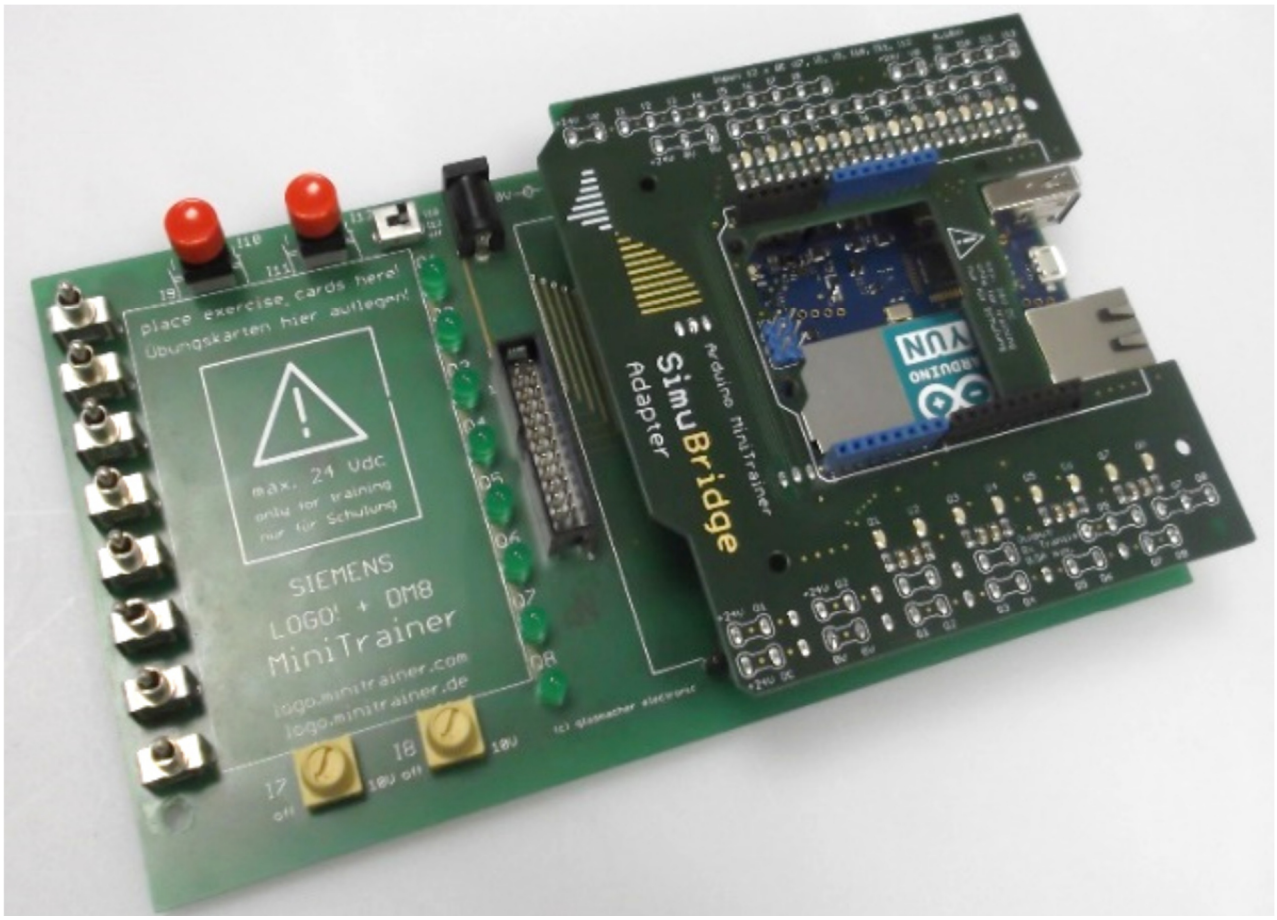
FELTRON ZEISSLER

MINITRAINERSCHULE

-AUFBAU-

für den ***Arduino / Genuino***



Modul C

EIGENE FUNKTIONSBAUSTEINE ERSTELLEN

Reinhold Grewe / Klaus Machalek

Um die umfangreichen Möglichkeiten zu vermitteln und den unterschiedlichen Vorkenntnissen gerecht zu werden, wurde das Konzept der *MiniTrainerSchule* in zwei Hefte unterteilt. Zum einen die ***MiniTrainerSchule-Basis*** mit den Modulen A und B:

- Modul A : Arduino-*MiniTrainer* - und Grundlagen der Programmierung
- Modul B : Programmierübungen elektrischer Grundschaltungen mit einer fertigen Baustein-Bibliotheken

Und zum anderen die ***MiniTrainerSchule-Aufbau*** mit dem Modul C :

- Modul C : Programmierung eigener Funktionen (Bausteine)

Alle Lösungen und Beispiele sind im jeweiligen Kapitel mit abgedruckt. Sie sind zusätzlich auch im Online-Shop von Feltron-Zeissler GmbH downloadbar.

