ECE PARIS
ÉCOLE D'INGÉNIEURS

DSP

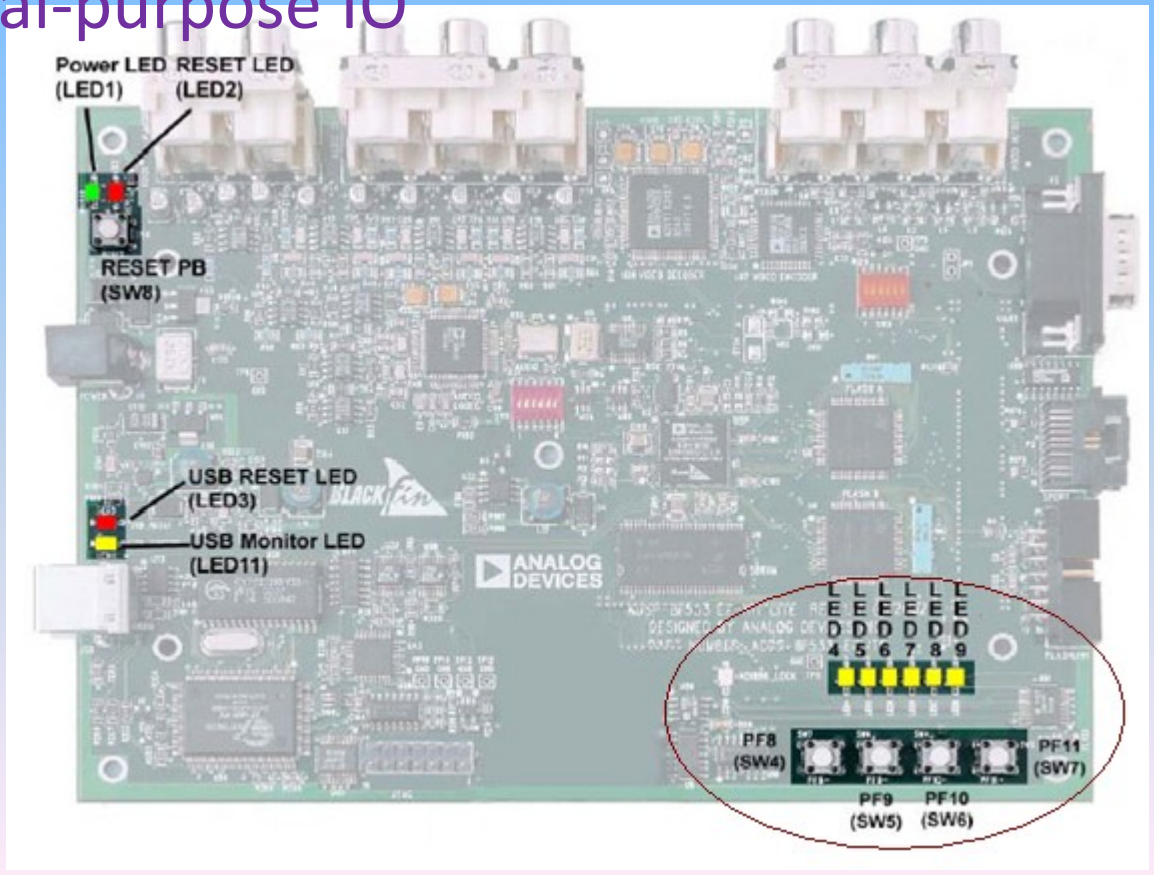
Lab. 4 – Blink a LED explained or when blinking is not as easy as you think

Stefan Ataman

2013 – 2014

The EZ-KIT Lite and the outside World

- Your EZ-KIT Lite provides
  - 4 push buttons (PF8, PF9, PF10, PF11)
  - 6 LEDs for general-purpose IO



BF533 EZ-Kit Lite

- 4 SW push-button switches
  - The four general-purpose push button switches are labeled SW4 through SW7
  - A status of each individual button can be read through programmable flag (PF) inputs: PF8-PF11

BF533 EZ-Kit Lite

- 4 SW push-button switches
  - A PF reads 1 when a corresponding switch is being pressed (pushed). When the switch is released, the PF reads 0.
  - A connection between the push button and PF input is established through the SW9 DIP switch.

BF533 EZ-Kit Lite

- 6 LEDs
  - The six LEDs, labeled LED4 through LED9, are accessed via some of the general-purpose IO pins of the flash memory interface.

LED Reference Designator	Flash Port Name
LED4	PB0
LED5	PB1
LED6	PB2
LED7	PB3
LED8	PB4
LED9	PB5

BF533 Interface to PFI/PFs

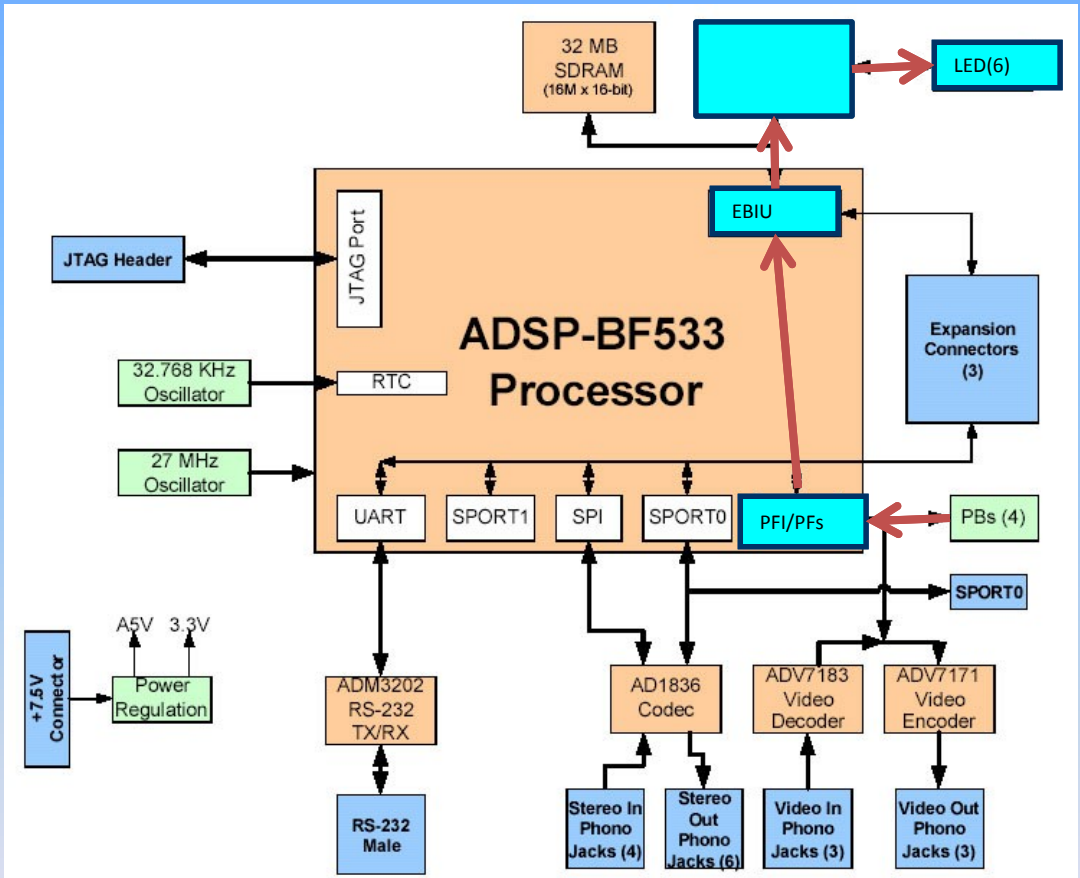
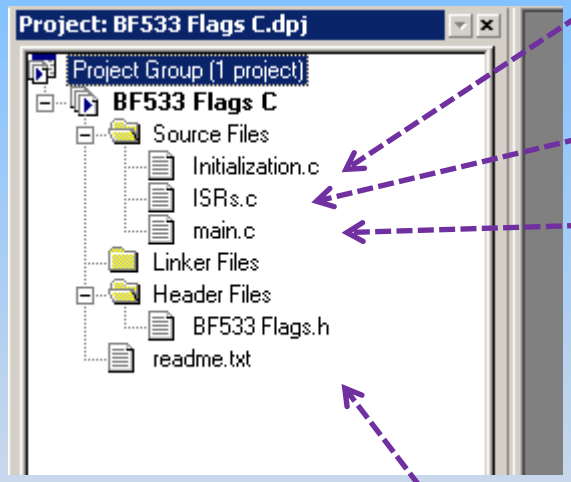


Figure 3-1. System Architecture

The Led Blink C project

Open the LED Blink project!

The Led Blink C project



initializations for the processor, codec and stuff

the interrupt service routines

the main file

the header file with parameters and function prototypes

The Led Blink C project

- This example demonstrates the initialization of the
  - External Bus Interface Unit - EBIU (asynchronous access)
  - Timer 0
  - FIO pins
  - Interrupts and the on-board Flash, which allows access to the six LEDs on the EZ-KIT.

The Led Blink C project

- The program simply turns on one LED and rotates the pattern left or right, depending on the state of an internal flag.

- The switch connected to PF8 (SW4) can be used to toggle the state of this flag, which results in a change of direction of the moving light.

Initialization.c – the Init_Flags

```
#include "BF533 Flags.h"
//-------------------------------------------------------------------------//
// Function:        Init_Flags                                             //
//                                                                         //
// Parameters:      None                                                   //
//                                                                         //
// Return:                    None                                        //
//                                                                         //
// Description:  This function configures PF8 as input for edge sensitive  //
//        interrupt generation.                                            //
//        The switch connected to PF8 (SW4) can be used to change the      //
//        direction of the moving light.                                   //
//-------------------------------------------------------------------------//
void Init_Flags(void)
{
    *pFIO_INEN                 = 0x0100;

    *pFIO_DIR                  = 0x0000;

    *pFIO_EDGE                 = 0x0100;

    *pFIO_MASKA_D              = 0x0100;
}
```

this value programs the PF8 pin

this reset value sets all pins as inputs

this value sets PF8 pin to be EDGE sensitive

this value enables PF8 pin to accept interrupts

Actually, in Initialize.c you find:

Initialization.c

```
#include "BF533 Flags.h"

//-------------------------------------------------------------------------//
// Function:    Init_Flags                                                 //
//                                                                         //
// Parameters:  None                                                       //
//                                                                         //
// Return:      None                                                       //
//                                                                         //
// Description: This function configures PF8 as input for edge sensitive  //
//              interrupt generation.                                      //
//              The switch connected to PF8 (SW7) can be used to change the //
//              direction of the moving light.                             //
//-------------------------------------------------------------------------//
void Init_Flags(void)
{
    FIO_ANOM_0311_INEN_W(0x0100);//*pFIO_INEN       = 0x0100, workaround for anomaly
    FIO_ANOM_0311_DIR_W(0x0000);//For more information please refer to the comment he
    FIO_ANOM_0311_EDGE_W(0x0100);
    FIO_ANOM_0311_MASKA_W(0x0100,pFIO_MASKA_D);
}

//-------------------------------------------------------------------------//
// Function:    Init_Timers                                                //
//                                                                         //
// Parameters:  None                                                       //
//                                                                         //
// Return:      None                                                       //
//                                                                         //
// Description: This function initialises Timer0 for PWM mode.            //
//              It is used as reference for the 'shift-clock'.             //
//-------------------------------------------------------------------------//
void Init_Timers(void)
{
    *pTIMER0_CONFIG     = 0x0019;
```

these are anomalies i.e. fixes for hardware/software bugs.

Message from ADI:
shut up and use them!

Confused? Want more?

Detailed explanation follows.

You will regret it.

But since you asked for it, here it comes!

Programmable Flags PFI/PFs

- The processor supports 16 bidirectional programmable flags (PFx) or general-purpose I/O pins, PF[15:0].
- Each pin can be individually configured as either an input or an output by using the Flag Direction register (FIO_DIR).
  - When configured as output:
    - the Flag Data register (FIO_FLAG_D) can be directly written to specify the state of all PFx pins.
    - the state written to the
      - Flag Set (FIO_FLAG_S),
      - Flag Clear (FIO_FLAG_C), and
      - Flag Toggle (FIO_FLAG_T) registers determine the state driven by the output PFx pin.

Programmable Flags PFI/PFs

  - Regardless of how the pins are configured, as an input or an output, reading any of these registers:
    - FIO_FLAG_D,
    - FIO_FLAG_S,
    - FIO_FLAG_C, or
    - FIO_FLAG_T returns the state of each pin.
- They can be enabled via bits in Flag Input Enable register (FIO_INEN)
- Input buffer associated with the PF flags is disabled by default. The Flag Input Enable register (FIO_INEN) is used to enable them.

Flag Input Enable Register

PF8 pin enable:

```
// --- this value programs PF8 pin
        *pFIO_INEN =  0x0100;
// --- i.e.  0000 0001 0000 0000
```

FIO_INEN Register

The Flag Input Enable register (FIO_EDGE) is used to enable the input buffers on any flag pin that is being used as an input. Leaving the input buffer disabled eliminates the need for pull-ups and pull-downs when a particular PFx pin is not used in the system. By default, the input buffers are disabled.

Note that if the PFx pin is being used as an input, the corresponding bit in the Flag Input Enable register must be set. Otherwise, changes at the flag pins will not be recognized by the processor.

