Versuch 4 – Interrupts

Dieser Versuch enthält zwei unabhängige Teile, die später nach Wunsch auch zusammengesetzt werden können. In beiden Teilen geht es um die Verwendung von Interrupts und den dazugehörigen Hilfsmitteln (Flags, Masken, ISR).
Bei dem ersten Teil liegt der Schwerpunkt auf dem Verständnis, **wie** eine Unterbrechung zustande kommt. Die tatsächliche Umsetzung (Code) ist dann sehr kurz. Lernziel ist Verwendung von Flags, Masken und des NVIC.
Bei dem zweiten Teil ist die Einstellung für die Anwendung bereits fertig und die einzige nötige ISR bereits vorhanden. Die ISR erfordert aber erheblich mehr Aufwand (Code). Lernziel ist die Verwendung bedingter Kompilierung, von lokalen statischer Variablen in einer ISR sowie von globalen Variablen zur Kommunikation mit der ISR.

1 Teil 1: Interrupts von GPIOs

Im zweiten Versuch wurde eine LED durch eine Taste gesteuert. Solange die Taste gedrückt war, leuchtete die LED, andernfalls blieb sie aus. Die Lösung bestand in einer Dauerschleife. Wie heißt dieses Verfahren und warum ist es im Allgemeinen keine gute Lösung?

Im ersten Teil des Versuchs soll die LED jetzt von den beiden Tasten über Interrupts gesteuert werden. Eine Taste schaltet die LED ein, die andere aus. Dabei sollen die Tasten jeweils eine eigene ISR auslösen, in der dann die gewünschte Aktion stattfindet. Die Endlosschleife bleibt leer. Der µC kann in dieser Zeit genauso gut schlafen.

1.1 Vorbereitung

Wechseln Sie in einen neuen, leeren Workspace. Importieren Sie dann aus dem Archiv *v4_vorbereitung.zip* die **drei** Projekte *lpc_chip_11uxx_lib*, *mch_aux_lib* und ***v4_p1***. Für diesen Versuch benötigen Sie das nur Piggyback ohne das externe Netzteil

1.2 Zeitlicher Signalverlauf

Beide Tasten sind einpolige Schließer auf Masse. Zur sicheren Erkennung wird also ein Pullup-Widerstand aktiviert. T1 (Boot) ist an PIO0_1 angeschlossen, T2 an PIO0_23. Für T1 wurde zudem auch der optionale Inverter eingeschaltet. Zeichnen Sie in Abbildung 1 den Signalverlauf (Werte 0 und 1) für die beiden GPIO ein. Zu Beginn sind beide Tasten nicht gedrückt.

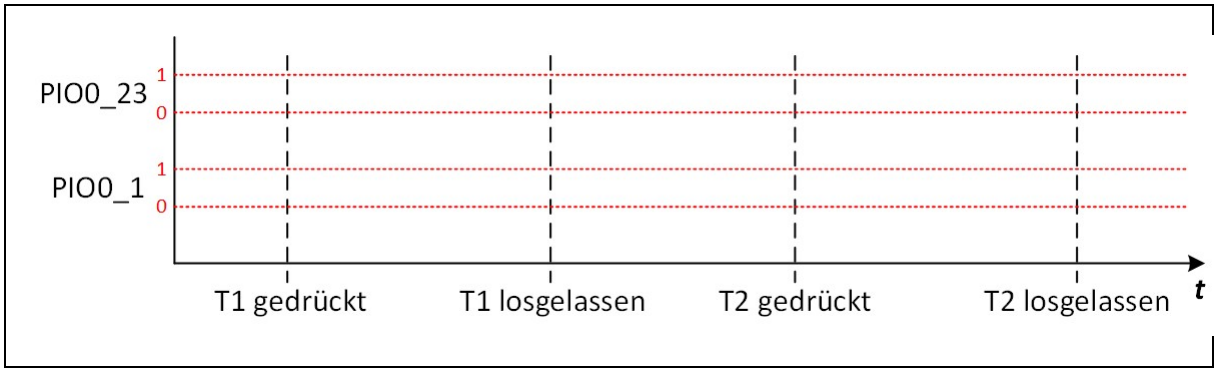

Abbildung 1: Signalverlauf am GPIO-Modul (im µC)

