5TC option AUD

Embedded Programming Basics : embedded peripherals

Romain Michon, Tanguy Risset

Labo CITI, INSA de Lyon, Dpt Télécom

GRAME-CNCM INSA INSTITUT NATIONAL DES SCIENCES APPLIQUÉES LYON GRAME CENTRE NATIONAL DE CRÉATION MUSICALE, LYON

2 novembre 2023

Table of Contents

Compiling for teensy regular C++ programs

```
const int led = 13;

void setup() {
  pinMode(led, OUTPUT);
}

void loop() {
  digitalWrite(led, HIGH);
  delay(1000);
  digitalWrite(led, LOW);
  delay(1000);
}
```

using arduino

```
#include <Arduino.h>

const int ledPin = 13;

extern "C" int main(void)
{
  pinMode(ledPin, OUTPUT);
  while (1) {
    digitalWrite(ledPin, HIGH);
    delay(100);
    digitalWrite(ledPin, LOW);
    delay(100);
  }

}
```

using gcc

How to compile Teensy programs with a Makefile

1. Identify all the directories with .C or .C++ files used for Audio processing on teensy :

```
  KERNEL_SOURCES = $(ARDUINOPATH)/hardware/teensy/avr/cores/teensy4
AUDIO_SOURCES = $(ARDUINOPATH)/hardware/teensy/avr/libraries/Audio
SPI_SOURCES = $(ARDUINOPATH)/hardware/teensy/avr/libraries/SPI
SD_SOURCES = $(ARDUINOPATH)/libraries/SD/src
SERIALFLASH_SOURCES = $(ARDUINOPATH)/hardware/teensy/avr/libraries/SerialFla
WIRE_SOURCES = $(ARDUINOPATH)/hardware/teensy/avr/libraries/Wire
```

2. Provide generic rules for compilation :

```
CPPFLAGS = -Wall -O2 $(CPUOPTIONS) -MMD $(OPTIONS) -I.$(INCLUDE_FLAGS)\
      -ffunction-sections -fdata-sections

build/%.o: %.c
      $(CC) $(CPPFLAGS) -c -o $@  $^
```

3. Additionnal small tricks from existing makefile (.S file and linker script)

```
      LIBPATH = $(ARDUINOPATH)/hardware/teensy/avr/cores/teensy4
      MCU_LD = $(LIBPATH)/imxrt1062.ld
```

4. use `teensy_loader` to load hex file on teensy

Example of basic teensy-makefile project

- Download and untar the `teensy_makefile.tar` at the end of *embedded system basics* lecture on `embaudio` web site (`https://embaudio.grame.fr/lectures/embedded/`)
- `tar xvf teensy_makefile.tar`
- Go in the directory and modify the Makefile by :
  - indicating the location of arduino
  - indicating the location of MyDsp library

```
ARDUINOPATH=/home/trisset/technical/teensy/arduino-1.8.19
MYDSPPATH = /the/place/where/you/downloaded/MyDsp/library/mydsp/src
```

- Have a look at `main.cpp`
- try `make` and check that LED is blinking

Table of Contents

Peripheral programming

- Peripherals are (nowadays) all programmed with *memory map*
  - Each peripheral contains configuration registers
  - These registers are *mapped* to special addresses in the memory
- Example : hardware multiplier of MSP430
  - Registers mapped between adresses `0x0130` et `0x013F`
  - Writing at adresse `0x130`, writes first operand
  - Writing at `0x138`, writes second operand and start the multiplication
  - The result is accessible by reading at address `0x013A` (on 32 bits)

MSP430 example of peripheral memory mapping

```
int main(void) {
    int i;
    int *p,*res;

    p=0x130;
    *p=2;
    p=0x138;
    *p=5;
    res=0x13A;
    i=*res;

    nop();
}
```

```
int main(void) {
    int i;
    int *p,*res;

    __asm__("mov #304, R4");
    __asm__("mov #2, @R4");
    // p=0x130;
    // *p=2;
    __asm__("mov #312, R4");
    __asm__("mov #5, @R4");
    // p=0x138;
    // *p=5;
    __asm__("mov #314, R4");
    __asm__("mov @R4, R5");
    // res=0x13A;
    //i=*res;

    nop();
}
```

Use of Macros for Code Clarity

```
int main(void)   {
    int i;
    int *p,*res;

    p=0x130;
    *p=2;
    p=0x138;
    *p=5;
    res=0x13A;
    i=*res;

    nop();
}
```

