

CAN

Der Device-Treiber 'CAN1_xx.TDD' unterstützt die interne CAN-Schnittstelle des BASIC-Tiger®-CAN-Moduls.

In diesem Abschnitt finden Sie:

- Beschreibung des Device-Treibers CAN1_xx.TDD
- CAN-Botschaften in den I/O-Puffern des Treibers
- CAN User-Function-Codes
- Bus-Timing und Übertragungsrate
- Error-Register
- Empfangsfilter mit Code und Mask
- Versenden von CAN-Botschaften
- Empfangen von CAN-Botschaften
- Ein- und Ausgangspuffer
- Automatische Bitratenerkennung
- CAN-Bus Hardware-Anschlußbeispiel
- Eine kurze Einführung zu CAN
- Besonderheiten des BASIC-Tiger®-CAN-Moduls
- Literaturhinweise zu CAN
- CAN-Board

2

Beschreibung des Device-Treibers CAN1_xx.TDD

Dieser Device-Treiber ermöglicht in Zusammenhang mit dem BASIC-Tiger®-CAN-Modul die Ein- und Ausgabe auf dem CAN-Bus. Bei der Installation des Treibers können die Parameter der CAN-Schnittstelle festgelegt werden. Einige Parameter sind durch Kommandos an den Treiber auch zur Laufzeit veränderbar.

Dateinamen: CAN1_K8.TDD (mit 8K Puffern)
 CAN1_K1.TDD (mit 1K Puffern)
 CAN1_R1.TDD (mit 256 Byte Puffern)

INSTALL DEVICE #D, "CAN1_xx.TDD", "Code, Mask, Bt0, Bt1, Mod, Oc"

D ist eine Konstante, Variable oder ein Ausdruck vom Datentyp BYTE, WORD, LONG im Bereich von 0...63 und steht für die Gerätenummer des Treibers.

Code ist ein Parameter zur Festlegung des Access-Codes. ‚Code‘ ist immer 4 Bytes lang. Der Wertebereich für den Access-Code ist bei Standard-Frames 0...7FFh und bei extended Frames 0...1FFF FFFF.
 Standardwert: 0

Device-Treiber

2

Mask ist ein Parameter zur Festlegung des Akzeptanzfilters. ‚Mask‘ ist immer 4 Bytes lang.
Standardwert: 0FFFFFFFh

Bt0 (Bustiming-Register-0) ist ein Parameter zur Festlegung des Baudraten-Prescalers sowie der Synchronisations-Sprungweite (1 Byte). Zusammen mit Bt1 wird dadurch die Übertragungsgeschwindigkeit festgelegt.
Standardwert: 0

Bt1 (Bustiming-Register-1) ist ein Parameter zur Festlegung des Bus-Timings und der Anzahl der Samples beim Empfang (1 Byte). Zusammen mit Bt0 wird dadurch die Übertragungsgeschwindigkeit festgelegt.
Standardwert: 2Fh (Tseg1=15, Tseg2=2)

Mod ist ein Parameter zur Festlegung des Modus (1 Byte).
Standardwert: 0

Bit	Symbol	wenn Bit gesetzt ('1')
2	CAN_LISTEN	Listen-Only-Mode
3	CAN_SELFTEST	Selftest-Mode
4	CAN_ACC_SINGLE	Single Acceptance-Filter-Mode (32-Bit-Filter)
5	CAN_SLEEP	Sleep-Mode
1,6,7		reserviert

Wird der Listen-Only-Mode installiert, dann versucht der Treiber, die Bitrate auf dem Bus anhand einer Tabelle mit vorgegebenen Bitraten automatisch zu erkennen.

Outctrl ist ein Parameter zur Festlegung der Ausgangsstufe der CAN-Hardware. Standardwert ist 1Ah zum Anschluß eines externen Treiberbausteins.

Beispiele für eine Installation für 500 kBit:

```
install_device #CAN, "CAN1_K1.TDD", &  
0,0,0,0, & ' access code  
0ffh,0ffh,0ffh,0ffh, & ' access mask  
0,2Fh, & ' bustim1, bustim2  
0,1Ah ' mode, outctrl
```

CAN-Botschaften in den I/O-Puffern des Treibers

In den Eingangs und Ausgangspuffern des Tiger-BASIC[®]-CAN-Device-Treibers befinden sich immer vollständige CAN-Botschaften und keine weiteren Bytes. Eine CAN-Botschaft beginnt mit dem Frame-Info-Byte, welches bestimmt, ob es sich um eine Botschaft mit 11- oder 29-Bit-Identifizierer handelt und wieviele Datenbytes enthalten sind. Außerdem befindet sich das RTR-Bit im Frame-Info-Byte. Je nach Frametyp folgen 3 Identifizierer-Bytes (Standardframe) oder 5 Identifizierer-Bytes (extended Frame) und anschließend die Datenbytes. Eine CAN-Botschaft kann 0...8 Bytes als Nutzdaten übertragen.

Im Frame-Info-Byte befindet sich die Information über

- den Frametyp (11 oder 29 ID-Bits)
- die Anzahl der Datenbytes (0...8)
- ob es sich um ein Remote-Transmit-Request handelt

Der Identifizierer kann

- 29 Bits lang sein und belegt dann 4 Bytes im Puffer
- 11 Bits lang sein und belegt dann 2 Bytes im Puffer

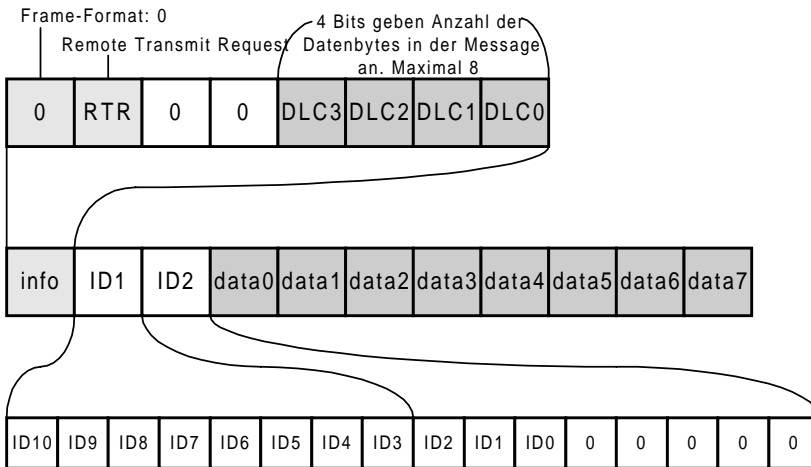
Ein Standardframe belegt maximal 11 Bytes, ein extended Frame maximal 13 Bytes im Puffer. Wenn der Device-Treiber beim Empfang nicht mindestens noch 13 Bytes im Puffer frei hat, dann wird die Botschaft verworfen und ein Fehler ‚Pufferüberlauf‘ registriert. Je nach Länge der einzelnen empfangenen CAN-Nachrichten passen zwischen 341 Nachrichten (nur Standardframes ohne Daten) und 78 Nachrichten (nur extended Frames, alle mit 8 Datenbytes) in einen 1kByte großen Puffer.

