Multi_RxIsr_Example

1.0.0

Features

- Support of up to 11 UART Rx PSoC ports with efficient ISR data processing with no loss of data.
- Rx ISR data acquisition is using a multi-byte circular FIFO buffer.
- 1 Tx stimulus port.
- Rx data processing includes the following detections:
  - Tx packet mismatch
  - Rx circular buffer overrun.
- Simple Debug Tx port for Terminal report of errors

General description

This example program shows how to process multiple UART Rx data inputs efficiently and therefore very quickly using Interrupt Service Routines (ISRs).

The goal of this example is to demonstrate “proper” SW coding where ISR code does not use blocking functions. The use of blocking functions in ISRs would potentially cause the overall system performance of Rx data capture to be significantly limited.

In ***Appendix A*** – Treatise on Efficient ISRs, I provide a historical and practical treatise on interrupts. It provides the reasoning for the coding style of this example. I encourage reading this treatise to better understand the choices made in this example project as well as choices you make in future projects.

Once you get this example to work “as is”, I encourage modifying it to serve your specific needs.

The SW was designed to be generic to be able to run virtually on any platform or PSoC HW with minimal SW or HW adaptions.

The SW principles used in this project can be used on any non-PSoC platform. However more modifications to the HW or SW may be needed.

Device Families Supported

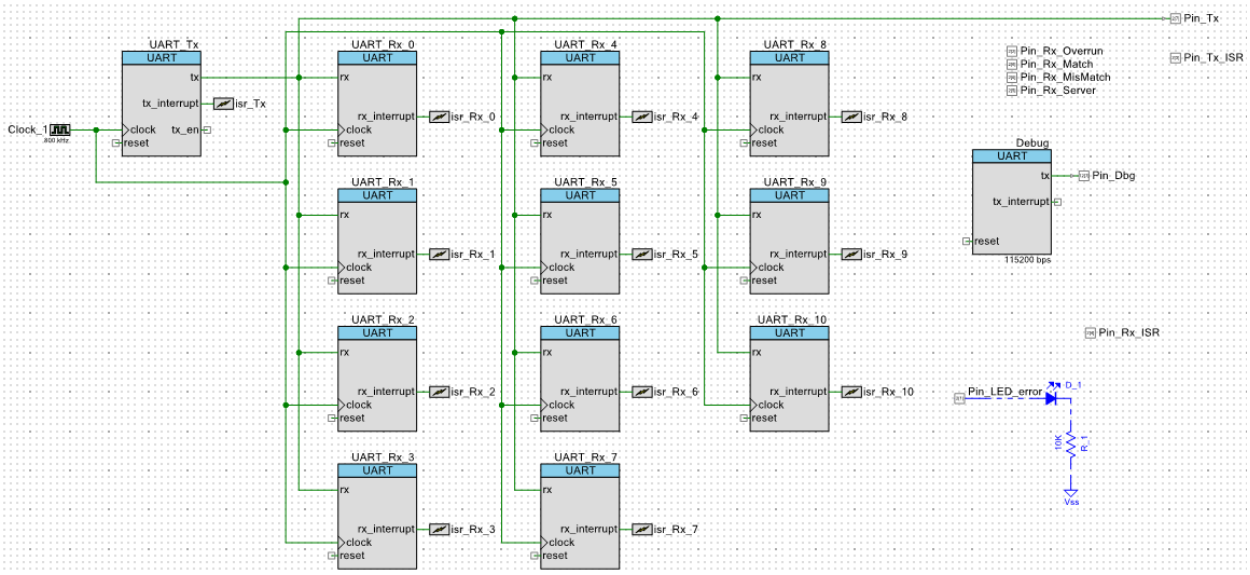
At this time, the following are the PSoC devices that support this component:

- PSoC5. In this implementation, all UART Rx inputs are configured to be connected to the Tx stimulus without need of GPIO pins. This makes it so that external connections are not needed.
- Other PSoC platforms may need minor SW modifications and may only support less than 11 UART Rx ports due to chip resource limitations.

Functional Description

This project demonstrates efficient processing of data via ISR using a circular buffering technique to minimize possible buffer overruns.

I provide one UART Tx to provide the stimulus data for the 11 UART Rx ports. The output of the Tx is directly routed to ALL the Rx ports internally to the PSoC to prevent external connections and eliminate line-induced noise.



The Tx stimulus data transmits four 7-byte message packets of different data. The continuous transmission of data occurs every 4 bytes (to fill the Tx FIFO) by ISR.

The Rx data is collected into the circular buffer assigned to each port via each port's ISR. When at least one message is received in the buffer, the ISR signals the main task to further process the data as a complete message.

