

Dokumentation des Projekts "Sequenzer mit Tonerzeugung" im Modul Programmiertechnik

Daniel Raab und Konstantin Schmidt

22. Juli 2018

Inhaltsverzeichnis

1 Vorwort	2
2 Struktur	2
2.1 Grundstruktur	2
2.2 Modulstruktur	4
3 Module	4
3.1 Hauptprogramm (<i>main</i>)	4
3.1.1 int <i>main</i> ()	5
3.1.2 void <i>button_SW1_pressed</i> ()	5
3.1.3 void <i>button_SW2_pressed</i> ()	5
3.1.4 void <i>button_SW3&4_pressed</i> ()	5
3.1.5 void <i>encoder_left&right</i> ()	5
3.1.6 void <i>potentionmeter_turned</i> ()	5
3.1.7 void <i>play_next_step</i> ()	6
3.1.8 void <i>update_display</i> ()	6
3.2 Sequenz (<i>sequence</i>)	6
3.2.1 enum <i>Tone_length</i>	6
3.2.2 struct <i>Step</i>	7
3.3 Display (<i>lcd</i>)	7
3.3.1 void <i>lcd_init</i> ()	7
3.3.2 void <i>lcd_clear</i> ()	7
3.3.3 Schreibfunktionen	7
3.3.4 void <i>lcd_set_cursor</i> (unsigned char <i>x</i> , unsigned char <i>y</i>)	8
3.3.5 void <i>write_pitch</i> (unsigned int <i>pitch</i>) . . .	8

3.3.6	void write_tone_length(enum Tone_length tone_length)	8
3.3.7	void write_tempo(unsigned int tempo)	8
3.4	Eingabe (<i>input</i>)	9
3.4.1	void input_init()	9
3.4.2	Flaggen	9
3.4.3	<u>interrupt</u> void P2_ISR()	9
3.4.4	<u>interrupt</u> void ADC10_ISR()	10
3.5	Ton (<i>tone</i>)	10
3.5.1	void ton_init()	10
3.5.2	void ton(unsigned int pitch, unsigned long Dauer)	11
3.5.3	void update_tempo(unsigned int tempo)	11
3.5.4	<u>interrupt</u> void ton_umschalten(void)	11
3.6	LED-Matrix (<i>led_matrix</i>)	12
3.6.1	void i2c_init()	12
3.6.2	void i2c_write_byte(unsigned char i2c_address, unsigned char expander_reg, unsigned char data)	12
3.6.3	void led_on(unsigned char led_nr)	12

1 Vorwort

Dies ist die technische Dokumentation des Projekts im Modul Programmiertechnik von Daniel Raab und Konstantin Schmidt. Es wird der Aufbau der Software beschrieben und dargestellt, sowie die einzelnen Teilmodule. Das Projekt wurde für das *MSP Education Board 3.0* der HTWK Leipzig entworfen und implementiert. Der Quellcode wurde in C mithilfe des *Code Composer Studio 6* geschrieben.

2 Struktur

2.1 Grundstruktur

Das Programm besitzt keinen sequentiellen Ablauf, da ein auf Interrupts (engl. Unterbrechungen) basierender Ansatz gewählt wurde, um alle Funktionen gut implementieren zu können.

Der Grundablauf ist in Abbildung 1 zusehen. Es wird zuerst der Mikrorechner und alle benötigten Systeme initialisiert und dann in den Interruptgesteuerten Betrieb übergegangen. Der Prozessor wird also in den Schlaf-

