1 Interrupts (Unterbrechungen)

Ein wesentlicher Unterschied zwischen Anwendungen auf einem PC und Anwendungen auf einem µC ist, dass ein µC meist innerhalb einer fest vorgegebenen Zeit auf Ereignisse reagieren muss. Ereignisse können zu beliebigen Zeitpunkten eintreten und sie können auch von nur kurzer Dauer sein. Trotzdem muss der µC das Ereignis erkennen können. Als Beispiel soll die Erkennung eines Werkstücks mittels einer Lichtschranke dienen (Abbildung 1) dienen.

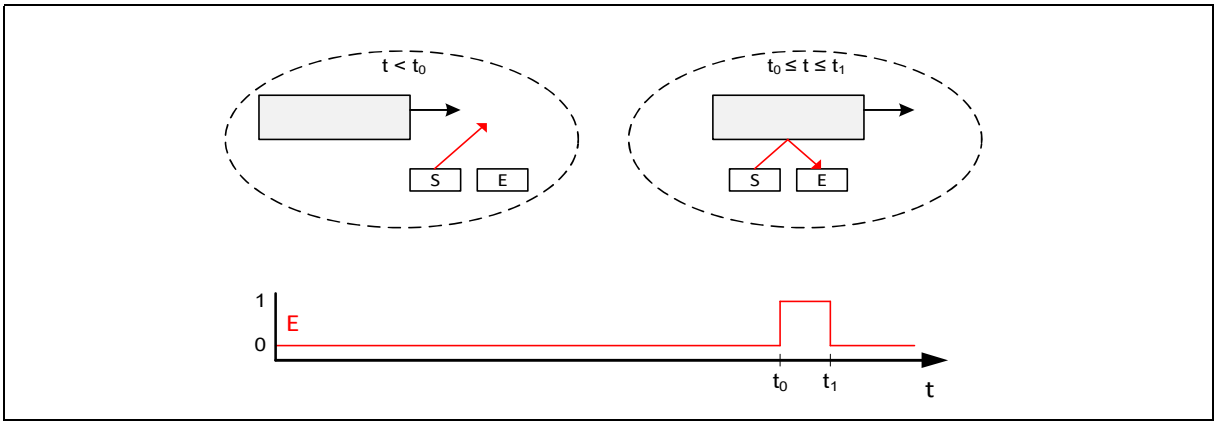

Abbildung 1: Beispiel Lichtschranke

Die Lichtschranke besteht aus dem Sender S und dem Empfänger E. Solange sich das Werkstück nicht vor der Lichtschranke befinden (hier $t < t_0$), wird das vom Sender abgestrahlte Licht nicht am Empfänger empfangen. Er gibt dann den Wert 0 aus. Befindet sich das Werkstück vor der Lichtschranke, dann wird Licht reflektiert und der Empfänger gibt eine 1 aus (hier im Zeitraum $t_0 \leq t \leq t_1$).
Man kann nicht vorhersehen, wann sich Werkstücke vor der Lichtschranke befinden werden und man kann, falls Größe und Geschwindigkeit der Werkstücke variabel sein können, auch die Dauer t_1-t_0 nicht vorhersehen. Trotzdem muss man das Ereignis „Werkstück hat Lichtschranke passiert“ zuverlässig erkennen können.

1.1 Polling (aktives Warten)

Die einfachste Lösung wäre, in einer Schleife auf das Ereignis zu warten (Listing 1).

```
while (1)
{
  // warte bis Werkstück erkannt
  while (E==0);

  // Werkstück ist angekommen (t0), erfassen
  anzahl++;

  // warte, bis Werkstück wieder weg ist
  while (E==1);
}
```

Listing 1: Polling

Diese Methode hat einen großen Nachteil: Der µC muss diese Schleife ständig ausführen, um auf keinen Fall den Zeitraum $t_0 \leq t \leq t_1$ zu verpassen. Er kann also außer der Beobachtung des Signals E nichts sinnvolles mehr tun.
In der Praxis ist diese Methode also nur dann anwendbar, wenn man sicher sein kann, dass der Zeitraum $t_0 \leq t \leq t_1$ so lang ist, dass der µC an der Abfrage des Signals E auch dann noch mindestens einmal innerhalb der Schleifen vorbeikommt, wenn man ein paar weitere Anweisungen innerhalb der Schleifen unterbringt. Andernfalls besteht die Gefahr, ab und zu ein Ereignis zu verpassen. Mit wenigen Ausnahmen ist diese Methode daher in der Praxis unbrauchbar.

1.2 Interrupts (Unterbrechungen)

Um dieses Problem zu lösen, können alle heutigen µC ohne aktive Abfrage durch ein Programm auf bestimmte Ereignisse reagieren. Falls ein solches Ereignis eintritt, dann wird (sofern der Anwender das wünscht) das gerade laufende Programm A abgebrochen, wobei sich der µC aber merkt, an welcher Stelle der Abbruch erfolgte. Dann wird automatisch ein ebenfalls vom Anwender vorgegebenes Programm gestartet, die sog. *Interrupt Service Routine* (ISR).
Nun kann innerhalb der ISR auf das Ereignis reagiert werden. Ist diese Bearbeitung abgeschlossen, dann kehrt der µC selbständig zu dem Punkt im Programm A zurück, an dem er unterbrochen wurde und das Programm A wird fortgesetzt (Abbildung 2).

