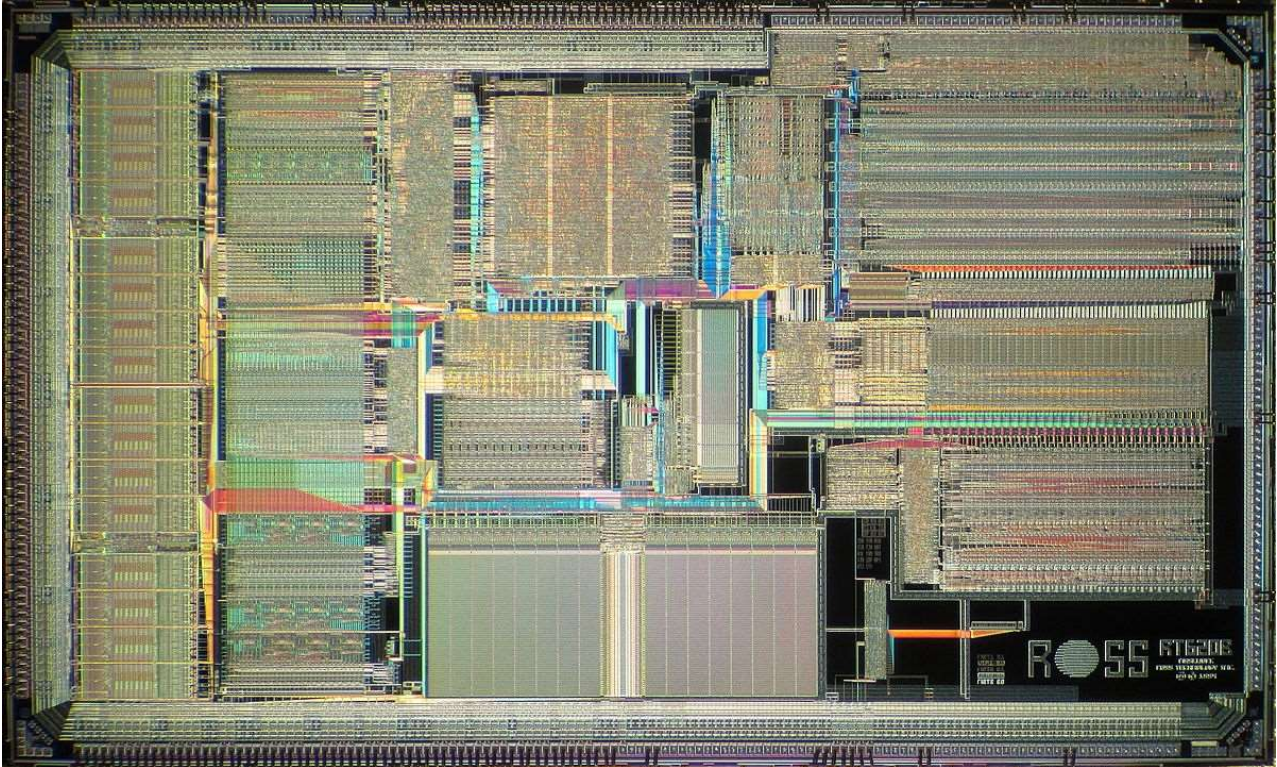


Digitale Systeme und Computersysteme

Kompetenzbereich Computerarchitekturen



Quelle: https://commons.wikimedia.org/wiki/File:Ross_hyperSPARC_RT620B_CPU_die.JPG

Auszug Lehrplan

Die Schülerinnen und Schüler können im Bereich Computerarchitekturen...

... im III. Jahrgang, 5. Semester

das Prinzip einer Mikrocontrollerarchitektur und die wesentlichen Schritte der Befehlsausführung erklären.

... im III. Jahrgang, 6. Semester

das Programmiermodell eines Mikrocontrollers erklären und einfache Programme entwickeln.

... im IV. Jahrgang, 7. Semester

die Funktionsweise von I/O-Komponenten erklären.

... im IV. Jahrgang, 8. Semester

das Hardware-Software-Interface von Standardschnittstellen erklären.

Lehrstoff:

- Peripheriekomponenten

1 I/O-Komponenten

Ein-/Ausgabekomponenten (daher auch die engl. Bezeichnung I/O für Input/Output) dienen als Bindeglied zwischen einem Mikroprozessorsystem und seiner Umwelt.

I/O-Komponenten sind über den "Component Catalog" auswählbar; auch der Zugriff auf Demobeispiele oder Datenblätter gestaltet sich sehr einfach.

Bei dem in der HTBLuVA Anichstraße eingesetzten PSoC wird zwischen drei Arten von Ein-/Ausgabeanschlüssen unterschieden: general purpose I/O (GPIO), speziellen I/O (SIO) sowie für USB (USBIO).

Auszug aus dem PSoC5LP Architecture Technical Reference Manual:

All I/O pins are available for use as digital inputs and outputs for both the CPU and digital peripherals. In addition, all I/O pins can generate an interrupt. All GPIO pins can be used for analog input, CapSense®, and LCD segment drive, while SIO pins are used for voltages in excess of V_{dda} and for programmable output voltages and input thresholds. [...]

The PSoC I/O system has these features, depending on the pin type.

Supported by both GPIO and SIO pins:

- *User programmable I/O state and drive mode on device reset*
- *Flexible drive modes*
- *Support level and edge interrupts on pin basis*
- *Slew rate control*
- *Supports CMOS and low voltage TTL input thresholds*
- *Separate port read and write registers*
- *Separate I/O supplies and voltages for up to four groups of I/O*

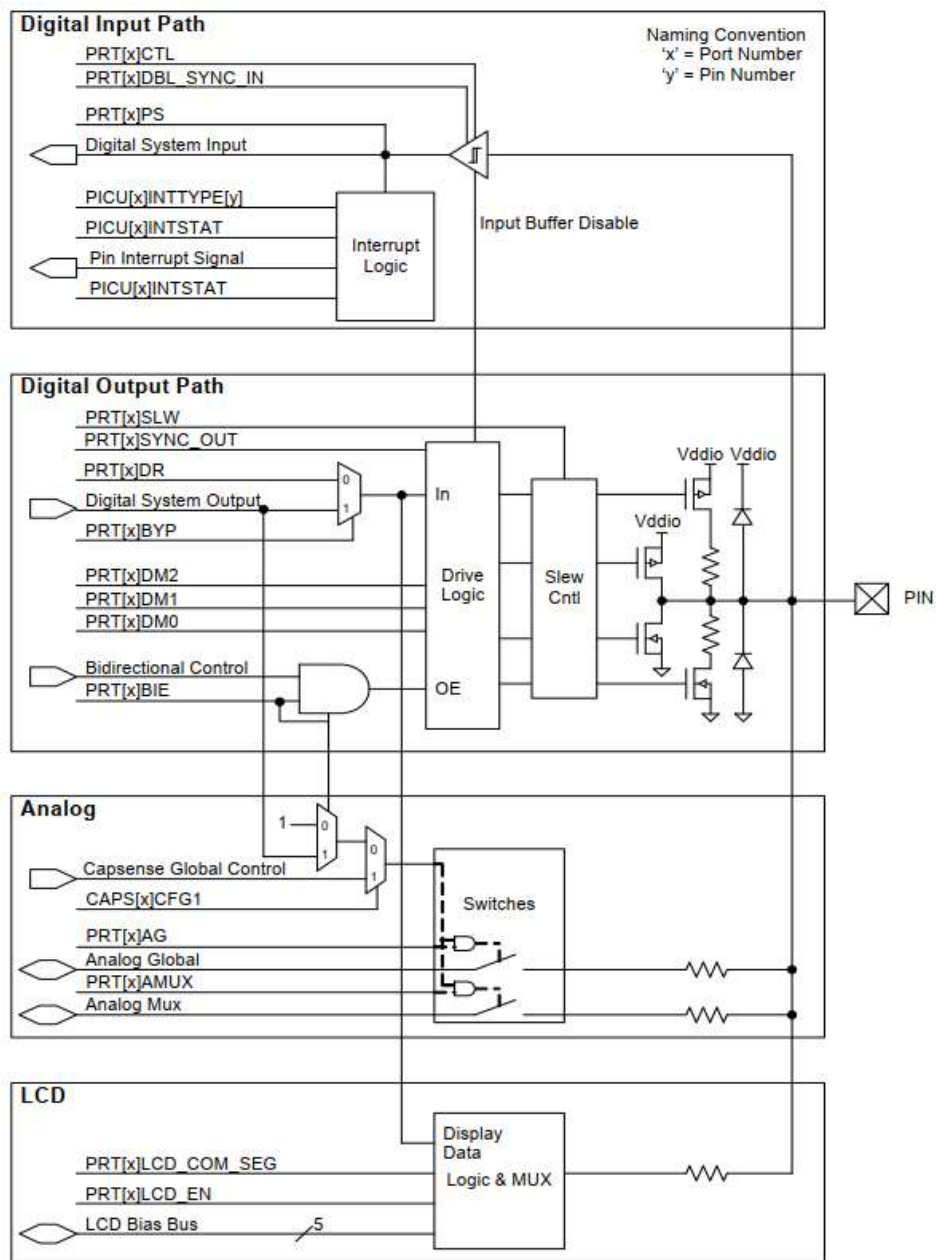
Provided only on the GPIO pins:

- *Supports LCD drive*
- *Supports CapSense*
- *Supports JTAG interface*
- *Analog input and output capability*
- *8 mA sink and 4 mA source current*
- *Ports can be configured to support EMIF address and data*

Provided only on SIO pins:

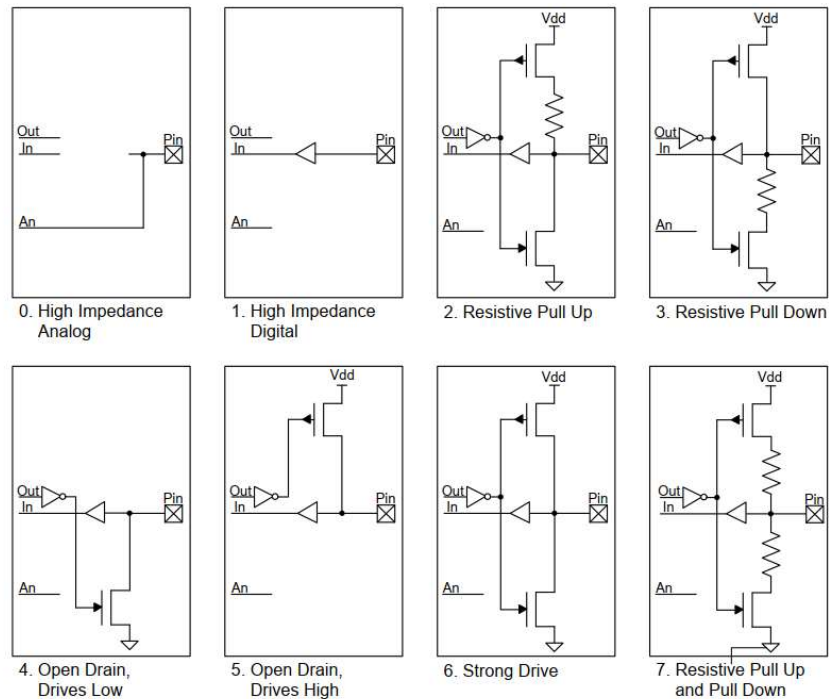
- *Hot swap capability (5 V tolerance at any operating V_{dd})*
- *Single enable and differential input with programmable threshold*
- *Regulated output voltage level option*
- *Overvoltage tolerance up to 5.5 V*
- *Higher drive strength than GPIO*
- *25 mA sink and 4 mA source current*

Blockdiagramm eines GPIO-Anschlusses:



Quelle: PSoC 5LP Architecture TRM, Document No. 001-78426 Rev. *C

Jeder GPIO und SIO Pin kann in einem von 8 Modi (drive mode) betrieben werden:



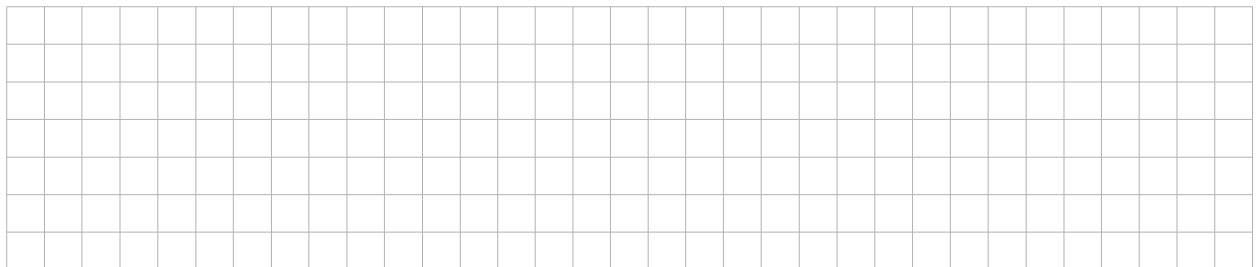
The 'Out' connection is driven from either the Digital System (when the Digital Output terminal is connected) or the Data Register (when HW connection is disabled).
 The 'In' connection drives the Pin State register, and the Digital System if the Digital Input terminal is enabled and connected.
 The 'An' connection connects to the Analog System.

