BestComm API
User's Guide

Revision 2.2

December 9, 2004

PREFACE

The *BestComm API User's Guide* describes the use of the BestComm API.

This manual is organized as follows:

- Section 1 gives on overview of the BestComm API.
- Section 2 describes the BestComm API functions.
- Section 3 describes the tasks included with the BestComm API release.
- Section 4 describes how to use the BestComm API and gives some sample code examples.
- Appendix A describes the tasks that are included with the BestComm API release in more detail.
- The doc/BestCommAPIReference directory contains an HTML reference to the BestComm API.

TABLE OF CONTENTS

SECTION **PAGE NUMBER**

Section 1
Overview

Section 2
Using the BestComm API

Section 3
Bestcomm Task Descriptions

Section 4
BestComm API Definitions

TABLE OF CONTENTS

SECTION **PAGE NUMBER**

Appendix A
Task Descriptions

SECTION 1
OVERVIEW

1.1 INTRODUCTION

1.1.1 MPC5200 BestComm DMA Engine

The BestComm DMA engine is designed to transfer data within the MPC5200 without the intervention of the embedded G2_LE processor. This means that data can be transferred between memory and the various peripherals on the MPC5200 using only the BestComm DMA engine. The engine is more flexible than a traditional DMA engine because it can handle some data structures such as buffer descriptor rings.

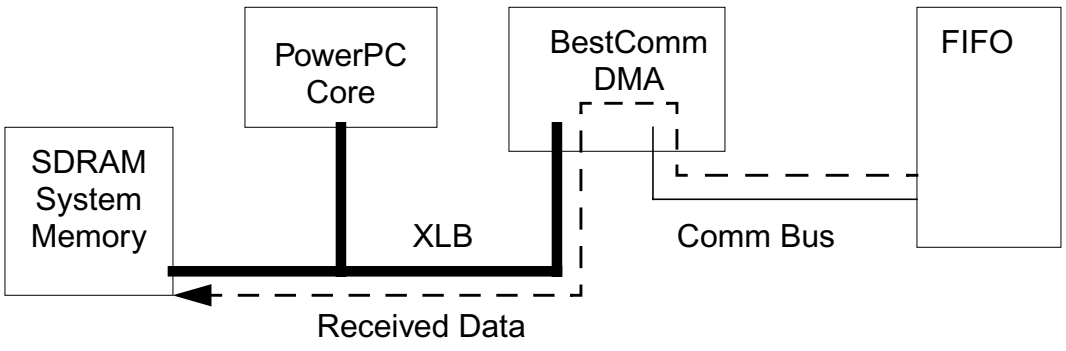
In order to begin a transfer, a task must be written for the BestComm DMA engine. The task must be set up with initial parameters and then enabled to begin the transfer. Usually the task can notify the CPU with an interrupt when it is done with the transfer, but it does not necessarily need to.

1.1.2 Why Use BestComm?

The MPC5200 is a multi-processor system with a PowerPC G2_LE processor and a BestComm Direct Memory Access (DMA) co-processor. Some applications require moving large blocks of data from one place in memory to another. For example, reading audio data from an audio peripheral requires a large transfer from a FIFO register to a buffer in memory. While the PowerPC is designed to efficiently execute instructions and perform computations on data in internal registers, the BestComm is designed to perform memory transfers.

The PowerPC and BestComm will contend for memory transfers on the same internal bus (XLB). Please see the figure below to show the data path inside the MPC5200.

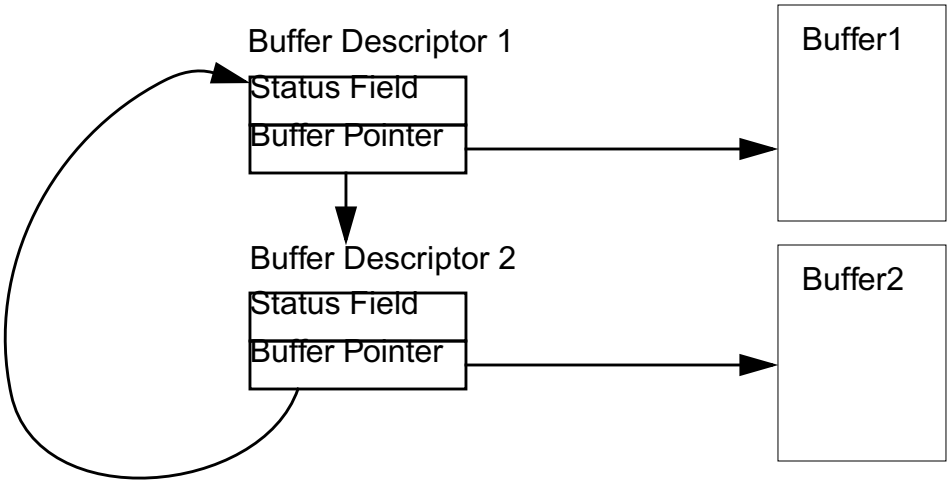
Figure 1-1. MPC5200 Internal Data Path



While the PowerPC is processing data, the BestComm can continue to move new data every XLB clock cycle. In addition, the BestComm is capable of performing some simple algorithms and data manipulation. For example, a common technique for handling data

buffers is to use a buffer descriptor ring. The figure below describes the buffer ring technique used in the BestComm buffer descriptor tasks.

Figure 1-2. Buffer Descriptors



The BestComm can be used to manage the buffer descriptors without taking processing from the PowerPC CPU. The status fields usually describe if the buffer is available to the CPU or if it is in use by the BestComm DMA engine. This data structure is flexible enough to allow applications to easily change the number of buffers that the DMA will operate on if necessary.

1.1.3 Why Use the BestComm API?

The BestComm DMA engine performs data transfers by executing firmware organized into separate tasks called an image loaded into the BestComm 16k SRAM. The MPC5200 BestComm can have 16 tasks enabled at once. This means that the BestComm can only execute one at a time, and it must determine which task will execute based on the priorities of the signals from the peripheral FIFOs. To understand more about how the BestComm operates in the MPC5200 please see Application Note *AN2604/D Introduction to BestComm*.

The BestComm API will help application programmers integrate BestComm tasks into their systems by providing functions to load and operate BestComm tasks. A task loader function loads a task image into the BestComm SRAM. The current release of the API comes with two task images with 16 tasks each providing most of the functionality needed by application developers.

With the functionality of the BestComm API, an application developer can quickly develop applications to utilize the MPC5200 BestComm without having to understand all of the specific interface registers and tasks.

1.2 BESTCOMM API

The purpose of the BestComm API is to make utilizing the BestComm engine easier for the application or device driver writer. The API gives the user an interface to the BestComm tasks that is easier to understand and use.

The main part of the BestComm API is a collection of functions used by the application writer. These functions will set up tasks, start and finish data transfers, and check on the status of tasks. A detailed description of the API is given at the end of this document.

1.2.1 BestComm API Files

The API is delivered as a set of source code, header files and a task image. The files listed in the following table are required for linking to create an executable using the API. There are two task images currently included in the API release. They are called image_rtos1 and image_rtos2 and have separate directories.

The only header file that needs to be included in source code is “capi/bestcomm_api.h”. The following tables describe the include paths and files needed to build an application using the BestComm API.

Include paths required	Description
capi	Contains bestcomm_api.h and code for the api.
code_dma/<image_name>	Contains a BestComm task image that is loaded into the BestComm SRAM. <image_name> is either image_rtos1 or image_rtos2 depending on what task image is used.

Table 1-1. Include paths required for Make

Files to be linked	Description
capi/load_task.c	Contains function to load the task image into the desired memory location. This file is needed if TasksLoadImage() is called.
capi/bestcomm_api.c	Contains function implementations for the api.
capi/task_api/ tasksetup_bdtable.c	Contains the setup function used by buffer descriptor tasks.
code_dma/<image_name>/ task_capi/tasksetup_*.c	There is one tasksetup_*.c file for each task in the image. Include only those corresponding to the tasks used by the application.

Table 1-2. Files necessary for linking in Make

Files to be linked	Description
code_dma/<image_name>/ dma_image.c	Contains structures for task variables.
code_dma/<image_name>/ dma_image.reloc.c	Contains a task image to be loaded into memory by the TasksLoadImage() function call.