Flag Input Enable Register (FIO_INEN)

For all bits, 0 - Input Buffer Disabled, 1 - Input Buffer Enabled

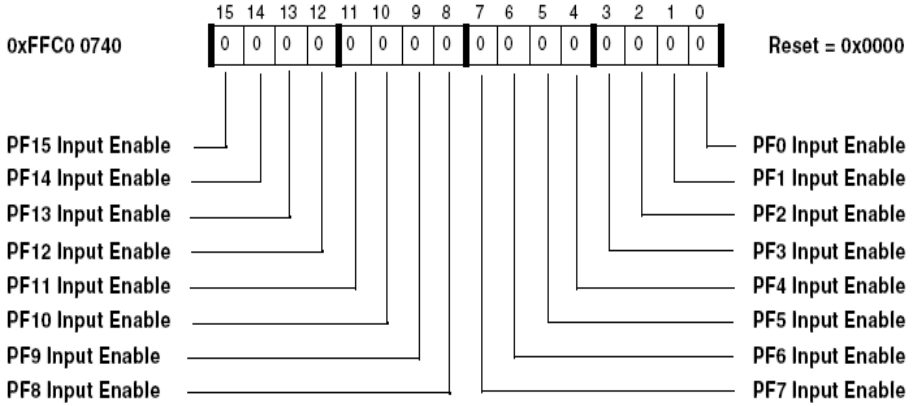
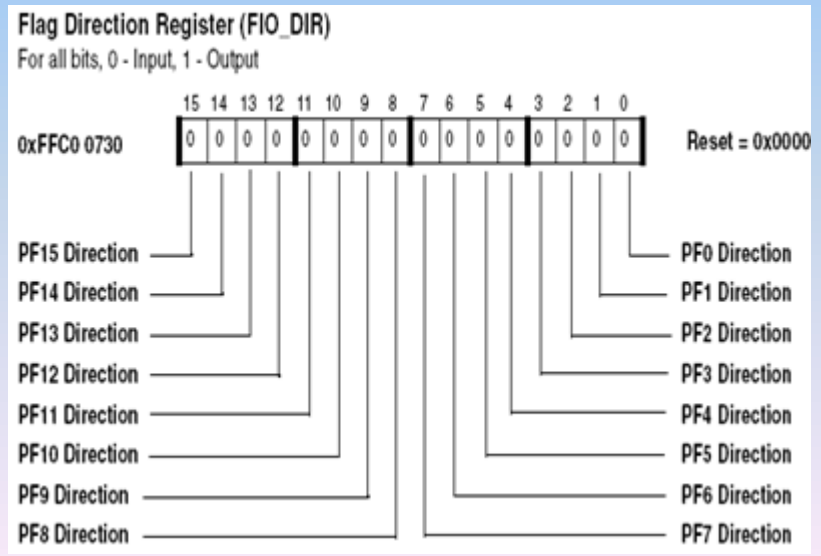


Figure 14-18. Flag Input Enable Register

FIO_DIR Register

- Each pin, PF[15:0], can be individually configured as either an input or an output by using the Flag Direction register (FIO_DIR)
- The FIO_DIR is a read-write (R/W) register
- Each bit position corresponds to a PFx pin.
  - A logic 1 configures a PFx pin as an output, driving the state contained in the FIO_FLAG_D register.
  - A logic 0 configures a PFx pin as an input. The reset value of this register is 0x0000, making all PF pins inputs upon reset.

```
// --- this reset value makes all pins as inputs
            *pFIO_DIR = 0x0000;
// --- ie.  0000 0000 0000 0000
```



Programming PFx to generate an interrupt

- Each PFx pin can be configured to generate an interrupt.

- When a PFx pin is configured as an input, an interrupt can be generated according to the state of the pin, either
  - high or low
  - an edge transition (low to high or high to low), or
  - on both edge transitions (low to high *and* high to low)

Programming PFx to generate an interrupt

- Input sensitivity is defined on a per-bit basis by
  - the Flag Polarity register (FIO_POLAR),
  - the Flag Interrupt Sensitivity register (FIO_EDGE) and
  - The Flag Set on Both Edges register (FIO_BOTH).

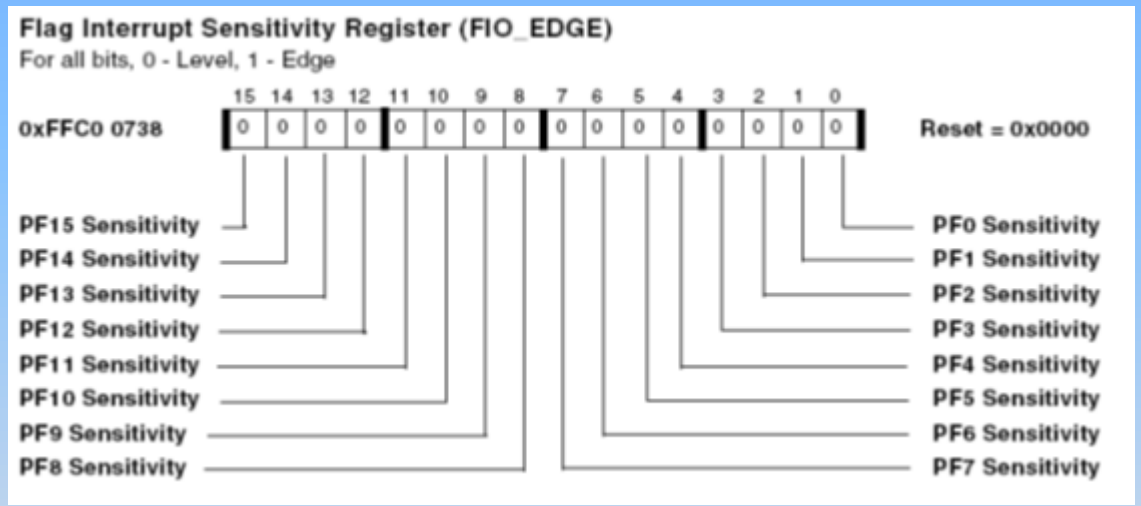
Programming PFx to generate an interrupt

- Input polarity is defined on a per-bit basis by
  - the Flag Polarity register.
- When
  - the PFx inputs are enabled and
  - a PFx pin is configured as an output,

  enabling interrupts for the pin allows an interrupt to be generated by setting the PFx pin.

FIO_EDGE: Flag Interrupt Sensitivity Register

For all bits:
0 – Level Sensitive
1 – EDGE Sensitive




```
// --- This value sets PF8 pin to be EDGE sensitive
            *pFIO_EDGE= 0x0100;
//  --- ie.  0000 0001 0000 0000
```

FIO_MASKx_D, FIO_MASKx_C, FIO_MASKx_S, FIO_MASKx_T,

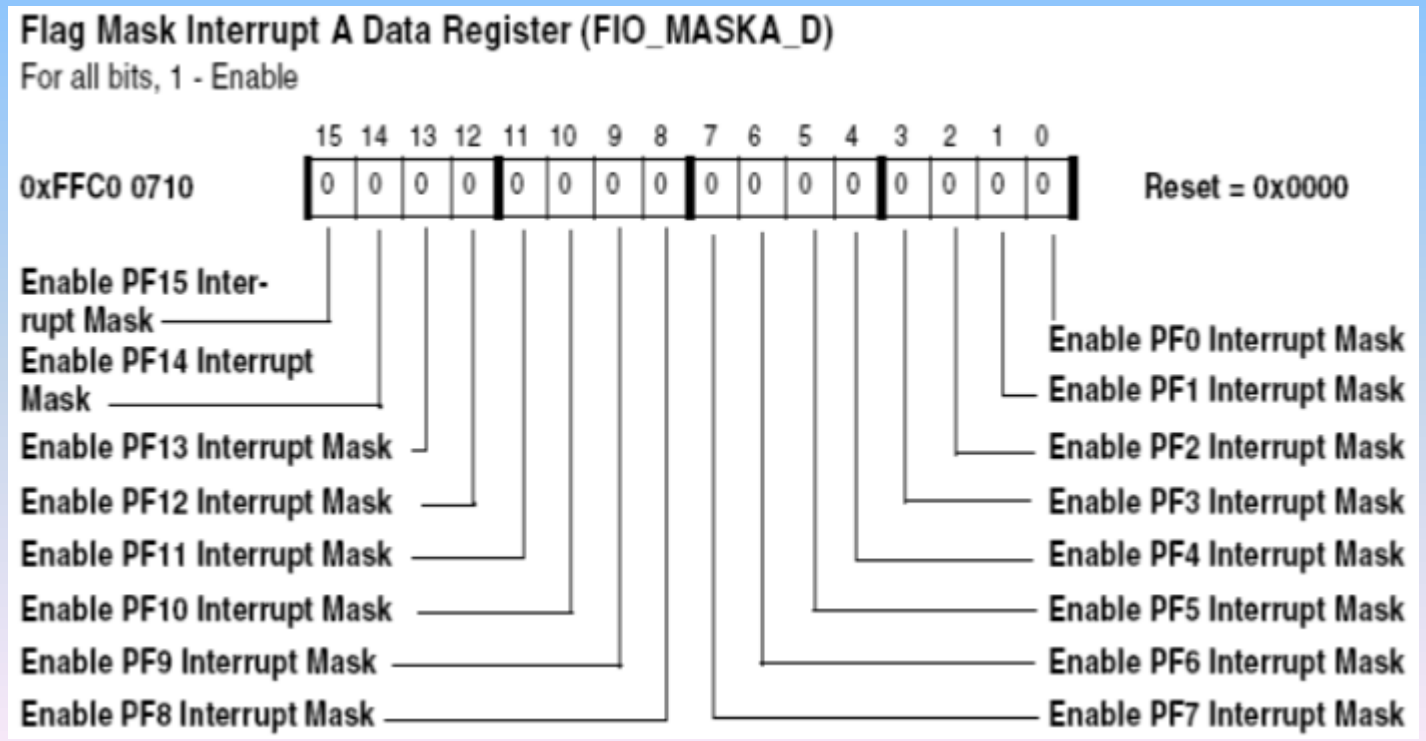
- Both Flag Interrupt A and Flag Interrupt B are supported by a set of four dedicated registers:
  - Flag Mask Interrupt Data register
  - Flag Mask Interrupt Set register
  - Flag Mask Interrupt Clear register
  - Flag Interrupt Toggle register

FIO_MASKx_D, FIO_MASKx_C, FIO_MASKx_S, FIO_MASKx_T,

- The Flag Mask Interrupt registers:
  - FIO_MASKA_D, FIO_MASKA_S, FIO_MASKA_C, FIO_MASKA_T, and
  - FIO_MASKB_D, FIO_MASKB_S, FIO_MASKB_C, FIO_MASKB_T
    are implemented as complementary pairs of:
  - Data “D”, write-1-to-set “S”, write-1-to-clear “C”, and write-1-to-toggle “T” registers.
- This implementation provides the ability to:
  - enable or disable a PFx pin to act as a processor interrupt without requiring read-modify-write accesses—or to
  - directly specify the mask value with the data register.

FIO_MASKA_D Register to Enable Interrupt

```
// --- This value enables PF8 pin to accept interrupts
        *pFIO_MASKA_D = 0x0100
// --- i.e. 0000 0001 0000 0000
```

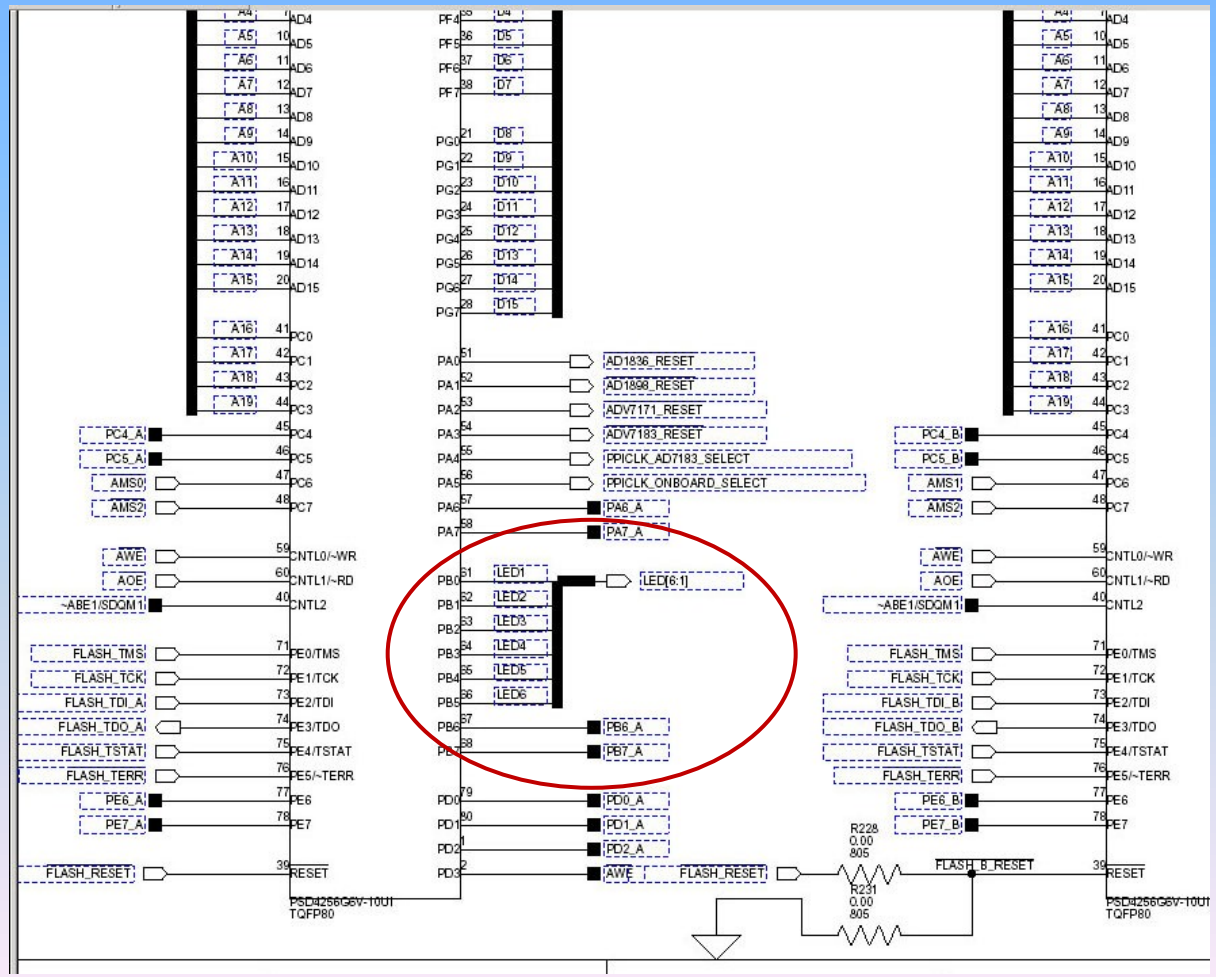


The LED Interface

or where the HELL are my LEDs connected?

