many buffers) assert the Transmit Frame Done and send an Interrupt to the Core if requested by the buffer status bits
5. Write the length field with the number of bytes left to be transmitted (usually for a normally terminated transfer this is equal to zero), increase the pointer to the buffer descriptor table and then go back to phase (2), where the next buffer length field is read. After the very last buffer, the task can be set to reload itself starting from the beginning of the buffer descriptor table. If the core has, in the meantime, refilled the length fields, a new transmission will start over immediately. In case a special interrupt is needed just at the end of the buffer descriptor table, it can be also generated

These phases are implemented in the following code:

Code Example 6. Ethernet TX Task's code

```
.long 0x8018001b : LCD: idx0 = var0; idx0 <= var0; idx0 += inc3 */

.long 0x60000005 : DRD2A: EU0=0 EU1=0 EU2=0 EU3=5 EXT init=0 WS=0 RS=0 */
```

```
.long 0x014cfc09 : DRD2B1: var5 = EU3(); EU3(*idx0,var9)  */

.long 0x808120e3 : LCD: idx0 = var1, idx1 = var2; idx1 <= var3; idx0 += inc4, idx1
+= inc3 */

.long 0x850002e4 : LCD: idx2 = var10, idx3 = var0; idx2 < var11; idx2 += inc4,
idx3 += inc4 */

.long 0x60800002 :DRD2A: EU0=0 EU1=0 EU2=0 EU3=2 EXT init=4 WS=0 RS=0 */

.long 0x088cfc4c :DRD2B1: idx2 = EU3(); EU3(*idx1,var12)  */

.long 0x70000002 : DRD2A: EU0=0 EU1=0 EU2=0 EU3=2 EXT MORE init=0 WS=0 RS=0 */

.long 0x01ccfc49 :DRD2B1: var7 = EU3(); EU3(*idx1,var9)  */

.long 0x70000003 :DRD2A: EU0=0 EU1=0 EU2=0 EU3=3 EXT MORE init=0 WS=0 RS=0 */

.long 0x0cccf1c5 :DRD2B1: *idx3 = EU3(); EU3(var7,var5)  */

.long 0xd9190340 :LCDEXT: idx2 = idx2; idx2 > var13; idx2 += inc0 */

.long 0xb8c56009 :LCD: idx3 = *(idx1 + var00000014); ; idx3 += inc1 */

.long 0x009ec398 :DRD1A: *idx0 = *idx3; FN=0 init=4 WS=3 RS=3 */

.long 0x991982ea :   LCD: idx2 = idx2, idx3 = idx3; idx2 > var11; idx2 += inc5,
idx3 += inc2 */

.long 0x088ac398 :     DRD1A: *idx0 = *idx3; FN=0 TFD init=4 WS=1 RS=1 */

.long 0x60000005 :   DRD2A: EU0=0 EU1=0 EU2=0 EU3=5 EXT init=0 WS=0 RS=0 */

.long 0x0c4cf88b :   DRD2B1: *idx1 = EU3(); EU3(idx2,var11)  */
```

Let’s analyze the major sections of this code.

The first loop clears in a specific DRD, pointed by index0 and passed by software, the bits at positions 26 and 27. These bits are the INT and TFD control bits of the DRD.

```
.long 0x8018001b :  /* LCD: idx0 = var0; idx0 <= var0; idx0 += inc3 */

.long 0x60000005 :  /* DRD2A: EU0=0 EU1=0 EU2=0 EU3=5 EXT init=0 WS=0 RS=0. Note
ALWAYS Initiator*/

.long 0x014cfc09 :  /* DRD2B1: var5 = EU3(); EU3(*idx0,var9)  */
```

This could be described as follows:

```
for(PtrDRD = AddrDRD; PtrDRD <= AddrDRD; PtrDRD +=Inc3)

{

        MaskDRD = (*PtrDRD) & [~0x0C0000000];

}
```

The exact DRD, on which this operation is applied, is, in our case, the third last from the bottom, precisely the one which performs the data movement (0x088AC398). In general, this DRD can change and may be chosen at run time. In a successive stage of the task, the TFD and the INT flags will be set as per the user's choice.

