

HARDWARE DOKUMENTATION

Avisaro Module (WLAN und CF Memory)

- Spannungsversorgung, Beschaltung IO-Pins
 - Schnittstellen: Belegung und Timing (RS232, I2C, SPI, CAN)
 - Pinbelegung der Avisaro Modul
 - Abmessungen
 - Programmierbeispiele
-



Änderungshistorie

2004-06-24		Erstellung
2004-09-27		Erweiterungen nach Kundenfeedback
2004-10-31		Erweiterungen Compact Flash Memory Modul
2005-02-03		Viele Detailergänzungen
2005-05-01		Schnittstellen Dokumentation hinzugefügt. Programmierbeispiele hinzugefügt.
2005-06-01		Fehler beim Parallel-Protokoll beseitigt
2007-02-26		Runderneuert
2007-03-28		CAN hinzugefügt

Kontakt:

Avisaro AG
Vahrenwalderstr. 7 (tch)
30165 Hannover

Telefon:

+49-(0)511-7809390

Telefax:

+49-(0)511-35319624

eMail:

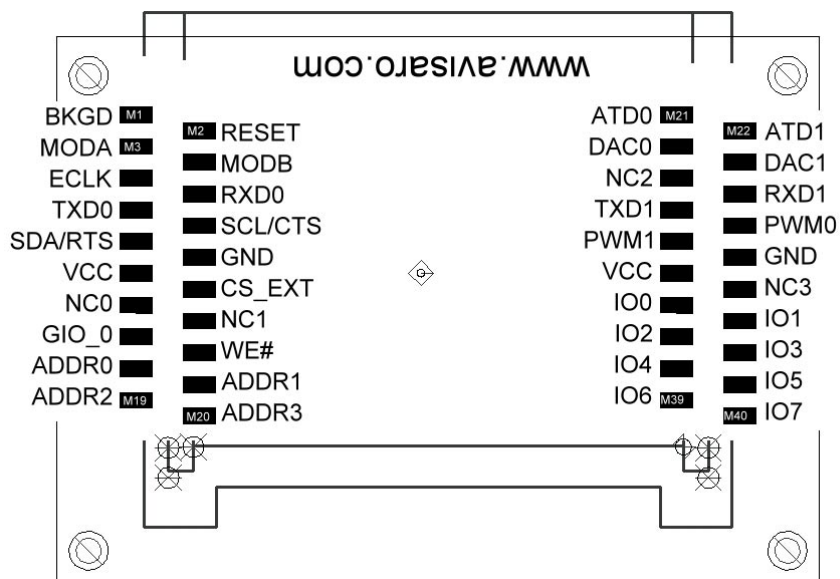
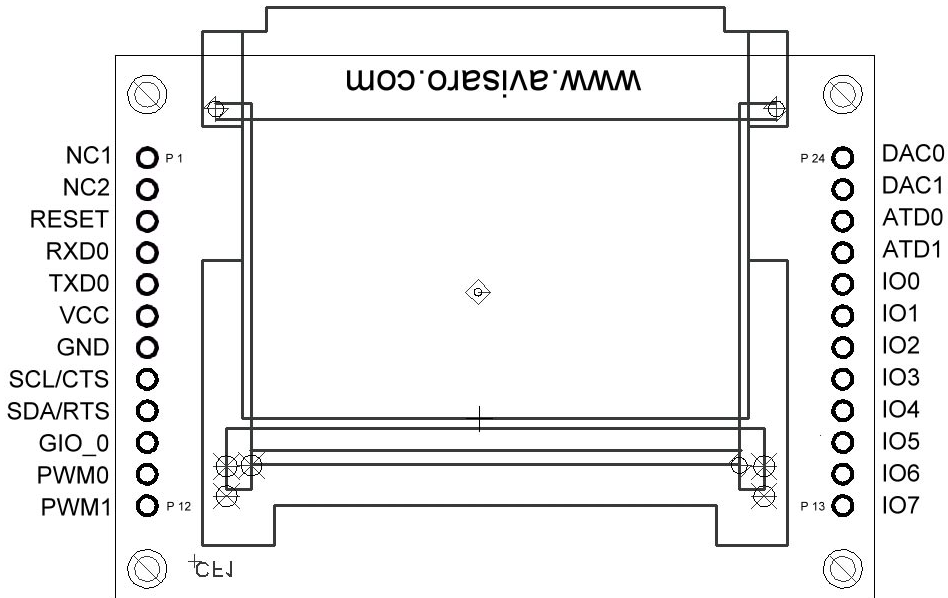
support@avisaro.com

INHALTSVERZEICHNIS

Inhaltsverzeichnis	3
PINLAYOUT	4
Abmessungen	6
Stromversorgung und IO-Pins	7
Stromversorgung	7
IO-Pins	7
Serielle Schnittstelle - RS232	8
Serielle Schnittstelle – I2C (Slave)	9
Einstellungen	9
Einfacher Test	9
Datenaustausch mit dem Avisaro-Modul über I ² C	10
Daten zum Avisaro-Modul senden	10
Daten vom Avisaro-Modul empfangen	10
Source-Code Beispiel	11
Serielle Schnittstelle – SPI (Slave)	12
Konfiguration	12
Byte Stuffing bei der Übertragung	13
Datenaustausch zwischen SPI Mastern und dem AVISARO-Modul	14
Datenflußkontrolle	15
Source-Code Beispiel	15
Serielle Schnittstelle – CAN	16
Defaulteinstellungen	16
Arbeitsweise	16
Status Pin Belegung „CF Memory Modul“	18
Status Pin-Belegung „WLAN“	19
MicroMatch Connector (SMD)	21
Programmierbeispiele	22
I2C Schnittstelle	22
SPI Schnittstelle	23
TCP/IP (Win-Socket) Programmierung	26

