EMU

Energy Micro University

UM004 - Interrupts, DMA and PRS

This lesson starts with some basic theory about interrupts and it will be compared to the more basic alternative called polling. A larger code example will be presented at the end of the lesson as well as code exercises for the student to solve.

The second part of this lesson goes into how the MCU can transmit signals and read and write to memory without any interaction with the processor. This is called Peripheral Reflex System (PRS) and Direct Memory Access (DMA).

Concepts that will be introduced include:

- Hardware and software interrupts
- PUSHing and POPing registers
- Interrupt priorities
- Nested interrupts
- Interrupts vs. polling
- Direct Memory Access (DMA)
- Peripheral Reflex System (PRS)

This lesson includes:

- This PDF document
- Source files
  - Example C code
  - Multiple IDE projects

ENERGY micro

1 Interrupts

1.1 What are interrupts and why use them?

An interrupt is an asynchronous signal sent to the processor telling it that some action has to be done. An asynchronous signal is a signal that suddenly can arise without any clock signal telling it to, this in contrast to a steady periodic stream. The interrupt signal can occur from a button being pushed, a measured change in voltage etc. The CPU can execute other instructions in stead of constantly checking if some state has changed in a peripheral, a process called polling. If the CPU is not executing any instructions, it can sleep to reduce power consumption. The interrupts will then wake the CPU up when needed. The use of interrupts is also a way to organize the execution of instructions by giving different interrupts different priorities.

1.2 How interrupts work

When some action or change of state occurs in a peripheral, an interrupt flag is raised and a signal is sent to the CPU telling it to stop and save whatever it is executing and start executing the interrupt handler. The interrupt handler or interrupt service routine (ISR) is a set of instructions dealing with the interrupt. Once the instruction handler has completed, the program will return from where it was stopped. Interrupts can be divided into hardware interrupts and software interrupts.

1.2.1 Hardware interrupts

When hardware interrupts occurs the hardware itself produce an interrupt when some condition is met. When the interrupt occurs, the hardware sends a signal to jump to the interrupt controller without any software help.

1.2.2 Software interrupts

Software interrupts are implemented in the instruction set so that the programmer can generate them at will. The interrupt is handled in the same way as a hardware interrupt, but are often deferred interrupts that will be executed when all higher ranking interrupts are finished executing.

1.3 Interrupt handlers

An interrupt handler is a subroutine of a program that is called when the CPU receives an interrupt. The handler consists of code and functions to be executed based on what caused the interrupt. A function that always should be called in the handler is the clear function to clear or reset the interrupt flag so that the handler will not be executed over and over again.

Example 1.1. An example of an interrupt handler

```
void RTC_IRQHandler(void)
{
  /* Clear interrupt source */
  RTC_IntClear(RTC_IFC_COMP0);
  .
  . /* More code */
  .
}
```

1.4 Interrupt vector - how many handlers are available?

The interrupt vector contains the memory address of all the interrupt handlers available. When an interrupt request occurs, the processor saves the address where the program counter (PC) points to, and starts executing from the address in the interrupt vector given by the interrupt vector number.

The interrupt controller in the Cortex-M3 processor is the Nested Vector Interrupt Controller (NVIC).

1.5 Priority

The different interrupt requests are assigned different priority, so that a request with a higher priority will be handled before one with a lower. The Cortex-M3 processor provides up to 8 priority levels from 0 to 7, where 0 has the highest priority.

Example 1.2. Setting priority for the interrupt

```
NVIC_SetPriority(RTC_IRQn,0); /* The interrupt from the RTC gets the highest
                                  priority as the second input argument is 0.*/
```

1.6 Enable/disable interrupts

Interrupts can be maskable or non-maskable. Maskable interrupts are hardware interrupts that may be ignored by setting a bit in the Interrupt Mask Register (IMR). When enabling an interrupt, one tells the processor that this interrupt should not be ignored. By default these interrupts are turned off, and should be enabled if the interrupt is to be used in a program. A non-maskable interrupt is a hardware interrupt that does not have this bit-mask, and can thus never be ignored. An example of a non-maskable interrupt is the Hard Fault interrupt handler. This handler will be called when some serious error happens.