What follows is shown below:

```
.long 0x808120e3 :  LCD: idx0 = var1, idx1 = var2; idx1 <= var3; idx0 += inc4, idx1
+= inc3 */

.long 0x850002e4 :    LCD: idx2 = var10, idx3 = var0; idx2 < var11; idx2 += inc4,
idx3 += inc4 */

.long 0x60800002 :      DRD2A: EU0=0 EU1=0 EU2=0 EU3=2 EXT init=4 WS=0 RS=0 */

.long 0x088cfc4c :      DRD2B1: idx2 = EU3(); EU3(*idx1,var12)  */

.long 0x70000002 :    DRD2A: EU0=0 EU1=0 EU2=0 EU3=2 EXT MORE init=0 WS=0 RS=0 */

.long 0x01ccfc49 :    DRD2B1: var7 = EU3(); EU3(*idx1,var9)  */

.long 0x70000003 :    DRD2A: EU0=0 EU1=0 EU2=0 EU3=3 EXT MORE init=0 WS=0 RS=0 */

.long 0x0cccf1c5 :    DRD2B1: *idx3 = EU3(); EU3(var7,var5)  */
```

is the main loop parsing the buffer descriptor table, followed by the check of the status/length field. This may be interpreted as follows:

```
for (PtrDst = AddrFIFODst, PtrBD = BDTAbleBase; PtrBD <= PtrTableLast; Ptr += 8)

{

        for(length = 0, PtrDRD=AddrDRD; length < 0x40000000)

        { /* Function #2 is an "And", function#3 is a "Or" */

        length = (*PtrBD) & (0x40000000 | 0x0C000000) /* Var 9 = 0x0C000000*/

        }

        cond = (*PtrBD) & (0x40000000);

        *PtrDRD = cond | (MaskDRD)

........................

}
```

The task will start at the buffer descriptor table base address passed to it as an initial parameter, and, at every new loop iteration, it will increment the pointer by eight bytes because, in this specific task, the table is composed just by the Status/Length field (four bytes), and the Source Pointer as the destination is fixed once and for all by the address of the FEC TX FIFO.

The first inner loop will check the Status bits: the most important is bit 30 (data-ready flag) indicating the buffer is ready for transmission or reception. As long as this bit is not asserted, the task will be sitting here, polling this DRD. For such a reason, it is always better to locate the buffer descriptor table in SRAM and not in SDRAM; otherwise, useless traffic will be generated on the XLB.

As BestComm polls the buffer descriptor Status/Length field, it may not give any chance for other tasks to run: task starvation is, therefore, possible. To avoid this livelock, it is recommended to set the TX buffer descriptor task initiator priority for less than the one for the RX task, and careful attention must be paid when more than one buffer descriptor task is enabled simultaneously.

NOTE: Priority Rule

> If different tasks have their initiators set at the same priority level, the task with the higher ordinal number in the tasks' list will be the one with the highest priority at arbitration time. There is no pre-emptying mechanism provided in BestComm. Arbitration happens when the granularity or alarm level are reached or when a loop exit condition is met.

As soon as the data ready flag is detected as asserted, the task proceeds and checks the INT and TFD flag of the Status/Length word. If they are asserted, the corresponding bits on the proper DRD, indicated by the user, is set. The users can pass the DRD address on which they desire to set these flags as a task's parameter.

During the next phase, data are moved, first, by moving at least a word at a time to maximize FIFO throughput, later when less than (or equal to) four bytes are left, by moving a byte at a time. This last phase is called the *misaligned* part of a loop.