PINLAYOUT

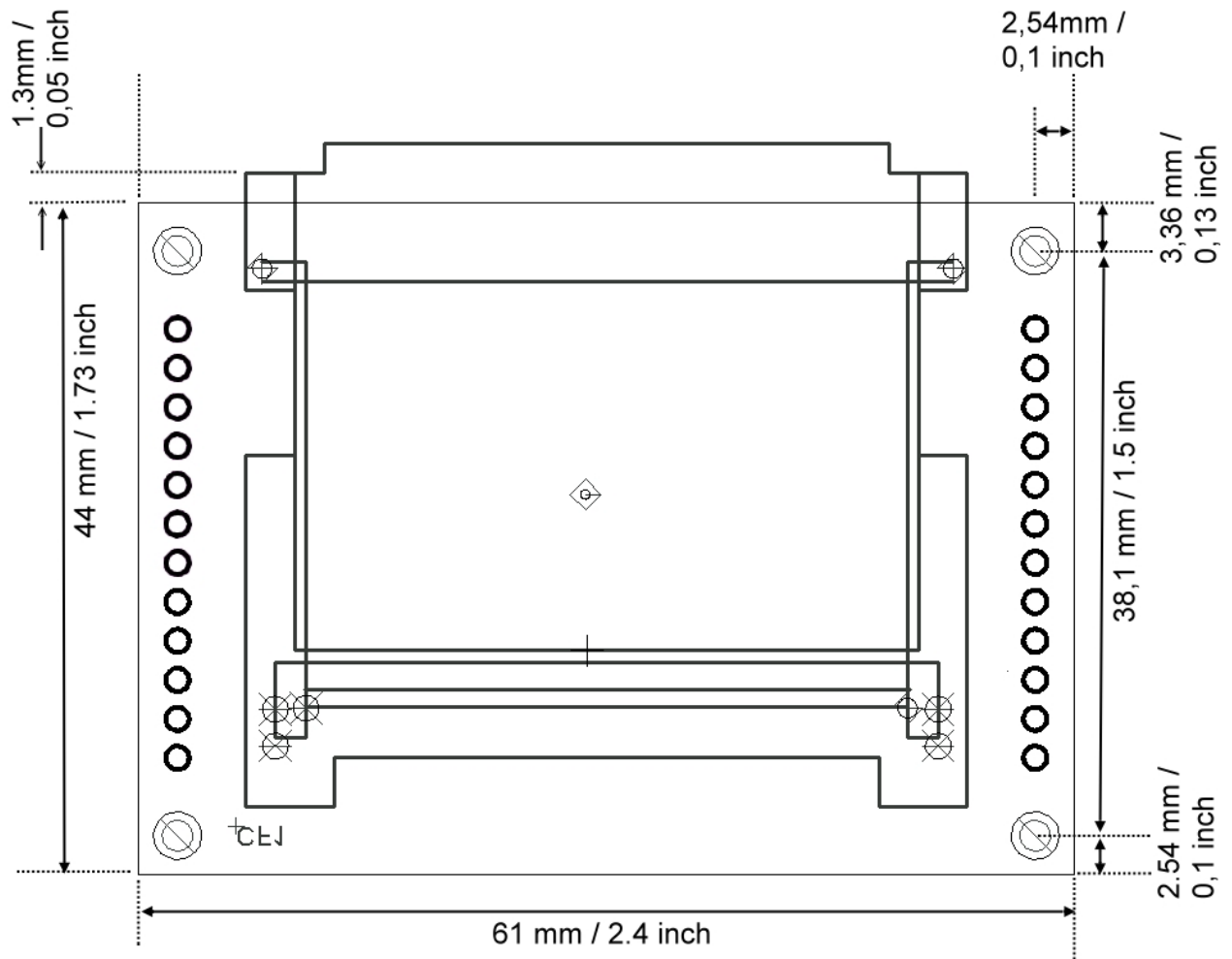
Das Avisaro Modul hat zwei Anschlussmöglichkeiten: Zwei Pin-Reihen für Stiftleisten im 2.54 mm Raster und zwei Reihen mit dem MikroMatch Steckverbinder. Die Abbildungen zeigen das Modul jeweils in Aufsicht.



Pin	MicroMatch	Stiftleiste	I/O	Beschreibung
ATD0	M21	P22	I	Analog zu Digital – Kanal 0
ATD1	M22	P21	I	Analog zu Digital – Kanal 1
DAC0	M23	P24	I	CAN RX / Digital zu Analog – Kanal 0
DAC1	M24	P23	O	CAN TX / Digital zu Analog – Kanal 1
NC2	M25	-	-	Nicht belegt 2
RXD1	M26	-	I	RS232 Kanal 1 (sekundär) - Empfang
TXD1	M27	-	O	RS232 Kanal 1 (sekundär) – Senden
PWM0	M28	P11	I/O	Puls Weiten Modulator 0 / GIO
PWM1	M29	P12	I/O	Puls Weiten Modulator 1 / GIO
GND	M30	P7	PWR	Versorgung: Masse
VCC	M31	P6	PWR	Versorgung: 3,3 V
NC3	M32	-	-	Nicht belegt 3
IO0	M33	P20	I/O	Benutzer Digital IO 0
IO1	M34	P19	I/O	Benutzer Digital IO 1
IO2	M35	P18	I/O	Benutzer Digital IO 2
IO3	M36	P17	I/O	Benutzer Digital IO 3
IO4	M37	P16	I/O	Benutzer Digital IO 4
IO5	M38	P15	I/O	Benutzer Digital IO 5
IO6	M39	P14	I/O	Benutzer Digital IO 6
IO7	M49	P13	I/O	Benutzer Digital IO 7
BKGD	M1	-	-	Debug / Programmier Modus
RESET	M2	P3	I	Reset (Microprozessor)
MODA	M3	-	-	Debug / Programmier Modus
MODB	M4	-	-	Debug / Programmier Modus
ECLK	M5	-	-	Debug / Programmier Modus
RXD0	M6	P4	I	RS232 Kanal 0 (primär) – Empfangen
TXD0	M7	P5	O	RS232 Kanal 0 (primär) – Senden
SCL/CTS	M8	P8	O	I2C Bus – SCL Leitung CTS Flusskontr.
SDA/RTS	M9	P9	I	I2C Bus – SDA Leitung RTS Flusskontr.
GND	M10	P7	PWR	Versorgung: Masse
VCC	M11	P6	PWR	Versorgung: 3,3 V
CS_EXT	M12	-	O	Chip Select – externe Komponente
NC0	M13	-	-	Nicht belegt 0
NC1	M14	-	-	Nicht belegt 1
GIO_0	M15	P10	I/O	Benutzer Digital IO – General Purpose
WE#	M16	-	O	Write Enable – externe Komponente
ADDR0	M17	-	O	Adressbus 0 – externe Komponente
ADDR1	M18	-	O	Adressbus 1 – externe Komponente
ADDR2	M19	-	O	Adressbus 2 – externe Komponente
ADDR3	M20	-	O	Adressbus 3 – externe Komponente

ABMESSUNGEN

Die Höhe des Avisaro Moduls beträgt 15 mm im aufgestecktem Zustand bei Verwendung der MikroMatch Steckerverbinder. Das Modul wiegt 18g (ohne Funk- bzw Speicherkarte).



Die MikroMatch Stecker haben in der Regel eine ausreichende Haltekraft, um das Modul zu halten. Die zusätzlichen Löcher für eine Verschraubung des Moduls sind nur notwendig, wenn Vibrationen (z.B. bei Verwendung in Baufahrzeugen) auftauchen.

STROMVERSORGUNG UND IO-PINS

STROMVERSORGUNG

Das Avisaro Modul wird mit 3.3 V (+/- 0,1V) betrieben. Der verwendete Prozessor kann auch bei 5V betrieben werden – da jedoch fast alle Compact Flash Medien nur für 3.3 V spezifiziert sind, empfiehlt es sich bei 3.3 V Spannungsversorgung zu bleiben. Für WLAN Anwendungen ist eine 3.3V Spannungsversorgung zwingend. Ein gewöhnlicher 3.3V Festspannungsregler hat sich bewährt, um das Modul in einer 5V – Umgebung zu betreiben.