Quelle: PSoC 5LP Architecture TRM, Document No. 001-78426 Rev. *C

Die Grundbetriebsart (factory drive mode default) ist 0 (**High Impedance Analog**) – kann jedoch geändert werden. Wie in der oben dargestellten Abbildung ersichtlich gibt es hier nur die Verbindungen zum Analogen Subsystem; versucht man über die digitale Domäne einen lesenden Zugriff, so erhält man stets den Rückgabewert 0. High Impedance Analog ist auch die Grundbetriebsart während eines Systemresets; d.h. für die Dauer des Systemresets sind sowohl der Eingangsbuffer als auch der Ausgangstreiber deaktiviert.

High Impedance Digital ist die empfohlene Betriebsart für einen digitalen Eingang.

Bei **Resistive Pull Up** bzw. **Resistive Pull Down** wird ein interner Widerstand (im Bereich von $3.5\text{k}\Omega$... $8.5\text{k}\Omega$; typisch $5.6\text{k}\Omega$) entweder gegen die positive Versorgungsspannung oder Masse geschaltet; der jeweils andere Signalzweig wird "strong drive" betrieben. Diese Pins können sowohl als Ein- als auch als Ausgang betrieben werden. Typischer Anwendungsfall ist der Anschluß eines Tasters / Schalters:



Bei **Open Drain, Drives Low** bzw. **Open Drain, Drives High** fehlen die internen Widerstände; d.h. wird der Pin als Ausgang betrieben, dann ist der Ausgang entweder hochohmig (High-Z) oder wird niederohmig mit GND bzw. Versorgungsspannung verbunden (strong drive). Typischer Anwendungsfall von Open Drain, Drives Low ist die Anbindung an einen I²C-Bus.

Wird ein "klassischer" digitaler Ausgangspin benötigt, der sowohl die Versorgungsspannung als auch Masse auf den Pin schalten kann, so ist **Strong Drive** zu verwenden.

Bei **Resistive Pull Up and Pull Down** gibt es keinen strong drive; die Ausgabe eines HIGH erfolgt über den internen Pull-Up Widerstand, die Ausgabe eines LOW über den internen Pull-Down Widerstand. Dieser Modus ist zu verwenden, wenn ausgangsseitig Kurzschlüsse auftreten können.

Die Konfiguration des Anschlusses (d.h. Auswahl der Betriebsart, Slew Rate Einstellung, ...) kann komfortabel und überwiegend graphisch über die "Pin Komponente" erfolgen.

Auszüge aus dem Pins Component - Datenblatt von Infineon (ehemals Cypress):

Use the Pins Component when a design must generate or access an off-device signal through a physical IO pin. Pins are the most commonly used Component in the Component Catalog. For example, they are used to interface with potentiometers, buttons, LEDs, and peripheral sensors such as proximity detectors and accelerometers.

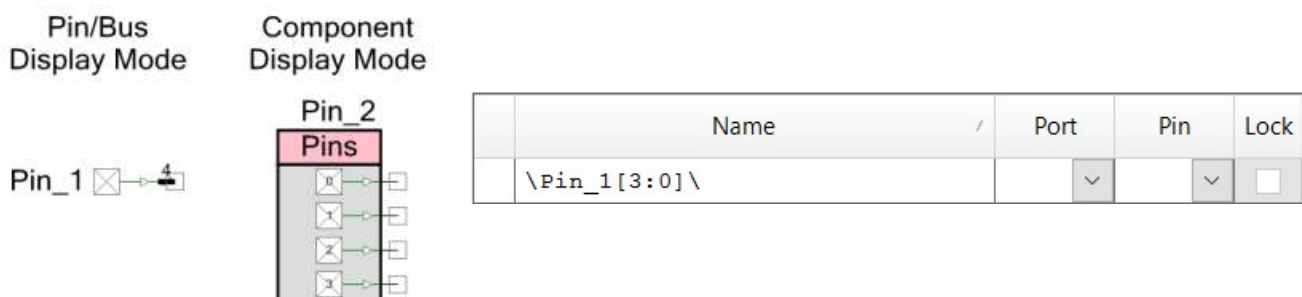
The Pins Component allows hardware resources to connect to a physical port-pin. It provides access to external signals through an appropriately configured physical IO pin. It also allows electrical characteristics (e.g., Drive Mode) to be chosen for one or more pins; these characteristics are then used by PSoC Creator to automatically place and route the signals within the Component.

Pins can be used with schematic wire connections, software, or both. To access a Pins Component from Component Application Programming Interfaces (APIs), the Component must be contiguous and non-spanning. This ensures that the pins are guaranteed to be mapped into a single physical port. Pins Components that span ports or are not contiguous can only be accessed from a schematic or with the global per-pin APIs.

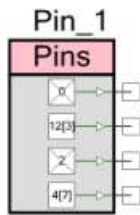
A Pins Component can be configured into many combinations of types. For convenience, the Component Catalog provides four preconfigured Pins Components: Analog, Digital Bidirectional, Digital Input, and Digital Output.

Pin-Grundkonfigurationen

Pins can be configured into complex combinations of digital input, digital output, digital bidirectional, and analog. Simple configurations are generally shown as single pins. More complex types of pins are shown as standard Components with a bounding box.



When a Pins Component is assigned to a physical General Purpose IO (GPIO) or Special IO (SIO) pin using the PSoC Creator Design-Wide Resources Pin Editor, the tooltip for the Pins Component shows the specific pin assignments. If a pin assignment is locked, the display of the Component indicates the assignment, as shown in the following example:



Note: If the Pins Component is set to Display as Bus, the display of the Component does not display any locked pin assignments; however, the tooltip still displays this information.

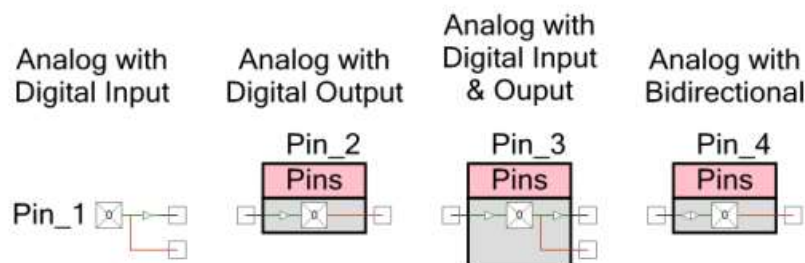
Software pins do not show any terminals as they can only be driven via the CPU/DMA. In order to read from the pin, the input buffer parameter should be enabled. The sync parameter has no effect on software pins.

Pin_1 

Configure the Pins Component as **Analog** any time the design requires a connection between a device pin and an internal analog terminal connected with an analog wire. When configured as analog, the terminal is shown on the right side of the symbol with the connection drawn in the color of an analog wire.

Pin_1  

An analog Pins Component may also support digital input or output connections, or both, as well as bidirectional connections. It is possible to short together digital output and analog signals on the same pin. This can be useful in some applications, but is not a general use case, and should be used with care.



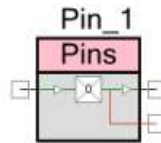
Configure a Pins Component as **digital input** any time your design requires a connection between a device pin and an internal digital input terminal, or if the pin's state is read by the CPU/DMA. In all cases using digital-input pins, the pin state is readable by the CPU/DMA. Additionally, if the schematic terminal (HW Connection) is displayed it can be routed to other Components in the schematic.

When visible, the terminal is shown on the right side of the symbol. The connection is drawn in the color of a digital wire with a small input buffer to show signal direction.

Pin_1  

A digital-input Pins Component may also support digital output and analog connections.

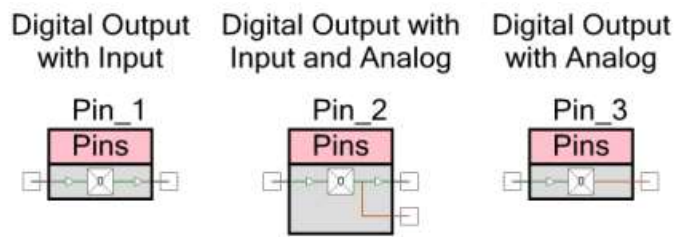
Digital Input with Output and Analog



Configure a Pins Component as **digital output** any time a device pin is to be driven to a logic high or low. In all such cases, the pin state is writable by the CPU/DMA. Additionally, if the terminal is displayed it can be routed from other Components in the schematic. When visible, the terminal is shown on the left side of the symbol. The connection is drawn in the color of a digital wire with a small output buffer to show signal direction.



A digital-output Pins Component may also support digital input and analog connections.

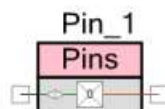


Configure a Pins Component as digital **bidirectional** any time your design requires a connection between a device pin and an internal digital bidirectional terminal. Digital bidirectional mode is most often used with a communication Component like I2C. When configured as digital bidirectional, the terminal is shown on the left side of the symbol with the connection drawn in the color of a digital wire with input and output buffers showing that the signal is bidirectional.



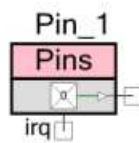
A bidirectional Pins Component may also support analog connections.

Digital Bidirectional with Analog

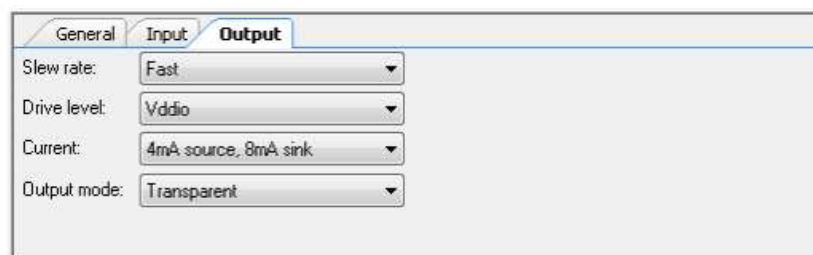


To configure a Pins Component with a **port-dedicated interrupt**, you must use a digital input and configure the Interrupt parameter. You must also select the check box that asks whether to use the dedicated port interrupt. When interrupts are used, the Pins Component displays with a bounding box, and the IRQ is displayed extending from the bottom of the Component. The typical use case is to connect an Interrupt Component to this terminal. This will allow the pin to use the dedicated

Port Interrupt to trigger its interrupts.



The **Output subtab** specifies output settings. If the Type is not "Digital Output" or "Bidirectional," this tab is disabled because you do not need to specify output information.



