5. TWI-Multi-Master für ATMEGA in C

5.1 Zusammenfassung

Am TWI-Bus gibt es im Normalfall (im "Single-Master-Betrieb") genau einen einzigen Master, der die Kontrolle über den Bus ausübt, das Taktsignal generiert und jegliche Kommunikation initiiert. Die Slaves sind passiv und können von sich aus keine Verbindung aufbauen.

Im "Multi-Master-Modus" können mehrere Master an einem gemeinsamen Bus koexistieren und direkt untereinander Daten austauschen.
Mit der Einschränkung, dass zu einem gegebenen Zeitpunkt maximal ein einziger Master aktiv sein darf, alle anderen "Master" müssen sich als Slave im Hintergrund halten.
Diese Einschränkung einzuhalten verkompliziert den Multi-Master-Betrieb deutlich gegenüber dem Single-Master-Betrieb.

Softwaretechnisch ist der TWI-Multi-Master ein TWI-Slave, der zusätzlich die Funktionalität eines TWI-Masters bietet.
Die Bedienung des Multi-Masters ist weitgehend identisch mit der des "normalen" TWI-Masters bzw. TWI-Slaves.

Der Multi-Master kennt im Slave-Modus die Funktionen:

```
TWI_SLA_Get_Rx_Cnt();                          // wie viele Bytes empfangen ?
TWI_SLA_Get_Data(buffer);                      // empfangene Daten abholen
TWI_SLA_Put_Data(buffer, bytes);               // Daten in den Ausgangspuffer
```

Die empfangenen Daten werden im Array "buffer[]" abgeholt. Das erste Byte des Arrays ist die Adresse, unter der der Slave angesprochen wurde:
Die neueren ATMegas können so konfiguriert werden, dass sie auf mehrerer Adressen antworten.
Und alle Controller können (optional) einen General Call (Adresse 0) entgegen nehmen.

Im Master-Modus gibt es diese Funktionen:

```
TWI_MM_Write(buffer, tx_bytes);                // Daten schreiben
TWI_MM_Read(buffer, rx_bytes);                 // Daten lesen
TWI_MM_Write_Read(buffer, tx_bytes, rx_bytes); // erst schreiben, dann lesen
```

Jede der drei Master-Funktionen liefert bei Erfolg ein TRUE, ansonsten ein FALSE zurück.

Die Daten werden immer in einem Array übergeben, in dem als erstes Byte die Adresse des Slaves erwartet wird.
Das Read/Write-Bit kann ignoriert werden, es wird vom Programm automatisch gesetzt.
Die 7-Bit TWI-Adresse muss linksbündig ausgerichtet sein.
Der zweite Parameter gibt an, wieviele Bytes (einschl. der TWI-Adresse) zu schreiben oder zu lesen sind.

Für die Verkettung von beliebigen Master-Transaktionen ohne Busfreigabe kennt der Master zusätzlich noch:

```
TWI_MM_Set_Stop_Mode([TWI_REPEATED_START | TWI_STOP]);
```

5.2 Schematischer Programmablauf

Schauen wir uns zunächst an, wie die Software aus Sicht des Anwenders arbeitet.
Im Slave-Modus arbeitet der Multi-Master vergleichbar mit dem TWI_Slave:

```
while(1)
{
  if (i = TWI_SLA_Rx_Cnt())              // Prüfen, ob Daten eingegangen
  {
    TWI_SLA_Get_Data(buffer);            // wenn ja, dann abholen

    /* Daten in buffer[] bearbeiten */

    TWI_SLA_Put_Data(buffer, i);         // Daten in den Sendepuffer stellen
  }
}
```

Der Master wird nur aktiv, wenn für ihn Arbeit ansteht:

```
while(1)
{
  // Wenn der Master Daten senden oder abholen soll, dann schreiben
  // Unterprogramme die notwendigen Daten in das Array buffer[],
  // schreiben in die Variable cnt wie viele Bytes zu lesen oder zu holen sind
  // und setzen ein Flag in der Variablen job.

  if (job)                               // steht ein Job für den Master an ?
  {
    if(job == TX)                        // ein Job zu Schreiben von Daten
    {
      e = (TWI_MM_Write(buffer, cnt);
      if(!(e)) Fehlerbehandlung();       // geeignet auf Fehler reagieren
    }
    else if(job == RX)                   // ein Job zum Lesen von Daten
    {
      e = TWI_MM_Read(buffer, cnt);
      if (!(e)) Fehlerbehandlung();      // geeignet auf Fehler reagieren
    }
    else if(job == TRX)                  // ein Job zum Lesen von Daten
    {
      e = TWI_MM_Write_Read(buffer, cnt);
      if (!(e)) Fehlerbehandlung();      // geeignet auf Fehler reagieren
    }
  }
  /* etwas nützliches ausführen */       // der Rest des Programmes
}
```

Das Programm läuft in einer Endlosschleife und prüft regelmäßig, ob als Slave Daten empfangen wurden. Wenn ja, dann werden sie abgeholt und verarbeitet.

Wenn Programmteile eine Aktion des Master anfordern, dann stellen sie die Daten im Array buffer[] und die Anzahl der zu übertragenden Bytes in der Variablen cnt bereit und setzen ein Flag.

Jeder der Multi-Master-Transfers liefert den Erfolg als Parameter zurück.
Auf Fehler muss das Programm nun in geeigneter Weise reagieren.
Etwa einen neuen Transaktionsversuche starten.