Table 1-2. Files necessary for linking in Make

SECTION 2
USING THE BESTCOMM API

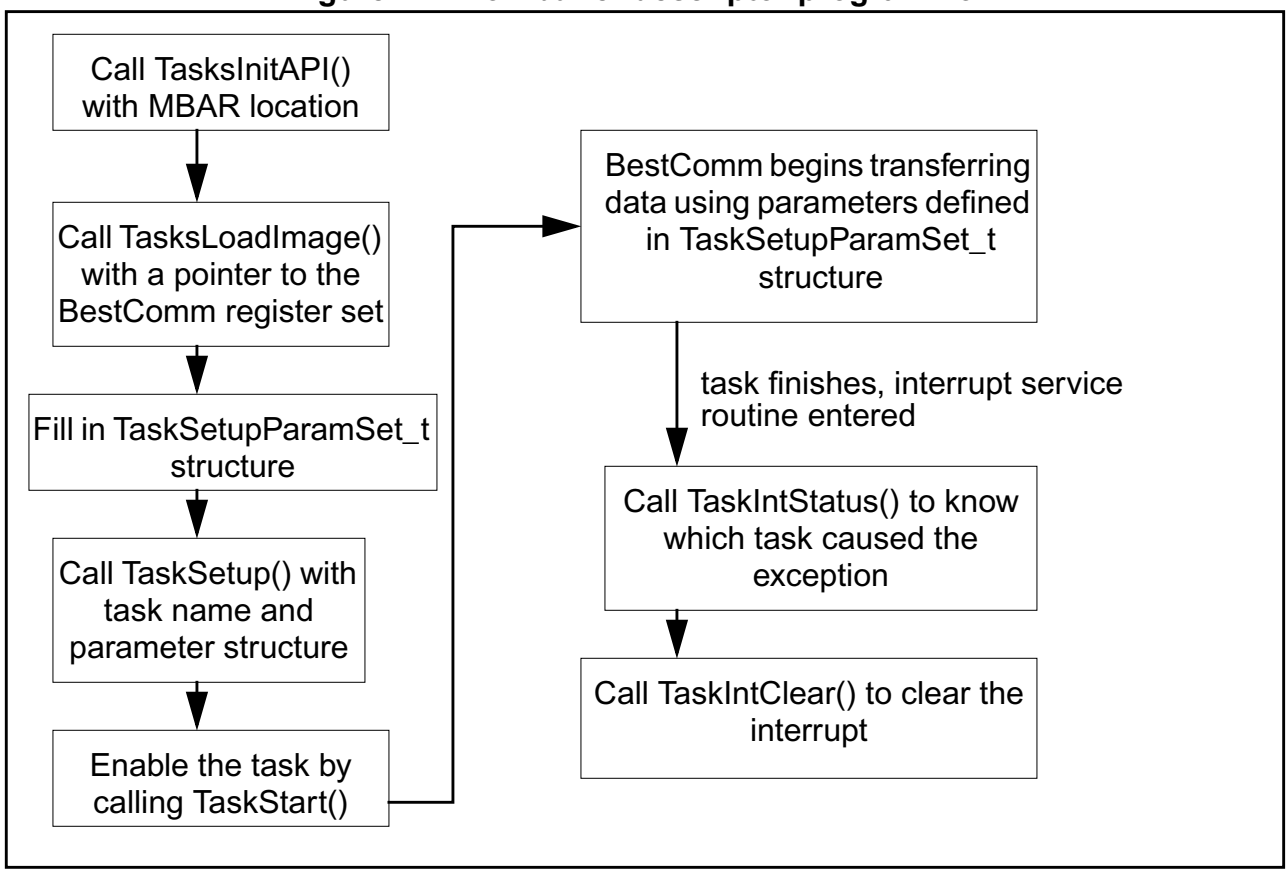
2.1 GENERAL PROGRAM FLOW

There are two types of tasks that have different usage models: buffer descriptor tasks, and non-buffer descriptor tasks. The next two sections will include flow diagrams and sample code for each usage model.

2.2 NON-BUFFER DESCRIPTOR TASKS

The non-buffer descriptor tasks are designed to transfer data in blocks one at a time. The simple program flow is given below. Figure 2-1 shows a flow if the tasks have not been loaded yet. If they have already been loaded then the diagram would start with filling in the TaskSetupParamSet_t structure.

Figure 2-1. Non-buffer descriptor program flow



If another transfer is necessary then TaskSetup() and TaskStart() must be called again. Starting a new transfer can take place in the interrupt service routine or not.

2.2.1 Non-buffer descriptor sample code

The example code given below demonstrates using the API to transfer a block of data using a non-buffer descriptor task. The sample code transfers 1024 bytes from physical address 0x50000 to the MPC5200 PSC1 transmit FIFO and generates an interrupt when finished.

Figure 2-2. Non-buffer descriptor sample code listing

```
#include "bestcomm_api.h"
#include "mgt5200/mgt5200.h"
#include "mgt5200/sdma.h"
#include "mgt5200/int_ctrl.h"
#include "exc5xxx.h"

#define MBAR 0xF0000000
#define SDMA_REG_OFFSET 0x1200
#define SRAM_OFFSET 0x8000
#define PSC1_TX_FIFO_ADDRESS (MBAR + 0x200C)


int interrupted = 0;
TaskSetupParamSet_t PSC1TaskParam;
TaskId PSC1TXTaskId;
sdma_regs *sdma;
int_ctrl_regs *int_ctrl;

/*
 * Interrupt routine prototype
 */

int main_interrupt_routine(void *arg0, void *arg1);

void main()
{


      /*
       * Register Exception. Bestcomm is a critical exception
       */
      exceptionRegister (EXC_CRITICAL_INT, 4, main_interrupt_routine, NULL,
NULL);
      int_ctrl = (int_ctrl_regs *)(MBAR + MBAR_INT_CTRL);
      int_ctrl->pimsk &= (~(PIMSK_BESTCOMM));

      /*
       * The API needs to be initialized before any other calls.
       * This needs to be passed the value of MBAR for the MPC5200.
       */

       TasksInitAPI((uint8 *) MBAR);

      /*
       * Somewhere before task_setup, the load task should be called
```

```
 * with the sdma register location after the taskbar has been
 * loaded with the destination address of the task image. In
 * this instance, the address is the beginning of SRAM.
 */
sdma = (sdma_regs *)(MBAR + SDMA_REG_OFFSET);
sdma->taskBar = MBAR + SRAM_OFFSET;

TasksLoadImage(sdma);

/*
 * Now the task is setup by filling out the parameter struct
 */

/*
 * We want to transfer from memory to the PSC1 module
 * For this we will use the General Single Pointer TX task
 */

/*
 * Since this is a general task the initiator must be setup
 */
PSC1TaskParam.Initiator = INITIATOR_PSC1_TX;

/*
 * Enter the number of bytes to transfer
 */
PSC1TaskParam.Size.NumBytes = 1024;

/*
 * Enter the source address of the data
 */
PSC1TaskParam.StartAddrSrc = 0x50000;

/*
 * Transfer a byte at a time
 */
PSC1TaskParam.IncrSrc = 1;
PSC1TaskParam.SzSrc = SZ_UINT8;
PSC1TaskParam.StartAddrDst = PSC1_TX_FIFO_ADDRESS;
PSC1TaskParam.SzDst = SZ_UINT8;
PSC1TaskParam.IncrDst = 0;

PSC1TXTaskId = TaskSetup(TASK_GEN_DP_0, &PSC1TaskParam);

/*
 * Begin the transfer
 */
TaskStart(PSC1TXTaskId,TASK_AUTOSTART_DISABLE, 0 , TASK_INTERRUPT_ENABLE);

while (!interrupted);

exceptionRemove (main_interrupt_routine);
```

```
}


/*
 * The interrupt routine clears the interrupt.
 */
int main_interrupt_routine(void *arg0, void *arg1)
{
#pragma unused (arg0, arg1)
      /*
       * Check to see if task called the interrupt. The return for
       * TaskIntStatus is the task number if an interrupt is
       * pending.
       */
      if(TaskIntStatus(PSC1TXTaskId) == PSC1TXTaskId)
      {
            /*
             * Clear the task interrupt
             */
            TaskIntClear(PSC1TXTaskId);

            /*
             * Transfer is finished
             */
            interrupted = 1;

      }
      return 1;
}
```