Device-Treiber

2

Standardframe

Die Abbildung zeigt den Aufbau des Standardframes mit vergrößertem Frame-Info-Byte (oben) und den ID-Bytes (vergrößert unten). Die Länge der Botschaft wird vom Device-Treiber automatisch eingesetzt. Die 11 ID-Bits müssen sich mit dem höchstwertigen Bit zuerst linksbündig in den beiden Bytes befinden, wie in der Abbildung dargestellt.



Aufbau des ‚Standard Frame‘

Standard Frame, Info-Bits:

- FF** Frame-Format-Bit, hier FF=0.
0: Standard Frame
1: extended Frame
- RTR** Remote Transmit Request, Sendeaufforderung. Botschaften mit gesetztem RTR-Bit werden vom Treiber direkt bearbeitet und erscheinen nicht im Puffer.
- DLC** 4 Bit geben die Anzahl der Datenbytes in der Message an (0...8). Diese Bits setzt der Device-Treiber.

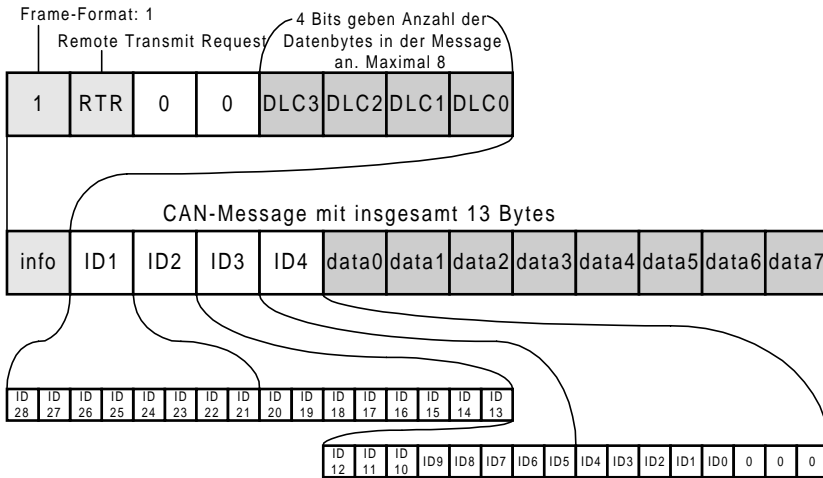
In beiden ID-Bytes zusammen befindet sich der um 5 Bits nach links verschobene 11-Bit-Identifizierer der CAN-Botschaft. Das Format ist hier ‚high-Byte first‘, im Gegensatz zu WORD-Variablen in Tiger-BASIC®, die ‚low-Byte first‘ sind.

Hinter den ID-Bytes folgen so viele Datenbytes, wie durch DLC angegeben.

Beispiel für das Erzeugen von Standardframes in Tiger-BASIC®:

```
t_id = 7FFh shl 5           ' Transmit-ID, linksbueendig in WORD
' Standardframe mit Frame-Info-Byte, 2 leeren ID-Bytes, Daten
msg$ = "<0><0><0>" + data$
msg$ = ntos$ ( msg$, 1, -2, t_id ) ' ID einbauen mit high-Byte zuerst
' Laenge wird vom Treiber eingesetzt
print #CAN, msg$;          ' PRINT, mit Semikolon!!
' oder
put #CAN, msg$
```

Extended Frame



Aufbau des 'extended Frame'

Extended Frame, Info-Bits:

- FF** Frame-Format-Bit, hier FF=1.
0: Standard Frame
1: extended Frame
- RTR** Remote Transmit Request, Sendeaufforderung. Messages mit gesetztem RTR-Bit werden vom Treiber direkt bearbeitet und erscheinen nicht im Puffer.
- DLC** 4 Bit geben die Anzahl der Datenbytes in der Message an (0...8).

In den 4 ID-Bytes befindet sich der um 3 Bits nach links verschobene 29-Bit-Identifizier der CAN-Botschaft. Das Format ist hier 'high-Byte first', im Gegensatz zu LONG-Variablen, die 'low-Byte first' sind.

Hinter den ID-Bytes folgen sovielen Datenbytes, wie durch DLC angegeben.

Beispiel für das Erzeugen von extended Frames in Tiger-BASIC®:

```
t_id = 1FFFFFFh shl 3      ' Transmit-ID, linksbueendig in LONG
' extended Frame mit Frame-Info-Byte, 4 leeren ID-Bytes, Daten
msg$ = "<80h><0><0><0><0>" + data$
msg$ = ntos$ ( msg$, 1, -4, t_id ) ' ID einbauen mit high-Byte zuerst
' Laenge wird vom Treiber eingesetzt
print #CAN, msg$;         ' PRINT mit Semikolon!!
' oder
put #CAN, msg$
```

CAN User-Function-Codes

User-Function-Codes für Abfragen (Instruktion GET):