Answer: to the Flash memory

How to “configure” the flash memory to control the LEDs



Parallel interfaces present on the FLASH memory chips

How to “configure” the flash memory to control the LEDs

- General-purpose IO signals are controlled by means of setting appropriate registers of the Flash A or Flash B.

- These registers are mapped into the processor’s address space

Flash General-Purpose IO

- The processor's address space is shown in the Table below:

(for more details, see ADSP-BF533 EZ-KIT Lite Evaluation System Manual – Flash Memory)

Table 1-4. Flash Memory Map

Start Address	End Address	Content
0x2000 0000	0x200F FFFF	Flash A primary (1MB)
0x2010 0000	0x201F FFFF	Flash B primary (1MB)
0x2020 0000	0x2020 FFFF	Flash A secondary (64KB)
0x2024 0000	0x2024 7FFF	Flash A SRAM (32KB)
0x2027 0000	0x2027 00FF	
0x2028 0000	0x2028 FFFF	Flash B secondary (64KB)
0x202C 0000	0x202C 7FFF	Flash B SRAM (32KB)
0x202E 0000	0x202E 00FF	
All other locations		Reserved

Flash General-Purpose IO

- Flash device IO pins are arranged as 8-bit ports labeled A through G.
- There is a set of 8-bit registers associated with each port. These registers are used for:
  - Direction,
  - Data In, and
  - Data Out.

Note: On power-up reset, the Direction and Data Out registers are cleared to all zeros.

Flash General-Purpose IO

- The ***Direction*** register controls IO pins direction. This is a 8-bit read-write register.
  - When a bit is 0, a corresponding pin functions as an input.
  - When the bit is 1, a corresponding pin is an output.

- The ***Data In*** register allows reading the status of port's pins. This is a 8-bit read-only register

- The ***Data Out*** register allows clearing an output pin to 0 or setting it to 1. This is a 8-bit read-write register

Flash General-Purpose IO

- The ADSP-BF533 EZ-KIT Lite board employs only flash A and flash B ports A and B.
- the next tables provide configuration register addresses for Flash A and Flash B, respectively

- The following bits connect to the expansion board connector.
  - Flash A: port A bits 7 and 6, as well as port B bits 7 and 6
  - Flash B: port A bits 7–0

Flash A & B Configuration Registers for Ports A and B.

Flash A Configuration Registers for Ports A and B

Register Name	Port A Address	Port B Address
Data In (read-only)	0x2027 0000	0x2027 0001
Data Out (read-write)	0x2027 0004	0x2027 0005
Direction (read-write)	0x2027 0006	0x2027 0007

Flash B Configuration Registers for Ports A and B

Register Name	Port A Address	Port B Address
Data In (read-only)	0x202E 0000	0x202E 0001
Data Out (read-write)	0x202E 0004	0x202E 0005
Direction (read-write)	0x202E 0006	0x202E 0007

IO Assignments for Port A and Port B

Flash A Port A Controls

Bit Number	User IO	Bit Value
7	Not defined	Any
6	Not defined	Any
5	PPI clock select bit 1	00=local OSC (27MHz)
4	PPI clock select bit 0	01=video decoder pixel clock 1x=expansion board PPI clock
3	Video decoder reset	0=reset ON; 1=reset OFF
2	Video encoder reset	0=reset ON; 1=reset OFF
1	Reserved	Any
0	Codec reset	0=reset ON; 1=reset OFF

PPI – Parallel Peripheral Interface & Associated Configuration Registers

IO Assignments for Port A and Port B

Flash A Port B Controls.

Bit Number	User IO	Bit Value
7	Not used	Any
6	Not used	Any
5	LED9	0=LED OFF; 1=LED ON
4	LED8	0=LED OFF; 1=LED ON
3	LED7	0=LED OFF; 1=LED ON
2	LED6	0=LED OFF; 1=LED ON
1	LED5	0=LED OFF; 1=LED ON
0	LED4	0=LED OFF; 1=LED ON

User LEDs (LED4-9)

- Six LEDs connect to six general-purpose IO pins of the flash memory (U5).
- The LEDs are active high and are lit by writing a 1 to the correct memory address in the flash memory.

Table 2-8. User LEDs

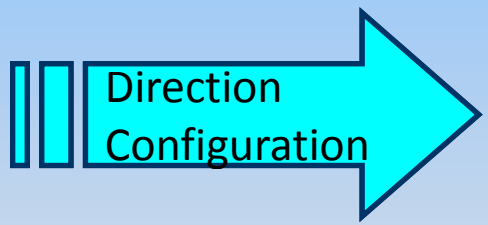
LED Reference Designator	Flash Port Name
LED4	PB0
LED5	PB1
LED6	PB2
LED7	PB3
LED8	PB4
LED9	PB5

Flash setup

- The ***Direction*** register controls IO pins direction. This is a 8-bit read-write register.
- When a bit is 0, a corresponding pin functions as an input.
- When the bit is 1, a corresponding pin is an output.

Initialization.c

```
//-------------------------------------------------------------------------//
// Function:        Init_Flash                                             //
//                                                                         //
// Parameters:      None                                                   //
//                                                                         //
// Return:                    None                                        //
//                                                                         //
// Description:     This function sets the pin direction of Port B in Flash A  //
//        to output.                                                       //
//        The LEDs on the ADSP-BF533 EZ-KIT are connected to Port B.       //
//-------------------------------------------------------------------------//
void Init_Flash(void)
{
    *pFlashA_PortB_Dir = 0x3f;
}
```



0x3f = 0011 1111

- The ***Direction*** register controls IO pins direction. This is a 8-bit read-write register.
- When a bit is 0, a corresponding pin functions as an input.
- When the bit is 1, a corresponding pin is an output.

BF533 Interrupts

or when you start regretting that you are not on a PIC16F

Interrupts versus Exceptions

- An interrupt is an event that changes the normal processor instruction flow and is asynchronous to program flow.
- An exception is a software initiated event whose effects are synchronous to program flow.
- The event system is nested and prioritized. Consequently, several service routines may be active at any time, and a low priority event may be pre-empted by one of higher priority.

Event Controller

- The processor employs a two-level event control mechanism.
  - The processor System Interrupt Controller (SIC) works with
  - The Core Event Controller (CEC) to prioritize and control all system interrupts.
- The SIC provides mapping between the many peripheral interrupt sources and the prioritized general-purpose interrupt inputs of the core.
  - This mapping is programmable, and individual interrupt sources can be masked in the SIC.

Event Controller

- The CEC supports nine (9) general-purpose interrupts (IVG7 – IVG15) in addition to the dedicated interrupt and exception events that are described in Table 4-6.
  - It is recommended that the
    - two lowest priority interrupts (IVG14 and IVG15) be reserved for software interrupt handlers, leaving
    - seven prioritized interrupt inputs (IVG7 – IVG13) to support the system.
  - Refer to the next table for details.

Core Event Mapping

Table 4-6. System and Core Event Mapping

	Event Source	Core Event Name
Core Events	Emulation (highest priority)	EMU
	Reset	RST
	NMI	NMI
	Exception	EVX
	Reserved	–
	Hardware Error	IVHW
	Core Timer	IVTMR

System Event Mapping

Table 4-6. System and Core Event Mapping (Cont'd)

	Event Source	Core Event Name
System Interrupts	PLL Wakeup Interrupt DMA Error (generic) PPI Error Interrupt SPORT0 Error Interrupt SPORT1 Error Interrupt SPI Error Interrupt UART Error Interrupt	IVG7
	Real-Time Clock Interrupts DMA0 Interrupt (PPI)	IVG8
	DMA1 Interrupt (SPORT0 RX) DMA2 Interrupt (SPORT0 TX) DMA3 Interrupt (SPORT1 RX) DMA4 Interrupt (SPORT1 TX)	IVG9
	DMA5 Interrupt (SPI) DMA6 Interrupt (UART RX) DMA7 Interrupt (UART TX)	IVG10
	Timer0, Timer1, Timer2 Interrupts	IVG11
	Programmable Flags Interrupt A/B	IVG12
	DMA8/9 Interrupt (Memory DMA Stream 0) DMA10/11 Interrupt (Memory DMA Stream 1) Software Watchdog Timer	IVG13
	Software Interrupt 1	IVG14
	Software Interrupt 2 (lowest priority)	IVG15

Processor Event Controller

- Processor Event Controller Consists of 2 stages:
  - The Core Event Controller (CEC)
  - System Interrupt Controller (SIC)

- Conceptually:
  - Interrupts from the peripherals arrive at SIC
  - SIC routes interrupts directly to general-purpose interrupts of the CEC.

Core Event Controller (CEC)

- CEC supports
  - 9 general-purpose interrupts: IVG15-7
    - IVG15-14 – (2) lowest priority interrupts for software handlers.
    - IRVG13-7 – (7) highest to support peripherals.
  - Additional dedicated interrupt and exception events.

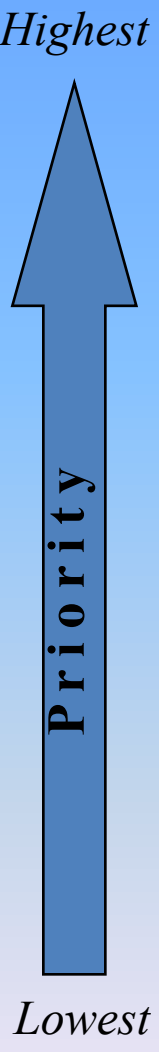
System Interrupt Controller (SIC)

- SIC provides mapping and routing of events:
  - From: Peripheral interrupt sources
  - To: Prioritized general-purpose interrupt inputs of the CEC.
- Note: Processor default mapping can be altered by the user via Interrupt Assignment Register (IAR).

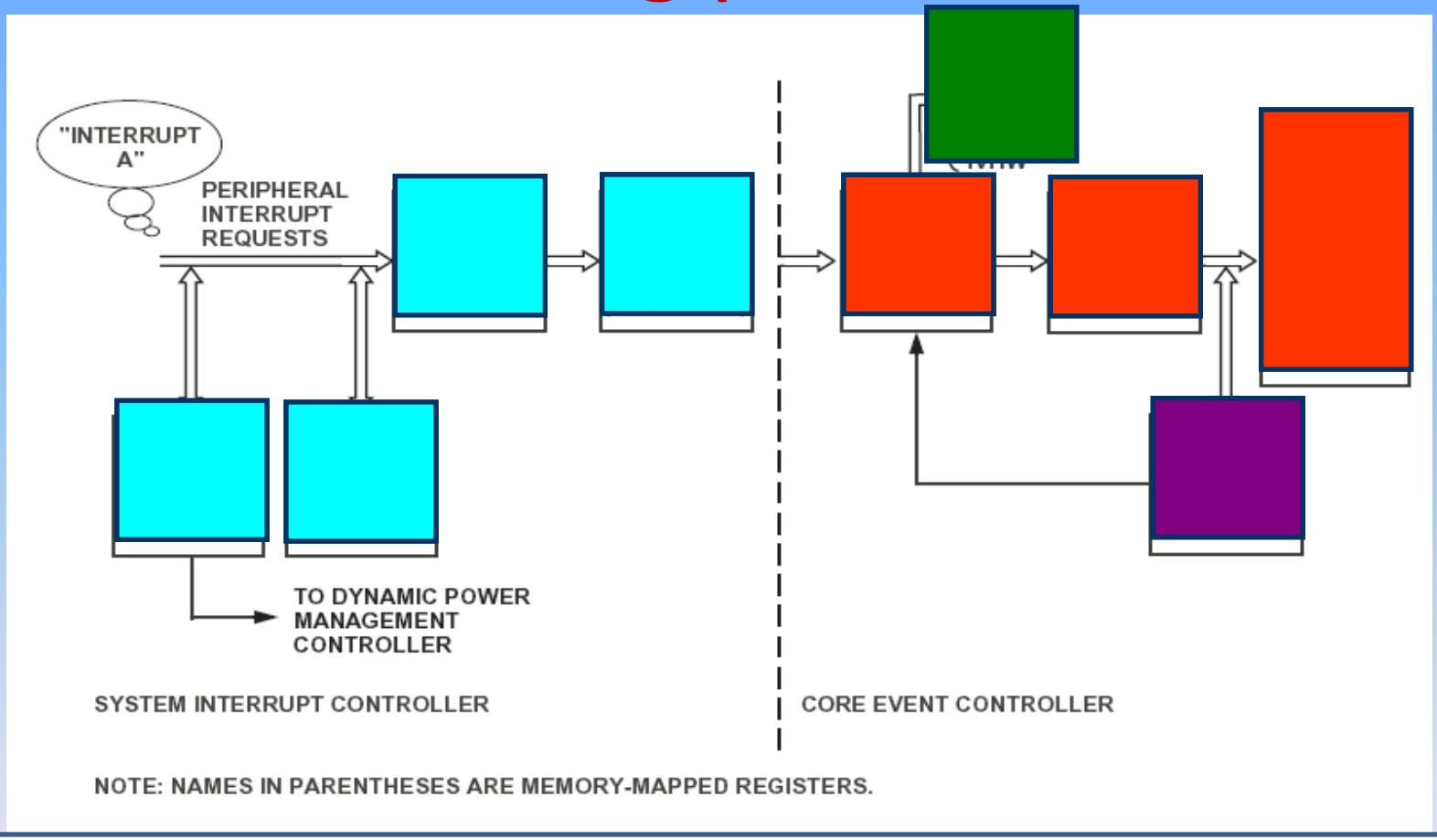
BF533 System & Core Interrupt Controllers