The slew rate parameter determines the rise and fall ramp rate for the pin as it changes output logic levels.

Drive Level selects the output drive voltage supply sourced by the pin. All pins can supply their respective VDDIO supply voltages. SIO pins can also supply a programmable or analog routed voltage for interface with devices at a different potential than the SIOs VDDIO voltage.

The drive current selection determines the maximum nominal logic level current required for a specific pin. Pins can supply more current at the cost of logic level compliance or can have a maximum value that is less than listed, based on system voltages. See the device datasheet for more details on drive currents.

- 4mA source, 8mA sink – Default
- 4mA source, 25mA sink – Requires SIO

Output synchronization reduces pin-to-pin output signal skew in high-speed signals requiring minimal signal skew. By default, this parameter is set to "Transparent" and no synchronization occurs. If "Single-Sync" is selected, the output signal is synchronized to the output clock.

Das CPU-Subsystem hat über das **API** Zugriff auf die Pin-Register und kann auf diese lesend und schreibend zugreifen. Damit ist es auch möglich, während der Laufzeit die Pineinstellungen zu ändern. Das Pin-API stellt dafür folgende Aufrufe zur Verfügung:

- `uint8 Pin_Read(void)`
Reads the associated physical port (pin status register) and masks the required bits according to the width and bit position of the component instance.
- `void Pin_Write(uint8 value)`
Writes the value to the physical port (data output register), masking and shifting the bits appropriately.
- `uint8 Pin_ReadDataReg(void)`
Reads the associated physical port's data output register and masks the correct bits according to the width and bit position of the component instance.

- `void Pin_SetDriveMode(uint8 mode)`

Sets the drive mode for each of the Pins component's pins.

- `void Pin_SetInterruptMode(uint16 position, uint16 mode)`

Configures the interrupt mode for each of the Pins component's pins. Alternatively you may set the interrupt mode for all the pins specified in the Pins component.

- `uint8 Pin_ClearInterrupt(void)`

Clears any active interrupts attached with the component and returns the value of the interrupt status register allowing determination of which pins generated an interrupt event.

2 Interrupts

Ein Auftreten einer **Ausnahmesituation (engl. Exception)** kann bewirken, dass die reguläre Programmabarbeitung unterbrochen wird. Prinzipiell unterscheidet man zwischen **synchroner Unterbrechung (engl. Traps)** und **asynchroner Unterbrechung (engl. Interrupt)**, die in der Regel auf programmexterne Ursachen zurückzuführen sind.

Traps ergeben sich aus dem unmittelbar ablaufenden Programm – und sind meist einer Fehlersituation gleichzusetzen (z.B. Division durch 0, Unter-/Überschreiten von Wertebereichen (arithmetische Überläufe)); tritt eine solche Ausnahmesituation auf, so wird in die -der jeweiligen Ausnahmesituation zugeordneten- Unterprogrammroutine verzweigt (engl. **exception routine** oder **exception handler**; wird im Deutschen oft als **Service-Routine** übersetzt). Einige µC-Architekturen erlauben jedoch auch die gezielte Auslösung von Traps mittels Befehlen. Damit verbunden kann meist in einen speziellen / abgesicherten Systemmodus gewechselt werden („Supervisory Modus“, „Protected Mode“); in diesem Modus können dann z.B. Debugdaten gesammelt werden oder in diesem Modus laufen (bestimmte) Betriebssystemroutinen.

Asynchrone Unterbrechungen haben prinzipiell nichts mit der gerade stattfindenden Programmabarbeitung zu tun. Es handelt sich hier um Ausnahmesituationen, die entweder unvorhersehbar eingetreten sind (z.B. HW-Defekt, Einbruch der Versorgungsspannung) oder -in den allermeisten Fällen- auf Ein-/Ausgabegeräte oder Peripheriekomponenten zurückzuführen sind. Dafür stehen ein- oder mehrere Interrupt-Eingänge zur Verfügung. Tritt ein Interrupt-Ereignis auf, wird in die -diesem Interrupt zugeordnete- **Interrupt-Service-Routine (ISR)** verzweigt.

Je nach Architektur muss entweder in der ISR erst festgestellt werden, wer der Auslöser dieses Interrupts war, oder es gibt eine Tabelle (**Vektortabelle**) im Speicher, in denen die Startadressen (werden als **Vektoren** bezeichnet) abgelegt sind.

Tritt eine Ausnahmesituation ein, so unterbricht der Prozessor die Abarbeitung des aktuellen Programms, der Programmzähler und Prozessorstatus (Flags) werden auf dem Stack abgelegt und die ISR wird aufgerufen (entweder die zentrale ISR oder es wird die Vektornummer bestimmt und gem. Vektortabelle in die entsprechende Routine verzweigt).

Da dieser Ablauf sowohl für Traps als auch für Interrupts nahezu gleich ist (oft gibt es nur marginale Unterschiede), wird nur eine Vektortabelle geführt. Der -prinzipiell einzige- Unterschied zwischen Trap und Interrupt besteht darin, wie die Vektornummer gebildet wird.

Jedem Interrupt wird auch eine Priorität zugeteilt; ein Interrupt mit einer „höheren Priorität“ kann einen Interrupt mit einer „niedrigeren Priorität“ unterbrechen – wobei eine „höhere Priorität“ nicht zwangsläufig in Form eines größeren Zahlenwertes ausgedrückt wird; bei vielen Architekturen hat die Prioritätsnummer 0 die allerhöchste Priorität.

Der Programmierer ist meist auch in der Lage, festzulegen, welche Interrupts zugelassen sind und welche nicht. D.h. je nach Programmablauf kann die Abarbeitung von ISR unterbunden oder eingeschränkt werden.

Umsetzung bei PSoC5LP - Auszug „PSoC 5LP Architecture TRM, Document No. 001-78426 Rev. *C“:

The Interrupt Controller provides the mechanism for hardware resources to change the program address to a new location independent of the current execution in the main code.

The interrupt controller also handles continuation of the interrupted code being executed after the completion of the interrupt service routine.

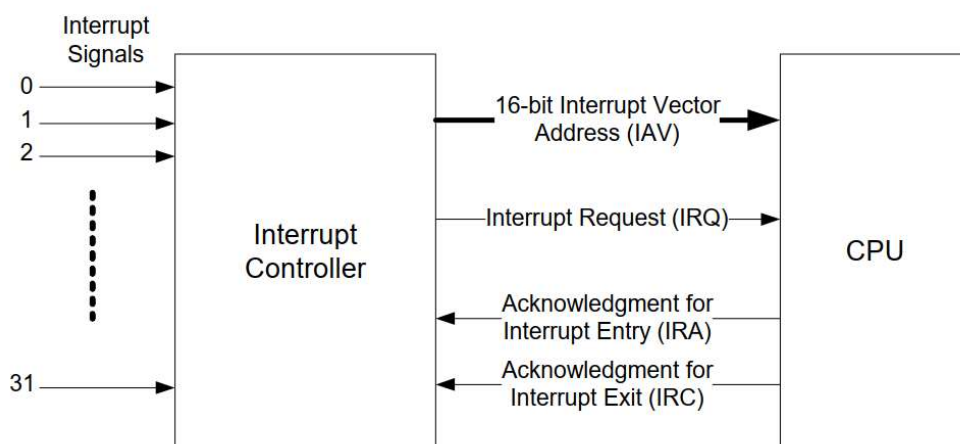
7.1 Features

The following are features of the interrupt controller:

- *Supports 32 interrupt lines*
- *Programmable interrupt vector*
- *Configurable priority levels from 0 to 7*
- *Support for dynamic change of priority levels*
- *Support for individual enable/ disable of each interrupt*
- *Nesting of interrupts*
- *Multiple sources for each interrupt line (can be either fixed function, UDB, or from DMA)*
- *Supports both level trigger and pulse trigger*
- *Tail chaining, late arrivals and exceptions are supported in PSoC 5LP devices*

7.2 Block Diagram

Figure 7-1. Interrupt Controller Block Diagram



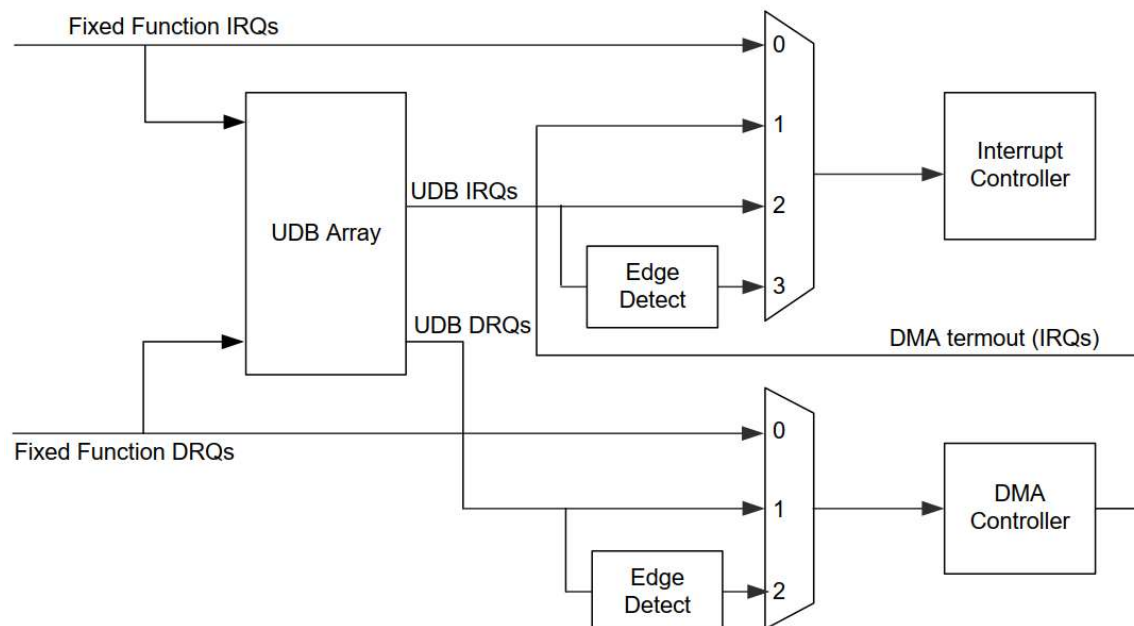
7.3 How It Works

The interrupt controller supports 32 interrupt signals. The interrupt signal can come from one of the three sources (see Figure 7-2):

- Fixed function block
- DMA channels
- UDB blocks

The interrupt signal routing is very flexible with PSoC 5LP architecture. The interrupt lines pass through a multiplexer. The mux selects one among the following: Fixed function IRQ (Interrupt request), UDB IRQ with level, UDB IRQ with Edge, and DMA IRQ. The `IDMUX.IRQ_CTL` register is used to configure the mux for the IRQ selection.

Figure 7-2. Interrupt and DMA Processing in the IDMUX



The interrupt controller unit prioritizes and sends the request to the CPU for execution. The list of interrupt sources and the corresponding interrupt number is available in the device datasheet.

[...]

7.3.3 Interrupt Priority

The interrupt controller provides a priority handling feature to help a user assign priority for each interrupt. Characteristics of this feature are as follows:

- Eight levels of interrupt priorities from 0 to 7.
- Priority level 0 is highest and level 7 is lowest.
- Priority levels set using the Interrupt Priority Registers `PRI[x]`.