Nr	Symbol Prefix UFCI_	Beschreibung
1	UFCI_IBU_FILL	Füllstand des Eingangspuffers (Byte)
2	UFCI_IBU_FREE	freier Platz im Eingangspuffer (Byte)
3	UFCI_IBU_VOL	Größe des Eingangspuffers (Byte)
33	UFCI_OBU_FILL	Füllstand des Ausgangspuffers (Byte)
34	UFCI_OBU_FREE	freier Platz im Ausgangspuffer (Byte)
35	UFCI_OBU_VOL	Größe des Ausgangspuffers (Byte)
65	UFCI_LAST_ERRC	letzter Error-Code
152	UFCI_CAN_MODE	liest CAN register MODE
153	UFCI_CAN_STAT	liest CAN register STAT
154	UFCI_CAN_ALC	liest Kopie vom 'arbitration lost register'
155	UFCI_CAN_ECC	liest Kopie vom 'error code capture register'
156	UFCI_CAN_EWL	liest Kopie vom 'error warning limit register'
157	UFCI_CAN_RXERR	liest Kopie vom 'rx error counter register'
158	UFCI_CAN_TXERR	liest Kopie vom 'tx error counter register'
159	UFCI_CAN_RMC	liest Kopie vom 'rx message counter'
99	UFCI_DEV_VERS	Version des Treibers

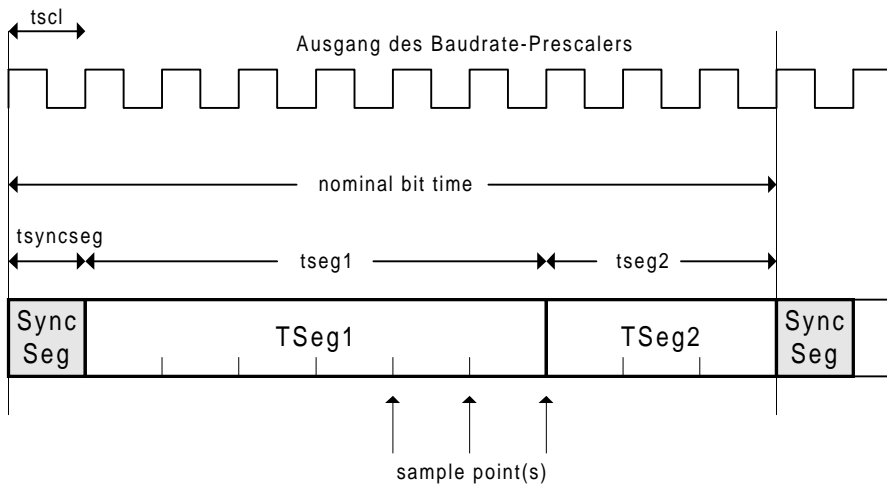
User-Function-Codes für Output (Instruktion PUT):

Nr	Symbol Prefix: UFCO_	Beschreibung
1	UFCO_IBU_ERASE	Eingangspuffer löschen
33	UFCO_OBU_ERASE	Ausgangspuffer löschen
65	UFCO_ERRC_RESET	setze letzten OK-/WARNING-/ERROR-Code zurück
136	UFCO_CAN_CODE	setzt CAN-Register CODE
137	UFCO_CAN_MASK	setzt CAN-Register MASK
138	UFCO_CAN_MODE	setzt CAN-Register MODE
139	UFCO_CAN_BUSTIM0	setzt CAN-Register BUSTIM0
140	UFCO_CAN_BUSTIM1	setzt CAN-Register BUSTIM1
141	UFCO_CAN_OUTCTRL	setzt CAN-Register OUTCTRL
176	UFCO_CAN_RESET	führt ein CAN-Soft-Reset durch

Bus-Timing und Übertragungsrate

Die Übertragungsrate wird durch die Länge eines Bits bestimmt. Ein Bit setzt sich aus drei Abschnitten zusammen, die wiederum aus einzelnen Zeitsegmenten bestehen:

- Sync-Segment, immer ein Zeitsegment lang.
- TSEG1 ist zwischen 5 und 15 Zeitsegmente lang. Innerhalb Tseg1 wird das Bit beim Empfang gesampelt.
- TSEG2 ist zwischen 2 und 7 Zeitsegmente lang.



Aufbau eines Bits

Die Einheit eines Zeitsegmentes wird im Bustiming-Register 0 festgelegt, die Anzahl Zeitsegmente, aus denen TSEG1 und TSEG2 bestehen im Bustiming-Register 1.

Bustiming-Register 0

Die Länge eines Zeitsegmentes ‚t_{scl}‘ wird im **Bustiming-Register-0** durch den Baudrate-Prescaler **BRP** festgelegt. Der 6-Bit-Prescaler kann Werte zwischen 0 und 31 annehmen.

1 Zeitsegment: $t_{scl} = 0,1 * (BRP+1) \text{ } \mu\text{sec}$

1 Bitzeit = T_{sync} + T_{seg1} + T_{seg2}

Die oberen Bits in diesem Register legen die Synchronisations-Sprungweite fest. Der Wert **SJW** bestimmt die maximale Anzahl von Clockzyklen, um die ein Bit verkürzt oder verlängert werden darf, um Phasenunterschiede zwischen verschiedenen Buscontrollern durch Neusynchronisation auszugleichen.

Bustiming-Register 0							
Bit7	Bit6	Bit5	Bit4	Bit3	Bit2	Bit1	Bit0
SJW1	SJW0	BRP5	BRP4	BRP3	BRP2	BRP1	BRP0

Bustiming-Register 1

Im **Bustiming-Register-1** wird festgelegt, aus wievielen Zeitsegmenten **Tseg1** und **Tseg2** bestehen und wie oft das empfangene Bit gesampelt wird (einmal oder dreimal).

Bustiming-Register 1							
Bit7	Bit6	Bit5	Bit4	Bit3	Bit2	Bit1	Bit0
SAM	TSEG2.2	TSEG2.1	TSEG2.0	TSEG1.3	TSEG1.2	TSEG1.1	TSEG1.0

SAM=1: Der Bus wird dreimal gesampelt. Empfohlen für langsame und mittelschnelle Busse, wenn Filterung von Spikes auf dem Bus Vorteile bringt.

SAM=0: Der Bus wird einmal gesampelt. Empfohlen für schnelle Busse.

Welche Werte für T_{seg1} und T_{seg2} Empfangssicherheit gewähren, hängt von den physikalischen Eigenschaften des Übertragungsmediums ab, inklusive Treiberbausteine, Optokoppler. Aus diesen Eigenschaften ergibt sich letztlich auch die erzielbare Baudrate und Leitungslänge.