An interrupt lock is an instruction that disables all interrupts. This is done when the CPU performs an important instruction that should not be interrupted. This should not be done too often as it for instance can slow down timers that send an interrupt when the time should be updated.

Example 1.3. enabling a maskable interrupt

```
/* Enabling Interrupt from RTC. This function sets the
   enable bit in the Interrupt Mask Register */
NVIC_EnableIRQ(RTC_IRQn);
```

1.7 Push and pop registers on stack

If an interrupt occurs, the Program Counter (PC), that holds the address for the next instruction to be executed, has to be stored. The `PUSH` operator stores register on the stack. The `POP` operator loads register from the stack. The routine goes as follows:

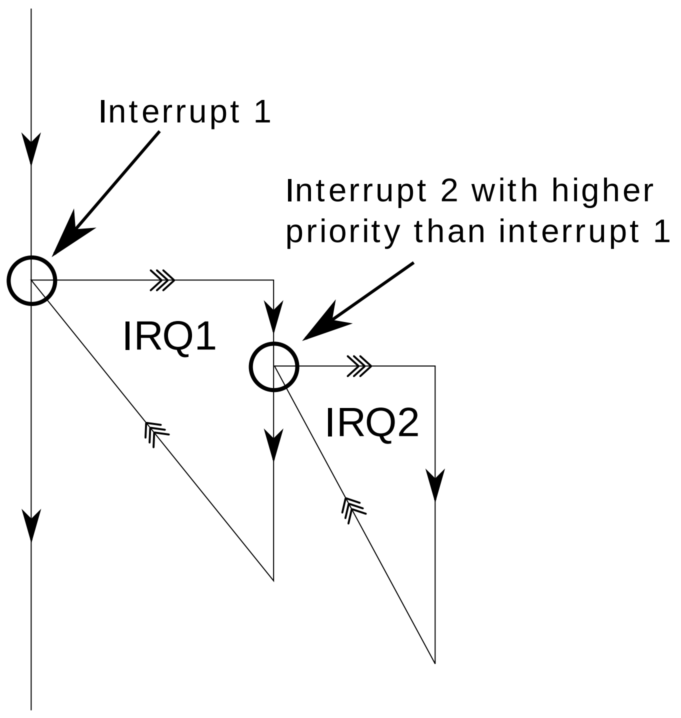
1. Get an interrupt and the associated vector number with the address to the interrupt handler.
2. Save current CPU information by pushing the PC and other registers on the stack.
3. Load new PC from the interrupt vector.
4. Execute Interrupt Service Routine (ISR).
5. Return from the interrupt handler and pop the PC and other registers from the stack.
6. Continue from where the interruption occurred.

The same procedure with pushing and popping registers is performed when doing regular function calls. This means that there is nothing special with the interrupt handlers - they are just like any other C function - something which is convenient for the programmer. Not all architectures support this, however, then it is up to the programmer to push and pop the things that need to be saved.

1.8 Nested interrupts

When an ISR is running, there can come a new interrupt with higher priority. Then the current ISR stops, its PC register is pushed on the stack as explained in Section 1.7 (p. 3) and the new ISR starts executing. This is called nested interrupts, and is illustrated in the following figure.

Figure 1.1. Showing the program flow of nested interrupts.



1.9 Interrupts vs. polling

Polling is the case where the CPU constantly checks for an event to happen in some hardware. This is synchronous communication with the hardware in contrast to the asynchronous communication when using interrupts. Interrupts is a better alternative than polling when such events happen rarely. This allows for reduced power consumption or freeing the CPU to do other operations.