This is required because the size of the frame can be anything between 64 and 1518 bytes (not including CRC); therefore, any intermediate number is possible, but moving words increases performance. What was just described is achieved by the following DRDs:

```
.long 0xd9190340 :    LCDEXT: idx2 = idx2; idx2 > var13; idx2 += inc0 */

.long 0xb8c56009 :    LCD: idx3 = *(idx1 + var00000014); ; idx3 += inc1 */

.long 0x009ec398 :      DRD1A: *idx0 = *idx3; FN=0 init=4 WS=3 RS=3 */

.long 0x991982ea :    LCD: idx2 = idx2, idx3 = idx3; idx2 > var11; idx2 += inc5,
idx3 += inc2 */

.long 0x088ac398 :      DRD1A: *idx0 = *idx3; FN=0 TFD init=4 WS=1 RS=1 */
```

This can be rewritten as follows:

```
for(PtrDest = StartAddrDst, PtrSrc = StartAddrSource, Bytes2Txr= Length; Bytes2Tx >
0x40000004; Byte2Tx-=4, PtrSrc+=4)

        {

        PtrDest = PtrSrc

        }

for(PtrDest = PtrDst, PtrSrc = PtrSrc, Bytes2Tx = Bytes2Tx; Bytes2Tx > 0x40000004;
Byte2Tx-=1, PtrSrc+=1)

        {

        PtrDest = PtrSrc, TFDonExit(),intOnExit()

        }
```

At the end of the *misaligned* byte-moving section, the frame signal is asserted as indicated by the TFD. Also the interrupt to the Core may be asserted, indicating that the frame has been written to the FIFO.

NOTE: Interrupt use

In a table composed of many buffers, it is often better to assert a core interrupt at an intermediate level, midway through the table and at the very end. The core will then know which buffers are available for refilling, and the task can run smoothly continuously.

The last part of the task will just write the value of the bytes *left* to be transmitted, which should be zero if everything finished normally, with a cleared data flag bit back into the Status/Length field, thus indicating to the upper layer of SW that the buffer has been transmitted.

The task then is ready to go on the next Buffer Descriptor until the very last one is encountered.

When all of the Buffer Descriptors have been sent, the task will be completed. It will either exits or if the auto-start bit has been selected, it will start again from the Buffer Descriptor Table first entry.

3 BestComm Debugging

The process of debugging a task is often frustrating and demands time and imaginative effort from the software engineer.

The fact is that BestComm offers limited resources for debugging a task, and occasionally the only way is to simulate the task. Because the latter option is not always offered to the user, some indications or, better said, common-sense suggestions are shared here on where to look first. To help us through this section, examples will be used frequently.

Let’s assume that a peripheral is started, expecting that some bytes should be moved from it to an external device such as a memory mapped unit (such as a SRAM). As the program is started, no data movement is detected on the external Local bus.

NOTE: A Simple Debug Method

Where possible, just for debugging reasons, an infinite loop can be used, conditioned possibly by a flag, set by an interrupt service routine servicing the task’s BestComm interrupt. Inside this loop, different registers’ content can be displayed to indicate the status of BestComm.

The first question to answer is whether or not the task is correctly enabled. For such a reason, printing the relative task control register (TCR) becomes a necessity. Each TCR register holds two control half-words for two separate tasks. The most significant bit of the control half-word indicates if a task is enabled or not. A task, when finished, will either shut itself off or eventually start another task as indicated in the same control half-word, including itself. Precisely observe whether the currently active initiator is active or not.

To visualize which requestors are currently active, the register at address MBAR + 0x1280 can be used.

This register outputs a 32-bit word whose bits ordered from 31 down to 0 represents each initiator, starting from initiator 31 (a reserved initiator for the current design) to the *always* initiator (initiator 0).

Another essential register is the current pointer register. The content of it is the address of the DRD, which is currently in execution. In case of a stuck task, this indication will be very useful. It is also often required to print the DRD itself to check that the correct initiator has been selected for this DRD.

Beside these three registers, the FIFO read and write pointer often provides helpful information which side, BestComm or the peripheral, is malfunctioning. Checking for FIFO overflow/underrun error condition makes sense at all times.

When this information is available, the software engineer can at least have a hint regarding the possible errors.

It is difficult to describe every possible error, and there is no fixed method on how to proceed. Several example follow, indicating possible methods of debugging:

Let's assume that a generic dual pointer buffer descriptor task (used, for example, to read/write to an ATA hard disk) gets stuck on its very first DRD, where the Status/Length word of the buffer descriptor is parsed. The task is allowed to proceed with the data movement only if this word is set to a not-null value. Then it is very likely that the buffer descriptor is not properly set up.