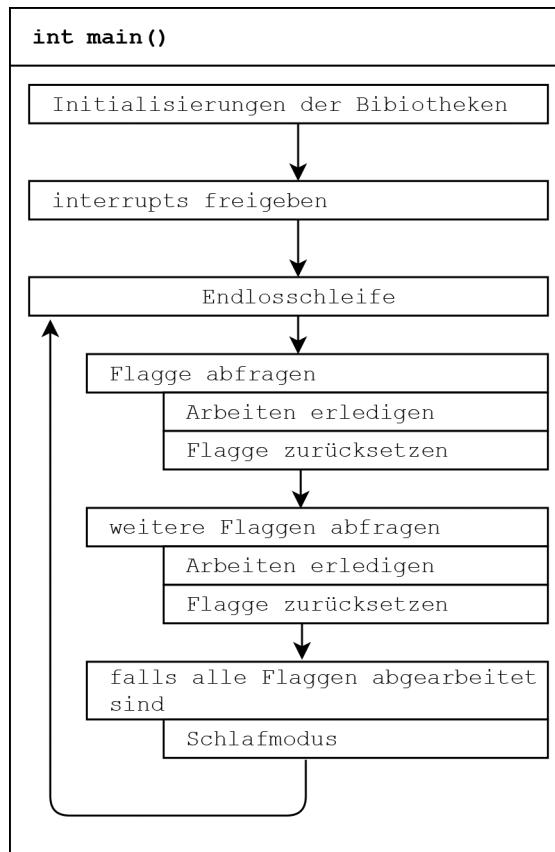


Abbildung 1: Der Grundablauf schematisch dargestellt

bzw. Low-Power-Modus versetzt und reagiert ab sofort nur noch auf auftretende Interrupts. Die aufgerufene Interrupt Service Routine (ISR) soll möglichst schnell abgearbeitet werden, also wird meist lediglich das Interrupt ausgewertet und dann eine entsprechende Flagge gesetzt. Dann wird der Schlafmodus verlassen und die ISR beendet. Durch das Verlassen des Schlafmodus werden nun in der `main()`-Funktion die nächsten Befehle ausgeführt. Hier werden nun die Flaggen abgefragt und die entsprechenden Arbeiten erledigt. Sobald alle Flaggen abgefragt und abgearbeitet sind, wird der Prozessor wieder in den Schlafmodus versetzt. Diese Abarbeitung in der `main()`-Funktion hat den großen Vorteil, dass lang andauernde Arbeiten, wie z.B. Ansteuerung des Displays mit Wartezeiten, durch Interrupts unterbrochen werden können, ohne dass erst auf die lange andauernden Anweisungen gewartet werden muss. Diese Blockierung der schnell abzuarbeitenden ISRs passiert, wenn alle Aufgaben in ISRs passieren und nicht in der `main()`-Funktion. Zeitkritische Aufgaben z.B. Tastendrücke oder Töne könnten nicht richtig oder zeitnah

ausgelesen bzw. ausgegeben werden. Eine Ausnahme dieser Regel betrifft die `play_tone`-Funktion der Tonbibliothek, siehe *Abschnitt 3.5*.

2.2 Modulstruktur

Die Programmierung des Mikrorechners erfolgte in Teilmodulen, die sich den verschiedenen Teilbereichen des Projekts widmen. Diese Teilmodule des Projekts wurden erst für sich entwickelt und sobald ein passabler Grad der Implementierung erreicht wurde in das Gesamtprojekt eingebunden und im Verbund getestet. Das Projekt wurde in folgende Module zerlegt:

- Sequenz-Datenstrukturen (*sequence.h*)
- Display-Ansteuerung (*lcd.h*)
- Eingaben, also Taster und Drehgeber (*input.h*)
- Tonerzeugung (*tone.h*)
- LED-Matrix (*led_matrix.h*)

Die Steuerung des Programms passiert in der *main.c* über die Funktionen, die auf die Eingaben des Benutzers reagieren. Dort werden alle Bibliotheken und Module eingebunden und zum Projekt vereinigt.

3 Module

Hier werden die Module genauer in ihrem Aufbau und ihrer Struktur beschrieben. Die Modulnamen für die tatsächlichen Dateien sind in Klammern angegeben und umfassen die entsprechende *.h und *.c Dateien.

3.1 Hauptprogramm (*main*)

Die Datei *main.c* ist der Einstiegspunkt des Programms und enthält die Funktion `int main()`. Es werden alle unserer Bibliotheken inkludiert und die Funktionen deklariert, die für die Bedienung der Ereignisse aus den Interrupts zuständig sind. In der Variable `mode` wird gespeichert, in welchem der beiden möglichen Modi sich der Sequenzer gerade befindet.

