Programmierung in “C” beim Microcontroller

1. What’s new?

1.1. Allgemeine Information über C

Immer mehr Programme und Anwendungen werden in C geschrieben, denn diese Programmiersprache

a) ist **nicht an einen bestimmten Rechnertyp oder ein Betriebssystem gebunden** (sie wurde für UNIX entwickelt, wird aber heute vom Microcontroller bis zur Workstation eingesetzt) und

b) Die geschriebenen C-Programme sind deutlich kürzer als Assemblerprogramme und ermöglichen ebenfalls die Lösung von Problemen, bei denen die „**Maschinennähe**" wichtig ist (ähnlich wie bei der Assemblerprogrammierung kann man auf einzelne Adressen, Bits und Portpins zugreifen).

c) Viele Problemlösungen werden bereits beim Kauf in Form von **"Bibliotheken"** mitgeliefert und können auf einfachste Weise vom Anwender benützt werden (von der Tastatureingabe beim PC über mathematische und logische Funktionen bis zur Bildschirmausgabe ist alles vorhanden.....).

d) Die Weiterentwicklungen (C++, Turbo-C++, VisualC++ usw) ermöglichen eine "**objektorientierte Programmierung**".
Hierbei wird das Prinzip der bisherigen **prozeduralen Programme** mit ihrem Aufbau

“nach einer Anweisung folgt die nächste. Ist diese ausgeführt, kommt die übernächste....."

verlassen. Vielmehr werden Daten oder Messwerte samt allen jemals nötigen Programmanweisungen und evt. auch erst später erforderlichen Bibliotheksfunktionen zu einem für sich geschlossenen Gebilde, dem

Objekt

zusammengefasst. An diesem Objekt kann später jederzeit herumgespielt werden, dabei wird erweitert, ausgewertet, gemessen ..... und das Objekt kann selbst wieder Informationen mit anderen Objekten austauschen und diese verarbeiten. Sogar die menschliche Fähigkeit der “Vererbung” ist möglich!

1.2. Einige Worte zur Microcontroller-Programmierung in C

Die Programmierung verschiedener Prozessortypen (z. B. 80 535, 68HC11, PIC...) wurde durch C beträchtlich erleichtert, da lediglich der Compiler gewechselt und Rücksicht auf bestimmte Eigenheiten (Port-Anzahl, Interrupt-Möglichkeiten usw.) genommen werden muß. So können einmal geschriebene Programme bei einem Systemwechsel sehr schnell übertragen und angepaßt werden.
Im Gegensatz zum PC geht es aber bei der Controllerprogrammierung meist weniger um die Ein- oder Ausgabe bzw. Bearbeitung von Texten. Im Vordergrund steht hier üblicherweise die Erfassung, Manipulation und Ausgabe irgendwelcher Messwerte oder Ansteuersignale, die Abfrage von Tasten oder Schaltern, die Verwirklichung von Zeitschleifen oder Verzögerungen, die Erzeugung bestimmten Signalformen oder Frequenzen, die Bereitstellung von Bitmustern usw.
Deshalb muss unsere Arbeit (und der Einstieg) beim Microcontroller auf eine andere Weise ablaufen, als es die Standard-C-Lehrbücher normalerweise vorsehen. Wir können viele Themen zunächst völlig ignorieren und zurückstellen, da sie hier erst viel später oder äußerst selten auftauchen.
(Eine verschämte Anmerkung ist deshalb nötig: wer den Microcontroller in C programmieren kann, muss folglich noch dazulernen, damit auch die Erstellung von PC-Programmen klappt...).

2. Erste Schritte in C

2.1. Prinzipieller Grundaufbau eines einfachen C-Programmes

Am Anfang **aller** C-Programme stehen die sogenannten "**Header-Dateien**". Sie sorgen dafür, daß die nötigen Bibliotheksfunktionen zur Verfügung stehen und werden mit

#include <**.h>**

aufgerufen.

Für unsere Controllerprogrammierung brauchen wir grundsätzlich **immer** mindestens folgenden Vorspann:

```
#include <reg515.h>          //Registerdefinitionen für unser Compuboard mit 80C535
```

```
#include <stdio.h>                    //Bibliothek mit "Standard-Input-Output" - Anweisungen
```

Anschließend können bestimmte **Vereinbarungen** -- z. B. über einen besonderen Namen bei einem Portpin oder einem kompletten Port -- getroffen werden.