1.3 Ereignisauswahl

Jeder individuelle Interrupt an einem GPIO kann auf die Erkennung bestimmter Ereignisse eingestellt werden. Bei vielen heutigen µC, so auch bei dem µC aus dem Praktikum, können die in Tabelle 1 aufgeführten Bedingungen durch den Programmierer gewählt werden. Tritt die eingestellte Bedingung ein, dann wird das Flag für dieses Ereignis gesetzt. Das Flag muss später (nach Behandlung des Ereignisses) im Programm durch einen entsprechenden Befehl gelöscht werden. Es wird automatisch wieder gesetzt, sobald die eingestellte Bedingung wieder eintritt.

Bedingung am GPIO	Engl. Bezeichnung	T?
Solange der Wert 0 ist	level sensitive, low active	
Solange der Wert 1 ist	level sensitive, high active	
Nur an der steigenden Signalflanke	edge sensitive, rising edge	
Nur an der fallenden Signalflanke	edge sensitive, falling edge	
An beiden Signalflanken	edge sensitive, both edges	

Tabelle 1: Typischerweise einstellbare Bedingungen

Tragen Sie in der letzten Spalte (T?) der Tabelle 1 ein, welche Bedingung hier für welche Taste passt.

1.4 Signalverlauf im µC

Damit ein Ereignis an einem GPIO tatsächlich eine zugehörige ISR aufruft, müssen einige Module zusammenarbeiten und entsprechend eingestellt werden. Abbildung 2 zeigt, welchen Weg ein Signal von einem Pin durch verschiedene Module des µC nimmt, bevor es die zugehörige ISR auslösen kann.

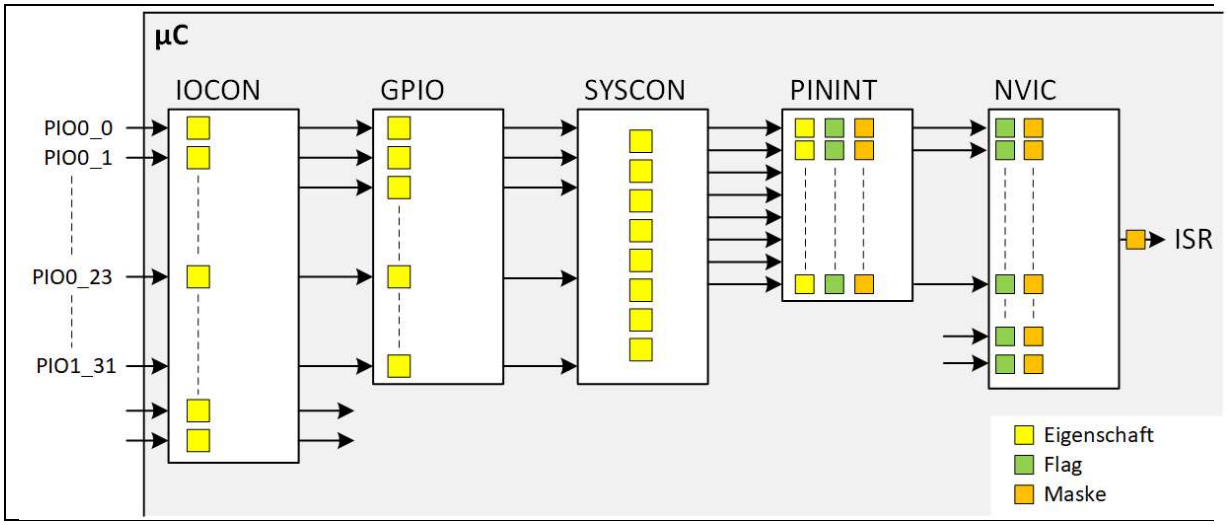

Abbildung 2: Signalverlauf durch beteiligte Module im µC

Zu jedem Modul werden im Folgenden auch Hinweise zur Programmierung gegeben, die Sie später bei der Programmierung nutzen können.