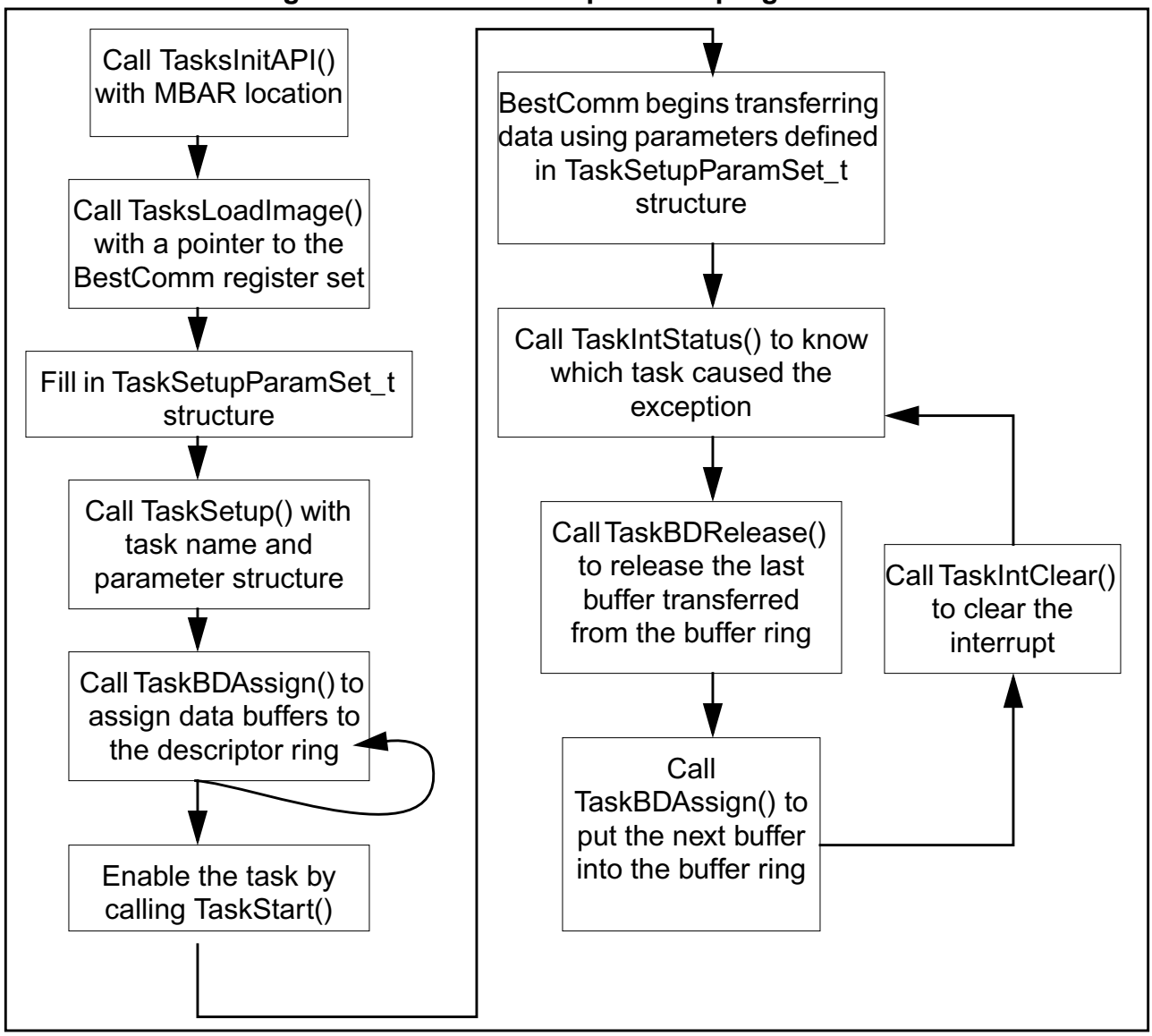
2.3 BUFFER DESCRIPTOR TASKS

In the buffer descriptor usage model, the buffers in the buffer descriptor ring can be pre-loaded for a transmit task (cleared for a receive task) using TaskBDAssign(). When TaskStart() is called the task will transfer the first buffer, generate an interrupt when finished and move to the next buffer in the ring. If the next buffer has not yet been assigned, the task will wait until TaskBDAssign() is called. TaskBDRelease() must be called to let the API know that a buffer descriptor is available for use again by TaskBDAssign().

The ownership of a buffer descriptor — the BestComm task or the application using the API — is determined by the "ready bit" (defined by SDMA_BD_MASK_READY) in the Status word. When set, the task owns the buffer descriptor and the application must not alter it. When clear, the descriptor is ready to be managed by the application. Using this lets the application traverse the buffer descriptor ring. The return value of TaskBDRelease() is an index to the next descriptor to be released. This index may be safely initialized to 0. The index can be translated to a pointer to a buffer descriptor using the TaskGetBD() function.

The diagram in Figure 2-3 illustrates what to do when the tasks need to be loaded. If the tasks do not need to be loaded then the diagram would start with filling in the TaskSetupParamSet_t structure.

Figure 2-3. Buffer descriptor task program flow



2.3.1 Buffer Descriptor Sample Code

The following sample code shows how to use the API with a buffer descriptor task. This sample uses two tasks. One receives data from the PSC2 RX FIFO into a memory buffer (0x60000). The other task transfers data from memory (0x50000) to the PSC2 TX FIFO. The buffer ring for each task that is set up has three buffers of 1024 bytes each. When a buffer is transmitted, the task will generate an interrupt and continue with the next buffer in the ring.

The interrupt routine determines what interrupt happened then releases the buffer and puts a new buffer in the ring using TaskBDAssign().