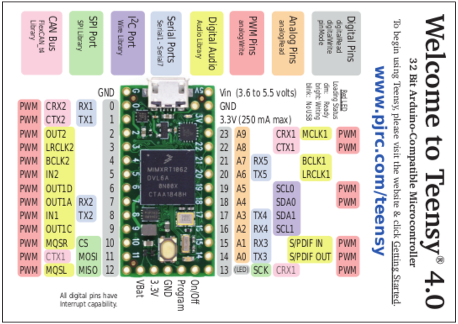
```
#include <themagicmacrofile.h>

int main(void) {
    int i;

    MULOP1=2;
    MULOP2=5;
    i=MULRES;

    nop();
}
```

Most basic peripheral : GPIO



- Teensy 4.0 has 40 physical I/O pad
- Some of them can be used for analog input or PWM output
- Digital I/O pins can be configured :
  - as GPIO or for trigerring a peripheral
  - GPIO can be configured
    - As input or output
    - Pulled up, pulled down, or not
    - Interrupt enable

Could you write the “blink” example ?

- The LED is connected to a teensy GPIO
- Blinking the LED is done using the following code :

```
// Pin 13 has an LED connected on most Arduino boards.
int led = 13;

void setup() {
  pinMode(led, OUTPUT);
}

void loop() {
  digitalWrite(led, HIGH);
  delay(1000);
  digitalWrite(led, LOW);
  delay(1000);
}
```

How to blink the LED on teensy

- Identify IO port connected to LED : teensy schematics (end of page `https://www.pjrc.com/store/teensy40.html`)

→ I/O pin number 13

- Configure I/O 13 in output mode : `pinMode()` function (see `https://www.pjrc.com/teensy/td_digital.html`)
- Write 1 or 0 at IO 13 port address : `digitalWrite()` function (see also `https://www.pjrc.com/teensy/td_digital.html`)

```
const int ledPin = 13;
  pinMode(ledPin, OUTPUT);
  while (1) {
    digitalWrite(ledPin, 1);
    delay(100);
    digitalWrite(ledPin, 0);
    delay(100);
  }
```

Better with macros...

```
const int ledPin = LED_BUILTIN;
  pinMode(ledPin, OUTPUT);
  while (1) {
    digitalWrite(ledPin, HIGH);
    delay(100);
    digitalWrite(ledPin, LOW);
    delay(100);
  }
```

in `$ARDUINOPATH/hardware/teensy/avr/cores/teensy4/pins_arduino.h`

```
    #define LED_BUILTIN   (13)
```

in `$ARDUINOPATH/hardware/teensy/avr/cores/core_pins.h`

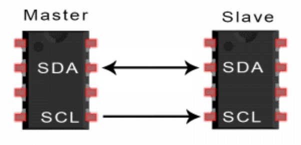
```
    #define HIGH 0x1
    #define LOW  0x0
```

How to configure a peripheral

- All peripherals (Timers, ADC, USB, ETH, etc.) are **dedicated circuits.**
- These circuits can be configured by a set of **registers**
- Each register has its own **address** (i.e. address within the peripheral) specified in peripheral datasheet.
- Two ways of writing these registers :
    - Memory map : the register corresponds to an address in Memory (see previous MSP430 multiplier example)
    - Use a serial protocol (I2C, SPI, . . .) to access the registers of the Peripheral

A (small) zoom on I2C

- I2C is a master/slave *synchronous* serial communication protocol
- It is used to communicate on both direction (R/W) bytes between master and slave
- *Synchronous* means that the clock synchronizing master and slave is sent by the master : no need of an agreement on transmission rate as in asynchronous protocol (such a UART : Universal Asynchronous Receiver Transmitter)
- I2C uses two wires : SCL (clock) and SDA (data)



I2C in brief (from SSM2603 codec doc)

