HARDWARE DOKUMENTATION

CF Module (WLAN und CF Memory)

Spannungsversorgung, Beschaltung IO-Pins

Schnittstellen: Belegung und Timing (RS232, I2C, Parallel)

Pinbelegung der Module

Abmessungen

Programmierbeispiele

Version 3.1

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

Kontakt:

Wilke Technology GmbH
Krefelder Str. 147
52070 Aachen

Telefon:
0241 / 918 900
Telefax:
0241 / 918 90 44

eMail:
info@wilke.de

INHALTSVERZEICHNIS

Inhaltsverzeichnis 3

Stromversorgung und IO-Pins 4
Stromversorgung 4
IO-Pins 4

Serielle Schnittstelle - RS232 5

Serielle Schnittstelle – I2C (Slave) 6
Einstellungen 6
Einfacher Test 6
Datenaustausch mit dem Modul über I^2C 7
Daten zum Modul senden 7
Daten vom Modul empfangen 7
Source-Code Beispiel 8

Parallele Schnittstelle 9
Source Code 9
Bustiming 9
Master sendet Daten zum Client 10
Client sendet Daten zum Master 10
Master möchte Daten senden, aber Client hat noch Daten 11

Pin Belegung – allgemein 12

Pin Belegung „CF Memory Modul“ 14

Pin-Belegung „WLAN RS232 / I2C“ 15

Pin Layout Stiftleiste 17

Pin Layout Micro Match Stecker 19

Abmessungen 21

MicroMatch Connector (SMD) 22

Programmierbeispiele 23
I2C Schnittstelle 23
Parallele Schnittstelle 25
TCP/IP (Win-Socket) Programmierung 28

STROMVERSORGUNG UND IO-PINS

STROMVERSORGUNG

Das 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. 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 Moduls sind 5V tolerant. D.h. ein Eingang des 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 ausreichend dokumentiert, sodass hier nicht das Timing wiederholt wird. Das 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	OST 6	OST 4	I	RS232 – RX
TXD0	OST 7	OST 5	O	RS232 – TX
SCL/CTS	OST 8	OST 8	O	CTS Flusskontrolle '1' = Stop Sende '0' = Ok zum Senden
SDA/RTS	OST 9	OST 9	I	RTS Flusskontrolle '1' = Stop Sende '0' = Ok zum Senden

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

SERIELLE SCHNITTSTELLE – I2C (SLAVE)

Die I^2C-Bus Implementation im Modul entspricht einem I^2C-Bus Slave d.h. das Modul kann nicht von sich aus kommunizieren und muß deshalb von einem Master-Device ständig gepollt werden. Es wird das I^2C-Bus Modul des Controllers verwendet. Die Taktfrequenz ist Standard-I^2C (100 kHz).

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

Pin	MicroMatch	Stiftleiste	I/O	Beschreibung
SCL/CTS	OST 8	OST 8	I	SCL Leitung (I2C Clock)
SDA/RTS	OST 9	OST 9	I/O	SDA Leitung (I2C Daten – In und Out)

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 Modul kennt keine 'Reservierten Adressen' (siehe I2C-Bus Specification, Seite 16), sondern reagiert nur auf seine eigene (Slave) Adresse.

Ein jungfräuliches Modul hat die 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 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 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 MODUL ÜBER I²C

Das Modul hat jeweils einen Eingangs- und Ausgangspuffer (FIFO). Der Eingangspuffer wird von einem externen I2C -Gerät gefüllt und vom Modul sehr schnell verarbeitet. Der Ausgangspuffer enthält Daten vom 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 MODUL SENDEN

Daten werden als "Stream" zum Modul gesendet. Der Ablauf aus Sicht des Busmasters ist folgender:

1. Setze die I2C START Bedingung.
2. Sende die I2C Adresse des 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 Modul (Bit 9).
4. Sende ein Byte der Daten (8 Bits).
5. Warte auf ACK vom Modul (Bit 9).
6. Gehe zu Schritt 4 bis alle Daten verschickt wurden.
7. Setze die I2C STOP Bedingung.

DATEN VOM 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 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 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 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 Modul (Bit 9).
4) Lese das erste Byte aus dem Modul (8 Bits).
5) Setze ACK (Bit 9).
6) Lese das zweite Byte aus dem Modul (8 Bits).
7) Verknüpfe beide Bytes zu einem 16-Bit Wert, Anzahl = Byte1<<8 | 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 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 Modul (8 Bits),
14) Setze kein ACK (Bit 9).
15) Setze die I2C STOP Bedingung.

SOURCE-CODE BEISPIEL

Siehe Kapitel „I2C Schnittstelle“ auf Seite 23.

PARALLELE SCHNITTSTELLE

Der Parallel-Bus besteht aus 8 Datenleitungen und 4 Steuerleitungen und benutzt die ganze “WEST”-Seite des Moduls. Die Datenrichtung der Steuerleitungen ist festgelegt, aber die Datenrichtung der Datenleitungen ist änderbar um einen abwechselnden Datenaustausch zu erreichen (Bidirektional, Halbduplex). Der Bus arbeitet asynchron – eine Clock-Leitung gibt es daher nicht. Alle Operationen werden durch „Valid“ und „Ack“ angekündigt bzw. bestätigt. So kann der Bus auch mit sehr langsamen Clients arbeiten.