■ *Support of dynamic configuration of priority levels – A change of priority level of an interrupt on the fly does not affect the current execution of the same interrupt; it takes effect for the next assertion.*

Priority handling is very important in the following cases:

■ *Case 1 – If an interrupt (INT B) is asserted when another interrupt (INT A) is being executed, there are three possibilities with unique handling sequences:*

□ *If INT A has lower priority than INT B:*

- 1. INT A is stopped at the point of execution.*
- 2. The details of INT A are pushed to the stack, and INT B begins to execute.*
- 3. After the execution of INT B, INT A execution is resumed from the point of its interruption.*

□ *If INT A has higher priority than INT B:*

- 1. INT B has to wait until INT A is executed.*
- 2. After the execution of INT A, INT B can start execution.*

□ *If INT A and INT B have equal priority:*

- 1. If INT A is being executed; INT B has to wait until INT A is executed. After the execution of INT A, INT B can start execution.*
- 2. If INT B is being executed; INT A has to wait until INT B is executed. After the execution of INT B, INT A can start execution.*

■ *Case 2 – During the simultaneous occurrence of interrupts:*

- *If INT A has lower priority than INT B, then INT B wins arbitration and begins to execute.*
- *If INT A has higher priority than INT B, then INT A wins arbitration and begins to execute.*
- *If INT A and INT B have equal priority, then the interrupt with the lower index number wins arbitration and begins to execute.*

7.3.4 Level versus Pulse Interrupt

The interrupt controller supports both Level and Pulse interrupts. The interrupt controller includes the Pulse detection logic, which detects the rising edge on the interrupt line. The pulse detection logic pends the interrupt bit whenever it detects the rising edge. The

interrupt controller detects any assertion in the interrupt signal and executes the interrupt as follows:

■ **Level Interrupt** – With level interrupts, the interrupt request bit in the corresponding peripheral register must be cleared by the firmware inside the interrupt service routine. [...]

■ **Pulse Interrupt** – A pulse occurs at the interrupt line. The low to high edge of the pulse sets the pending bit and the corresponding interrupt is executed. If the pulse occurs while the pending bit is already set, the second pulse has no effect, because the pending bit is already set. The Pending bit is automatically cleared by the interrupt controller at ISR entry. [...]

[...]

7.4.6 Exceptions

PSoC 5LP architecture supports 15 different exceptions, as shown in Table 7-3.

These exceptions are used to handle fault conditions that can occur in the system. Exceptions can have fixed priority or configurable priority. Exceptions are handled in the same manner as interrupts. The State register is used to enable or disable exceptions.

Table 7-3. PSoC 5LP Exceptions

Interrupt Number	Exception Type	Priority	Comments
1	Reset	-3 (highest) not programmable	Reset
2	NMI	-2 not programmable	Non-Maskable Interrupt
3	Hard Fault	-1 not programmable	All fault conditions if the corresponding handler is not enabled
4	MemManage Fault	Programmable	Memory management fault; MPU violation or access to illegal locations
5	Bus Fault	Programmable	Bus error; occurs when AHB interface receives an error response from a bus slave (also called prefetch abort if it is an instruction fetch or data abort if it is a data access)
6	Usage Fault	Programmable	Exceptions due to program error
7	Reserved	NA	--
8	Reserved	NA	--
9	Reserved	NA	--
10	Reserved	NA	--
11	SVCall	Programmable	System Service Call
12	Debug Monitor	Programmable	Debug monitor (watchpoints, breakpoints, external debug request)
13	Reserved	NA	--
14	PendSV	Programmable	Pendable request for system device
15	SYSTICK	Programmable	System Tick Timer

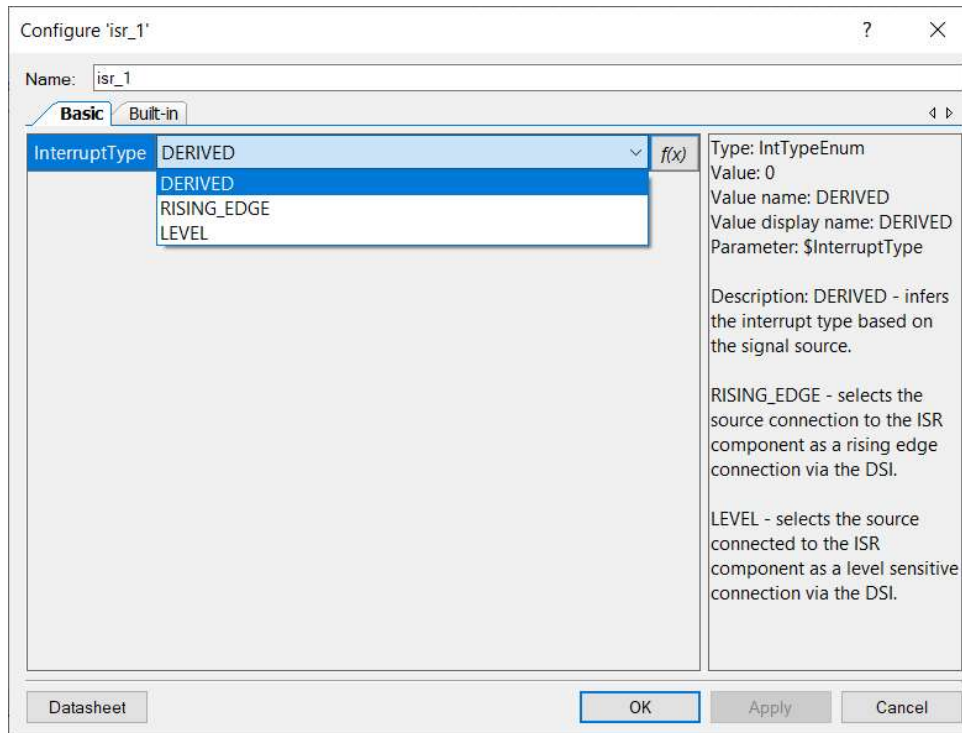
1. Übung: UDB-Interrupt Rising-Edge / Level

PSoC Creator Schaltung:



Die Interrupt-Komponente wird aus dem „Component Catalog“ ausgewählt und im Zeichenbereich platziert. „Taster1“ ist ein digitaler Eingangspin (mit Drive Mode = Resistive Pullup), LED_D7 ein digitaler Ausgangspin (Default-Einstellungen).

Die Interrupt-Komponente bietet nur eine Einstelloption – InterruptType:



Auswahlmöglichkeit ist hier „Rising Edge“ (Auslösen des Interrupts bei steigender Flanke), „Level“ (Auslösen des Interrupts bei log. 1) und Derived (= Default-Einstellung). Hier wird - abhängig von der Interruptquelle- automatisch zwischen Rising_Edge und Level gewählt.

Auszug Datenblatt Interrupt-Komponente:

DERIVED – This is the default setting. It inspects the driver of the “int_signal” and derives the interrupt type based on what it is connected to. For most fixed-function blocks, the interrupt type is LEVEL. For UDB signal sources, the interrupt type is RISING_EDGE.

Der Interrupt-Type ist auf **RISING_EDGE** zu ändern.

Die 4 Taster (Buttons) des LearnSoC! Boards schalten nur gegen GND; in Zusammenhang mit dem internen Pull-Up Widerstand (Pin-Konfiguration) bedeutet dies:

- wird der Taster nicht gedrückt, so liegt log. 1 am Eingangspin
- wird der Taster gedrückt, dann log. 0 → fallende Flanke

Da man die Interruptkomponente jedoch nicht für eine fallende Flanke oder log. 0 Pegel konfigurieren kann, wird der Inverter (Not) benötigt.

Ein „Generate Application“ (Build → Generate Application) ist empfehlenswert, bevor mit der Erstellung der FW begonnen wird.

Es gibt zwei Möglichkeiten, wo man den Code der Interrupt-Service-Routine codieren kann; in speziell vorgesehenen Abschnitten der `isr_1.c` oder außerhalb mittels Makros – in dieser Übungsanleitung wird nur die 2. Möglichkeit (Makros) verwendet.

Main.c:

```
#include "project.h"

CY_ISR(Taster1_ISR) {
    LED_D7_Write(1);
    CyDelay(500);
    LED_D7_Write(0);
    CyDelay(100);
}

int main(void) {

    isr_1_StartEx(Taster1_ISR);
    isr_1_ClearPending();

    CyGlobalIntEnable; /* Enable global interrupts. */

    /* Place your initialization/startup code here (e.g. MyInst_Start()) */

    for(;;)
    {
        /* Place your application code here. */
    }
}
```

Der Name der Interrupt-Service-Routine kann prinzipiell frei gewählt werden – wichtig ist jedoch, dass der selbe Name beim `CY_ISR`-Makro und bei `StartEx` verwendet wird:

```
CY_ISR(Taster1_ISR)
isr_1_StartEx(Taster1_ISR);
```

... die Namensgebung sollte natürlich auch zu einem leichter lesbaren Code führen.

Beim Auslösen des Interrupts wird die LED D7 für 500ms ein- und danach wieder ausgeschaltet. Nach dem Ausschalten wird 100ms gewartet.

Mit dem `startEx()`-Aufruf werden alle internen Konfigurationen vorgenommen – Auszug Datenblatt:

This function disables the interrupt, sets the interrupt vector based on the address passed in, sets the priority from the value in the Design Wide Resources Interrupt Editor, then enables the interrupt in the interrupt controller.

`ClearPending()` löscht ein eventuell gesetztes Interruptflag. Mit `CyGlobalIntEnable` werden systemweit alle Interrupts freigegeben.

Pin-Mapping:

	Name	Port	Pin	Lock
<input checked="" type="checkbox"/>	LED_D7	P2[7]	2	<input checked="" type="checkbox"/>
<input checked="" type="checkbox"/>	Taster1	P0[7]	56	<input checked="" type="checkbox"/>

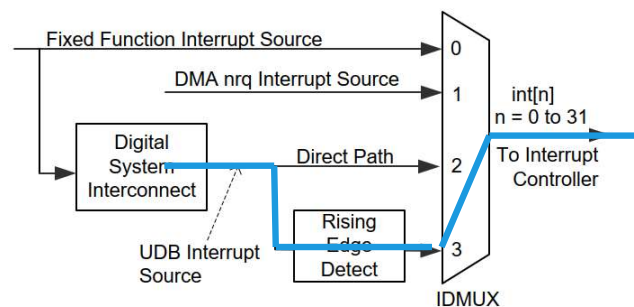
... es ist auch eine physikalische Verbindung (Drahtbrücke) zwischen „Button 1“ und P0[7] herzustellen.

Wird das Projekt kompiliert und auf das LearnSoC!-Board gespielt, dann sollte folgendes Systemverhalten beobachtet werden können:

- bei Betätigung des Tasters leuchtet LED D7 für ½ Sekunde
- egal wie lange der Taster gedrückt bleibt, die LED D7 bleibt aus (die ISR wird nur 1x durchlaufen)
- beim Loslassen kann die LED D7 noch einmal leuchten – dies ist auf das „Prellen“ des Tasters zurückzuführen!

Anmerkung: das „Prellen“ lässt sich hier schön demonstrieren, wenn man mehrmals auf den Taster drückt.

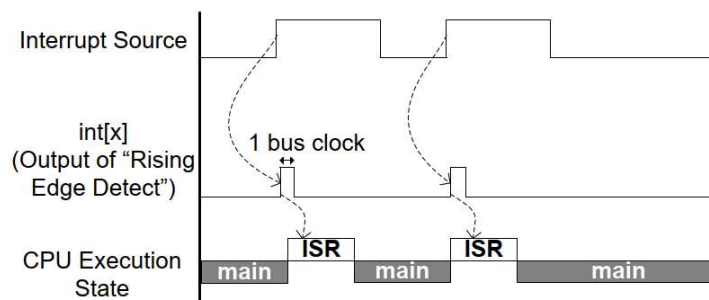
Aufgrund der Konfiguration wird intern folgendes Routing vorgenommen:



... NOT wird im UDB realisiert und der Ausgang über die „Digital System Interconnect“ dem IDMUX zugeführt. Aufgrund der „Rising Edge“-Einstellung befindet sich der MUX in Stellung 3.