1.4.1 IOCON (Input Output CONtrol)

Dieses Modul ist bereits bekannt. Hier werden zunächst die elektrischen Eigenschaften des Pins eingestellt. Falls ein Pin mehr als eine Funktion ausführen kann, dann wird hier auch festgelegt, welche Funktion gerade aktiv ist. Bei dem Praktikums-µC können alle Pins, bei denen die elektrischen Eigenschaften eingestellt werden können, auch als GPIO verwendet werden. Es gibt viele µC, bei denen nicht alle Pins auch GPIOs sein können, daher die beiden unten eingezeichneten Signale, die nicht weiter zum GPIO-Modul führen.

Kapitel im UM	Chapter 7: LPC11U3x/2x/1x I/O configuration
Dateien in der Bibliothek	*iocon.h*
Handle	*LPC_IOCON*
Nummerierung der Pins	port, pin
Initialisierung	Notwendig, Modul ist zunächst abgeschaltet

1.4.2 GPIO (General Purpose Input Output)

Auch dieses Modul ist bereits bekannt. Hier wird die Richtung eingestellt. Im Fall eines Ausgangs kann der Ausgabewert festgelegt werden und im Fall eines Eingangs kann der aktuell anliegende Wert eingelesen werden.

Kapitel im UM	Chapter 9: LPC11U3x/2x/1x GPIO
Dateien in der Bibliothek	*gpio.h*
Handle	*LPC_GPIO*
Nummerierung der Pins	port, pin
Initialisierung	Notwendig, Modul ist zunächst abgeschaltet

1.4.3 SYSCON (SYStem CONtrol)

Dieses Modul ist neu. Er vereint viele unterschiedliche Funktionen, von denen hier nur die Auswahlfunktion für GPIO-Interrupts interessiert. Insgesamt können aus allen vorhandenen GPIO acht ausgewählt werden, die dann jeweils eine eigene ISR aufrufen können. Die in Abbildung 2 eingezeichneten acht Eigenschaftsblöcke erlauben jeder für sich die Zuordnung eines GPIO (links) zu einem von 8 Kanälen zum nächsten Modul. In diesem Versuch werden nur zwei Kanäle benötigt. Welche Kanäle benutzt werden, ist nicht entscheidend. Der einzige Unterschied ist die festgelegte Priorität, die hier ohne jede Bedeutung ist.

Kapitel im UM	Chapter 3: LPC11U3x/2x/1x System control block
Dateien in der Bibliothek	Entfällt hier, benötigte Funktion in v4.h enthalten
Handle	Entfällt hier
Nummerierung der Pins	Port, pin
Nummerierung der Kanäle	Zahl 0-7
Initialisierung	Nicht erforderlich, Modul ist immer aktiv

Zuordnung eines Kanals (*intno*) zu einem GPIO (*port*, *pin*):
void Chip_SYSCTL_SetPinInterrupt(uint32_t intno, uint8_t port, uint8_t pin);

1.4.4 PININT (PIN INTerrupt)

Dieses Modul ist neu. Es hat drei Aufgaben:

1. Einstellung der Eigenschaften des Signals zur Erkennung eines Ereignisses. Für jeden der acht Kanäle kann die Eigenschaft unabhängig eingestellt werden.
2. Verwaltung des zugehörigen Flags (pro Kanal ein Flag)
3. Verwaltung der zugehörigen individuellen Masken (pro Kanal eine Maske)

Für die Programmierung wichtig:
Flags werden solange gesetzt, wie die eingestellte Eigenschaft anliegt. Flags müssen durch das Programm gelöscht werden. In der Regel wird man das in der zugehörigen ISR tun, denn dann hat man ja das Ereignis sicher zur Kenntnis genommen. Bei der erstmaligen Initialisierung ist es sinnvoll, die Flags auch zu löschen, denn sie könnten ja von lange zurückliegenden (zufällig eingetretenen) Ereignissen gesetzt worden sein. Diese Ereignisse interessieren bei einem Neustart natürlich nicht mehr.