Pin	MicroMatch	Stiftleiste	I/O	Beschreibung
ATD0	WEST 1	WEST 1	I	Client Bus Request
ATD1	WEST 2	WEST 2	I	Client Data Valid / Data Acknowledge
DAC0	WEST 3	WEST 3	O	Master Bus Acknowledge
DAC1	WEST 4	WEST 4	O	Master Data Valid / Data Acknowledge
IO0	WEST 13	WEST 5	I/O	Bidirectional Data 0
IO1	WEST 14	WEST 6	I/O	Bidirectional Data 1
IO2	WEST 15	WEST 7	I/O	Bidirectional Data 2
IO3	WEST 16	WEST 8	I/O	Bidirectional Data 3
IO4	WEST 17	WEST 9	I/O	Bidirectional Data 4
IO5	WEST 18	WEST 10	I/O	Bidirectional Data 5
IO6	WEST 19	WEST 11	I/O	Bidirectional Data 6
IO7	WEST 20	WEST 12	I/O	Bidirectional Data 7

SOURCE CODE

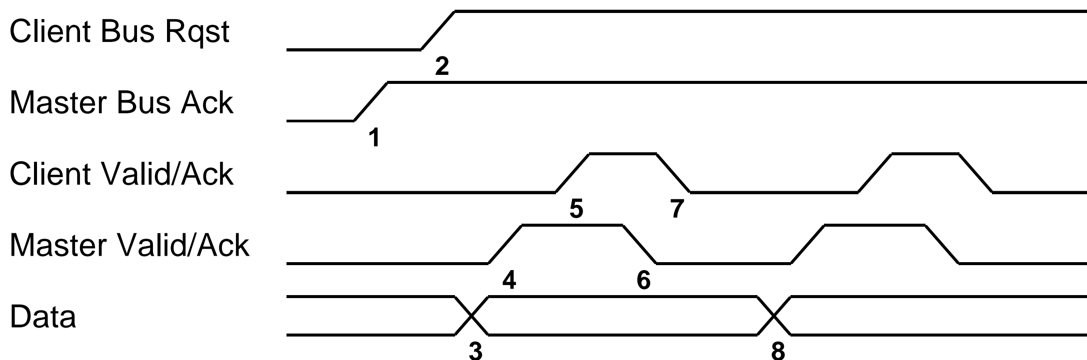
Siehe Kapitel „Parallele Schnittstelle“ auf Seite 25.

BUSTIMING

Bei dem Bustiming gibt es zwei Mechanismen: Die Richtung des Datenbusses muss ausgehandelt werden und die Daten müssen angekündigt und bestätigt werden. Die folgenden Diagramme beschreiben die verschiedenen Szenarien. Kollisionen am Bus sind nicht möglich, wenn sich Client und Master richtig verhalten.

MASTER SENDET DATEN ZUM CLIENT

1) Der Master fordert den Bus an, indem die Bus Ack Leitung auf High gesetzt wird.
2) Der Client bestätigt die Busanforderung indem die Bus Rqst Leitung auf High gesetzt wird. Nur wenn sowohl Bus Ack als auch Bus Rqst Leitung auf High sind, kann der Master zum Client Daten senden. So lange der Client die Busumschaltung nicht bestätigt, kann er weiter Daten zum Master senden. Der Master ist also abhängig von der Cooperation des Clients – dies macht jedoch Sinn, da so der Client z.B. ein Datenpacket fertig übertragen kann.
3) Der Master legt Daten an den Bus
4) Der Master setzt Data Valid
5) Der Client bestätigt die Daten
6) Der Master setzt Data Valid zurück
7) Der Client setzt Data Valid zurück
8) Der Master legt neue Daten auf den Bus und der Vorgang beginnt neu. Der Bus bleibt die ganze Zeit im selben Zustand.

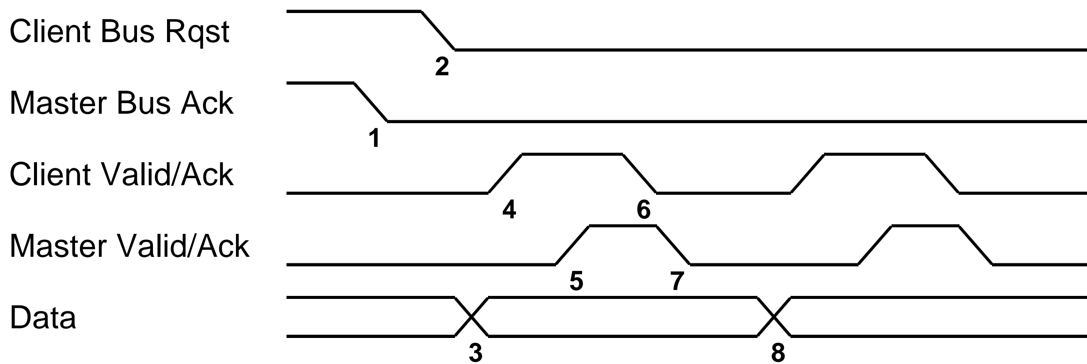


CLIENT SENDET DATEN ZUM MASTER

1) Der Master gibt den Bus frei, dass der Client senden kann, indem die Bus Leitung auf Null gesetzt wird. Der Client muss also so lange mit dem Senden warten.
2) Der Client bestätigt die Umschaltung.
3) Der Client legt Daten an den Bus an.
4) Der Client setzt Data Valid um die Gültigkeit der Daten zu signalisieren
5) Der Master quitiert mit Data Ack
6) Der Client setzt Data Valid wieder zurück

7) Der Master setzt Data Ack zurück

8) Der Client beginnt mit einer neuen Datenübertragung. Der Bus bleibt die ganze Zeit in die selbe Richtung geschaltet.



MASTER MÖCHTE DATEN SENDEN, ABER CLIENT HAT NOCH DATEN

1) Eine Datenübertragung von Client an Master ist im Gang (Bus Ack und Bus Request Leitung auf High). Master fordert den Bus an um Daten zum Client zu senden.

2) Der Client hat noch Daten und möchte diese an den Master senden. Der Client legt die Daten an den Bus.