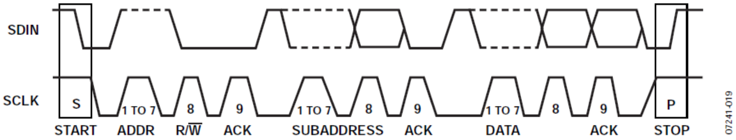


Figure 28. 2-Wire I²C Generalized Clocking Diagram

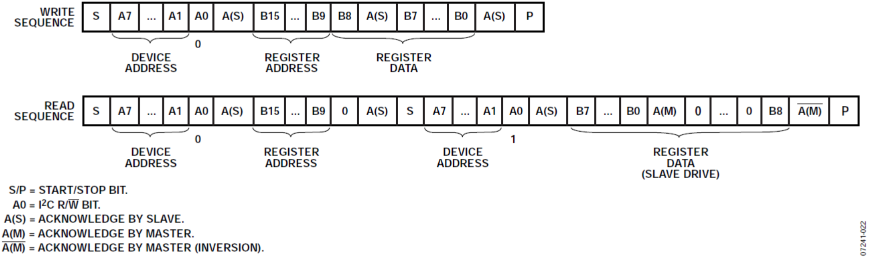


Figure 29. I²C Write and Read Sequences

How to use I2C on teensy 4.0

1. Learn I2C protocol (`https://fr.wikipedia.org/wiki/I2C`)
2. Read the teensy I2C documentation (`https://www.pjrc.com/teensy/td_libs_Wire.html`)
   - Teensy uses a arduino library (`Wire`) which provides higher level API, such as a serial device.
   - Example : from arduino

     `Examples -> i2C_T3 -> basic_master`

```
#include <Wire.h>
[...]
// Setup for Master mode, pins 18/19, external pullups, 400kHz,
Wire.begin(I2C_MASTER, 0x00, I2C_PINS_18_19, I2C_PULLUP_EXT, 400000);
[...]
// Print message
Serial.printf("Sending to Slave: '%s' ", databuf);
// Transmit to Slave
Wire.beginTransmission(0x66);   // Slave address
Wire.write(databuf,strlen(databuf)+1); // Write to I2C Tx buffer
Wire.endTransmission();           // Transmit to Slave

[...]
```

Table of Contents

Interrupt mechanism principle

- By default, the program `main` is executed infinitely, it generally contains an infinite loop that never ends.
- The processor can receive *interrupts* at any time (*hardware interrupts*).
- An interrupt can be sent by a peripheral of the micro-controller (timer, radio chip, serial port, etc...), or received from outside (on a GPIO) like the `reset` for example.
- It is the programmer who configures the peripherals (for example the timer) to send an interrupt on certain events
- It is a common naming habit to say that Interrupts arrive on a *port* of the micro-controller.
- An interrupt is processed by a dedicated *interrupt service routine* (ISR).
- Each interrupt has its own ISR. it is a function written by the programmer which has some special properties.

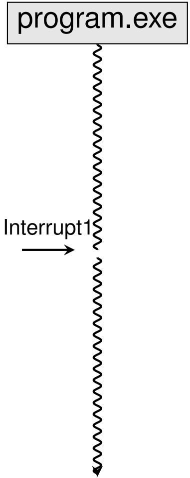
Processing an Interrupt

- Interrupts (i.e. “hardware interrupts”) are essential for the operation of any computer.
- When an interrupt occurs, the microprocessor saves the current state of its running program :
  - all general registers
  - the status register
  - the program counter
- It then executes a specific piece of code to process this interrupt (interrupt handler or ISR)
- when the handler is finished, it restores the state of the processor and resumes execution of the interrupted program

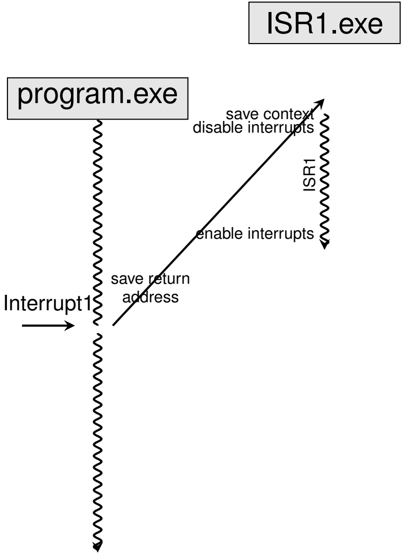
Interrupt Service Routine (ISR)

- The call to the interrupt handling routine is not exactly a function call like the others.
- It must be compiled a little differently, so it is usually identified by a *pragma* for the compiler. Example for `gcc` :
  `interrupt(PORT1_VECTOR)port1_irq_handler(void)`
- an interrupt handler can itself be interrupted or not by another interrupt (interrupt priority).
- User can write its own interrupt routines in C, the compilers provide facilities for this.
- On slightly more advanced systems, the ISR is provided by the programming environment which offers the user to write a function that will be called during the interruption : *callback* mechanism