Later on, after having fixed the first error, if the test is repeated continuously, the data read in are offset by one address location in the final receiving buffer. Usually this is a good indication of the CommBus prefetch error (see NOTE : CommBus Prefetch Bug.); trying to disable the CommBus is the first avenue to pursue.

In a different case, an audio file is processed in real time. Two tasks, a receiving one and a transmitting one, are used to drive an external set of codecs using a double buffering technique. After having set everything properly, it is detected that, while the TX task interrupt is always received, the RX FIFO never gets filled up to the alarm level. (This is visible by printing the FIFO status register. The task itself, this time, has no special fault, but possibly the requestor priority is wrongly set. The TX task, by default at start, finds its alarm enabled, and the BestComm's engine moves data much faster than the codec can use it.

What happens is that the RX FIFO fills in at a slower rate than the TX FIFO empties, and the frequency with which the TX task requires the use of the BestComm's engine is much higher than that of the RX task. If the priority settings between the two are favoring the TX, the RX task will run a high risk of being starved.

NOTE: BestComm's Task Switch

> It is important to remind the user again that BestComm is not a pre-emptying engine; it is not allowed to stop a task at any point in time but only when a task can give up the bus it can be stop, which is more frequently when an initiator becomes inactive.

In the last example, a BestComm's task is supposed to move data from an external device (an FPGA, for instance) via the Local Plus interface. The attached devices could support dynamic bus size, i.e., the size of the read data changes dynamically from words to bytes as indicated by the TSIZ bits of the Local Plus interface. (Such a device could be the Epson-Seiko S1D13086 LCD driver.)

The BestComm Local Plus task is based on a generic single pointer task (similar to the PCI task analyzed above).

This task allows the user to enable the *misaligned* option as part of its settings. This option forces the task to try to move data with the biggest size specified as long as possible. When the number of bytes left to be transferred is less than the data size, it switches to a single-byte movement.

To improve performance, the Local Plus has been set up to move eight bytes per transfer burst. In this case, even if the BestComm task is properly set, it still might fail during either a write or a read phase.

The write access might fail if the alarm level is set to less than eight in a similar fashion to the previous PCI case. However, in the read case, the problem derives from the fact that the Local Plus interface does not support any *misaligned* reads, which is a read done from an address that is not aligned with the size of the requested data (i.e., for a word in which the starting *aligned* address would end only with 0x0, 0x4,0x8 or 0x0C).

The solution is straightforward; ensure that the starting address of the LP task is aligned with the data size, which will be read in the port and which will increase the alarm level.

As these three examples show, an error can be anywhere, hidden perhaps in the API functions during the set-up phase of the task, in the silicon implementation of the BestComm or even in the peripheral used. It is up to the user’s experience to detect the most probable cause of the failure.

A debug’s submodule exists within BestComm. It allows having software breakpoints (and theoretically hardware breakpoints). However, its usage is limited.

3.1 BestComm, The Core and the Data Cache

The PowerPC 603e core disposes of two internal caches: one for instruction, and one for data, each the size of 16 kB. Correct utilization of the cache can definitely increase performance, while a wrong use often causes difficulties or even complete failure. The biggest issues are normally tied with the data cache and the memory management unit (MMU).

The data cache can be used in two modes: write-through or copy-back. The first implies that when a data is written by the core to the D-cache, it is automatically written out to the external memory; the second one instead does not, deferring this operation to a later moment, when it is required. The advantage of a copy-back mode is that only necessary data will be updated to the external slower memory, using burst write on the XLB, while the write-through mode can only write to the XLB in a single word.

The MMU allows the user, via the Block Address Translation registers (BAT), to specify if a given address range has to be cached in write-through or copy-back mode, if it is cache-inhibited or is cacheable, if memory coherency must be enforced (enabling the of the snoop operation of the core) or if out-of-order operations are possible (not guarded versus guarded memory). These are referred to WIMG bits of the BAT.

As soon as the MMU is enabled, every used BAT is by default enabled with the WIMG bits set as b’0011, meaning the address range is cached in a copy-back mode; memory coherency is enabled, and the memory is guarded against out-of-order operations.