Example 1.4. Interrupts vs. polling

In this example something is to be done when PB0 (Push Button 0) is pressed. This is done by calling the function `doSomething()`. PB0 is connected to pin B9, and this pin is configured such that its value can be read:

```
/* By setting the mode to gpioModeInput its value can be read */
GPIO_PinModeSet(gpioPortB, P, gpioModeInput, 1);
```

When using interrupts the following interrupt handler is made:

```
void GPIO_ODD_IRQHandler(void)
{
  /* Clear flag for Push Button 0 (pin B9) interrupt */
  GPIO_IntClear(0x0100);

  doSomething();

}
```

This handler is enabled and configured to give an interrupt on falling edge, i.e when button is being pressed:

```
/* Enable GPIO_ODD interrupt in NVIC */
NVIC_EnableIRQ(GPIO_ODD_IRQn);

/* Configure interrupt on falling edge for pins B9 (Push Button 0) */
GPIO_IntConfig(gpioPortB, 9, false, true, true);
```

Now the CPU can be put to sleep. When the button is being pushed, an interrupt will wake up the CPU and start executing the interrupt handler.

When using polling on the other hand, the CPU has to stay awake all the time executing the following loop:

```
while(1)
{
  while ( GPIO_PinInGet(gpioPortB, 9) ) ; /* returns 1 when the button is being pressed */

  doSomething();
}
```

2 Interrupt Code Examples

In this chapter we will look at three interrupt examples related to GPIO and Real Time Counter (RTC). The third example will include both GPIO and RTC interrupts with different level of priorities in order to create a simple stopwatch. In order to fully understand the examples some GPIO concepts must first be introduced.

2.1 GPIO background theory

2.1.1 GPIO pins

Some GPIO pins can be connected to modules such as the ADC, DAC or LCD, but all GPIO pins can be used to set the value of the pin or read its value. Pins can be set or read to be high (1) and low (0). High means that the voltage corresponds to the chips supply voltage (VDD), low means that it is connected to ground. To drive a LED, a GPIO pin can be set to high. Conversely, to see if a button is being pressed the value of the associated pin can be read.

2.1.2 Ports

Pins are collected into ports. Some operations, like writing to multiple pins, are more easily done when the involved pins are in the same port. A port has typically 16 pins and the ports are given the labels A,B,C,...

2.1.3 Pull ups and pull downs

To ensure that a GPIO pin is in the correct state, pull up and pull down are used. If a button is defined as not pressed when the pin is high, a pull up is used to keep it at VDD. When the button is pressed the pin is shorted to ground and the pin will read as low (0), see Figure 2.1 (p. 7) .

2.1.4 GPIO interrupt handlers

The number of GPIO pins on a MCU range from only a few to 50 and beyond. To keep the number of interrupt handlers manageable, several pins might share a handler. In the EFM32s there are two interrupt handlers for GPIO pins, one for pins that have an even pin index in its port and another for odd ones. Within the two interrupt handlers pins can be distinguished by reading what port pin index that has registered an interrupt. Up to 16 GPIO interrupts can be enabled, but only one per port index at the same time, i.e. A0 and B0 cannot both be enabled.

2.2 Push button interrupt example

In this first example we will look at how to drive a LED as long as a button is pushed. Two GPIO pins will be used, one for the LED and one for the button. The example is written for the Giant Gecko STK3700, and so the location of the pins might not be the same if you are using another kit. Each MCU has its own layout of the available pins and capabilities called a pinout. The pinout can be found in the MCUs datasheet. When an MCU is placed on a development board, some choices are made as to which pins should be connected to which features on the board such as push buttons, touch sliders and LEDs. Even though pins are often given meaningful names through defines in software it can be insightful to learn how to find the right pin and to use it directly. The pins of Push Button 0 and the LED can be found in the schematics chapter of the reference manual for the kit.

Figure 2.1. Push Button 0 and 1 are connected to UIF_PB0 and UIF_PB1 respectively.