Figure 2-4. Buffer Descriptor Sample Code Listing

```
#include "bestcomm_api.h"
#include "mgt5200/mgt5200.h"
#include "mgt5200/sdma.h"
#include "mgt5200/int_ctrl.h"

#include "core5200.h"
#include "exc5xxx.h"


#define MBAR 0xF0000000
#define SDMA_REG_OFFSET 0x1200
#define SRAM_OFFSET 0x8000

TaskSetupParamSet_t tx_setup, rx_setup;

TaskIdtx_task_id, rx_task_id;

uint32 frags = 3;
uint32 buff_size = 1024;
uint8 *tx_phys_data;
uint8 *rx_phys_data;
BDIdx tx_next_bd, rx_next_bd;

sdma_regs *sdma;
int_ctrl_regs *int_ctrl;

int bcHandler (void *arg0, void *arg1);

void main()
{

      /*
       * Register Exception
       */
      exceptionRegister (EXC_CRITICAL_INT, 4, bcHandler, NULL, NULL);

      int_ctrl = (int_ctrl_regs *)(MBAR + MBAR_INT_CTRL);
      int_ctrl->pimsk &= (~(PIMSK_BESTCOMM));

      /*
       * Setup the source and destination pointers
       */
       tx_phys_data = (uint8 *) 0x50000;
       rx_phys_data = (uint8 *) 0x60000;

      /*
       * The API needs to be initialized before any other calls.
       * This needs to be passed the value of MBAR for the MPC5200.
```

```
 */
 TasksInitAPI((uint8 *) MBAR);

/*
 * Somewhere before task_setup, the load task should be called
 * with the sdma register location after the taskbar has been
 * loaded with the destination address of the task image. In
 * this instance, the address is the beginning of SRAM.
 */
sdma = (sdma_regs *)(MBAR + SDMA_REG_OFFSET);
sdma->taskBar = MBAR + SRAM_OFFSET;

TasksLoadImage(sdma);

/*
 *    Initialize the setup structure with defaults
 */
tx_setup.NumBD = frags;
tx_setup.Size.MaxBuf = buff_size;
tx_setup.Initiator = INITIATOR_PSC2_TX; /* Necessary for the general task */
tx_setup.StartAddrDst = 0xf000220C; /* Fifo address */
tx_setup.SzSrc = SZ_UINT32;/* Size of data in bytes */
tx_setup.IncrSrc = 4;/* Transfer to FIFO 4 bytes at a time*/
tx_setup.SzDst = SZ_UINT32;/* Size of data in bytes */

/*
 * Now the tx task can be setup using TaskSetup.
 */
tx_task_id=TaskSetup( TASK_GEN_TX_BD, &tx_setup );

/*
 * Assume that three buffers worth of data are ready to transfer,
 * so TaskBDAssign is used to describe them. Store the first buffer
 * descriptor index to retire.
 */
tx_next_bd = TaskBDAssign(tx_task_id,
       tx_phys_data, NULL, (int) buff_size, 0);
TaskBDAssign(tx_task_id,
      (tx_phys_data + (buff_size * 1)), NULL, (int)buff_size, 0);
TaskBDAssign(tx_task_id,
      (tx_phys_data + (buff_size * 2)), NULL, (int)buff_size, 0);

/*
 * Setup the receive task now.
 */
rx_setup.NumBD = frags;
rx_setup.Size.MaxBuf = buff_size;
rx_setup.Initiator = INITIATOR_PSC2_RX; /* The PSC2 receive initiator */

rx_setup.StartAddrSrc = 0xf000220C;
rx_setup.SzDst = SZ_UINT32;    /* Transfer size in bytes */
rx_setup.IncrDst = 4;  /* Increment for destination buffer */
rx_setup.SzSrc = SZ_UINT32;    /* Transfer size in bytes */
```

```
      rx_task_id = TaskSetup( TASK_GEN_RX_BD, &rx_setup );

      /*
       * There are three buffers available for the receive data. Also,
       * store the first receive buffer index to be filled.
       */
      rx_next_bd = TaskBDAssign(rx_task_id,
             rx_phys_data, NULL, (int)buff_size, 0);
      TaskBDAssign(rx_task_id,
             (rx_phys_data + (buff_size * 1)), NULL, (int)buff_size, 0);
      TaskBDAssign(rx_task_id,
             (rx_phys_data + (buff_size * 2)), NULL, (int)buff_size, 0);


      /*
       * Enable both tasks
       */
      TaskStart(rx_task_id, TASK_AUTOSTART_ENABLE, rx_task_id,
TASK_INTERRUPT_ENABLE);
      TaskStart(tx_task_id, TASK_AUTOSTART_ENABLE, tx_task_id,
TASK_INTERRUPT_ENABLE);
}

/*
 * Once this is enabled then the interrupt routine will be called after
 * every buffer descriptor is finished. The interrupt routine should
 * finish each buffer with TaskBDRelease which returns the next
 * free buffer in the ring.
 */

int bcHandler (void *arg0, void *arg1)
{
#pragma unused (arg0, arg1)

      static BDIdx bdi = 0;

      /*
       * Check transmit interrupts. This handler releases one buffer
       * descriptor at a time.
       */
      if (TaskIntStatus( tx_task_id ) == tx_task_id )
      {
            /*
             * Clear the task interrupt.
             */
            TaskIntClear( tx_task_id );

            /*
             * The return of TaskBDRelease is the next buffer available for use.
             */
            tx_next_bd = TaskBDRelease( tx_task_id);

            /*
             * Passing the next available buffer for use
```

```
             */
            TaskBDAssign( tx_task_id,
             (tx_phys_data + (buff_size * tx_next_bd)), NULL, (int)buff_size, 0);

        }

        /*
         * Check receive interrupts. This handler attempts to release as
         * many buffer descriptors as are available.
         */
        if (TaskIntStatus( rx_task_id ) == rx_task_id )
        {
            TaskBD1_t *bd;

            /*
             * Clear the task interrupt.
             */
            TaskIntClear( rx_task_id );

            /*
             * Get a pointer to the next buffer to be released
             * tracked by the bdi static variable.
             */
            bd = (TaskBD1_t *)TaskGetBD( rx_task_id, bdi );

            /*
             * Release as many buffers as possible.
             */
            while( !(bd->Status & SDMA_BD_MASK_READY) ) {
                  /*
                   * Handle the incoming packet.
                   */

                  /*
                   * Finished with the buffer descriptor.
                   */
                  bdi = TaskBDRelease( rx_task_id );

                  /*
                   * Assign a new, empty buffer to the RX ring.
                   */
                  TaskBDAssign( rx_task_id,
                        (rx_phys_data + (buff_size * rx_next_bd)),
                        NULL, (int)buff_size, 0);

                  bd = (TaskBD1_t *)TaskGetBD( rx_task_id, bdi );
            }
        }
       return 1;
}
```

2.4 USING VIRTUAL MEMORY AND MULTIPLE PROCESSES

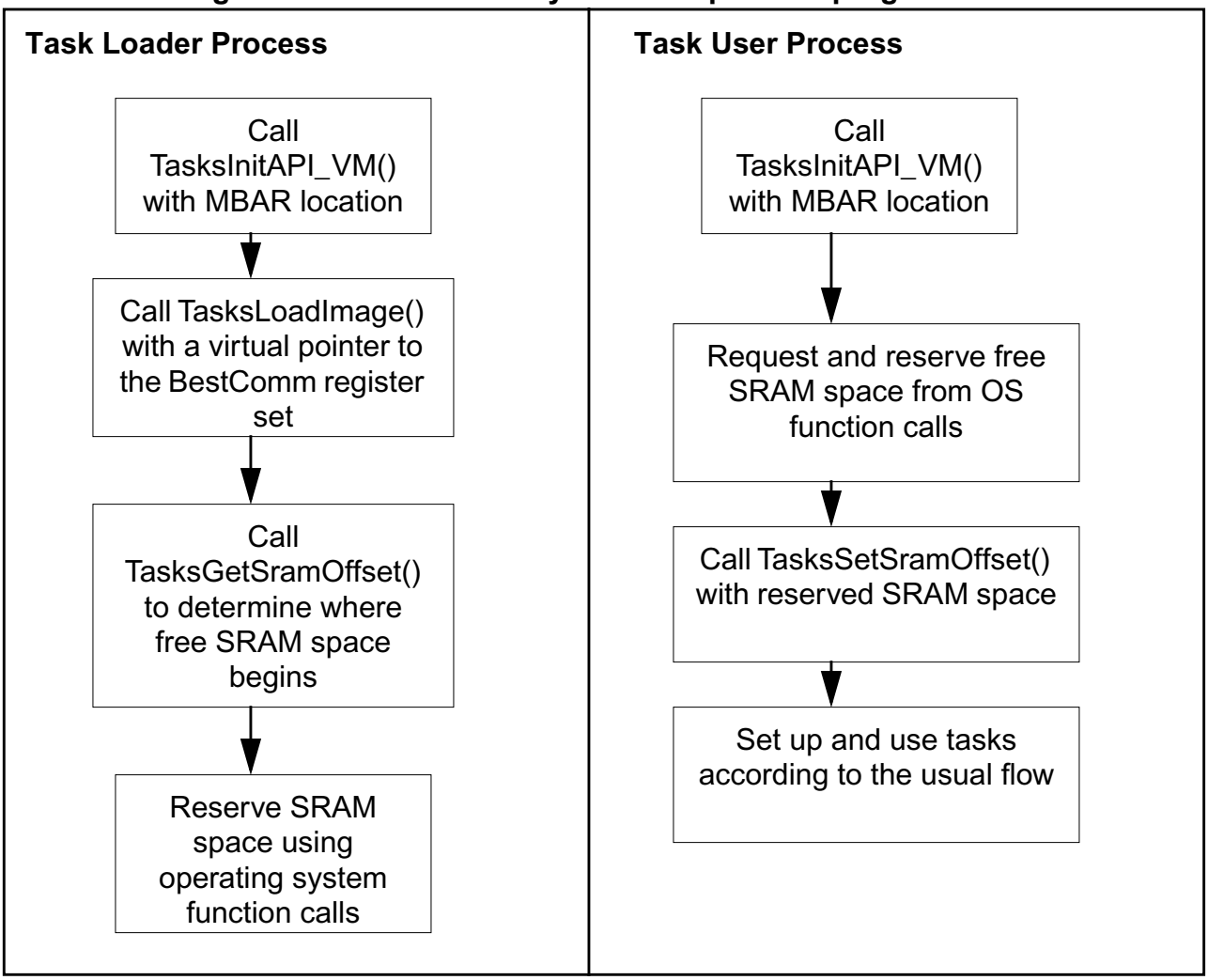
Some operating systems (e.g. microkernels) use virtual memory and multiple processes within their device driver model. These features require special consideration when using the BestComm API. Some functions of the API were designed to handle these cases.

Instead of calling TasksInitAPI(), TasksInitAPI_VM() is called with the physical location and the virtual location of the register map. Any addresses that will be used by the BestComm tasks must be physical since the BestComm does not use virtual memory. These are usually the destination and source addresses.

Since the tasks and the API use the internal SRAM, each process must be aware of what area in SRAM has already been used by other processes. Usually, an OS will have a way to keep track of shared resources. These can be used with the TasksGetSramOffset() and TasksSetSramOffset() API calls. The use of these calls are demonstrated in the sample code that was written to work in QNX Neutrino.