System Interrupt Source	IVG # [1]
PLL Wakeup interrupt	IVG7
DMA error (generic)	IVG7
PPI error interrupt	IVG7
SPORT0 error interrupt	IVG7
SPORT1 error interrupt	IVG7
SPI error interrupt	IVG7
UART error interrupt	IVG7
RTC interrupt	IVG8
DMA 0 interrupt (PPI)	IVG8
DMA 1 interrupt (SPORT0 RX)	IVG9
DMA 2 interrupt (SPORT0 TX)	IVG9
DMA 3 interrupt (SPORT1 RX)	IVG9
DMA 4 interrupt (SPORT1 TX)	IVG9
DMA 5 interrupt (SPI)	IVG10
DMA 6 interrupt (UART RX)	IVG10
DMA 7 interrupt (UART TX)	IVG10
Timer0 interrupt	IVG11
Timer1 interrupt	IVG11
Timer2 interrupt	IVG11
PF interrupt A	IVG12
PF interrupt B	IVG12
DMA 8/9 interrupt (MemDMA0)	IVG13
DMA 10/11 interrupt (MemDMA1)	IVG13
Watchdog Timer Interrupt	IVG13

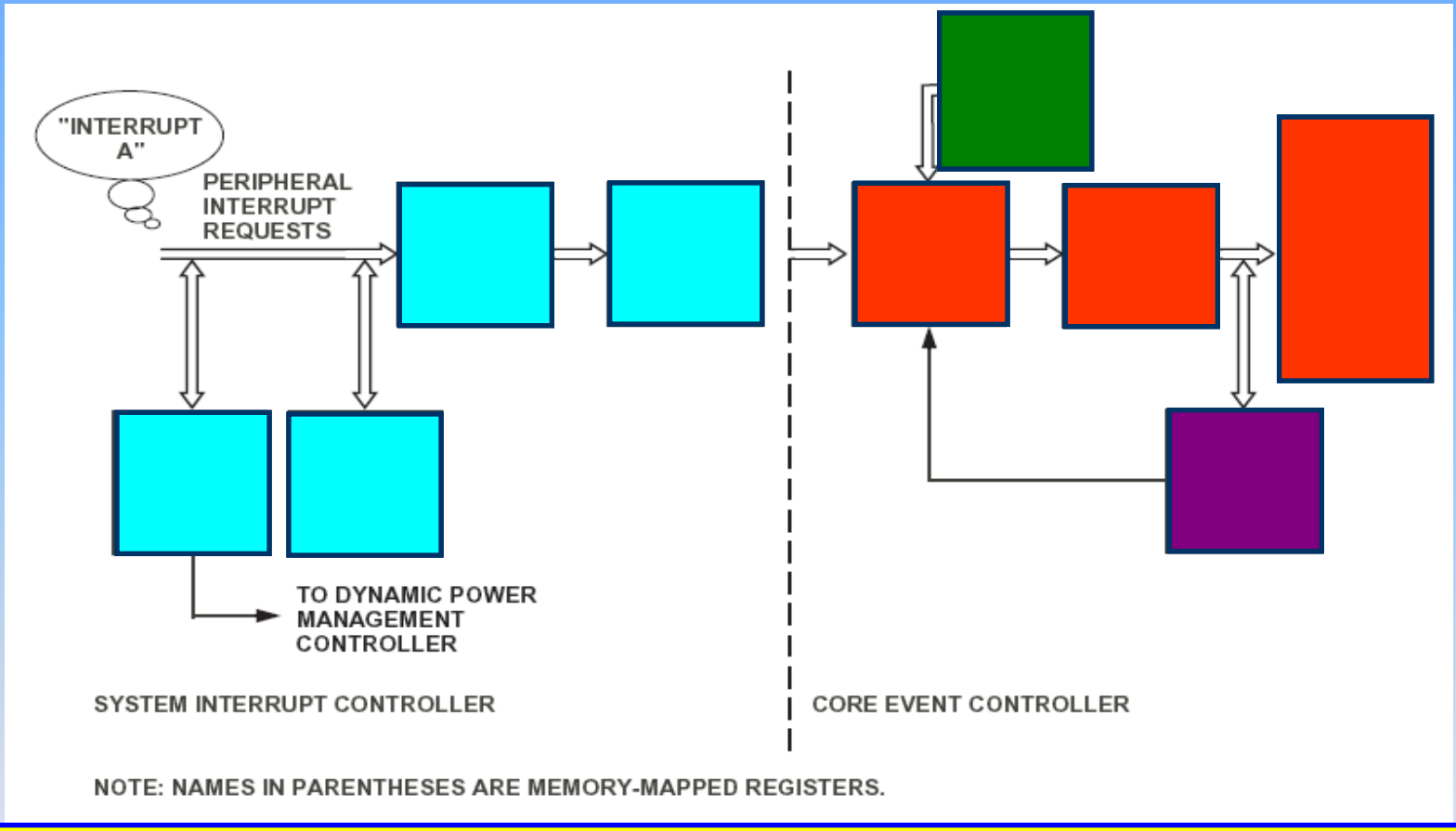
Event Source	IVG #	Core Event Name
Emulator	0	EMU
Reset	1	RST
Non Maskable Interrupt	2	NMI
Exceptions	3	EVSW
Reserved	4	-
Hardware Error	5	IVHW
Core Timer	6	IVTMR
General Purpose 7	7	IVG7
General Purpose 8	8	IVG8
General Purpose 9	9	IVG9
General Purpose 10	10	IVG10
General Purpose 11	11	IVG11
General Purpose 12	12	IVG12
General Purpose 13	13	IVG13
General Purpose 14	14	IVG14
General Purpose 15	15	IVG15



System Interrupt Processing: the big picture

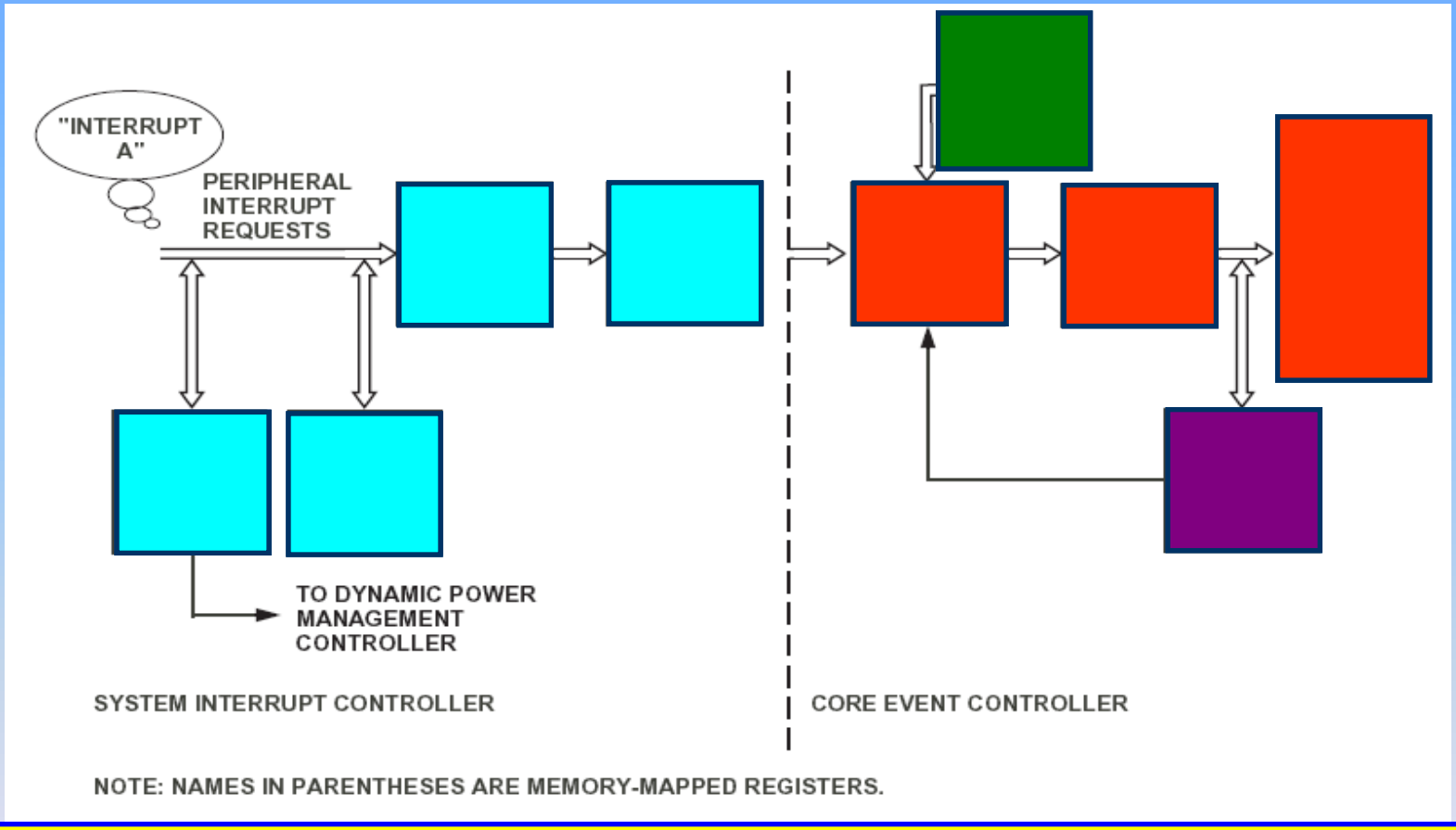


Interrupt Processing: step 1



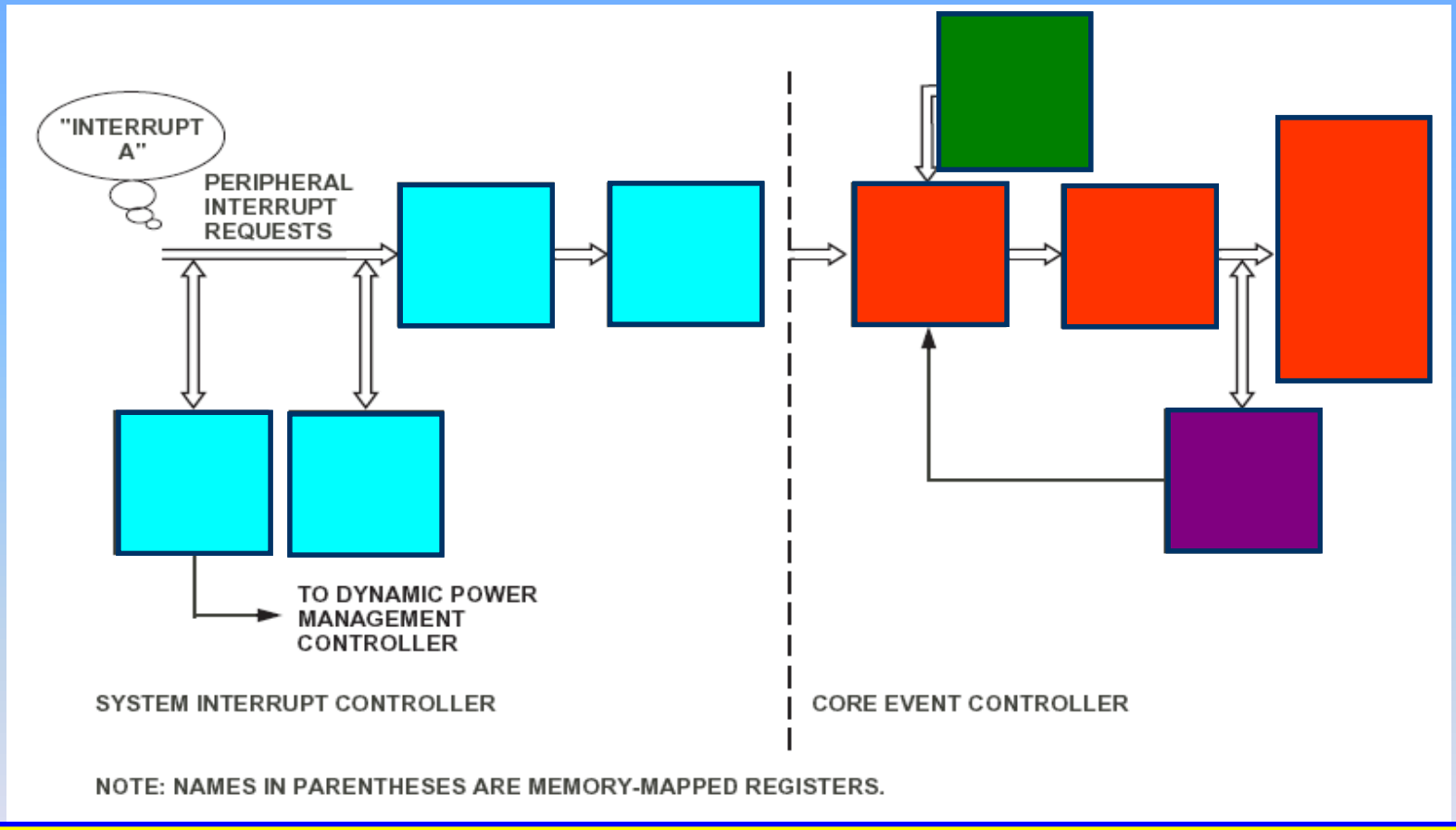
1. SIC_ISR logs the request and keeps track of system interrupts that are asserted but not yet serviced (that is, an interrupt service routine hasn't yet cleared the interrupt).