Kapitel im UM	Chapter 9: LPC11U3x/2x/1x GPIO
Dateien in der Bibliothek	*pinint_11xx.h*
Handle	*LPC_PININT*
Nummerierung der Kanäle	Symbole *PININTCH0* bis *PININTCH7* (**nicht** Zahlen 0-7)
Initialisierung	Notwendig, Modul ist zunächst abgeschaltet

Initialisierung des Moduls;
*void Chip_PININT_Init(LPC_PIN_INT_T *pPININT);*

Löschen des Flags eines Kanals (ein Kanal wird hier *pins* genannt):
*void Chip_PININT_ClearIntStatus(LPC_PIN_INT_T *pPININT, uint32_t pins);*

Festlegen der Ereigniseigenschaft edge oder level pro Kanal:
*void Chip_PININT_SetPinModeEdge(LPC_PIN_INT_T *pPININT, uint32_t pins);*
*void Chip_PININT_SetPinModeLevel(LPC_PIN_INT_T *pPININT, uint32_t pins);*

Festlegen der Ereigniseigenschaft rising (falls edge) oder high (falls Level)
*void Chip_PININT_EnableIntHigh(LPC_PIN_INT_T *pPININT, uint32_t pins);*

Festlegen der Ereigniseigenschaft falling (falls edge) oder low (falls level)
*void Chip_PININT_EnableIntLow (LPC_PIN_INT_T *pPININT, uint32_t pins);*

Man kann bei dem Parameter *pins* mehrere Kanäle zusammenfassen (bitweises Oder), man kann die Funktionen aber auch nacheinander mit jeweils einem Kanal aufrufen.

1.4.5 NVIC (Nested Vectored Interrupt Controller)

Das Modul ist aus der Vorlesung bekannt. Es verwaltet die Zuordnung der Ereignisquellen zu ISRs mit Hilfe einer Tabelle. Die Tabelle selbst steht im Speicher und wird in der Regel mit vordefinierten Funktionsnamen gefüllt (*startup-code*). Der NVIC hat selber für jede der (hier) möglichen 32 externen Ereignisquellen je ein Flag und eine Maske.

Für die Programmierung wichtig:
Das Modul PININT ist fest mit den ersten acht Eingängen verbunden, so dass die ersten acht Tabelleneinträge für die Zuordnung verwendet werden.
Die Flags im NVIC werden beim Aufruf der zugehörigen ISR automatisch gelöscht. Hier muss der Programmierer also nichts weiter tun.

Kapitel im UM	Chapter 6: LPC11U3x/2x/1x NVIC
Dateien in der Bibliothek	*core_cm0.h*
Handle	Entfällt hier
Nummerierung der Kanäle	Zahl 0-31, besser Symbole aus *cmsis-11uxx.h*
Initialisierung	Nicht erforderlich, Modul ist immer aktiv

Die Symbole zu den einzelnen Tabelleneinträgen sind als Enumerationsdatentyp mit dem Namen *IRQn_Type* definiert. Für die ersten acht Tabelleneinträge lauten sie *PIN_INT0_IRQn* - *PIN_INT7_IRQn*. Diese symbolischen Nummern kann man dann in den Funktionen für den NVIC benutzen.

Löschen eines Flags im NVIC: void *NVIC_ClearPendingIRQ(IRQn_Type IRQn*);
Öffnen einer Maske im NVIC: *void NVIC_EnableIRQ(IRQn_Type IRQn);*

1.4.6 Globale Maske

Die globale Maske ist in Abbildung 2 ganz rechts am Ausgang des NVIC eingezeichnet. Sie ist tatsächlich dem Rechenkern (µP) zugeordnet und wird direkt über Prozessorbefehle gesteuert. Zur einfacheren Benutzung gibt es dafür in der Regel C-Funktionen.