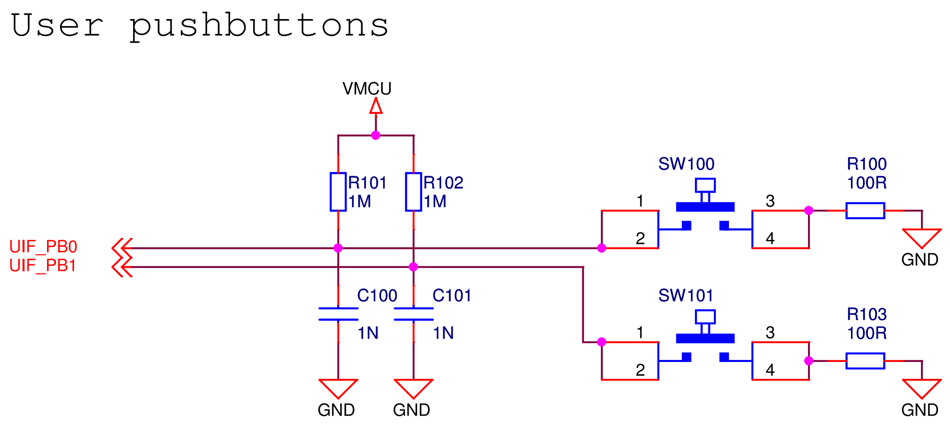
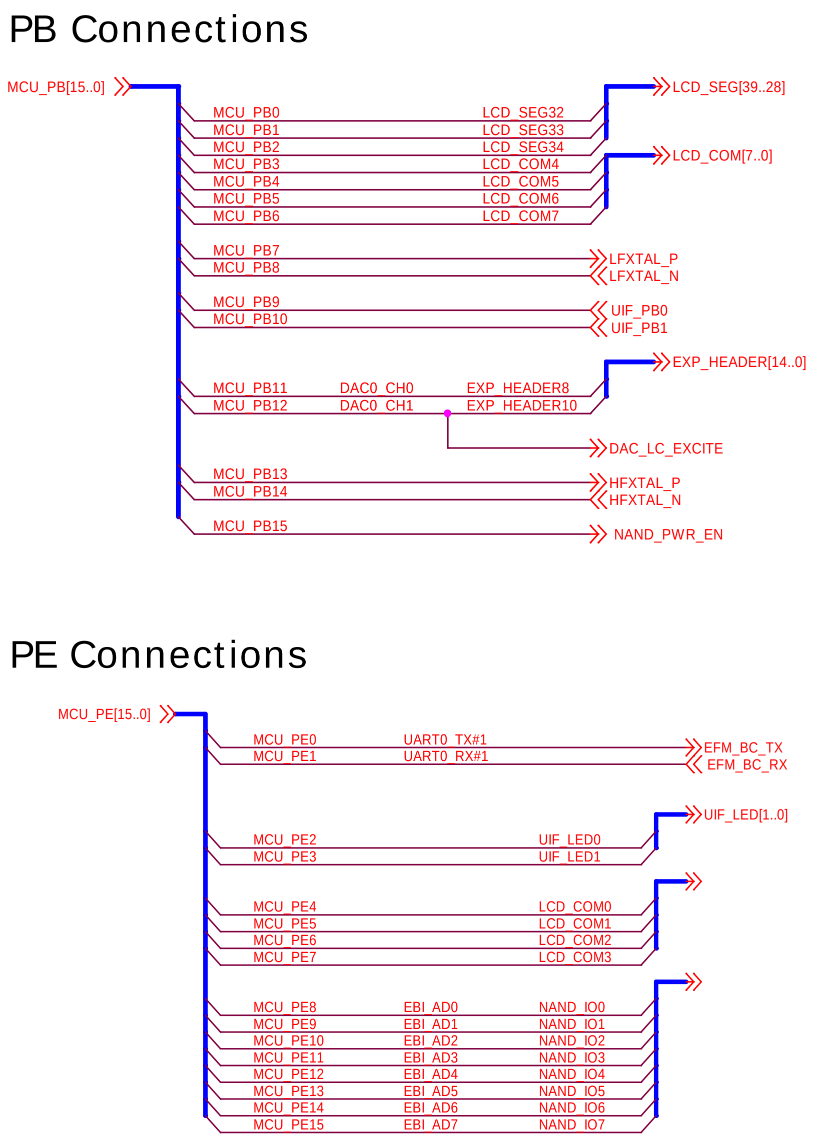


Figure 2.2. UIF_PB1, UIF_PB0 and the LED (UIF_LED0) are connected to PB11, PD8 and PD7 respectively.