5.3 Problematik des Multi-Master-Betriebes

Im Normalfall arbeitet der Multi-Master passiv als TWI-Slave und beantwortet die Anfragen externer Master bzw. nimmt Daten von ihnen entgegen.

Nur auf besondere Anforderung hin wird er zum aktiven TWI-Master und kann Daten zu anderen Busteilnehmer senden oder von dort abholen.

In diesem Falle arbeitet alle anderen Busteilnehmer als Slaves.
Es gilt die strikte Regel, dass zu einem gegebenen Zeitpunkt immer nur ein einziger Master den Bus regieren darf.

Im Multi-Master-Betrieb muss sichergestellt sein, dass nie mehrerer Master parallel im Master-Modus arbeiten.
Sollte es - und das ist kaum vermeidbar - zu Kollisionen kommen, dann müssen die Beteiligten sich so organisieren, dass am Ende alle Daten sicher zugestellt werden.

Was ist im Multi-Master-Betrieb zu beachten?

1.) Ein Master darf nur dann eine Übertragung starten, wenn der TWI-Bus frei ist.
Diese Aufgabe übernimmt das grundsätzlich das Hardware-TWI-Modul:
Es reicht einen TWI-START erst dann auf den Bus weiter, wenn keine laufende Transaktion mehr erkannt wird.

Während meiner Tests hat sich gezeigt, dass nach dem Setzen des Flags TWEA das auf einen Stop wartende TWI-Modul nicht mehr als Slave aktiv ist und keine Interrupts (als Slave) bearbeiten kann.
Daraus resultiert das Problem, dass der wartende Master - als Slave adressiert - mit einem NACK antwortet.
Um das zu vermeiden, wird vor dem Start einer Master-Transaktion in einer Warteschleife durch Pollen der SCL-Leitung das Ende der laufenden Übertragung abgewartet.
Innerhalb dieser Warteschleife ist der Slave-Modus noch aktiv und auf Anfragen externer Master kann interruptgesteuert reagiert werden.

2.) Häufig werden nacheinander mehrere Aufrufe von TWI-Transaktionen erforderlich, um eine komplexe Aufgabe zu lösen. Etwa beim Auslesen der Daten eines Eeproms:
Hier muss zunächst die auszulesende Speicheradresse gesendet werden, erst dann können die Daten eingelesen werden.

Dabei darf das Schreiben der Adresse nicht mit einem TWI-STOP beendet werden.
Denn ein TWI-STOP würde jedem anderen, wartenden Master die Chance einräumen, unvorhersehbar die Kontrolle über den Bus zu übernehmen.
Statt des TWI-STOP muss ein REPEATED START gesendet werden (ein Repeated Start gibt den Bus nicht frei).

Die Funktion TWI_MM_WriteRead(buffer, tx_cnt, rx_cnt) verbindet das Write (der Adresse) mit dem Read (der Daten) über einen Repeated Start.

Mit Hilfe der Funktion TWI_MM_Set_Stop_Mode() können beliebige Transaktionen über einen Repeated Start ohne Busfreigabe zusammengehalten werden.
Nach dem Aufruf von TWI_MM_Set_Stop_Mode(TWI_REPEATED_START) werden alle nachfolgenden Transaktionen mit einem Repeated Start beendet.
Erst wenn nach dem Aufruf von TWI_MM_Set_Stop_Mode(TWI_STOP) wird der Bus nach der darauf folgenden Transaktion freigegeben.

Bleibt die Umschaltung auf "TWI_STOP" aus, dann wird SDA/SCL nach Ende der Funktion weiter auf low gehalten und der Bus ist dauerhaft blockiert.

3.) Je höher die Auslastung auf dem Bus ist, desto größer wird die Gefahr, dass mehrere Master einen Transfer beginnen wollen und dabei auf den Abschluss einer laufenden Übertragung warten müssen.
Erkennen die Master das TWI-STOP (das eine Transaktion abschließt), dann glauben sie alle, der Bus sei frei - und senden los.
In dieser Situation entstehen unvermeidbare Kollisionen.

Auch dieser Fall wird vom Hardware-TWI-Modul erkannt, es setzt ein Interrupt-Flag und schreibt einen Statuscode in das Register TWSR.

Wie weiter auf den Fehler reagiert wird, das muss die Software des Multi-Masters regeln.

Die denkbaren Kollisionen sind im Manual zu den ATMEGAs beschrieben:

3.a Zwei oder mehrere Controller starten die gleiche Übertragung an den gleichen Slave.
Weder die Master noch der Slave stellen einen Fehler fest.

3.b Zwei oder mehrere Controller starten mit dem gleichen Slave eine Übertragung mit unterschiedlichem R/W-Bit.
Der Master, der lesen will (R/W-Bit gesetzt) verliert die Arbitrierung.
Er muss den Abschluss der Übertragung abwarten und einen neuen Leseversuch starten.

3c Zwei oder mehrere Controller starten eine Übertragung mit unterschiedlichen Slaves.
Hier ereignet sich ein Fehler während die Adresse gesendet wird.
Der Master, der die Arbitrierung verliert, schaltet in den Slave-Modus und prüft, ob er möglicherweise vom gewinnenden Master als Slave adressiert wurde.
Wenn ja, dann prüft er, ob der Master lesen oder schreiben will und verfährt in geeigneter Weise weiter.
Wurde er nicht als Slave adressiert, dann wartet er bis der Bus frei ist und startet einen neuen Versuch.