Interrupt Processing: step 2



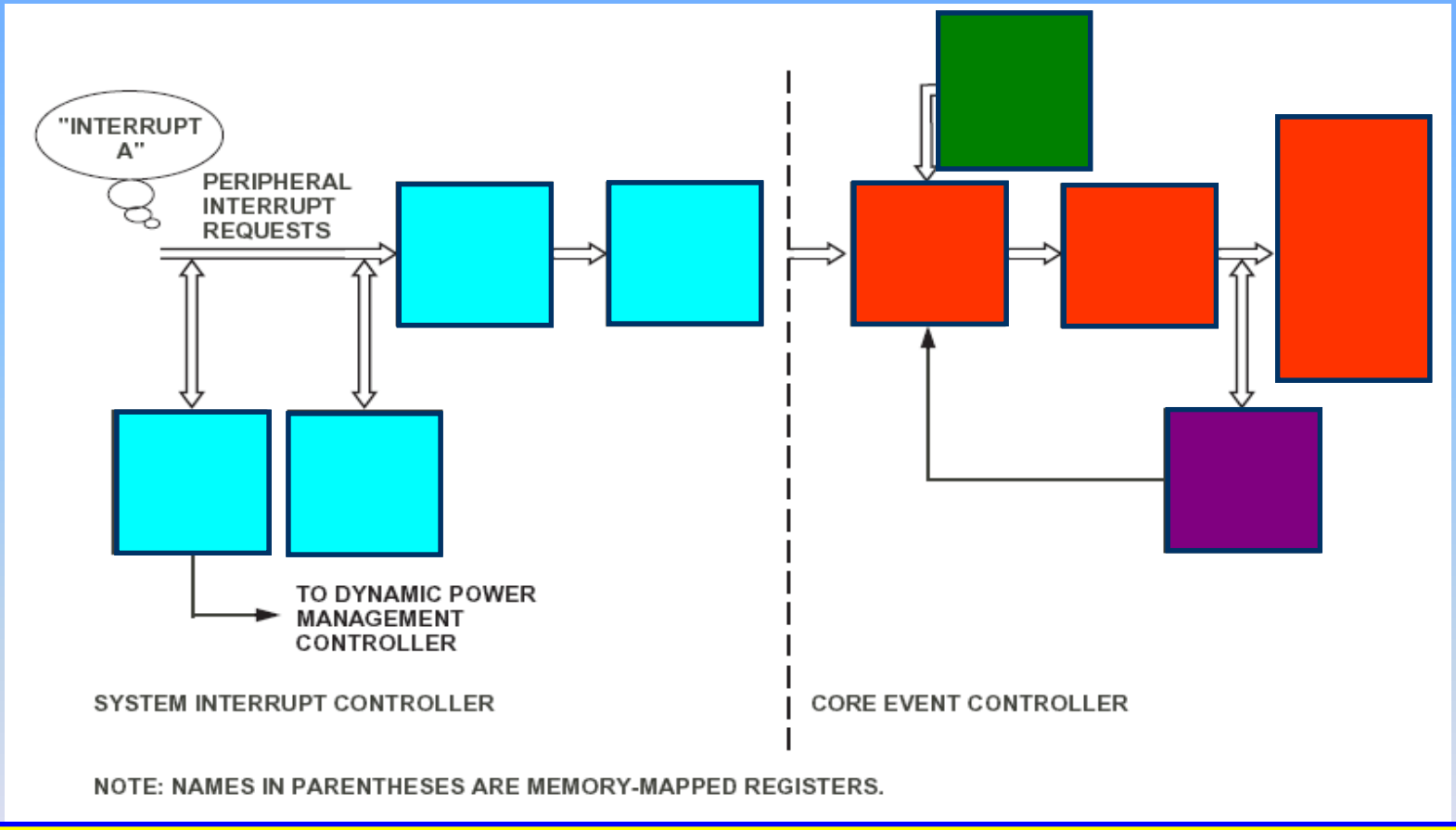
2. SIC_IWR checks to see if it should wake up the core from an idled state based on this interrupt request.

Interrupt Processing: step 3



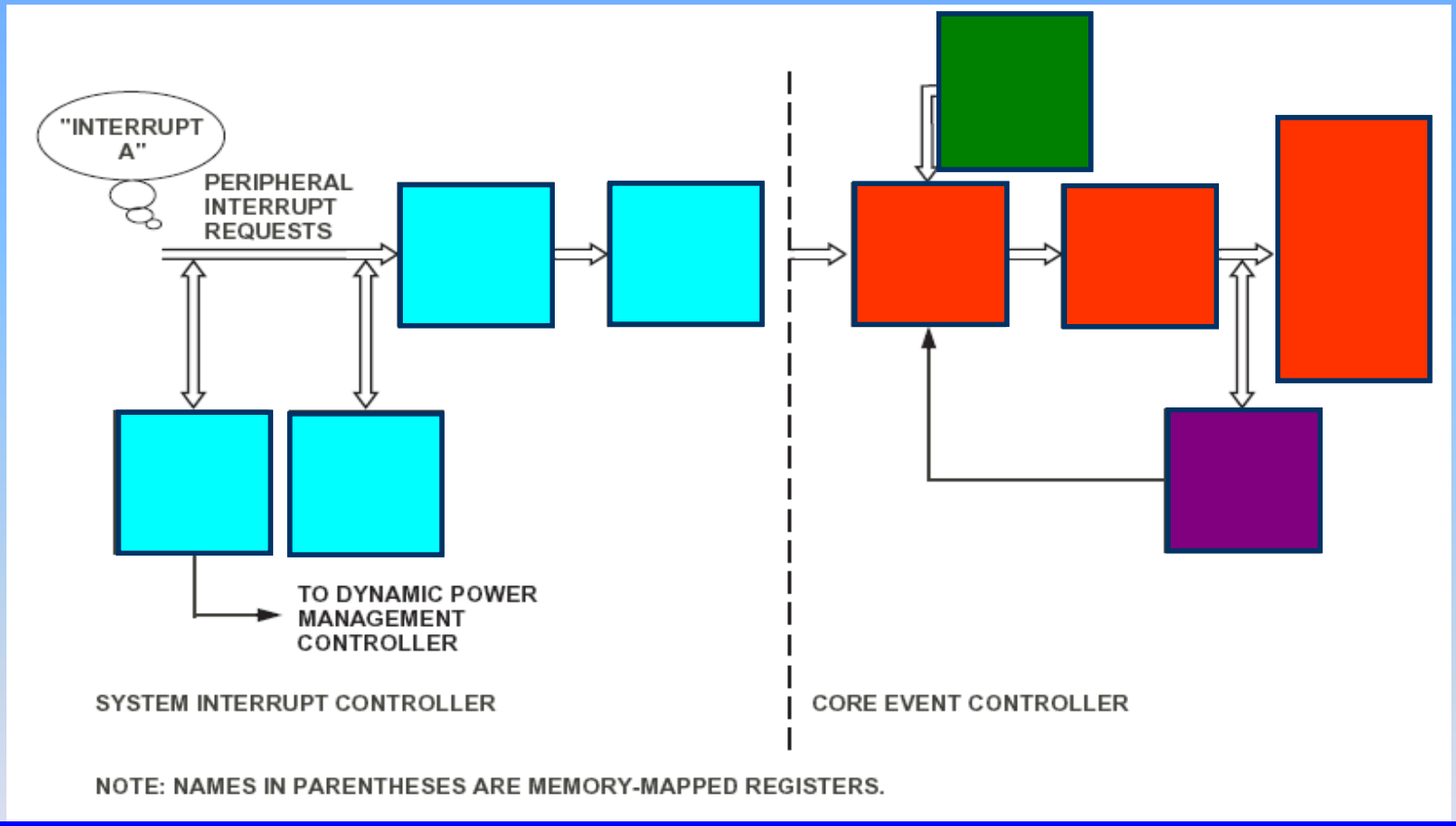
3. SIC_IMASK masks off or enables interrupts from peripherals at the system level. If Interrupt A is not masked, the request proceeds to Step 4.

Interrupt Processing: step 4



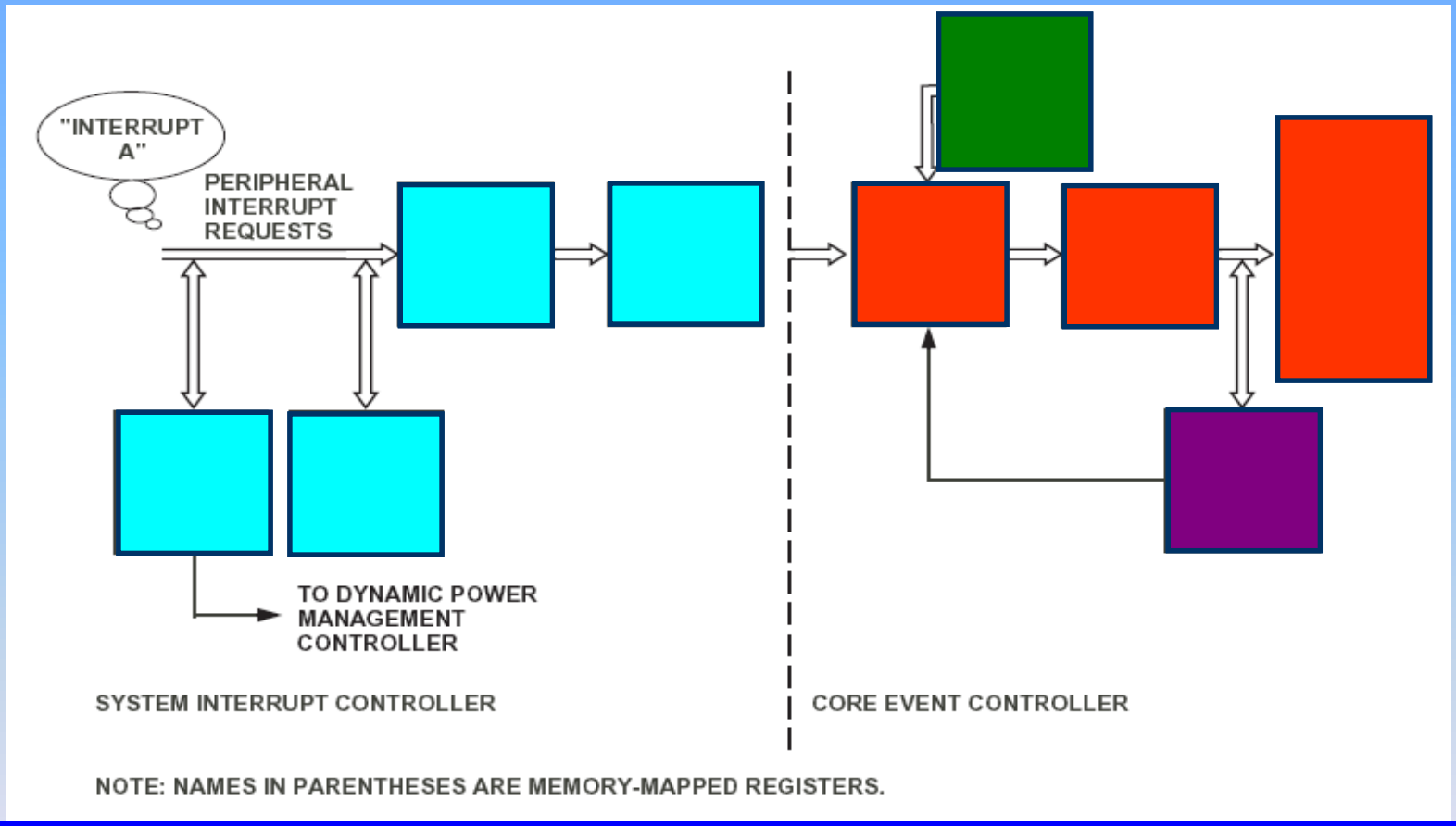
4. The SIC_IARx registers, which map the peripheral interrupts to a smaller set of general-purpose core interrupts (IVG7 – IVG15), determine the core priority of Interrupt A.

Interrupt Processing: step 5



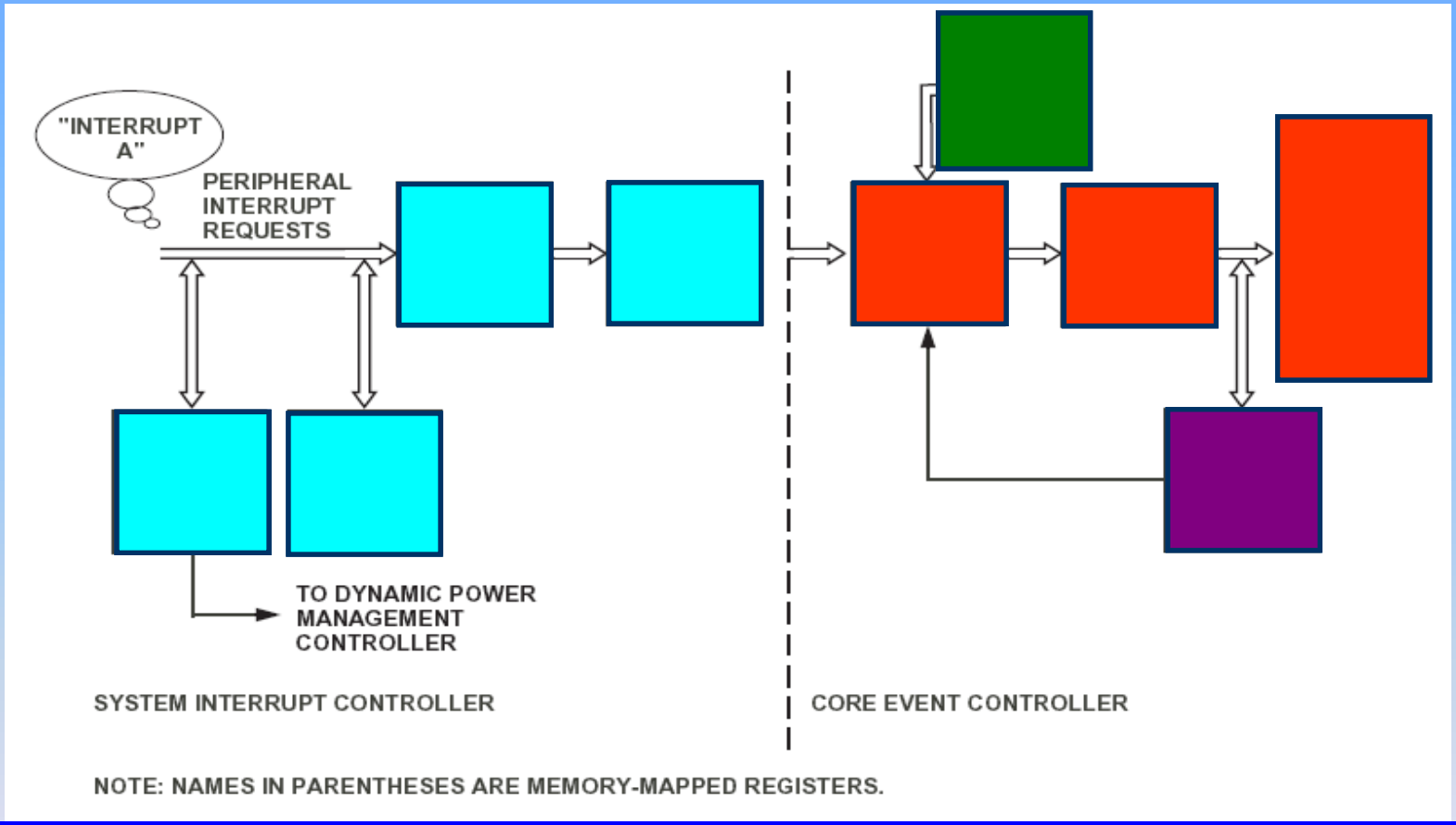
5. ILAT adds Interrupt A to its log of interrupts latched by the core but not yet actively being serviced.