Beispiel: Wir wollen am Portpin P1.0 ein Signal ausgeben und beobachten. Dieser Pin soll deshalb "**ausgang**" heißen.
Das geschieht mit folgender Anweisung:

```
sbit ausgang=0x90;
```

(Erklärung: sbit heißt "special Bit". "0x90" ist die **korrekte Schreibweise einer Hex-Zahl in C** und bedeutet, daß mit dem Wort "ausgang" das erste Bit der Port 1 - Adresse 090H gemeint ist).

Darnach folgen die eigentlichen Programmanweisungen. Sie sind **immer** in sogenannten **"Funktionen**" enthalten.

Achtung:
Wenn eine Funktion **definiert** wird, folgen **direkt hinter dem Funktionsnamen nur zwei runde Klammern, dahinter aber nix mehr (also NIE ein Strichpunkt)!!**

Wenn eine Funktion **aufgerufen (= also verwendet)** wird, dann **muss der Strichpunkt hinter der Funktion stehen...**

Die nötige "**Start- und Grundfunktion**" (= sozusagen das **"Hauptprogramm**") trägt den Namen

```
main()
```

Leider gibt es in der Zwischenzeit mehrere Entwicklungsstufen von C. Deshalb kann bei bestimmten Versionen auch die Formulierung

void main() oder auch **void main(void)**

nötig sein, um Fehlermeldungen zu vermeiden.....

In der nächsten Zeile steht nur **eine geöffnete, geschweifte Klammer**. Sie eröffnet die Sammlung von Anweisungen, die durchgeführt werden sollen.

Niemals vergessen: jede Anweisung innerhalb der geschweiften Klammer muss immer durch einen Strichpunkt beendet werden!!!!!!!!!

Hinter den Programmschritten wird (wieder in einer neuen, getrennten Zeile) die **geschweifte Klammer geschlossen** und damit die Funktion komplettiert.

Hier folgt nun unser erstes Microcontroller-Programm. Damit wollen wir am Portpin P1.0 pausenlos den Pegel invertieren und auf diese Weise ein Rechtecksignal mit hoher Frequenz erzeugen.

```
#include <stdio.h>                          /*Header-Dateien*/
#include <reg515.h>

sbit ausgang=0x90;                          //Deklaration des Portpins P1.0 als Ausgang

void main(void)                             //Hauptprogramm
{
        while(1)                            //Endlosschleife
        {
                ausgang=~ausgang;           //Ausgang invertrieren
        }
}
```

2.2. Umgang mit der Software „KEIL µvision" sowie dem Simulator und Debugger „dscope"

Siehe hierzu die gesonderte Anleitung zum Umgang mit der KEIL-Software. Sie enthält auch alle erforderlichen Voreinstellungen und Handgriffe, um das Programm zum Leben zu erwecken.

---

3. Übersicht: zulässige Datentypen in C-Programmen

Für die allgemeine C-Programmierung gilt der folgende “Vorrat” an Zahlentypen.

Verständliche Bezeichnung	Offizieller Name	Wertebereich	Belegter Speicherplatz
Einzelnes Bit	bit	0 oder 1	1 Bit
Positive ganze Zahlen von Null bis 255	**unsigned char**	0 bis 255	1 Byte
Ganze Zahlen von -127 bis +127	**(unsigned) char**	-127 bis +127	1 Byte
Positive ganze Zahlen zwischen Null und 65535	**unsigned short int** oder einfach nur **unsigned int**	0 bis 65535	2 Byte
Ganze Zahlen zwischen -32768 und +32767	(signed) **short int** oder einfach nur **int**	-32768 bis +32767	2 Byte
Positive ganze Zahlen zwischen Null und 4 294 967 295	**unsigned long**	0 bis 4 294 967 295	4 Byte
Ganze Zahlen zwischen -2 147 483 648 und +2 147 483 647	**(signed) long**	-2 147 483 648 bis +2 147 483 647	4 Byte
Gleitkommazahlen im Bereich von 10^{-37} bis 10^{+37} mit 6 Nachkommastellen	**float**	10^{-37} bis 10^{+37}	8 Byte
Gleitkommazahlen im Bereich von 10^{-308} bis 10^{+308} mit 16 Nachkommastellen	**double**	10^{-308} bis 10^{+308}	8 Byte

Bei unseren 8-Bit-Microcontrollern ist es aber vernünftig, **möglichst nur die beiden Typen**

Integer (belegt 2 Byte) oder **Character** (belegt 1 Byte)

zu verwenden, da sonst bei irgendwelchen Operationen **ungeheuer lange Bearbeitungszeiten** auftreten und der **Speicherplatz** des Controllers sehr schnell aufgefressen ist! Außerdem sind die „float“ und „double“-Funktionen bei der KEIL-**Test-CD sowieso gesperrt**...

