1. TWI-Master für ATMEGA in C (ohne TWI-Start/Stop, ACK/NACK)

Ok, der Betreff/Titel soll provozieren.
Natürlich ist ein Betrieb des TWI-Busses ohne TWI-Start/Stop-Condition nicht möglich.
Und auch das ACK/NACK muss ausgewertet werden.

Die Frage ist aber, ob der Anwender diesen Verwaltungskram wirklich selbst regeln muss.

Der Einsatz des TWI-Busses könnte benutzerfreundlicher sein als es in manchen Bibliotheken der Fall ist.
Selbst die Arduino-Fraktion quält die geneigte Anwenderschaft mit

```
- wire.beginn();
- wire.beginTransmission(0x40);
- wire.send('A');
- wire.send('B');
- wire.endTransmission();
```

Dabei kann man den Beispielcode aus der AVR315 von ATMEL nehmen, einige kleine Ergänzungen vornehmen - und die Arbeit läuft fast von selbst.

Als Beispiel:
Um ein "0x55" an einen PCF8574 mit der TWI-Adresse "0x40" zu senden:

```
twi_buffer[0] = 0x40;
twi_buffer[1] = 0x55;

TWI_MA_Write(twi_buffer, 2);
```

Um den Status des PCF8574 anschließend zur Kontrolle wieder auszulesen:

```
TWI_MA_Read(twi_buffer, 2);
```

In twi_buffer[1] steht dann der Status der PortPins des PCF8574.

Ein anderes Beispiel:
Um 16 Byte ab der 16-Bit Adresse "eeprom_adr" aus einem Eeprom mit der TWI-Adresse 162 auslesen:

```
twi_buffer[0] = 162;
twi_buffer[1] = (uint8_t) (eeprom_adr >> 8);
twi_buffer[2] = (uint8_t) (eeprom_adr);

TWI_MA_Write(twi_buffer, 3);
TWI_MA_Read(twi_buffer, 17);
```

Die Daten stehen nun in twi_buffer[1] .. [17].

Wie man sieht:

Das umständliche Hantieren mit TWI-Start, TWI-Stop, das Abfragen bzw. Setzen von ACK's und NACK's, das Setzen von Flags und von Read/Write Bits ist nicht wirklich notwendig - zumindest nicht durch den Anwender.

Er muss lediglich beachten:

- Einmalig zu Beginn des Programmablaufes muss das TWI-Modul initialisiert werden.
- Die zu sendenden/empfangenen Daten werden in einem Array übergeben.
- Das erste Byte des Arrays (Index 0) muss die Adresse des Slave beinhalten!
- Die zu sendenden/empfangenen Daten stehen im Array immer beginnend an Index 1!
- Bei der Anzahl der zu sendenden/empfangenden Daten muss die Adresse mitgezählt werden!

Dass die TWI-Adresse an Position 0 innerhalb des Arrays steht, das sieht im ersten Moment befremdlich aus.
Hat aber den enormen Vorteil, dass die TWI-Adresse den Aufruf eines Read/Write überlebt und mehrfach nacheinander verwendet werden kann:

Häufig schreibt man eine Adresse auf einen Slave und liest anschließend Daten aus - die TWI-Adresse in Byte[0] muss dabei nicht neu ins Array geschrieben werden.

Achtung, die 7-Bit TWI-Adresse wird links ausgerichtet erwartet!
Das Read/Write-Bit (das Bit.0) kann völlig ignoriert werden, das TWI-Modul wählt die erforderlichen Einstellungen ohne Zutun von aussen!

Ein weiterer Vorteil dieser "Bibliothek":
Das TWI-Modul arbeitet interruptgesteuert.
Sobald die zu sendenden Daten an das Modul übergeben sind, kann das Hauptprogramm weiterarbeiten, das Versenden der Daten erfolgt im Hintergrund.
Zu beachten ist, dass dazu die Interrupts freigegeben sein müssen!

Die Funktion TWI_MA_Read() gibt als Parameter den Erfolg (TRUE) oder Misserfolg (FALSE) der Aktion zurück. Diese Information kann ausgewertet werden - muss aber nicht.