Die steigende Flanke löst im „Rising Edge Detection“ Block einen kurzen Impuls mit der Länge einer „Bus Clock Periode“ aus. Egal wie lange der High-Pegel am Interrupt-Eingang anliegt und/oder wann die fallende Flanke kommt... erst mit der nächsten steigenden Flanke wird die ISR aufgerufen.

Figure 5. Edge Triggered UDB Interrupt Source



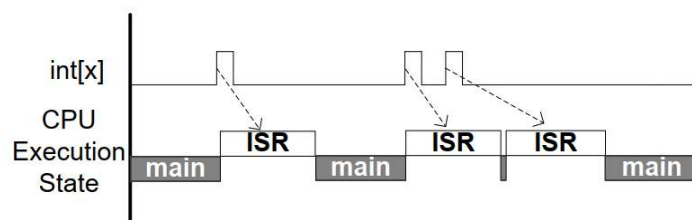
Bildquelle: Application Note AN54460 von Infineon (ehemals Cypress)

Mit dem Aufruf der ISR wird auch das Interrupt-Flag zurückgesetzt. Um dieses Verhalten zu demonstrieren, wird die Zeitspanne in der LED D7 ein ist, um den Faktor 10 verlängert:

```
LED_D7_Write(1);
CyDelay(5000);
LED_D7_Write(0);
```

Beim 1. Tastendruck wird die ISR aufgerufen und das Interrupt-Flag zurückgesetzt. Wenn innerhalb der Abarbeitungszeit der ISR (d.h. innerhalb der ca. 5100ms) noch eine steigende Flanke detektiert wird, dann führt diese steigende Flanke zum Setzen des Interrupt-Flags; nach der Beendigung der ISR wird zurück in die main und dann wieder sofort in die ISR gesprungen. Mit dem Aufruf der ISR wird das Flag wieder zurückgesetzt.

Figure 3. Edge Triggered Interrupts

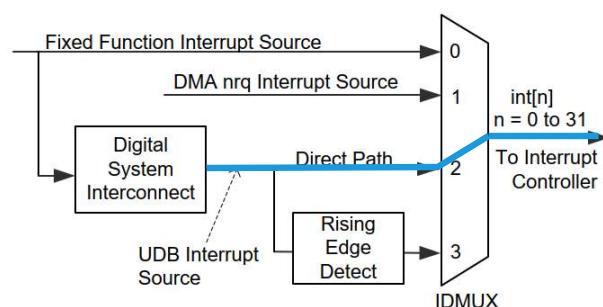


Bildquelle: Application Note AN54460 Infineon (ehemals Cypress)

Treten jedoch während der Abarbeitungszeit zwei oder mehr steigende Flanken auf, so gehen diese Requests verloren. Es gibt nur „ein“ Interrupt-Flag; dieses kann nur gesetzt / zurückgesetzt sein, ein Zähler wie oft dieses Flag „gesetzt werden müsste“ existiert nicht.

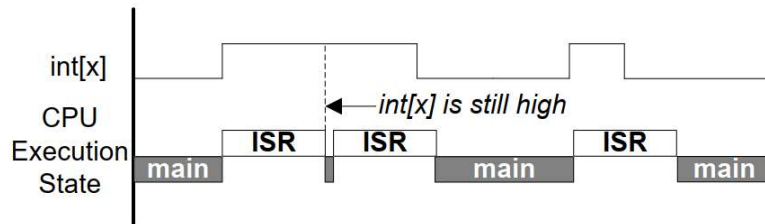
Interrupt-Service-Routinen sollten so performant wie möglich codiert sein. Verzweigungen in andere Unterprogramme und Verzögerungen, die eine weitere Abarbeitung blockieren (wie hier mit den CyDelay-Aufrufen) sind absolut zu vermeiden. Oft wird in der ISR nur ein Variableninhalt verändert – alles weitere erfolgt dann im eigentlichen Programmablauf. Mehr dazu aber später...

Die Delay-Zeit ist wieder auf 500ms zu setzen und der **InterruptType** ist auf **Level** zu ändern.



Solange der Taster gedrückt bleibt, wird die ISR erneut aufgerufen – d.h. jeder Beginn / Aufruf ist durch das Flackern der LED erkennbar.

Figure 2. Level Triggered Interrupts

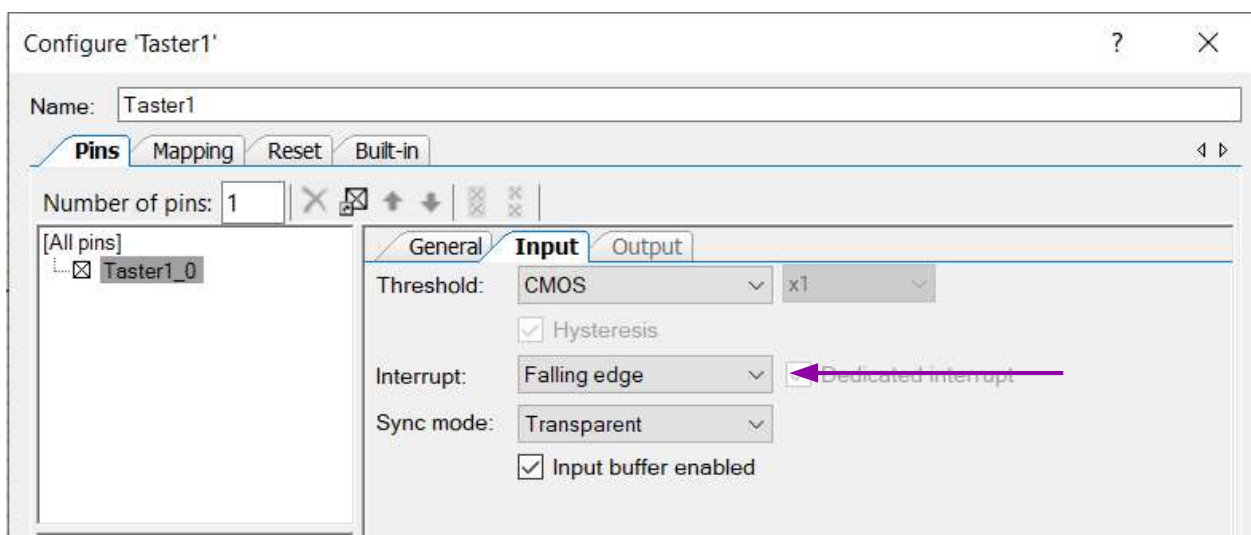


Bildquelle: Application Note AN54460 von Infineon (ehemals Cypress)

Abschließend ist der **InterruptType** auf **Derived** umzustellen. Nun entscheidet die PSoC-Creator-IDE anhand der Interrupt-Quelle, ob ein „Rising Edge“- oder „Level“-Verhalten anzuwenden ist. Bei UDB ist diese Default-Einstellung suboptimal... da hier der Entwickler eigentlich wissen sollte, ob Flanken- oder Pegeltriggerung anzuwenden ist. Anders sieht dies bei PSoC-Komponenten aus, die einen dedizierten Interrupt-Ausgang besitzen (und meist auch erweiterte Interrupt-Einstellungen bieten).

2. Übung: Pin-Komponente & Interrupt

Es kommt in der Praxis sehr oft vor, dass eine Zustandsänderung an einem Eingangspin einen Interrupt auslösen muss; ein „Pollen“ (d.h. ein permanentes Abfragen als Teil der Programmabarbeitung) ist eher eine Seltenheit. Die Pin-Komponente besitzt daher eine **Port Interrupt Controller Unit (PICU)**, die Einstellung erfolgt auf der Registerseite „Input“:



Wird eine Interrupt-Option ungleich „None“ gewählt, so wird ein zusätzlicher irq-Anschluss eingeblendet. Die PSoC Creator-Schaltung ist nun wie folgt abzuändern:



(Anmerkung: zusätzlich wurde auf der Registerseite General die Option „HW Connection“ ausgewählt)

Beim Taster1 ist Interrupt auf „**Falling edge**“ zu setzen, bei der Interrupt-Komponente muss „**Derived**“ ausgewählt sein:



Am C-Code ist nichts zu ändern; d.h. die main.c sieht wie folgt aus:

```
#include "project.h"

CY_ISR(Taster1_ISR) {
    LED_D7_Write(1);
    CyDelay(500);
    LED_D7_Write(0);
    CyDelay(100);
}

int main(void) {

    isr_1_StartEx(Taster1_ISR);
    isr_1_ClearPending();

    CyGlobalIntEnable; /* Enable global interrupts. */

    /* Place your initialization/startup code here (e.g. MyInst_Start()) */

    for(;;)
    {
        /* Place your application code here. */
    }
}
```

Wird nun das neue Projekt auf das LearnSoC!-Board gespielt und der Taster 1 betätigt, so sieht man ein wiederholtes Aufrufen der ISR (!).

Die Erklärung findet man im Datenblatt der Pin-Komponente:

[... It] uses dedicated pin interrupt logic, which latches the pins that generated the interrupt events. After an interrupt occurs, the Pin_ClearInterrupt() function must be called to clear the latched pin events to enable detection of future events.

D.h. der Aufruf der ISR führt nicht zu einem automatischen Zurücksetzen des Interrupt-Flags der Pin-spezifischen Interrupt-Logik. Da kein `Pin_ClearInterrupt()` Aufruf in unserer `main.c` erfolgte, bleibt das Flag gesetzt... und damit erklärt sich auch das wiederholende Aufruf-Verhalten.

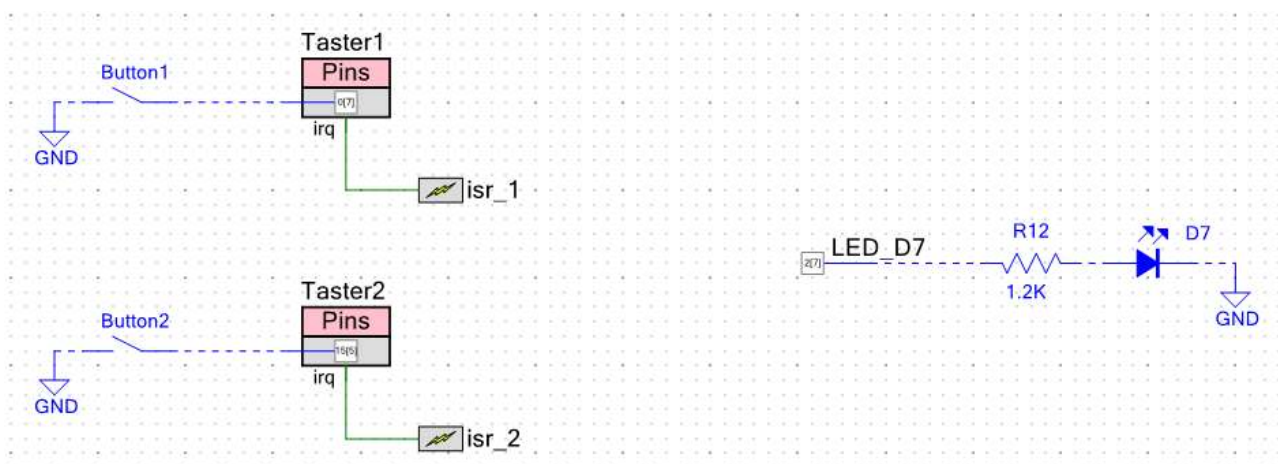
Die ISR ist daher wie folgt zu ändern:

```
CY_ISR(Taster1_ISR) {  
  
    Taster1_ClearInterrupt();  
  
    LED_D7_Write(1);  
    CyDelay(500);  
    LED_D7_Write(0);  
    CyDelay(100);  
}
```

... mit dem ISR-Aufruf wird nun auch das Interrupt-Flag der Pin-spezifischen Interrupt-Logik zurückgesetzt. Tritt während der ISR-Bearbeitung ein Interrupt-Event auf (→ prellender Taster!), so wird das Flag erneut gesetzt und man sieht einen erneuten ISR-Aufruf. Wird `ClearInterrupt` am Ende der ISR aufgerufen, so wird erst am Ende das Flag gelöscht... da die Taster prellen, sieht man hier schön die Unterschiede.