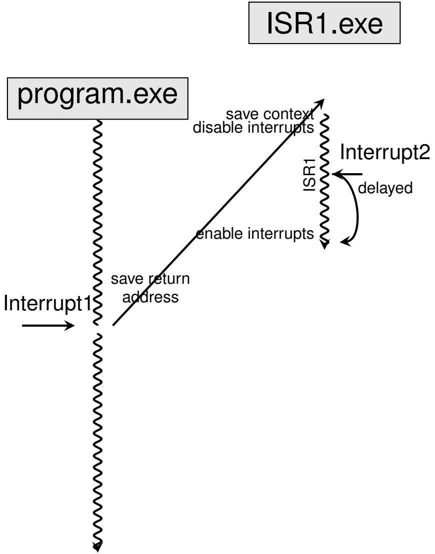
Interrupt mecanism



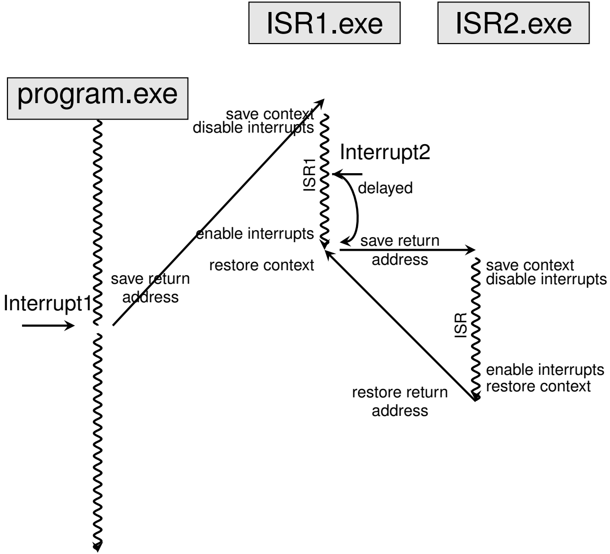
Interrupt mecanism



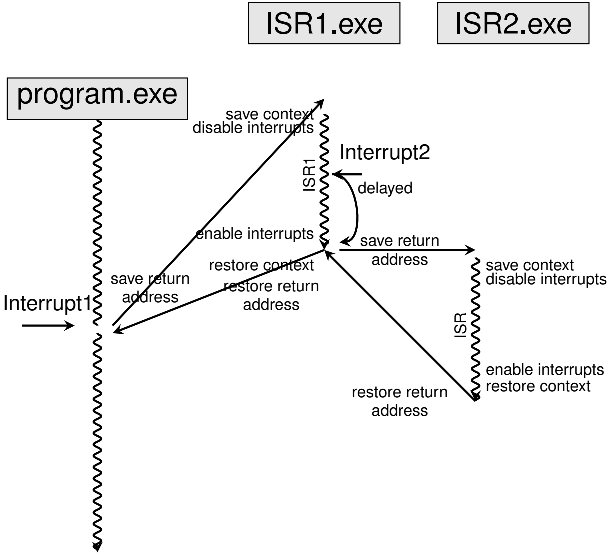
Interrupt mecanism



Interrupt mecanism

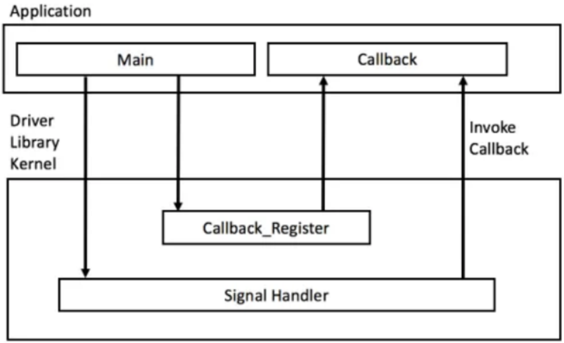


Interrupt mecanism



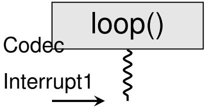
Callback mechanism

The Callback mechanism allows to define ISR behaviour is as a regular function.

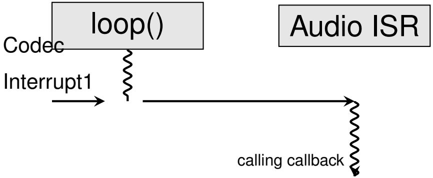


Interrupt Callback principle
(Image Source : Reusable Firmware Development book)