Für die Programmierung wichtig:
Nach der Ausführung des *startup-code* ist die Maske geschlossen. Sie muss also nach der Initialisierung aller gewünschten Interrupts einmalig geöffnet werden. Wenn eine ISR aufgerufen wird, dann geschieht das mit einer im NVIC eingestellten Priorität. Diese Priorität kann der Programmierer festlegen. Im Projekt haben alle Ereignisquellen am NVIC dieselbe Priorität.
Wenn eine ISR aufgerufen wird, dann können nur noch Ereignisse mit einer höheren Priorität diese ISR unterbrechen. Nach dem Ende einer ISR wird automatisch wieder die Priorität des unterbrochenen Programms hergestellt.
Die globale Maske wird weder automatisch geschlossen noch geöffnet.

Öffnen der globalen Maske: *void __enable_irq(void);*
Schließen der globalen Maske: *void __disable_irq(void);*

Die beiden Unterstriche zu Beginn der Funktionen sind kein Fehler, sie sind so deklariert.

1.5 Programmierung der Anwendung

Jetzt können Sie das Skelett im Projekt (*v4_p1.c*) entsprechend ausfüllen und testen. Beim Test kann es sinnvoll sein, auf eine oder beide ISR einen Breakpoint zu setzen, um zu sehen, ob der jeweilige Tastendruck auch tatsächlich zum Aufruf der ISR führt.
Probieren Sie auch andere Einstellungen für die Eigenschaften der Pins (IOCON) und der Ereigniseigenschaften (PININT) aus und prüfen Sie, ob das Ergebnis jeweils Ihren Erwartungen entspricht.

Versuch 4 – Interrupts

2 Ansteuerung aller Stellen des Displays

Im zweiten, unabhängigen, Teil des Versuchs sollen alle vier Stellen des LED-Displays individuell angesteuert werden können. Das wäre einfach, wenn es zu jeder LED eine eigene Leitung geben würde. In dem Fall bräuchte man aber $7 * n$ (hier also 28) Leitungen am Display und entsprechend viele GPIOs am µC. Diese Art der Ansteuerung ist zu teuer, da Anschlüsse vergleichsweise viel Geld kosten (auch Platz auf der Platine kostet Geld) und mehr Anschlüsse auch mehr potentielle Fehler in der Produktion bedeuten.
Die übliche Lösung ist das Multiplex-Verfahren, bei dem die einzelnen Stellen zeitlich nacheinander so schnell eingeschaltet werden, dass für den Menschen kein Flimmern mehr erkennbar ist. Dann benötigt man nur $7 + n$ Anschlüsse (hier also 11). Nachteilig ist die verringerte Helligkeit jeder Stelle, da jede Stelle nur für $1/n$ der Zeit leuchtet.
Außerdem muss man nun natürlich die Segmente und Stellen zeitlich richtig ansteuern.
Dieser Versuchsteil wird in drei Schritten (S1, S2 und S3) durchgeführt.

2.1 Vorbereitung

Wechseln Sie in einen neuen, leeren Workspace. Importieren Sie dann aus dem Archiv *v4_vorbereitung.zip* die **drei** Projekte *lpc_chip_11uxx_lib*, *mch_aux_lib* und ***v4_p2***. Für diesen Versuch benötigen Sie wieder wie in V3 das externe Netzteil, damit das Display nicht über die GPIOs des µC mit Strom versorgt wird.

2.2 Bedingte Kompilierung

Sehen Sie sich zunächst den Header *v4.h* im Verzeichnis *inc* des Projektes *v4_p2* an. Dort sind die Symbole *V4_P2_S1* bis *V4_P2_S4* vorbereitet, aber noch auskommentiert. Direkt darunter finden Sie die Anweisung *#if ... #else ... #endif*. Der obere Teil ist im Editor ausgegraut. Das weist darauf hin, dass dieser Text nicht kompiliert wird. Damit wird dem Symbol *V4_FREQUENCY* der Wert 250 zugewiesen.
Entfernen Sie jetzt den Kommentar für das Symbol *V4_P2_S1*. Sie sollten nach einer sehr kurzen Verzögerung sehen können, dass jetzt das Symbol *V4_FREQUENCY* den Wert 4 erhält. Das ist ein typisches Beispiel für die bedingte Kompilierung, bei der manche Teile des Quelltextes inaktiv sind. Welche Teile aktiv sind, wird dabei durch Symboldefinitionen und passende Bedingungen gesteuert.
Sehen Sie sich dazu auch *v4_p2.c* an. Hier werden ebenfalls Teile des Quelltexts in Abhängigkeit der vier Symbole *V4_P2_S1* bis *V4_P2_S4* aktiviert bzw. deaktiviert. Optisch sichtbar wird die Änderung hier aber erst, wenn Sie *v4.h* auch gespeichert haben.