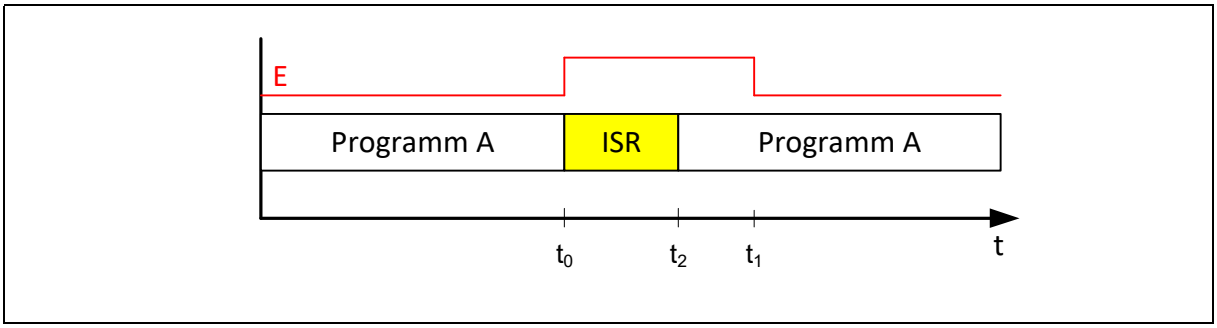

Abbildung 2: Prinzipieller Ablauf mit ISR

Das Ereignis, das zum Aufruf des ISR führt, ist hier der Übergang 0->1 des Signals E. Die Bearbeitung des Ereignisses ist zum Zeitpunkt t_2 abgeschlossen. Das Werkstück befindet sich zwar immer noch vor der Lichtschranke, aber das Programm A kann trotzdem schon fortgesetzt werden. Der Übergang 1->0 des Signals E führt ja nicht zum Aufruf der ISR.

Im Anwenderprogramm (Listing 2) befinden sich jetzt zwei völlig voneinander unabhängige Funktionen: Das Programm A (hier als Funktion *Programm_A()* dargestellt) sowie die dem Ereignis E 0->1 zugeordnete ISR (hier *ISR_E0_1()* genannt).

```
Programm_A()
{
  while (1)
  {
    // führe beliebige Berechnungen aus
    …
  }
}

void ISR_E0_1(void)
{
  // Werkstück ist angekommen, erfassen
  anzahl++;
}
```

Listing 2: Prinzipieller Programmaufbau mit ISR

Die Funktion *ISR_E0_1()* wird **nicht** aus anderen Funktionen, z.B. der Funktion *Programm_A*, aufgerufen. Sie kann daher weder Parameter empfangen noch Parameter zurückgeben.

2 Funktionsweise

Der Teil innerhalb eines µC, der für die Verwaltung der Interrupts zuständig ist, wird als Interruptsystem bezeichnet. Einige Teile davon sind global für alle Interrupts zuständig, andere Teile beziehen sich nur auf jeweils einen bestimmten Interrupt aus den verschiedenen Modulen. Entsprechend finden sich die nötigen Bits für die Einstellung meist verstreut in den Registern der verschiedenen Module.

2.1 Ereignisquellen, Interruptvektoren

Die möglichen Ereignisquellen sind für jeden µC fest vordefiniert und könne nicht verändert werden. Für jede Quelle ist eine Unterbrechungsroutine (*ISR, Interrupt Service Routine*) zuständig, die aufgerufen wird, wenn das Ereignis eintritt. Allerdings ist es durchaus üblich, dass sich mehr als ein Ereignis dieselbe ISR teilt. In diesem Fall muss innerhalb der ISR erst noch durch eine Abfrage entsprechender Bits in Registern herausgefunden werden, welches Ereignis tatsächlich den Interrupt ausgelöst hat. Wenn einer ISR nur genau ein Ereignis zugeordnet ist, dann ist eine solche Abfrage nicht mehr erforderlich.
Die ISR beginnen entweder an fest vordefinierten Adressen im Programmspeicher oder aber es gibt eine Tabelle, in der die jeweilige Startadresse der ISR gespeichert ist.
Bei dem Praktikums-µC ist eine solche Tabelle (UM, Table 59) definiert.
Diese Tabelle heißt üblicherweise *Interrupt Vector Table*. Der Linker[1] sorgt dafür, dass die Einträge richtig belegt werden. Jedem Eintrag entspricht dann eine C-Funktion – das ist dann die ISR zu diesem Ereignis.

[1] Programm, das nach dem Compiler und Assembler läuft und die einzelnen Teile (Code, Daten) bestimmten Speicherbereichen im µC zuordnet.

2.2 Masken

Masken sind programmierbare Sperren, mit denen verhindert werden kann, dass der aktuelle Programmfluss durch ein Ereignis unterbrochen werden kann. Sie haben die Funktion eines „Bitte nicht stören“-Schildes. In der Regel gibt es zwei Arten von Masken: solche, die einem bestimmten Ereignis zugeordnet sind (individuelle Masken) und eine, die alle Ereignisse betrifft (globale Maske).
Damit bei einem entsprechenden Ereignis die zugehörige ISR auch wirklich aufgerufen wird, müssen sowohl die globale Maske als auch die individuelle Maske für das Ereignis geöffnet sein. Die globale Maske hat den Zweck, schnell sicherstellen zu können, dass ein Programmteil von keinem Ereignis unterbrochen werden kann, ohne dass man dazu alle individuellen Masken einzeln schließen müsste. Nach der Bearbeitung dieses kritischen Programmteils kann man dann die globale Maske wieder öffnen und hat vorherigen Zustand (welche Ereignisse dürfen den Programmfluss unterbrechen, welche nicht) wiederhergestellt.
Masken werden allgemein durch Bits in Registern dargestellt.
Obwohl nicht standardisiert, ist die Funktion *sei()* sehr häufig definiert und öffnet die globale Maske. Die Funktion *cli()* schließt dann die globale Maske. Bei den weit verbreiteten µC mit ARM-Rechenkern, zu denen auch der Praktikums-µC gehört, heißen die entsprechenden Funktionen *__enable_irq()*[2] (Öffnen) und *__disable_irq()* (Schließen).