Interrupt Processing: step 6



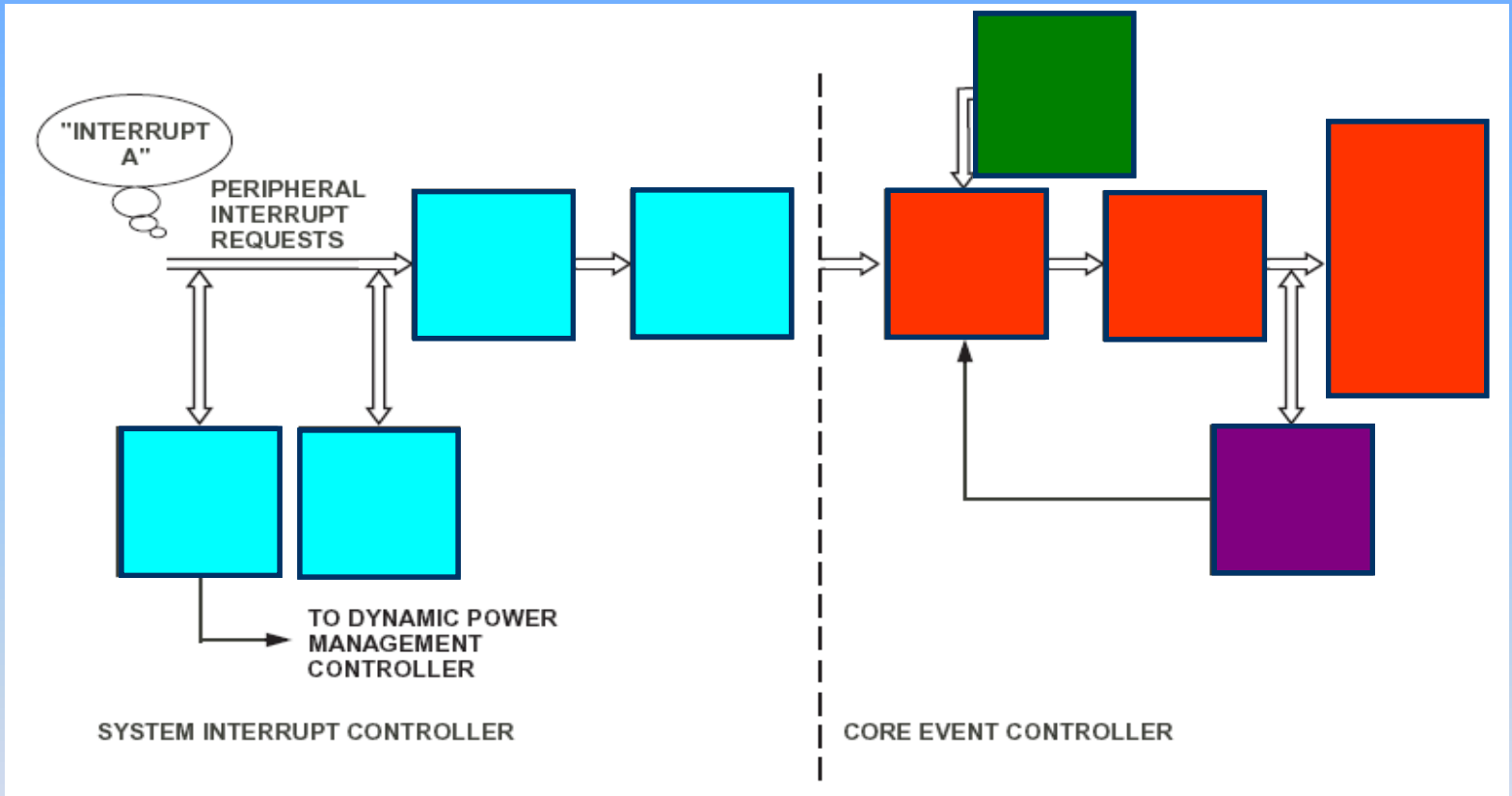
6. IMASK masks off or enables events of different core priorities. If the IVGx event corresponding to Interrupt A is not masked, the process proceeds to Step 7.

Interrupt Processing: step 7



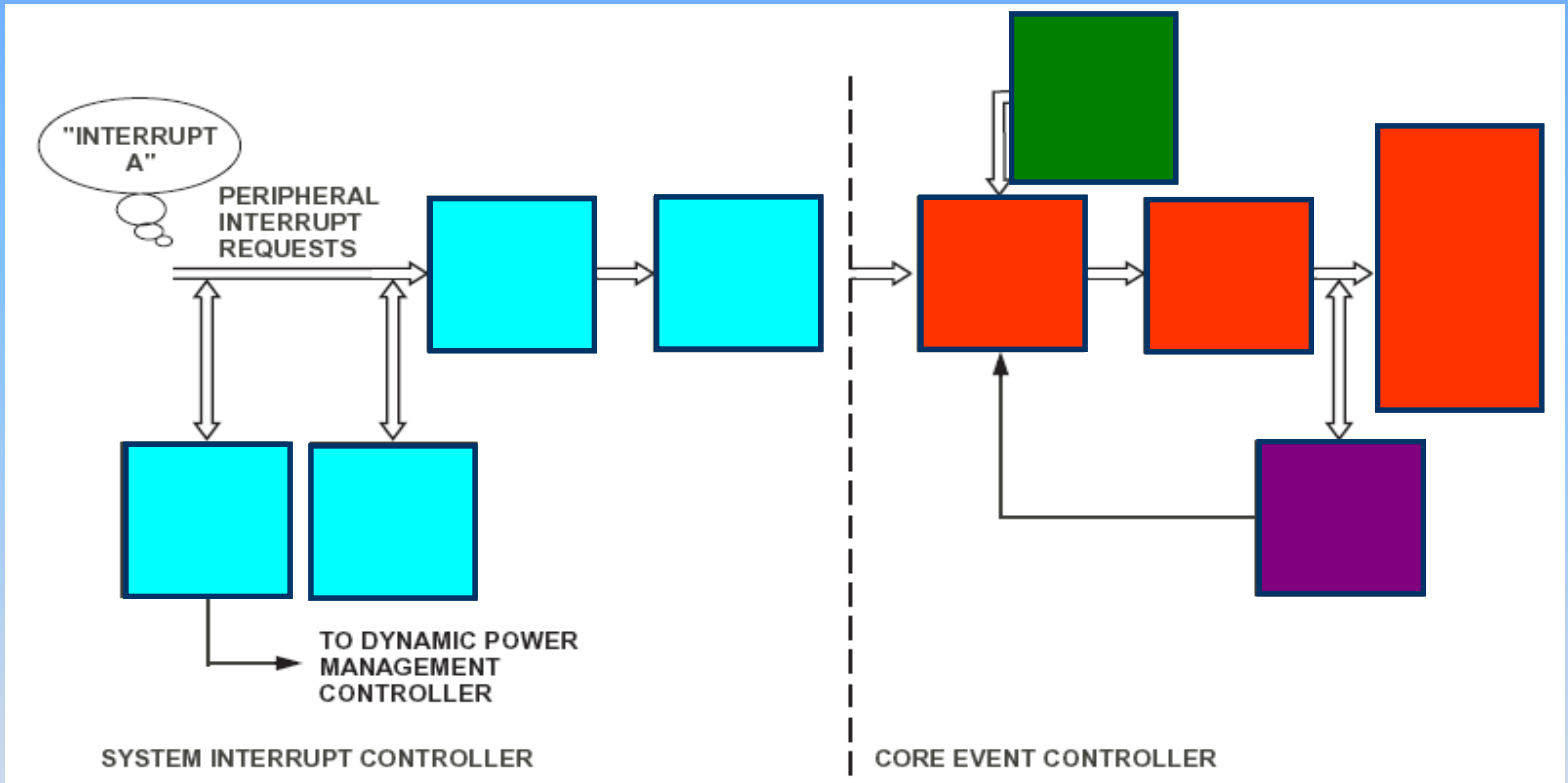
7. The Event Vector Table (EVT) is accessed to look up the appropriate vector for Interrupt A's interrupt service routine (ISR).

Interrupt Processing: step 8



8. When the event vector for Interrupt A has entered the core pipeline, the appropriate IPEND bit is set, which clears the respective ILAT bit. Thus, IPEND tracks all pending interrupts, as well as those being presently serviced.

Interrupt Processing: step 9



9. When the interrupt service routine (ISR) for Interrupt A has been executed, the RTI instruction clears the appropriate IPEND bit. However, the relevant SIC_ISR bit is not cleared unless the interrupt service routine clears the mechanism that generated Interrupt A, or if the process of servicing the interrupt clears this bit.

System Interrupt Processing

Note: emulation, reset, NMI, and exception events, as well as hardware error (IVHW) and core timer (IVTMR) interrupt requests, enter the interrupt processing chain at the ILAT level and are not affected by the system

System Peripheral Interrupts

- The processor system has numerous peripherals, which therefore require many supporting interrupts. Table 4-7 lists:
  - The Peripheral Interrupt source
  - The Peripheral Interrupt ID used in the System Interrupt Assignment registers (SIC_IARx). See “System Interrupt Assignment Registers (SIC_IARx)” on page 4-30 of BF533 HRM.
  - The general-purpose interrupt of the core to which the interrupt maps at reset
  - The Core Interrupt ID used in the System Interrupt Assignment registers (SIC_IARx). See “System Interrupt Assignment Registers (SIC_IARx)” on page 4-30 of BF533 HRM.

Peripheral Interrupt Source at Reset State

Table 4-7. Peripheral Interrupt Source Reset State

Peripheral Interrupt Source	Peripheral Interrupt ID	General-purpose Interrupt (Assignment at Reset)	Core Interrupt ID
PLL Wakeup Interrupt	0	IVG7	0
DMA Error (generic)	1	IVG7	0
PPI Error Interrupt	2	IVG7	0
SPORT0 Error Interrupt	3	IVG7	0
SPORT1 Error Interrupt	4	IVG7	0
SPI Error Interrupt	5	IVG7	0
UART Error Interrupt	6	IVG7	0
Real-Time Clock Interrupts (alarm, second, minute, hour, countdown)	7	IVG8	1
DMA 0 Interrupt (PPI)	8	IVG8	1
DMA 1 Interrupt (SPORT0 RX)	9	IVG9	2

Peripheral Interrupt Source at Reset State

Table 4-7. Peripheral Interrupt Source Reset State (Cont'd)

Peripheral Interrupt Source	Peripheral Interrupt ID	General-purpose Interrupt (Assignment at Reset)	Core Interrupt ID
DMA 2 Interrupt (SPORT0 TX)	10	IVG9	2
DMA 3 Interrupt (SPORT1 RX)	11	IVG9	2
DMA 4 Interrupt (SPORT1 TX)	12	IVG9	2
DMA 5 Interrupt (SPI)	13	IVG10	3
DMA 6 Interrupt (UART RX)	14	IVG10	3
DMA 7 Interrupt (UART TX)	15	IVG10	3
Timer0 Interrupt	16	IVG11	4
Timer1 Interrupt	17	IVG11	4
Timer2 Interrupt	18	IVG11	4
PF Interrupt A	19	IVG12	5
PF Interrupt B	20	IVG12	5
DMA 8/9 Interrupt (Memory DMA Stream 0)	21	IVG13	6
DMA 10/11 Interrupt (Memory DMA Stream 1)	22	IVG13	6
Software Watchdog Timer Interrupt	23	IVG13	6
Reserved	24-31	-	-

Initialization of Interrupts

- If the default assignments shown in Table 4-7 are acceptable, then interrupt initialization involves only:
  1. Initialization of the core Event Vector Table (EVT) vector address entries. ***Why is this needed?***
  2. Initialization of the IMASK register
  3. Unmasking the specific peripheral interrupts in SIC_IMASK that the system requires

Initialization of Core Vector Table

- The Event Vector Table (EVT) is a hardware table with ***sixteen*** entries that are each ***32 bits wide***.

- The EVT contains an entry for each possible core event.
  - Entries are accessed as MMRs, and
  - Each entry can be programmed at reset with the corresponding vector address for the interrupt service routine.

Initialization of Core Vector Table

When an event occurs, instruction fetch starts at the address location in the EVT entry for that event. The processor architecture allows unique addresses to be programmed into each of the interrupt vectors; that is, interrupt vectors are not determined by a fixed offset from an interrupt vector table base address.

- This approach minimizes latency by not requiring a long jump from the vector table to the actual ISR code.