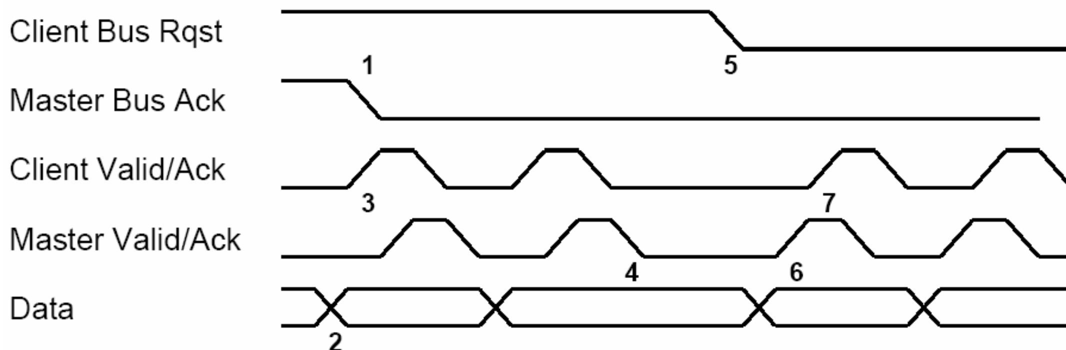
3) Der Client fährt mit der Datenübertragung fort und validiert die Daten per Data Valid. Der Master akzeptiert die Daten und quittiert.

4) Der Client ist mit der Datenübertragung fertig.

5) Der Client bestätigt die Busumschaltung

6) Der Master legt Daten an den Bus und setzt das Valid Signal

7) Der Client empfängt die Daten und bestätigt.



PIN BELEGUNG – ALLGEMEIN

Das Modul hat zwei Stecksysteme – ein Micro Match Stecker und eine Lochleiste zur Aufnahme von z.B. Stiftleisten. Der MicroMatch bietet etwas mehr Funktionen aufgrund der höheren Polzahl.

Die Signale und Stromversorgungsleitung der beiden Stecker sind direkt miteinander verbunden. Haben Sie also z.B. auf der Stiftleiste 3,3 V angelegt, benötigen Sie dies nicht noch einmal auf dem MicroMatch Stecker zu tun.

Mit ‚West' und ‚Ost' sind die beiden Seiten bezeichnet. Auf den unten gezeigten Fotos ist die Bezeichnung herausgestellt.

Pin	MicroMatch	Stiftleiste	I/O	Beschreibung
ATD0	WEST 1	WEST 1	I	Analog zu Digital – Kanal 0
ATD1	WEST 2	WEST 2	I	Analog zu Digital – Kanal 1
DAC0	WEST 3	WEST 3	O	Digital zu Analog – Kanal 0
DAC1	WEST 4	WEST 4	O	Digital zu Analog – Kanal 1
NC2	WEST 5	-	-	Nicht belegt 2
RXD1	WEST 6	-	I	RS232 Kanal 1 (sekundär) - Empfang
TXD1	WEST 7	-	O	RS232 Kanal 1 (sekundär) – Senden
PWM0	WEST 8	OST 11	I/O	Puls Weiten Modulator 0 / GIO
PWM1	WEST 9	OST 12	I/O	Puls Weiten Modulator 1 / GIO
GND	WEST 10	OST 7	PWR	Versorgung: Masse
VCC	WEST 11	OST 6	PWR	Versorgung: 3,3 V
NC3	WEST 12	-	-	Nicht belegt 3
IO0	WEST 13	WEST 5	I/O	Benutzer Digital IO 0
IO1	WEST 14	WEST 6	I/O	Benutzer Digital IO 1
IO2	WEST 15	WEST 7	I/O	Benutzer Digital IO 2
IO3	WEST 16	WEST 8	I/O	Benutzer Digital IO 3
IO4	WEST 17	WEST 9	I/O	Benutzer Digital IO 4
IO5	WEST 18	WEST 10	I/O	Benutzer Digital IO 5
IO6	WEST 19	WEST 11	I/O	Benutzer Digital IO 6

IO7	WEST 20	WEST 12	I/O	Benutzer Digital IO 7
BKGD	OST 1		-	Debug / Programmier Modus
RESET	OST 2	OST 3	I	Reset (Microprozessor)
MODA	OST 3		-	Debug / Programmier Modus
MODB	OST 4		-	Debug / Programmier Modus
ECLK	OST 5		-	Debug / Programmier Modus
RXD0	OST 6	OST 4	I	RS232 Kanal 0 (primär) – Empfangen
TXD0	OST 7	OST 5	O	RS232 Kanal 0 (primär) – Senden
SCL/CTS	OST 8	OST 8	O	I2C Bus – SCL Leitung | CTS Flusskontr.
SDA/RTS	OST 9	OST 9	I	I2C Bus – SDA Leitung | RTS Flusskontr.
GND	OST 10	OST 7	PWR	Versorgung: Masse
VCC	OST 11	OST 6	PWR	Versorgung: 3,3 V
CS_EXT	OST 12		O	Chip Select – externe Komponente
NC0	OST 13		-	Nicht belegt 0
NC1	OST 14		-	Nicht belegt 1
GIO_0	OST 15	OST 10	I/O	Benutzer Digital IO – General Purpose
WE#	OST 16		O	Write Enable – externe Komponente
ADDR0	OST 17		O	Adressbus 0 – externe Komponente
ADDR1	OST 18		O	Adressbus 1 – externe Komponente
ADDR2	OST 19		O	Adressbus 2 – externe Komponente
ADDR3	OST 20		O	Adressbus 3 – externe Komponente

PIN BELEGUNG „CF MEMORY MODUL“

Zusätzlich zur Schnittstelle werden einige Statusinformation gegeben. Diese Statusinformationen sind nur bei den Modulen ‚RS232’ und ‚I2C’ gültig, da der parallele Port die selben Pins benutzt. Die Software “Memory Modul” belegt folgende Pins mit den beschriebenen Funktionen:

Pin	MicroMatch	Stiftleiste	I/O	Beschreibung
GND	WEST 10	OST 7	PWR	Versorgung: Masse
VCC	WEST 11	OST 6	PWR	Versorgung: 3,3 V
IO0	WEST 13	WEST 5	O	Anzeige: „Betrieb / Power on“ (grüne LED)
IO1	WEST 14	WEST 6	O	Anzeige: „Lese – Zugriff“ (grüne LED)
IO2	WEST 15	WEST 7	O	Anzeige: „Schreib – Zugriff“ (rote LED)
IO3	WEST 16	WEST 8	O	Anzeige: „Error“ (rote LED)
IO4	WEST 17	WEST 9	I	Taster: „Operation beenden“
GND	OST 10	OST 7	PWR	Versorgung: Masse
VCC	OST 11	OST 6	PWR	Versorgung: 3,3 V

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.