2.3 Flags

Die meisten Ereignisse sind zu kurz, um zufällig gerade zu dem Zeitpunkt vom µC abgefragt zu werden, in dem sie eintreten. Deswegen wird der Eintritt des Ereignisses in einem *Flag* gespeichert. Dieses Flag dient dann für den µC als eigentliche Ereignisquelle, d.h. ein gesetztes Flag zeigt an, dass (vor einer unbestimmten Zeit!) das auslösende Ereignis eingetreten ist. Tritt ein Ereignis noch einmal ein, während das zugehörige Flag noch gesetzt ist, dann geschieht nichts weiter – die Anzahl der Ereignisse wird also nicht gezählt.
Ein Flag wird entweder automatisch mit dem Eintritt in die zugehörige ISR gelöscht oder es muss im Programm (der ISR) selbst erledigt werden. Ob es einen Automatismus gibt hängt vom µC ab – man muss das für jedes Flag nachlesen. Der Praktikums-µC verwendet keinen Automatismus. Flags werden genau wie Masken durch Bits in Registern dargestellt,

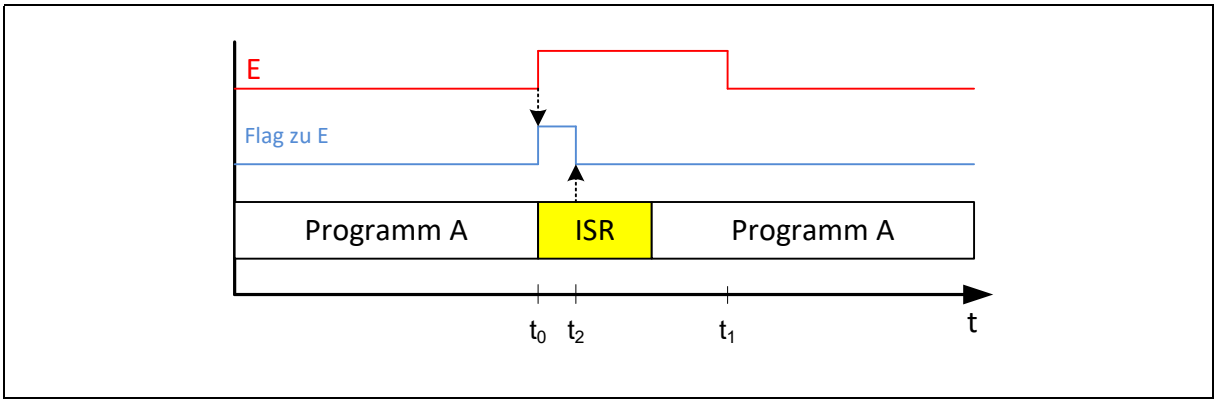

Abbildung 3: Zeitlicher Verlauf Ereignis – Flag (sofortiger Aufruf der ISR)

Für das Beispiel aus Abbildung 1 zeigt Abbildung 3 einen Ablauf, in dem die ISR sofort aufgerufen wurde, weil alle beteiligten Masken geöffnet waren. Das Signal E löst bei der steigenden Flanke (t_0) das Setzen des zugehörigen Flags aus. Dies wiederum ruft die ISR auf. Innerhalb der ISR wird dann per Befehl das Flag manuell gelöscht (t_2).

[2] Beachten Sie beim Einsatz, dass der Funktionsname mit zwei Unterstreichungszeichen beginnt.

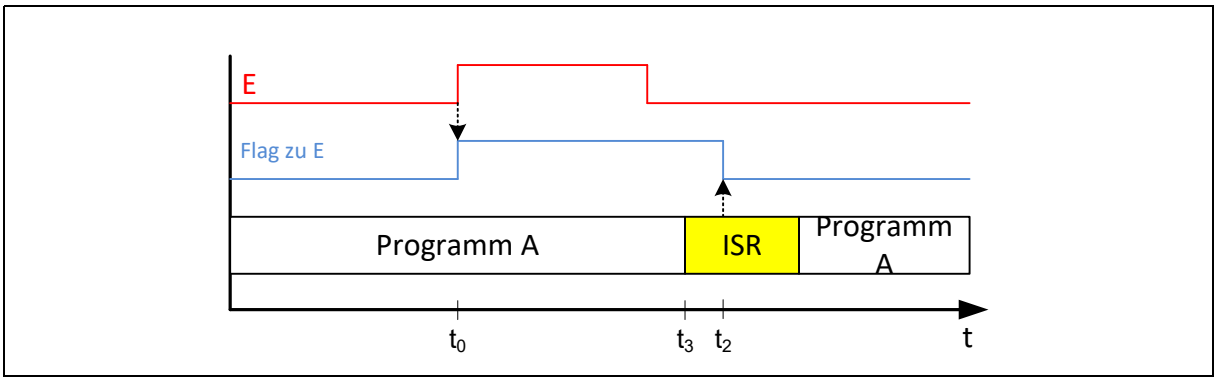

Abbildung 4: Zeitlicher Verlauf Ereignis - Flag (verzögerter Aufruf der ISR)

Falls aber zum Zeitpunkt t_0 irgendeine Maske den Aufruf der ISR verhindert, dann kann es zum Ablauf nach Abbildung 4 kommen. Hier wird erst zum Zeitpunkt t_3 über die Masken der Aufruf der ISR erlaubt. Das wird hier vom Programm A erledigt. Sobald die Maske offen ist, sieht das Interruptsystem, dass das Flag gesetzt ist und die ISR wird aufgerufen. Das auslösende Ereignis an E kann zu diesem Zeitpunkt prinzipiell beliebig lange zurückliegen.

2.4 Prioritäten

Unterbrechungen sind priorisiert, d.h. falls zwei Ereignisse gleichzeitig eintreten wird immer die ISR mit der höheren Priorität aufgerufen. Bei µCs ist diese Priorität entweder unveränderlich fest vorgegeben oder man kann sie in einem meist engen Rahmen ändern. Bei dem Praktikums-µC ist eine solche beschränkte Priorisierungseinstellung möglich.
Eine völlig freie Wahl der Prioritäten ist aus Aufwandsgründen in der Regel nicht möglich und sie wird in der Praxis auch nicht gebraucht (siehe Kapitel 3.2).
Gleichzeitigkeit bezieht sich hier **nicht** auf den Zeitpunkt, an dem ein Ereignis **tatsächlich** eingetreten ist. Ereignisse setzen ja zunächst einmal Flags, die solange gesetzt bleiben, bis sie entweder manuell oder automatisch gelöscht werden. Falls nun über einen längeren Zeitraum Ereignisse in Flags gesammelt werden, weil über die Masken die Bearbeitung gesperrt ist, dann **scheinen** beim Öffnen der Masken plötzlich alle unbearbeiteten Ereignisse gleichzeitig eingetreten zu sein.