Der Stromverbrauch schwankt je nach Applikation. Bei wireless LAN Karten liegt dieser bei ca. 300mA, mit eingeschaltetem Power Save Mode bei 80 mA. Bei Memory Modulen liegt er bei ca. 100mA.

IO-PINS

Die digitalen Eingänge des Avisaro Moduls sind 5V tolerant. D.h. ein Eingang des Avisaro Moduls kann mit 5V angesprochen werden, auch wenn das Modul sonst mit 3.3V arbeitet. So ist eine Integration in vorhandene 5V Systeme einfach.

Die digitalen Ausgänge haben eine Belastbarkeit von 50mA.

Lassen Sie alle nicht benötigten Leitung offen. Pull-Up oder Pull-Down Widerstände sind nicht notwendig.

SERIELLE SCHNITTSTELLE - RS232

Die RS232 Schnittstelle ist im Internet ausreichend dokumentiert, sodass hier nicht das Timing wiederholt wird. Das Avisaro Modul unterstützt Baudraten von 1.200 bis 115.200 Baud in den üblichen Abstufungen. Es werden verschiedene Parity unterstützt. Es werden nur 8 Datenbits unterstützt. Zur Flusskontrolle wird „keine“, „Software“ und „Hardware“ unterstützt.

Entsprechende Signale liegen wie folgt am Modul an:

Pin	MicroMatch	Stiftleiste	I/O	Beschreibung
RXD0	M6	P4	I	RS232 Kanal 0 (primär) – Empfangen
TXD0	M7	P5	O	RS232 Kanal 0 (primär) – Senden
SCL/CTS	M8	P8	O	CTS Flusskontrolle '1' = Stop Sende '0' = Ok zum Senden
SDA/RTS	M9	P9	I	RTS Flusskontrolle '1' = Stop Sende '0' = Ok zum Senden

Da „Rx“ und „Tx“ von der Betrachtung abhängen, sei die Datenrichtung klargestellt: RX – Das Avisaro Modul empfängt Daten. TX – Das Avisaro Modul sendet Daten. CTS – Das Avisaro Modul signalisiert Empfangsbereitschaft. RTS – Das Avisaro Modul bekommt Sendeerlaubnis.

Die „AT-Befehle“ zur Konfiguration der Schnittstelle finden Sie im Programmierhandbuch.

SERIELLE SCHNITTSTELLE – I2C (SLAVE)

Die I²C-Bus Implementation im Avisaro-Modul entspricht einem I²C-Bus Slave d.h. das Avisaro-Modul kann nicht von sich aus kommunizieren und muss deshalb von einem Master-Device ständig gepollt werden. Es wird das I²C-Bus Modul des Controllers verwendet. Die Taktfrequenz ist Standard- oder Fast I²C (100 kHz / 400 kHz). Das I2C Clockstretching zur Flusskontrolle wird unterstützt (kann per AT-Befehl ein und ausgeschaltet werden).

Die Default Adresse auf dem I2C Bus ist die ‚73‘.

Pin	MicroMatch	Stiftleiste	I/O	Beschreibung
SCL/CTS	M8	P8	O	I2C Bus – SCL Leitung CTS Flusskontr.
SDA/RTS	M9	P9	I	I2C Bus – SDA Leitung RTS Flusskontr.

Der I2C Bus ist ausreichend z.B. im Internet dokumentiert, so dass hier nicht das Timing wiederholt werden muss.

EINSTELLUNGEN

Die einzige Einstellung, die der Benutzer ändern kann, ist die I2C-Bus Adresse des Moduls. I2C-Bus Adressen gehen von 0...127 wobei '0' die Broadcast-Adresse (General Call Address) ist. Das Avisaro-Modul kennt keine 'Reservierten Adressen' (siehe I2C-Bus Specification, Seite 16), sondern reagiert nur auf seine eigene (Slave) Adresse.

Ein jungfräuliches Modul hat die Adresse '73'. Wenn Sie ein Flash-Update mit einem Modul vor dem 1. April 2005 gemacht haben, dann steht die Adresse auf ‚128‘.

Um die Adresse zu ändern, muß der Benutzer das AT-Kommando:

```
AT+I2C ADDRESS <xxx>
```

eingeben, wobei <xxx> ein beliebiger Wert zwischen 0 und 127 sein darf. Zum Anzeigen der Adresse gibt es das Kommando:

```
AT+STATUS I2C
```

Um das 'Huhn und Ei' Problem zu lösen (Sie müssen die I2C Adresse ändern, um überhaupt Befehle an das Modul schicken zu können), können Sie eine Compact Flash Speicherkarte benutzen, um das Avisaro Modul zu konfigurieren. Sie dazu das Benutzerhandbuch. (Kurz: erzeugen Sie eine Datei avi_config.txt mit den at-Befehlen – diese wird nach dem Einschalten wie eine Skriptdatei ausgeführt).

EINFACHER TEST

Das Avisaro-Modul speichert kurz nach dem Einschalten die Ausgabe der AT-Maschine OK (einschliesslich CR/LF, also 4 Bytes) in seinem Ausgangspuffer. Wenn der I2C Busmaster sich diese Daten abholen kann, funktioniert die I2C-Kommunikation.

DATENAUSTAUSCH MIT DEM AVISARO-MODUL ÜBER I²C

Das Avisaro-Modul hat jeweils einen Eingangs- und Ausgangspuffer (FIFO). Der Eingangspuffer wird von einem externen I2C -Gerät gefüllt und vom Avisaro-Modul sehr schnell verarbeitet. Der Ausgangspuffer enthält Daten vom Avisaro-Modul, die sich ein I2C Busmaster abholen kann. Sollte der Busmaster längere Zeit keine Daten aus dem Modul auslesen (im Falle einer offenen TCP-Verbindung und wenn die Gegenstelle etwas sendet), blockiert das Modul. Der blockierende Zustand kann durch Senden der Escape-Sequenz (+<pause>+<pause>+) aufgehoben werden. Die über TCP empfangenen Daten gehen dabei allerdings verloren. Besser ist es also, man holt die Daten ab.

DATEN ZUM AVISARO-MODUL SENDEN

Daten werden als “Stream” zum Avisaro-Modul gesendet. Der Ablauf aus Sicht des Busmasters ist folgender:

1. Setze die I2C START Bedingung.
2. Sende die I2C Adresse des Avisaro-Moduls um eine Stelle nach links geshiftet als Byte (8 Bits), Bit 0 muß gelöscht sein (I2C Adressen belegen die Bits 1...7, Bit 0 ist das Read/Write Bit).
3. Warte auf ACK vom Avisaro-Modul (Bit 9).
4. Sende ein Byte der Daten (8 Bits).
5. Warte auf ACK vom Avisaro-Modul (Bit 9).
6. Gehe zu Schritt 4 bis alle Daten verschickt wurden.
7. Setze die I2C STOP Bedingung.