The Push Buttons on the kit have external pull ups, so we do not have to set up our own. When the program is run, GPIO is configured before the chip enters Energy Mode 3 (EM3), to save power. When the interrupt is triggered from the Push Button (B9), its interrupt handle is called. The interrupt is triggered on both falling edge (button push) and rising edge (button release). A flag, indicating if the LED should be turned on or off, is toggled so that the LED is turned on when the button is pushed, and turned off when the button is released. Afterwards, the execution returns to `main` where it enters EM3 again.

2.3 Timer interrupt example

In this example the RTC is used as the interrupt source instead of Push Button 0. The RTC is configured to produce an interrupt every second. The LED is toggled on each interrupt.

2.4 Stopwatch example

In this example the Push Button 0 and 1 are used together with the RTC to implement a simple stopwatch. In order to illustrate how to use more than one interrupt source for each GPIO handler, a third pin is used: D3. When D3 is shorted to ground with e.g. a piece of wire (Be careful NOT to connect it to a higher voltage than what the chip can handle!) it will toggle the Gecko symbol on the LCD. The RTC interrupt handler is given the highest priority, while the even GPIO handler has higher priority than the odd GPIO handler. Since the RTC handler is keeping track of time, it is important that is executed immediately, even if the other handlers are running. To see the effect of different levels of priority a delay is added after the toggling of the Gecko. The RTC handler is not affected by this, but trying to pause using Push Button 1 will stall until the higher level interrupts, like the delay in GPIO even has finished. The time is written to the LCD with a software interrupt which has the lowest priority. The screen will therefore not be updated while the delay is happening, but once updated, it will show the correct time. If there is a critical section that has to run uninterrupted within a low priority handler, interrupts can be disabled with `__disable_irq()`. This will prevent interrupts to intervene. When the critical operation is finished, interrupts can be turned back on with `__enable_irq()`. Try to uncomment this code around the delay in the GPIO odd handler. This will make all interrupts stall when D3 is shorted - including the RTC handler, making the time incorrect.

3 Direct Memory Access (DMA)

3.1 What is DMA and why use it?

The Direct Memory Access controller is a hardware system that can transfer data, i.e reading and writing, independently of the CPU. Possible transfers might be RAM/Flash to a peripheral, Flash to RAM or Peripheral to RAM. The DMA frees the CPU to do other tasks or to sleep. When the transfer is completed, the DMA can wake up the CPU using an interrupt.

3.2 Channels

3.2.1 Priority

The EFM32 has 8 independent channels used to transfer data. The different channels can be configured with information like word count, word size and priority by writing to the configuration registers. The channels are numbered from 0 to 7, where 0 has the highest priority. Each channel also has a priority level; default or high. This means that channel 0 with priority level high has the highest possible priority. The second highest is channel 1 with priority level high, and the least prioritized is channel 7 with default priority level. A high priority level has to be set in the DMA_CHPRIS register.

3.2.2 Request line from peripherals

The DMA channel can be triggered from sources such as communication modules (USART, LEUART), timers (TIMER), analog modules (DAC, ADC) and software. The peripherals are connected to the DMA by request lines, and a request signal from the peripheral is sent through these lines. The channels have to be configured to which request lines they should be connected to. When a signal is triggered, the DMA performs the number of transfers specified by the channel descriptor.