PIN-BELEGUNG „WLAN RS232 / I2C“

Im WLAN RS232 Modul kann die Verwendung der I/O Leitungen geändert werden. Zur Auswahl steht: ‚Box’, ‚Embedded’ und ‚I/O’. So stehen verschiedene Statusinformation zur Verfügung. Die parallele Schnittstelle verwendet die selben IO Leitungen – daher stehen in diesem Fall keine Statusinformationen zur Verfügung. Die Software im Modul WLAN – RS232 benutzt folgende Pinbelegung zusätzlich zur Datenschnittstelle

Pin	MicroMatch	Stiftleiste	I/O	Beschreibung
GND	WEST 10	OST 7	PWR	Versorgung – Masse
VCC	WEST 11	OST 6	PWR	Versorgung: 3,3 V
GND	OST 10	OST 7	PWR	Versorgung - Masse
VCC	OST 11	OST 6	PWR	Versorgung: 3,3 V
RESET	OST 2	OST 3	I	Reset des Mikroprozessors. Aktiv low. (Interner Pullup zu high)
IO0	WEST 13	WEST 5	O	Box: Verbindung zum Access Point ok Embedded: Verbindung zum Access Point ok (high: Verbindung ok, low: nicht ok) I/O: Benutzerdefiniert
IO1	WEST 14	WEST 6	O	Box: Datenverbindung (TCP) besteht Embedded: Datenverbindung (TCP) besteht (high: Verbindung besteht, low: ~nicht) I/O: Benutzerdefiniert
IO2	WEST 15	WEST 7		Box: Daten Empfang und Gesendet Embedded: Daten Empfang und Gesendet I/O: Benutzerdefiniert
IO3	WEST 16	WEST 8		Box: Error Embedded: Error (high: Fehlerfall liegt vor, low: alles ok) I/O: Benutzerdefiniert
IO4	WEST 17	WEST 9	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)

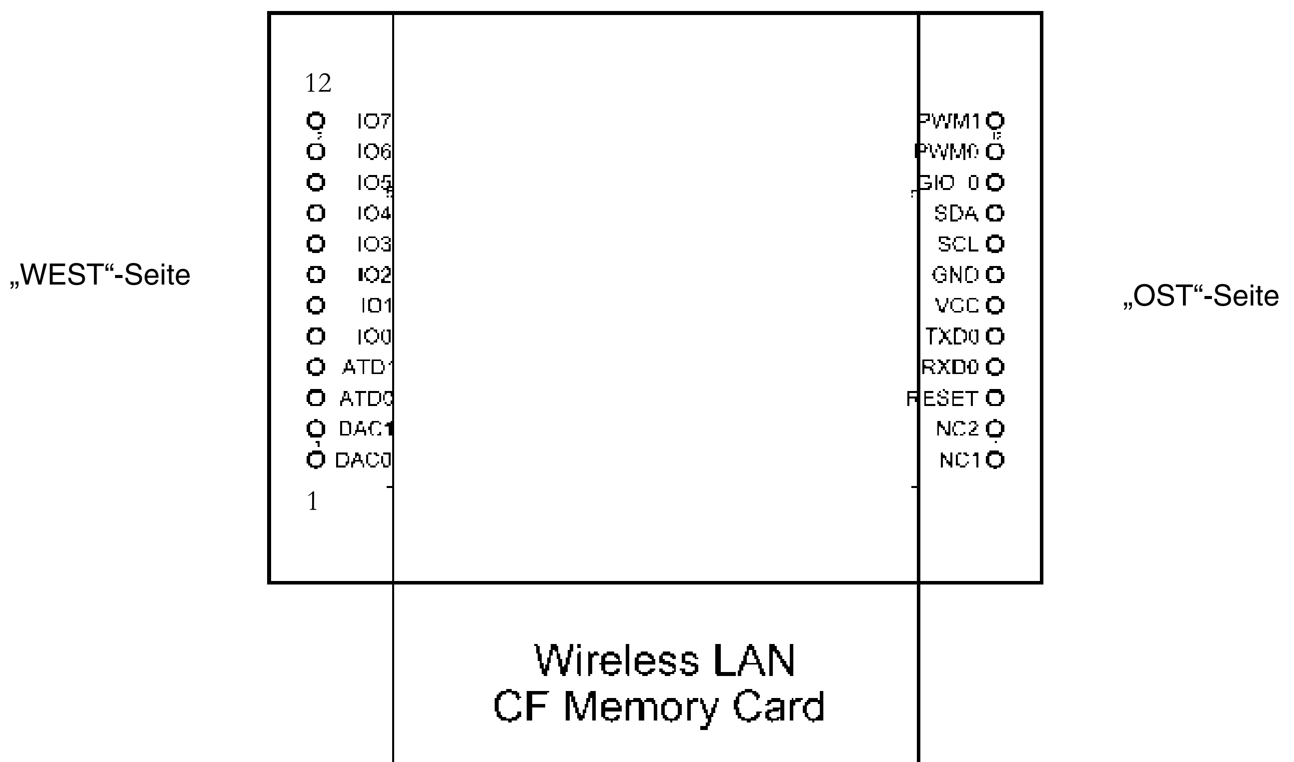
				I/O: Benutzerdefiniert
IO5	WEST 18	WEST 10		Box & Embedded: keine Bedeutung I/O: Benutzerdefiniert
IO6	WEST 19	WEST 11		Box & Embedded: keine Bedeutung I/O: Benutzerdefiniert
IO7	WEST 20	WEST 12	I	Box: keine Bedeutung Embedded: Reset zu Default-Werten ‚Overwrite' [1)] (Interner Pullup zu high) I/O: Benutzerdefiniert