`outdated_display` ist eine Flagge, die signalisiert, ob das LCD aktualisiert werden muss. Sie wird hier am Ende jeder Funktion gesetzt, weil jede Funktion Einfluss auf die Displayausgabe hat.

3.1.1 int main()

Gleich zu Beginn werden die einzelnen Systeme initialisiert. Dann werden die Startwerte der Variablen zugewiesen. Dazu gehört auch die Startmelodie, für die jedem der 16 Schritte eine Tonhöhe und eine Tonlänge zugewiesen werden muss. Nachdem LCD und LED-Matrix gestartet wurden, werden die Interrupts freigegeben und es startet die Hauptschleife und damit der Interrupt-basierte Ablauf, wie im *Abschnitt 2.1 Grundstruktur* und in *Abbildung 1* erläutert.

3.1.2 void button_SW1_pressed()

Wenn der Taster **SW1** gedrückt wird, wird der Modus gewechselt. Falls vom Bearbeitungsmodus in den Abspielmodus gewechselt wird, wird die Tonerzeugung gestartet und `current_step` auf 0 gesetzt, damit die Sequenz vom Anfang an abgespielt wird. `update_tempo(tempo)` wird in *Abschnitt 3.5* erklärt. Die LED-Matrix wird auch aktualisiert.

3.1.3 void button_SW2_pressed()

Mit Taster **SW2** wird im Bearbeitungsmodus die Tonlänge eingestellt. Sie wird mit jedem Tastendruck erhöht.

Wenn `sequence[current_step].tone_length = full` erreicht ist, wird die Tonlänge beim nächsten Tastendruck wieder auf `pause` gesetzt.

3.1.4 void button_SW3&4_pressed()

Taster **SW3** und **SW4** verringern und erhöhen das aktuelle Tempo. Wenn die Unter- oder Obergrenze erreicht ist, bewirkt ein weiterer Tastendruck keine Veränderung.

3.1.5 void encoder_left&right()

Mit dem Drehgeber (hier **Encoder**) wird im Bearbeitungsmodus die Tonhöhe des zu bearbeitenden Schritts eingestellt. Eine Drehung nach links bewirkt eine Verringerung der Tonhöhe und eine Drehung nach rechts eine Erhöhung. Wiederum wird der Wert nur verändert, wenn weder Ober- noch Untergrenze erreicht ist.

3.1.6 void potentiometer_turned()

Im Bearbeitungsmodus kann mit dem **Potentiometer** der zu bearbeitende Schritt ausgewählt werden. Dazu wird `pot_value` der aktuellen Schritt-

nummer `current_step` zugewiesen. Die LED-Matrix wird aktualisiert und die aktuelle Tonhöhe des Schritts wird kurz angespielt.

3.1.7 `void play_next_step()`

Diese Funktion ist dafür verantwortlich, dass die Sequenz im Abspielmodus kontinuierlich abgespielt wird. Abhängig von der Tonlänge wird ein Ton mit der Tonhöhe des aktuellen Schritts abgespielt. Die übergebene Tonlänge wird in Hundertstelsekunden übergeben. Da diese abgängig vom Tempo ist, muss sie hier berechnet werden. Der Faktor ergibt sich daraus, dass das Tempo in BPM (engl. beats per minute; *Schläge pro Minute*) angegeben ist. Ein Ton mit der Länge `full` dauert also $\frac{1}{\text{tempo}} \cdot 60$ Sekunden. Wenn man das mit 100 multipliziert, kommt man auf das Ergebnis in Hundertstelsekunden. Die anderen Tonlängen sind lediglich mit 0.25, 0.5 und 0.75 multipliziert.

$$\text{Tonlänge full in Hundertstelsekunden} = \frac{1}{\text{tempo}} \cdot 60 \cdot 100 = \frac{6000}{\text{tempo}}$$

Die LED-Matrix wird wieder aktualisiert und die Schrittnummer wird inkrementiert, wenn der letzte Schritt noch nicht erreicht ist. Falls doch, wird `current_step` wieder nullgesetzt, sodass die Sequenz wieder von vorne beginnt.