Typically, the task loader process is separate from the processes that use the tasks. The following diagram shows the program flow when the API is used with QNX Neutrino. Only the API initialization is shown. Once the API is initialized, the program flow follows one of the flows described in the earlier section.

Figure 2-5. Virtual memory and muti-process program flow



2.4.1 Virtual Memory and Multi-Process Sample Code

The following example demonstrates using the API in a virtual memory environment. The sample code was written for QNX Neutrino 2.1. Instead of using TasksInitAPI() this example uses TasksInitAPI_VM() which takes two addresses for the entire MPC5200 register space: virtual and physical. The virtual address should be mapped into the OS as non-cacheable, shareable, and contiguous.

The example contains two separate applications to show how to use the TasksSetSramOffset() and TasksGetSramOffset() functions. The BestComm API uses the MPC5200 SRAM to store buffer descriptor tables. Each buffer descriptor table is allocated during TaskSetup(). In a multi-process environment, the area reserved by the BestComm API for the table will not be known by other processes. Therefore, it is the responsibility of

the programmer to keep track of reserved SRAM. In this example system calls are provided by Neutrino for this. NOTE: TasksSetSramOffset() must be called before TaskSetup().

Figure 2-6. Virtual Memory and Multi-Process Sample Code Listing

Task Loader

```
#define DMA_TASK 0

#include <stdio.h>
#include "../capi/bestcomm_api.h"

#include "mgt5200/xlb_arb.h"
#include "mgt5200/mgt5200.h"
#include "mgt5200/int_ctrl.h"
#include "mgt5200/sdma.h"
#include "mgt5200/cdm.h"
#include "mgt5200/psc.h"


/*****************************************************************************/

#include <stdlib.h>
#include <sys/mman.h>
#include <stdio.h>
#include <sys/rsrcdbmgr.h>
#include <sys/rsrcdbmsg.h>

#define SOURCE_DMA_ADDRESS 0x50000
#define DEST_DMA_ADDRESS0x60000
#define DMA_WORDS256
#define MBAR 0xF0000000UL

#define SRAM_OFFSET 0x8000UL
#define SRAM_SIZE8192


int_ctrl_regs *int_ctrl;
sdma_regs *sdma;
cdm_regs *cdm;
psc_regs *psc1;
TaskSetupParamSet_t dma_task_param;
TaskId dma_task_id;

off_t src_phys;
off_t dst_phys;

rsrc_alloc_t ralloc;

int *src_addr, *dst_addr;

int main(void)
```

```
{


      int *i,*j;
      int count;
      int test;
      int x,y;
      int bdidx;
      uint8 *v_mbar;
      uint32 sram_offset;

      printf("DMA Task Loader\n");

      /* Inititalize Buffers */
      count = 0;
      src_addr = (int *) mmap( 0, DMA_WORDS * 4,
      PROT_READ|PROT_WRITE|PROT_NOCACHE, MAP_ANON,
            NOFD, 0);
      dst_addr = (int *) mmap( 0, DMA_WORDS * 4,
      PROT_READ|PROT_WRITE|PROT_NOCACHE, MAP_ANON,
            NOFD, 0);

      if (src_addr == MAP_FAILED)
            printf("Source Map Failed\n");
      if (dst_addr == MAP_FAILED)
            printf("Destination Map Failed\n");

      for(count=0; count < DMA_WORDS; count++)
      {
            src_addr[count]=count;
            dst_addr[count]=0;
      }


      /* Create mmap for entire register space */
      v_mbar = (uint8 *) mmap_device_memory( 0, 0xC000, \
            PROT_READ|PROT_WRITE|PROT_NOCACHE, 0, MBAR);

      /* Initialize the API using virtual memory */

      printf(" Before TaskInit %08x\n",v_mbar);

      TasksInitAPI_VM((uint8 *)v_mbar, (uint8 *)MBAR);

      /*Loading the tasks */
      sdma=(sdma_regs *) (v_mbar + MBAR_SDMA);

      /* This must be physical location of SRAM*/
      sdma->taskBar = (MBAR + MBAR_SRAM);

      printf("Before Loading \n");
    TasksLoadImage(sdma);
    printf("After Loading \n");
```

```
    /* Call TasksGetSramOffset() to get the free SRAM address after the tasks. */
    sram_offset = TasksGetSramOffset();

    printf("Sram Offset = %d\n", sram_offset);

    /* Seed the SRAM resource in the system database */
    memset(&ralloc, 0, sizeof(ralloc));
    ralloc.start    = MBAR + SRAM_OFFSET + sram_offset;
    ralloc.end      = MBAR + SRAM_OFFSET + SRAM_SIZE - 1;
    ralloc.flags    = RSRCDBMGR_FLAG_NAME |
            RSRCDBMGR_FLAG_NOREMOVE;
    ralloc.name     = "mpc5200_sram";

    if (rsrcdbmgr_create(&ralloc, 1) == -1) {
         perror("Unable to seed SRAM resource: ");
         exit(EXIT_FAILURE);
    }
    else
    {
      printf("SRAM allocated to system\n");
    }
}
```

Task User Application

```
#include <stdio.h>
#include "../capi/bestcomm_api.h"

#include "mgt5200/xlb_arb.h"
#include "mgt5200/mgt5200.h"
#include "mgt5200/int_ctrl.h"
#include "mgt5200/sdma.h"
#include "mgt5200/cdm.h"
#include "mgt5200/psc.h"



/*****************************************************************************/


#include <sys/mman.h>
#include <stdio.h>
#include <sys/rsrcdbmgr.h>
#include <sys/rsrcdbmsg.h>

#define SOURCE_DMA_ADDRESS 0x50000
#define DEST_DMA_ADDRESS0x60000
#define DMA_WORDS256
#define MBAR 0xF0000000UL


volatile int IntDone;
```

```
int_ctrl_regs *int_ctrl;
sdma_regs *sdma;
cdm_regs *cdm;
psc_regs *psc1;
TaskSetupParamSet_t dma_task_param;
TaskId dma_task_id;

off_t src_phys;
off_t dst_phys;

int *src_addr, *dst_addr;

rsrc_request_t  req = {0};
rsrc_alloc_t qu = {0};


int main(void)
{


      int *i,*j;
      int count;
      int test;
      int x,y;
      int bdidx;
      uint8 *v_mbar;

      printf("Multi DMA Test\n");


      /* Inititalize Buffers */

      IntDone=0;
      count = 0;
      src_addr = (int *) mmap( 0, DMA_WORDS * 4,
PROT_READ|PROT_WRITE|PROT_NOCACHE, MAP_ANON,
            NOFD, 0);
      dst_addr = (int *) mmap( 0, DMA_WORDS * 4,
PROT_READ|PROT_WRITE|PROT_NOCACHE, MAP_ANON,
            NOFD, 0);

      if (src_addr == MAP_FAILED)
            printf("Source Map Failed\n");
      if (dst_addr == MAP_FAILED)
            printf("Destination Map Failed\n");

      for(count=0; count < DMA_WORDS; count++)
      {
            src_addr[count]=count;
            dst_addr[count]=0;
      }


      /* Create mmap for entire register space */
```

```
    v_mbar = (uint8 *) mmap_device_memory( 0, 0xC000, \
            PROT_READ|PROT_WRITE|PROT_NOCACHE, 0, MBAR);



      /* Initialize the API using virtual memory */

      printf(" Before TaskInit %08x\n",v_mbar);

      TasksInitAPI_VM((uint8 *)v_mbar, (uint8 *)MBAR);

    sdma=(sdma_regs *) (v_mbar + MBAR_SDMA);

      /* This must be the physical location of the SRAM */
      sdma->taskBar = (MBAR + MBAR_SRAM);

      printf("Here\n");
      /* TasksAttachImage() is used if the tasks have already been loaded by
      another process */
      TasksAttachImage(sdma);

      memset(&req, 0, sizeof(req));
      /* This must be the largest possible amount of buffers */
      /* In this example the value 2 in the next statement is the MAX_BD define
for the */
      /* TASK_GEN_DP_BD_0 task and TaskBD2_t is the structure used */
      /*  for the BD table. Other tasks that work with a FIFO */
      /*  may use TaskBD1_t */

      req.length = 2 * sizeof(TaskBD2_t);
      req.align = 4;
      req.flags = RSRCDBMGR_FLAG_ALIGN | RSRCDBMGR_FLAG_NAME |
RSRCDBMGR_FLAG_NOREMOVE;
      req.name = "mpc5200_sram";

      if (rsrcdbmgr_attach(&req, 1) == -1) {
            perror("sram alloc");
      }

      printf("req.start = 0x%llx offset = %d req.length = %llu\n",req.start,
(uint32)(req.start - (MBAR + MBAR_SRAM)), req.length );
      /* Tell api where free SRAM starts using TasksSetSramOffset before calling
TaskSetup */
      TasksSetSramOffset((uint32)(req.start - (MBAR + MBAR_SRAM)));

    /* Set the physical address of the dma pointer */
      if(mem_offset(src_addr, NOFD, 1, &src_phys, 0) == -1) {
         printf("Memory problem finding phys addr for source pointer\n");
    }

    /* Set the physical address of the dma pointer */
      if(mem_offset(dst_addr, NOFD, 1, &dst_phys, 0) == -1) {
         printf("Memory problem finding phys addr for destination pointer\n");
    }
```

```
    dma_task_param.Size.MaxBuf = DMA_WORDS * 4; /* bytes */
    dma_task_param.NumBD = 1;
    dma_task_param.Initiator = INITIATOR_ALWAYS;
    dma_task_param.StartAddrSrc = src_phys;
    dma_task_param.StartAddrDst = dst_phys;
    dma_task_param.IncrSrc = 4;
    dma_task_param.IncrDst = 4;
    dma_task_param.SzSrc = SZ_UINT32;
    dma_task_param.SzDst = SZ_UINT32;

    /* Setup the task */

    printf("Task Setup Next \n");
      dma_task_id = TaskSetup(TASK_GEN_DP_BD_0, &dma_task_param);

      printf("TaskGetSramOffset = %d\n",TasksGetSramOffset());
      printf("Task BDAssign is next\n");

      bdidx = TaskBDAssign(dma_task_id, (void *) src_phys, (void *) dst_phys, 4 *
      DMA_WORDS, NULL);

      printf("bdidx = %d\n", bdidx);

      printf("Task Start Next\n");

      /* Start task without autostart and interrupts.*/
      TaskStart(dma_task_id, TASK_AUTOSTART_DISABLE, 0, TASK_INTERRUPT_DISABLE);

      /* Wait for interrupt count. This is just simulating an interrupt */

      printf("Waiting for Interrupt Count\n");

      while(IntDone != 10000) { IntDone++; }

      printf("Done\n");
      TaskBDRelease(dma_task_id);
      printf("Buffer Descriptor Released\n");

      test=0;
      for(count=0; count < DMA_WORDS; count++)
      {

            if((src_addr[count]) != (dst_addr[count]))
                  test++;
      }

      printf("Test Failures = %d %s\n",test,(test? "FAIL" : "PASS"));

/* Infinite loop to test resource allocation since SRAM resource is */
/* automatically deallocated when the process exits. */

      while(1) {
            sleep(10);
```

```
        }
}
```