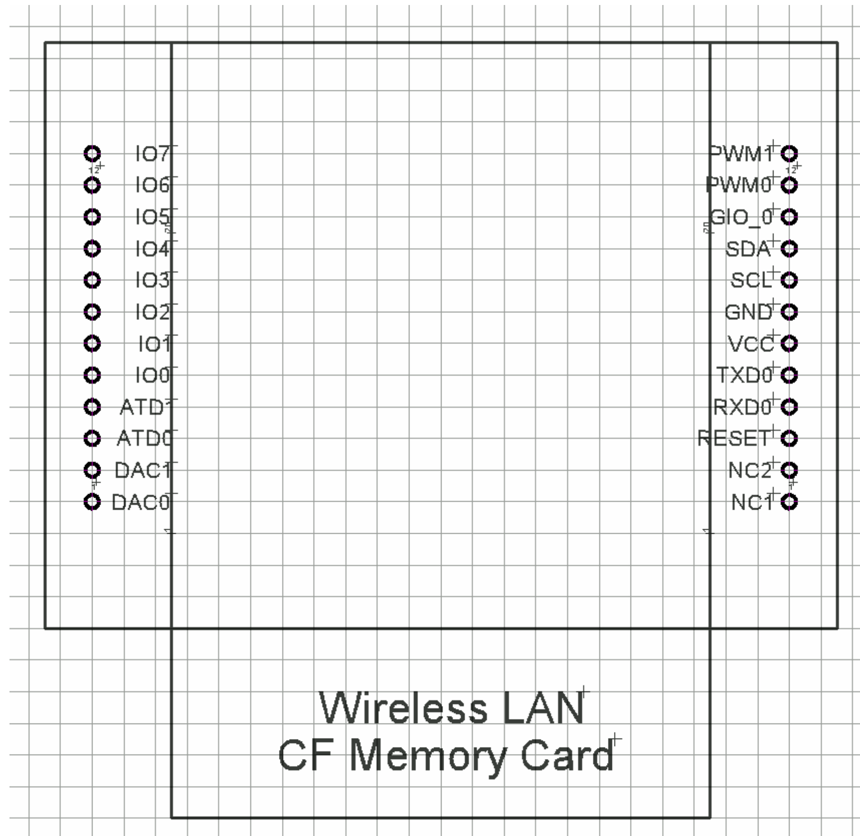
1) Normalerweise werden Default-Werte benutzt nachdem das Modul ohne eingelegte Karte eingeschaltet wird. Mit diesem Pin wird diese Eigenschaft abgeschaltet. Dazu wird der PIN schon beim Einschalten des Moduls auf Masse gezogen (z.B. über 2,2kOhm). Nun wird das Modul nicht in den Default Modus geschaltet, 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.

PIN LAYOUT STIFTLEISTE

Das Modul verfügt über Stiftleisten im 2.54mm Rastermaß. So können Sie ohne Spezialstecker das Modul in Ihre Anwendung integrieren – oder einfach auch Drähte anlöten, um schnell einen Testaufbau zu realisieren.

Die Zeichnungen zeigen das Modul in der Aufsicht. Das Foto dient zu Ihrer Orientierung – es ist in der gleichen Perspektive wie die Zeichnungen aufgenommen. In der dritten Zeichnung ist das Modul mit einem 2.54 Rastermaster abgebildet – dies hilft Ihnen ggf. beim Ausrichten eines Steckers auf einer Trägerplatine.