Die Deklaration eines Typs muss übrigens immer **innerhalb derjenigen Funktion** geschehen, in der diese Variable verwendet werden soll.

Der Vorgang selbst ist höchst einfach.

1. Beispiel:

```
void main (void)
{
        unsigned int x;                 //Definition von x als Zahl zwischen 0 und 65535
.........
}
```

2. Beispiel:

```
void main (void)
{
        char x;                         //x liegt zwischen -128 und +127
........
```

}

4. Die Zuweisung von Werten bei Variablen

Vorsicht:
Das **Gleichheitszeichen =** hat in der Sprache C eine ganz andere, eigenartige Aufgabe:

Damit kann einer Variablen alles mögliche aufs Auge gedrückt werden (es ist nämlich ein "**Zuweisungs-Operator**").

1. Beispiel: **Zuweisung eines bestimmten Zahlenwertes beim Programmstart ("Initialisierung")**

int strom = 5; bedeutet, dass für die Variable "strom" der Startwert 5 gespeichert wird (und "int" schreibt die "Ganzzahl-Darstellung mit positivem oder negativem Vorzeichen" vor).

2. Beispiel: **Hochzählen einer Zahl**

summe = summe + 5 bewirkt, dass zum vorhandenen "summe-Ausgangswert" die Zahl 5 addiert und das Ergebnis wieder unter "summe" gespeichert wird.

3. Beispiel: **Zuweisung eines Rechenergebnisses**

widerstand = spannung / strom

strom = spannung / widerstand

spannung = strom * widerstand wären Anwendungen des Ohm'schen Gesetzes, bei denen das Rechenergebnis der rechten Seite anschließend in den Speicherplatz der linken Seite geschoben wird.

5. Schleifen

Sie dienen zur Wiederholung von Programmteilen. Hierbei gibt es **drei verschiedene Möglichkeiten**:

a) Eine Bedingung am **Schleifenanfang** erzwingt eine (ganz exakt und genau berechenbare) **Zahl der Wiederholungen:** for-Schleife

b) Es wird **nur solange** wiederholt, wie eine am **Schleifenanfang** stehende Bedingung **erfüllt** ist: while-Schleife

c) Die Schleife wird prinzipiell erst mal durchlaufen. **Am Ende** des **ersten Durchganges** steht eine Prüfbedingung. Durch sie wird die **Zahl der Wiederholungen** festgelegt: do-while-Schleife

5.1. Vergleichsoperatoren

In den Prüfbedingungen unserer Schleifen werden immer Vergleiche angestellt. Deshalb heißen sie auch

"Vergleichsoperatoren".

Folgende Vergleichsoperatoren stehen uns zur Verfügung:

<	bedeutet **"kleiner als"**
>	bedeutet **"größer als"**
<=	bedeutet **"kleiner oder gleich"**
>=	bedeutet **"größer oder gleich"**
!=	bedeutet **"ungleich**"
==	bedeutet **"gleich"**

5.2. Die "for" - Schleife

Sie steht immer **am Anfang** einer Schleife und **erzwingt solange eine Wiederholung**, bis die in der Klammer auf “for” folgende **Bedingung nicht mehr wahr** ist.

(In der Klammer steht üblicherweise eine Vorschrift , mit der eine Zahl x bis zu einem bestimmten Höchstwert inkrementiert wird).

Das kann für folgende Anwendungen genutzt werden:

a) Exakt **vorgeschriebene Anzahl von Wiederholungen eines Programmteiles**.

Beispiel: Der Ausgang soll 50mal invertiert werden.

```
{
.........
        for(x=0; x<=50; x++)
        {
            ausgang =~ausgang;
        }
.........
}
```

c) Programmierung von **Warte- und Zeitschleifen**.