Initialization of Core Vector Table

- next table lists events by priority.
- Each event has a corresponding bit in the event state registers:
  - ILAT,
  - IMASK, and
  - IPEND

Core Event Vector Table

Table 4-9. Core Event Vector Table

Event Number	Event Class	Name	MMR Location	Notes
EVT0	Emulation	EMU	0xFFE0 2000	Highest priority. Vector address is provided by JTAG.
EVT1	Reset	RST	0xFFE0 2004	
EVT2	NMI	NMI	0xFFE0 2008	
EVT3	Exception	EVX	0xFFE0 200C	
EVT4	Reserved	Reserved	0xFFE0 2010	Reserved vector
EVT5	Hardware Error	IVHW	0xFFE0 2014	
EVT6	Core Timer	IVTMR	0xFFE0 2018	
EVT7	Interrupt 7	IVG7	0xFFE0 201C	
EVT8	Interrupt 8	IVG8	0xFFE0 2020	
EVT9	Interrupt 9	IVG9	0xFFE0 2024	

Core Event Vector Table

Table 4-9. Core Event Vector Table (Cont'd)

Event Number	Event Class	Name	MMR Location	Notes
EVT10	Interrupt 10	IVG10	0xFFE0 2028	
EVT11	Interrupt 11	IVG11	0xFFE0 202C	
EVT12	Interrupt 12	IVG12	0xFFE0 2030	
EVT13	Interrupt 13	IVG13	0xFFE0 2034	
EVT14	Interrupt 14	IVG14	0xFFE0 2038	
EVT15	Interrupt 15	IVG15	0xFFE0 203C	Lowest priority

Initialization.c

```
//-------------------------------------------------------------------------//
// Function:        Init_Interrupts                                        //
//                                                                         //
// Parameters:      None                                                   //
//                                                                         //
// Return:                    None                                         //
//                                                                         //
// Description: This function initializes the interrupts for Timer0 and    //
//        FlagA (PF8).                                                     //
//-------------------------------------------------------------------------//
void Init_Interrupts(void)
{
    // assign core IDs to interrupts
    *pSIC_IAR0 = 0xffffffff;
    *pSIC_IAR1 = 0xffffffff;
    *pSIC_IAR2 = 0xffff5ff4;  // Timer0 -> ID4; FlagA -> ID5

    // assign ISRs to interrupt vectors
    register_handler(ik_ivg11, Timer0_ISR);       // Timer0 ISR -> IVG 11
    register_handler(ik_ivg12, FlagA_ISR);        // FlagA ISR -> IVG 12

    // enable Timer0 and FlagA interrupt
    *pSIC_IMASK = 0x00090000;
}
```

System Interrupt Assignment Register

System Interrupt Assignment Registers (SIC_IARx)

- The relative priority of peripheral interrupts can be set by mapping the peripheral interrupt to the appropriate general-purpose interrupt level in the core.

- The mapping is controlled by the System Interrupt Assignment register settings, as detailed in the next slides

System Interrupt Assignment Registers (SIC_IARx)

- If more than one interrupt source is mapped to the same interrupt, they are logically OR-ed, with no hardware prioritization. Software can prioritize the interrupt processing as required for a particular system application.

Mapping of Values in SIC_IARx Register to General Purpose Interrupt

- Table 4-8 defines the value to write in SIC_IARx to configure a peripheral for a particular IVG priority.

Table 4-8. IVG Select Definitions

General-purpose Interrupt	Value in SIC_IAR
IVG7	0
IVG8	1
IVG9	2
IVG10	3
IVG11	4
IVG12	5
IVG13	6
IVG14	7
IVG15	8

System Interrupt Assignment Register

```
0xffffffff = 1111 1111 1111 1111 1111 1111 1111 1111
```

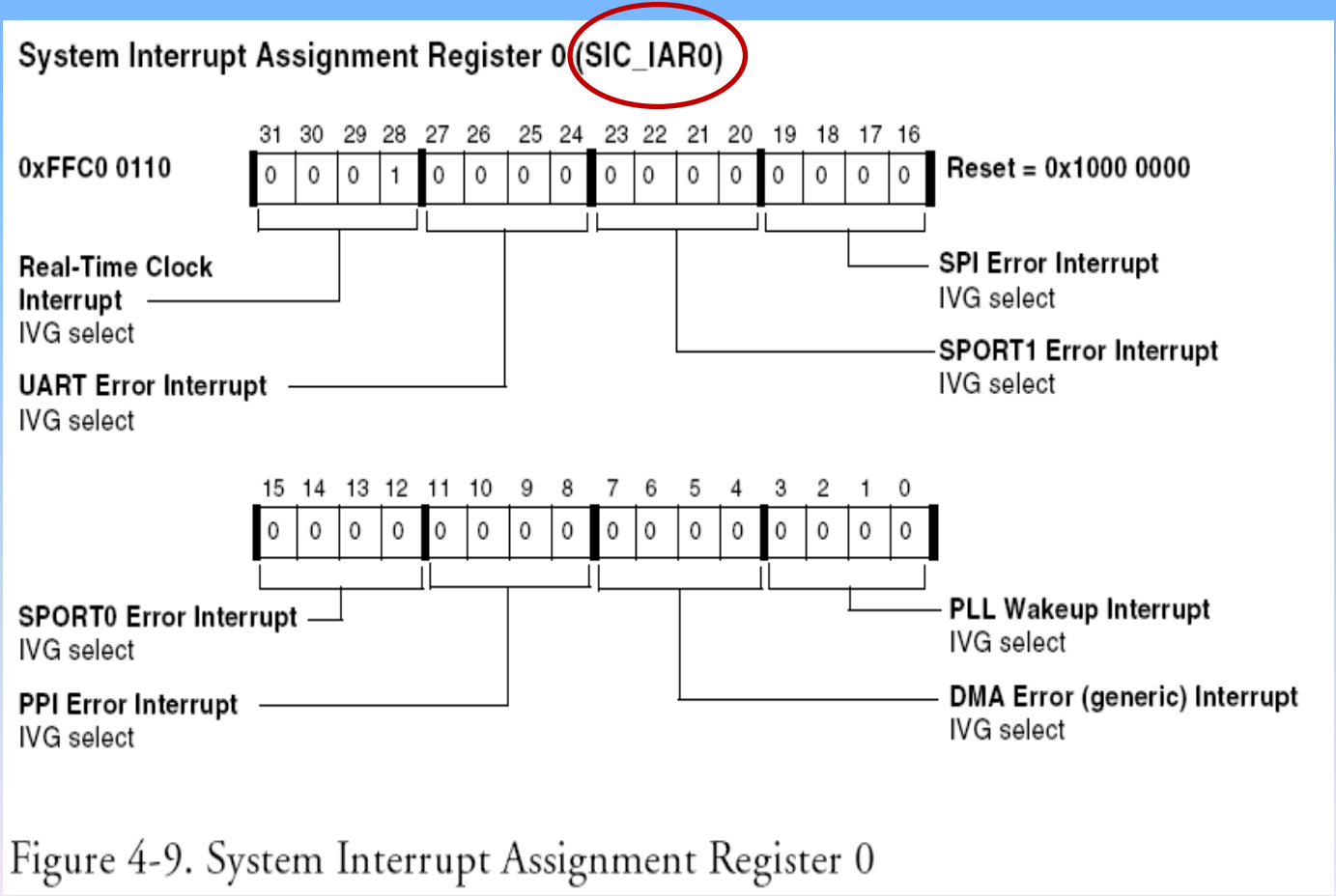


Figure 4-9. System Interrupt Assignment Register 0

System Interrupt Assignment Register

```
0xffffffff = 1111 1111 1111 1111 1111 1111 1111 1111
```

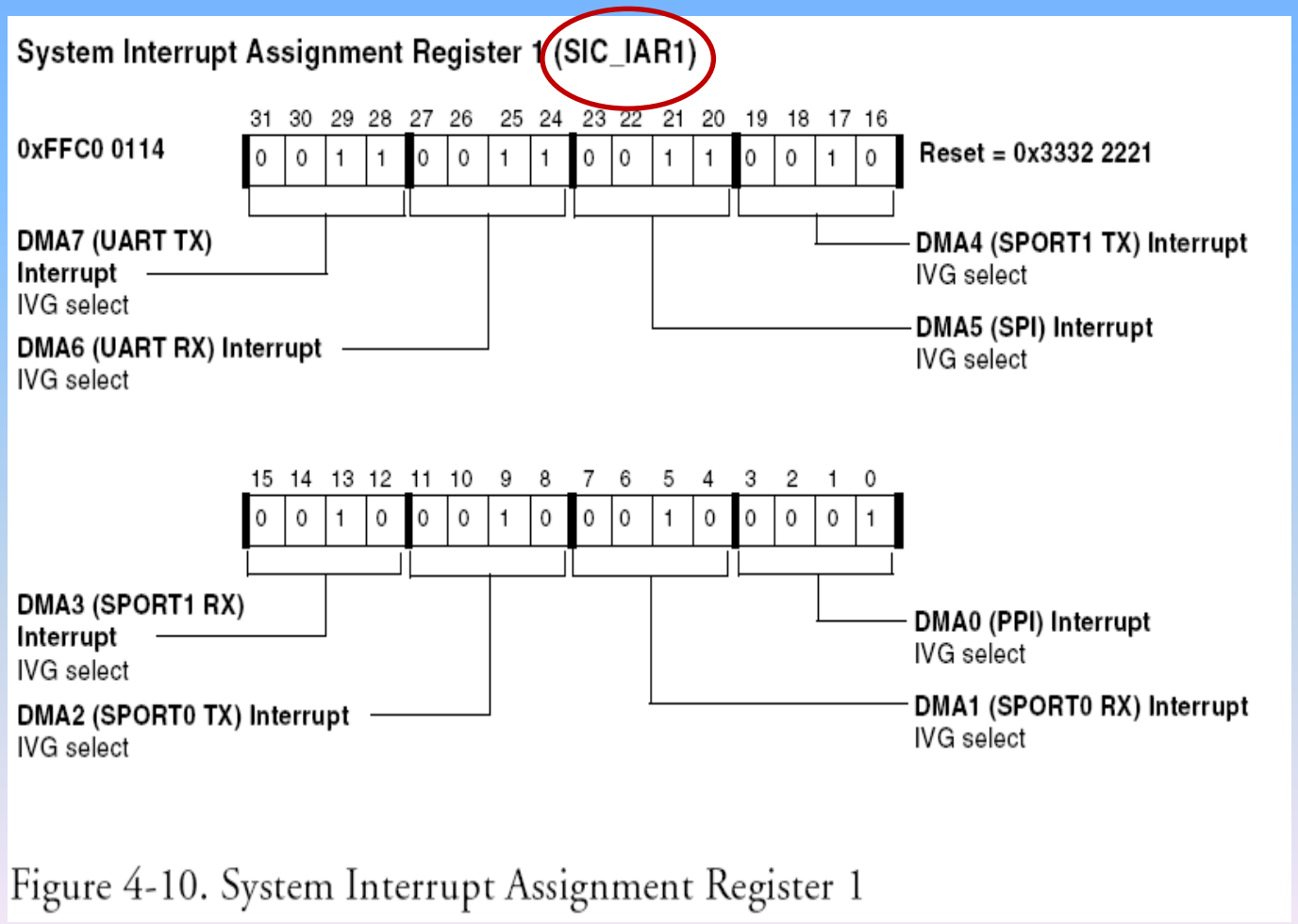


Figure 4-10. System Interrupt Assignment Register 1

System Interrupt Assignment Register

```
0xffff5ff4 = 1111 1111 1111 1111 0101 1111 1111 0100
```