5.4 Fehlerbehandlung durch den TWI-Master

Nachfolgend wird beschrieben, wie die Software des TWI-Masters bei Fehlern vorgeht.

Zu Beginn einer jeden Master-Transaktion wird die Variable 'TWI_success' mit FALSE und 'TWI_Arbitration_State' mit '0' vorbelegt.
Nur wenn innerhalb des TWI-Interrupts ein Status eintritt, der auf einen Erfolg hinweist, wird TWI_success auf TRUE geändert. Das sind im einzelnen folgende Situationen:

- Im Master-Transmittermodus, wenn alle Bytes des Sendepuffers versandt sind und das letzte Byte vom adressierten Slave mit einem ACK bestätigt wurde.
- Im Master-Receivermodus, wenn alle Bytes abgeholt worden sind und der Master ein abschließendes TWI-STOP oder REPEATED START gesendet hat.

Ist das Ende einer Transaktion erreicht und TWI_success ist FALSE, dann werden - mit einer kurzen Wartezeit - neue Übertragungsversuche gestartet.

Fehler können entstehen, wenn der Slave auf das Senden seiner Adresse oder von Daten keine Reaktion zeigt (das wird als NACK interpretiert).
Oder wenn beim Versuch eines Verbindungsaufbaus eine Kollision mit anderen Mastern auftrat, die

zeitgleich senden wollten (Arbitrierungsfehler).
Bei einem Arbitrierungsfehler wird der Status in TWSR in die Variable TWI_Arbitration_State geschrieben und kann von Fehlerbehandlungsroutinen ausgewertet werden.

Bei einem Arbitrierungsfehler werden aufgrund der Situation, in der der Fehler erkannt wurde, vier States unterschieden.
Drei der vier Arbitrierungsfehler betreffen Situationen, in denen der Master verloren hat, aber von der Gegenstelle als Slave adressiert wurde.

Diese Situation lässt sich ausnutzen:

Wenn der eigene Slave adressiert wurde, dann werden die notwendigen Vorkehrungen getroffen, um die Transaktion erfolgreich fortsetzen zu können.
Bei einem Slave-Transmit wird das erste Byte aus dem Sendepuffer nach TWDR geschrieben und der Index auf 1 (das nächste zu sendende Byte) gesetzt.
Bei einem Slave-Receive wird das empfangene Byte im Empfangspuffer abgelegt und der zugehörige Index auf das nächste Element gesetzt.

Während meiner Tests trat bei Arbitrierungsfehlern grundsätzlich nur der Code '0x38' auf.
Die anderen Fehler wurden nie erkannt und der Programmablauf konnte nicht getestet werden.
Der zugehörige Programmcode ist daher auskommentiert, alle Arbitrierungsfehler werden wie allgemeine Fehler behandelt.

Auf das Ende jeder Master-Transaktion - ob mit oder Erfolg - wird in der Schleife while(TWI_BUSY) gewartet, in der das Bit TWIE im Register TWCR geprüft wird.

Ist eine Übertragung erfolgreich abgeschlossen oder wird ein Fehler erkannt, dann wird das Flag TWIE in TWCR gelöscht und TWI_BUSY liefert FALSE zurück.
Damit ist das Zeichen für den Abschluss der Übertragung gegeben.

Bei einem Fehler wird die Transaktion sofort beendet und nach einer Wartezeit wird ein neuer Versuch gestartet. Nach mehreren erfolglosen Versuchen wird abgebrochen, die Funktion liefert ein FALSE zurück.

Durch die Abfrage des TWI_Arbitration_State kann festgestellt werden, ob ein Arbitrierungsfehler oder ein anderer Fehler die Ursache war.

Die Wartezeiten, die nach Fehlerbedingungen eingehalten werden, wurden so definiert, dass sie auf jedem der Master von unterschiedlicher Dauer sind.
Dazu wird die TWI-Adresse des Slaves, die innerhalb des Busses einmalig ist, als Teil der zeitbestimmenden Konstanten gewählt:
Haben mehrere Master mit Fehlern abgebrochen, dann starten sie zu unterschiedlichen Zeitpunkten einen neuen Transferversuch.
Dauert allerdings die laufende Transaktion länger als die eingestellte Wartezeit gewählt ist, dann geht diese Strategie nicht mehr auf: alle Master sind bereits vorzeitig aktiv geworden und lauern wieder gemeinsam auf ein TWI-STOP.

Im erneuten Fehlerfalle müssen weitere Wiederholungen versucht werden.

Andererseits darf die Anzahl der Wiederholungen auch nicht zu groß gewählt werden, um durch fehlende oder ausgefallene Slaves den Bus nicht unnötig zu belasten.

5.5 Programmtest

Ob die Strategie der Fehlerbehandlung auch in der Praxis erfolgreich ist, das ist natürlich die große Unwägbarkeit.

Um die kritischen Situationen bei Kollisionen zu testen, habe ich einige Testszenarien mit drei identischen Mastern, einem PCF8574 und einem seriellen Eeprom AT24C16 aufgebaut und den Netzwerkverkehr am Logic-Analyser beobachtet.