3.2.3 Channel descriptor

The channel descriptor (channel control data structure) specifies the address to be read from, the address to write to, word length and the number of transfers/words sent in the same process. It can be configured when the controller arbitrates during an transition. To arbitrate means choosing the channel with a channel request that has the highest priority. It can be set how many transfers a channel performs before the controller rearbitrates, i.e. checks for the channel with the highest priority. The number of transfers is given by the number R which is set in the four R_power bits. The arbitration rate is then given by 2^R. Channels with low priority should not be assign large R-values, as this would give higher priority transfers longer latency.

Each channel can be assigned two configuration descriptors, one primary and one alternate, which are assigned different specifications. Sometimes it is convenient to switch between these two.

Example 3.1. Typedef of a descriptor struct and an example of an activation function.

```
typedef struct
{
  DMA_DataInc_TypeDef  dstInc; /* Destination increment size for each DMA transfer */

  DMA_DataInc_TypeDef  srcInc; /* Source increment size for each DMA transfer */

  DMA_DataSize_TypeDef size;   /* Word length */

  DMA_ArbiterConfig_TypeDef arbRate; /* Arbitration rate */

 uint8_t hprot; /* Signal state */
} DMA_CfgDescr_TypeDef;
```

After this struct is set up, the mode has to be set. If the mode is basic mode, the following function has to be used:

```
void DMA_ActivateBasic(
unsigned int channel, /* Channel number */
bool primary,         /* true if primary descriptor, false if alternate */
bool useBurst,        /* Only used on peripherals supporting DMA bursts */
void *dst,            /* Pointer to location to transfer data to */
void *src,            /* Pointer to location to transfer data from */
unsigned int nMinus1) /* Number of DMA transfer elements */
```

3.3 Transfer modes

The channels are also configured with transfer mode, specifying how to do the transfers.

3.3.1 Basic mode

In this mode the controller uses either the primary or the alternate descriptor. After the channel is enabled, the procedure goes as follows:

1. The controller performs 2^R transfers. If the number of transfers remaining is zero the flow continues at step 3.
2. The controller arbitrates:
   - If a higher-priority channel is requesting service then the controller services that channel.
   - If the peripheral or software signals a request to the controller then it continues at step 1.
3. The transfer is complete and the associated callback function is executed.

Notice that the controller needs a new request for each transfer it performs.

3.3.2 Auto-request

In this mode the controller only needs to receive a single request to enable the channel and to complete the entire DMA cycle. This enables large data transfers to occur without significantly increasing the latency for servicing higher priority requests. The controller is configured to use either the primary or alternate descriptor. When a channel, C, is enabled and the controller receives a request, the procedure goes as follows:

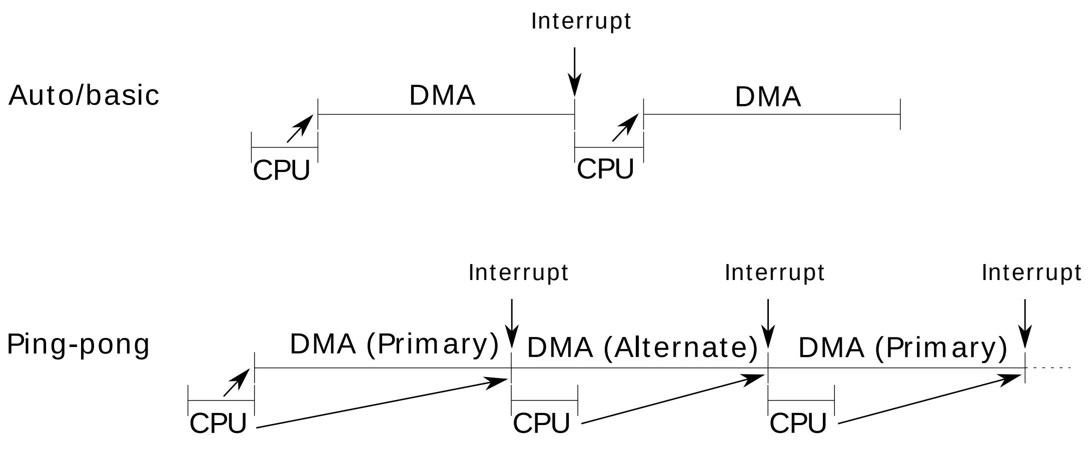
1. The controller performs 2^R transfers for channel C. If the number of transfers remaining is zero the flow continues at step 3.
2. The controller arbitrates. When channel C has the highest priority the DMA continues the transfers at step 1. Notice that the controller does not need to receive a new request.