Device-Treiber

Einige gängige Einstellungen finden Sie in folgender Tabelle (erreichbare Buslängen sind Anhaltspunkte):

Bitrate	Bustim0	Bustim1	Bt1 Tseg1	Bt1 Tseg2	Bus- länge
1 Mbit	0	43h	3	4	25m
500 kBit	0	5Ch	12	5	100m
250 kBit	1	5Ch	12	5	250m
125 kBit	3	5Ch	12	5	500m
100 kBit	4	5Ch	12	5	650m

Bei der Installation des Treibers kann durch Parameter die Bitrate festgelegt werden.

Während der Laufzeit können die Einstellungen des Bustimings mit Hilfe von User-Function-Codes verändert werden.

Anmerkung: der Ausgangspuffer sollte während des Setzens von Bustim0 oder Bustim1 leer sein, da sich der internen CAN-Chip vorübergehend im Resetmodus befindet. Er ist auch vorübergehend nicht empfangsbereit.

Beispiel: stelle 100kBit nach obiger Tabelle während der Laufzeit ein:

```
PUT #CAN, #0, #UFCO_CAN_BUSTIM0, 4  
PUT #CAN, #0, #UFCO_CAN_BUSTIM1, 5CH
```

2

Error-Register

Sowohl der korrekte Empfang einer CAN-Botschaft als auch fehlerhafte Zustände auf dem CAN-Bus lösen einen Empfangs-Interrupt aus. In der Interrupt-Bearbeitung stellt der Device-Treiber fest, ob es sich um ein fehlerfrei empfangenes Paket handelt oder ob Fehler aufgetreten sind. In jedem Falle werden die Werte aufgefrischt, die mit Fehlerzuständen zu tun haben und für die nächste Fehlerabfrage mit einem User-Function-Code bereitgestellt. Treten vor der Fehlerabfrage weitere Fehler auf, dann wird der jeweils neueste Fehlercode abgelegt.

Device-Treiber

Folgende Fehlerabfragen sind möglich:

User-Function-Code	Bit(s)	Bedeutung
UFCL_CAN_STAT	0	Receive Buffer Status: 0: leer 1: voll
	1	Receive Overrun: 0: nein 1: ja Data-Overrun. Tritt ein, wenn eine neue CAN-Message bereits eintrifft, obwohl noch nicht genügend Platz im Empfangsbereich im CAN-Chip ist. Dies betrifft nicht den Puffer des Device-Treibers.
	2	Transmit Buffer: 0: blockiert 1: frei
	3	Senden: 0: läuft 1: fertig
	4	Empfangen: 0: frei 1: läuft
	5	Senden: 0: frei 1: läuft
	6	Fehler: 0: ok 1: ein Fehlerzähler oder beide (RXERR, TXERR) hat den gesetzten Wert für Error-Warning-Limit überschritten.
	7	Bus-Status: 0: ON 1: OFF Bei OFF beteiligt sich die CAN-Hardware nicht mehr an Aktivitäten auf dem Bus.
UFCL_CAN_ALC	0..4	Arbitration-Lost-Capture. Zeigt an bei welchem Bit incl. RTR-Bit der Buszugriff verloren ging.
	5..7	Reserviert
UFCL_CAN_ECC	0..4	Error-Code-Capture gibt an in welchem Segment der Fehler auftrat. Nähere Beschreibung weiter unten im Text.
	5	Richtung: 0: Tx 1: Rx
	6,7	Errortyp
UFCL_CAN_RXERR	0..7	Rx-Fehlerzähler. Zählt bei Empfangsfehler hoch und bei korrektem Empfang wieder runter bis 0. Siehe auch Error-Warning-Limit
UFCL_CAN_TXERR	0..7	Tx-Fehlerzähler. Zählt bei Sendefehler hoch und bei korrekter Sendung wieder runter bis 0. Siehe auch Error-Warning-Limit

Arbitration-Lost-Fehler

Die Abfrage des ALC-Registers kann näheren Aufschluß darüber geben, an welcher Bitposition der Buszugriff verloren ging. Auf dem CAN-Bus erscheint nach dem Startbit zunächst das höchstwertigste Identifier-Bit. Es folgen im Falle eines Standard-Frames 10 weitere Identifier-Bits. Da die ‚extended Frames‘ zu den Standard-Frames kompatibel sein müssen, folgt nach diesen 10 Identifier-Bits in jedem Fall ein RTR-Bit. Das nächste Bit entscheidet nun, ob es sich um ein Standard-Frame oder ein ‚extended Frame‘ handelt. Es heißt IDE-Bit, **I**dentifier **E**xtension. Nach einem reservierten Bit folgen im Falle des ‚extended Frame‘ die restlichen 18 Identifier-Bits. Das Arbitration-Lost-Register kann bis zum 31. Bit die Arbitrierung mitverfolgen, das ist bis zum RTR-Bit eines ‚extended Frame‘.

Da alle Teilnehmer gleichzeitig auf den Bus zugreifen, zeigt das erste rezessive Bit, welches von einem dominanten Bit überschrieben wurde, den verlorenen Buszugriff an. Die Bitposition ist dabei ein Maß für die Priorität des Teilnehmers, der den Buszugriff verhindert.

Beachte: Bei jedem Interrupt wird der gepufferte Wert im Device aufgefrischt. Da das ALC-Register der CAN-Hardware zurückgesetzt wird, wenn es gelesen wird, wird ein einmal aufgetretener und registrierter Arbitration-Lost-Fehler beim nächsten korrekten Empfang überschrieben. Einzelne Arbitration-Lost-Zustände können daher nur erfaßt werden, wenn genügend Zeit besteht, um den Wert vom Treiber auszulesen. Immer wieder auftretende Arbitration-Lost-Zustände werden statistisch erfaßt.