3.1.8 `void update_display()`

Immer, wenn die Flagge `outdated_display` beim Durchlaufen der Schleife in `int main()` aktiviert ist, wird diese Funktion aufgerufen. Hier werden alle Funktionen aufgerufen, die für das Beschreiben des LC-Displays zuständig sind.

3.2 Sequenz (*sequence*)

Diese Datei beinhaltet die Datenstrukturen und globalen Variablen der Sequenz. Globale Variablen sind das Tempo in BPM (`unsigned int tempo`) und die aktuelle Position innerhalb der Sequenz (`unsigned int current_step`).

3.2.1 `enum Tone_length`

Diese Enumeration definiert die fünf verschiedenen Tonlängen, die ein Schritt der Sequenz haben kann. Dies sind Pause - `pause`, Viertel - `quarter`, Halb - `half`, Dreiviertel - `three_quarters` und Ganz - `full`. Dies röhrt daher, dass die Sequenz in verschiedenen Tempi abgespielt werden kann und die Tonlänge deswegen relativ zum Tempo ist - wie bei geschriebener Musik.

3.2.2 struct Step

Diese Struktur beschreibt einen Schritt einer Sequenz und die Sequenz besteht aus 16 solcher Schritte: `struct Step sequence[16]`. Jeder Schritt besitzt dabei Tonlänge als `enum Tone_length tone_length` und Tonhöhe als `unsigned int pitch`. Die Tonhöhe ist eine Ganzzahl, die den Abstand in Halbtorschritten von C1 angibt. Mit zwölf Halbtönen pro Oktave bedeutet das: C4 entspricht beispielsweise 36 oder #G5 entspricht 55.

3.3 Display (*lcd*)

Die Display Bibliothek steuert das LC-Display und stellt verschiedene Methoden zur Beschreibung und Steuerung dessen zur Verfügung. Außerdem wurden besondere Methoden für dieses Projekt implementiert, die die Anzeige an vorher festgelegten Stellen mit bestimmten Werten beschreibt: `write_pitch`, `write_tone_length` und `write_tempo`.

Alle Funktionen des Displays benötigen (relativ) viel Zeit, um mit dem Display-Controller zu kommunizieren und sollten demnach nicht in Interrupt Service Routinen benutzt werden.

3.3.1 void lcd_init()

Das LC-Display wird gestartet und initialisiert über die Funktion `void lcd_init()`. Es wird dann die Initialisierungssequenz für das Display gesendet und es für die weitere Verwendung konfiguriert. Es wird das Display geleert und sowohl der Cursor als auch Blinken ausgeschaltet.

3.3.2 void lcd_clear()

Eine Funktion, die alles Angezeigte vom Display löscht und den Cursor zurück in die obere, linke Ecke (0, 0) setzt. Sollte nicht zu häufig verwendet werden, da es sonst zu störendem Flackern der Anzeige kommt.

3.3.3 Schreibfunktionen

Als grundlegenden Schreibfunktionen, um das Display zu verwenden, sind

- `void lcd_write(unsigned char character)`,
- `void lcd_write_string(unsigned char string[])` und
- `void lcd_write_int(unsigned int number, int digits)`

für die verschiedenen wichtigen Datentypen implementiert. Es können einzelne Zeichen (`unsigned character`) gesendet werden, sowie ganze strings, also Felder von Zeichen, die jedoch mit dem terminierendem Zeichen `0x0` oder `'\0'` abgeschlossen werden müssen. Dies passiert automatisch, wenn ein string mit doppelten Anführungszeichen angegeben wird. Um Zahlen (`int`) anzuzeigen muss noch ein zweites Argument übergeben werden, das die Anzahl der Stellen bestimmt. Falls die Zahl weniger Stellen besitzt, wird der Rest mit Nullen aufgefüllt, also wird beispielsweise bei der Zahl 45 und drei Stellen `[045]` angezeigt.