3. Übung: 2 Taster = 2 Interruptvektoren?

Mit Taster 1 soll die LED D7 ein-, mit Taster 2 wieder ausgeschaltet werden. Die Realisierung im PSoC Creator könnte wie folgt aussehen:



... und die `main.c` könnte wie folgt aussehen:

```
#include "project.h"  
  
CY_ISR(Taster1_ISR) {  
    LED_D7_Write(1);  
    Taster1_ClearInterrupt();  
}
```

```

CY_ISR(Taster2_ISR) {
    LED_D7_Write(0);
    Taster2_ClearInterrupt();
}

int main(void) {
    isr_1_StartEx(Taster1_ISR);
    isr_1_ClearPending();
    isr_2_StartEx(Taster2_ISR);
    isr_2_ClearPending();




    CyGlobalIntEnable; /* Enable global interrupts. */

    /* Place your initialization/startup code here (e.g. MyInst_Start()) */

    for(;;)
    {
        /* Place your application code here. */
    }
}

```

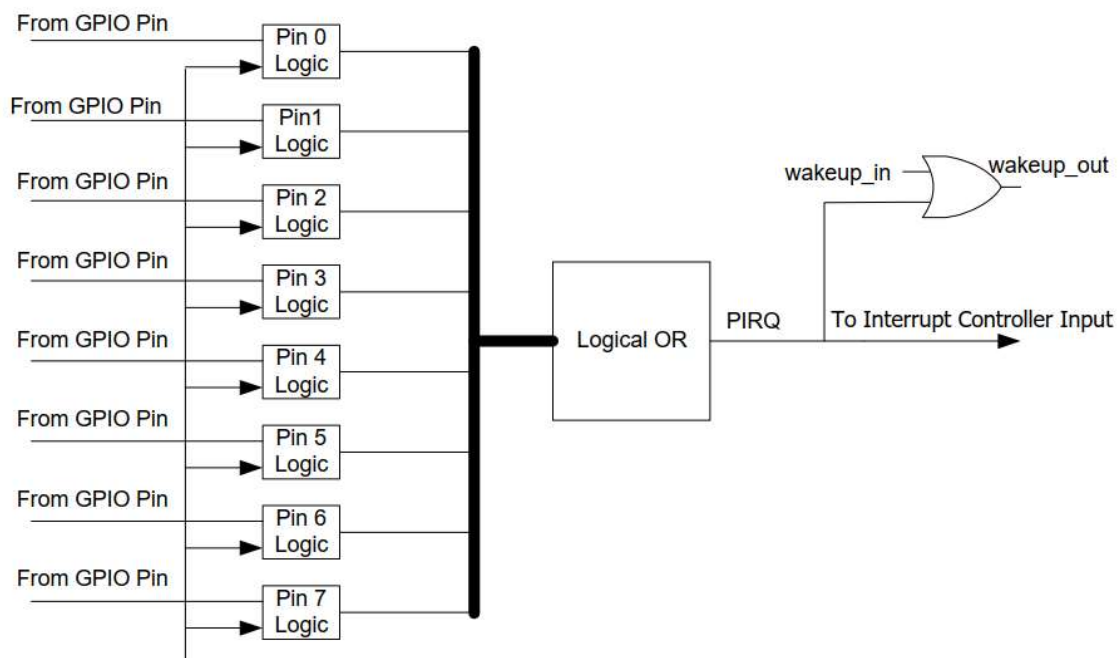
Versucht man nun folgende Pin-Zuordnung ...

	Name	Port	Pin	Lock
	LED_D7	P2[7]	2	<input checked="" type="checkbox"/>
	Taster1	P0[7]	56	<input checked="" type="checkbox"/>
	Taster2	P0[6]	55	<input checked="" type="checkbox"/>

... wird der Compile-Vorgang mit folgender Fehlermeldung abgebrochen:

Notice List			
Workspace	2 Errors	0 Warnings	0 Notes
	Description	File	Error...
1	apr.M0038:IO "Taster2(0)" cannot be placed into "P0[6]" because it has an interrupt that conflicts with another interrupt already in this port. Pick a different place for this IO.		Interrupt02
2	fit.M0050:The fitter aborted due to errors, please address all errors and rebuild.		Interrupt02

Begründung: P0[6] und P0[7] gehören zum selben Port; und pro Port gibt es nur eine PICU.



Bildquelle: Bildauszug aus dem PSoC 5LP Architecture TRM von Infineon (ehemals Cypress)

... d.h. die Interrupt-Information aller 8 Pins eines Ports sind über ein ODER-Gatter miteinander verknüpft; egal ob die Ursache des Interrupts auf ein Ereignis am Pin0 oder Pin6 zurückzuführen ist; der Interrupt-Vektor ist stets der gleiche.

Es ist daher nicht möglich, P0[6] und P0[7] auf zwei unterschiedliche Interrupt-Vektoren zeigen zu lassen. Wenn man mit getrennten Interrupt-Vektoren arbeiten will, dann sind zwei verschiedene Portgruppen auszuwählen:

	Name	Port	Pin	Lock
<input checked="" type="checkbox"/>	LED_D7	P2[7]	2	<input checked="" type="checkbox"/>
<input checked="" type="checkbox"/>	Taster1	P0[7]	56	<input checked="" type="checkbox"/>
<input checked="" type="checkbox"/>	Taster2	P15[5]	61	<input checked="" type="checkbox"/>

... mit dieser Pin-Zuordnung funktioniert dieses Umsetzungsbeispiel einwandfrei.

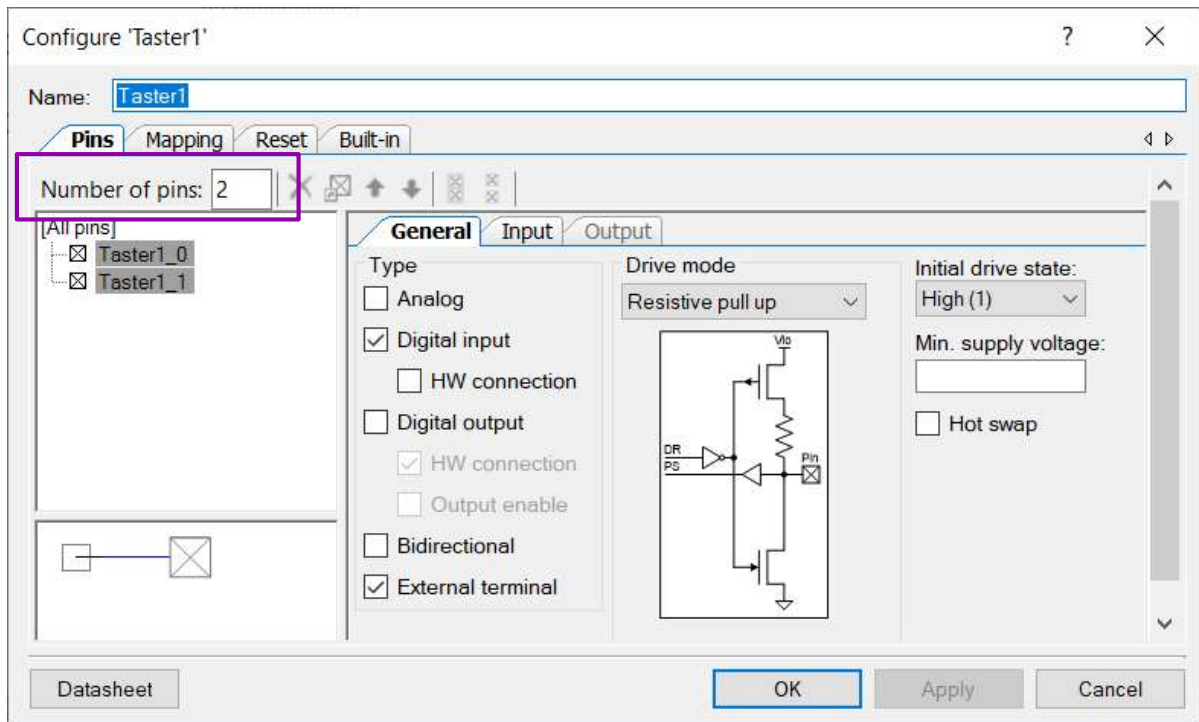
4. Übung: 2 Taster = 1 Interruptvektor?

Die Übung 3 lässt sich sehr elegant umsetzen, wenn man zwei Pins innerhalb der selben Portgruppe wählt – also z.B. P0[6] und P0[7]. Aufgrund der Oder-Verknüpfung wird zwar immer der selbe Interrupt-Vektor angesprungen, die `Pin_ClearInterrupt()` - Funktion liefert aber zurück, welcher Pin den Interrupt ausgelöst hat:

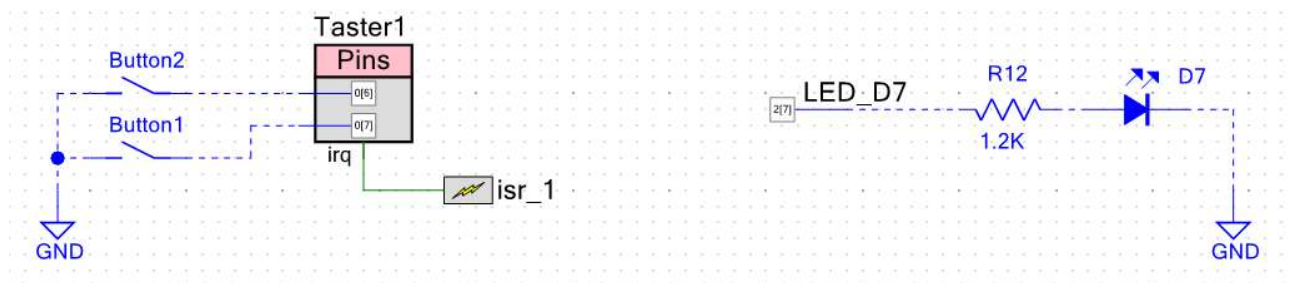
Auszug Datenblatt Pin-Komponente:

If more than one pin in the Pins Component can generate an interrupt, the `Pin_ClearInterrupt()` return value can be decoded to determine which pins generated interrupt events.

Um dies zu demonstrieren, wird die Anzahl der Pins des digitalen Tastereingangs auf 2 erhöht:



... und damit folgt für den PSoC Schaltplan:



	Name	Port	Pin	Lock
<input checked="" type="checkbox"/>	\Taster1[1:0]\	P0[7:6]	56, 55	<input checked="" type="checkbox"/>
<input checked="" type="checkbox"/>	LED_D7	P2[7]	2	<input checked="" type="checkbox"/>

Löst nun der Button2 (verbunden mit P0[6]) den Interrupt aus, so liefert `Taster1_ClearInterrupt()` den Rückgabewert 0x01. Wird Button 1 gedrückt und dadurch der Interrupt ausgelöst, so ist der Rückgabewert 0x02.