3 Interruptbearbeitung

3.1 Zeitlicher Ablauf

In einem typischen µC wird jeweils vor der Abarbeitung eines neuen Maschinenbefehls das Interruptsystem abgefragt. Liegen ein oder mehrere freigegebene Interrupts an, so wird zunächst die Adresse des Programmzählers auf dem Stack gespeichert. Der Programmzähler enthält die Adresse des auszuführenden Maschinenbefehls. Damit kann das unterbrochene Programm später an der richtigen Stelle fortgeführt werden. Zudem wird ein Minimalsatz der Arbeitsregister des µC auf dem Stack gespeichert. Auch dies dient der späteren Wiederherstellung des Maschinenzustands, wenn die ISR beendet ist.
Gleichzeitig wird, sofern so vorgesehen, das zugehörige Flag gelöscht. Ab jetzt können neu eintretende Ereignisse das jeweilige Flag wieder setzen. Zuletzt wird das Programm mit dem ersten Befehl der zugehörigen ISR fortgesetzt.
Wenn die ISR beendet werden soll, dann werden mit einem speziellen Maschinenbefehl die zuvor automatisch gespeicherten Registerinhalte wieder vom Stack gelesen bzw. auf den zuvor verwendeten Registersatz zurückgeschaltet. Mit dem Zurücklesen des Programmzählers wird zuletzt an die Stelle zurückgekehrt, an der das Programm unterbrochen wurde. Das alles kostet Zeit, die von der eigentlichen Programmbearbeitung abgeht.

3.2 Schachtelung

Da die ISR ebenfalls nur ein Programm ist, kann sie natürlich bei Bedarf ebenfalls von anderen ISRs unterbrochen werden. In diesem Fall läuft die Sequenz aus Kapitel 3.1 erneut ab. Die einzelnen ISR werden also in diesem Fall geschachtelt (*nested*) bearbeitet. Nach dem Eintritt in eine ISR sind Interrupts jedoch je nach µC entweder global gesperrt, auf derselben Prioritätsebene gesperrt oder zumindest das auslösende Flag wird automatisch gelöscht. Andernfalls käme es zu einer unendlichen Rekursion. Bei Bedarf müssen innerhalb einer ISR also erst einmal Interrupts wieder zugelassen werden. Da man ja Interrupts auch individuell sperren kann, ist es so möglich, genau zu bestimmen, welche Interrupts innerhalb einer bestimmten ISR noch zulässig sind und welche bis zum Ende dieser ISR ignoriert werden. Auf diese Weise kann man per Software eine beliebig feine Priorisierung nachbilden.

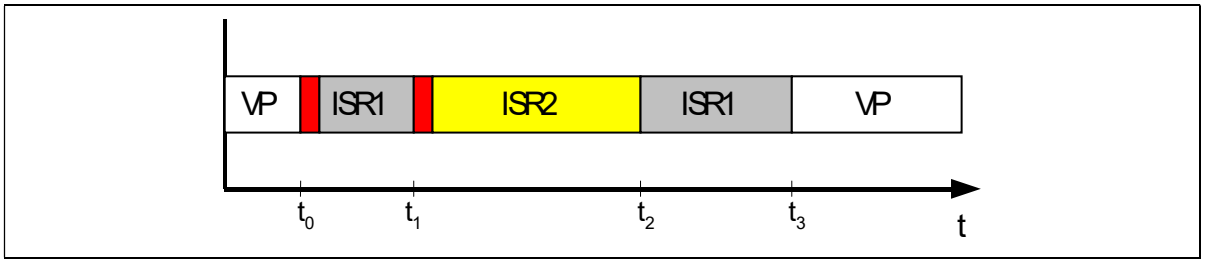

Abbildung 5: Geschachtelte ISR

In Abbildung 5 arbeitet der µC zunächst das normale Programm (VP) ab. Zum Zeitpunkt t_0 tritt ein Ereignis auf, das zur ISR1 führt. Für eine kurze Zeit (rot dargestellt) ist hier die globale Maske geschlossen. Mit dem grau dargestellten Teil beginnt die Abarbeitung der ISR1. Es sei nun angenommen, dass das Programm mittels *sei()* oder einer analogen Methode die globale Maske wieder geöffnet hat. Wenn nun zum Zeitpunkt t_1 ein zweites Ereignis auftritt, das zur ISR2 führt, dann wird jetzt eben die ISR1 unterbrochen. Zum Zeitpunkt t_2 sei ISR2 beendet. Dann wird ISR1 zu Ende geführt und nach deren Ende (t_3) wird das Programm VP weitergeführt.

3.3 Latenz

Unter der Latenz versteht man die Zeit, die vom Eintritt des Ereignisses bis zur Reaktion innerhalb der ISR im ungünstigsten Fall vergeht.
Die maximal zulässige Latenz ist durch die Anwendung bestimmt. Wenn beispielsweise Zeichen über eine Schnittstelle empfangen werden sollen, dann muss ein empfangenes Zeichen (Ereigniseintritt) spätestens bis zum Empfang des folgenden Zeichens abgeholt werden – sonst geht es verloren.