Haftungsausschluss

Der Inhalt der MiniTrainerSchule ist ausschließlich zu Ausbildungszwecken erstellt. Wir haben den Inhalt und die Übungen sorgfältig getestet. Die Autoren übernehmen keinerlei Gewähr für die Aktualität, Richtigkeit und Vollständigkeit der bereitgestellten Informationen in dieser MiniTrainerSchule. Haftungsansprüche gegen die Autoren, welche sich auf Schäden materieller oder ideeller Art beziehen, die durch die Nutzung oder Nichtnutzung der dargebotenen Informationen bzw. durch die Nutzung fehlerhafter und unvollständiger Informationen verursacht wurden, sind grundsätzlich ausgeschlossen, sofern seitens der Autoren kein nachweislich vorsätzliches oder grob fahrlässiges Verschulden vorliegt. Die Autoren behalten es sich ausdrücklich vor, Teile der Seiten ohne gesonderte Ankündigung zu verändern, zu ergänzen oder zu löschen.

Autoren: Reinhold Grewe / Klaus Machalek

1. Auflage 2017 (Stand 12/17)

MODUL C:
PROGRAMMIERUNG EIGENER BAUSTEINE

Lernziele:

Während im Modul B fertige Bausteine benutzt werden, soll im Modul C das Schreiben eigener Funktionsbausteine behandelt werden. Dazu werden die in Modul A vorgestellten Sprachelemente genutzt und die Beschreibung wird um einige Elemente erweitert.

Voraussetzungen : Kenntnisse aus den Modulen A und B

Wenn Sie dieses Modul bearbeitet haben, können Sie Bausteine wie Zähler und Timer selbst erstellen und eine eigene Bibliothek aufbauen.

INHALTSVERZEICHNIS MODUL C

INHALT

1 RÜCKBLICK AUF DIE MODULE A UND B

Im ***Modul A*** werden die Grundlagen der Arduino-Programmierung behandelt. Um die Übersichtlichkeit und damit die Lesbarkeit der Programme zu erhöhen, wurde die Bibliotheksdatei "MyArduinoLib.h" entwickelt. In dieser Datei sind die Definitionen und Funktionen zusammengefast, die für den Betrieb des MiniTrainers wichtig sind, aber in der Regel nicht mehr verändert werden müssen.

Das sind im Detail:

- Definitionen zur Anpassung der Bezeichner „I“ (Eingänge) und „Q“ (Ausgänge). Damit kann im Programm Beispielsweise mit I3 oder Q5 statt mit den Pin-Bezeichnungen des Arduino gearbeitet werden.
- Die Funktion setPins() übernimmt das Setzen der Prozessorpins als Input oder Qutput.
- Die Funktion MyWriteByte (byte, ByteToWrite) zum Schreiben eines kompletten Bytes an die Ausgänge.
- Die Funktion MyReadByte() zum Lesen eines kompletten Bytes von den Eingängen.

Die "MyArduinoLib.h" wird auch in den Modulen B und C benutzt.

Im ***Modul B*** werden die Wirkungsweise und die Anwendung der Grundschaltungen erläutert und in Übungen vertieft. Um einen solchen Baustein selbst zu erstellen, sollte die Wirkungsweise verstanden sein.

2 ZIEL UND VORGEHENSWEISE IN MODUL C

Ziel dieses Moduls ist das Erstellen eigener Bibliotheken mit eigenen Funktionsbausteinen.

Zunächst werden die Grundanforderungen an Funktionsbausteine beschrieben. Als Erweiterung der Sprachbeschreibung werden Arrays eingeführt. Die Funktion als Basis der Bausteine wird aus den anderen Modulen wiederholt und die Notwendigkeit von Baustein-Instanzen erläutert und programmtechnisch gelöst.

Weil es eine hohe Erwartung an Bibliotheken ist, getestete Bausteine entnehmen zu können, wird auch das Thema „Systematisches Testen“ kurz gestreift.