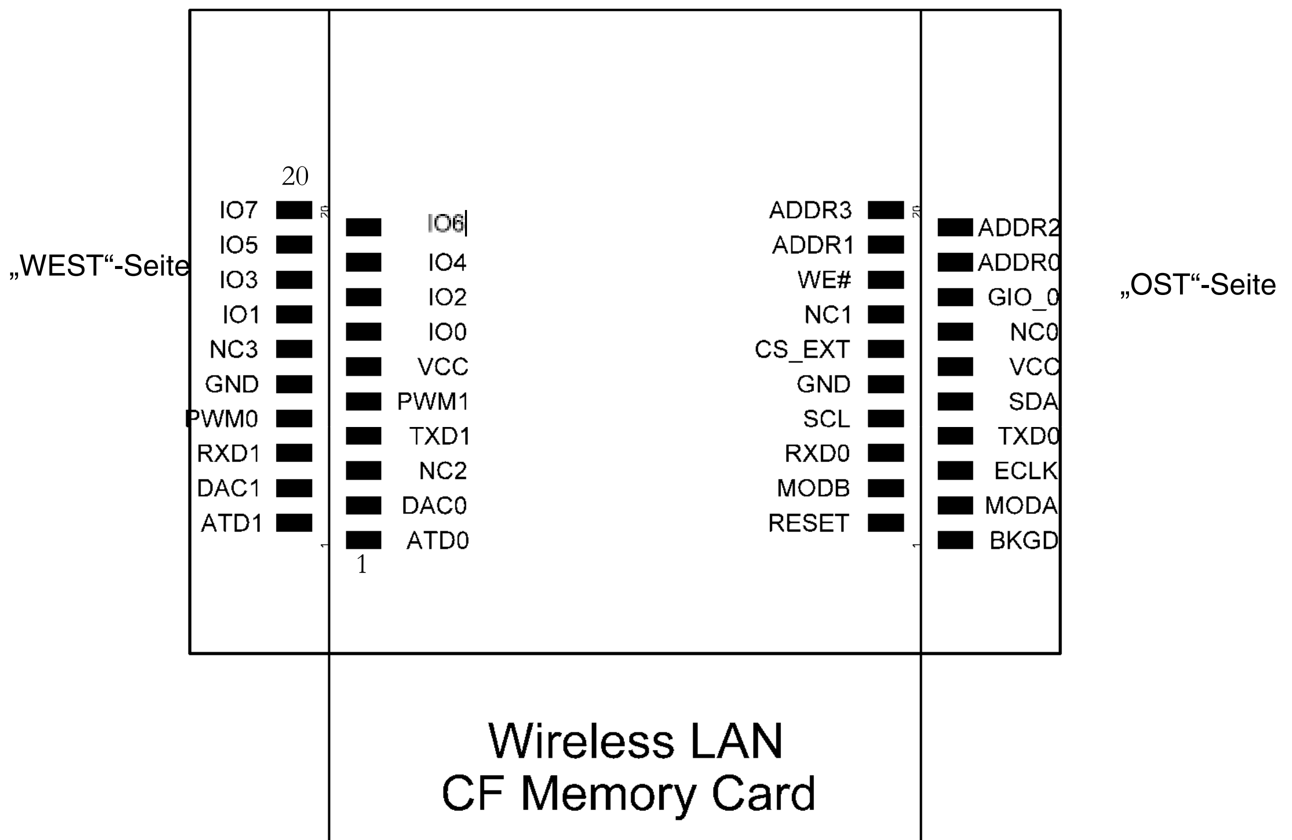


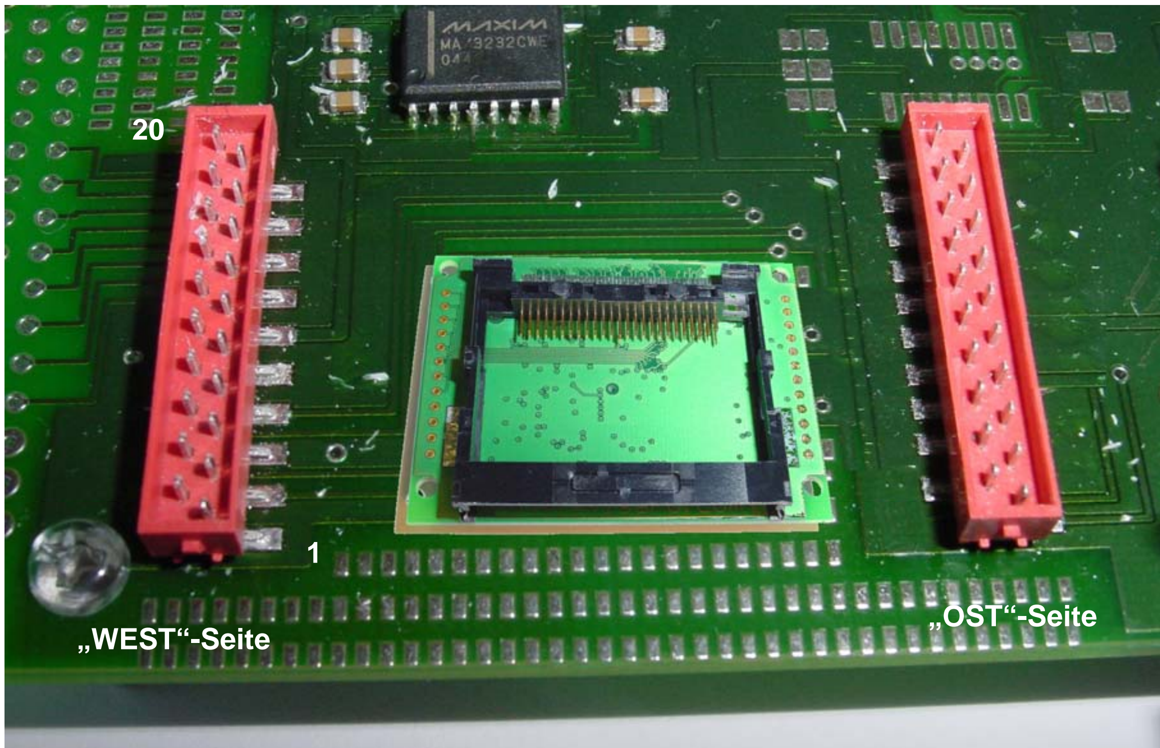

IO7
IO6
IO5
IO4
IO3
IO2
IO1
IO0
ATD1
ATD0
DAC1
DAC0
PWM1
PWM0
GIO_0
SDA
SCL
GND
VCC
TXD0
RXD0
RESET
NC2
NC1
Wireless LAN
CF Memory Card