Beispiel: Programmierung einer Zeitschleife.

```
{
........
        int x;
        for(x=0; x<=5000; x++);
.......
}
```

ODER SO:

```
{
......
    int x;
    for(x=0; x<=5000; x++)
        {   }
......
}
```

5.3. Die “while”-Schleife

Prinzip: a) Wenn die am Schleifenanfang stehende Bedingung **nicht gilt**, dann wird die gesamte Schleife **übersprungen**

b). **Solange** die am Schleifenanfang stehende Bedingung **gilt**, wird die Schleife **wiederholt.**

Das ist natürlich z. B. eine ideale Methode zur Abfrage von Schaltern, Tasten, Sensoren....., um anschließend einen bestimmten Vorgang auszulösen.

Hinweis: ein wichtiger Sonderfall ist die Formulierung “while (1)”.

Da die Bedingung in der Klammer immer stimmt, erhält man damit eine **Endlosschleife**.

1. Beispiel: Solange “taste1” an LOW-Pegel liegt, wird “ausgang” invertiert.

```
{
.............
                while (taste1==0)
                {
                        ausgang=~ausgang;
                }
.............
}
```

2. Beispiel: Es wird die Invertierung des Ausgangs um eine Zeitschleife ergänzt und alles dauernd wiederholt. So entsteht z. B. unser berühmter 1 kHz-Ton am Ausgangspin P1.0.

```
sbit ausgang=0x90;
void main (void)
{
        while(1)
        {
            unsigned char x;
            for(x=0; x<=250; x++);
            ausgang=~ausgang;
        }
}
```

3. Beispiel: Der 1 kHz-Ton lässt sich auch ausschließlich mit “while”-Schleifen realisieren:

```
setb ausgang=0x90;
void main (void)
{
    unsigned char x;
    while(1)                    //Endlosschleife
    {
        x=0;                    //fange mit x = 0 an
        while (x<=250)          //inkrementiere x bis 250
            {
            x++
            }
        ausgang=~ausgang;
    }
}
```

5.4. Die “do while” - Schleife

Bei der gerade besprochenen “while” - Schleife stand die Prüfbedingung **vor** den Anweisungen. Sie heißt deshalb “kopfgesteuerte Schleife”.

Im Gegensatz dazu wird bei der “do while” - Schleife die Prüfbedingung an das **Ende der Anweisung** gesetzt (= fußgesteuerte Schleife). Dadurch erzwingt man **mindestens einen Durchgang** durch das Schleifenprogramm, der anschließend -- solange die Bedingung erfüllt ist -- beliebig oft wiederholt werden kann.

Auch damit lässt sich sehr elegant eine Tastenabfrage durchführen.
Beispiel: Die Schaltung macht dann weiter, wenn die Taste 1 auf LOW gelegt wird. Das kann z. b. der Kontakt einer Alarmanlage sein.

```
{
........
        do
        {   }                   //tue gar nix!
        while (taste1==1);      //solange taste1 geöffnet ist
........
}
```

6. Auswahl

Bisher haben wir folgende Programmstrukturen kennen gelernt:

a) In den vorhergehenden Kapiteln die **sequentielle Struktur** (= ein Schritt nach dem anderen wird abgearbeitet, bis alles erledigt ist).

b) Speziell in Kapitel 5 die **Wiederholstruktur** (for, do, while).

Deshalb wollen wir uns jetzt die **Auswahl** (also die Wahl zwischen verschiedenen Alternativen) ansehen.

6.1. Die “if” - Anweisung

Bei der “if” - Anweisung werden die folgende Anweisung (oder ein ganzer Anwendungsblock) **nur dann ausgeführt**, wenn die **hinter ”if” stehende Bedingung wahr** ist. Auch das eignet sich vorzüglich für irgendwelche Entscheidungen oder Abfragen.

Beispiel: Wenn “taste_1” gedrückt ist, soll “ausgang” gesetzt werden. Drückt man dagegen “taste_2”, wird “ausgang” wieder gelöscht (Reset). Drückt man “taste_3”, so wird geblinkt.

```
{
.....
        if (taste_1==0)
        {
            ausgang=1;
        }

        if (taste_2==0)
        {
            ausgang=0;
        }

        if (taste_3==0)
        {
            blink();
        }
.....
}
```

6.2. Die “if - else “ - Anweisung

Mit diesem Zusatz kann **nur zwischen zwei Alternativen** gewählt werden.