SECTION 3
BESTCOMM TASK DESCRIPTIONS

3.1 STANDARD TASK IMAGES

The BestComm API includes two standard task images that can be used by the user for applications that need BestComm. The BestComm task image is a collection of microcode that runs on the BestComm engine to perform a task. The API includes two versions of the image named image_rtos1 and image_rtos2. They can be found under the code_dma directory.

These images are loaded into the BestComm SRAM when TasksLoadImage() is called. The microcode is found in the file named dma_image.reloc.c under the code_dma/image_rtos1 or code_dma/image_rtos2 directory, and this file must be included by the executable that calls TaskLoadImage().

For a detailed description of the tasks and what fields need to be filled in for TaskSetup() please see Appendix A.

3.1.1 Task Image: image_rtos1

The following table describes the tasks included in the image_rtos1 task image. The TaskName_t refers to the task name define that should be included as a parameter to TaskSetup(). The task descriptions are given in Appendix A.

TaskName_t	Task Description
TASK_PCI_TX	PCI TX
TASK_PCI_RX	PCI RX
TASK_FEC_TX	Ethernet TX
TASK_FEC_RX	Ethernet RX
TASK_LPC	General Dual-Pointer
TASK_ATA	General Dual-Pointer Buffer Descriptor
TASK_CRC16_DP_0	General Dual-Pointer + CRC
TASK_CRC16_DP_1	General Dual-Pointer + CRC
TASK_GEN_DP_0	General Dual-Pointer
TASK_GEN_DP_1	General Dual-Pointer
TASK_GEN_DP_2	General Dual-Pointer
TASK_GEN_DP_3	General Dual-Pointer
TASK_GEN_TX_BD	General Dual-Pointer Buffer Descriptor TX
TASK_GEN_RX_BD	General Dual-Pointer Buffer Descriptor RX

Table 3-1. Task Image for image_rtos1

TaskName_t	Task Description
TASK_GEN_DP_BD_0	General Dual-Pointer Buffer Descriptor
TASK_GEN_DP_BD_1	General Dual-Pointer Buffer Descriptor

Table 3-1. Task Image for image_rtos1

3.1.2 Task Image: image_rtos2

The following table describes the tasks included in the image_rtos2 task image. The TaskName_t refers to the task name define that should be included as a parameter to TaskSetup(). The task descriptions are given in Appendix A.

TaskName_t	Task Description
TASK_PCI_TX	PCI TX
TASK_PCI_RX	PCI RX
TASK_FEC_TX	Ethernet TX
TASK_FEC_RX	Ethernet RX
TASK_LPC	General Dual-Pointer
TASK_ATA	General Dual-Pointer Buffer Descriptor
TASK_CRC16_DP	General Dual-Pointer + CRC
TASK_CRC16_DP_BD	General Dual-Pointer Buffer Descriptor + CRC
TASK_GEN_DP_0	General Dual-Pointer
TASK_GEN_DP_1	General Dual-Pointer
TASK_GEN_DP_2	General Dual-Pointer
TASK_GEN_DP_3	General Dual-Pointer
TASK_GEN_TX_BD_0	General Dual-Pointer Buffer Descriptor TX
TASK_GEN_RX_BD_0	General Dual-Pointer Buffer Descriptor RX
TASK_GEN_TX_BD_1	General Dual-Pointer Buffer Descriptor TX
TASK_GEN_RX_BD_1	General Dual-Pointer Buffer Descriptor RX

Table 3-2. Task Image for image_rtos2

SECTION 4
BESTCOMM API DEFINITIONS

4.1 FUNCTION DESCRIPTIONS

This section will describe the functions that are available in the API and give a brief description of what they are used for. Please see doc/BestCommAPIReference/globals.html for a more detailed description.

4.1.1 Initialization Functions

*TasksInitAPI(uint8 *MBarRef)*

API initialization function used when virtual memory is not used. The input parameter is the base address of the MPC5200 register map (MBAR). This function MUST be called before any of the other functions are called.

*TasksInitAPI_VM(uint8 *MBarRef, uint8 *MBarPhys)*

API Initialization function used when virtual memory is used. One parameter is the virtual address and the other is the physical address of the register map. This function MUST be called before any of the other functions are called.

4.1.2 Task Loader Functions

*TasksLoadImage(sdma_regs *sdma)*

Loads task image into the BestComm SRAM. The input parameter is the address of the TaskBase register in the BestComm. Before calling this function, the TaskBase Register should contain the physical destination address where the image will be loaded. The image should be loaded into the beginning of SRAM.

*TasksAttachImage(sdma_regs *sdma)*

TasksAttachImage() is deprecated. Its functionality has been moved into TaskSetup() but it remains in the API for backward compatibility.

4.1.3 Multiple Process Helper Functions

TasksSetSramOffset(uint32 sram_offset)

This function sets the beginning Sram offset that is used in the API. This function is only used when another process using the API is also running in the system. The sample code illustrates how to use this function.

TasksGetSramOffset(uint32 sram_offset)

This function returns the offset set aside in the sram by the API. The sample code illustrates how to use this function.

4.1.4 Task Related Functions

*TaskSetup(TaskName_t TaskName, TaskSetupParamSet_t *TaskParams)*

The *TaskSetup()* function prepares a task for use. The TaskName is the name of the task to use. The TaskParams structure should be filled in before calling this function. This function returns a handle to the task to be used by other API functions.

TaskStart(TaskId taskId, uint32 autoStartEnable, TaskId autoStartTask, uint32 intrEnable)

This function starts the task represented by taskId with the option to auto start another task or the same task when done. The task can also be started with interrupts enabled.