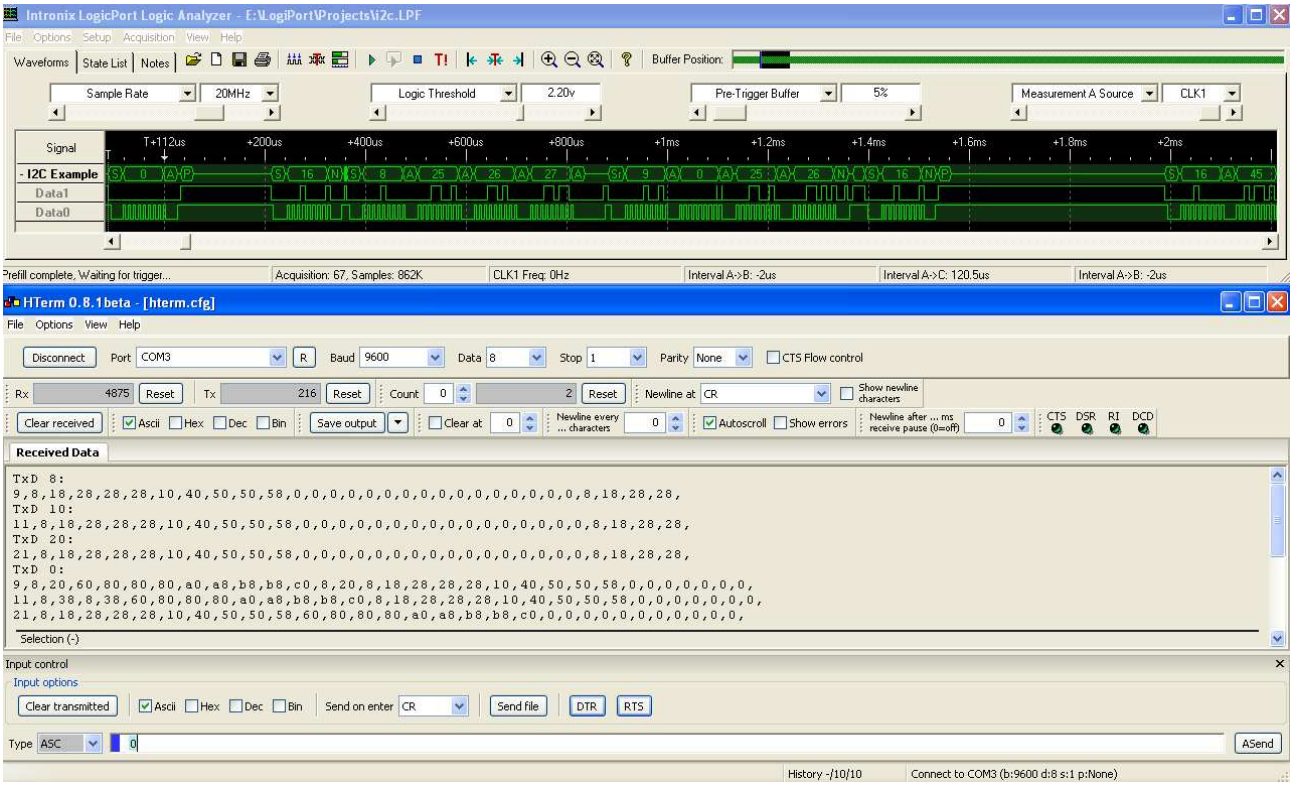
Jeder Master hat eine eigene TWI-Adresse, hört aber auch auf einen General Call.

Wird die Adresse eines Masters auf den Bus gegeben, dann startet genau dieser Master eine Nachricht.
Wird ein General Call auf den Bus gegeben, dann starten alle Master gleichzeitig ihre Nachricht.

Der erste Ansatz dient dazu, die versandte Nachricht der Master grundsätzlich zu prüfen.
Der zweite Ansatz zeigt dann, ob sich nach der Kollision (Arbitirierungs-) Fehler auflösen.

Die Tests werden vom PC aus gesteuert.
Mit Hilfe eines Interface werden von einem Terminalprogrammes Daten auf den TWI-Bus geschickt.
Die Bus-Aktivität wird mit einem Logic-Analyser aufgezeichnet.

Jeder Master protokolliert während jedes Interruptaufrufs den Status des Registers TWSR und sendet nach Ende der Übertragung (mit einigen Millisekunden Verzögerung) das gesamte Array mit den Protokolldaten zurück an den PC.



Hinter "TxD" folgt der Wert, der zum TWI-Bus gesendet wurde (alles in Hex-Darstellung).
Die darauf folgende Zeilen zeigen die TWSR-States, die während der Übertragung aufgezeichnet wurden. Das erste Byte ist die Adresse des (Slaves + 1), dessen Daten gesendet werden.

Nachfolgend sind die Tests dokumentiert.

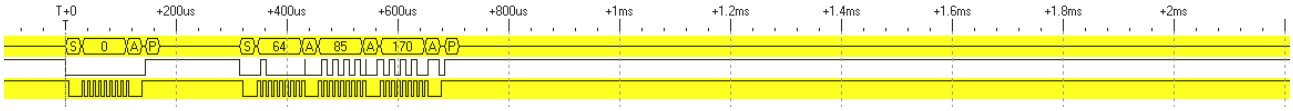
Dargestellt wird jeweils das mit einem Logic-Analyser aufgezeichnete Signal am TWI-Bus (als Screendump mit veränderter Farbdarstellung) sowie das zugehörige Protokoll der TWSR-States, jeweils separat für jeden der drei Master.
In der ersten Zeile jedes Protokolls ist die Nachricht zu sehen, die im Single-Master-Betrieb aufgezeichnet wurde (hier wurde jeweils ein einzelner Master mit seiner TWI-Adresse aufgerufen).

