AN4623
Application note

Memory monitoring using JDC module

Introduction

The document describes how to implement an application allowing an external tool to monitor the memory of the device by the JTAG pins. It uses the JTAG Data Communication (JDC) module to implement this monitor using only JATG pins.

The JDC module captures data into a memory mapped register that can be accessed via IPS for output via JTAG port.

The application is tested and validated using SPC574S60xx target and Lauterbach TRACE32 as external tool but it is valid for all targets with JDC modules listed below:

- SPC574K70xx
- SPC57EM80xx
- SPC574S60xx
- SPC570S50xx
- SPC572L64xx

Contents

1 JTAG Data Communication (JDC) 4

1.1 Introduction 4

1.2 Overview 4

1.3 JDC Module Registers 5

1.4 JDC Non memory mapped registers definition 5

1.5 Functional description 6

1.6 JTAG register access 6

2 Application: memory monitor using JDC module 8

2.1 Overview 8

2.2 Implementation 8

2.3 Application environment 9

2.4 Lauterbach TRACE32 9

2.5 Further developments 12

Appendix A 13

Revision history 15

List of tables

Table 1. JTAG instructions 7
Table 2. Lauterbach TRACE32 Commands 10
Table 3. Document revision history 15

1 JTAG Data Communication (JDC)

1.1 Introduction

The JTAG Data Communication (JDC) module provides the capability to move register data between the IPS and JTAG domains. This way facilitates communication between internal resources that access memory mapped register space and an external tool that accesses the JTAG port.

1.2 Overview

The JDC module consists of IPS accessible registers, JTAG accessible registers, and associated logic in order to coordinate movement of data from one register domain to another.

The JDC implements the following IPS data registers, occupying separate memory space:

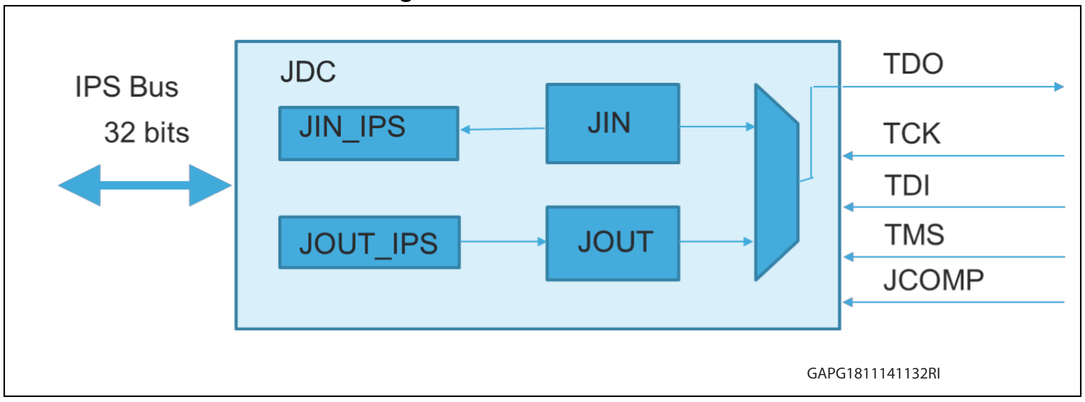
- 32-bit memory mapped register that can be read or written via IPS (JOUT_IPS), and whose contents are ported out for capture into a JTAG register (JOUT) to be read via the JTAG port.
- 32-bit memory mapped register that can only be read via IPS (JIN_IPS), and whose contents are loaded from a JTAG register (JIN).

In addition to the data registers themselves, logic is implemented to indicate when new data has been shifted in via the JTAG port and is ready to be read from the JIN_IPS register, and when new data has been written to the JOUT_IPS register and is ready to be read via the JTAG port. The state of these flags is captured into a status register (MSR), and also provided to a JTAG register (JOUT) for tool visibility.

The JTAG accessible registers are provided through an instance of the existing JTAGC (JTAG Controller) module.

There is a single bus interface to port register data out to the JTAGC, and a single bus interface to port data in from the JTAGC. The architecture is shown in *Figure 1*.

Figure 1. JDC Architecture



1.3 JDC Module Registers

The JDC module has four 32-bit registers.