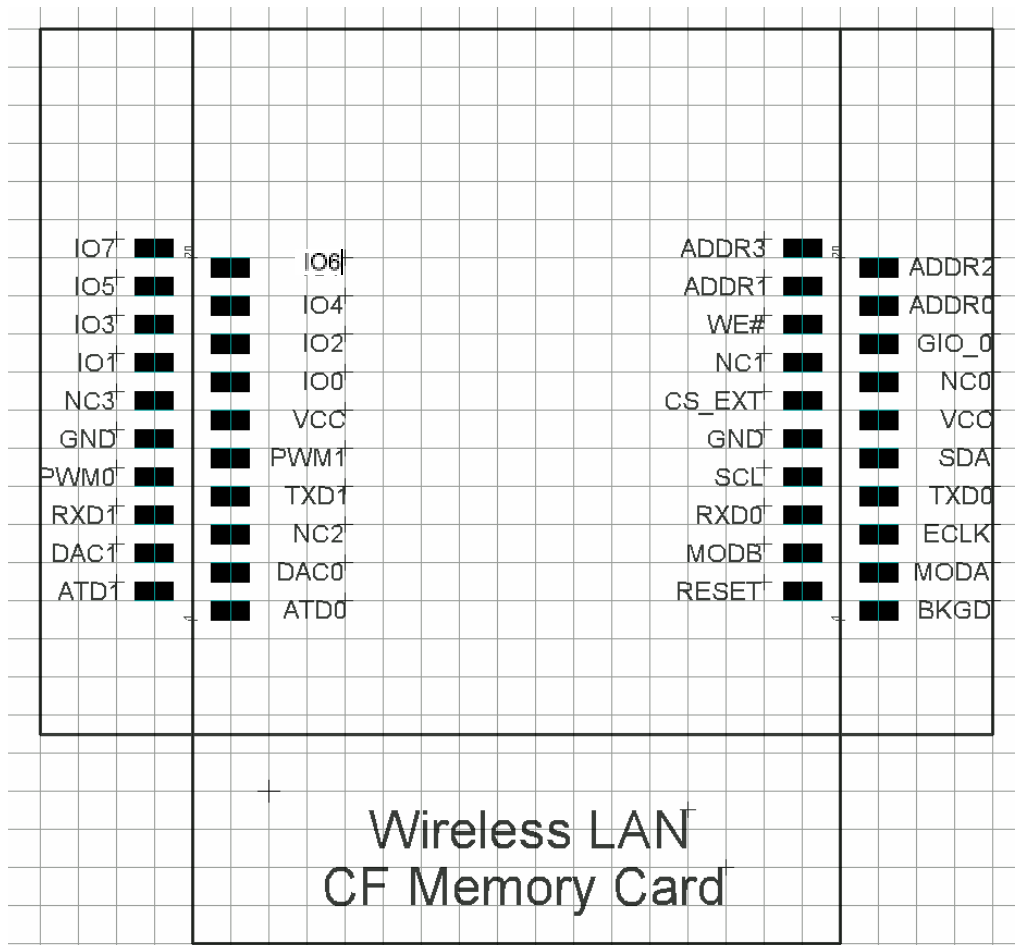
PIN LAYOUT MICRO MATCH STECKER

Das Modul verfügt über ein MicroMatch Stecksystem. Dieses Stecksystem ermöglicht Ihnen auf einer Trägerplatine SMD bestückbare Sockelsteckter zu verwenden, oder auch Flachbandkabel mit aufgequetschtem Stecker zu verwenden – für fast alle Anwendungen finden Sie den richtigen Stecker aus dem MicroMatch System.

Die Zeichnungen zeigen das Modul in der Aufsicht. Das Foto einer Trägerplatine (mit kleinem Modul als Fotomontage) dient zu Ihrer Orientierung – es ist in der gleichen Perspektive wie die Zeichnungen aufgenommen. In der dritten Zeichnung ist das Modul mit einem 2.54 Rastermaster abgebildet – dies hilft Ihnen ggf. beim Ausrichten eines Steckers auf einer Trägerplatine.



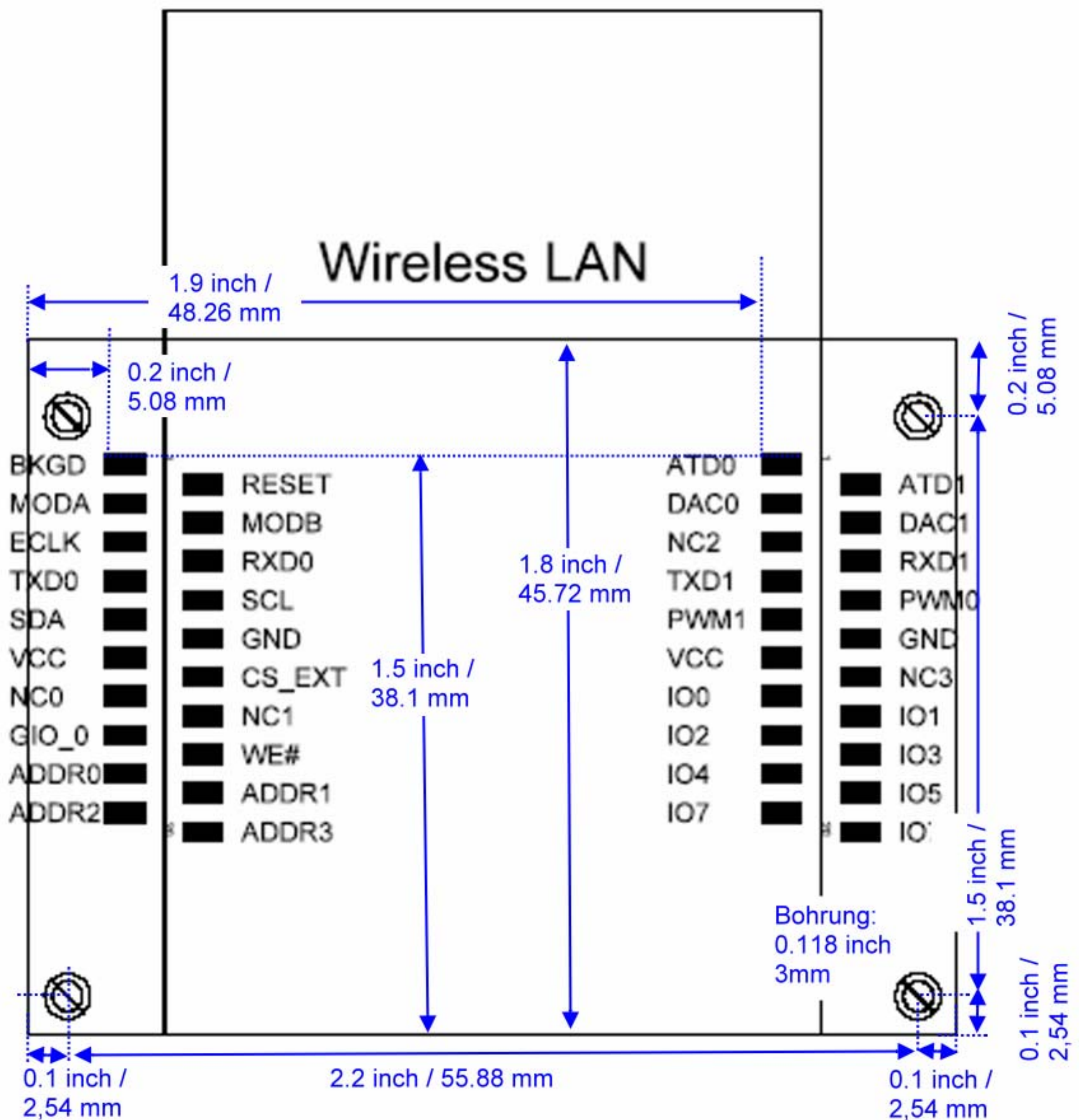

20
1
„WEST“-Seite
„OST“-Seite


IO7
IO5
IO3
IO1
NC3
GND
PWM0
RXD1
DAC1
ATD1
IO6
IO4
IO2
IO0
VCC
PWM1
TXD1
NC2
DAC0
ATD0
ADDR3
ADDR1
WE#
NC1
CS_EXT
GND
SCL
RXD0
MODB
RESET
ADDR2
ADDR0
GIO_0
NC0
VCC
SDA
TXD0
ECLK
MODA
BKGD
Wireless LAN
CF Memory Card

ABMESSUNGEN

Die Abmessungen für den Steckplatz für das Modul sind in der folgenden Darstellung dargestellt.