Die Master benutzen die Adressen #8, #16 und #32. Sie kommunizieren in der Regel jeweils mit dem Nachbarn, also #8 mit #16, #16 mit #32 und #32 mit #8.
Zur besseren Übersicht sind die Status-Codes des Protokolls farbig hinterlegt:

				8	10	Start, Repeated Start
		18	20	28	30	Master-Transmitter
		40	48	50	58	Master-Receiver
		a8	b8	c0	c8	Slave-Transmitter
60	70	80	88	90	98	Slave-Receiver
38	b0	68	78		a0	Arbitration Error, Stop

Test 1

Drei Master senden zeitgleich dieselbe Nachricht ("0x55+0xAA") an einen PCF8574



Die freistehende "0" am Trigger ist der General Call, der die Master zeitgleich startet.
Da die Controllertyp, der Quarztakt und der Programmcode auf allen Mastern gleich ist, sollten alle Controller auch weitgehend zeitgleich mit dem Start ihrer Transaktionen beginnen.
Und tatsächlich, alle drei Übertragungen fallen zusammen.
Zumindest bei vielen Tests. Gelegentlich werden 2, manchmal auch 3 Übertragungen verzeichnet.

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q
	8	18	28	28													

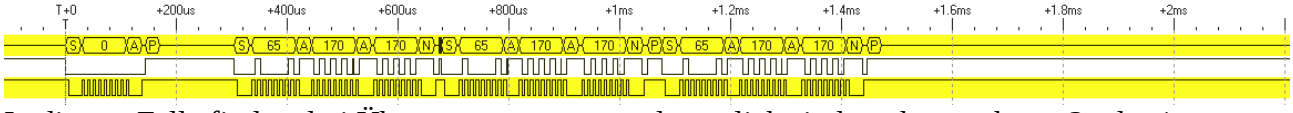
Start, Master-Transmitter

#08	8	18	28	28
#16	8	18	28	28
#32	8	18	28	28

Das TWSR-Protokoll zeigt, dass alle drei Master ihre Nachricht versandt und keine Fehlerbedingung erkannt haben.

Test 2

Drei Master lesen zeitgleich 2 Byte von einem PCF8574 (der immer das selbe Byte liefert)



In diesem Falle finden drei Übertragungen statt, gelegentlich sind es aber auch nur 2 oder 1.

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q
	8	40	50	58													

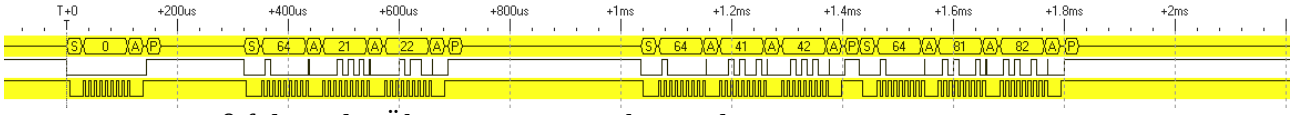
Start, Master-Receiver

#08	8	40	50	58		
#16	8	40	50	58		
#32	8	38	8	40	50	58

Master #32 hat einen Arbitrierungsfehler erkannt und musste seinen Transfer neu startet.

Test 3

Drei Master senden unterschiedliche Daten (#8: "21+22", #16: "41+42", #32: "81+82") an einen PCF8574.



Erwartungsgemäß folgen die Übertragungen nacheinander

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q
	8	18	28	28													

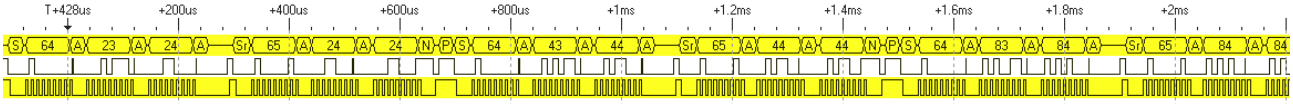
Start, Master-Transmitter

#08	8	18	28	28													
#16	8	18	38	8	18	28	28										
#32	8	18	38	8	18	28	28										

Master #8 setzt seine Nachricht direkt ab, Master #16 und #32 melden Arbitrierungsfehler und starten den Transfer neu.

Test 4

Die drei Master senden jeweils zwei Byte an den PCF8475 und lesen sofort zwei Byte wieder aus. Da der PCF8574 nur das letzte Byte speichert, müssen die beiden gelesenen Bytes identisch sein mit dem 2. gesendeten Byte.



Wie man sehen kann, stimmen Theorie und Praxis überein.