All diese Funktionen schreiben immer ab der aktuellen Position des Cursors und der Cursor ist nach dem Schreiben beim nachfolgenden Zeichen des letzten neu auf das Display geschriebenen Zeichens.

3.3.4 `void lcd_set_cursor(unsigned char x, unsigned char y)`

Der Cursor kann mithilfe der Funktion `void lcd_set_cursor(unsigned char x, unsigned char y)` an eine beliebige Stelle im Display gesetzt werden. `x` steht für die Position innerhalb der Zeile (beginnend mit 0, also maximal 15) und `y` für eine der beiden Zeilen (0 oder 1).

3.3.5 `void write_pitch(unsigned int pitch)`

Diese Funktion wandelt die als Zahl angegebene Tonhöhe (wie in X erläutert) in die gewohnte Notation (Vorzeichen, Notename und Oktavnummer, z.B. `[#G4]`) um. Sie wird dann auf dem Display an der festgelegten Stelle (0, 0 - linke obere Ecke) für die Tonhöhe angezeigt.

3.3.6 `void write_tone_length(enum Tone_length tone_length)`

Mithilfe dieser Funktion wird auf dem Display ab der festgelegten Stelle 0, 1 ein Balkendiagramm als Darstellung der Tonlänge angezeigt. Übergeben wird die in `sequence.h` festgelegte Enumeration dafür und je nach Wert zeigt der Balken keine bis vier Segmente: beispielsweise `[000.]` oder `[....]`.

3.3.7 `void write_tempo(unsigned int tempo)`

Diese Funktion schreibt das Tempo als Zahl in BPM an die dafür festgelegte Stelle auf dem Display: 9, 0.

3.4 Eingabe (*input*)

In der Eingabe-Bibliothek werden die Taster, das **Potentiometer** und der **Encoder** ausgelesen und dem Rest des Programms in Form von Flaggen zur Auswertung zur Verfügung gestellt. Es werden die Interrupts an Port 2 empfangen und ausgelesen, die jeweilige Flagge gesetzt und der Low-Power-Modus beendet, damit die Flaggen sofort ausgelesen und darauf reagiert werden kann.

3.4.1 `void input_init()`

Die Funktion initialisiert den Port 2 so, dass die Interrupts ausgelöst werden und damit auf die Eingaben reagiert werden kann. Außerdem wird der Analog-Digital-Umsetzer (engl. ADC) passend konfiguriert, um das **Potentiometer** auszulesen. Dafür wird das Capture/Compare-Register 0 (CCR0) des Timer A verwendet, das den ADC hunderte Male pro Sekunde zum Auslesen des gerade anliegenden Potentiometerwertes anregt. CCR0 wird dementsprechend konfiguriert und das Auslesen wird in `__interrupt void CCR0_ISR()` angeregt.

3.4.2 Flaggen

Es gibt für jeden Taster eine Flagge (`bool button_SW4` bis `bool button_SW1` entsprechend der Tasternummer), eine Flagge `potentiometer_new` für einen neuen Wert des Potentiometers und zwei Flaggen `bool encoder_l` und `bool encoder_r` für die beiden Drehrichtungen des Encoders.

3.4.3 `__interrupt void P2_ISR()`

Die ISR für Port 2, die aufgerufen wird, sobald ein Taster oder der **Encoder** betätigt werden. Dort wird überprüft, ob der **Encoder** und wenn ja, in welche Richtung er gedreht wurde. Da die Taster prellen, dürfen die Tastendrücke nicht sofort ausgewertet werden, da ansonsten bei jedem Tastendruck fälschlicherweise viele Tastendrücke erkannt werden würden. Deswegen wird über die Wartefunktion `void debounce_delay()` eine passende Wartezeit vor der Auswertung realisiert. Je nachdem, welcher Taster oder welche **Encoder**-Richtung erkannt wurde, wird die entsprechende Flagge gesetzt und der Prozessor aus dem Schlafmodus aufgeweckt, um in der `main()`-Funktion darauf zu reagieren.