- Module Configuration Register (MCR): The MCR is used to enable the interrupt outputs of the JDC. This register is reset by power-on reset, system destructive reset, and system functional reset. The MCR register fields are:
  - JIN_IEN: JIN Interrupt Enable. [1 asserts; 0 not assert]
  - JOUT_IEN: JOUT Interrupt Enable. [1 asserts; 0 not assert]
- Module Status Register (MSR): The MSR holds the JTAG register status and interrupt bits. This register is reset by power-on reset, system destructive reset, and system functional reset. The MSR register fields are:
- JIN_RDY: JIN Ready (read only)
  - 1: Set when new data is written to the JIN_IPS register
  - 0: Cleared upon software read of JIN_IPS contents via IPS
- JIN_INT: JIN Interrupt
  - 1: Set when new data is written to the JIN_IPS register
  - 0: Cleared by writing logic 1
- JOUT_RDY: JOUT Ready (read only)
  - 1: Set when new data is written to the JOUT_IPS register
  - 0: Cleared upon tool read of JOUT register via JTAG port
- JOUT_INT: JOUT Interrupt.
  - 1: Set when JOUT_RDY bit is cleared by tool reading JOUT register
  - 0: Cleared by writing logic 1
- JTAG Output IPS Data Register (JOUT_IPS): Contains JOUT_IPS data. The JOUT_IPS register holds data written via IPS. The JOUT_IPS contents are ported out and captured into the JOUT register to be read via the JTAG port. This register is reset by power-on reset, system destructive reset, and system functional reset.
- JTAG Input IPS Data Register (JIN_IPS): Contains JIN_IPS data. Data written to the JTAG input data register (JIN) via the JTAG port is held in the JIN_IPS register, where it can be read via IPS. This register is reset by power-on reset, system destructive reset, and system functional reset.

1.4 JDC no memory mapped registers definition

The JDC also implements two JTAG accessible registers that are not memory mapped. One JTAG register is used to shift in data to be placed in the JIN_IPS register. The other JTAG register captures data from the JOUT_IPS register and ready status from the MSR to be shifted out via the JTAG port.

- JTAG output data register (JOUT): The JOUT register captures data from the JOUT_IPS register upon execution of the JOUT_READ JTAG instruction. It also holds the JIN_RDY and JOUT_RDY status bits. This register is reset by system destructive reset and JTAG reset.

JOUT JTAG register field descriptions:

- – JOUT_IPS: Data value from JOUT_IPS register [32 bit]
- – JIN_RDY: State of JIN_RDY bit from MSR
- – JOUT_RDY: State of JOUT_RDY bit from MSR

- JTAG input data register (JIN): Data is written to the JIN register via JTAG during execution of the JIN_WRITE JTAG instruction. The JIN data is later captured in the JIN_IPS register to be read via IPS. This register is reset by system destructive reset and JTAG reset.

JIN JTAG register field descriptions:

- JIN Data: Contains data to be captured in JIN_IPS register upon exit of Update-DR state when executing WRITE_JIN JTAG instruction.

1.5 Functional description

The JDC module provides the ability to shift in data via the JTAG port and capture that data into a memory mapped register that can be accessed via IPS. It also provides the ability to capture data written to a memory mapped register into a JTAG shift register for output via the JTAG port. An overview of the module functionality is described below.

A software writes to the JOUT_IPS register sets the JOUT_RDY flag bit, that indicates new data is available to be read from the JOUT register via the JTAG port. The state of the JOUT_RDY bit is reflected in the MSR and also ported out to the JOUT register. A JTAG read of the JOUT register via execution of the JOUT_READ instruction with a JOUT_RDY bit whose value is logic 1 indicates the register contains new data. The JOUT_RDY flag bit is cleared upon exit of the Capture-DR JTAG state during execution of the JOUT_READ instruction. Clearing the JOUT_RDY bit indicates to software that a new data value can be written to the JOUT_IPS register.