This could mean, for example, that the external memory (SDRAM or DDR), where BestComm buffers are usually located, is considered by the core as a copy-back, cached address range.

What would then happen if the core prepared, for instance, a buffer to be transmitted to a FIFO by a BestComm task?

This data will only be written locally in the d-cache because the cache policy is to copy-back if requested by another master. If BestComm then independently accesses a datum from this buffer, it is possible that a stale value is still present and, therefore, wrong data would be transmitted.

The object is clearly to make sure that, when the BestComm task requests the data, this access is *snooped* by the core. Snooping refers to the operation performed by the core, checking whether a datum requested by another master on the XLB is currently present in its internal d-cache. If this is the case (and the cache line is in the modified state), an address retry is asserted on the bus. The requesting master is asked by such a signal to back off, allow the core to update the data on the physical memory and then retry the request.

To force this operation, the BestComm's request has to be a *global* request. In particular, it has to assert the internal $\overline{\text{GBL}}$ bar signal, forcing the core to snoop this particular address. A snooping window is available within the XLB Arbiter registers to perform this operation.

The user must be very careful because there is only one snooping window available in the MPC5200. Besides this, any time an address retry happens, a performance hit occurs because the core has to go out and update the data (one cache line, i.e., 32 bytes) on the external memory, thus delaying the BestComm's data movement.

In general, an area of memory shared between many masters shall not be cached or could be cached in write-through mode. When copy-back cache is needed, the snooping window must be used to ensure memory coherency.

It is also important to recall that only the core can snoop on the XLB (only the core has an internal d-Cache!). BestComm cannot be expected to be aware of the core's function regarding the data that it was intended to move.

In the figure below, the address retry case is shown with more detail. BestComm requests a burst read at 0x50000. This data is currently valid only in d-cache; therefore, the core asserts the address retry, and the BestComm backs off, permitting the core to perform a write-with-kill burst. This transfer means that the addressed block is flushed to the memory and invalidated in the cache. Only after this request has been acknowledged can BestComm request the same address again.