3. The transfer is complete and a the associated callback function is executed.

3.3.3 Ping-pong mode

In this mode the channel switches between the primary and the alternate descriptor. The controller performs one DMA cycle using one of the descriptors, and then performs the next using the alternate. This is convenient because it enables the controller to immediately switch to a new DMA cycle after the first has completed. When receiving an interrupt, the CPU has to wake up and configure the descriptor. When using two descriptors two operations can be done at the same time as shown in the following figure:

Figure 3.1. Showing the difference between auto mode and ping-pong mode.



3.3.4 Scatter-gather mode

In scatter-gather mode, the controller uses the primary data structure to program the alternate data structure. After receiving a request, the controller performs four DMA transfers. These transfers write the alternate data structure for a task to be performed. The controller then generates an auto-request for the channel and then arbitrates. The controller performs the task using the alternate descriptor. After it completes the task, it generates an auto-request for the channel and then arbitrates. The controller continues to switch between the primary and the alternate descriptor until finished.

4 DMA Code Examples

In this chapter we will look at two DMA examples transferring data from Flash to RAM. The setup of the DMA channel and transfer descriptor are identical, but the DMA transfers are activated differently. The DMA control block must be aligned to a 256 byte boundary, which means that the last 8 bits of its address must be zero. Different compilers use different syntax to achieve this, but the example code cover the most common by use of preprocessor directives. For more examples please see Application Note 13.

4.1 Auto

The auto configuration is used to transfer data between Flash and RAM. The data is automatically being sent until the completion of the transfer. A callback function is then executed, which allow us to mark the completion and print the result to the LCD.

4.2 Basic

The basic configuration is mainly used to transfer between peripherals and RAM. After each arbitration, a request must be sent to continue the transfer. This is useful if you want to transfer N values from a peripheral to RAM, but only M values are ready at any given time. If the arbitration rate is set to M the DMA transfers M values before waiting for a new request. When the peripheral is ready it can send a a software request from e.g. an interrupt or without CPU intervention using the Peripheral Reflex System (PRS). PRS is explained in the next chapter. The DMA continues to transfer and wait for requests until completion, then the callback function is executed. To keep the example simple, data in transferred from FLASH to RAM as in the last example. A string containing "TESTING" is transferred one byte at a time. The next transfer request is issued from a timer interrupt. This interrupt is disabled by itself when the transfer has completed. The callback function turns on the antenna segment and prints the result of the transfer to the LCD.

5 Peripheral Reflex System (PRS)

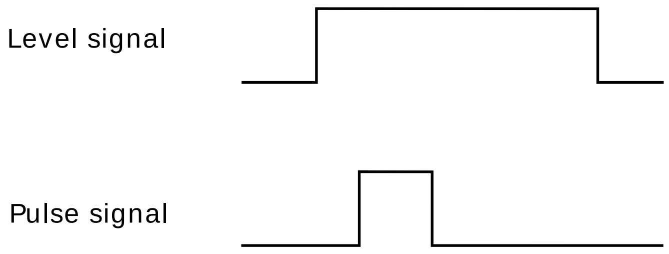
5.1 What is PRS and why use it?

The PRS is a network that allows the peripherals to communicate with each other without involving the CPU. The peripherals can send reflex signals to be used as triggers to other peripherals. The PRS is as interrupts a way to free the CPU to do other work or sleep which leads to reduced power consumption.