The main() task (called Rx_Server_vTask()) detects a message is available for a specific port and processes the message. In this example, the UART Rx port is checked for a buffer overrun. If "no

overrun", then the message is compared to the 'known' Tx stimulus message data. If ANY of the received data does not match the Tx stimulus messages, a "mis-match" is detected.

With the additional UART Tx port called "Debug", I push out diagnostic data which includes whether a mis-match ('#') or a "buffer overrun" ('1') occurred and which port where it occurred.

A diagnostic LED is turned on if ANY mismatch or buffer overrun occurred for any UART Rx port. This is a visual indicator of an error without using a Debug output on a host terminal.

Circular Buffering (with Threshold Detection)

The Rx data is accumulated into a SW circular FIFO buffer using the port's UART Rx ISR.

The principle of a good circular buffer uses a 'head' index and a 'tail' index.

Head Index

The head index is used by the ISR as a displacement in the RAM array allocated for the port's buffer to load the incoming data. The head index controls the 'FI' of FIFO.

After the Rx data is placed into the circular buffer at the current head index, it increments the head index. When the head index exceeds the buffer size, it resets to '0' index (hence the 'circular' definition).

Tail Index

The tail index indicates the next displacement in the Rx buffer to read the oldest data by the application. The tail index controls the 'FO' of FIFO.

Once the data is read by the application, the tail index is incremented to the next position in the FIFO. When the tail index exceeds the buffer size, it resets to '0' index.

Threshold Detection

In this circular buffer implementation, I've added **two** index threshold detections:

- Message Size – This detects that the number of bytes currently in the FIFO equals or exceeds the message size to be processed. This is done by comparing the head to the tail indexes.

  Once detected, it sets a "RX_MSG_RCVD" signal for the Rx_Server_vTask() to start the processing of a complete message.
- Buffer Overrun – This detects if the application is not processing the input data fast enough from the FIFO. When this happens, the latest input data has overwritten the oldest data in the FIFO before the application can get to it. This condition is detected when the head index equals the tail index. Since the head index is advanced AFTER writing the data to the buffer, the oldest Rx data not processed is overwritten.

  Once detected, it sets a "RX_BUFFER_OVERRUN" signal for the Rx_Server_vTask() to know if the application can trust the oldest message.

Increasing the Rx FIFO buffer size can sometimes prevent buffer overruns (but not always depending on other system design considerations).

Project Performance Measurements

I have gathered performance data using this example project on a PSoC5LP with a CY8CKIT-059 board. Your results on other PSoC5 boards, PSoC5 devices and PSoCs will differ but the results should scale according.

The factors I have found to contribute to the performance with my circular FIFO buffer implementation are:

- BUS_CLK
- FIFO buffer depth (Note: larger is not necessarily better)

The table below shows the BUS_CLK frequency, the maximum UART Baudrate and the minimum Rx FIFO depth. I've included a nominal Tx ISR to transmit 4 bytes and Rx ISR time to receive 1 byte. Note: Less time is better for performance.

BUS_CLK	Rx baud (max)	Rx FIFO size (min)	TX ISR time	RX ISR time
79.5 MHz	292.28 KBaud	16 bytes	4 us	1.4 us
48 MHz	193.55 KBaud	11 bytes	4.9 us	2.5 us
24 MHz	100.00 KBaud	15 bytes	9.5 us	5 us
12 MHz	51.72 KBaud	16 bytes	19 us	2.3 us

The performance results you receive may vary based on your application needs.

For example, if you only have 4 Rx ports, your maximum Rx baud will be higher. If your data processing is more time intensive, you may not be able to process your circular FIFO fast enough. You may need to increase the Rx FIFO size.

Appendix A – Treatise on Efficient ISRs

The History of Interrupts (and ISRs)

When the earliest computers first appeared, they were effectively calculators with some conditional testing. For example, some of these computers were used to calculate trajectories of firing mortars at a target. (If it weren't for war, computers would have been invented much later.)

In this case, the influencing factors of mortar weight, mortar explosive amount and distance to target were the inputs and the resultant firing angle of the armament would be its output.

In this type of computing, all the known factors were input and the computer would 'batch' compute.

In later more complex computers, computations would calculate multiple results on the known input data. The results would be either printed for a human to further process or compared for the optimum results. An example of this type of computer was the ones used to decode cryptography. (Another example of a war-time application.)

In effect, this type of computing is mostly "open-loop". At that time, interrupts were not envisioned.

As the computer become cheaper (and smaller), people wanted to use it for more mundane purposes such as control of a building's Heating or Cooling mechanisms. To do so, they needed to implement "closed-loop" SW where the inputs for each processing cycle of the application were regularly read to compute a potentially new result. They did this by reading a temperature sensor frequently. This is a polling technique in the application.