	8	18	28	28	10	40	50	58

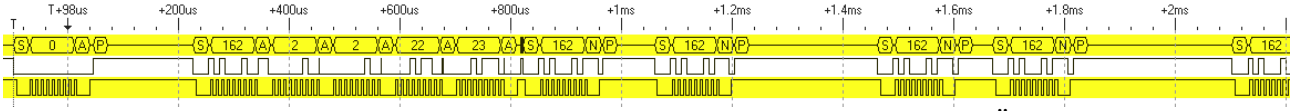
Start, Master-Transmitter, Rep.Start, Master-Receiver

#08	8	18	28	28	10	40	50	58							
#16	8	18	38	8	18	28	28	10	40	50	58				
#32	8	18	38	8	18	38	8	18	28	28	10	40	50	58	

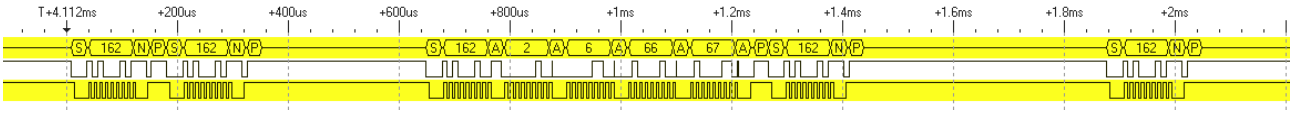
Zuerst gelingt #08 ein Transfer, #16 und #32 werden mit Arbitrierungsfehler in Warteschleifen geschickt, nacheinander können aber beide ihre Meldung absetzen.

Test 5

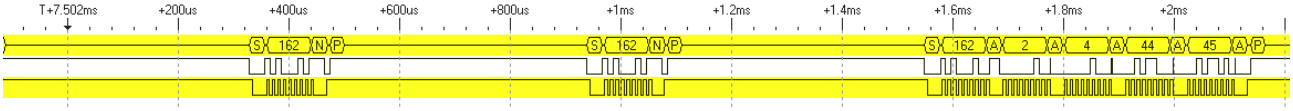
Die drei Master schreiben jeweils zwei unterschiedliche Bytes auf unterschiedliche Adressen eines Eeproms mit der TWI-Adresse 162.



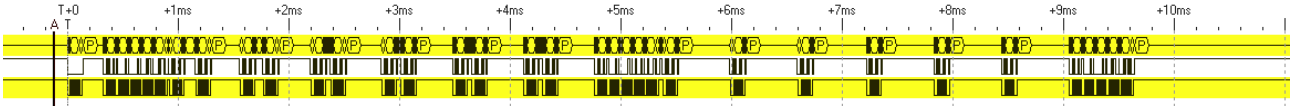
Der erste Master (mit den Bytes "22+23") kommt durch, danach werden alle Übertragungsversuche vom Eeprom mit einem NACK abgewiesen.



Erst nach ca. 5ms gelingt es dem nächsten Master, seine Daten ("66,67") zu schreiben.
Danach ist das Eeprom wieder damit beschäftigt, die Bytes einzusortieren.



Erst ca. 9ms nach Beginn der Transaktionen gelingt es auch dem 3. Master seine Daten ("44,45") loszuwerden.



Hier ist die gesamte Datenübertragung komprimiert dargestellt.

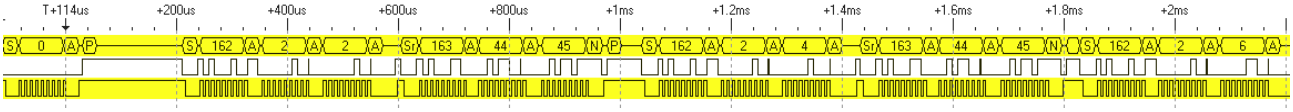
	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V
	8	18	28	28	28	28																
#08	8	18	28	28	28	28																
#16	8	18	28	38	8	20	8	20	8	20	8	20	8	20	8	20	8	20	8	20	8	20
#32	8	20	8	20	8	20	8	20	8	20	8	20	8	18	28	28	28	28				

Start, Master-Transmitter

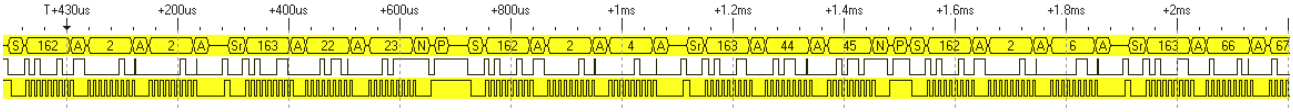
Im Protokoll sind die Transaktion von #16 ist noch vollständig dargestellt.
Man sieht die wiederholten Versuche zum Verbindungsaufbau bei #16 und #32 (Code 8), die durch ein NACK (Code 20) abgewiesen werden, weil das Eeprom noch busy ist.

Test 6

Drei Master lesen jeweils 2 Byte von unterschiedlichen Adressen eines Eeproms (siehe Test 5). Dazu wird zunächst die Adresse geschrieben, ein Repeated Start gesetzt und anschließend werden zwei Byte gelesen.



Das Eeprom kann hier ohne Zeitverzögerung nach dem Setzen der Adresse die Daten ausliefern. Schauen wir aber die Inhalte kritisch an, dann ist festzustellen, dass die gelesenen Daten bei allen Zugriffen gleich sind, obwohl unterschiedliche Adressen abgefragt und unterschiedliche Daten erwartet werden.
Das benutzte Eeprom (16kBit) ist offensichtlich nicht in der Lage, mit einem Repeated Start umzugehen.
Es wurde durch ein AT24C256 ersetzt.



Hier entsprechen die gelesenen Daten den vorher geschriebenen.
Auch der Schreibvorgang in Test 5 verläuft mit diesem Eeprom deutlich zügiger.