5.2 Reflex signals

The signals sent between two peripherals can be pulse signals or level signals. Furthermore, the signals can be synchronous or asynchronous. Synchronous reflexes are clocked and asynchronous reflexes are not. In that way asynchronous signals can be used in sleeping modes, though to receive these signals are restricted to only a subset of the peripherals.

Figure 5.1. Showing an example of a level signal and a pulse signal. The level signal has to be turned on to get to its high level and stays there until it is turned off returning to its low level. The pulse signal comes as a single pulse, and does not need to be turned on or off.



5.3 Producers, consumers and channels

Peripherals that produce the reflex signals are called producers, and the ones receiving them are called consumers. The PRS for EFM32GG consists of 12 channels for transmitting signals, where each channel can be connected to any producer. In that way the consumer can choose which channel to listen to. Some consumers are able to receive level signals, but most of them expect pulse signals. To solve this the channels include an edge detector to enable generation of pulse signals from level signals. The consumers can perform actions based on the signal it is receiving from the producer.

The channels can be configured using the following emlib function

```
void PRS_SourceSignalSet(unsigned int ch,
uint32_t source, uint32_t signal, PRS_Edge_TypeDef edge)
```

The function specifies the PRS channel, the peripheral that produces the signal, signal and edge for pulse generation. It is also possible to generate output reflex signals by software writing to PRS_SWPULSE and PRS_SWLEVEL registers producing respectively pulse and level signals.

6 PRS Code Example

In this chapter we will look at how to start and stop a timer using PRS and a Push Button.

6.1 PRS and timer example

Pin B9 (Push Button 0) is configured to send a PRS signal in channel 5 when the button is pushed (on falling edge). TIMER0 is configured to reset the count and start counting at falling edge in PRS channel 5. On rising edge in channel 5, i.e. when Push Button 0 is released, TIMER0 will stop the count and store the result to CC0. A TIMER0 interrupt is then produced, which displays the count on the LCD.

7 Summary

This lesson has introduced three important concepts that frees the CPU from performing certain tasks: interrupts, DMA and PRS. Interrupts allow the CPU to do other work or sleep while waiting for an event. The CPU then executes the interrupt handler for the given interrupt. DMA transfers data between peripherals and RAM without CPU intervention. PRS enable peripherals to send each other signals, so that a sequence of tasks can be done without the help of the CPU.

8 Exercises

Frameworks and solutions to the exercises below are provided together with the other example code.

8.1 Exercise 1: Toggle LED

Create a program that toggles the LED when both buttons on the kit are pressed.

8.2 Exercise 2: Countdown Timer

Create a program that count down from a given value when both buttons are pressed on the kit. If only one button is pressed the countdown time should be increased or decreased.

9 Revision History

9.1 Revision 1.00

2011-06-22

Initial revision.

9.2 Revision 1.10

2012-07-27

Updated for Giant Gecko STK.

A Disclaimer and Trademarks

A.1 Disclaimer

A.2 Trademark Information

B Contact Information

B.1 Energy Micro Corporate Headquarters

Postal Address	Visitor Address	Technical Support
Energy Micro AS P.O. Box 4633 Nydalen N-0405 Oslo NORWAY	Energy Micro AS Sandakerveien 118 N-0484 Oslo NORWAY	support.energymicro.com Phone: +47 40 10 03 01

www.energymicro.com
Phone: +47 23 00 98 00
Fax: + 47 23 00 98 01

B.2 Global Contacts

Visit **www.energymicro.com** for information on global distributors and representatives or contact **sales@energymicro.com** for additional information.

Americas	Europe, Middle East and Africa	Asia and Pacific
www.energymicro.com/americas	www.energymicro.com/emea	www.energymicro.com/asia

Table of Contents

List of Figures

List of Examples

Energy Micro AS
Sandakerveien 118
P.O. Box 4633 Nydalen
N-0405 Oslo
Norway

www.energymicro.com