Die Funktion TWI_MA_Write() dagegen kopiert nur die zu sendenden Daten in den Speicherbereich des Moduls, startet die TWI-Übertragung und kümmert sich nicht um den weiteren Ablauf der Transaktion.

Ein verschmerzbarer Nachteil dieser TWI-Master-Lösung:
Innerhalb des TWI-Moduls muss zur Pufferung der Daten ein zusätzliches, lokales Array zur Verfügung stehen, das ebenso groß ist wie die maximal zu versendende Nachricht + 1 (für die TWI-Adresse).

2. TWI-Slave für ATMEGA in C

Hier folgt die Beschreibung zum Pendant des TWI-Masters, dem TWI-Slave.

Dieses Programm ist auf der Grundlage der AppNote AVR311 entstanden.

Das TWI-Slave-Modul arbeitet vollständig interruptgesteuert im Hintergrund - unabhängig und unsichtbar für die Hauptanwendung.

Es verfügt über 2 unabhängige Puffer - jeweils für Daten, die der Master sich abholen kann und für Daten, die der Master an den Slave sendet.

Wenn der Master Daten an den Slave sendet, dann beendet er die Transaktion mit einer Stop-Condition.
Wird diese vom Slave erkannt, dann schaltet er den Empfang ab (das Flag TWIE wird gelöscht).

Will main() wissen, ob Daten empfangen wurden, dann muss es eine Abfrage losschicken:

```
if (i = TWI_SLA_Rx_Cnt())
```

Die Funktion liefert NULL zurück, wenn keine Daten empfangen wurden, ansonsten die Anzahl der Bytes einschließlich der gesendeten TWI-Adresse.
Main() muss nun die Daten aus dem Empfangspuffer abholen, erst dann ist das TWI-Modul zum Empfang neuer Daten bereit.

Der schematische Programmablauf sieht so aus:

```
TWI_SLA_Init();
asm volatile("sei");

while(1)
{
  if (i = TWI_SLA_Rx_Cnt())
  {
    TWI_SLA_Get_Data(twi_buffer);
    /*
    Die empfangenen Daten stehen nun in twi_buffer
    es sind genau (i-1) Nutzbytes
    twi_buffer[0] ist die verwendete TWI-Slave-Adresse
    */
  }
  // Daten in einen Puffer schreiben, den der Master auslesen kann
  TWI_SLA_put_data(twi_buffer, 3);

  /*
  tu sonst noch was
  */
}
```

Wie beim TWI-Master liefert das erste Byte des Empfangsarrays die TWI-Adresse.
In diesem Falle diejenige TWI-Adresse, unter der der Slave angesprochen wurde.
Anmerkung dazu:
Die ATMegas können so konfiguriert werden, dass sie auf mehreren Adressen lauschen.
Im Register TWAR definiert das Bit.0, ob ein General Call erkannt wird.
Im Register TWAMR lässt sich eine Maske für die Adresse in TWAR anlegen.

Wenn der Master vom Slave Daten abholen will, müssen diese vorher auf dem Slave bereitgestellt werden.

Dazu werden die Daten in ein Array verpackt und auf das TWI-Modul kopiert:

```
TWI_SLA_Put_Data(twi_buffer, anzahl_der_bytes);
```

Alternativ kann der Pointer auf den Sendepuffer des TWI-Moduls auf eine beliebige Speicherstelle verbogen werden (sinnigerweise das erste Element eines Arrays).

```
TWI_SLA_Set_TxPtr(&datenarray[0]);
```

Holt der Master nun Daten ab, dann werden die aus dem Array datenarray[] geliefert.

Diese Methode erspart das Umkopieren und reduziert den Speicherbedarf (weil der Sende-Puffer im TWI_Slave-Modul entfallen kann).
Ausserdem kann man auf diesem Wege (zum Debuggen) Speicherinhalte des Slave auslesen - gelegentlich ist das ganz hilfreich.

PS.

Der TWI-Slave hört nun - auf Wunsch - auch auf einen General Call (Adresse 0).
Dazu muss das Bit.0 im Register TWAR (TWI-Slave-Adresse) gesetzt sein.