DATEN VOM AVISARO-MODUL EMPFANGEN

Anders als beim Versenden der Daten läuft der Empfang ab. Die ersten beiden Bytes, die der Busmaster beim Empfang liest, geben an wieviele Daten im Ausgangspuffer des Avisaro-Moduls bereit stehen. Das erste Byte ist dabei das MSB (höherwertige Teil) und das LSB kommt direkt danach. Dieser 16-Bit Wert gibt an, wieviele Daten in Folge aus dem Modul gelesen werden können. Der Busmaster kann, muß aber nicht, die erhaltene Anzahl an Daten sofort auslesen. Wichtig dabei ist: Beim letzten gelesenen Byte darf der Busmaster kein ACK senden (siehe I2C-Bus Specification Seite 14). Sollte er es doch tun, dann geht ein Byte verloren. (Das Avisaro-Modul würde ein weiteres Byte senden, dass der Busmaster aber nicht abholen würde).

Der Ablauf beim Empfang ist folgender:

- 1) Setze die I2C START Bedingung.
- 2) Sende die I2C Adresse des Avisaro-Moduls um eine Stelle nach links geshiftet, bei gesetztem Bit 0, als Byte (8 Bits), Bit 0 muß gesetzt sein (I2C Adressen belegen die Bits 1...7, Bit 0 ist das Read/Write Bit).
- 3) Warte auf ACK vom Avisaro-Modul (Bit 9).

- 4) Lese das erste Byte aus dem Avisaro-Modul (8 Bits).
- 5) Setze ACK (Bit 9).
- 6) Lese das zweite Byte aus dem Avisaro-Modul (8 Bits).
- 7) Verknüpfe beide Bytes zu einem 16-Bit Wert, Anzahl = $\text{Byte1} \ll 8 \mid \text{Byte2}$
- 8) Ist dieser Wert 0 (keine Daten) oder 0xffff (Kein Kontakt zum Modul) gehe zu Schritt 15
- 9) Setze ACK (Bit 9).
- 10) Lese ein Byte der Daten aus dem Avisaro-Modul (8 Bits).
- 11) Setze ACK (Bit 9).
- 12) Bis Anzahl-1 oder ein Byte weniger als gewünscht: gehe zu Schritt 10.
- 13) Lese letztes Byte der Daten aus dem Avisaro-Modul (8 Bits),
- 14) Setze kein ACK (Bit 9).
- 15) Setze die I2C STOP Bedingung.

SOURCE-CODE BEISPIEL

Siehe Kapitel „I2C Schnittstelle“ auf Seite 22.

SERIELLE SCHNITTSTELLE – SPI (SLAVE)

Die AVISARO-Module verwenden das Standard 4-Draht SPI (Serial Peripheral Interface) und sind als SPI-Slaves ausgelegt. SPI basiert auf einem verteilten 16-Bit Schieberegister wobei sich die eine Hälfte im Master und die andere Hälfte im Slave befindet.

Zum Datenaustausch mit dem AVISARO-Modul muß ein SPI-Master...

- Den Eingang /SS auf “0” ziehen
- Taktimpulse auf der SCK-Leitung erzeugen
- Bits auf der MOSI-Leitung ausgeben
- Bits von der MISO-Leitung empfangen

Pro Taktimpuls wird ein Bit vom Master in den Slave und gleichzeitig ein Bit vom Slave in den Master befördert. Nach acht Taktimpulsen wurde ein Byte vom Master zum Slave und eins vom Slave zum Master übertragen. Das Schieberegister des AVISARO-Moduls ist nicht konfigurierbar d.h. daß die Anzahl der vom Master erzeugten Taktimpulse immer ein Vielfaches von Acht sein muß.

Pin	MicroMatch	Stiftleiste	I/O	Beschreibung
IO4	M37	P16	O	Master in / Slave out Datentransfer: Slave → Master
IO5	M38	P15	I	Master out / Slave in Datentransfer: Master → Slave
IO6	M39	P14	I	Serial Clock - Takt vom Master
IO7	M49	P13	I	Slave Select (low aktiv)

Ist die /SS-Leitung “1”, dann sind alle anderen SPI-Leitungen des AVISARO-Moduls hochohmig. Daher können mehrere Geräte und AVISARO-Module an einem Bus betrieben werden.

KONFIGURATION

Einige Eigenschaften des SPI im AVISARO-Modul sind konfigurierbar. Die Konfiguration geschieht über den AT+SPI Befehl und wird im EEPROM gespeichert. Eine Änderung der Konfiguration ist erst nach dem nächsten Neustart des Moduls wirksam.

AT+SPI <polarity> <phase> <bit order>

AT+SPI benötigt drei Parameter.

Der erste Parameter (*polarity*) teilt dem AVISARO-Modul die Polarität des Taktsignals mit, welches der Master verwenden möchte. Erlaubt sind die Schlüsselwörter HIGH und LOW. HIGH sagt dem Modul, daß der Master positive Taktsignale verwendet. Bei LOW dagegen reagiert das Modul auf negative Taktsignale des Masters.

Mit dem zweiten Parameter (*phase*) wird dem AVISARO-Modul mitgeteilt, welche Flanke des Taktsignals es als Änderung desselben erkennen soll. Erlaubt sind hierbei die Schlüsselwörter ODD und EVEN. Mit ODD wird das Modul so konfiguriert, daß bei jeder ungeradzahligen Flanke des Taktsignals ein Datenaustausch (ein Bit) stattfindet. Das heißt, daß die Flanken 1, 3, 5, 7... 15 (bei acht Taktimpulsen) ausschlaggebend sind. Bei EVEN werden die Flanken 2, 4, 6, 8, ... 16 verwendet.

ACHTUNG/BESONDERHEIT: Ist der Parameter *<phase>* EVEN, dann kann der Master nach dem herunterziehen der Slave-Select Leitung (/SS) kontinuierliche Taktimpulse erzeugen (immer ein Vielfaches von Acht). Dies ist die bevorzugte Betriebsart. Ist *<phase>* ODD, dann muß der Master nach jedem übertragenen Byte die Slave-Select Leitung wieder hochziehen. Das Hochziehen der /SS-Leitung setzt auch den "Byte Stuffer" und den Status der Flußkontrolle (siehe unten) zurück. Diese Übertragungsart sollte deshalb nur für langsamen Betrieb und reine Textdaten benutzt werden.

Der dritte und letzte Parameter (*bit order*) bestimmt, die Arbeitsrichtung der Schieberegister. Erlaubt sind die Schlüsselwörter LSB und MSB. Wird LSB angegeben, dann muß der Master das jeweils niederwertigste Bit (LSB) als erstes auf den Ausgang (MOSI Leitung) legen, bevor er ein Taktsignal ausgibt. Parallel dazu bekommt er vom Slave dessen niederwertigstes Bit zuerst gesendet. Ist der Parameter MSB, dann arbeiten beide Schieberegister anders herum.

Per Default ist das AVISARO-Modul mit...