Beispiel: “taste_1” soll “ausgang” auf 1 setzen. Im andern Fall soll “ausgang” gleich Null sein.

```
{
.....
    if (taste_1==0)
    {
        ausgang=1;
    }

    else
    {
        ausgang=0;
    }
.....
}
```

6.3. Die “switch” - Anweisung

Mit Hilfe der “**switch**”- Anweisung ist es möglich, aus einer **ganzen Reihe von Alternativen** (in Form der **“case”** - **Liste**) den gewünschten Fall herauszufischen und eine Reaktion auszulösen.

Achtung:	a) Es ist zulässig, dass mehrere Möglichkeiten gültig sind und dieselbe Wirkung haben. Sie werden einfach nacheinander aufgelistet und dann mit der zutreffenden Vorschrift versehen. b) Passt **keine der Möglichkeiten**, dann wird die "**default**" - Einstellung genommen und ausgeführt.

Beispiel: In einem Register mit dem Namen "ergebnis" ist ein Messergebnis oder eine Zahl gespeichert.
Abhängig vom genauen Wert sollen nun bestimmte Reaktionen erfolgen.

```
{
.............

    switch (ergebnis)
    {
        case 0x00:
        case 0x10:
        case 0x20:
        case 0x30:
                ausgang=1;
                break;

        case 0x40:
        case 0x50:
        case 0x60:
        case 0x70:
                ausgang=0;
                break;

        default:
                ausgang=~ausgang;
                break;
    }
..........
}
```

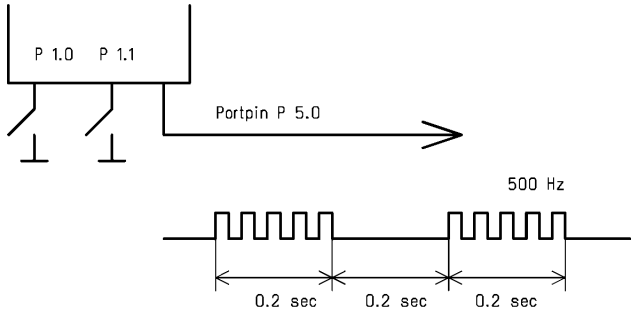
7. C - Anwendungsbeispiele

7.1. Alarmanlage

Aufgabe: es ist eine Alarmanlage aufzubauen, die folgende Eigenschaften aufweist:

a) Am Portpin P 1.0 ist die Taste T1 angeschlossen. Wird sie betätigt, dann erzeugt der Controller am Portpin P 5.0 ein Alarmsignal mit dem angegebenen Verlauf.

b) Das Alarmsignal kann nur durch Schließen einer weiteren Taste T2 am Portpin P1.1 wieder abgeschaltet werden. Anschließend soll die Alarmanlage sofort wieder "scharf" sein.