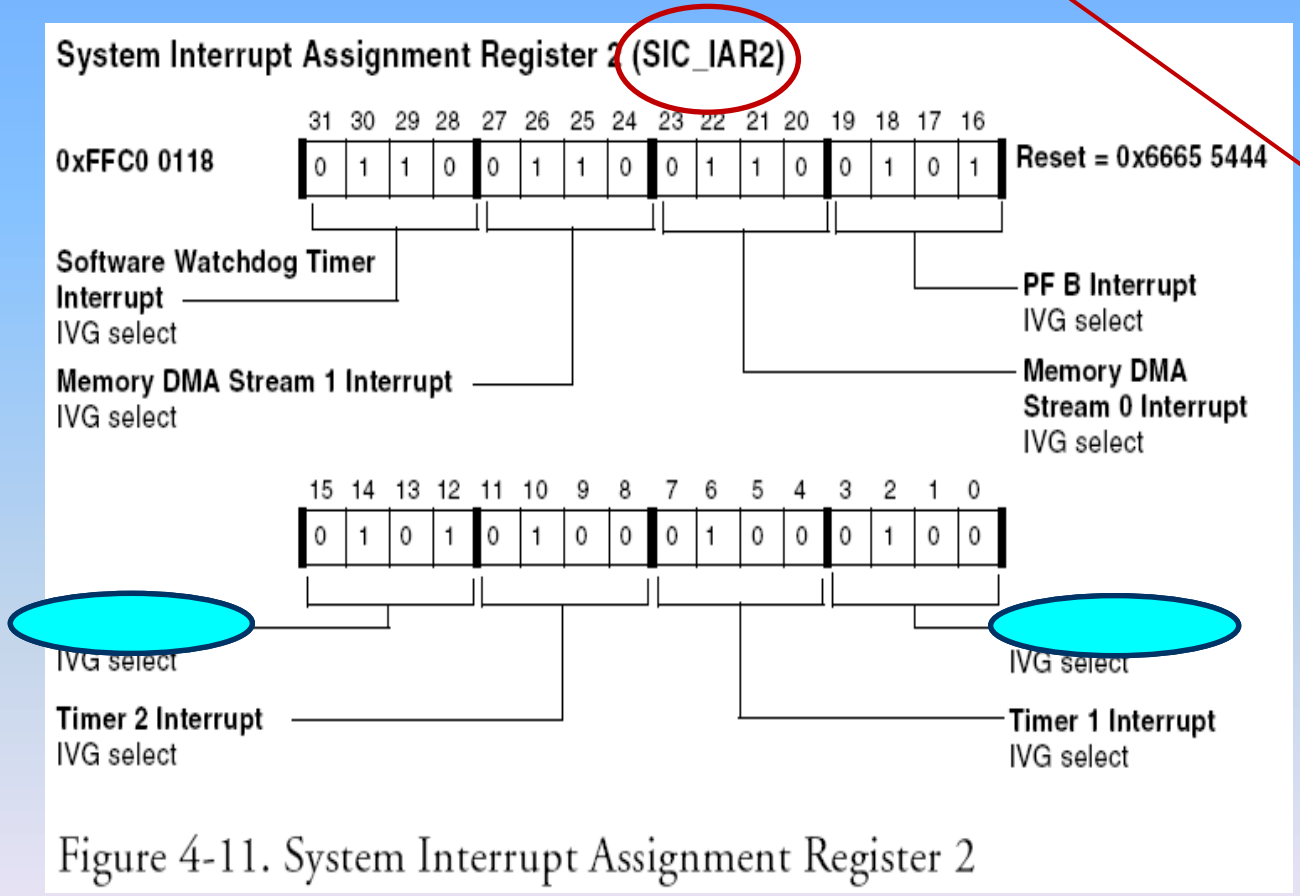


Figure 4-11. System Interrupt Assignment Register 2

Timer 0 interrupt

PFA interrupt

Initialization.c

```
//-------------------------------------------------------------------------//
// Function:        Init_Interrupts                                        //
//                                                                         //
// Parameters:      None                                                   //
//                                                                         //
// Return:                    None                                         //
//                                                                         //
// Description: This function initializes the interrupts for Timer0 and    //
//        FlagA (PF8).                                                     //
//-------------------------------------------------------------------------//
void Init_Interrupts(void)
{
    // assign core IDs to interrupts
    *pSIC_IAR0 = 0xffffffff;
    *pSIC_IAR1 = 0xffffffff;
    *pSIC_IAR2 = 0xffff5ff4;  // Timer0 -> ID4; FlagA -> ID5

    // assign ISRs to interrupt vectors
    register_handler(ik_ivg11, Timer0_ISR);       // Timer0 ISR -> IVG 11
    register_handler(ik_ivg12, FlagA_ISR);        // FlagA ISR -> IVG 12

    // enable Timer0 and FlagA interrupt
    *pSIC_IMASK = 0x00090000;
}
```

System Interrupt Assignment Register

Initialization.c

```
//-------------------------------------------------------------------------//
// Function:        Init_Interrupts                                        //
//                                                                         //
// Parameters:      None                                                   //
//                                                                         //
// Return:                    None                                         //
//                                                                         //
// Description: This function initializes the interrupts for Timer0 and    //
//        FlagA (PF8).                                                     //
//-------------------------------------------------------------------------//
void Init_Interrupts(void)
{
    // assign core IDs to interrupts
    *pSIC_IAR0 = 0xffffffff;
    *pSIC_IAR1 = 0xffffffff;
    *pSIC_IAR2 = 0xffff5ff4;  // Timer0 -> ID4; FlagA -> ID5

    // assign ISRs to interrupt vectors
    register_handler(ik_ivg11, Timer0_ISR);       // Timer0 ISR -> IVG 11
    register_handler(ik_ivg12, FlagA_ISR);         // FlagA ISR -> IVG 12

    // enable Timer0 and FlagA interrupt
    *pSIC_IMASK = 0x00090000;
}
```

Unmasking of the Interrupt by setting corresponding bits in SIC_IMASK register

Timer Status Register (TIMER_STATUS)

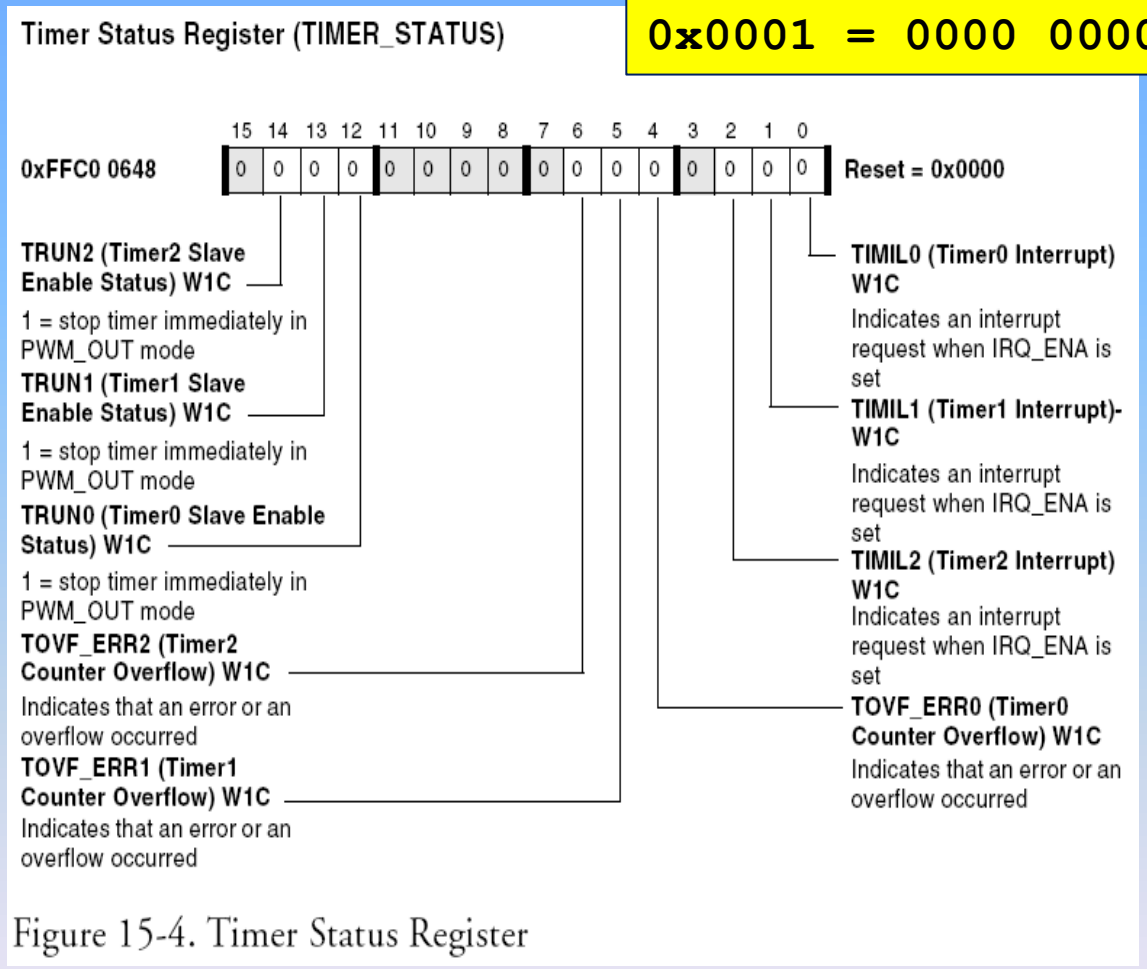


Figure 15-4. Timer Status Register

0x0001 = 0000 0000 0000 0001

Timer 0 interrupt enable

ISR.c: ISR for Timer0

```
#include "BF533 Flags.h"
//-------------------------------------------------------------------------//
// Function:        Timer0_I
//
//-----------------------------------------
EX_INTERRUPT_HANDLER(Timer0_ISR)
{
    static unsigned char ucActive_LED = 0x01;

    // confirm interrupt handling
    *pTIMER_STATUS = 0x0001;

    // shift old LED pattern by one
    if(sLight_Move_Direction)
    {
          if((ucActive_LED = ucActive_LED >> 1) == 0x00) ucActive_LED = 0x20;
    }
    else
    {
          if((ucActive_LED = ucActive_LED << 1) == 0x40) ucActive_LED = 0x01;
    }

    // write new LED pattern to Port B
    *pFlashA_PortB_Data = ucActive_LED;
}
```

0x0001 = 0000 0000 0000 0001

0x01 >> 1 i.e. 0000 0001 >> 1
=> 0x00 i.e. 0000 0000
then
0x20 = 0010 0000
i.e. 6th LED is switched on

0x40 = 0100 0000
go back to 1st LED on

Send LED Pattern to FlashA PortB

ISR.c: ISR for Push-Buttons

```
//-------------------------------------------------------------------------//
// Function:        FlagA_ISR                                              //
//                                                                         //
// Parameters:      None                                                   //
//                                                                         //
// Return:                    None                                          //
//                                                                         //
// Description:  This ISR is called every time the button connected to PF8 //
//               is pressed.                                               //
//               The state of flag sLight_Move_Direction is changed, so the //
//               shift-direction for the LED pattern in Timer0_ISR changes. //
//-------------------------------------------------------------------------//
EX_INTERRUPT_HANDLER(FlagA_ISR)
{
    // confirm interrupt handling
    *pFIO_FLAG_C = 0x0100;

    // toggle direction of moving light
    sLight_Move_Direction = ~sLight_Move_Direction;
}
```

FIO_FLAG_S, FIO_FLAG_C, and FIO_FLAG_T Registers

- As discussed in FIO Set-up section, the
  - Flag Set register (FIO_FLAG_S),
  - Flag Clear register (FIO_FLAG_C), and
  - Flag Toggle register (FIO_FLAG_T) are used to:
    - Set, clear or toggle the output state associated with each output PFx pin
    - Clear the latched interrupt state captured from each input PFx pin

FIO_FLAG_C Register

0x0100 = 0000 0001 0000 0000

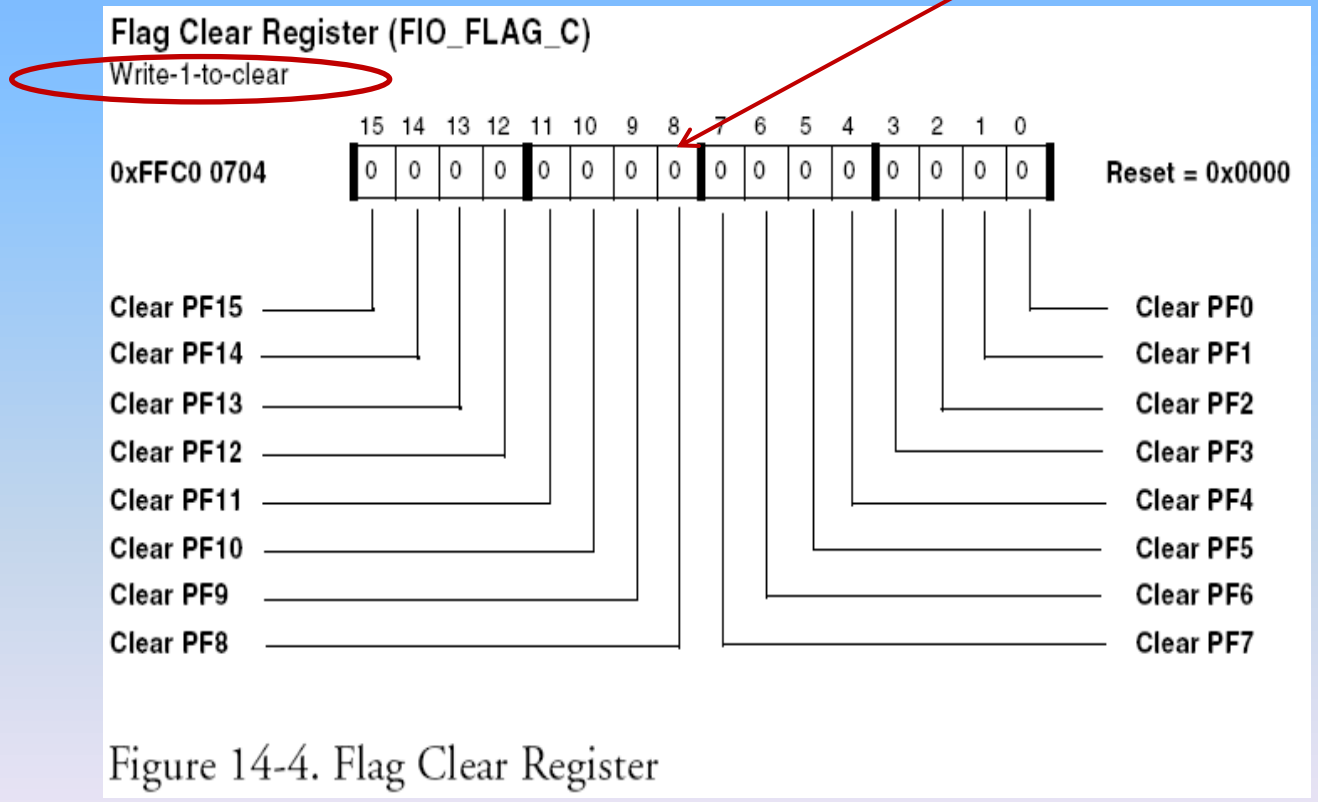


Figure 14-4. Flag Clear Register

ISR.c: ISR for Push-Buttons

```
//-------------------------------------------------------------------------//
// Function:        FlagA_ISR                                              //
//                                                                         //
// Parameters:      None                                                   //
//                                                                         //
// Return:                    None                                          //
//                                                                         //
// Description:  This ISR is called every time the button connected to PF8 //
//               is pressed.                                               //
//               The state of flag sLight_Move_Direction is changed, so the //
//               shift-direction for the LED pattern in Timer0_ISR changes. //
//-------------------------------------------------------------------------//
EX_INTERRUPT_HANDLER(FlagA_ISR)
{
    // confirm interrupt handling
    *pFIO_FLAG_C = 0x0100;

    // toggle direction of moving light
    sLight_Move_Direction = ~sLight_Move_Direction;
}
```

0x0100 = 0000 0001 0000 0000

Toggle direction flag