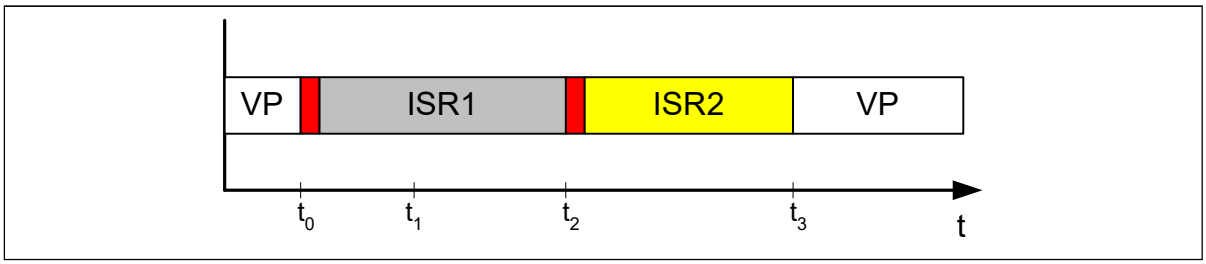

Abbildung 6: ISR ohne Schachtelung

In Abbildung 6 treten dieselben Ereignisse zum selben Zeitpunkt wie in Kapitel 3.2 beschrieben auf. Allerdings gibt die ISR1 die Interrupts hier nicht frei. Sie wird also am Stück bis t_2 abgearbeitet. Erst wenn sie beendet ist, kann ISR2 beginnen. Die Latenz für die ISR2 ist hier also die Zeitspanne $t_2–t_1$.

4 Interruptsystem

4.1 Allgemeine Struktur

Die ersten Mikroprozessoren (bis ca. dem Typ 8080, 1974) verfügten zwar bereits über eine Unterbrechungsmöglichkeit, aber es gab nur eine einzige Ereignisquelle und damit auch nur ein Flag und eine Maske (die man nach Belieben global oder individuell nennen kann – es gibt ja nur eine). Man musste sich also über Prioritäten und dergleichen auch keine Gedanken machen, denn es gab nur „alles oder nichts".
Heutige µC verfügen über Hunderte von Ereignisquellen, entsprechend gibt es Hunderte Masken und Flags und die Priorisierung ist zu einem wesentlichen Bestandteil eines µC-Interruptsystems geworden.
Im Lauf der Zeit hat sich dabei eine typische Struktur herausgebildet (Abbildung 7).

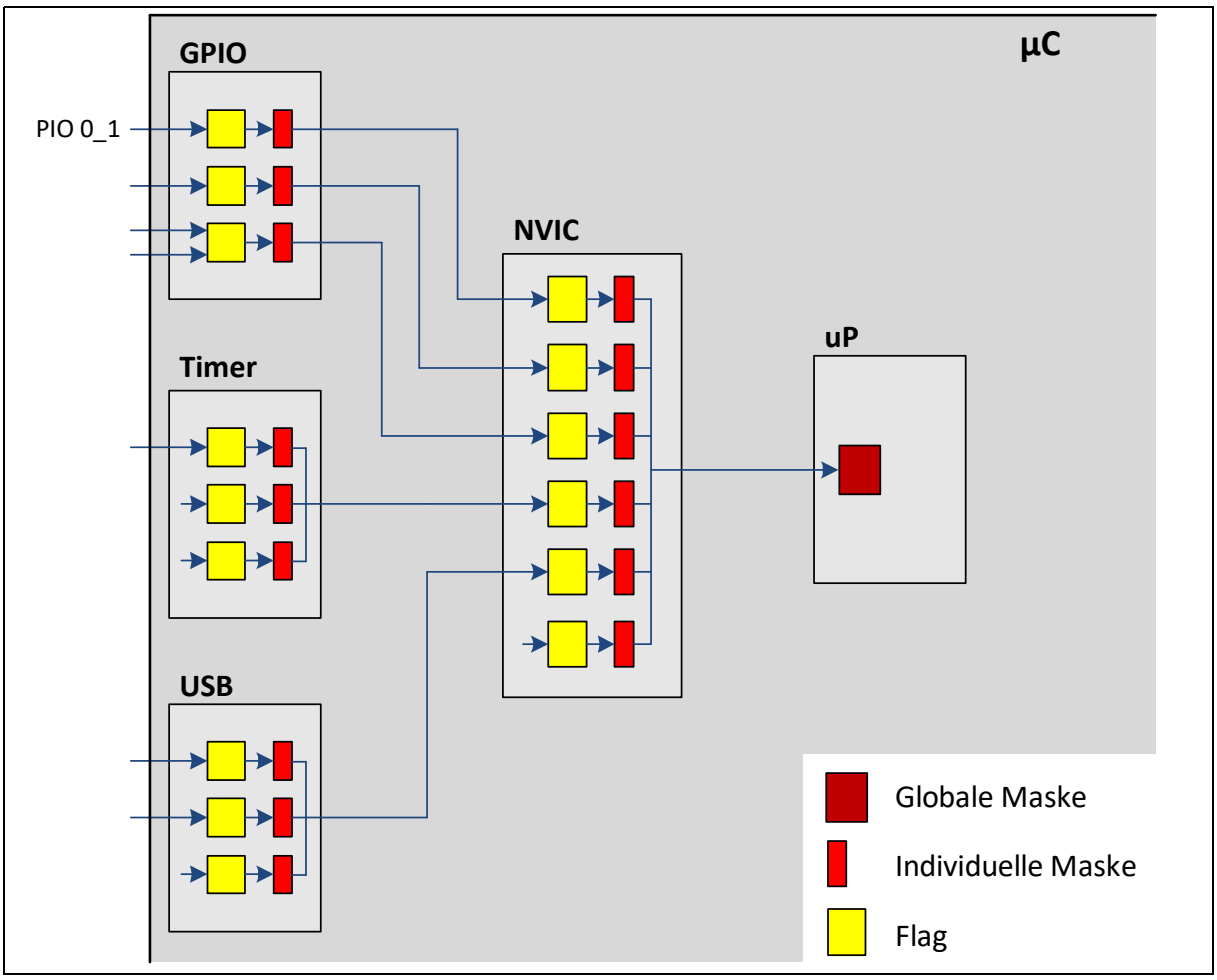

Abbildung 7: Typisches Interruptsystem eines µC mit ARM-Rechenkern

Der Weg von einer Ereignisquelle zum Ziel (Unterbrechung des Programflusses im µP) geht dabei hierarchisch durch zwei Ebenen. Die erste Ebene ist das Peripheriemodul, das dem Ereignis zugeordnet ist bzw. in dem es auftritt. Im Bild sind drei Peripheriemodule dargestellt (GPIO, Timer, USB), von denen das GPIO-Modul bereits bekannt ist.
Alle Peripheriemodule, die einen Interrupt auslösen können sind mit einem internen Modul (hier: NVIC[3]) verbunden, das die zweite Ebene darstellt. Dieses Modul hat zwei Aufgaben: Es priorisiert die Anforderungen der einzelnen Peripheriemodule und es löst zu jedem Ereignis an