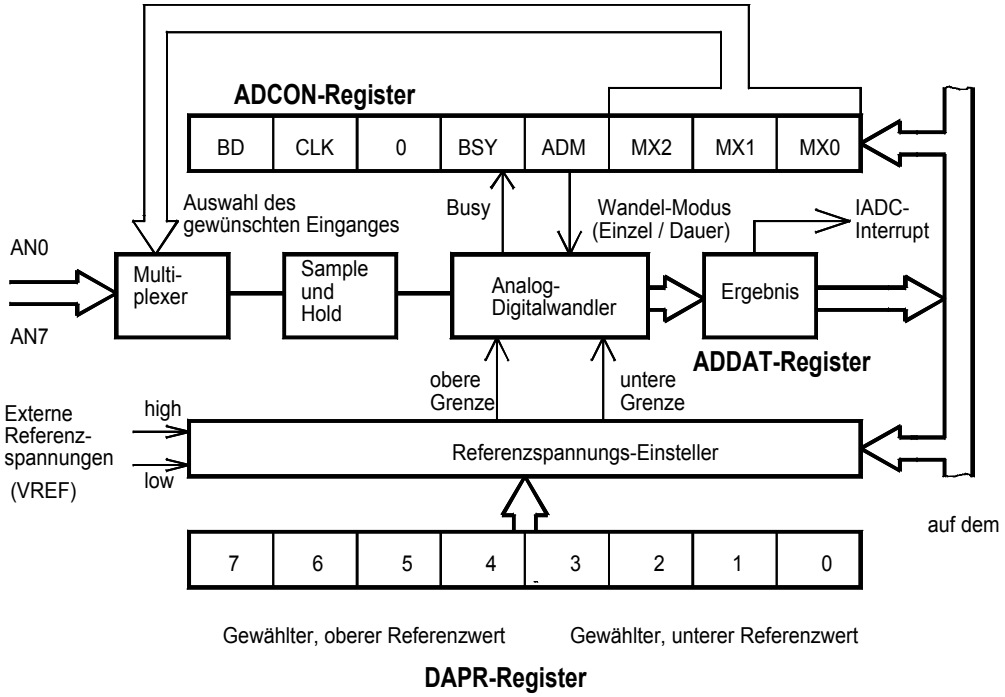
Lösung:

```
#include <stdio.h>
#include <reg515.h>

/************************** Konstanten ******************************/

sbit ausgang=0xF8;                      //Portpin P 5.0 als Audio-Ausgang
sbit alarm=0x90;                            //Portpin P 1.0 als Alarm-Auslöser
sbit reset=0x91;                        //Portpin P 1.1 als Reset-Taste

/************************ Hauptprogramm ******************************/
void main(void)
{
    unsigned  int x,y;                  //x und y als 16 - Bit - Zahlen
    while (1)                         //Endlosschleife
    {
        do                              //Abfrage der Taste P1.0
        {   }
        while (alarm==1);

        while(1)                        //Endlosschleife für Alarmsignal
        {
            for (y=0; y<=200; y++)      //200 Millisekunden piepsen
            {
                for (x=0; x<=100; x++);
                ausgang=~ausgang;
            }


            if (reset==1)               //Reset nicht gedrückt?
            {
                ausgang=0;              // dann 200 Millisekunden Ruhe
                for(x=0; x<=12000; x++);
            }
            else                        //sonst: Alarm aus, weil RESET gedrückt
            {
                break;                  //Rücksprung aus der Alarmschleife zur Tastenabfrage
            }
        }
    }
}
```

7.2. Umgang mit dem AD - Wandler

Arbeitsweise des A-D-Teiles auf dem Chip

Der A-D-Teil des 80C535 ist nach folgendem Übersichts-Schaltplan aufgebaut **und verträgt nur Analogsignale, die zwischen Null Volt und +5 Volt** liegen (Die Controller**eingänge** sollten deshalb mit geeigneten **Schutzschaltungen versehen sein!).**



Beim Einsatz dieses Wandlers müssen wir uns um 3 Register kümmern:

a) **ADCON-Register**

Mit den **unteren 3 Bits (MX0....MX2)** wird der **gewünschte Analogeingang** ausgewählt.

ADM bestimmt den **Wandler-Modus** (HIGH ergibt Dauermessung, LOW dagegen Einzelmessung).

BUSY ist HIGH, solange gewandelt wird und geht auf LOW, sobald das Ergebnis an das ADDAT-Register übergeben wird. Damit lässt sich leicht eine **Abfrageschleife für die Wartezeit** während der AD-Wandlung programmieren.

(Die übrigen Bits erfüllen andere Aufgaben: BD schaltet den internen Baudratengenerator für die Serielle Schnittstelle ein, CLK ermöglicht die Ausgabe der durch 12 geteilten Oszillatorfrequenz an Portpin P1.6, “0” heißt Null und muss immer auf Null bleiben).

b) **ADDAT-Register**

Hier kann mit einem MOV-Befehl das Wandelergebnis abgeholt werden. Der Controller meldet das **Wandlungsende** aber nicht nur durch das **Zurücksetzen des BUSY-Flags**, sondern auch durch das **Setzen des IADC-Interrupt-Flags** (es befindet sich im IRCON-Register).

Soll dadurch in eine Interrupt-Routine gesprungen werden, dann muss man vorher nicht nur den EADC-Interrupt, sondern auch die “globale Interrupt-Freigabe” per Software einschalten (sie finden sich in den Registern IEN0 und IEN1).