3.4.4 __interrupt void ADC10_ISR()

Diese Interrupt Service Routine wird immer dann aufgerufen, sobald ein neuer Wert vom ADC gemessen wurde. Dies bedeutet aber nicht notwendigerweise eine Änderung des Wertes also eine Benutzereingabe. Es muss also ein Vergleich mit dem vorherigen Wert stattfinden, deswegen gibt es zwei Variablen: `pot_value` und `pot_value_old`. Der ADC gibt 10bit-Werte aus, für das Projekt wollen wir aber nur zwischen 16 Werten unterscheiden (die Schritte der Sequenz). Dies entspricht einer 4bit-Zahl, also wird der ADC-Wert um 6 Werte nach rechts bit-verschoben und dann erst in `pot_value` gespeichert. Falls dieser Wert nicht mit `pot_value_old` übereinstimmt wird die Flagge `potentiometer_new` gesetzt und der Schlafmodus verlassen, um darauf zu reagieren.

3.5 Ton (*tone*)

In diesem Modul sind alle Funktionen hinterlegt, die für die Ton- und Melodieerzeugung notwendig sind. Die Tonerzeugung wird realisiert, indem die Ausgangseinheit von Capture/Compare-Register 2 (CCR2) im Toggle-Modus ist und direkt Portpin 2.4 ansteuert. Am Port 2.4 ist der Piezolautsprecher des Education Boards angeschlossen. Jedes mal, wenn der Stand des Timers A mit dem Wert von CCR2 übereinstimmt, toggelt die Ausgangseinheit und der Ausgangspegel von Portpin 2.4 wechselt. Somit schaltet auch die Membran des Piezos nach dieser Zeit um. Daraus folgt, dass die Zeit, nach der die Ausgangseinheit toggelt, die halbe Periodendauer des Tons ist, der vom Piezo ausgegeben wird. Der Timer A läuft im Continuous-Up-Modus, sodass der Wert, der die Periodendauer bestimmt, der Wert von CCR2 ist. Genauer gesagt ist es die Wertdifferenz zwischen altem und neuem Wert des Registers. Sein Wert wird nämlich jedes mal, wenn eine Übereinstimmung mit dem Timer A erreicht ist, im Interrupt um den Wert erhöht, der der gewünschten halben Periodendauer entspricht.

3.5.1 void ton_init()

Für die Tonerzeugung und auch für das Starten des nächsten Schritts wird Timer A verwendet. Dafür werden hier die nötigen Grundeinstellungen vorgenommen. Timer A sowie CCR2 erhalten die Interruptfreigabe und die Ausgangseinheit von CCR2 wird direkt mit dem Piezo-Lautsprecher am Portpin 2.4 verbunden. Dann wird Timer A im Continuous-Up-Mode gestartet. Damit ist die Initialisierung abgeschlossen.

3.5.2 void ton(unsigned int pitch, unsigned long Dauer)

Diese Funktion wird aus der Funktion `void play_next_step()` (siehe Abschnitt 3.1.7) heraus aufgerufen. Das passiert innerhalb der ISR `__interrupt void ton_umschalten(void)`. Da die folgende Berechnung von `pitch` relativ viel Zeit beansprucht, ist es für die flüssige Abfolge des Programms wichtig, dass innerhalb dieser Funktion weitere Interrupts zugelassen werden. Aus der übergebenen Tonhöhe wird nun die zugehörige Frequenz des Tons berechnet.

$$\text{pitch} = 2^{\frac{\text{pitch}-57}{12}} \cdot 440$$

Das Gleichheitszeichen ist hier als Zuweisung zu verstehen, so wie es auch im Quellcode angewendet wird. Das bedeutet, dass der Inhalt der Variable `pitch` von einer ganzen Zahl, die den Ton auf der Notenskala repräsentiert, in eine Frequenz umgewandelt wird, die für die Tonerzeugung genutzt werden kann.

`ccn` ist die Anzahl der Timerperioden, deren Dauer die halbe Periodendauer der Frequenz ist. Die halbe Periodendauer erhält man, wenn man das Reziproke der doppelten Frequenz bildet. Dieser Wert muss dann noch mit 32768 multipliziert werden, weil der Timer pro Sekunde um 32768 hochzählt. `t` steht für die Anzahl der Perioden, für die der Ton erklingen soll. Die Variable `steht` also für die Dauer des Tons.