Für eine Auswahl von Bausteinen aus Modul B wird der innere Aufbau beispielhaft erläutert. Als Übungsaufgabe sollen die Bausteine neu entwickelt und in eine Bibliothek integriert werden. Zur Überprüfung des Ergebnisses werden die aus Modul B bekannten Übungsaufgaben und Übungskarten verwendet.

Die beispielhaft beschrieben Lösungen stellen einen von mehreren Lösungswegen dar. Sie sollen anregen, sich über andere Lösungswege Gedanken zu machen und die Erkenntnisse für die Entwicklung eigener Bausteine zu nutzen.

3 Grundsätzliche Anforderungen an einen Baustein

3.1 Übergabe von Eingangsdaten und Rückgabe eines Ergebniswertes

Die Bausteine benötigen in der Regel Eingangsdaten (auch Parameter genannt) mit denen sie arbeiten. Bei einem eingangsverzögerten Timer sind dieses z.B. das Start-Signal und die Verzögerungszeit. Diese Parameter haben unterschiedliche Datentypen: in unserem Timerbeispiel sind es die Datentypen „boolean“ für das Start-Signal und „long“ für die Verzögerungszeit.

Bausteine müssen auch Daten an das aufrufende Programm zurückgeben können. In unserem Timerbeispiel ist es das verzögerte Signal.

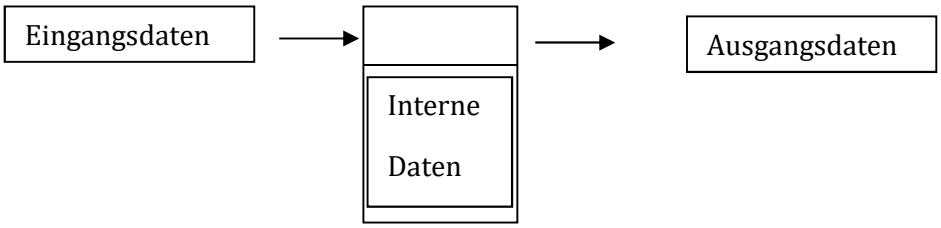


Bild1

3.2 Datenkapselung

Die internen Daten der Bausteine sollen geschützt sein vor beliebigen Zugriff aus dem Programm. Die Bausteine sollen nur über die Eingangsdaten und die Ergebnisdaten mit dem Programm kommunizieren. Die internen Daten sind nur dem Baustein selbst bekannt. Das vermeidet Fehler und unterstützt klare Programmstrukturen.

3.3 DATENERHALT

Datenerhalt ist die Eigenschaft eines Bausteins, seine Werte von Bausteinaufruf zu Bausteinaufruf zu behalten. In einem Zähler muss z.B. der Zählwert (Istwert) erhalten werden, um bei jedem Zählsignal den jeweils aktuellen Zählerstand zu erhöhen.

3.4 MEHRERE BAUSTEINE EINES TYPS PARALLEL VERWENDEN

Die Idee einer Bausteinbibliothek ist es, fertige Bausteine in das eigene Programm einzubinden. Nun kann es die Aufgabe zum Beispiel erfordern, mehrere Zählvorgänge durchzuführen. Daraus folgt, dass Bausteine gleichen Typs mehrfach nebeneinander existieren müssen, ohne sich zu beeinflussen.

4 GRUNDLAGE-WISSEN FÜR DIE BAUSTEIN-PROGRAMMIERUNG

4.1 FUNKTIONEN

Mit Funktionen (functions) lassen sich die in Kap3. beschriebenen Anforderungen an einen Baustein realisieren.

Hier eine Zusammenfassung der Wirkungsweise und Anwendung der Funktionen aus Modul A.

Als Beispiel nehmen wir eine Funktion, die zwei Werte vom Datentyp Byte vergleicht und als Ergebnis 0 für Gleichheit, 1 für Ungleichheit als Byte zurückliefert.

4.1.1 DEKLARATION DER FUNKTION

Bevor eine Funktion genutzt werden kann, muss sie im System bekannt sein.