Mit Button1 soll (unverändert) LED D7 ein- und mit Button2 ausgeschaltet werden. Der C-Code kann wie folgt aussehen:

```
#include "project.h"

CY_ISR(Taster1_ISR) {
```

```

/* read the PICU interrupt status register, with a clear on read */
uint8 taster = Taster1_ClearInterrupt();

if (taster & 0x01)
    LED_D7_Write(0);    //Button 2 = 0.6 = "Switch 1" = aus

if (taster & 0x02)
    LED_D7_Write(1);    //Button 1 = 0.7 = "Switch 2" = ein
}

int main(void) {

    isr_1_StartEx(Taster1_ISR);
    isr_1_ClearPending();

    CyGlobalIntEnable; /* Enable global interrupts. */

    /* Place your initialization/startup code here (e.g. MyInst_Start()) */

    for(;;)
    {
        /* Place your application code here. */
    }
}

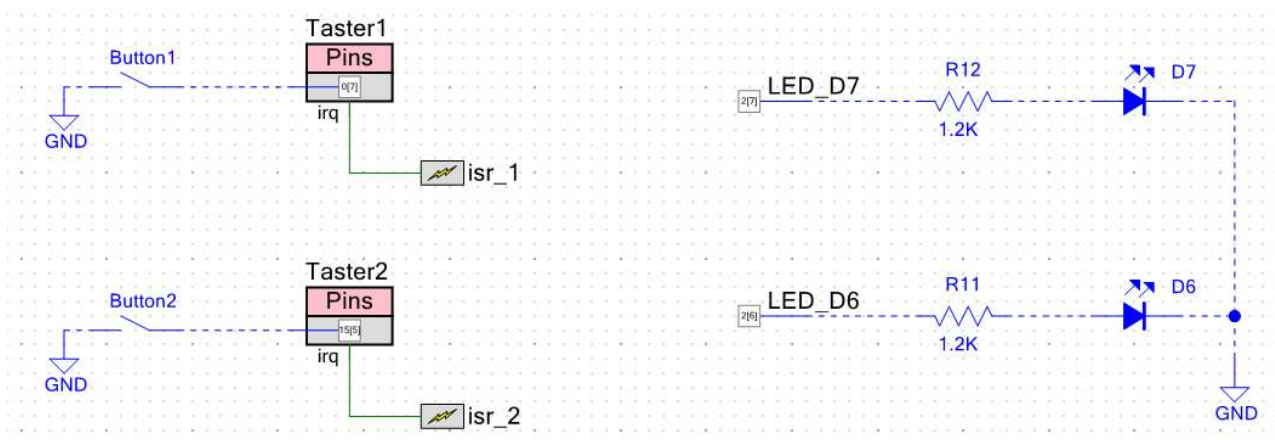
```

5. Übung: Interrupt-Prioritäten

Jedem Interrupt muss nicht nur ein Interrupt-Vektor, sondern auch eine Priorität zugewiesen werden. Beim PSoC5LP gibt es einen Prioritätswertebereich zwischen 0 und 7, wobei der Wert 0 der allerhöchsten Priorität entspricht. Default-mäßig wird jedem Interrupt der Prioritätswert 7 zugeteilt.

Um die Auswirkungen unterschiedlicher Prioritäten zu demonstrieren, benötigt man (mindestens) zwei Interrupt-Komponenten (und damit 2 Interrupt-Vektoren).

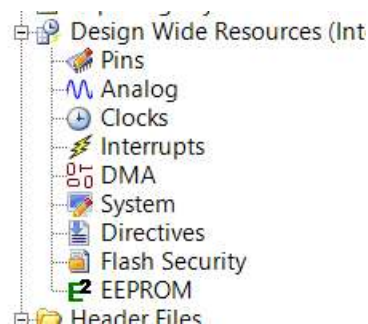
PSoC Creator Schematic:



... mit der Pinzuordnung:

	Name	Port	Pin	Lock
<input type="checkbox"/>	LED_D6	P2[6]	1	<input checked="" type="checkbox"/>
<input type="checkbox"/>	LED_D7	P2[7]	2	<input checked="" type="checkbox"/>
<input type="checkbox"/>	Taster1	P0[7]	56	<input checked="" type="checkbox"/>
<input type="checkbox"/>	Taster2	P15[5]	61	<input checked="" type="checkbox"/>

Die Interrupt-Priorität kann (im einfachsten Fall) über „Design Wide Resources“ → Interrupts eingesehen und verändert werden:



Instance Name	Interrupt Number	Priority (0 - 7)
isr_1	4	7
isr_2	12	7

Die „Interrupt Number“ wird automatisch vergeben und zugewiesen, die Priorität ist veränderbar – man sieht hier den Default-Wert von 7... der erstmals auch so belassen wird.

Beim Drücken auf Taster1 soll eine eventuell leuchtende LED D6 ausgeschaltet sowie die LED D7 ein- und nach 5sek wieder ausgeschaltet werden. Beim Drücken auf Taster2 wird nur die LED D6 eingeschaltet.

Die main.c kann wie folgt aussehen:

```
#include "project.h"

CY_ISR(Taster1_ISR) {
    LED_D6_Write(0);
    LED_D7_Write(1);
    CyDelay(5000);
    LED_D7_Write(0);

    Taster1_ClearInterrupt();
}

CY_ISR(Taster2_ISR) {
```

```
    LED_D6_Write(1);

    Taster2_ClearInterrupt();
}

int main(void) {

    isr_1_StartEx(Taster1_ISR);
    isr_1_ClearPending();

    isr_2_StartEx(Taster2_ISR);
    isr_2_ClearPending();

    CyGlobalIntEnable; /* Enable global interrupts. */

    /* Place your initialization/startup code here (e.g. MyInst_Start()) */

    for(;;)
    {
        /* Place your application code here. */
    }
}
```

Nach Kompilieren und Übertragen dieses Beispiels auf das LearnSoC!-Board soll nun folgendes Verhalten beobachtet werden:

Wird Button1 und unmittelbar danach Button2 gedrückt, so leuchtet für ca. 5 sek die LED D7 und im Anschluss LED D6. D.h. erst nachdem Taster1_ISR abgearbeitet wurde, wird die ISR des Tasters2 aufgerufen.

D.h.: während der Abarbeitung eines Interrupts werden weitere Unterbrechungsanforderungen, deren Prioritäten gleich oder niedriger sind, zwischengespeichert. D.h. die gerade stattfindende Abarbeitung der ISR wird nicht unterbrochen und es gehen auch keine Interrupt-Anfragen verloren (ausgenommen: Mehrfach-Interruptereignisse am selben Pin).

Nun wird dem Interrupt „isr_2“ eine höhere Priorität zugewiesen:

Instance Name /	Interrupt Number	Priority (0 - 7)
isr_1	4	7
isr_2	12	6

Wird nun Button1 und unmittelbar danach Button2 gedrückt, so leuchten beide LEDs und nach ca. 5sek nur mehr LED D6. D.h. mit Drücken des Tasters1 wird die ISR des Tasters1 aufgerufen und die blaue LED D7 eingeschaltet. Mit Drücken des Taster2 wird -aufgrund der höheren Priorität- die Abarbeitung von Taster1_ISR unterbrochen und die Taster2_ISR ausgeführt. Nach dem Ende von Taster2_ISR wird in die ISR von Taster1 zurückgesprungen – und mit der Abarbeitung dieser ISR an jener Stelle weitergefahren, die ohne isr_2 die nächste gewesen wäre.

Mit `GetPriority()` kann man die Interrupt-Priorität eines Interrupts im laufenden Programm abfragen, mit `SetPriority()` verändern. Dies soll anhand der folgenden kleinen Modifikation demonstriert werden:

```
CY_ISR(Taster2_ISR) {  
    LED_D6_Write(1);  
    Taster2_ClearInterrupt();  
    isr_2_SetPriority(7);  
}
```

In den Design Wide Resources wird dem Interrupt `isr_2` die Priorität 6 zugeteilt; ist daher von höherer Priorität als `isr_1`. D.h. `isr_2` kann die Abarbeitung der `Taster1_ISR` unterbrechen. In der Interrupt-Service-Routine `Taster2_ISR` wird jedoch die Priorität auf 7 verändert; d.h. nach dem ersten Abarbeiten der ISR erhält `isr_2` die selbe Priorität wie `isr_1`.

Wird das Programm kompiliert und auf das LearnSoC! Board übertragen, so sieht man:

Werden die Taster 1 und 2 unmittelbar hintereinander gedrückt, so unterbricht `isr_2` die Abarbeitung von `Taster1_ISR`. Wurde `Taster1_ISR` vollständig abgearbeitet, d.h. ist keiner der Interrupts aktiv und werden erneut die Taster 1 und 2 unmittelbar hintereinander gedrückt, so erfolgt keine Unterbrechung mehr... die Interrupts werden nun nacheinander abgearbeitet.

Mittels `Disable()` lässt sich auch ein Interrupt deaktivieren (und mittels `Enable()` wieder aktivieren). Auch dies lässt sich einfach anhand dieser Übungskonfiguration zeigen, indem `Taster2_ISR` wie folgt abgeändert wird:

```
CY_ISR(Taster2_ISR) {  
    LED_D6_Write(1);  
    Taster2_ClearInterrupt()  
    isr_2_Disable();  
}
```

Werden die Taster 1 und 2 unmittelbar hintereinander gedrückt, so unterbricht `isr_2` die Abarbeitung der `Taster1_ISR`. Allerdings wird in der ISR von `Taster2` der `isr_2` Interrupt deaktiviert – sobald einmal die ISR abgearbeitet wurde, wird der Interrupt gesperrt. Ein erneutes Betätigen von Taster 2 löst keine weitere Programmunterbrechung mehr aus.

6. Übung: vola... was?

In sehr vielen Anwendungsfällen wird in einer ISR nur ein Variableninhalt verändert. Zum Beispiel bei der Einstellung einer Lautstärke... je nach Tastendruck wird z.B. der Inhalt volume um eins erhöht oder verringert. Im 6. Übungsbeispiel demonstrieren wir diese Möglichkeit, wieder in einer sehr einfachen Umgebung:

In der ISR des Buttons1 wird ein Flag gesetzt. Dieses Flag wird im Hauptprogramm ausgewertet und je nach Status wird die LED D7 eingeschaltet oder nicht.

PSoC Creator Schematic:



... und das C-Programm könnte wie folgt aussehen:

```
#include "project.h"

uint8 isr_flag = 0;

CY_ISR(Taster1_ISR) {
    Taster1_ClearInterrupt();
    isr_flag = 1;
}

int main(void) {
    isr_1_StartEx(Taster1_ISR);
    isr_1_ClearPending();

    CyGlobalIntEnable; /* Enable global interrupts. */

    /* Place your initialization/startup code here (e.g. MyInst_Start()) */

    for(;;) {
        /* Place your application code here. */

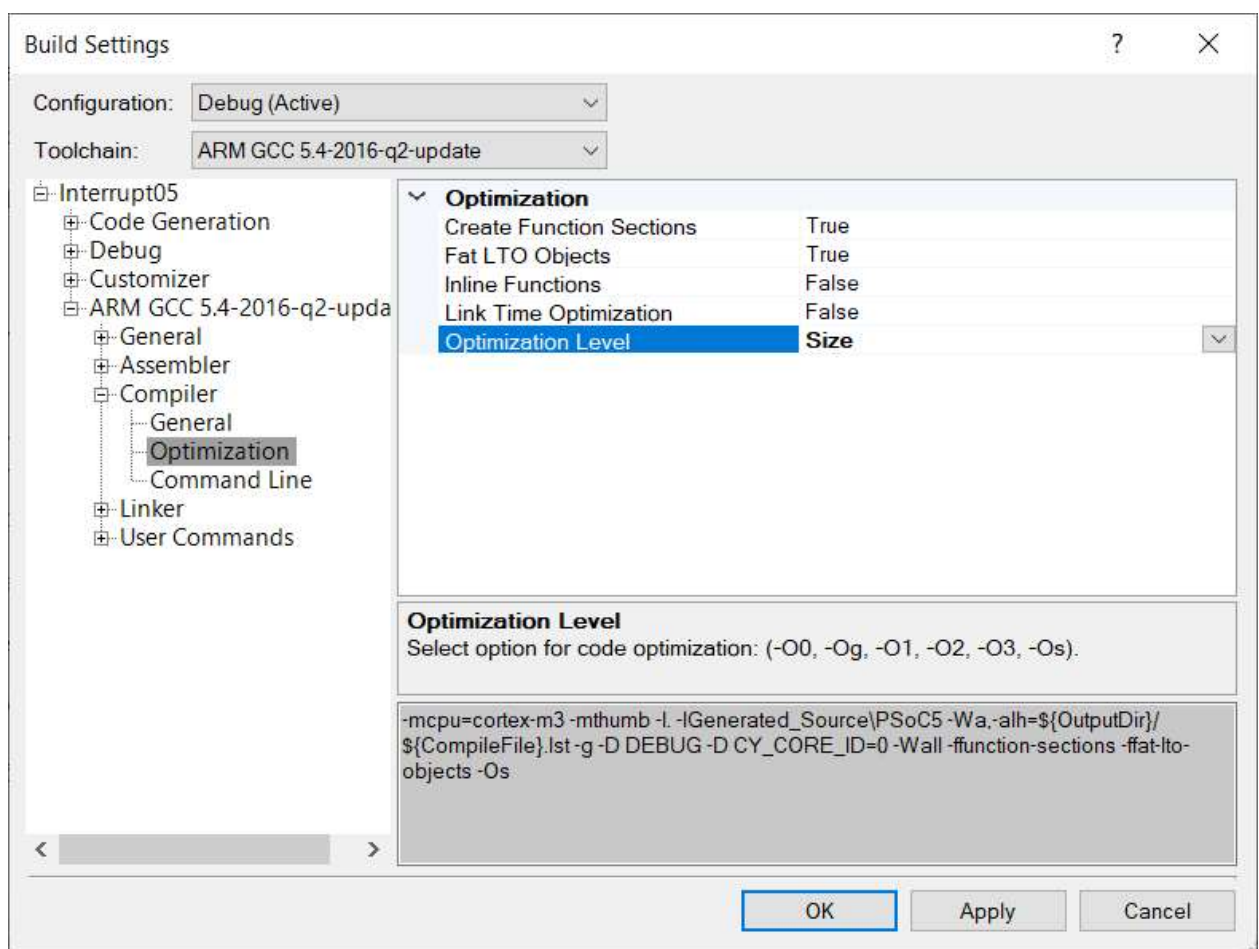
        if (isr_flag != 0)
            LED_D7_Write(1);
    }
}
```