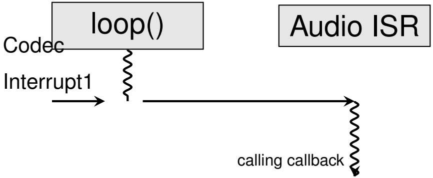
Audio Callback



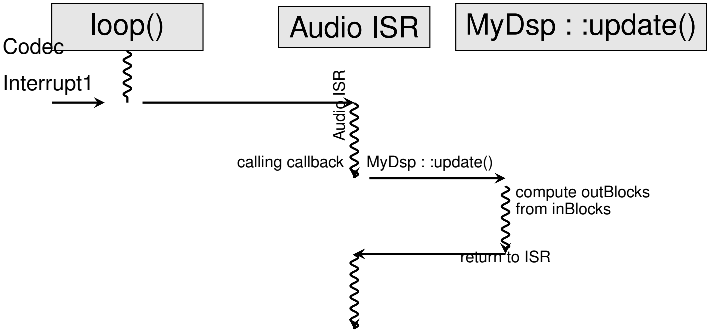
Audio Callback



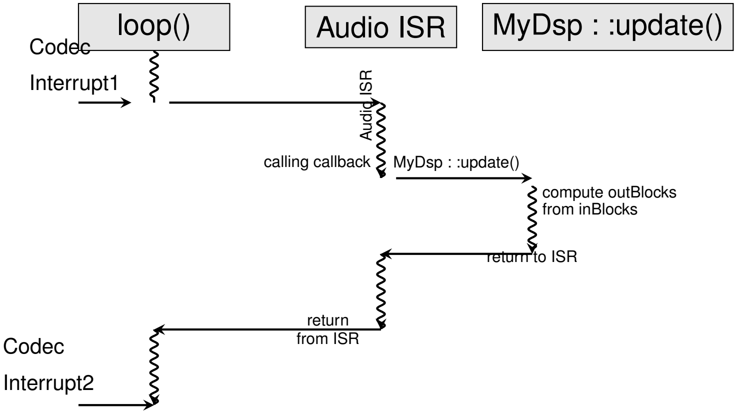
Audio Callback



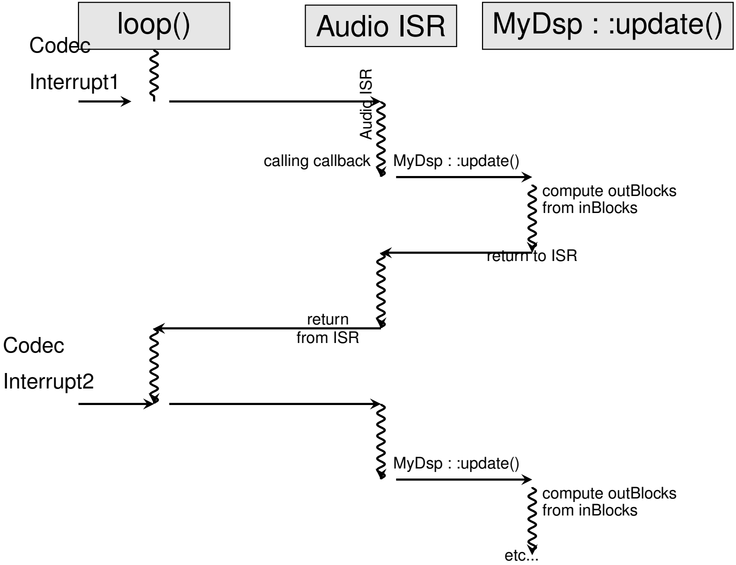
Audio Callback



Audio Callback



Audio Callback



Callback mecanism

- A callback mecanism is used to allow the user to write its own ISR function
- In primitive systems (bare metal) :
  - The compiler uses pragmas to distinguish between regular function and ISR.
  - Each interrupt has a dedicated number corresponding to its entry in the *interrupt vector table*
- In more elaborate systems :
  - A function pointer mecanism is used to *register* a user fonction as callback for a given interrupt
  - Examples on the teensy : intervalTimer
  - Examples on the teensy : the audio callback (`void MyDsp::update(void)`

Timer example

- Teensy provide the `intervalTimer` object (`https://www.pjrc.com/teensy/td_timing_IntervalTimer.html`) dedicated to provide *regular interrupts*.

```
// Create an IntervalTimer object
IntervalTimer myTimer;
```

- At timer initialization :
  - Set the frequency of interrupts (e.g. every 150 ms)
  - Register the callback function (e.g. `blinkLED`)

```
myTimer.begin(blinkLED, 150000)
```

- callback function (i.e. `blinkedLED`) must have fixed type : `void blinkedLED(void):`

Hands on

- As explained on Embaudio web site (lecture9), from the `teensy_example`
    - Create a `teensy_led` example that blinks the led with the `delay()` function.
    - Create a `teensy_timer` example that blinks the led with a timer.
    - Create a `teensy_serial` example that blinks the led with a timer and prints out on UART port every seconds, the number of blinks occured since the beguinning.
    - download the `teensy_audio` from the embaudio web site, run it and make it *click* by adding a `delay(10)` in the timer callback