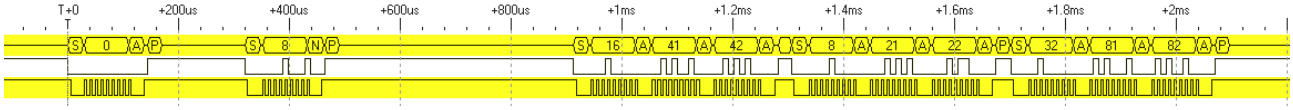
	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q
	8	18	28	28	10	40	50	58									
#08	8	18	28	28	10	40	50	58									
#16	8	18	28	38	8	18	28	28	10	40	50	58					
#32	8	18	28	38	8	18	28	28	10	40	50	58					

Start, Master-Transmitter, Rep.Start, Master-Receiver

Das Protokoll zeigt keine Auffälligkeiten.

Test 7

Jeder Master sendet 2 Byte an seinen Nachfolger, Master #8: ("41+42") an #16, #16: ("81+82") an #32 und #32: ("21+22") an #8.



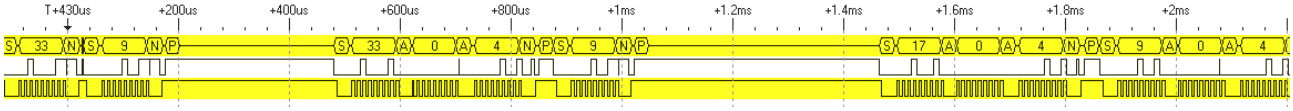
Der Sendeversuch von #32 an #8 scheitert zunächst, danach läuft alles rund.

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q
	8	18	28	28	Start, Master-Transmitter												
#08	8	38	8	18	28	28	60	80	80	a0							
#16	8	38	60	80	80	a0	8	18	28	28							
#32	8	20	8	18	28	28	60	80	80	a0							

Im Protokoll wird erkennbar: #8 und #16 sehen einen Arbitrierungsfehler, #32 erhält wird mit einem NACK ausgebremst. Danach läuft die Komunikation reibungslos.

Test 8

Jeder Master liest 2 Byte von seinem Nachfolger, Master #8 von #16, #16 von #32 und #32 von #8



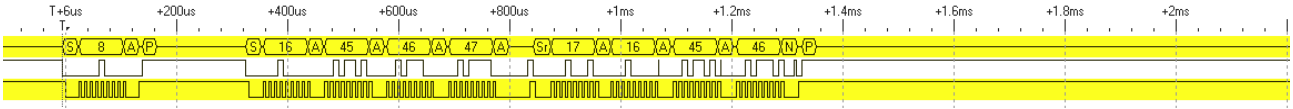
Master #8 und Master #32 scheitern zunächst, danach läuft die Transaktion.

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q
	8	40	50	58	Start, Master-Receiver												
#08	8	38	8	38	8	40	50	58	a8	b8	c0						
#16	8	48	8	40	50	58	a8	b8	c0								
#32	8	48	a8	b8	c0	8	48	8	40	50	58						

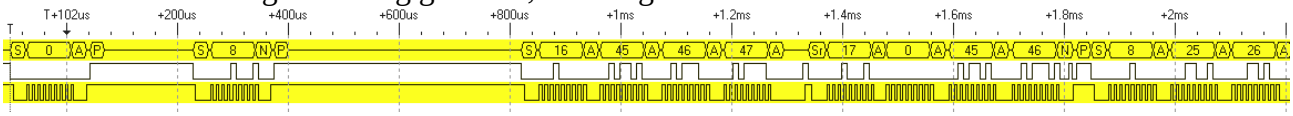
In der ersten Runde scheitern alle Master, #8 sieht einen Arbitrierungsfehler, #16 und #32 erhalten ein NACK zurück.
Im zweiten Versuch klappt dann die Verbindung.

Test 9

Jeder Master schreibt drei Byte an seinen Nachfolger, sendet ein Repeated Start und liest die drei Byte wieder aus..
Lässt man einen einzelnen Master die Aktion ausführen, dann zeigt sich der nachfolgende Verlauf.



An die Adresse 16 werden die Byte "45, 46, 47" gesendet und anschließend werden die Werte "16, 45, 46" ausgelesen. Das ist soweit korrekt.
Werden alle Master gleichzeitig gestartet, dann ergibt sich dieses Bild:



Zuerst ereignet sich eine Kollision, anschließend wird "45, 46, 47" geschrieben, aber "0, 45, 46" gelesen.
Was ist hier schief gelaufen?

Wenn ein einzelner Master eine Transaktion startet, dann arbeiten die anderen Master als Slave, prüfen laufend, ob Daten eingegangen sind und wenn ja, dann kopieren sie die empfangenen Daten in den Ausgangspuffer.
Der Master kann daher sofort die geänderten Daten aus dem Sendepuffer des Slaves lesen.

Anders sieht die Situation aus, wenn alle Master gleichzeitig starten.
Einer von ihnen wird zum Master, die anderen wollten Master werden, haben aber die Arbitrierung verloren oder ein NACK erhalten und warten in einer Warteschleife darauf, die Transaktion beginnen zu können.

In dieser Zeit können sie zwar interruptgesteuert als Slave arbeiten und Daten annehmen oder abholen lassen, sie haben aber keine Gelegenheit, eingegangene Daten auszuwerten.
Darum gelingt es ihnen nicht, den Sendepuffer wie erwartet neu zu schreiben.
Im Sendepuffer stehen unverändert die Daten, die am Beginn der Transaktion dort standen.
Erst wenn der wartende Master die Warteschleifen beendet hat und wieder im Slave-Modus arbeitet, kann er den Empfangspuffer auslesen und die Daten in den Sendepuffer schreiben.