[3] Nested Vectored Interrupt Controller: Nested -> Verschachtelung möglich, Vectored -> Tabelle mit ISRs

einem seiner Eingänge die passende ISR aus. Dieses Modul verwaltet also die in Kapitel 2.1 angesprochene Tabelle der den Ereignissen zugeordneten ISR.
An sich wäre es wäre es wünschenswert, wenn jeder Ereignisquelle ihre eigene ISR zugeordnet bekommen könnte und wenn zusätzlich auch noch die Priorität jedes einzelnen Ereignisses einstellbar wäre. Das ist aber ökonomisch nur für eine geringe Anzahl von Ereignisquellen umsetzbar. Heute (2017) haben die Tabellen mit einzeln zugeordneten ISR eine Größe im Bereich 32-256 Einträge und die Anzahl der einstellbaren Prioritäten liegt bei 4-8 oder ist ohnehin für jede ISR vordefiniert (z.B. indem die Priorität durch die Reihenfolge der Tabelleneinträge festgelegt ist) .

4.2 Vorverarbeitung in Peripheriemodulen

In nahezu allen Anwendungsfällen stellt die Beschränkung auf wenige Prioritätsebenen und eine geringere Anzahl an individuell zuordenbaren ISR als Ereignisquellen vorhanden sind, kein Problem dar. Als Beispiel soll ein Gerät gewählt werden, das 13 allgemeine Eingabetasten (z.B. Ziffern 0-9, +, -, OK) besitzt sowie einen Notaus-Taster. Über die allgemeinen Tasten kann die gewünschte Funktion eingegeben werden, mit dem Notaus-Taster kann im Gefahrfall eine laufende Aktion sofort abgebrochen werden.
Alle Tasten sind an GPIO angeschlossen und sollen per Interrupt abgefragt werden.
Es ist offensichtlich, dass der GPIO, an dem der Notaus-Taster angeschlossen ist, eine höhere Priorität haben muss als die anderen Tasten. Bei den anderen Tasten braucht man aber keine weitere Priorisierung, sie sind alle gleich wichtig. Außerdem ist es bei den anderen Tasten auch nicht nötig, dass jeder Taste eine eigene ISR zugeordnet wird. Man kann alle Tasten auf eine einzige ISR legen, die aufgerufen wird, wenn mindestens eine Taste gedrückt wird. In der ISR kann man dann die Tasten einzeln nacheinander abfragen, um herauszufinden, welche Taste gedrückt worden ist. Das kostet etwas Zeit, aber das fällt bei der Bedienung nicht auf (typische Latenz < 100µs). Bei der Notaus-Taste ist eine eigene ISR dagegen günstig, denn dann muss man in der ISR nicht mehr herausfinden, wodurch sie aufgerufen wurde – es kann nur die Betätigung des Notaus-Tasters gewesen sein (typische Latenz < 10 µs).
Daher konzentrieren die meisten Peripheriemodule alle Ereignisse, die von ihnen erkannt werden, auf eine einzige Interruptanforderung, die dann dem NVIC zugeleitet wird. Nur selten tritt der Fall ein, dass ein Peripheriemodul mehr als eine Interruptanforderung erzeugen kann – dies ist beim GPIO-Modul der Fall. In Abbildung 7 ist das links oben dargestellt: Für zwei GPIO ist im Modul jeweils ein eigenes Flag, eine eigene Maske und eine eigene Interruptanforderung zum NVIC eingebaut. Alle anderen GPIO teilen sich gemeinsam ein Flag und eine Maske.
Den Notaus-Taster würde man hier an PIO0_1 anschließen und im NVIC dieser Interrupt-anforderung eine eigene ISR zuweisen. Zudem würde diese ISR eine hohe Priorität erhalten.
Alle anderen Tasten würden an die GPIO angeschlossen, die sich eine Interruptanforderung teilen und die zugehörige ISR bekäme eine niedrige Priorität.
Die Aufgabe kann leicht mit zwei Prioritäten (statt 12+1=13) und zwei individuellen ISR (ebenfalls statt 13) gelöst werden.
Bei dem Praktikums-µC sind im GPIO-Modul 8 GPIO auswählbar, die eine eigene Interruptanforderung[4] zum NVIC leiten können und für alle anderen stehen zwei weitere Interruptanforderungen[5] zur Verfügung, auf die man die GPIO-Ereignisse verteilen kann (Sammelereignisse).
Damit belegt allein das GPIO-Modul schon 10 von den 32 möglichen Tabelleneinträgen. Nur das USB-Modul kann überhaupt noch mehr als eine Interruptanforderung stellen (2 Stück), alle anderen Modul nur je eine. Das ist typisch für heutige µC.

[4] Tabellenplätze 0-7, UM Table 59
[5] Tabellenplätze 8 und 9, UM Table 59

4.3 Prioritätsverwaltung im NVIC

Der NVIC hat zwei Aufgaben: Er enthält die Tabelle, in dem jeder der möglichen Interruptanforderungen eine ISR zugeordnet wird und er sorgt für die Priorisierung der Anforderungen. In der Praxis genügen wenige Prioritätsstufen, denn man kann die Ereignisse recht gut in „extrem wichtig“ (Not Aus), „sollte schnell behandelt werden“ (Audioausgabe ohne Aussetzer), „mittelwichtig“ (Eingabetasten aus dem Beispiel) und „wenn sonst nichts los ist“ (USB Stick angesteckt?) einteilen. Das sind nur 4 anstelle von (hier) 32 vorstellbaren Stufen.
In Abbildung 8 ist eine erweiterte Darstellung des NVIC eingezeichnet.