TaskStop(TaskId taskId)

This function stops the task represented by taskId.

TaskStatus(TaskId taskId)

This function returns the enable/disable status.

4.1.5 Task Interrupt-Related Functions

TaskIntSource(void)

This function will return the taskId of the interrupting function.

TaskIntStatus(TaskId taskId)

This function returns the interrupt status by returning the taskId if the task caused an interrupt of the task represented by taskId. A more intuitive function call is *TaskIntPending()*.

TaskIntPending(TaskId taskId)

This function returns the pending interrupt status of taskId with a 0 for no interrupt or a 1 for a pending interrupt.

TaskIntClear(TaskId taskId)

This function will clear the interrupt for a the task represented by taskId.

4.1.6 Buffer Descriptor Related Functions

*TaskBDAssign(TaskId taskId, void *buffer0, void *buffer1, int size, uint32 bdFlags)*

This function assigns a buffer descriptor to the buffer ring for the buffer descriptor task represented by taskId.

TaskBDRelease(TaskId taskId)

This function removes the last buffer descriptor that was used from the buffer ring for the task represented by taskId.

TaskGetBD(TaskId taskId, BDIdx bd)

This function returns a pointer to the buffer descriptor structure defined by taskId and BDIdx.

TaskGetBDRing(TaskId taskId)

This function returns a pointer to the beginning of the buffer descriptor ring for the task represented by taskId.

TaskBDReset(TaskId taskId)

This function clears the ready bit on all buffer descriptors in the ring and resets the task and API buffer descriptor pointers to zero.

4.2 STRUCTURE DEFINITIONS

4.2.1 TaskSetupParamSet_t struct

Members

uint32 NumBD
Number of buffer descriptors used in a buffer descriptor task. If the task is a non-buffer descriptor task then this field is not used.

union uint32 Size
Union with MaxBuf or NumBytes. MaxBuf is used for the maximum buffer size (bytes) in a a buffer descriptor task. NumBytes is used for non-buffer descriptor tasks to describe the buffer size in bytes.

MPC5200Initiator_t Initiator
An MPC5200Initiator type that is used by general tasks that interface to a peripheral FIFO. Each peripheral FIFO has a hardware initiator that the Bestcomm uses to gate the transfer to and from the peripheral FIFO.

Table 4-1. Initiator Definitions

Enumeration values
INITIATOR_ALWAYS
INITIATOR_SCTMR_0
INITIATOR_SCTMR_1
INITIATOR_FEC_RX
INITIATOR_FEC_TX
INITIATOR_ATA_RX
INITIATOR_ATA_TX
INITIATOR_SCPCI_RX
INITIATOR_SCPCI_TX
INITIATOR_PSC3_RX
INITIATOR_PSC3_TX
INITIATOR_PSC2_RX
INITIATOR_PSC2_TX
INITIATOR_PSC1_RX
INITIATOR_PSC1_TX
INITIATOR_SCTMR_2
INITIATOR_SCLPC
INITIATOR_PSC5_RX
INITIATOR_PSC5_TX
INITIATOR_PSC4_RX
INITIATOR_PSC4_TX
INITIATOR_I2C2_RX
INITIATOR_I2C2_TX
INITIATOR_I2C1_RX
INITIATOR_I2C1_TX
INITIATOR_PSC6_RX
INITIATOR_PSC6_TX
INITIATOR_IRDA_RX
INITIATOR_IRDA_TX
INITIATOR_SCTMR_3
INITIATOR_SCTMR_4
INITIATOR_SCTMR_5
INITIATOR_SCTMR_6
INITIATOR_SCTMR_7

uint32 StartAddrSrc
Address of the DMA source buffer. This can be a hardware register such as a peripheral FIFO.

uint32 StartAddrDst
Address of the DMA destination buffer. This can be a hardware register such as a peripheral FIFO.

sint16 IncrSrc
DMA source pointer increment amount in bytes. This usually matches the SzSrc element unless the source address is a peripheral FIFO register. If this is the case the increment should be 0.

sint16 IncrDst
DMA destination pointer increment amount in bytes. This usually matches the SzDst element unless the destination address is a peripheral FIFO register. If this is the case the increment should be 0.

Sz_t SzDst
The DMA transfer size in bytes for the destination pointer. Sz_t can be SZ_UINT8, SZ_UINT16, or SZ_UINT32.SzSrc and SzDst should usually be set the same.

Sz_t SzSrc
The DMA transfer size in bytes for the source pointer. Sz_t can be SZ_UINT8, SZ_UINT16, or SZ_UINT32. SzSrc and SzDst should usually be set the same.

APPENDIX A
TASK DESCRIPTIONS

A.1 TASK DESCRIPTIONS

This section describes the operation of the tasks in the included standard images. They are organized by name giving a description then a listing of the TaskSetupParamSet_t structure members that need to be initialized.

A.1.1 PCI TX

The PCI TX task will transfer data from memory to the PCI bus. This task also writes the Packet Size register in the PCI module with the number of bytes to transfer. An interrupt is generated when all of the buffer has been transferred to the PCI transmit FIFO if the interrupt is enabled when the task is started by TaskStart().

TaskSetupParamSet_t structure elements that must be initialized:

Size.NumBytes - Number of bytes in the memory buffer to transfer

StartAddrSrc - Address of source buffer to be transferred

IncrSrc - Transfer size in bytes

SzSrc - Transfer size in bytes

A.1.2 PCI RX

The PCI RX task will read data from the PCI Receive FIFO and write it to memory. An interrupt is generated when the task is finished transferring the requested amount of data to memory when enabled by TaskStart().

TaskSetupParamSet_t structure elements that must be initialized:

Size.NumBytes - Number of bytes in the memory buffer to transfer

StartAddrDst - Address of source buffer to be transferred

IncrDst - Transfer size in bytes

SzDst - Transfer size in bytes

A.1.3 Ethernet TX

The Ethernet TX task will transfer data from memory to the FEC module using a buffer descriptor data structure that contains a pointer to the data buffer and a status word. This task uses the bdFlags parameter in TaskBDAssign(). An interrupt is generated by the FEC module when a frame has been transmitted, so the task interrupt should not be enabled whenTaskStart() is called. This task never ends, so it does not need to be started with auto start enabled.

The Ethernet TX task keeps the FEC Transmit FIFO full when a buffer is available for transfer. The task continuously checks the TASK_BD_TFD flag which is set when TaskBDAssign() is called. When the flag is set, the task will begin transferring the buffer pointed to by the buffer descriptor. When the buffer has been transferred, the task will go to the next buffer descriptor in the ring to check the TASK_BD_TFD flag.

TaskSetupParamSet_t structure elements that must be initialized:

NumBD - number of buffer descriptors in the buffer descriptor ring

Size.MaxBuf - maximum size of a buffer in bytes

StartAddrDst - this should be set to the address of the FEC TX FIFO

IncrSrc - this should be set to 4 since the FEC FIFO deals with 4-bytes at a time

SzSrc - this should be set SZ_UINT32

SzDst - this should be set to SZ_UINT32

A.1.4 Ethernet RX

The Ethernet RX task will read data from the FEC and put it into memory using a buffer descriptor data structure that contains a pointer to the data buffer and a status word. This task also uses the buffer descriptor flags. An interrupt is generated by the BestComm when a complete Ethernet packet is transferred to memory. This task is meant to be started with interrupts enabled in TaskStart().

The Ethernet RX task never ends. It continually checks the status flags of a buffer descriptor for a clear field. When the FEC receives Ethernet data in the receive FIFO, the Bestcomm task will begin filling the memory buffer described by the buffer descriptor until the packet is finished. At this time, the task will generate an interrupt.

TaskSetupParamSet_t structure elements that must be initialized:

NumBD - number of buffer descriptors in the buffer descriptor ring

Size.MaxBuf - maximum size of a buffer

StartAddrSrc - this should be set to the address of the FEC RX FIFO

IncrDst - this should be set to 4 since the FEC only deals with 4 byte quantities

SzDst - this should be set to SZ_UINT32

SzSrc - this should be set to SZ_UINT32

A.1.5 CRC16 Dual-Pointer

This task is designed to add a 16-bit cyclic redundancy check (CRC) error correction at the end of the data transferred for transmit tasks. An interrupt is generated by the BestComm when a transfer is complete if the task is started with the interrupt enabled. A 16-bit CRC is appended to the received data on a receive task. It should be 0 if the CRC passed. The peripheral initiator must be set if a peripheral FIFO is used. Otherwise, the define INITIATOR_ALWAYS can be used.