Tipp:
Man kann im Editor die nicht aktiven Teile des Quelltextes aus- und einblenden, wenn man ihn entsprechend einstellt. Dazu in einem Quelltext an beliebiger Stelle mit der rechten Maustaste das Kontextmenü einblenden und den letzten Eintrag *Preferences* wählen. Links unter *C/C++* den Eintrag *Editor* expandieren und dort *Folding* wählen. Nun kann man rechts *Enable folding of preprocessor branches* aktivieren. Möchte man haben, dass beim Öffnen einer Datei die inaktiven Teile gleich ausgeblendet werden, dann kann man auch den letzten Eintrag unter *Initially fold these region types* aktivieren (*Inactive Preprocessor Branches*).
In beiden Fällen kann man dann in einer Quelldatei (Änderung wird erst nach dem erneuten Öffnen der Datei angewandt!) links neben den Anweisungen zur bedingten Kompilierung Code-Teile ein- bzw. ausblenden (Symbole + und – in den kleinen Kreisen anklicken).

2.3 S1 - Timer-Interrupt

Definieren Sie nur das Symbol *V4_P2_S1* in *v4.h* und kompilieren Sie dann das Projekt. Das sollte fehlerfrei möglich sein. Testen Sie es dann, sofern vorhanden, auf dem Kit (nur Piggyback ohne Netzteil nötig). Die LED blinkt mit etwa 2 Hz.
Wenn Sie sich den Quelltext ansehen, dann sind vier Funktionen neu für Sie:

void SystemCoreClockUpdate(void);

Das ist eine herstellerübergreifende Standardfunktion für µC mit ARM-Rechenkern. Sie bestimmt die aktuelle Systemfrequenz und legt sie in der globalen Variable *SystemCoreClock* ab. Läuft der µC aktuell mit 48 MHz, dann steht nach Aufruf der Funktion 48000000 in der Variable.

uint32_t SysTick_Config(uint32_t ticks);

Mit dieser Funktion wird ein ebenfalls herstellerübergreifend vorhandener Timer (er heißt SYSTICK, daher der Funktionsname) auf eine gewünschte Frequenz eingestellt. Der Parameter *ticks* enthält den Endwert (siehe Vorlesung, MATCH-Register). Nach Aufruf der Funktion zählt der Zähler von 0 bis ticks-1 und beginnt dann automatisch von neuem. Bei jedem Überlauf löst er einen Interrupt aus.
Der Rückgabewert wird gibt an, ob die Funktion erfolgreich war (0) oder nicht (1). Die Funktion liefert „nicht erfolgreich“, wenn der Endwert zu groß ist.

void SysTick_Handler(void);

Das ist die ISR, die bei jedem Überlauf des SYSTICK-Timers aufgerufen wird. Man braucht hier kein Flag manuell zurücksetzen, das geschieht automatisch.

*void Chip_GPIO_SetPinToggle(LPC_GPIO_T *pGPIO, uint8_t port, uint8_t pin);*

Diese Funktion schaltet den Ausgabewert des GPIOs *port*, *pin* um (H->L, L->H).

Schreiben Sie jetzt die ISR so um, dass Sie die Umschaltfunktion *Chip_GPIO_SetPinToggle* nicht mehr benötigen. Sie müssen sich also den aktuellen Zustand der LED merken, damit Sie bei jedem Aufruf der ISR in den jeweils anderen Zustand wechseln können. Zugelassen sind nur die „Grundfunktionen“ *Chip_GPIO_SetPinOutLow*, *Chip_GPIO_SetPinOutHigh* und *Chip_GPIO_SetPinState*. Verwenden Sie keine globalen Variablen. Lokale Variable in der ISR sind zulässig.

```
#if defined(V4_P2_S1) || defined(V4_P2_S2)

 Ersatzcode hier

#endif
```

Tipp:
Schreiben Sie den Ersatzcode im Quelltext wie oben angedeutet in einen bedingten Teil, damit der gesamte Ersatzcode später mittels bedingter Kompilierung leicht aktiviert/deaktiviert werden kann.