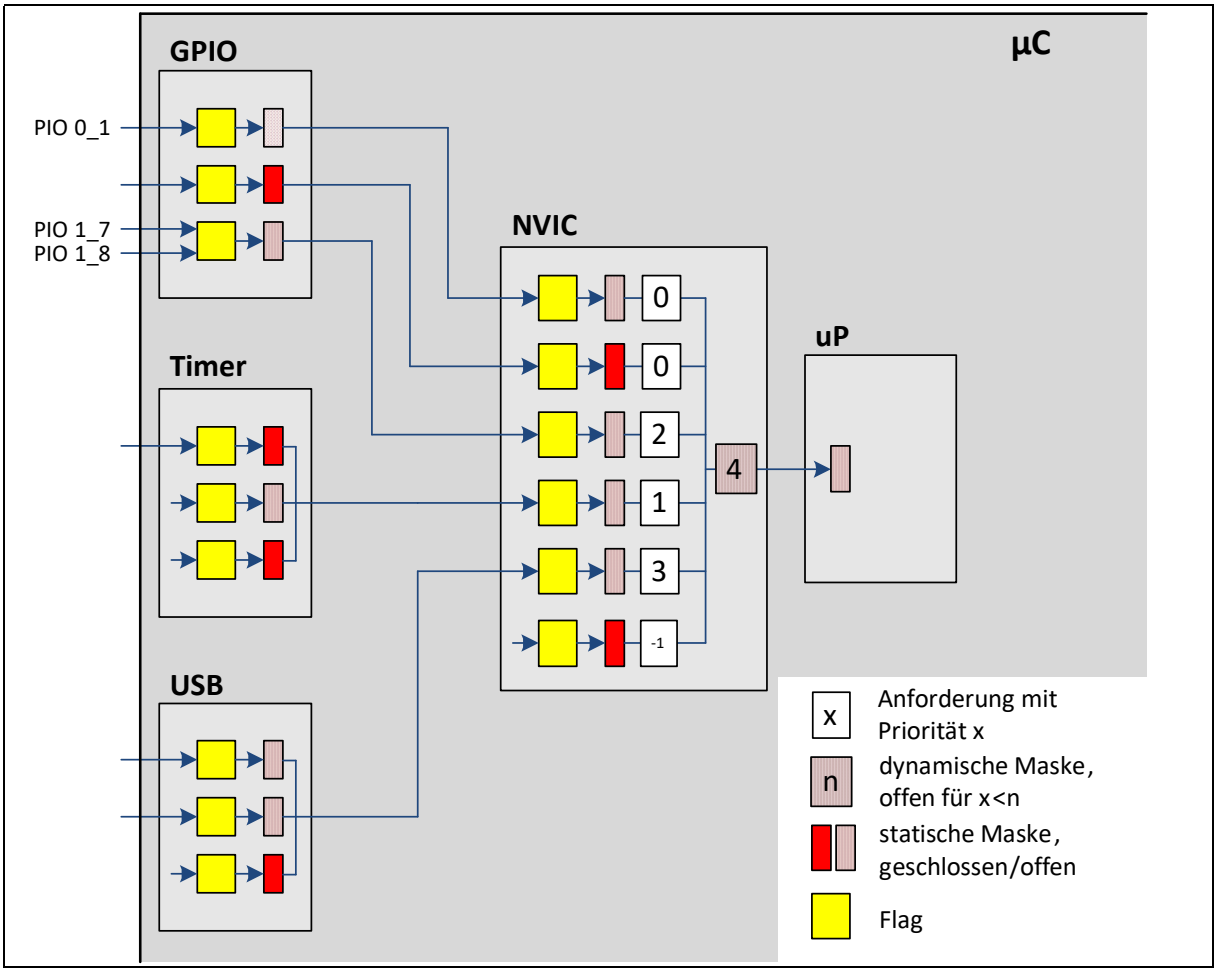

Abbildung 8: NVIC mit Prioritätsvergabe

Man kann per Programm jeder Interruptanforderung eine Priorität geben (weiße Kästchen). Kleinere Zahlen bedeuten eine höhere Priorität. Zudem verwaltet der NVIC selbständig eine dynamische Maske, in der die Priorität des gerade laufenden Programms eingetragen ist. Der NVIC fordert nur dann einen Interrupt beim µP an, wenn die Anforderung eine höhere Priorität als das gerade laufende Programm hat. Im Bild wären alle Interruptanforderungen möglich.
Falls eine Anforderung ankommt (z.B. vom Timer), dann wird diese Anforderung an den µP weitergeleitet und zugleich wird die dynamische Maske auf den Wert 1 gesetzt. Kommen später Anforderungen vom USB (Priorität 3) oder von dem gemeinsamen Interrupt der GPIO (Priorität 2), dann werden zwar die entsprechenden Flags im NVIC gesetzt, aber diese Anforderungen werden nicht an den µP weitergeleitet.
Eine Anforderung vom PIO 0_1 würde dagegen das laufende Programm (die ISR des Timer) unterbrechen und die dynamische Maske hätte dann den Wert 0.
Wird eine ISR beendet, dann stellt der NVIC bei der dynamischen Maske den vorherigen Wert ein. Damit kommen später auch die derzeit zurückgestellten ISR zum Zug.