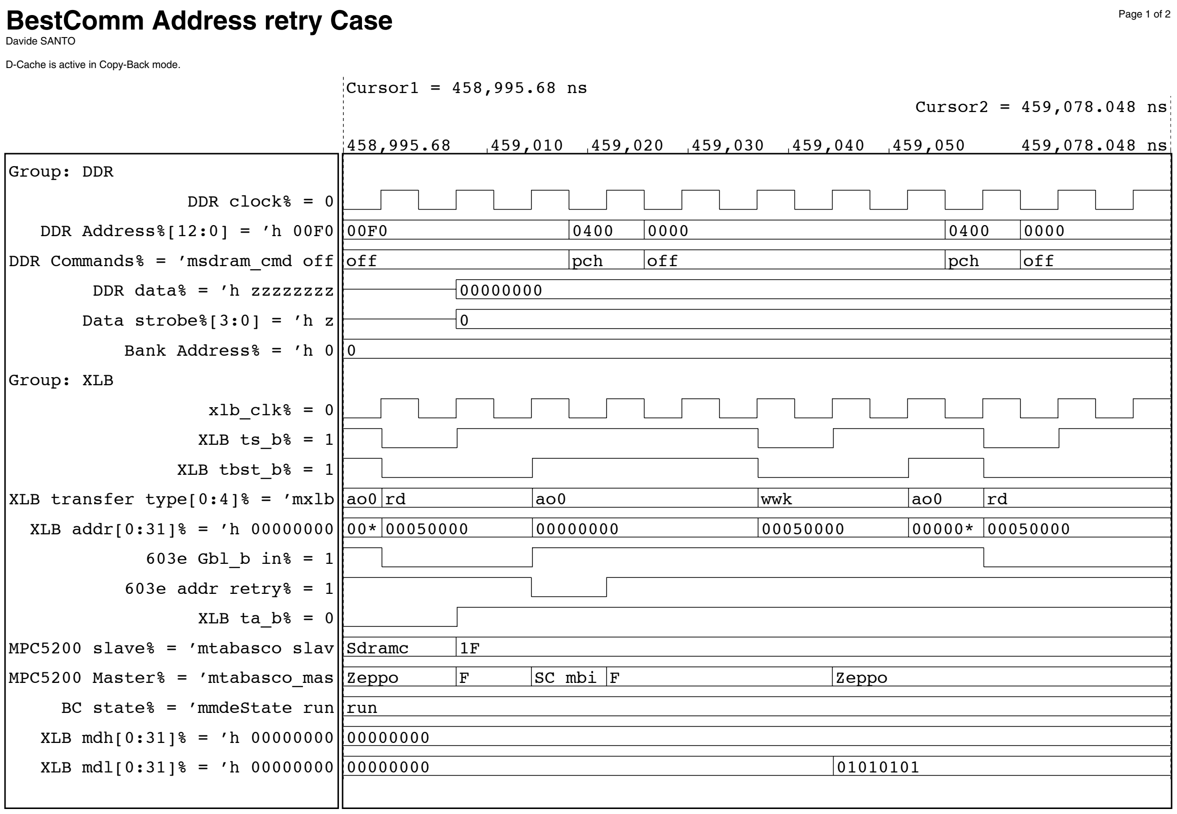

Figure 5. Address Retry

4 BestComm Limitations

There are several shortcomings of BestComm, meaning things that the DMA engine cannot perform or challenges that need to be overcome. Among others, let's note the following:

1. Granularity range is limited (between 0 and 7): this partially limits the 'flexibility' when many tasks are concurrently running
2. No external DMA request is available: no task can be directly invoked by an external device; an interrupt to the core can be used, but the latency is not fixed
3. No support for MAC or any complex protocol oriented functions (such as checking a CAN ID before starting a transfer, supporting AC97 control slots, etc.): this limits the core functionalities of BestComm to basic DMA movement with a restricted possibility of operation on the transferred data
4. Difficult-to-ensure bursting, thus high performance, on the XLB: this may require careful programming of the task with a re-evaluation of the alarm level. While this is possible, it is still difficult to achieve under all real-time operating conditions

5. Difficult-to-predict performance (bandwidth and throughput) because the BestComm performance is highly affected by the other masters and the collateral effects of the ‘task’ switching
6. Fixed FIFO size
7. Non-support of the PCI Target mode, CAN and the standalone SPI by BestComm; other applications requiring intensive BestComm usage (such as reading PCI data in continuos mode or handling audio data) present challenges because the XLB bus may use the total bandwidth
8. Difficulty in customization of a task; often a simulation is the only safe way to do this
9. Task debugging requires experience, and the learning curve is steep
10. Performance degrades when multiple tasks are enabled simultaneously; this is a direct consequence of a single central engine handling many separate, independent peripherals

5 Performance Analysis

While it is often very difficult to assess the performance of BestComm because it depends heavily on the application currently running on the MPC5200, it is interesting to note how well the DMA engine can perform while reading in data, storing it into the local dynamic memory, saving or restoring a task and so on. The intent of this paragraph is to give a rough idea of the overhead hidden in a BestComm data transfer, where this originates from and how it can affect the overall total performance.

As mentioned before, not every situation can be analyzed at this stage; therefore, a few examples follow.

The first one is the PCI continuos-mode read task. This task reads in data from the PCI interface with the PCI controller’s *Full Burst* bit of the receiving submodule enabled.

With this feature enabled, no check of the emptiness of the receiving PCI FIFO is performed, and the PCI controller continues to fill the FIFO. The PCI bus is never relinquished until the full pre-programmed packet (maximum 65536 bytes long) has been received. The PCI controller also needs at least one turnaround clock every 16 words, but no rearbitrating cycle will be inserted.

The maximum theoretical throughput achievable based on these constraints would be about 100 MB/s with PCI runs at 33 MHz (or 200 MB/s if running at 66 MHz). This is true, given that the XLB is always available for BestComm to use it (i.e., either the core is basically running idle, or it is seldom fetching instruction/data in case the caches are used).