2.4 S2 – Displaystellen einzeln ansteuern

Definieren Sie nur das Symbol *V4_P2_S2* in *v4.h* und sehen Sie sich dann *v4_p2.c* wieder an (*v4.h* speichern, *v4_p2.c* auch speichern und dann erneut öffnen. Wenn Sie sich den Quelltext ansehen, dann sind folgende Code-Teile neu aktiviert worden:

1. Einstelldaten der benötigten Pins jetzt wie in V3 (Display-Anschlüsse) vorhanden
2. Funktionen zur Umsetzung einer Zahl 0-15 in ein Muster für eine Siebensegment-anzeige sind vorhanden: *v3_get_pattern*, *v3_set_pattern*
3. Eine Struktur *display* ist definiert
4. Eine Funktion *v4_print_int* zum „Drucken“ einer Zahl ähnlich *printf(“%d“,x)* ist vorhanden
5. Ein Feld *digits* mit 4 Elementen ist definiert

Die Punkte 1 und 2 sind aus V3 bekannt bzw. von Ihnen entsprechend gelöst. Sie können selbstverständlich anstelle der vorgeschlagenen Lösung hier Ihre eigene Lösung übernehmen.

Die Struktur *display* enthält ein Element (*control*), mit dem später entschieden werden soll, ob die Anzeige ein- oder ausgeschaltet ist. Das zweite Element ist ein Feld (*digits*), in dem die Muster stehen, die an den vier Stellen des Displays angezeigt werden sollen. Diese Struktur ist global sichtbar.

Die Funktion *v4_print_int* „druckt“ einen Zahlenwert (Integer) *x* in ein Feld in Form von Mustern für eine Siebensegmentanzeige. Man muss der Funktion den Beginn des Feldes, in das gedruckt werden soll, übergeben. Das ist der Parameter *buf*. Da die Funktion nicht wissen kann, wie viele Stellen man sehen möchte, muss man die Anzahl der Stellen als Parameter *n* mit übergeben. Das Feld muss dann natürlich mindestens *n* Elemente haben, sonst kommt es zu einem Speicherüberlauf. Man kann mit dem Parameter *base* sogar noch die Zahlenbasis für den Druck angeben, z.B. 10 für dezimal.
In *main()* sehen Sie eine Anwendung zu Testzwecken.

Das Feld *digits* könnte eine Arbeitserleichterung darstellen – Nutzung optional.

Ihre Aufgabe ist es, die ISR wie folgt zu erweitern:
Wenn in der Struktur *display* das Element *control* auf falsch steht, dann werden alle 4 Stellen einfach ausgeschaltet. Andernfalls wird die **aktuell** eingeschaltete Stelle abgeschaltet. Danach wird zur **nächsten** Stelle gewechselt (1->2, 2->3, 3->4, 4->1) und für diese Stelle aus dem Musterspeicher (siehe Struktur *display*) das Muster ausgegeben (Segmente a-g). Danach wird die neue Stelle eingeschaltet. Sie müssen sich also merken, welche Stelle gerade aktuell ist.
Bei **jedem** Aufruf der ISR ändert sich also die gerade aktuell eingeschaltete Stelle.
Verwenden Sie zur Lösung nur lokale Variable. Der entsprechende Teil ist in der ISR gekennzeichnet:

```
#if defined(V4_P2_S2) || defined(V4_P2_S3) || defined(V4_P2_S4)

Code hier

#endif
```

Wenn der Test zufriedenstellen verläuft, dann können sie ja testweise die Frequenz des SYSTICK-Timers auf 250 Hz erhöhen, um ein „stehendes Bild” auf dem Display zu bekommen.