5 Interrupt, Exception, Traps

Bisher war immer von Interrupts die Rede. Daneben gibt es auch die Begriffe Exception (Ausnahme) und Trap[6] (Falle), die beide weitgehend synonym gebraucht werden. Bei einem Interrupt tritt das auslösende Ereignis (z.B. der Tastendruck) zu einem unvorhersehbaren Zeitpunkt ein, der mit dem aktuellen Programmablauf nichts zu tun hat. Bei einem Trap wird das auslösende Ereignis unmittelbar vom Programm selbst erzeugt. Ein Beispiel wäre die Division durch 0. Trifft der µP auf eine solche Anweisung, dann kann eine Programmunterbrechung ausgelöst werden. Ein anderes Beispiel wäre der Zugriff auf eine Adresse, an der sich kein Speicher oder kein Register befindet. Auch das kann eine Programmunterbrechung auslösen.
Auch hier können ggf. Prioritäten vergeben werden. In Abbildung 8 sieht man im NVIC ganz unten eine Interruptanforderung, die nicht von einem Peripheriemodul kommt. Das weist darauf hin, dass es sich um einen Trap handelt – ein intern erzeugtes Ereignis. Die zugehörige Priorität ist hier -1, so dass der Trap auch innerhalb einer ISR mit der höchsten Priorität eines Peripheriemoduls bearbeitet wird.

Über welche Traps ein µP verfügt ist von der Architektur des µP abhängig. Bei dem Praktikums-µC sind die vorhandenen Traps (hier Exceptions genannt) im UM in Tabelle 421 aufgeführt. Man kann dort sehen, dass für drei Traps die Prioritäten vorgegeben sind und sie sich für 3 weitere programmieren lässt. Tatsächlich handelt es sich bei dem Trap 0 (Reset) um einen Sonderfall, da das unterbrochene Programm nicht fortgesetzt werden kann (Neustart) und bei dem Trap 15 (Systick) um einen Timer, der also sonst zu den Interrupts zu zählen wäre.

6 Nutzung von Interrupts

6.1 Vordergrund- /Hintergrundprogrammierung

Bei der Arbeit mit ISRs nennt man die Anteile des Programms, die von den ISRs gebildet werden, Hintergrundprogramme. Sie laufen „im Hintergrund“ ereignisgesteuert ab. Die Reihenfolge der einzelnen ISRs ist in der Regel nicht vorhersehbar. Das (einzige) in der vom Programmierer vorgegebenen Reihenfolge abgearbeitete Programm (in C beginnend mit *main()*) wird Vordergrundprogramm genannt. In manchen Fällen besteht es aus einer leeren Schleife – dann finden alle Aktionen nur in den ISRs statt.

6.2 Kommunikation mit ISRs

ISRs können weder Parameter übernehmen, noch können sie Werte zurückgeben. Das liegt daran, dass eine ISR ja nicht von einer anderen Funktion aufgerufen wird, die die nötigen Werte liefern oder einen Rückgabewert entgegennehmen könnte. Wenn zwischen ISRs oder dem Vordergrundprogramm Werte ausgetauscht werden müssen, dann muss das über globale Variablen geschehen. Diese Variablen müssen in C als *volatile* gekennzeichnet werden, da sich deren Werte ja zu unvorhergesehenen Zeitpunkten ändern können.

```
1   volatile uint8_t  start ;
2   volatile uint16_t sekunden;
3
4   void zeitgeber_isr(void)
5   {
6     static uint16_t z=0;
7
8     if (start != 0)
9     {
10     sekunden=0 ;
11     z=0;
```

[6] wird im Folgenden als Bezeichnung auch für Exception gebraucht

```
12      start=0;
13    }
14
15    if (z<1000) z++ ;
16    else
17    {
18      z=0;
19      sekunden++;
20    }
21  }
22
23  void warte_sekunden(uint16_t t)
24  {
25    start=1;
26    while (start) {};
27    while (t != sekunden) {};
28  }
```

Listing 3: Beispielcode ISR

Listing 3 zeigt eine ISR *void zeitgeber_isr(void)*, die hier jede Millisekunde einmal aufgerufen werden soll. Zunächst prüft die ISR in Z. 8, ob ein Neustart der Uhr gewünscht wird. Falls ja, dann wird die Uhr (Sekunden und Millisekunden) auf null gestellt (Z. 10,11) und der Startbefehl gelöscht (Z. 12).
In Zeile 15-20 wird dann die Uhr jede Millisekunde weitergeschaltet, wobei nur die Sekunden der globalen Variable *sekunden* sichtbar sind. Die Millisekunden *z* sind nur in der ISR sichtbar. Möchte man nun im Programm genau *t* Sekunden warten, dann gibt man der Uhr den Startbefehl (Z. 25). Nun wartet man darauf, dass die Uhr den Befehl ausführt (Z. 26) – das wird nach spätestens einer Millisekunde der Fall sein. Anschließend wartet man darauf, dass die geforderte Anzahl Sekunden vergangen ist (Z. 27).

6.3 Initialisierung

Es ist sehr wichtig, *alle* notwendigen Initialisierungen vorzunehmen, bevor man Interrupts zulässt. Sinnvoll ist es zudem, eventuell schon gesetzte Flags zu löschen, bevor man erstmals den zugehörigen Interrupt zulässt. Andernfalls bekommt man möglicherweise sofort einen Interrupt, der zu einem längst vergangenen Ereignis gehört.
Die richtige Initialisierung kann durchaus aufwendig sein, weil die benötigten Einstellungen für eine einzige Aufgabe möglicherweise in mehreren Modulen vorgenommen werden müssen.

Für das gegebene Beispiel mit den Tasten am GPIO-Modul sind dies beim Praktikums-µC:

- Modul „System Control Block“:
  Zuordnung des GPIO, an dem Not Aus angeschlossen ist, zu einer eignen ISR (UM, Table 40)
- Modul GPIO:
  Festlegen der Ereigniseigenschaft, die zum Auslösen führt (UM, Table 141 - Table 47) und zudem Sperren/Öffnen der Maske für diese ISR
  Festlegen der GPIO die zu einer gemeinsamen Gruppe gehören (UM, Table 151 – Table 155)
- Modul NVIC:
  Sperren/Öffnen der beiden ISR über eine weitere Maske (UM, Table 61) und setzen der Prioritäten (UM, Table 66 und Table 68).

Auch Flags finden sich zu diesen Ereignissen in zwei Modulen:

- Modul GPIO: Flags müssen manuell gelöscht werden
- Modul NVIC: Flags werden automatisch bei Aufruf der ISR gelöscht