ECC-Fehler-Register

Nachdem ein Bus-Error aufgetreten ist, rettet der Device-Treiber das ECC-Register (Error Code Capture) des CAN-Chips. Die Abfrage des ECC-Registers gibt Aufschluß über den Busfehler:

2

ECC-Code	Ursache des Busfehler
2	ID.28 bis ID.21
3	„start of frame“
4	Bit SRTR (Substitute RTR)
5	Bit DIE (Identifier Extended)
6	ID.20 bis ID.18
7	ID.17 bis ID.13
8	CRC-Sequenz
9	reserviertes Bit 0
10	Datenfeld
11	DLC (Data Length Code)
12	Bit RTR
13	reserviertes Bit 1
14	ID.4 bis ID.0
15	ID.12 bis ID.5
17	Active Error Flag
18	Intermission
19	tolerate dominant bits
22	Passive Error Flag
23	Error Delimiter
24	CRC-Delimiter
25	Acknowledge-Slot
26	End of Frame
27	Acknowledge-Delimiter
28	Overload Flag

RXERR-Empfangsfehlerzähler

Bei jedem CAN-Interrupt wird im Device-Treiber der Empfangsfehlerzähler ausgelesen. Der letzte Wert kann mit einem User-Function-Code abgefragt werden. Die Abfrage verändert nicht den Zählerstand.

```
...  
get #CAN, #0, #UFCI_CAN_RXERR, 1, rx_err  
...
```

Wenn der Zählerstand den gesetzten Error-Warning-Limit (Standard: 96) überschreitet, wird das Bit 6 im Statusregister gesetzt.

Wenn der Zählerstand 127 überschreitet, geht der interne CAN-Chip in den Modus ‚Bus-Error-Passive‘ über und das Bit 7 im Statusregister wird gesetzt. In diesem Modus versendet die CAN-Hardware keine Fehlertelegramme mehr, sendet und empfängt seine Telegramme aber weiterhin. Fehlerfreie Datentelegramme auf dem Bus reduzieren den Fehlerzähler wieder.

TXERR-Sendefehlerzähler

Bei Fehler-Interrupts wird im Device-Treiber der Sendefehlerzähler ausgelesen. Der letzte Wert kann mit einem User-Function-Code abgefragt werden. Die Abfrage verändert nicht den Zählerstand.

```
...  
get #CAN, #0, #UFCI_CAN_TXERR, 1, tx_err  
...
```

Wenn der Zählerstand den gesetzten Error-Warning-Limit (Standard: 96) überschreitet, wird das Bit 6 im Statusregister gesetzt.

Wenn der Zählerstand 127 überschreitet, geht der interne CAN-Chip in den Modus ‚Bus-Error-Passive‘ über und das Bit 7 im Statusregister wird gesetzt. In diesem Modus versendet die CAN-Hardware keine Fehlertelegramme mehr, sendet und empfängt seine Telegramme aber weiterhin. Fehlerfreie Datentelegramme auf dem Bus reduzieren den Fehlerzähler wieder.

Wenn der Zählerstand 255 überschreitet, geht der CAN-Chip in den Buss-Off-Zustand. Dieser Zustand kann nur durch einen Hardware-Reset oder Software-Reset verlassen werden.

Empfangsfilter mit Code und Mask

Der gesetzte Access-Code bestimmt zusammen mit dem Access-Filter, welche CAN-Botschaften empfangen werden. Durch die Access-Maske werden Bits zu ‚don’t care‘ gesetzt, wenn dies erforderlich ist. Die Bits des empfangenen Identifiers, die nicht ‚don’t care‘ sind, müssen mit dem Code übereinstimmen, damit die Botschaft empfangen wird.

Es folgen die Beschreibungen:

- Setzen von Access-Code und Access-Mask
- Standard-Frame mit Single filter configuration
- Extended Frame mit Single filter configuration
- Standard-Frame mit Dual-Filter-Konfiguration
- Extended Frame mit Dual-Filter-Konfiguration

Die empfangene CAN-Botschaft kann als Standard-Frame oder als Extended-Frame vorliegen. Ausserdem gibt es zwei Filtermodi: ‚Single filter configuration‘ oder ‚Dual filter configuration‘.

Setzen von Access-Code und Access-Mask

Access-Code und Access-Mask sind als Register Bestandteil der CAN-Hardware und werden bei der Installation des Device-Treibers gesetzt. Wenn keine Parameter angegeben sind, wird Access-Code auf 0 und Access-Mask auf 0FFFFFFFh gesetzt, so daß alle Nachrichten den Filter passieren.

Man kann den Code und die Maske als einfaches Bitmuster oder als Zahl sehen. Zum Beispiel eignet sich eine LONG-Zahl, um die Bits des Access-Codes oder der Access-Mask zu speichern. Ein Problem ergibt sich dadurch, daß die CAN-Zahl mit dem höchstwertigen Byte beginnt, die Tiger-BASIC®-LONG-Zahl jedoch mit dem niedrigwertigsten:

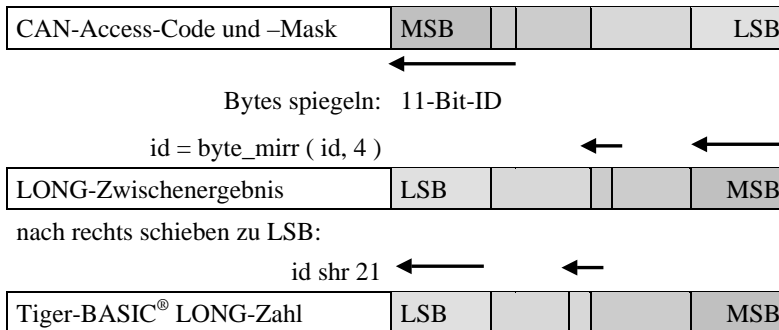
CAN-Access-Code und -Mask	MSB			LSB
---------------------------	-----	--	--	-----

Tiger-BASIC® LONG-Zahl	LSB			MSB
------------------------	-----	--	--	-----

Hinzu kommt, daß je nach Frame-Typ die 11 Bits bzw. die 29 Bits linksbündig in den 32 Bit für den Identifier stehen. Zahlen beginnen jedoch rechts mit dem niedrigsten Bit und haben keine ‚don’t care‘-Bits rechts davon stehen. Links einer Zahl können jedoch Nullen stehen, die keine Rolle spielen.

Will man also den Identifier aus dem Access-Code als Zahl sehen, dann müssen die Bytes erst gespiegelt werden, und

- bei 11-Bit-Identifier der Wert von Access-Code um 21 Bit (5+16) nach rechts geschoben werden.
- bei 29-Bit-Identifier der Wert von Access-Code um 3 Bit nach rechts geschoben werden.