TaskSetupParamSet_t structure elements that must be initialized:

Size.NumBytes - number of bytes to be transferred

Initiator - set according to peripheral used for the task

StartAddrSrc - Address of source buffer to be transferred

SzSrc - Transfer size in bytes (1, 2 or 4)

IncrSrc - Increment of source pointer. This should be 0 if the source is a FIFO, or the increment should match SzSrc.

SzDst - Transfer size in bytes (1, 2 or 4)

IncrDst - Increment of destination pointer. This should be 0 if the source is a FIFO, or the increment should match SzDst.

StartAddrDst - Address of destination buffer to be transferred

A.1.6 General Single-Pointer TX

This general single pointer task has a source pointer that can increment to transfer from memory to a peripheral transmit FIFO. The IncrDst member does not need to be set since it is 0. An interrupt is generated when the task is finished if it is enabled when TaskStart() is called.

TaskSetupParamSet_t structure elements that must be initialized:

Size.NumBytes - number of bytes to be transferred

Initiator - set according to peripheral used for the task

StartAddrSrc - Address of source buffer to be transferred

SzSrc - Transfer size in bytes (1, 2, or 4)

IncrSrc - Increment of source pointer. This should be 0 if the source is a FIFO, or the increment should match SzSrc.

SzDst - Transfer size in bytes (1, 2 or 4)

StartAddrDst - FIFO address to transfer to

A.1.7 General Single-Pointer RX

This general single pointer task has a destination pointer that can increment to receive data transferred by the BestComm from an RX FIFO. The IncrSrc member does not need to be set since it is 0. An interrupt is generated when the task is finished if it is enabled when TaskStart() is called.

TaskSetupParamSet_t structure elements that must be initialized:

Size.NumBytes - number of bytes to be transferred

Initiator - set according to peripheral used for the task

StartAddrSrc - FIFO address to transfer from

SzSrc - Transfer size in bytes (1, 2 or 4)

SzDst - Transfer size in bytes (1, 2, or 4)

IncrDst - Increment of destination pointer. This should be 0 if the source is a FIFO, or the increment should match SzDst.

StartAddrDst - Address of destination buffer to be transferred

A.1.8 General Dual-Pointer

This task takes two pointers and transfers data from one location to the other while incrementing both pointers until the specified number of bytes has been transferred. When the specified number of bytes has been transferred the BestComm will generate an interrupt if it is enabled in TaskStart(). This task can also be used with a peripheral FIFO by setting the increment on the source or destination to 0 and using the FIFO address as a source or destination.

TaskSetupParamSet_t structure elements that must be initialized:

Size.NumBytes - number of bytes to be transferred

Initiator - set according to peripheral used for the task

StartAddrSrc - Address of source buffer to be transferred. This can be an RX FIFO address.

SzSrc - Transfer size in bytes (1, 2 or 4)

IncrSrc - Increment of source pointer. This should be 0 if the source is a FIFO, or the increment should match SzSrc.

SzDst - Transfer size in bytes (1, 2 or 4)

IncrDst - Increment of destination pointer. This should be 0 if the source is a FIFO, or the increment should match SzDst.

StartAddrDst - Address of destination buffer to be transferred. This can be a TX FIFO address.

A.1.9 General Single-Pointer Buffer Descriptor TX

The single pointer buffer-descriptor task uses a buffer descriptor to transfer data from a memory buffer to a TX FIFO. An interrupt is generated by the BestComm when the task finishes transferring a buffer if it is enabled when TaskStart() is called. This task is also meant to be started with the auto start enabled in TaskStart(). The task will continue to work through each buffer in the buffer descriptor ring until it gets to the end when it will continue at the beginning of the ring.

Buffers are added to the buffer descriptor ring by calling TaskBDAssign(). A call to TaskBDRelease() removes a buffer from the ring. The first buffer pointer parameter passed to TaskBDAssign() is used as the source address. The second buffer pointer parameter is ignored.

TaskSetupParamSet_t structure elements that must be initialized:

Size.MaxBuf - maximum buffer size in bytes

Initiator - set according to peripheral used for the task

StartAddrSrc - This does not really matter since the source address is set in TaskBDAssign()

SzSrc - Transfer size in bytes (1, 2, or 4)

IncrSrc - Increment of source pointer. This should be 0 if the source is a FIFO, or the increment should match SzSrc.

SzDst - Transfer size in bytes (1, 2 or 4)

StartAddrDst - FIFO address to transfer to

A.1.10 General Single-Pointer Buffer Descriptor RX

The single pointer buffer-descriptor task uses a buffer descriptor to transfer data to a memory buffer from an RX FIFO. An interrupt is generated by the BestComm when the task finishes transferring a buffer if it is enabled when TaskStart() is called. This task is also meant to be started with the auto start enabled in TaskStart(). The task will continue to work through each buffer in the buffer descriptor ring until it gets to the end when it will continue at the beginning of the ring.

Buffers are added to the buffer descriptor ring by calling TaskBDAssign(). A call to TaskBDRelease() removes a buffer from the ring. The first buffer pointer passed to TaskBDAssign() is used as the destination address base pointer. The second buffer pointer parameter is ignored.

TaskSetupParamSet_t structure elements that must be initialized:

Size.MaxBuf - Maximum buffer size in bytes

Initiator - Set according to peripheral used for the task

StartAddrSrc - FIFO address to transfer from

SzSrc - Transfer size in bytes (1, 2 or 4)

SzDst - Transfer size in bytes (1, 2 or 4)

IncrDst - Increment of destination pointer. This should be 0 if the source is a FIFO, or the increment should match SzDst.

StartAddrDst - This does not matter since the destination address is set in TaskBDAssign()

A.1.11 General Dual-Pointer Buffer Descriptor Task

The dual pointer buffer-descriptor task uses a buffer descriptor ring to transfer data from one location to another. The difference between this task and the single pointer is that there are two pointers that can be incremented, which allows data to be transferred from memory to memory or to and from peripheral FIFOs. An interrupt is generated by the BestComm when the task finishes transferring a buffer if it is enabled when TaskStart() is called. This task is also meant to be started with the auto start enabled in TaskStart(). The task will continue to work through each buffer in the buffer descriptor ring until it gets to the end when it will continue at the beginning of the ring.

When the TaskBDAssign() function is called two pointers must be passed in. The first buffer pointer is the source, and the second buffer pointer is the destination.

TaskSetupParamSet_t structure elements that must be initialized:

Size.MaxBuf - maximum buffer size

Initiator - set according to peripheral used for the task

StartAddrSrc - This parameter can be ignored since it is set in TaskBDAssign().

SzSrc - Transfer size in bytes (1, 2 or 4)

IncrSrc - Increment of source pointer. This should be 0 if the source is a FIFO, or the increment should match SzSrc.

SzDst - Transfer size in bytes (1, 2 or 4)

IncrDst - Increment of destination pointer. This should be 0 if the source is a FIFO, or the increment should match SzDst.

StartAddrDst - This parameter can be ignored since it is set in TaskBDAssign().

A.1.12 General Dual-Pointer Buffer Descriptor + CRC16 Task

This task is the same as the regular dual pointer buffer-descriptor task that uses a buffer descriptor ring to transfer data from one location to another. Additionally, this task will append a 16-bit CRC on the end of the transferred buffer. An interrupt is generated by the BestComm when the task finishes transferring a buffer if it is enabled when TaskStart() is called. This task is also meant to be started with the auto start enabled in TaskStart(). The task will continue to work through each buffer in the buffer descriptor ring until it gets to the end when it will continue at the beginning of the ring.

When the TaskBDAssign() function is called two pointers must be passed. The first buffer pointer is the source, and the second buffer pointer is the destination.

TaskSetupParamSet_t structure elements that must be initialized:

Size.MaxBuf - maximum buffer size

Initiator - set according to peripheral used for the task

StartAddrSrc - This parameter can be ignored since it is set in TaskBDAssign().

SzSrc - Transfer size in bytes (1, 2 or 4)

IncrSrc - Increment of source pointer. This should be 0 if the source is a FIFO, or the increment should match SzSrc.

SzDst - Transfer size in bytes (1, 2 or 4)

IncrDst - Increment of destination pointer. This should be 0 if the source is a FIFO, or the increment should match SzDst.

StartAddrDst - This parameter can be ignored since it is set in TaskBDAssign().