Let’s follow the data flowing through the processor. We shall judge performance of the BestComm engine by two arbitrary methods:

1. Initial Latency is defined as the time between the initiator becoming active after an idle period and the time when the first data to be read in reaches its final destination in DDR.
2. The ratio between the time needed by the PCI controller on the external bus to read in 1024 bytes (our test-case packet size) and the time required by BestComm to move all of this data into its final destination is a good indication of performance.

We will then also evaluate how much of the XLB bandwidth this process uses by computing the ratio of the XLB clocks used by BestComm while performing the task over the total number of clocks available per transfer.

5.1 Initial Latency

The initial latency is measured exactly as the time between the RX FIFO initiator becoming active and the the very first data being written physically in the DDR memory.

The given internal bus frequencies are XLB set to 132 MHZ, IP (BestComm's CommBus) set to 132 MHz and external PCI set to 33 MHz. The core is basically idle, not fetching any instruction from the XLB bus, thus leaving the XLB bus all for BestComm (best case analysis).

The RX FIFO alarm has been set to 32 while the Granularity is set to four. The RX FIFO is 512-bytes deep; therefore, the RX requestor will assert as the FIFO is almost full at a watermark level given by:

$$512 - 36 - 4 = 476 \qquad \textit{Eqn. 1}$$

while the requestor will de-assert because four words are left in the FIFO.

The task begins by writing the RX Packet Register with the value 1024. This automatically forces the PCI controller to request the PCI bus and, when granted by the internal PCI arbiter, to transfer data from the addressed external PCI target.

When 476 bytes are detected in the FIFO, the PCI will burst in the data, and the RX FIFO will get filled in at PCI speed (33 MHz). Then the initiator gets active; at this moment, the only task enabled in the BestComm is the PCI receiving task, which is in an idle state, waiting for the initiator.

BestComm starts to run; in the best case, no context restore is necessary, and, after 30 XLB clock cycles, the first burst write request on the internal XLB bus is visible.

Because of the DDR controller's latency, the first data is written 36 XLB clock later than the assertion of the requestor (roughly 270 nsec). A complete burst of eight words will follow in the next four cycles. It is a duty of the write combined buffer and the XLB interface to gather the data necessary to perform a burst write on the XLB bus to the DDR memory.

In the end, this is the very first initial latency (36 XLB clocks).

The task then proceeds; as granularity level is reached, the task is saved and moved into the idle state waiting for more data to be input.

Again, when 476 bytes are detected in the FIFO, the initiator will go active. In this case 38 XLB cycles are needed in between the initiator going active and the write command appearing on the idle XLB bus. Six clocks later the first data gets written. This time, the latency increases because the BestComm had to spend a few cycles (eight in total) to restore the task's state.

The latency, therefore, is not a fixed quantity, but it will vary according to the current situation of the BestComm's engine (e.g., if another task is running, the length of time it takes to restore depends on the indexes to be retrieved, etc.) and on the current availability of the XLB bus.

NOTE: Worst Case Analysis

The worst case analysis is not easily done because the boundary conditions given by the system under test conditions must be reproduced in a simulated environment, a task which is often too complicated.

5.2 BestComm Efficiency

Next let's evaluate the efficiency of BestComm in transporting the data from the RX FIFO to the DDR memory as a block.

Every single 64-byte burst takes 21 PCI clocks as described: one clock for PCI Frame to be asserted + one frame for PCI Target Ready + 16 clocks for the data + two clocks for the Stop (Disconnect with data performed by the Target in this case) + 1 clock for bus turnaround.

To transfer 1024 bytes (16 bursts), it will take PCI 336 clocks, and, since one PCI clock. is roughly 30 nsec at 33 MHz, this yields a bit more than 10 microseconds.

BestComm will transfer the data internally on the XLB in burst of eight words each, as long as the initiator is active. Being faster (132 MHz) than the PCI, it will reach the granularity level and then wait for more data to be input.

It takes two *block* transfers to move all the 1024 bytes. The first takes 358 XLB clocks, while the second about 149 XLB clocks. In total, it takes 507 XLB clocks, which at a simulated XLB clock period of 7.488 nsec yields roughly 3796.4 nsec.