3. USI-Slave für ATTINY in C

Nur die "größeren" Controller verfügen über eine Hardware-TWI-Schnittstelle.

Aber auch auf den ATtiny's lässt sich eine TWI-Schnittstelle einrichten.
Anstelle der (nicht vorhandenen) TWI-Hardware wird hier das USI-Modul eingesetzt.

Die nachfolgenden Routinen für den USI-Slave funktionieren aus Sicht des Anwenders nach der gleichen Methode wie beim TWI-Slave.

Das USI-Slave-Modul arbeitet wieder vollständig interruptgesteuert im Hintergrund - unabhängig und unsichtbar für die Hauptanwendung.

Es verfügt über 2 unabhängige Puffer - jeweils für Daten, die der Master sich abholen kann und für Daten, die der Master an den Slave sendet.

Wenn der Master Daten an den Slave sendet, dann beendet er die Transaktion mit einer Stop-Condition.

Dieses Stop wird vom USI-Slave als Zeichen für den Abschluss einer Übertragung gewertet.
(Im Gegensatz zum TWI-Slave bleibt der USI-Slave z.Z. weiterhin empfangsbereit.
Folgt eine neue Übertragung, bevor die Daten abgeholt sind, dann wird der Empfangs-Puffer überschrieben.)

Will main() wissen, ob Daten eingegangen sind, dann muss es eine Abfrage losschicken.

```
if (i = USI_SLA_Rx_Cnt())
```

Die Funktion liefert NULL zurück, wenn keine Daten empfangen wurden, ansonsten die Anzahl der Bytes einschließlich der gesendeten TWI-Adresse.
Sofern Daten eingegangen sind, muss main() sie aus dem Empfangspuffer abholen.

Der schematische Programmablauf sieht so aus:

```
USI_SLA_Init();
asm volatile("sei");

while(1)
{
  if ((i=USI_SLA_Rx_Cnt()))
  {
    USI_SLA_Get_Data(twi_buffer);
    /*
    Die empfangenen Daten stehen nun in twi_buffer,
    es sind genau (i-1) Nutzbytes
    twi_buffer[0] ist die verwendete USI-Slave-Adresse
    */
  }
  /*
  tu sonst noch was
  */
}
```

Wie bei den TWI-Routinen liefert das erste Byte des Empfangsarrays die TWI-Adresse.
(Die Interrupt-Routine kann mit geringem Aufwand so geändert werden, dass der USI-Slave auf mehrere Adressen reagiert - der General Call ist bereits möglich.)

Wenn der Master vom Slave Daten abholen will - oder soll, müssen diese vorher auf dem Slave bereitgestellt werden.

Dazu werden die Daten in ein Array verpackt (ohne dass eine TWI-Adresse vorangestellt wird !) und auf das TWI-Modul kopiert :

```
USI_SLA_Put_Data(twi_buffer, anzahl_der_bytes);
```

Alterativ kann man den Pointer auf den Sendepuffer auf jeden beliebigen Speicherbereich des Controllers verbiegen:

```
USI_SLA_SetPtr(&datenarray[0]);
```

Wenn der Master nun Daten aus dem Slave ausliest, dann werden sie aus dem Array datenarray[] geliefert.

Die Funktionen sind aus dem Beispielcode zur AppNote AVR 312 entwickelt.

Jetzt fehlt nur noch der USI_Master.

PS.

Der USI-Slave hört nun - auf Wunsch - auch auf einen General Call (Adresse 0).
Dazu muss gewählt werden: #define GENERAL_CALL TRUE.

4. USI-Master für ATTINY in C (ohne TWI-Start/Stop, ACK/NACK)

Wenngleich eher selten benötigt, hier zum Abschluss der Serie das letzte Teil im Puzzle:
der USI-Master.

Das Handling ist aus Sicht des Anwenders identisch mit dem des TWI-Masters.