- Taktsignale sind high-aktiv
- Bei geradzahlige Flanken findet der Datentransfer statt
- Es werden die niederwertigsten Bits zuerst ausgetauscht

...konfiguriert, was dem AT-Befehl "AT+SPI HIGH EVEN LSB" entspricht.

BYTE STUFFING BEI DER ÜBERTRAGUNG

Da beim SPI immer ein synchroner Datenaustausch stattfindet, gibt es keine Möglichkeit, Daten zu senden, ohne auch gleichzeitig etwas zu empfangen. Natürlich gilt das auch für die umgekehrte Richtung: Wer etwas empfangen will, muß auch senden.

Um mit dem SPI asynchron Daten auszutauschen, muß man sich eines Protokolls bedienen, daß eine "ungültige Information" definiert. Diese ungültige Information muß ein Teilnehmer immer dann senden, wenn er nichts mitzuteilen hat. Um diese Information in einem Datenstrom unterzubringen, der für gewöhnlich alle 256 Bitkombinationen als gültige Daten enthalten kann, bedient man sich einer einfachen Technik - dem sogenannten "Byte Stuffing".

Mit Byte Stuffing kann man bestimmte Werte aus dem Datenstrom ausblenden, indem diese Bytewerte durch eine Folge von zwei anderen Bytes ersetzt werden. Weil sich so eine Bytefolge natürlich auch in den Originaldaten befinden kann, muß das erste Byte einer solchen Folge ebenfalls ersetzt werden – einfach durch eine Verdopplung desselben.

Beispiel: Es soll mit Byte Stuffing der Wert 192 durch 201, 202 ersetzt werden.

Original	Kodiert
1,2,202,3,4	1,2,202,3,4
1,2,192,3,4	1,2,201,202,3,4
1,2,201,3,4	1,2,201,201,3,4
1,2,201,201,3,4	1,2,201,201,201,201,3,4
1,2,201,192,3,4	1,2,201,201,201,202,3,4

DATENAUSTAUSCH ZWISCHEN SPI MASTERN UND DEM AVISARO-MODUL

Ein SPI-Master, der mit dem AVISARO-Modul kommuniziert, muß Byte Stuffing beim Senden anwenden und empfangene “byte stuffed” Daten wieder dekodieren.

Das AVISARO-Modul verwendet folgende Definitionen für sein Byte Stuffing:

Name	Wert (hex)	Bedeutung
BSB_DUMMY	0xd4	Das “Dummy Byte”. Wird immer dann gesendet, wenn nichts zu senden ist.
BSB_ESCAPE_1	0xd5	Byte Stuffing Header Byte. Wird als erstes Byte einer 2-Byte Kombination gesendet.
BSB_ESC_DUMMY	0xd6	Byte Stuffed Dummy Byte. Wird gesendet, wenn das Dummy Byte in den Originaldaten enthalten ist.
BSB_XOFF	0xd7	Dient zur Flußkontrolle. Damit kann das Senden der Gegenstelle angehalten werden.
BSB_XON	0xd8	Flußkontrolle. Damit wird das Senden der Gegenstelle wieder zugelassen.

Bedeutung	Kodiert
Dummy Byte in den Originaldaten	0xd5, 0xd6
Byte Stuffing Header in den Originaldaten	0xd5, 0xd5
Gegenstelle soll das Senden anhalten	0xd5, 0xd7
Gegenstelle kann weitersenden	0xd5, 0xd8
Dummy, keine Information	0xd4

DATENFLUßKONTROLLE

Das AVISARO-Modul fügt eine XOFF-Message (0xd5, 0xd7) in den Datenstrom ein, wenn sein Empfangspuffer überzulaufen droht. Der SPI-Master sollte dann die Sendegeschwindigkeit drosseln, indem er “Dummy Bytes” (0xd4) sendet, bis das Modul eine XON-Message (0xd5, 0xd8) gesendet hat.

Auch kann der Master den gleichen Mechanismus verwenden, also mit XON/XOFF dafür sorgen, daß das AVISARO-Modul seinen Empfangspuffer nicht zum Überlaufen bringt.

SOURCE-CODE BEISPIEL

Source-Code Beispiele für die Programmierung der SPI Schnittstelle finden Sie im Anhang (

SERIELLE SCHNITTSTELLE – CAN

Die CAN Schnittstelle am Avisaro Modul unterstützt das „Base Frame Format“ und das „Extended Frame Format“ mit Geschwindigkeit bis zu 1 Mbit/s. Damit werden CAN 2.0 A und CAN 2.0 B unterstützt.

Das Avisaro Modul hat die Signale „CAN RX“ und CAN TX“ herausgeführt. Ein „CAN Bus Transceiver“ Chip ist notwendig, wenn das Modul an einen CAN Bus angeschlossen werden soll. Solche Transceiver kosten wenige Euro und sind sehr leicht anzuschließen. Bauteile wie die SN65 Familie von Texas Instruments sind geeignet.

Pin	MicroMatch	Stiftleiste	I/O	Beschreibung
DAC0	M23	P24	I	CAN RX / Digital zu Analog – Kanal 0
DAC1	M24	P23	O	CAN TX / Digital zu Analog – Kanal 1

Im „Programmierhandbuch“ sind die Befehle zur Konfiguration der Schnittstelle beschrieben.

DEFAULTEINSTELLUNGEN

Das Avisaro Modul ist im Auslieferungszustand auf folgende Werte eingestellt:

- Baudrate: 125 kbit/s
- Standard ID / Base Frame Format / CAN 2.0 A
- Das Modul empfängt Nachrichten auf ID: hex 0x49 (dezimal 73)
- Das Modul antwortet mit Nachrichten auf der ID: hex 0x49 (dezimal 73)

Diese Einstellungen können über entsprechende AT-Befehle verändert werden (siehe Programmierhandbuch). Hat Ihr System andere Einstellungen, können die Module über folgende drei Arten umkonfiguriert werden (siehe dazu ebenfalls Programmierhandbuch):

- 1) Über eine Konfigurationsdatei auf einer Speicherkarte
- 2) Über die Konfigurationswebseite (nur bei WLAN)
- 3) Über die Datenschnittstelle an einem passenden Host.

ARBEITSWEISE

Das Avisaro Modul lauscht auf der „Standard Message ID“ (at+canstdrx / at+can stdrx) auf Nachrichten. Nachrichten an diese ID werden, je nach momentanen Betriebszustand, interpretiert bzw gespeichert oder weitergeleitet.

Wird z.B. die Nachricht „at+status module \r\n“ (bei WLAN Modul) / „at+status“ (bei CF Modul) in CAN Nachrichten zerlegt an das Modul geschickt, so antwortet dieses mit CAN Nachrichten mit entsprechendem Inhalt.