Der erhaltene Wert für `ccn` wird nun auf den aktuellen Stand des Timers A aufaddiert und das Ergebnis in CCR2 geschrieben. Die Tonerzeugung wird gestartet, indem für CCR2 in den Toggle-Modus geschaltet und die Interruptfreigabe erteilt wird. Damit auch neue Töne gestartet werden können, wird auch für CCR1 die Interruptfreigabe erteilt.

3.5.3 void update_tempo(unsigned int tempo)

Hier wird die Anzahl der Timerperioden ermittelt, die erreicht sein muss, damit der nächste Schritt abgespielt wird. Das passiert abhängig vom Tempo. `step_CC_number` muss also immer neu berechnet werden, wenn das Tempo geändert wird.

3.5.4 __interrupt void ton_umschalten(void)

In einer `switch case`-Anweisung wird unterschieden, welche Capture/Compare-Einheit den Interrupt verursacht.

Im Fall 2 ist es Capture/Compare-Einheit 1. CCR1 wird erhöht, sodass wieder die Zeit bis zum nächsten Schritt ablaufen muss. In `play_next_step()` wird der aktuelle Schritt abgespielt. Weil sich beim Abspielen der Schritte

auch die Ausgabe auf dem LCD und der LED-Matrix ändert, wird nun die CPU aus dem Low-Power-Modus aufgeweckt.

Im Fall 4 hat Capture-Compare-Einheit 2 den Interrupt ausgelöst. Hier wird CCR2 so erhöht, dass der Interrupt das nächste mal nach der halben Periodendauer des Tons ausgelöst wird. Die Zählvariable m bestimmt, ob der Ton schon über seine bestimmungsgemäße Dauer gespielt wurde oder nicht.

3.6 LED-Matrix (*led_matrix*)

Diese Bibliothek steuert die LED-Matrix, die über die i2c-Schnittstelle an dem Mikrorechner angeschlossen ist. Die Matrix dient der visuellen Ausgabe der aktuellen Position innerhalb der Sequenz, indem eine der 16 LEDs leuchtet.

3.6.1 **void i2c_init()**

Diese Funktion initialisiert alle benötigten Komponenten: Port 3, um die Datenleitungen ansteuern zu können, das Universal Serial Communication Interface (USCI), das die Kommunikation mit der LED-Matrix über i2c erledigt und dann die Initialisierung der Steuerung der LED-Matrix.

3.6.2 **void i2c_write_byte(unsigned char i2c_address, unsigned char expander_reg, unsigned char data)**

Diese Funktion wird benötigt, um eine einzelnes Byte in den Speicher der Steuerung der LED-Matrix zu schreiben. Dabei muss zuerst die i2c-Adresse der LED-Matrix übergeben werden sowie das zu beschreibende Register und natürlich das zu schreibende Byte. Diese Funktion wird innerhalb des Moduls verwendet, um das Register GPIO zu beschreiben, das die digitalen Pins der LEDs steuert und damit bestimmt welche LEDs leuchten.

3.6.3 **void led_on(unsigned char led_nr)**

Diese Funktion wird verwendet, um auszuwählen welche der 16 LEDs leuchten soll. Die Funktion berechnet aus der übergebenen Zahl in welcher Spalte und Zeile sich die entsprechende LED befindet und berechnet dann mithilfe von bit-Verschiebungen das Datenbyte, das in das GPIO-Register (kontrolliert die Ausgänge an denen die LED-Matrix angeschlossen ist) geschrieben wird. Die Zahl darf dabei nur zwischen 0 und 15 liegen, ansonsten leuchtet keine der LEDs. Das Datenbyte besteht aus zwei Teilen, den Bits 0 bis 3 und Bit 4 bis Bit 7, die entsprechend die vier Zeilen und vier Spalten aktivieren. Nur die LEDs leuchten deren Spalte *und* Zeile aktiviert sind.