A JTAG writes to the JIN register via execution of the JIN_READ JTAG instruction updates the contents of the JIN_IPS register upon exit of the Update-DR state. An update of the JIN_IPS register sets the JIN_RDY flag bit, that indicates new data is available to be read via IPS. The state of the JIN_RDY bit is reflected in the MSR register and also ported out to the JOUT register. The JIN_RDY flag bit is cleared upon software read of the JIN_IPS register. A JTAG reads of the JOUT register with a JIN_RDY value of logic 0 indicates that new data can be written to the JIN register.

1.6 JTAG register access

The JDC block implements the IEEE 1149.1-2001 defined instructions listed in *Table 1*. This section gives an overview of each instruction; refer to the IEEE 1149.1-2001 standard for more details. All undefined opcodes are reserved.

Table 1. JTAG instructions

Instructions	Code[4:0] instructions	Instruction summary
Reserved	00001	Factory debug reserved
JOUT_READ	00010	Selects JOUT data register. Data from JOUT_IPS is captured into JOUT data register upon entry to Capture-DR state while JOUT_READ is active.
JIN_WRITE	01110	Selects JIN data register. Data from JIN is captured into JIN_IPS upon exit of Update-DR state while JIN_WRITE is active.
BYPASS	11111	Selects bypass register for data operations
Reserved	All other opcodes	Decoded to select bypass register

2 Application: memory monitor using JDC module

2.1 Overview

This application note describes how to use the JDC module to implement a memory monitor. To do this, the application implemented uses JOUT_IPS register to get out the data and the JOUT Flag of MSR register to synchronize the external tool.

2.2 Implementation

The memory monitored is a 10 elements buffer of 8 bits. The application transfers the content of each element of the buffer into the JOUT_IPS register and waits for the tool read the data. When the tool read the data, the JOUT_INT (JOUT interrupt) occurs. At this point the application can put a new data into JOUT_IPS register.

Following the buffer used by application:

- uint8_t *buffer[MAX_BUFFER];//Buffer containing the RAM variable address to monitor
- uint32_t nbuffer = 0;//buffer index used to indicate the next element to send

The first step consists of enabling of the JOUT interrupt:

- JDC.MCR.B.JOUT_IEN = 0x1; //Enable JOUT Interrupt
- INT_IRQ_INIT(JDC_JOUT_INT, JDC_JOUT_INT_PRIORITY); //Set priority for JDC JOUT interrupt

The second step consists of setting first time the JOT_IPS register to activate the process monitor:

- JDC.JOUT_IPS.R = *(buffer[nbuffer]);

The code below is the ISR (Interrupt Service Routine) executed each time the JOUT_INT occurs:

```
void JDC_JOUT_ISR(void)
{
    JDC.MSR.R = 0x1;//Clear JOUT interrupt flag;
    nbuffer++;//Increase buffer index counter
    if (nbuffer == MAX_BUFFER)//Check if buffer index counter reach the
MAX_BUFFER value
    {
     nbuffer = 0;//reset buffer index counter
    }
    JDC.JOUT_IPS.R = *(buffer[nbuffer]);//update JOUT register value
}
```