Daraus erkennt man, dass man als Programmierer das Ende des Wandelvorgang entweder durch dauerndes Abfragen des Busy-Flags oder durch Auslösung des IADC-Interrupts erkennen bzw. auswerten kann....

c) **DAPR-Register**

Dazu vorher einige Worte zum Eingangsspannungsbereich des AD-Wandlers:

Er benötigt immer **zwei Referenzpotentiale** zum korrekten Arbeiten:

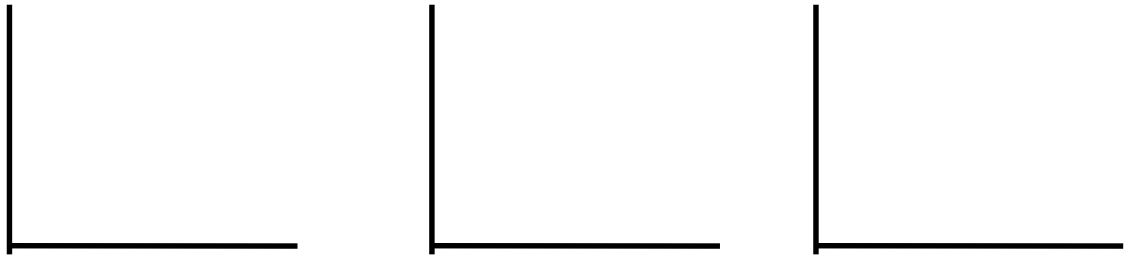
a) der **tiefere** Wert heißt "V_{AGND}" und entspricht meist dem **Massepotential** (= Null Volt).

b) der **höhere** Wert heißt **"V_{AREF}"** und wird normalerweise durch die **Betriebsspannung (+5V)** gebildet.

Die (interne oder externe) Referenzquelle speist auf dem Chip einen **Spannungsteiler aus mehreren Widerständen**, zwischen dessen Abgriffen elektronisch umgeschaltet werden kann. Auf diese Weise lässt sich

1) der **"Endausschlag des Messbereiches**" bei kleinen Signalen verändern oder

2) die Auswertung von **kleinen Änderungen** bei einer **relativ großen Gleichspannung** erleichtern.



Der Spannungsbereich zwischen diesen beiden Referenzpunkten wird stets mit einer Genauigkeit von 8 Bit aufgelöst, d. h., er ergibt max. 255 verschiedene Digital-Werte in Form eines 8-Bit-Wortes. (Dieses 8-Bit-Wort steht am Ende des Wandelvorganges im Register **"ADDAT**" und muß dort abgeholt werden!).

Vorsicht: der Hersteller verlangt **mindestens 1 Volt Unterschied** zwischen den beiden Referenzpunkten. Damit beträgt die **feinste und kleinste Auflösung bei einer Messung mindestens ca. 4 mV** bei 255 Stufen.....

Bei hohen Präzisionsanforderungen oder starken Störungen auf der Versorgungsleitung kann auch eine **getrennte, externe Referenzspannungsquelle** benützt werden. Dazu zieht man 2 Jumper auf dem Board und führt die externe Spannung über den entsprechenden Steckverbinder zu.

Die **Programmierung der ausgewählten Referenzpunkte** sowie der **Beginn der A-D-Wandlung** erfolgt immer durch **Beschreiben des DAPR-Registers** mit einem entsprechenden Codewort!

Für die Auswahl der beiden Referenzpunkte gilt dafür folgende Codewort-Tabelle:

Oberes Nibble des DAPR	**Oberer Ref.-Punkt**	**Unteres Nibble des DAPR**	**Unterer Ref.-Punkt**
0000	5,0V	0000	0V
0001	-----	0010	0,3125V
0010	-----	0010	0,625V
0011	-----	0011	0,9375V
0100	1,25V	0100	1,25V
0101	1,5625V	0101	1,5625V
0110	1,875V	0110	1,875V
0111	2,1875V	0111	2,1875V
1000	2,5V	1000	2,5V
1001	2,8125V	1001	2,8125V
1010	3,125V	1010	3,125V
Oberes Nibble des DAPR	**Oberer Ref.-Punkt**	**Unteres Nibble des DAPR**	**Unterer Ref.-Punkt**
1011	3,4375V	1011	3,4375V

1100	3,75V	1100	3,75V	
1101	4,0625V	1101	------	
1110	4,375V	1110	------	
1111	4,6875V	1111	------	

1. Aufgabe:
Speisen Sie den Analogeingang AN0 mit einer Gleichspannung zwischen 0 V und +5 V. Schalten Sie den Wandler auf "Dauermessung" und geben Sie das Wandelergebnis an Port 1 aus. Benützen Sie bei der Programmerstellung das "BUSY"-Flag des Wandlers. Beobachten Sie mit der "LED-Status-Anzeigeplatine" die Pegel der Pins an Port 1 und drehen Sie dabei am Einstellpoti.
Schreiben Sie das Programm unter Verwendung der **"IF - ELSE" - Anweisung**.