2.5 S3 – Sekundenzähler

Für eine Eieruhr oder Ähnliches soll eine Zeitmessung in Sekunden auf der Anzeige erscheinen. Dazu muss eine Zahl im Sekundentakt erhöht werden.

Definieren Sie nur das Symbol *V4_P2_S3* in *v4.h* und sehen Sie sich dann *v4_p2.c* wieder an (*v4.h* speichern, *v4_p2.c* auch speichern und dann erneut öffnen. Wenn Sie sich den Quelltext ansehen, dann sind folgende Code-Teile neu aktiviert worden:

1. Eine globale Variable *count* ist definiert
2. Das Anzeigemuster wird in *main()* einmalig mit dem Wert von *count* vorbelegt

Außerdem wird die Multiplex-Frequenz auf 269 Hz angehoben.

Diese Frequenz erscheint reichlich seltsam, ist aber bewusst so gewählt. Warum könnte eine solche Frequenz günstiger als Werte wie 250 Hz oder 300 Hz sein?

Man könnte jetzt einen zweiten Timer so einstellen, dass er jeder Sekunde einen Interrupt auslöst und in dessen ISR die Variable *count* immer um 1 erhöhen. Das wäre eine durchaus praktikable Lösung.
Hier soll aber das Problem gleich mit der schon vorhandenen ISR des SYSTICK-Timers gelöst werden. Sie brauchen ja nur bei jedem 269-ten Mal die Variable *count* zu erhöhen. Schreiben Sie dazu passenden Code an die angegebene Stelle in der ISR:

```
#if defined(V4_P2_S3) || defined(V4_P2_S4)

Code hier

#endif
```

Benutzen Sie zur Lösung außer *count* nur lokale Variable. Wenn Sie den Wert 269 benötigen, dann verwenden Sie im Code nicht die Zahl, sondern das passende Symbol. Vergessen Sie nicht, nach der Änderung des Zählers auch das Muster für die Anzeige neu zu bestimmen (mittels *v4_print_int()*), sonst wird ja der neue Zählerstand nicht angezeigt.

3 Zusammensetzen beider Teile

Wenn Sie beide Versuchsteile gelöst haben, dann können Sie mit den beiden Tasten leicht das Display oder der Zähler steuern (z.B. an/aus, start/stopp, …).

Definieren Sie nur das Symbol *V4_P2_S4* in *v4.h* und sehen Sie sich dann *v4_p2.c* wieder an (*v4.h* speichern, *v4_p2.c* auch speichern und dann erneut öffnen. Wenn Sie sich den Quelltext ansehen, dann sind folgende Code-Teile neu aktiviert worden:

1. Einstelldaten der Tasten vorhanden
2. Funktion *v4_init* benötigt neuen Code
3. Die beiden ISR aus *v4_p1.c* werden benötigt

Den nötigen Code für die zusätzliche Initialisierung können Sie 1:1 aus *v4_p1.c* übernehmen.

Bei den beiden ISR für die beiden Tasten können Sie für einen ersten Test anstelle der LED einfach das Kontrollelement *control* in der Struktur *display* entsprechend setzen. Das tatsächliche Ein-/Ausschalten des Displays übernimmt ja dann die ISR des SYSTICK-Timers.

Für andere Anwendungen (z.B. Zählrichtung ändern) könnten Sie eine zusätzliche globale Variable einführen, in der dann die Richtung und/oder die Schrittweite steht. Diese Variable wird dann in den ISR für die Tasten verändert und in der ISR beim Zählen benutzt.

Lernziele

- Nutzung der bedingten Kompilierung
- Variableneigenschaften static und volatile
- Umgang mit Strukturen und Feldern aus Strukturen
- Einstellung und Nutzung von Interrupts (ISR, Flag, Maske)

Material zur Vorbereitung

- User Manual UM10462, Revision 5.3 für den µC
- Pinliste lpc11u_pins.pdf
- Projektarchiv v4_vorbereitung.zip für MCUXpresso

Material zur Durchführung

- Basisplatine, Piggyback weiß, Flachbandkabel, LED-Display, Netzteil
- Entwicklungswerkzeug (hier MCUXpresso Version 10.0.0)