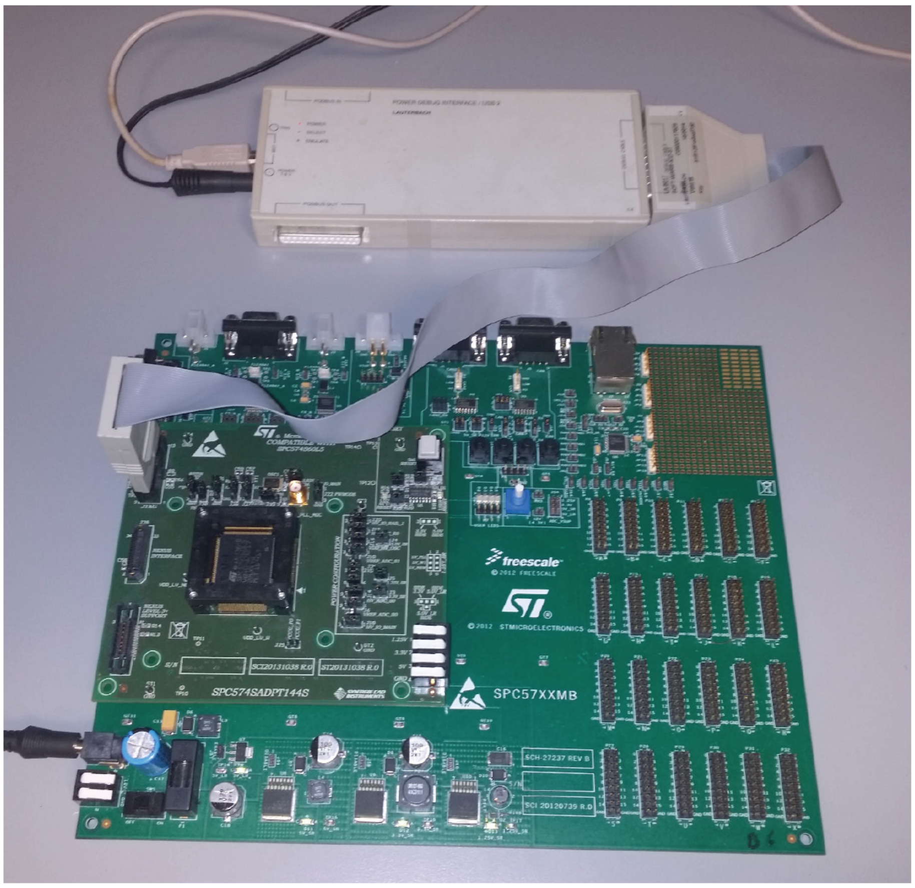
2.3 Application environment

The environment used to test the application consists of:

- Target: SPC574S60L5
- Mini-module: SPC574SADPT144S
- Motherboard: SPC57XXMB
- External tool: Lauterbach TRACET32

Figure 2 shows the environment used.

Figure 2. Application schema



2.4 Lauterbach TRACE32

Lauterbach TRACE32 is the external tool used to implement the application. TRACE32 allows showing the data received by JDC using a terminal windows.

Table 1 reports the commands of TRACE32 allowing to open and configure a terminal window to show the data.

Table 2. Lauterbach TRACE32 Commands

Command	Description
TERM.RESet	It close all terminal windows opened.
TERM.METHOD <mode>	It defines how data is exchanged between the target application and the debugger. On some targets additional processor specific modes may be available. The mode option used in this application is DCC. DCC option defines to use the DCC port of the JTAG interface.
TERM.Mode	It defines the terminal type used for new terminal windows. The possible options are: ASCII, STRING, RAW, HEX and VT100. VT100 implements a subset of the VT100 control codes. To show the data in hexadecimal format the application uses HEX option.
TERM.view	It creates a terminal emulation window. The window can be used to communicate with the software running in the target.

The step needs to execute the application is described following:

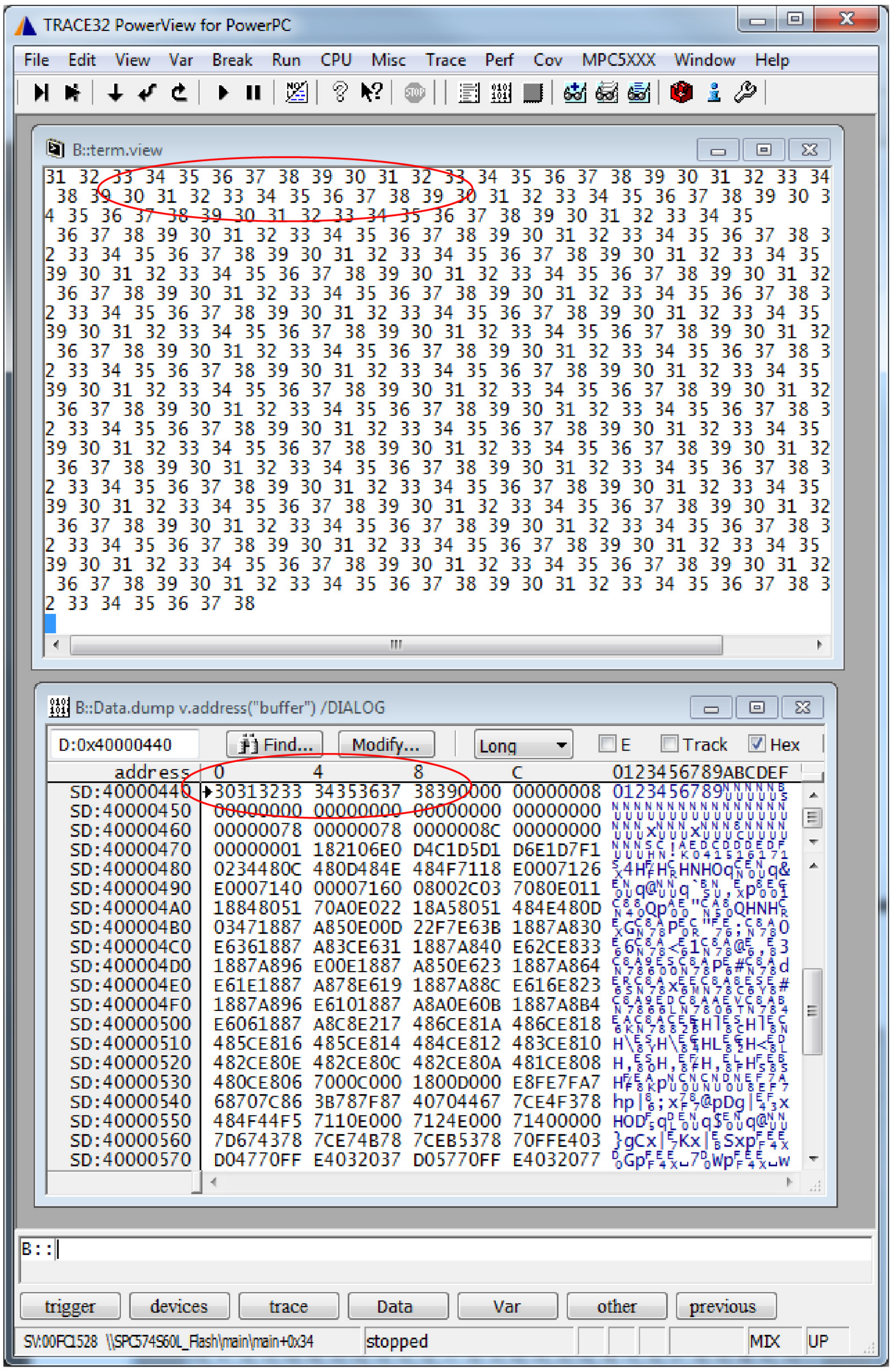
- Start Laurterbach TRACET32
- Load the application image in flash of the target using the Lauterbach script SPC574s.cmm available on installation folder of TRACET32 (.\T32\demo\powerpc\flash)
- Execute the TRACE32 commands to open and configure the terminal windows in the following sequence:

1. TERM.RESet
2. TERM.METHOD DCC
3. TERM.Mode HEX
4. TERM.view

- Run the application.

Figure 3 shows the windows terminal opened showing the data sent by the target using JDC module and a dump of the part of memory containing the buffer transferred by JDC module. As it is possible to observe, the data transferred continuously (a group is circled in red) in the terminal windows correspond to the data read dumping the memory (circled in red in dump window).