Lösung:

```
#include <stdio.h>
#include <reg515.h>

/************************** Konstanten ******************************/
sbit busy=0xDC;                 /Busy-Flag deklarieren

/*********************** Hauptprogramm ****************************/
void main(void)
{
        ADCON=0x08;             //Dauermessung an Eingang AN0 waehlen
        DAPR=0x00;              //Start der Messung mit Bereich 0...5 Volt
        while(1)                //Endlosschleife
        {
            if(busy==1)         //Wandlungsende abwarten
            {   }
            else
            {
            P1=ADDAT;           //Übergabe des Ergebnisses an Port P1
            }
        }
}
```

---

2. Aufgabe:
Schreiben Sie dieses Programm nochmals **nur** unter Verwendung von **"WHILE" - Anweisungen**.

Lösung:

```
#include <stdfio.h>
#include <reg515.h>

/************************** Konstanten ******************************/
sbit busy=0xDC;                 //Busy-Flag deklarieren

/*********************** Hauptprogramm ****************************/
void main(void)
{
        ADCON=0x08;             //Dauermessung an Eingang AN0 wählen
        DAPR=0x00;              //Start der Messung mit Bereich 0...5 Volt

        while(1)                //Endlosschleife
        {
            while(busy);        //Wandlungsende abwarten
            P1=ADDAT;           //Übergabe des Ergebnisses an Port P1
        }
}
```

3. Aufgabe: Schreiben Sie dieses Programm nochmals unter **Verwendung des IADC - Interrupts.**

Schalten Sie dazu den Wandler auf “Einzelmessung”. Schreiben Sie eine IADC- Interrupt-Service-Routine, durch die das Wandelergebnis an Port P1 übergeben, das auslösende Flag gelöscht und der Wandler wieder neu gestartet wird.

Bitte daran denken:
Am Anfang des Programms nicht vergessen, das Interruptflag IADC zu löschen sowie alle Interrupts mit EAL und den AD-Wandler-Interrupt mit EADC freizugeben.

Lösung:

```
#include <stdio.h>
#include <reg515.h>


void main(void);                    //Benützte Funktionen (= „Prototypen“) anmelden
void AD_ISR(void);

void main(void)
{
    ADCON=0x00;                     //Eingang AN0 und Einzelmessung wählen
    IADC=0;                         //Interrruptflag löschen
    EAL=1;                          //Alle Interrupts freigeben
    EADC=1;                         //AD-Interrupt freigeben
    DAPR=0x00;                      //Messung starten

    while(1);                       //Endlosschleife
}


void AD_ISR(void)   interrupt 8     //Interrupt-Service-Routine
{
    IADC=0;                         //Interruptflag löschen
    P1=ADDAT;                       //Ergebnis an P1 übergeben
    DAPR=0x00;                      //Neue Messung starten
}
```

Kleiner Hinweis zur Ermittlung der richtigen “Interrupt - Nummer”, die in der Interrupt-Service-Routine **immer** angegeben werden muss:

1. Schritt: Man nehme (aus der Tabelle des Datenblattes oder aus diesem Manuskript) die **Hex - Adresse des** betreffenden **originalen Interruptvektors** und wandle sie in eine **Dezimalzahl** um.

Beispiel: AD - Wandler - Interrupt - Vektor hat Adresse 43hex. Das entspricht der Zahl 67.

2. Schritt: Man **dividiere diese Adresse durch 8** und lasse beim Ergebnis **eine Kommastelle** zu.

Beispiel: 67 : 8 = 8,3

3. Schritt: Die **Zahl vor dem Komma** (hier: 8) beim Ergebnis der Division ist **die gesuchte Interrupt - Nummer**. Sie muss bei der Interrupt - Service - Routine eingetragen werden.