Ist das Modul im Logging Modus (eine Datei ist zum Schreiben geöffnet) bzw im Datentransfer Modus (eine Verbindung ist geöffnet), dann wird der Inhalt der CAN Nachrichten an das Modul ausgepackt und in die Datei gespeichert / über TCP weitergeleitet. Umgekehrt werden Inhalte aus einer Datei / aus einem TCP Packet in CAN Nachrichten verpackt und auf den Bus ausgegeben.

Es lässt sich ein zweiter ID Bereich definieren. Dieser Bereich wird über eine ID und eine MASKE definiert. Der Inhalt der Nachrichten an diesen Bereich werden empfangen und in eine geöffnete Datei oder geöffnete TCP Verbindung geschrieben. Über diesen Bereich lassen sich keine Befehle an das Modul schicken.

In Kürze Verfügbar:

- Messaging Mode: CAN Nachrichten werden als komplette Nachricht gespeichert bzw Weitergeleitet. So ist mit WLAN eine CAN – WLAN Brücke möglich. Mit dem Speichermodul ist ein CAN Nachrichtenlogger möglich.

STATUS PIN BELEGUNG „CF MEMORY MODUL“

Bei den Avisaro Compact Flash Memory Modulen werden zusätzlich zur Schnittstelle einige Statusinformation gegeben. Die Software “Memory Modul” belegt folgende Pins mit den beschriebenen Funktionen:

Pin	MicroMatch	Stiftleiste	I/O	Beschreibung
IO0	M33	P20	I/O	Anzeige: „Betrieb / Power on“ (grüne LED)
IO1	M34	P19	I/O	Anzeige: „Lese – Zugriff“ (grüne LED)
IO2	M35	P18	I/O	Anzeige: „Schreib – Zugriff“ (rote LED)
IO3	M36	P17	I/O	Anzeige: „Error“ (rote LED)
IO4	M37	P16	I/O	Taster: „Operation beenden“ (Mit Taster gegen Masse ziehen)

Die Belegung für die jeweilige Schnittstelle – siehe oben in diesem Dokument.

Alle LEDs werden gegen GND geschaltet.

Lassen Sie alle nicht benötigten Leitung offen. Pull-Up oder Pull-Down Widerstände sind nicht notwendig.

STATUS PIN-BELEGUNG „WLAN“

Im WLAN RS232 Modul kann die Verwendung der I/O Leitungen geändert werden. Zur Auswahl steht:

- “Box”
- “Embedded”
- “I/O”

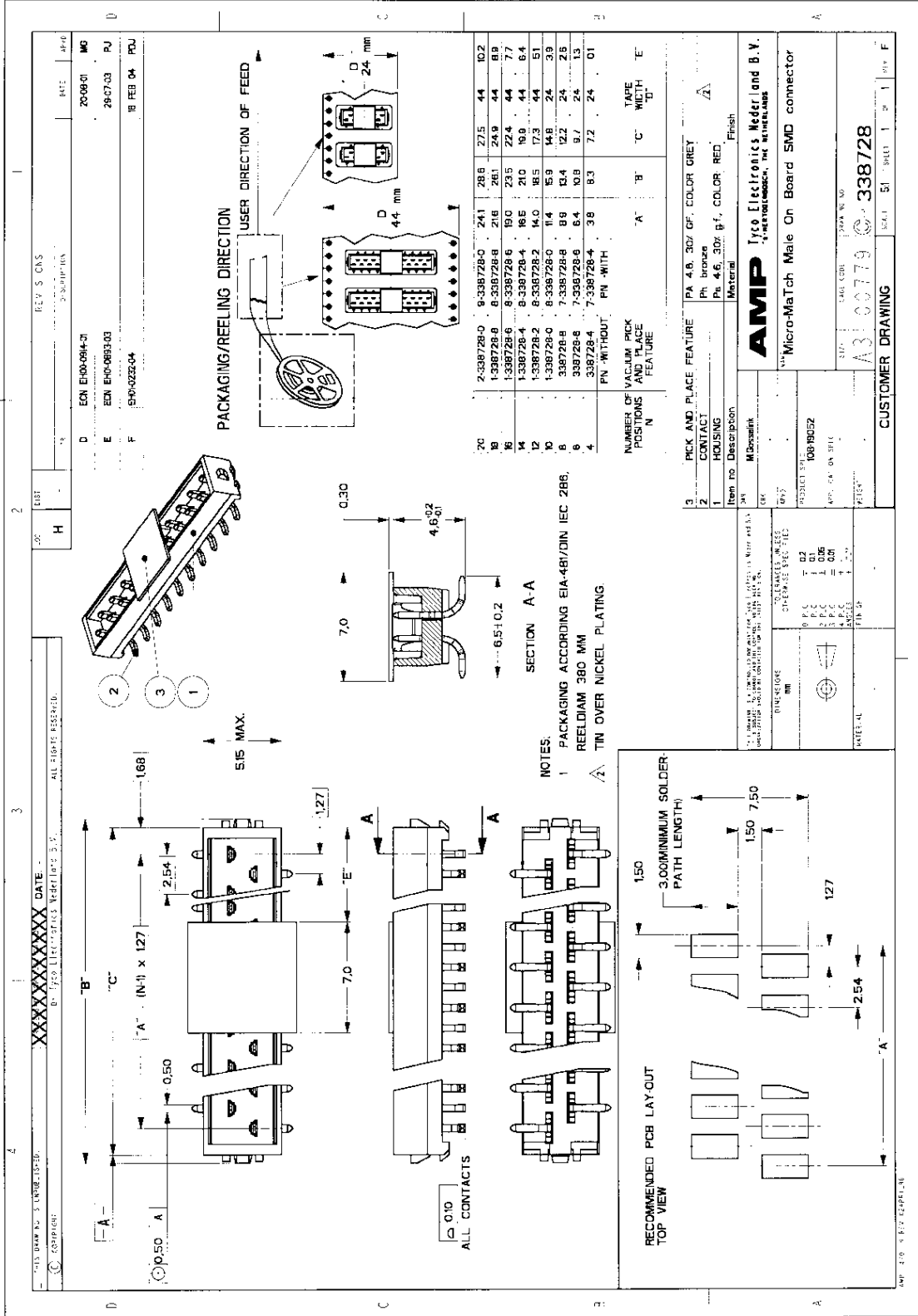
So stehen verschiedene Statusinformation zur Verfügung. Die Software im Modul WLAN – RS232 benutzt folgende Pinbelegung zusätzlich zur Datenschnittstelle:

Pin	MicroMatch	Stiftleiste	I/O	Beschreibung
IO0	M33	P20	O	Box: Verbindung zum Access Point ok Embedded: Verbindung zum Access Point ok (high: Verbindung ok, low: nicht ok) I/O: Benutzerdefiniert
IO1	M34	P19	O	Box: Datenverbindung (TCP) besteht Embedded: Datenverbindung (TCP) besteht (high: Verbindung besteht, low: ~nicht) I/O: Benutzerdefiniert
IO2	M35	P18	I/O	Box: Daten Empfang und Gesendet Embedded: Daten Empfang und Gesendet I/O: Benutzerdefiniert
IO3	M36	P17	I/O	Box: Error Embedded: Error (high: Fehlerfall liegt vor, low: alles ok) I/O: Benutzerdefiniert
IO4	M37	P16	I	Box & Embedded: Verbindung aufbauen / abbauen. Übergang von high zu low: 1) Wenn keine Verbindung besteht (siehe IO1) wird diese aufgebaut gemäß Einstellungen „Verbindung aufbauen zu IP/Port“. 2) Wenn eine Verbindung besteht (siehe IO1) wird diese abgebaut. (Interner Pullup zu high) I/O: Benutzerdefiniert
IO5	M38	P15	I/O	Box & Embedded: keine Bedeutung I/O: Benutzerdefiniert
IO6	M39	P14	I/O	Box & Embedded: keine Bedeutung I/O: Benutzerdefiniert
IO7	M49	P13	I/O	Box: keine Bedeutung Embedded: Reset zu Default-Werten ‚Overwrite‘ ¹⁾ (Interner Pullup zu high) I/O: Benutzerdefiniert
RESET	M2	P3	I	Reset des Mikroprozessors. Aktiv low. (Interner Pullup zu high)

- 1) Normalerweise werden Default-Werte benutzt nachdem das Modul ohne eingelegte Karte eingeschaltet wird. Mit diesem Pin wird diese Eigenschaft abgeschaltet. Dazu wird der PIN schon beim Einschalten des Moduls auf Masse gezogen (z.B. über 2,2kOhm). Nun wird das Modul nicht in den Default Modus geschaltet, auch wenn keine Karte eingelegt ist. Um das Modul in den Default-Zustand zurück zu setzten, wird der PIN bei eingeschaltetem Modul kurz (min. 0,5 sec auf VCC) getoggelt. Beim nächsten Neustart des Moduls werden nun Default-Werte verwendet.
Wird der Pin nicht beschaltet, bleibt das Feature aktiv.

MICROMATCH CONNECTOR (SMD)

Für Ihre Trägerplatte verwenden Sie den MicroMatch 9-338728-0 von AMP-Tyco.



PROGRAMMIERBEISPIELE

I2C SCHNITTSTELLE

Um den I2C Busmaster zu programmieren, kann folgender Code ggf. weiterhelfen:

```
// Diese Funktion sendet Daten an das Avisaro-Modul
// -----
// Input:
// device_address ---> I2C-Adresse des Avisaro-Moduls
// *data          ---> Zeigt auf den Anfang der zu versendenden Daten
// size          ---> Anzahl der Daten die zu versenden sind
void I2C_Send (UINT8 device_address, UINT8 *data, int size)
{
    // Setze START Bedingung
    I2C_start();
    // Sende die Geräteadresse des Moduls
    I2C_out_byte (device_address<<1);
    // Warte Antwort ab
    I2C_wait_ack();
    // Für alle Daten...
    while (size-->0)
    {
        // Sende ein Byte
        out_byte (*data);
        data++;
        // Warte Antwort ab
        I2C_wait_ack();
    }
    // Fertig, setze STOP Bedingung
    I2C_stop();
}

// Diese Funktion empfängt Daten vom Avisaro-Modul
// -----
// Der Benutzer gibt einen Puffer und die Grösse des Puffers an, in den
// die Daten kopiert werden sollen.
// Die Funktion überträgt empfangene Daten bis zur Größe des Puffers
// Input:
// device_address ---> Geräteadresse des Avisaro-Moduls
// *dest          ---> Adresse des Empfangspuffers
// how_much       ---> Maximale Anzahl (Grösse des Empfangspuffers)
// Return:
// Der Rückgabewert ist die Anzahl tatsächlich übertragener Daten
UINT16 I2C_Receive (UINT8 device_address, UINT8 *dest, UINT16 how_much)
{
    UINT16 size;
    UINT16 cnt = 0;

    // Sende START Bedingung
    I2C_start();
    // Sende Geräteadresse mit gesetztem Bit 0 (READ)
    I2C_out_byte (device_address<<1 | 1);
    // Warte Bestätigung ab
    I2C_wait_ack();
    // Hole Anzahl Bytes im Puffer des Moduls (High-Byte)
    size = I2C_in_byte();
    // Sende Bestätigung
    I2C_send_ack();
    // Hole Anzahl Bytes im Puffer des Moduls (Low-Byte) und
    // berechne Gesamtgrösse
    size <<= 8;
    size |= I2C_in_byte();
    // Fehler oder keine Daten im Puffer des Moduls?
    if (size == 0 || size == 0xffff)
        // Ein Takt weiter (ohne Bestätigung)

```

```

    I2C_nop();
// Es sind Daten da !!!
else
{
    // Sende ACK für das 2 Byte der Länge
    I2C_send_ack();
    // Bis entweder alle Daten gelesen wurden oder der Puffer voll ist
    while (size && how_much)
    {
        // Hole ein Byte und übertrage es in den Puffer
        *dest = I2C_in_byte();
        // Ist es das letzte Byte?
        if (size == 1 || how_much == 1)
            // Sende kein ACK
            I2C_nop();
        // Für alle anderen Bytes...
        else
            // Sende ACK
            I2C_send_ack();
        // Alle Zähler und Pointer weiterzählen
        dest++;
        cnt++;
        size--;
        how_much--;
    }
}
// Setze I2C STOP Bedingung
I2C_stop();
// Rückgabe: Anzahl in den Puffer übertragener Zeichen
return cnt;
}

```

SPI SCHNITTSTELLE

Zum Kodieren der Daten, die ein SPI-Master an das AVISARO-Modul sendet, kann folgende C-Funktion eingesetzt werden:

```

// Byte stuffing:
// Encodes raw data stream to byte-stuffed stream
// Input:  in --> The current raw byte
//        out --> Receives a pointer to byte-stuffed data
// Returns:
//        Size of output bytes-1 (either 0 or 1)
int BS_encode (unsigned char in, unsigned char **out)
{
    static unsigned char escaped_dummy[] = {BSB_ESCAPE_1, BSB_ESC_DUMMY};
    static unsigned char escaped_escape[] = {BSB_ESCAPE_1, BSB_ESCAPE_1};
    switch (in)
    {
        case BSB_DUMMY:
            *out = escaped_dummy;
            return 1;

        case BSB_ESCAPE_1:
            *out = escaped_escape;
            return 1;

        default:
            *out = &in;
            return 0;
    }
}

```

Die Funktion wandelt Rohdaten ggf. in 2-Byte Kombinationen um, indem das Dummy-Byte und der Byte Stuffing Header ersetzt werden. Je nachdem, welche Eingabe die Funktion bekommt, gibt sie entweder ein oder zwei Zeichen aus. Zur Ausgabe dient ein Pointer, der von der Funktion verändert wird und der nach der Rückkehr auf die erzeugten Zeichen (Ein Zeichen - Rückgabewert 0, Zwei Zeichen - Rückgabewert 1) zeigt.