Als Beispiel:

Um ein "0x55" an einen PCF8574 mit der TWI-Adresse "0x40" zu senden:

```
twi_buffer[0] = 0x40;
twi_buffer[1] = 0x55;

USI_MA_Write(twi_buffer, 2);
```

Um den Status des PCF8574 anschließend zur Kontrolle wieder auszulesen:

```
USI_MA_Read(twi_buffer, 2);
```

In twi_buffer[1] steht dann der Status der PortPins des PCF8574.

Ein anderes Beispiel:
Um 16 Byte ab der 16-Bit Adresse "eeprom_adr" aus einem Eeprom mit der TWI-Adresse 162 auslesen:

```
twi_buffer[0] = 162;
twi_buffer[1] = (uint8_t) (eeprom_adr >> 8);
twi_buffer[2] = (uint8_t) (eeprom_adr);

USI_MA_Write(twi_buffer, 3);
USI_MA_Read(twi_buffer, 17);
```

Die Daten stehen nun in twi_buffer[1] .. [17].

Zu beachten ist wieder:

- Einmalige zu Beginn des Programmablaufes muss das USI-Modul initialisiert werden.
- Die zu sendenden/empfangenen Daten werden in einem Array übergeben.
- Das erste Byte des Arrays (Index 0) muss die Adresse des Slave beinhalten!
- Die zu sendenden/empfangenen Daten stehen im Array immer beginnend an Index 1!
- Bei der Anzahl der zu sendenden/empfangenden Daten muss die Adresse mitgezählt werden!

Dass die TWI-Adresse an Position 0 innerhalb des Arrays steht, das sieht im ersten Moment befremdlich aus.
Hat aber den enormen Vorteil, dass die TWI-Adresse den Aufruf eines Read/Write überlebt und mehrfach nacheinander verwendet werden kann:
Häufig schreibt man eine Adresse auf einen Slave und liest anschließend Daten aus - die TWI-Adresse in Byte[0] muss dabei nicht neu ins Array geschrieben werden.

Achtung, die 7-Bit TWI-Adresse wird links ausgerichtet erwartet!
Das Read/Write-Bit (das Bit.0) kann völlig ignoriert werden, das TWI-Modul wählt die erforderlichen Einstellungen ohne Zutun von selbst!

Programmintern gibt es wesentliche Unterschiede zwischen der Arbeitsweise des TWI-Masters und der des USI-Masters:

Der USI-Master arbeitet nicht interruptgesteuert und die zu sendenden Daten werden nicht ins USI-Modul kopiert.

Das Programm schiebt eigenhändig Bit für Bit auf die SDA-Leitung, erzeugt das Clocksignal auf SCL, setzt START- und STOP Conditions und prüft die ACKs und NACKs der Gegenstelle.

Aber das alles wollen wir im Detail eigentlich alles garnicht wissen.

Der Programmcode ist mit nur geringfügigen Ergänzungen aus dem Beispielcode zur AppNote AVR312 von ATMEL übernommen.

Mit dem USI-Master ist das Quartett endlich komplett:

Es gibt den Master und den Slave jeweils auf Basis von Hardware-TWI für die ATMEGAs bzw. Software-USI für die ATTINYs - mit einheitlicher Bedienung und ohne die Notwendigkeit, eigenhändig Start- Stop- Bedingungen setzen oder ACKs und NACKs prüfen oder setzen zu müssen.

Die Bedienung beschränkt sich (nach der Initialisierung) auf folgende Funktionen:

Bei einem Master:

```
- MA_Write(Array_mit_Daten, Anzahl_Bytes)      Daten auf den Slave schreiben
- MA_Read(Array_mit_Daten, Anzahl_Bytes)       Daten vom Slave lesen
```

Bei einem Slave:

```
- SLA_Rx_Cnt()                                 Prüfen, ob Daten empfangen wurden
- SLA_Get_Data(Array_mit_Daten)                Wenn ja, dann die Daten abholen
- SLA_Put_Data(Array_mit_Daten, Anzahl_Bytes)  Daten zur Abholung durch Master
                                               bereitstellen
```

TWI kann ja so einfach sein.

Fehlt jetzt noch etwas ?

Ja, der TWI-Multi-Master.

Mal abwarten, vielleicht gelingt er. Der Anfang ist zumindest gemacht.

mfg

Michael S.