Umgekehrt: hat man eine Zahl vorliegen und will sie in ein CAN-Register Access-Code oder Access-Mask ablegen, dann

- müssen die Bits der Zahl zuerst nach links geschoben werden
- dann die Bytes der Zahl gespiegelt werden

Beachten Sie, daß die Funktion `NTOS$` die Spiegelung der Bytes vornehmen kann indem als Argument für die Anzahl der Bytes eine negativer Wert angegeben wird:

- `msg$ = ntos$ (msg$, 1, -2, t_id)` baut einen 11-Bit-Identifier, der als WORD-Zahl mit den ID-Bits bereits an der richtigen Stelle vorliegt, in einen String ein und spiegelt dabei die Bytes.
- `msg$ = ntos$ (msg$, 1, -4, t_id)` tut das gleiche für einen 29-Bit-Identifier, der als LONG-Zahl mit den ID-Bits bereits an der richtigen Stelle vorliegt.

In einem String verändert sich die Reihenfolge nicht:

```
id$ = "<1Fh><AAh><BBh><33h>"
```

oder

```
id$ = "1F AA BB 33"%
```

Steppen Sie das nachfolgende Beispielprogramm, um in den 'Überwachten Ausdrücken' die Gegebenheiten nachzuvollziehen.

Device-Treiber

Programmbeispiel:

2

```
-----
' Name: CAN_SET_FILTER.TIG
' Setzt Filter-Konfiguration
' Demonstriert das Setzen von Access-Code und Access-Mask
' in verschiedenen Varianten
' Nur ein CAN-Tiger notwendig, da nicht gesendet oder empfangen wird
' Benutzen Sie das Kommando 'Ueberwachte Ausdruecke' aus
' dem Menue 'Anzeige'
-----
user var strict                ' unbedingte Var.deklaration
#include UFUNC3.INC            ' User Function Codes
#include DEFINE_A.INC          ' allg. Symbol-Definitionen
#include CAN.INC                ' CAN-Definitionen

LONG ac_code, ac_mask
STRING id$

-----
TASK MAIN
  install_device #LCD, "LCD1.TDD" ' LCD-Treiber installieren
  install_device #CAN, "CAN1_K1.TDD", & ' CAN-Treiber installieren
    "12 34 56 78 &                ' access code
    EF FF FE FF &                ' access mask
    10 45 &                       ' bustim1, bustim2
    08 1A"%                       ' single filter mode, outctrl

  using "UH<8><8> 0 0 0 4 4"      ' fuer ID Anzeige im gesamten Prg

' zeige Access-Code und Access-Mask wie installiert
get #CAN, #0, #UFCl_CAN_CODE, 0, ac_code ' und access code lesen
ac_code = byte_mirr ( ac_code, 4 ) ' Bytefolge gespiegelt fuer LONG
print using #LCD, "<1>ac_code: ";ac_code
get #CAN, #0, #UFCl_CAN_MASK, 0, ac_mask ' und access mask lesen
ac_mask = byte_mirr ( ac_mask, 4 ) ' Bytefolge gespiegelt fuer LONG
print using #LCD, "ac_mask: ";ac_mask
' dieselben Zeilen in show_codemask
wait_duration 1000

' siehe Byte-Reihenfolge ('Ueberwachte Ausdruecke' id$ und ac_code)
get #CAN, #0, #UFCl_CAN_CODE, 4, id$ ' Test: Access-Code lesen
get #CAN, #0, #UFCl_CAN_CODE, 0, ac_code ' und in LONG lesen
wait_duration 1000

' wenn der Code als Zahl vorliegt:
ac_code = byte_mirr ( (1FFFFFFFh shl 3), 4 ) ' groesster Access-Code
put #CAN, #0, #UFCl_CAN_CODE, ac_code ' und Access-Code setzen
call show_codemask ' und anzeigen
wait_duration 1000

' das ist dasselbe:
id$ = "FF FF FF F8"% ' 1FFFFFFF linksbuendig
put #CAN, #0, #UFCl_CAN_CODE, id$ ' und setzen
call show_codemask ' und anzeigen
wait_duration 1000

' neuen Code setzen fuer nachfolgenden Lesetest
ac_code = byte_mirr ( (12345678h shl 3), 4 ) ' wird 0C0B3A291h
put #CAN, #0, #UFCl_CAN_CODE, ac_code ' und Access-Code setzen
```

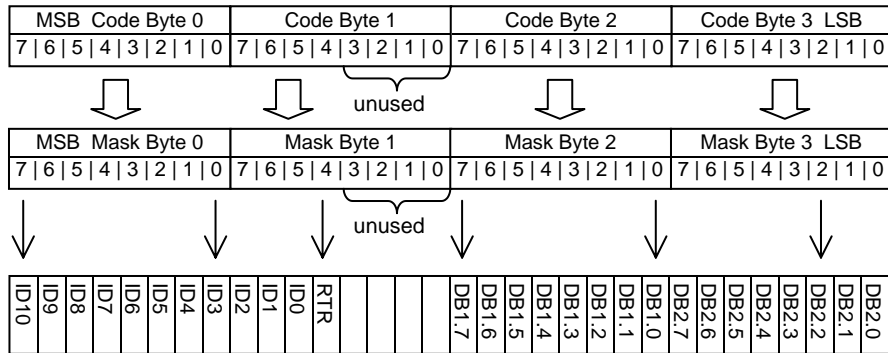
```
wait_duration 1000
' ab hier steppen
get #CAN, #0, #UFCI_CAN_CODE, 0, ac_code ' siehe Byte-Reihenfolge
ac_code = byte_mirr ( ac_code, 4 ) ' nach jedem Schritt
ac_code = ac_code shr 3
print_using #LCD, "<1>ac_code: ";ac_code

END

' -----
' zeigt Access-Code und Access-Mask an
' -----
SUB show_codemask
get #CAN, #0, #UFCI_CAN_CODE, 0, ac_code ' und access code lesen
ac_code = byte_mirr ( ac_code, 4 ) ' Bytefolge gespiegelt fuer LONG
print_using #LCD, "<1>ac_code: ";ac_code
get #CAN, #0, #UFCI_CAN_MASK, 0, ac_mask ' und access mask lesen
ac_mask = byte_mirr ( ac_mask, 4 ) ' Bytefolge gespiegelt fuer LONG
print_using #LCD, "ac_mask: ";ac_mask
END
```

Standard-Frame mit Single-Filter-Konfiguration

Im Modus ‚single filter‘ werden beim **Standard-Frame** alle ID-Bits inclusive RTR-Bit und den ersten beiden Datenbytes durch den Access-Filter geleitet und mit dem gesetzten Code verglichen. Es können jedoch auch Botschaften mit weniger als 2 Datenbytes den Filter passieren, d.h. nicht vorhandene Datenbytes erfüllen immer die Filterbedingung. Die 4 untersten Bits zwischen RTR-Bit und Daten sollten aus Gründen der Kompatibilität ‚don’t care‘ maskiert werden.