The BestComm efficiency is the ratio between the PCI required time and the BestComm needed clocks. In this case this yields:

BestComm efficiency ***Eqn. 2***

$$\eta = \frac{\text{336 PCI clocks}}{\text{507 XLB clocks}} = \frac{10,181\ \mu s}{3,796\ \mu s} = 2,68$$

which can be read as follows: it takes BestComm (at these fixed frequencies and conditions) 37.3% (= 1/2.68) of the PCI input time to transfer the data to its final destination.

Included in this computation is the time that the DDR memory requires to be written, plus all the overhead of BestComm.

What happens if we change the IP frequency, thus BestComm's internal speed, to 66 MHz? The efficiency will diminish because BestComm will take a bigger quota of clocks compared with the PCI input time to move data from the FIFO to the memory. This will be roughly one-half of the value given by Equation 2, but not exactly, because the overhead time is not a linear function with the frequency. Usually when IP is set to 132 MHz, some operations such as writing an internal register can take two clocks to meet internal timing constraints, while at 66 MHz these operations might take only one clock.

5.3 XLB Bandwidth Usage

Examining only the XLB bus, its bandwidth usage can be approximately computed as follows.

Every request (in this specific case) from BestComm is translated into a BURST of 32 bytes (eight words). Between two successive burst requests on the bus, a total of 14 XLB clocks are counted, and the last piece of data is valid on the XLB bus data after eight XLB clocks, thus leaving six XLB clocks potentially free for an alternative use by another master.

The task then requires at best case:

$$\frac{8\ \text{XLB clocks}}{14\ \text{XLB clocks}} = 0,57 = 57\ \% \qquad \textbf{\textit{Eqn. 3}}$$

of the bus

In reality, this happens when no active command is needed. (Writing to the same page as the previous data does not require precharging and activating the page at the DDR memory level.) The bus usage may degrade to 62% in case these operations are needed.

If the IP bus frequency is set to 66 MHz, a XLB usage of 30% is seen since BestComm is slower. Therefore, it goes out less frequently to request the XLB, and it uses less bandwidth.

6 Conclusion

BestComm is an *efficient* DMA engine, neither process nor protocol oriented, but primarily designed for simple data movement. Its main focus is to off-load the core from having to continuously move data, and its main advantage lies in the fact that it is easy to program. As with many other co-processors, there is a learning curve, and some experience is required to best use it. However, with the usage of the Task builder graphical interface and the provided API set of functions, these requirements are greatly reduced. BestComm will have a greater impact while integrating diverse functions when the tasks have been carefully selected and properly programmed, including their initiator priorities and their interactions with the core and other potential masters (PCI and USB).

How to Reach Us:

Home Page:
www.freescale.com

E-mail:
support@freescale.com

USA/Europe or Locations Not Listed:
Freescale Semiconductor
Technical Information Center, CH370
1300 N. Alma School Road
Chandler, Arizona 85224
+1-800-521-6274 or +1-480-768-2130
support@freescale.com

Europe, Middle East, and Africa:
Freescale Halbleiter Deutschland GmbH
Technical Information Center
Schatzbogen 7
81829 Muenchen, Germany
+44 1296 380 456 (English)
+46 8 52200080 (English)
+49 89 92103 559 (German)
+33 1 69 35 48 48 (French)
support@freescale.com

Japan:
Freescale Semiconductor Japan Ltd.
Headquarters
ARCO Tower 15F
1-8-1, Shimo-Meguro, Meguro-ku,
Tokyo 153-0064
Japan
0120 191014 or +81 3 5437 9125
support.japan@freescale.com

Asia/Pacific:
Freescale Semiconductor Hong Kong Ltd.
Technical Information Center
2 Dai King Street
Tai Po Industrial Estate
Tai Po, N.T., Hong Kong
+800 2666 8080
support.asia@freescale.com

For Literature Requests Only:
Freescale Semiconductor Literature Distribution Center
P.O. Box 5405
Denver, Colorado 80217
1-800-441-2447 or 303-675-2140
Fax: 303-675-2150
LDCForFreescaleSemiconductor@hibbertgroup.com