However as the need for the most current input data came at a higher frequency, many of these early computers could not keep up using polling. Therefore the HW interrupt was conceived and implemented.

The Interrupt Concept

Although polling a signal at the application level did not require HW changes in the processor, it had a practical limit in data acquisition speed.

The Interrupt was designed to 'interrupt' the processor at the moment of the 'event'. It preserved the processor state such as the registers (using stack memory), processed the 'event' and then returned the processor state prior to the interrupt.

The interrupt was the earliest implementation of what we call Real-Time Operating System (RTOS) "multi-tasking" today. This multi-tasking was simple and lacked certain features we have come to appreciate with a well-implemented RTOS.

The Modern Use of Interrupts

I know of NO modern-day processor or microprocessor that doesn't have interrupt capability. It is a system resource crucial to servicing time-critical events in "closed-looped" systems whether single-tasked or using an RTOS SW architecture.

There are SW coding techniques that should be preferred when writing ISRs to prevent system errors and maximize system performance. I will only cover some of the basics here. There are other "dos" and "don'ts" I'm not covering to limit the size of this document. I suggest you consult other sources of "good ISR SW coding".

"Get in and get out QUICKLY"

There a number of tips I learned from a seasoned expert in the field of SW coding of ISR. Write your ISR code to:

- Quickly determine what caused the interrupt.
- If input data is the reason for the interrupt, get it and store it for the application.
- Do not use functions that 'block' the return waiting for something to potentially happen.
- Use while/do () statements ONLY if you know it will not block.
- If possible, try to avoid using recursive function calls. This could cause a stack overflow issue.
- AVOID, final processing of the data if at all possible in the ISR. The notable exception is if you are sure final processing can be very quick. For example: if you can set a GPIO port pin quickly, this should be acceptable.

In the above cases, defer all meaningful processing of the event or event data for the application. If needed, set a signal in the ISR that can be used by the application to begin data processing.

The overall goal is to spend the minimum amount of time in the ISR and to return to the application level as quickly as possible.

The next topics will elaborate why this is a good policy.

Simple Interrupts

The interrupt is designed to temporarily deviate from the application level. This means that any "real-time" outputs derived at the application level may be delayed and may have negative consequences.

Also, if a single interrupt is taking too long to return to the application level, other events causing interrupts may not be processed in time to acquire the data these interrupts are trying to provide. This is because the simple interrupt is "unnested". This means ONLY ONE interrupt can be processed at a time and will not be processed until the processor is back to the application level. This is because in an ISR the Global interrupt enable flag in the CPU is disabled and won't be re-enabled until this ISR is completed.

Nested Interrupts

Many processors allow for "nested" interrupts. In this architecture, if done properly, another interrupt can be allowed to interrupt an ISR being currently processed.

There are two types of "nested" interrupts: non-prioritized and prioritized.

Virtually all modern processors allow for non-prioritized nesting. This occurs when you are in an ISR, the code can select to re-enable the Global interrupt. This allows another interrupt to run its' ISR on top of the previous ISR. Once the latest ISR is complete, it returns to the previous level. This continues until back at the application level.

The second nesting type depends if the processor provides a priority level. This level can be 'hard-coded' in the architecture or 'soft-coded' in registers. In this type of nesting, if enabled, only an interrupt with a higher priority can interrupt the ISR of lower priority. As with the non-priority nesting, returning from an ISR will begin to unravel the nesting all the way down to the application level.

Nesting Caveats

Although it would appear that "nesting" is a great solution to spending more time in an interrupt, avoid that temptation.

Every time an interrupt is executed with an ISR, stack memory is being used. This also includes stack consumed if making function calls within the ISR. If you process nested interrupts, make sure you allocate enough stack to prevent a stack overflow. Not doing so, could corrupt your application.

Also, analyze your design for the worst-case nesting condition. Make sure you don't have recursive ISRs.

PSoC Interrupt Architecture

All the Cypress PSoC architectures have the ability to use all forms of interrupt types. Simple and Nested (Non-priority and Priority). My advice is to always defer to the Simple-type for ISR coding. Use nested types wisely only if event causing the interrupt MUST be processed ASAP.

"Final Notes"

Since the interrupt ISR is not actually a "true" or "well-behaved" task in an RTOS, there are additional precautions when coding the ISR. There are potential issues when sharing global variables that should be taken into account. These "dos" and "don'ts" are not covered here.

Project Changes

Version	Description of changes	Reason for changes/impact
1.0.0	first release of the component	

References

Guru's guide to the Commodore Amiga - Meditation #1 – Interrupts [C Sassenrath]	