In dem Beispielprogramm CAN_FILTER_SS.TIG wird der Access-Code nach der Installation auf 4EE0 0000 gesetzt. Die Maske bestimmt, welche Bits des gesetzten Codes relevant sind. Der Wert F11F FFFF hat im Bereich des Identifiers (die 11 Bits linksbündig) insgesamt 6 ‚0‘-Bits, die besagen, daß dieses Bits in der Nachricht auf dem Bus mit dem Access-Code übereinstimmen müssen, damit die Nachricht empfangen wird. Der Test zeigt, daß die Werte durchkommen, die an zweiter Stelle ein ‚E‘ oder ‚F‘ haben und an dritter Stelle ein ‚E‘. An dritter Stelle kann kein F stehen, da das 12. Bit das RTR-Bit ist. Es werden also genau die Nachrichten empfangen, deren Bits zu den relevanten Bits des Access-Code passen.

Die Abbildung zeigt den Access-Code, die Access-Maske und einen Identifier als Beispiel. Nur die ID-Bits sind gezeigt. Die anderen Bits sind im Beispiel sowieso ‚don’t care‘:

	ID10	ID9	ID8	ID7	ID6	ID5	ID4	ID3	ID2	ID1	ID0	RTR
Code: 4EEh	0	1	0	0	1	1	1	0	1	1	1	0
Maske: F11h	1	1	1	1	0	0	0	1	0	0	0	1
x=nicht relevant	x	x	x	x	1	1	1	x	1	1	1	x
ID: 0EEh	0	0	0	0	1	1	1	0	1	1	1	0
ID: 7FEh	0	1	1	1	1	1	1	1	1	1	1	0

Programmbeispiel:

```

'-----
' Name: CAN_Filter_SS.TIG
' Single-Filter-Konfiguration
' Sendet Standard Frames mit verschiedenen IDs fuer Filtertest
' empfaengt gefiltert CAN-Botschaften und zeigt sie auf LCD an
' unterscheidet Standard und extended Frame
' Verbinde einen zweiten CAN-Tiger mit demselben Programm
'-----
user var strict                ' unbedingte Var.deklaration
#include UFUNC3.INC             ' User Function Codes
#include DEFINE_A.INC          ' allg. Symbol-Definitionen
#include CAN.INC                ' CAN-Definitionen

BYTE frameformat, msg_len, can_stat
LONG ac_code, ac_mask
LONG r_id                      ' Rx ID
STRING id$(4), msg$(13), data$(8)

'-----
TASK MAIN
  BYTE ever                    ' fuer Endlosschleife
  WORD ibu_fill                ' Eingangspufferfuellung

  install_device #LCD, "LCD1.TDD" ' LCD-Treiber installieren
  install_device #CAN, "CAN1_K1.TDD", & ' CAN-Treiber installieren
    "4E E0 00 00 &           ' access code
    F1 1F FF FF &           ' access mask
    10 45 &                  ' bustim1, bustim2
    08 1A"%                  ' single filter mode, outctrl

' Code und Mask sind nun so gesetzt:
' 01001110111 RTR --data-- --data-- code (11 relevant Bits)
' 11110001000 1 11111111 11111111 mask (0-Bits zaehlen, 1=don't care)
' es passieren also alle Botschaften, die folgende Bitmuster haben:
' -----ID----- RTR --data-- --data-- code (11 relevant Bits)
' xxxxx111x111 x xxxxxxxxxx xxxxxxxxxx
' empfangen wird 0EEh, 0FEh, 1EEh, 1FEh, etc

using "UH<8><8> 0 0 0 4 4"
get #CAN, #0, #UFCI_CAN_CODE, 0, ac_code ' und access code lesen
ac_code = byte_mirr ( ac_code, 4 ) ' Bytefolge gespiegelt fuer LONG
print_using #LCD, "<1>ac_code:";ac_code

get #CAN, #0, #UFCI_CAN_MASK, 0, ac_mask ' und access mask lesen
ac_mask = byte_mirr ( ac_mask, 4 ) ' Bytefolge gespiegelt fuer LONG
print_using #LCD, "ac_mask:";ac_mask

run_task generate_frames          ' erzeuge aufsteigende IDs

' zeige nun IDs der empfangenen Botschaften an
for ever = 0 to 0 step 0          ' Endlosschleife
  get #CAN, #0, #UFCI_IBU_FILL, 0, ibu_fill
  print #LCD, "<1Bh>A<0><2><0F0h>IBU_FILL:";ibu_fill; " ";
  if ibu_fill > 2 then            ' wenn mindestens eine Message
    get #CAN, #0, 1, frameformat ' hole Frame-Info-Byte
    msg_len = frameformat bitand 1111b ' Laenge
    if frameformat bitand 80h = 0 then ' wenn Standard-Frame
      get #CAN, #0, CAN_ID11_LEN, r_id ' hole ID-Bytes