Die Deklaration hat folgende Schreibweise:

- Datentyp des Rückgabewertes
- Funktionsname
- Parameter1 (hier als „byte“)
- Parameter2 (hier als „byte“)
- Lokale Variable (hier als „xyz“)
- Programmcode der Funktion (hier als „…..“)
- Rückgabewert (hier als „byte“)

```
byte myFunction(byte Wert1, byte Wert2) {
byte yxz
......
......
return Ergebnis
}
```

4.1.2 FUNKTIONSAUFRUF

Der Aufruf der Beispielfunktion hat die folgende Schreibweise:

```
…..
byte Wert1;
byte Wert2;
byte Gleichheit;
Gleichheit = myFunction (Wert1, Wert2);
….
…..
```

Mit diesem Konzept der Funktionsaufrufe wird die Forderung aus Kapitel 3.*1.1* *„Übergabe von Eingangsdaten und Rückgabe eines Ergebniswertes“* erfüllt

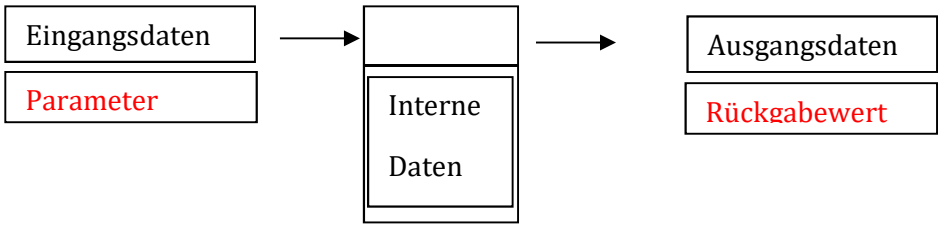


Bild 2

Übung 1

Schreiben Sie ein Programm mit einer Funktion. Die Funktion vergleicht zwei Eingangswerte miteinander und gibt bei Gleichheit 0 zurück. Ist Wert 1 größer als Wert 2, soll das Ergebnis 1 sein, ist Wert 2 größer als Wert 1, soll das Ergebnis 2 sein.

Lösung Übung1

```
#include "MyArduinoLib.h"

void setup() {       // wird Programmstart einmal durchlaufen
  setPins();         // Modus für Eingänge und Ausgänge setzen
}

void loop(){

MyWriteByte (Vergleicher(100,200));

}
//Vergleicher
//----------------------

byte Vergleicher(byte Wert1,byte Wert2) {
if(Wert1 == Wert2){return 0;}
if(Wert1 > Wert2) {return 1;}
if(Wert1 < Wert2) {return 2;}
}
```

Hinweis: Die Werte dürfen in diesem Beispiel zwischen 0 und 255 liegen!

4.2 Eigenschaften der Variablen

Für das Erstellen von Bausteinen müssen wir uns noch einmal mit den Eigenschaften der Variablen beschäftigen. Aus dem Modul A wissen wir, dass Variable mit einem Datentyp gekennzeichnet sind. Weiter von Bedeutung sind:

- Der Gültigkeitsbereich der Variablen und
- Datenerhalt

4.2.1 Gültigkeitsbereich

Eine *globale* Variable ist eine Variable, die von allen Funktionen genutzt werden kann. Sie wird *außerhalb* der Klammern {} einer Funktion deklariert.

Im Gegensatz dazu steht die *lokale* Variable. Sie ist nur der Funktion bekannt, in der sie deklariert ist. Folglich wird sie *innerhalb* der Klammern {} einer Funktion deklariert.

Mit diesem Konzept wird die Forderung aus Kapitel 3*.1.1 „Datenkapselung“* erfüllt.

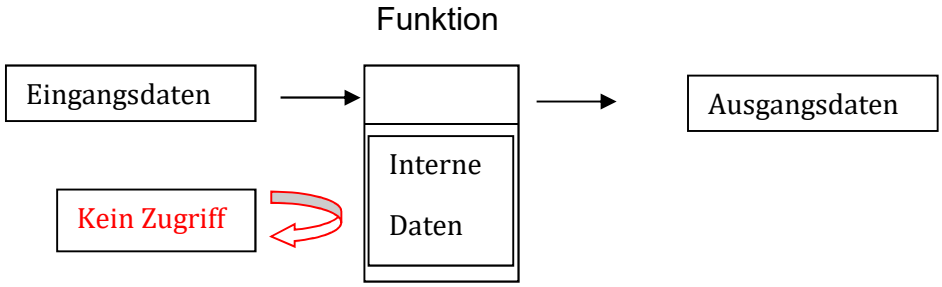


Bild 3

4.2.2 Datenerhalt

Wichtig ist auch, wie eine Funktion mit den Daten umgeht, wenn sie aufgerufen oder verlassen wird. Im Normalfall werden mit dem Funktionsaufruf die Variablen angelegt und mit der Verlassen der Funktion wieder zerstört. Für einen Zählerbaustein ist das Verhalten natürlich nicht brauchbar, er würde immer wieder bei Null beginnen. Benötigt wird ein Verhalten, bei dem der Wert über mehrere Funktionsaufrufe erhalten bleibt. Dieses wird durch *statische* Variablen erreicht. Eine *statische* Variable wird mit dem Zusatz „static“ deklariert.

```
static byte Wert1;
```

Mit diesem Konzept der Funktionsaufrufe wird die Forderung aus Kapitel 3*.1.1 „Datenerhalt“* erfüllt.