Figure 3. TRACE32 Terminal window


TRACE32 PowerView for PowerPC
File Edit View Var Break Run CPU Misc Trace Perf Cov MPC5XXX Window Help
B::term.view

```
31 32 33 34 35 36 37 38 39 30 31 32 33 34 35 36 37 38 39 30 31 32 33 34
 38 39 30 31 32 33 34 35 36 37 38 39 30 31 32 33 34 35 36 37 38 39 30 3
4 35 36 37 38 39 30 31 32 33 34 35 36 37 38 39 30 31 32 33 34 35
 36 37 38 39 30 31 32 33 34 35 36 37 38 39 30 31 32 33 34 35 36 37 38 3
2 33 34 35 36 37 38 39 30 31 32 33 34 35 36 37 38 39 30 31 32 33 34 35
39 30 31 32 33 34 35 36 37 38 39 30 31 32 33 34 35 36 37 38 39 30 31 32
 36 37 38 39 30 31 32 33 34 35 36 37 38 39 30 31 32 33 34 35 36 37 38 3
2 33 34 35 36 37 38 39 30 31 32 33 34 35 36 37 38 39 30 31 32 33 34 35
39 30 31 32 33 34 35 36 37 38 39 30 31 32 33 34 35 36 37 38 39 30 31 32
 36 37 38 39 30 31 32 33 34 35 36 37 38 39 30 31 32 33 34 35 36 37 38 3
2 33 34 35 36 37 38 39 30 31 32 33 34 35 36 37 38 39 30 31 32 33 34 35
39 30 31 32 33 34 35 36 37 38 39 30 31 32 33 34 35 36 37 38 39 30 31 32
 36 37 38 39 30 31 32 33 34 35 36 37 38 39 30 31 32 33 34 35 36 37 38 3
2 33 34 35 36 37 38 39 30 31 32 33 34 35 36 37 38 39 30 31 32 33 34 35
39 30 31 32 33 34 35 36 37 38 39 30 31 32 33 34 35 36 37 38 39 30 31 32
 36 37 38 39 30 31 32 33 34 35 36 37 38 39 30 31 32 33 34 35 36 37 38 3
2 33 34 35 36 37 38 39 30 31 32 33 34 35 36 37 38 39 30 31 32 33 34 35
39 30 31 32 33 34 35 36 37 38 39 30 31 32 33 34 35 36 37 38 39 30 31 32
 36 37 38 39 30 31 32 33 34 35 36 37 38 39 30 31 32 33 34 35 36 37 38 3
2 33 34 35 36 37 38 39 30 31 32 33 34 35 36 37 38 39 30 31 32 33 34 35
39 30 31 32 33 34 35 36 37 38 39 30 31 32 33 34 35 36 37 38 39 30 31 32
 36 37 38 39 30 31 32 33 34 35 36 37 38 39 30 31 32 33 34 35 36 37 38 3
2 33 34 35 36 37 38
```

```
     address  0        4        8        C        0123456789ABCDEF
 SD:40000440 ▸30313233 34353637 38390000 00000008 0123456789
 SD:40000450  00000000 00000000 00000000 00000000
 SD:40000460  00000078 00000078 0000008C 00000000
 SD:40000470  00000001 182106E0 D4C1D5D1 D6E1D7F1
 SD:40000480  0234480C 480D484E 484F7118 E0007126
 SD:40000490  E0007140 00007160 08002C03 7080E011
 SD:400004A0  18848051 70A0E022 18A58051 484E480D
 SD:400004B0  03471887 A850E00D 22F7E63B 1887A830
 SD:400004C0  E6361887 A83CE631 1887A840 E62CE833
 SD:400004D0  1887A896 E00E1887 A850E623 1887A864
 SD:400004E0  E61E1887 A878E619 1887A88C E616E823
 SD:400004F0  1887A896 E6101887 A8A0E60B 1887A8B4
 SD:40000500  E6061887 A8C8E217 486CE81A 486CE818
 SD:40000510  485CE816 485CE814 484CE812 483CE810
 SD:40000520  482CE80E 482CE80C 482CE80A 481CE808
 SD:40000530  480CE806 7000C000 1800D000 E8FE7FA7
 SD:40000540  68707C86 3B787F87 40704467 7CE4F378
 SD:40000550  484F44F5 7110E000 7124E000 71400000
 SD:40000560  7D674378 7CE74B78 7CEB5378 70FFE403
 SD:40000570  D04770FF E4032037 D05770FF E4032077
```

```
B::
```

```
SV:00FC1528 \\SPC574S60L_Flash\main\main+0x34   stopped   MIX UP
```