Die MikroMatch Stecker haben 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.



MICROMATCH CONNECTOR (SMD)

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

PROGRAMMIERBEISPIELE

I2C SCHNITTSTELLE

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

```
// Diese Funktion sendet Daten an das Modul
// ----------------------------------------------
// Input:
// device_address  ---> I2C-Adresse des 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--)
  {
    // Sende ein Byte
    out_byte (*data);
    data++;
    // Warte Antwort ab
    I2C_wait_ack();
  }
  // Fertig, setze STOP Bedingung
  I2C_stop();
}



// Diese Funktion empfängt Daten vom 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 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;
}
```

PARALLELE SCHNITTSTELLE

Der folgende Code ist ein Beispiel für eine Client Implementierung der Parallelen Schnittstelle

```
// Diese Funktion sendet ein Byte an das Modul
// ------------------------------------------------
// Input:
//          b ---> Das zu versendende Byte
// Return:
//          TRUE, wenn es geklappt hat, sonst FALSE
BOOL tester_tx (UINT8 b)
{
  unsigned long timeout;

  // Ist der Bus frei?
  if (DATA_ACK_NOT_SET)
  {
    // Daten auf den Bus legen
    WRITE_DATA_LINES(b);
    // Dem Modul signalisieren dass Daten da sind
    DATA_VALID_ON;
    // Auf Bestätigung warten
    timeout = PPR_GET_COUNTER_VALUE() + 2000;
    while (DATA_ACK_NOT_SET)
    {
      if (timeout < PPR_GET_COUNTER_VALUE())
      {
        // Timeout, Fehler
        DATA_VALID_OFF;
        return FALSE;
      }
    }
    // OK, Data-Valid wieder zurücknehmen
    DATA_VALID_OFF;
    // Darauf warten, dass Modul das ACK zurücknimmt
    timeout = PPR_GET_COUNTER_VALUE() + 2000;
    while (DATA_ACK_SET)
    {
      if (timeout < PPR_GET_COUNTER_VALUE())
        // Timeout, Fehler
        return FALSE;
    }
    // OK, Byte versendet
    return TRUE;
  }
  // Bus nicht frei, Fehler
  return FALSE;
}
```

```
// Diese Funktion empfängt ein Byte vom Modul
// -----------------------------------------------
```

```
// Input:
//     *b ---> Zeigt auf eine Adresse, in der das empfangene Byte gespeichert wird
// Return:
//     TRUE wenn alles geklappt hat.
//     FALSE bei Fehler oder wenn keine Daten da waren.
BOOL tester_rx (UINT8 *b)
{
  unsigned long timeout;

  // Sind Daten da?
  if (DATA_ACK_SET)
  {
    // Ja, Datenwort lesen und speichern
    *b = READ_DATA_LINES;
    // Bestätigen
    DATA_VALID_ON;
    // Warten, bis das Modul das ACK zurücknimmt
    timeout = PPR_GET_COUNTER_VALUE() + 2000;
    while (DATA_ACK_SET)
    {
      if (timeout < PPR_GET_COUNTER_VALUE())
      {
        // Timeout, Fehler
        DATA_VALID_OFF;
        return FALSE;
      }
    }
    // Bestätigung wieder zurücknehmen
    DATA_VALID_OFF;
    // Alles OK
    return TRUE;
  }
  // Keine Daten vorhanden
  return FALSE;
}
```

```
// Globale Variable speichert die Busrichtung
// 0 == Undefiniert
// 1 == Client empfängt
// 2 == Client darf senden
int direction = 0;
```

```
// Steuert die Client-seitige Busumschaltung
void tester_tick (void)
{
   // Master möchte senden, Client noch nicht auf Empfang
   if (BUS_ACK_IS_TX && direction != 1)
   {
     // Neue Richtung merken
     direction = 1;
     // Client Datenrichtung auf Empfang umschalten
     BUS_DIRECTION_RX;
     // Beim Master bestätigen
     BUS_REQUEST_TX;
   }
```

```
    // Master möchte empfangen, Client noch nicht auf Sendung
    else if (BUS_ACK_IS_RX && direction != 2)
    {
      // Neue Richtung merken
      tdir = 2;
      // Client Datenrichtung auf Empfang umschalten
      BUS_DIRECTION_TX;
      // Beim Master bestätigen
      BUS_REQUEST_RX;
    }
}
```

```
// Hauptprogramm
// Simuliert einen Parallel Bus Client auf einem Microcontroller
int main (void)
{
    // I/O-Ports initialisieren
    tester_init();

    // Endlosschleife
    while (1)
    {
      // Auf Busumschaltung reagieren
      tester_tick();

      if (direction == 0)
      {
        // Undefiniert, Startbedingung
      }

      // Der Client darf Empfangen
      if (direction == 1)
      {
        UINT8 b;
        if (TRUE == tester_rx(&b))
        {
            // Empfangene Daten verarbeiten
            // .....
        }
      }

      // Der Client darf Senden
      if (direction == 1)
      {
        UINT8 b;
        // Wenn Daten zum Versenden da sind dann jetzt senden
        if (get_data_for_tx(&b))
          tester_tx (b);
      }
    }
}
```

TCP/IP (WIN-SOCKET) PROGRAMMIERUNG

```
/*
    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,
      -> support@wilke.de
*/

// 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 WSAStartup
  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 != WSAStartup (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;
    }
```