Übung 2:

Schreiben Sie einen Baustein, der die durchlaufenen Programmzyklen von 0 bis 255 zählt (bei jedem Aufruf einen Zähler um 1 erhöht). Das Zählergebnis soll an den 8 Ausgangsbytes ausgegeben werden. Um den Ablauf besser sichtbar zu machen, können die Zyklen mit der „delay“-Funktion verzögern.

Experimentiren Sie mit und ohne „static“.

Lösung Übung 2:

```
// Testprogramm für Zyklenzähler
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

#include "MyArduinoLib.h"

void setup() {          // wird Programmstart einmal durchlaufen
  setPins();            // Modus für Eingänge und Ausgänge setzen
}
// Deklaration

byte Wert;

void loop(){
Wert= ZyklenZaehler();
MyWriteByte(Wert);
delay (1000);
}
//
//Funktion ZyklenZaehler
byte ZyklenZaehler() {
static byte zykl;

zykl = zykl +1;
return zykl;
}
```

4.3 FLANKENERKENNUNG

Für viele Bausteine ist es wichtig, den Wechsel eines Zustands genau zu erkennen, um zum Beispiel eine Zeitmessung zu starten. In dem Diagramm ist es der steile Anstieg von LOW nach HIGH oder der Abfall von HIGH nach LOW. Man spricht deshalb von "ansteigenden" bzw. " abfallenden" Flanken.

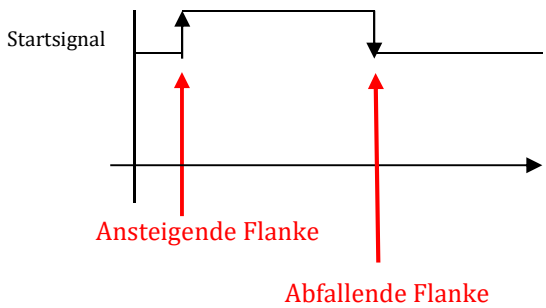


Bild 4

Die programmtechnische Erkennung von Flanken ist mit Hilfe einer Variablen einfach zu gestalten. Diese Variable merkt sich zyklisch den Zustand des zu überwachenden

Bits. Im Beispiel der ansteigenden Flanke ist es "false" (logische 0). Dann vergleicht das Programm den aktuellen Zustand, im Beispiel "true" (logische 1) mit dem zuletzt gespeicherten Wert "false" (logische 0). Weil der Zustand vorher "false" war und jetzt "true" ist, hat das Programm eine ansteigende Flanke erkannt.

Die abfallende Flanke wird sinngemäß genauso erkannt. Hier ist der letzte gemerkte Zustand "true" und der aktuelle ist "false".

Wenn dieses Verfahren zu Flankenerkennung in Bausteinen genutzt wird, muss die Variable, die den letzten Zustand speichert, eine „static“- Variable sein, denn der Wert muss beim nächsten Bausteinaufruf noch gültig (vorhanden) sein. (Siehe Kapitel *„Datenerhalt“*)

4.4 Timer-Grundlagen

Zunächst benötigen wir für alle Timer eine Methode, mit dem wir eine Zeitbasis bilden können. Dazu verwenden wir die im System enthaltene Funktion millis().

Die Funktion millis() liefert fortlaufend die verstrichenen Millisekunden. Die Zeitbildung erfolgt durch Speichern des millis-Wertes zum Start der Timerfunktion. Die Differenz zwischen gespeicherter und aktuell erfasster Zeit ist die Timerzeit. Der Fehler dieser Methode, der beim Übergang von -1 nach 0 auftritt, wird zunächst toleriert, weil die Auftrittswahrscheinlichkeit sehr gering ist.

Beispiel:

….	1012	1013	1014	1015	1016	1017	1018	1019	1020
		Start					aktuell		

Die Startzeit ist 1013. Dieser Wert wird von millis() geliefert und gespeichert. Nach einiger Zeit wird millis() erneut aufgerufen und liefert den Wert 1018. Die Differenz aus den beiden Werten ist die die seit Start verstrichene Zeit, im Beispiel 1018ms - 1013ms= 5ms.