Das ist aber leider zu spät.

Diese Problem tritt nur dann auf, wenn zwei Master zeitgleich sich gegenseitig Daten zusenden, bei denen der Schreibzugriff Veränderungen in den unmittelbar danach zu lesenden Daten verursacht.
(Das ist also das gleiche Problem, das im Test 6 das zuerst verwendete Eeprom hat.)

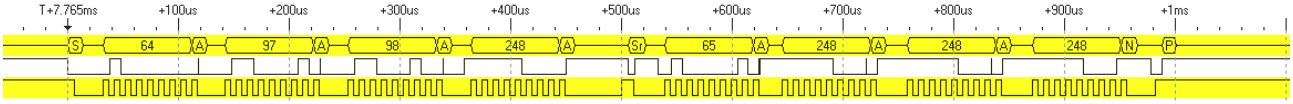
	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W
	8	18	28	28	28	10	40	50	50	58	Start, Master-Transmitter, Rep.Start, Master-Receiver												
#08	8	38	8	18	28	28	28	10	40	50	50	58	60	80	80	80	a0	a8	b8	b8	c0		
#16	8	38	60	80	80	80	a0	a8	b8	b8	c0	8	38	8	18	28	28	28	10	40	50	50	58
#32	8	20	8	18	28	28	28	10	40	50	50	58	60	80	80	80	a0	a8	b8	b8	c0		

Zur Vervollständigung hier noch das Protokoll der TWSR-States.

Stresstest

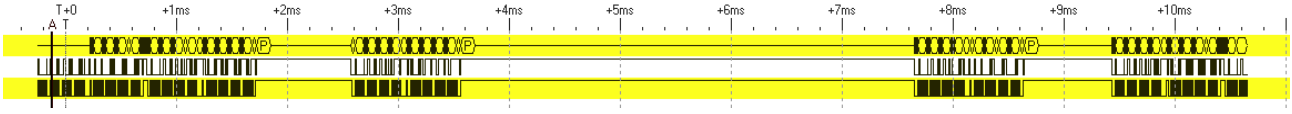
Ein weiterer Test sollte zeigen, wie sich Kollisionen bei zunehmender Busaktivität entwickeln und ob sie den Verkehr auf dem Bus zusammenbrechen lassen.

Der Bus ist mit 100 kHz getaktet, jede einzelne der Übertragung dauert ca. 1ms.
Die Master arbeiten von Timern gesteuert und senden im Abstand von knapp 8ms jeweils ein Datenpaket.
Um Interferenzen zu erzeugen, ist die Wartezeit der Timer geringfügig unterschiedlich gewählt.



Jeder Master schreibt drei unterschiedliche Bytes an den PCF8574 und liest anschließend 3 Bytes wieder aus. Beim Zurücklesen muss 3x das 3. gesendete Byte empfangen werden.
Bei einem Bus-Fehler oder einem Fehler im Vergleich der Daten beendet der Master seine Aktivität mit blinkender LED (PB0).

Bei drei aktiven Mastern sieht der Busverkehr so aus:



Der Bus ist während des Grundtaktes von 8ms etwa für 3ms belegt, das ist eine Auslastung von fast 40%. Es wurden keine Probleme erkennbar.

5.5 Schlussfolgerung

Unter den Bedingungen der Tests lösen sich alle Kollisionen auf.

Ein Problem, das nur in einer ganz spezieller Situation auftritt, muss bei der aktuellen Version der Software des Masters beachtet werden:

Wenn ein Master Daten an einen anderen Master schreibt, dann kann dieser seinen Sendepuffer erst dann neu schreiben, wenn er als Slave dazu die Gelegenheit hat.
Bei einem Write() mit anschließendem Read() ohne Busfreigabe dazwischen ist diese Bedingung nicht erfüllt.
Diese Situatution entsteht dann, wenn sich zufälligerweise beide Master zeitgleich gegenseitig adressieren.

Zu beachten ist auch, dass durch Veränderung der Testbedingung (Länge der Datensätze, Einstellung der Wartezeiten, Anzahl der an einer Kollision beteiligten Master, Auslastung des Busses) die Karten neu gemischt werden und möglicherweise nicht vorhergesehene neue Probleme entstehen.

Eine Fehlerfreiheit können die Tests leider nicht beweisen.

Es wird also ratsam sein, zusätzliche Maßnahmen zu treffen, damit die fehlerfreie Zustellung von Daten überprüft werden kann.

Außerdem sollte man darüber nachdenken, ob eine "Single-Master-Bus"-Lösung die gestellte Aufgaben nicht auch erfüllen kann.

Michael S.

15.03.2012

Als Interface zwischen PC (Terminalprogramm) und TWI-Bus dient ein ATMEGA, der als TWI-Master bidirektional Daten zwischen Serieller Schnittstelle und TWI-Bus vermitteln kann. Nebenbei ist er als TWI-Slave auf dem TWI-Bus adressierbar, um Debug-Informationen über die Serielle Schnittstelle am PC ausgeben zu können.

Die Software dazu findet man hier:

`http://www.mikrocontroller.net/topic/106752#2429368`

Die Software für den TWI-Multi-Master beruht auf Komponenten, die hier dokumentiert sind:

`http://www.mikrocontroller.net/topic/249639#2576717`

Sie alle basieren auf AppNotes und überarbeiteten Beispielcodes der Fa. Atmel.