Die Steuerung der Flußkontrolle wird nicht von dieser Funktion behandelt, da diese Nachrichten nur bei Bedarf in den Datenstrom eingefügt werden müssen.

Zum Dekodieren des Datenstroms kann der Master etwa so vor sich gehen:

```
// Byte stuffing:
// Decodes a byte-stuffed data stream to raw
// Input:  in  --> Byte from byte-stuffed stream
// Returns:
//         0...255 ---> The decoded byte
//         -1      ---> No output
//         -2      ---> Received XOFF from master
//         -3      ---> Received XON from master
//         -4      ---> Unknown sequence
int BS_decode (unsigned char in)
{
    static int decoderstate = 0;

    // Always return 'no output' on dummy bytes
    // without affecting the state machine
    if (in == BSB_DUMMY)
        return -1;

    // Do the decoder state machine...
    switch (decoderstate)
    {
        // Unescaped state
        case 0:
            // Escape byte?
            if (in == BSB_ESCAPE_1)
            {
                // Switch state and return 'no output'
                decoderstate = 1;
                return -1;
            }
            // Not an escape byte - return the input
            return (int)in;

            // Escaped state
        case 1:
            // Always switch back to unescaped
            decoderstate = 0;
            // Return something depending on input
            switch (in)
            {
                case BSB_ESC_DUMMY:
                    return (int)BSB_DUMMY;

                case BSB_ESCAPE_1:
                    return (int)BSB_ESCAPE_1;

                case BSB_XOFF:
                    return -2;

                case BSB_XON:
                    return -3;

                default:
                    return -4;
            }
        }
    }
}
```

Je nach Eingabe gibt diese Funktion einen positiven Wert (0...255) oder einen negativen Wert (-4...-1) zurück. Bei positiven Werten handelt es sich direkt um das dekodierte Byte. Negative Werte sind Statusinformationen, die z.B. anzeigen ob die Gegenstelle temporär nichts empfangen möchte (XOFF).

Unter Zuhilfenahme der oben beschriebenen Funktionen könnte eine Kommunikation mit dem AVISARO-Modul so aussehen (ohne Berücksichtigung der Flußkontrolle):

```
int encoded_size = 0;
unsigned char *encoded_bytes;
unsigned char decoded_byte;
int num_of_decoded_bytes;

// Aktiviere den das AVISARO-Modul
Pull_Slave_Select_Low();

for (;;)
{
    int txbyte;
    unsigned char rxbyte;

    // Ist noch ein Byte da von vorheriger Kodierung?
    if (encoded_size)
    {
        // Sende zweites Byte von Kodierung
        txbyte = encoded_bytes[1];
        encoded_size = 0;
    }
    // Nein,
    else
    {
        // Hole ein Byte von Datenquelle
        txbyte = Hole_Neues_Byte();
        if (txbyte == KEINE_DATEN)
            // Send Dummy Byte
            txbyte = DUMMY_BYTE;
        else
        {
            encoded_size = BS_encode ((unsigned char)txbyte, &encoded_bytes);
            // Sende erstes Byte von Kodierung
            txbyte = encoded_bytes[0];
        }
    }

    // Senden und Empfangen jeweils 8 Bits gleichzeitig über SPI
    rxbyte = SPIMasterRXTX ((unsigned char)txbyte);

    // Dekodiere empfangene Daten...
    num_of_decoded_bytes = BS_decode(rxbyte, &decoded_byte);
    // ... und gib diese ggf. aus
    if (num_of_decoded_bytes)
        Ausgabe(decoded_byte);
}
}
```

TCP/IP (WIN-SOCKET) PROGRAMMIERUNG

Für die Kommunikation mit dem WLAN Modul können Port-Redirectoren verwendet werden. Diese leiten einen seriellen Anschluss an Ihrem PC auf eine TCP Verbindung um. Alternativ kann auch direkt über TCP kommuniziert werden:

```

/*
   Simple client example using WINSOCK
   -----
   Link with ws2_32.lib (MSVC) or libws2_32.a (GCC/MinGW).

   This program connects to a web server and displays all received bytes
   (including HTTP headers) in the console window.

   You may also visit these sites:
   * http://msdn.microsoft.com/library/default.asp?url=/library/en-
us/winsock/winsock/getting_started_with_winsock.asp
   * http://www.hal-pc.org/~johnnie2/winsock.html
   * http://tangentsoft.net/wskfaq/
   * http://www.vijaymukhi.com/
   * http://cs.ecs.baylor.edu/~donahoo/practical/CSockets/winsock.html
   * http://www.sockets.com/

   Have fun,
   -> peter@avisaro.com
*/

// This is the webserver address
// www.yahoo.akadns.net (yahoo.com)
#define IPADDRESS "216.109.117.204"
// This is the port number (HTTP)
#define PORT 80

// For 'printf' etc.
#include <stdio.h>
// Also include socket definitions
#include <windows.h>

// Macro which prints an error message
// and exits the program
#define ERR(_x){printf("Could not %s\n",_x);Sleep(INFINITE);exit(1);}

// This is the simplest HTTP GET request
#define HTTP_REQUEST "GET / HTTP/1.0\r\n\r\n"

// Let's go
int main()
{
    // Structure gets filled by WSStartup
    WSADATA wsa;
    // Our socket handle
    SOCKET sock;
    // Structure holding the complete network address
    SOCKADDR_IN sockaddr;
    // Temporary return value from socket functions
    int result;

    // Initialize the socket library
    if (0 != WSStartup (0x0202, &wsa))
        ERR ("initialize socket library");

    // Get a socket instance
    sock = socket (AF_INET, SOCK_STREAM, IPPROTO_TCP);
    if (sock == INVALID_SOCKET)
        ERR ("create socket instance");

    // Connect to the remote service
    sockaddr.sin_family = AF_INET;
    sockaddr.sin_port = htons (PORT);
    sockaddr.sin_addr.s_addr = inet_addr (IPADDRESS);
    result = connect (sock, (SOCKADDR*)&sockaddr, sizeof (SOCKADDR_IN));
    if (result == SOCKET_ERROR)
        ERR ("connect");
}

```

```

printf ("Connected to %s\n", IPADDRESS);

// Now send an HTTP request
result = send (sock, HTTP_REQUEST, sizeof(HTTP_REQUEST)-1, 0);
if (result == SOCKET_ERROR)
    ERR ("send HTTP request");

// Display all received data
while (1)
{
    // Receive buffer
    char buff[256];
    // Temporary pointer for displaying received bytes
    char *temp;

    // Try to get some data
    // The return value is the number of received bytes.
    result = recv (sock, buff, 256, 0);
    // Leave the loop if either the connection was closed
    // or there was an error. The web server closes the
    // connection when all data was send.
    if (result == 0 || result == SOCKET_ERROR)
        break;
    // Something has been received - show it now.
    temp = buff;
    while (result--)
        printf ("%c", *temp++);
}
printf ("\nDisconnected\n");

// Finally close the socket
closesocket (sock);

// Ready
return 0;
}

```