2.5 Further developments

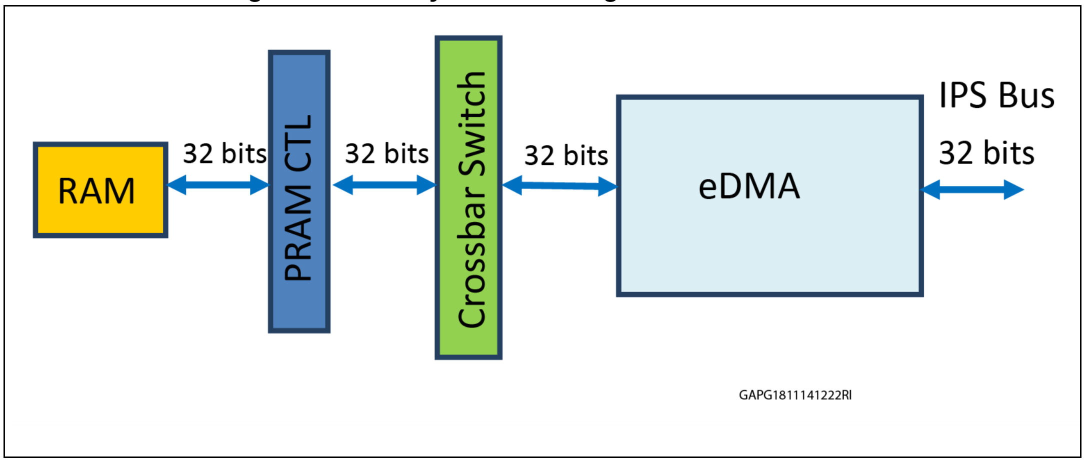
Another possible solution is to use the eDMA module to access the memory with the minimal intervention from the core.

The eDMA module is a highly programmable data transfer engine and it is used to transfer data from crossbar switch to IPS bus.

The crossbar switch connects bus musters and bus slaves using a hardware interconnect matrix. DMA is Xbar master and RAM controller is Xbar slave.

The PRAM controller (PRAM CTL) acts as an interface between the system bus and the dedicated RAM array interface.

Figure 4. Memory access using eDMA architecture



Appendix A Source code example

```
Main file code example:
/*****************************************************************
* PROJECT : Sphaero JDC Example
* FILE : main.c
*
* DESCRIPTION :
*
* COPYRIGHT :(c) 2014, Freescale & STMicroelectronics
*
* VERSION : 1.0
* DATE :   11/12/2014
* AUTHOR : Alessandro Polizzi
* HISTORY :
*****************************************************************/

#include "..\platform\typedefs.h"
#include "SPC574S60L.h"
#include "spc57_init.h"

/******* Macros and Prototypes **********/
void JDC_Init(void);
#define INT_IRQ_INIT(irq_number, priority)INTC.PSR[irq_number].R = 0x8000 |
priority;
#define JDC_JOUT_INT 675
#define JDC_JOUT_INT_PRIORITY 10
#define MAX_BUFFER 10

//******* Variables ********************/
//Buffer containing the RAM variable address to monitor
uint32_t buffer[MAX_BUFFER] = {0x30, 0x31, 0x32, 0x33, 0x34, 0x35, 0x36,
0x37, 0x38, 0x39};
uint32_t nbuffer = 0;//buffer index

//Main function
void main(void)
{
    JDC_Init();//Init JDC

    INT_IRQ_INIT(JDC_JOUT_INT, JDC_JOUT_INT_PRIORITY );  //Set priority for
JDC JOUT interrupt

    JDC.JOUT_IPS.R = buffer[nbuffer];  //Set first time JOUT register to
activate monitor process
```

```
    while(1){}  //Main loop
}


/*************************************************************************/
/*  Function: void JDC_Init(void)*/
/*                      */
/*************************************************************************/
void JDC_Init(void)
{
    JDC.MCR.B.JOUT_IEN = 0x1;//Enable JOUT Interrupt;
}

/*************************************************************************/
/*  Function: void JDC_JOUT_ISR(void)*/
/*                      */
/*************************************************************************/
void JDC_JOUT_ISR(void)
{
    JDC.MSR.R = 0x1;//Clear JOUT interrupt flag;
    nbuffer++;//Increase buffer index counter
    if (nbuffer == MAX_BUFFER)//Check if buffer index counter reach the
MAX_BUFFER value
    {
   nbuffer = 0;//reset buffer index counter
    }
    JDC.JOUT_IPS.R = buffer[nbuffer];//update JOUT register value
}
```

Revision history

Table 3. Document revision history

Date	Revision	Changes
18-Nov-2014	1	Initial release.