Wird das Projekt kompiliert, auf das LearnSoC! Board übertragen und der Taster1 gedrückt, so sieht man die LED D7 leuchten... d.h. das Programm scheint einwandfrei zu funktionieren.

Wird jedoch die Code-Optimierung (Project → Build Settings → ARM GCC ... → Compiler → Optimization) verändert, dann kann das Resultat ganz anders aussehen. Standardmäßig ist die Code-Optimierung auf „Debug“ eingestellt und dieses Programm benötigt

```
Flash used: 1382 of 262144 bytes (0,5 %).  
SRAM used: 2505 of 65536 bytes (3,8 %). Stack: 2048 bytes. Heap: 128 bytes.
```

Ändert man nun die Code-Optimierung z.B. auf Size...



... um einen Code zu erhalten der möglichst wenig Speicherplatz benötigt, und führt man eine erneute Kompilierung durch, so sieht man folgende Ergebnisse:

```
Flash used: 1224 of 262144 bytes (0,5 %).  
SRAM used: 2505 of 65536 bytes (3,8 %). Stack: 2048 bytes. Heap: 128 bytes.
```

... d.h. anstelle von 1382 Bytes Flash-Speicher werden jetzt nur mehr 1224 Bytes benötigt... allerdings funktioniert das Programm auch nicht mehr. D.h. egal wie oft man Taster1 drückt, die LED D7 bleibt dunkel.

Strenggenommen müsste die globale Variable „isr_flag“ `static` sein, da die Variable nur 1x initialisiert werden darf (und zwar vom Compiler) und die Variable an einer festen Speicheradresse abzulegen ist; d.h.

```
static uint8 isr_flag = 0;
```

... aber auch diese Änderung zeigt keine Auswirkung auf das Systemverhalten – sprich: die LED D7 bleibt dunkel.

Was passiert? Im Zuge der Codeoptimierung stellt der Compiler fest, dass sich `isr_flag` im Programmfluß nie ändert. Die Variable wird mit 0 initialisiert und in der `main` wird dann in der `if`-Abfrage geprüft, ob der Variablenwert ungleich 0 wird... was -isoliert betrachtet- nie der Fall ist. Deshalb sieht der Compiler hier „Optimierungsbedarf“.

Dass sich „isr_flag“ außerhalb des regulären Programmflusses ändern kann, ist dem Compiler gezielt mitzuteilen. Es kann sein, dass sich hinter dieser Adresse eine Ein-/Ausgabeeinheit befindet und über diese Adresse die Kommunikation erfolgt. Es kann sein, dass eine Interrupt-Service-Routine den Variableninhalt verändert, dass im Rahmen eines direkten Speicherzugriffs (DMA) eine Änderung erfolgt, ... => so dass es keine Optimierungen im Zusammenhang mit dieser Variable geben darf.

Dies erfolgt mit dem Schlüsselwort `volatile` (deutsch: wechselhaft):

```
static volatile uint8 isr_flag = 0;
```

... und damit ist nicht nur die Funktion wieder hergestellt, der C-Code ist nun auch korrekt und vollständig.

7. Übung: SysTick

... in vielen Anwendungsfällen benötigt man eine „Systemuhr“, um Zeiten zu messen oder zeitliche Abläufe zu ermöglichen. Jedes Betriebssystem benötigt einen solchen Zeitgeber. Selbst Arduino besitzt eine solche Möglichkeit – die „Systemzeit“ kann z.B. mittels der Funktion `millis()` abgerufen werden:

millis()

Description

Returns the number of milliseconds since the Arduino board began running the current program. This number will overflow (go back to zero), after approximately 50 days.

Parameters

None

Returns

Number of milliseconds since the program started (unsigned long)

(Auszug Arduino Reference / Version 1.18.10)

Wirft man einen Blick „unter die Haube von Arduino“ (in diesem Falle: sieht man sich die wiring.c an), so sieht man, dass der Timer 0 verwendet wird... bei jedem Timer-Überlauf wird ein Interrupt erzeugt und in der ISR werden verschiedenen Variablen aktualisiert, die eine Zeitmessung in μs (\rightarrow Funktion `micros()`) und ms (\rightarrow Funktion `millis()`) ermöglichen (unter anderem).

Die `millis()` Funktion ist mit dem PSoC 5LP nachzubilden. Dabei wird mit einem ganz einfachen Ansatz gearbeitet – wie die professionelle Umsetzung aussieht, wird später erläutert. Da der PSoC eine Clock-Komponente besitzt, wird auf einen Timer gänzlich verzichtet (nicht optimal). Die Clock-Komponente wird auf 1kHz eingestellt, und ist mit der Interrupt-Komponente (`Rising_Edge`) verbunden. Jede ms wird somit ein Interrupt erzeugt. In der ISR wird der Variablenwert erhöht und so die Zeitmessung durchgeführt.

Die LED D7 soll im Sekundenrhythmus blinken; d.h. 500ms ein- und 500ms ausgeschaltet werden. Dafür sollte nicht die `CyDelay()`-Funktion sondern unser „millis“-Wert herangezogen werden. Auch hier wieder ein gaaaaaanz einfacher Ansatz: ist der `millis`-Wert ganzzahlig durch 500 teilbar, dann soll der Zustand der LED toggeln. Verwenden wir für `millis` einen `uint32` Datentyp, dann tritt nach etwas mehr als 49 Tagen eine Abweichung in unserer Blinkdauer auf (da 500 kein ganzzahliger Teiler von 2^{32} ist). Auch wenn es uns schwer fallen mag, wir gehen dieses Risiko ein :-)

Unsere PSoC Creator-Schaltung sieht daher wie folgt aus:



Und die einfache Umsetzung des C-Codes sieht z.B. wie folgt aus:

```
#include "project.h"

static volatile uint32 m = 0;    /* millis-Wert */

CY_ISR(systick_ISR) {
    m++;
}

int main(void) {

    isr_1_StartEx(systick_ISR);
    isr_1_ClearPending();

    CyGlobalIntEnable;    /* Enable global interrupts. */

    uint8 led_status = 0;

    for(;;) {
```

```
    if ((m % 500L) == 0) {  
        led_status = ~ led_status;    /* toggle */  
        LED_D7_Write(led_status);  
    }  
}  
}
```

Kompiliert man nun diese FW (Compiler-Optimierung wieder auf Debug zurückgestellt / Debug ist auch die Default-Einstellung), so sieht man die blaue LED D7 blinken.

Die FW ist nicht optimal – hin und wieder wird man Aussetzer des Blinkmusters feststellen... vor allem wenn man beginnt in der main() Änderungen vorzunehmen. Der systick-Interrupt kann zu jedem Zeitpunkt der Programmabarbeitung auftreten, daher kann der „ungeschützte“ Zugriff auf unsere Variable `m` problematisch sein (der Cortex M3 ist zwar ein 32bit Prozessor und `m` eine 32bit Variable, das ganze sieht aber bei einem 64bit Datentyp anders aus !!).

Es ist daher ratsamer, vor dem Lesezugriff auf die Variable `m` die Interrupts zu sperren... und danach wieder freizugeben:

```
uint32 getMilli() {  
    CyGlobalIntDisable;  
    uint32 temp = m;  
    CyGlobalIntEnable;  
    return temp;  
}
```

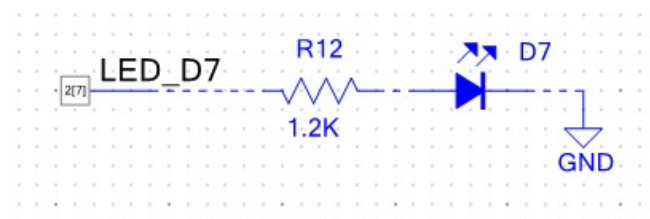
... und in der if-Abfrage mit getMillis() zu arbeiten:

```
if ((getMilli() % 500L) == 0) {
```

Würde nun während dem Lesezugriff auf `m` ein Interrupt eintreffen, dann würde diese Interrupt-Anforderung erst nach `CyGlobalIntEnable` abgearbeitet werden.

Da eine „Systemuhr“ für Betriebssysteme eine zwingende Voraussetzung ist, wurde von ARM bereits ein **SysTick-Feature als Teil der Cortex-M3-CPU** vorgeleistet. Über API-Aufrufe kann die SysTick-Rate eingestellt werden und fest zugeordnet wurde Interrupt-Vektor 15.

Das Top-Design besteht in diesem Falle nur aus dem digitalen Ausgangspin für die LED D7:



... und der C-Code kann wie folgt aussehen:

```
#include "project.h"

/* clock and interrupt rates, in Hz */
#define CLOCK_FREQ      BCLK__BUS_CLK__HZ
#define INTERRUPT_FREQ  5u      /* Interrupt Frequency = 5 Hz */

CY_ISR(SysTick_ISR) {
    static uint8 led_status = 0;
    led_status = ~ led_status;    /* toggle */
    LED_D7_Write(led_status);
}

int main(void) {
    /*Point the SysTick vector to the ISR in this file */
    CyIntSetSysVector(15, SysTick_ISR);

    /* Set the number of ticks between interrupts.
       Ignore the function success/fail return value.
       Defined in auto-generated core_cm3.h */
    (void)SysTick_Config(CLOCK_FREQ / INTERRUPT_FREQ);

    CyGlobalIntEnable; /* Enable global interrupts. */

    for(;;)
    {
        /* Do nothing in the main loop; code to do something is in the ISR */
    }
}
```

... dieses Beispiel wurde in Anlehnung an die (wirklich ausgezeichnete) Application Note AN54460 von Infineon (ehemals Cypress) erstellt. An dieser Stelle soll

- auf die AN54460
- auf das PSoC 5LP Architecture TRM (Technical Reference Manual)

verwiesen werden; es gäbe noch vieles zum Thema „Interrupt“ zu erzählen!