4.5 Arrays: Eine kurze Erläuterung

In den folgenden Kapiteln werden wir Arrays oder Felder benutzen. Deshalb hier eine kurze Erläuterung.

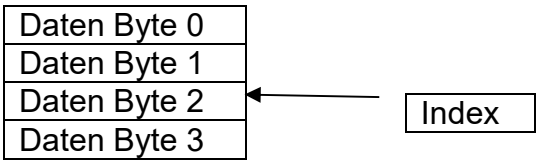
Array oder Felder sind eine Zusammenfassung von Daten gleichen Typs.

Die Deklaration eines Arrays enthält in eckigen Klammer die Anzahl der Elemente.

```
byte Arrayname [4];     // Array mit  4 Elementen vom Typ Byte
```

Der Zugriff auf Daten in einem Feld erfolgt über einen Index, die der Position der Daten im Array kennzeichnet.

Beispiel für ein Array mit dem Namen „Beispielarray“ und mit 4 Byte-Elementen




```
Beispielarray[2] = 150;              // 150 in Array-Element 2 schreiben
byte Wert = Beispielarray[2];        // Wert aus Array-Element 2 lesen
```

Hinweis:

Wir benötigen hier eindimensionale Arrays. Es sind auch mehrdimensionale Arrays programmierbar, wobei jede Dimensionen einen eigenen Index hat.

Bitte beachten:

- Die Adressierung der Elemente beginnt bei 0, gefolgt von 1,2,3, usw.. Der Index des ersten Elementes ist also 0. Der Index des letzten Elementes ist die Anzahl der deklarierten Elemente -1.
- Wird der Index überschritten, werden ungültige Daten gelesen oder andere Daten beliebig überschrieben. Deshalb ist es besonders wichtig, die Indexwerte auf gültige Bereiche zu überwachen.

4.6 BAUSTEIN-INSTANZEN

In Kapitel 3 wurde die Anforderung beschrieben, nach der mehrere Bausteine eines Typs nebeneinander funktionieren müssen, ohne sich gegenseitig zu beeinflussen.

Wenn wir die Funktion Zyklen Zähler aus der Übung 2 zweimal benutzen wollen, werden beide Zählvorgänge auf die gleiche static-Variable zykl abgebildet. Beide Zählvorgänge überschreiben sich genseitig. Die Forderung, dass die Zählvorgange parallel und unabhängig funktionieren, ist so nicht erfüllbar.

Lösungsmöglichkeiten

a) Mehrere Bausteine mit gleicher Funktion aber mit unterschiedlichen Namen in der Bibliothek. Diese Lösung ist unflexibel und führt zu großen Bibliotheken.
b) Klassenbibliotheken: In der der Objektorientierten Programmierung wird das Problem mit Hilfe von Klassen und Instanzen, die während der Laufzeit erzeugt werden, elegant gelöst. Der Einstieg in objektorientierte Programmierung mit Klassenbibliotheken würde den Rahmen dieses Moduls sprengen.
c) Programmtechnische Lösung mit separaten Daten je Bausteinanwendung

Wir wollen Lösung c) weiterverfolgen und das Problem selbst lösen, um das Verständnis zu vertiefen und die Programmierkenntnisse bzgl. Arrays zu erweitern.

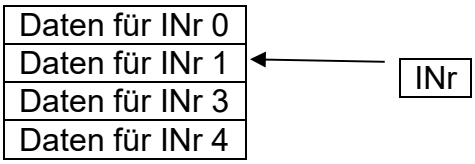
Wir benötigen zunächst eine eindeutige Unterscheidung zwischen den Datensätzen, die wir Bausteininstanzen nennen. Dazu führen wir eine Instanz-Nummer ein, die als erster Paramater mit an die Funktion übergeben wird.

```
Wert= Funktionsname (INr, param1, param2);
```

Weiter sind einige Programmcode-Zeilen notwendig, die dafür sorgen, dass der Baustein mit den zur Instanz gehörenden Daten arbeitet. Das Ziel ist also, die Variablen mehrfach anzulegen und den Zugriff darauf mit der Instanz-Nummer zu steuern.

Statt einer Variablen legen wir ein „Array“ an, das mehrere Variable beinhaltet. Wie in Kapitel „Datenerhalt“ beschrieben, müssen es „static“-Variable sein. Die Instanz-Nummer ist der Index auf die Array-Elemente.

Static byte Array



Deklaration des Arrays

```
static byte Arrayname [4];      // Array für 4 Instanzen eines Bausteins
```