```

```

        disable_tsw
        using "UH<4><4>  0 0 0 0 4"
    else                                     ' sonst ist es extended frame
        get #CAN, #0, CAN_ID29_LEN, r_id' und damit nicht von SLIO
        r_id = byte_mirr ( r_id, 4 )
        disable_tsw
        using "UH<8><8>  0 0 0 4 4"
    endif
    print_using #LCD, "<1Bh>A<0><2><0F0h>ID rcvd:";r_id;
    enable_tsw

    if msg_len > 0 then                     ' wenn Daten
        get #CAN, #0, msg_len, data$      ' hole sie aus dem Puffer
    endif
endif
'   get #CAN, #0, #UFCI_CAN_STAT, 0, can_stat
'   using "UH<2><2>  0 0 0 0 2" ' HEX-Format fuer ein Byte
'   print_using #LCD, "<1Bh>A<1><1><0F0h>";can_stat;
next
END

' -----
' Erzeugt Standardframes mit aufsteigender ID
' -----

TASK generate_frames
BYTE ever                                     ' fuer Endlosschleife
WORD obu_free                               ' Platz im Ausgangspuffer
LONG t_id                                    ' Tx ID
STRING msg$(13)

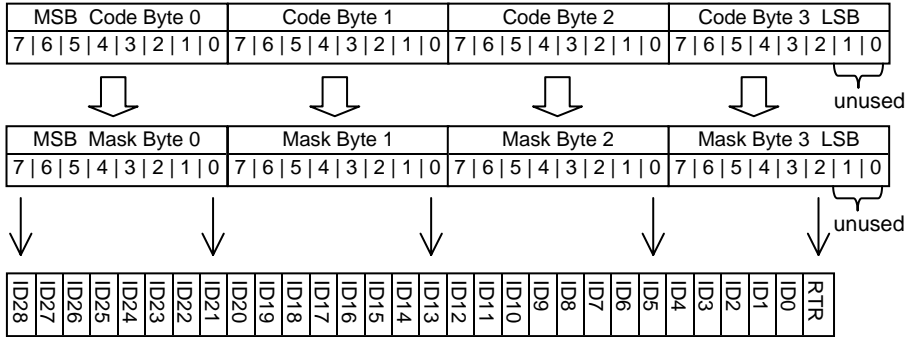
t_id = 0                                     ' Standard Identifier
for ever = 0 to 0 step 0                    ' Endlosschleife
    get #CAN, #0, #UFCI_OBU_FREE, 0, obu_free
    if obu_free > 13 then
' Frame-Info 0 = standard, 2 ID-Bytes, keine Daten
        msg$ = "<0><0><0>"
        msg$ = ntos$( msg$, 1, -2, t_id ) ' ID high-Byte zuerst einb
        put #CAN, #0, msg$                ' sende Message im Standard-Frame
        disable_tsw
        using "UH<4><4>  0 0 0 0 4" ' fuer ID Anzeige
        print_using #LCD, "<1Bh>A<0><3><0F0h>ID sent:";t_id;
        enable_tsw

                                     ' dies zaehlt t_id um 1 hoch
                                     ' wenn der Shift um 5 des ID
                                     ' bruecksichtigt wird
        t_id = t_id + 100000b          ' naechste ID
        t_id = t_id bitand 0FFFFh     ' bleibe im Bereich Standardframe-ID
    endif
    wait_duration 30
next
END

```


Extended Frame mit Single-Filter-Konfiguration

Beim **Extended-Frame** werden alle ID-Bits inklusive RTR-Bit durch den Filter geleitet. Die 2 untersten Bits sollten aus Gründen der Kompatibilität ‚don’t care‘ maskiert werden.



Device-Treiber

Programmbeispiel:

2

```
-----
' Name: CAN Filter ES.TIG
' Single-Filter-Konfiguration
' Sendet extended Frames mit verschiedenen IDs fuer Filtertest
' empfaengt gefiltert CAN-Botschaften und zeigt sie auf LCD an
' unterscheidet Standard und extended Frame
' Verbinde einen zweiten CAN-Tiger mit demselben Programm
-----
user var strict                ' unbedingte Var.deklaration
#include UFUNC3.INC            ' User Function Codes
#include DEFINE A.INC          ' allg. Symbol-Definitionen
#include CAN.INC                ' CAN-Definitionen

BYTE frameformat, msg_len, can_stat
LONG ac_code, ac_mask
LONG r_id
STRING id$(4), msg$(13), data$(8)

-----
TASK MAIN
BYTE ever                      ' fuer Endlosschleife
WORD ibu_fill                  ' Eingangspufferfuellung

install_device #LCD, "LCD1.TDD" ' LCD-Treiber installieren
install_device #CAN, "CAN1_K1.TDD", & ' CAN-Treiber installieren
    "6D 55 D9 98 &            ' access code
    EF FF FE FF &            ' access mask
    10 45 &                   ' bustim1, bustim2
    08 1A"%                   ' single filter mode, outctrl

using "UH<8><8> 0 0 0 4 4"      ' fuer ID Anzeige im gesamten Prg

get #CAN, #0, #UFICI_CAN_CODE, 4, id$ ' Test: Access-Code lesen
' Bytefolge mit Anzeige - ueberwachte Ausdruecke ansehen
get #CAN, #0, #UFICI_CAN_CODE, 0, ac_code ' und access code lesen
ac_code = byte mirr ( ac_code, 4 ) ' Bytefolge gespiegelt fuer LONG
print_using #LCD, "<1>ac_code: ";ac_code
wait_duration 2000

' Code und Mask werden fuer extended Frames nun so gesetzt:
' 22222222 21111111 111
' 87654321 09876543 21098765 43210Rxx RTR, 2x don't care
' 01101101 01010101 11011001 10011000 code (29 relevante Bits+RTR)
' 11101111 11111111 11111110 11111111 mask (0-Bits sind relevant)
'                                     RTR und 2 nicht benutzte Bits
' es passieren also alle Botschaften, die folgende Bitmuster haben:
' xxx0xxxx xxxxxxxx xxxxxxxx1 xxxxxxxx
' Bit 5 muss gesetzt und Bit 25 0 sein

' wenn der Code als Zahl vorliegt:
ac_code = byte mirr ( (0DAABB33h shl 3), 4 ) ' neuer Access-Code
put #CAN, #0, #UFICO_CAN_CODE, ac_code ' und Access-Code setzen
' das ist dasselbe:
' id$ = "6D 55 D9 98"%                ' neuer Access-Code
' put #CAN, #0, #UFICO_CAN_CODE, id$ ' und Access-Code setzen

' Bytefolge wieder mit Anzeige - ueberwachte Ausdruecke ansehen
get #CAN, #0, #UFICI_CAN_CODE, 4, id$ ' und Access-Code in String
```

```

get #CAN, #0, #UFCI_CAN_CODE, 0, ac_code ' und in LONG lesen
ac_code = byte_mirr ( ac_code, 4 )
print_using #LCD, "<1>ac_code:";ac_code
wait_duration 1000

ac_mask = byte_mirr ( 0FFFFFFFh, 4 ) ' Access-Maske
put #CAN, #0, #UFCI_CAN_MASK, ac_mask ' und access mask setzen
get #CAN, #0, #UFCI_CAN_MASK, 0, ac_mask ' und access mask lesen
ac_mask = byte_mirr ( ac_mask, 4 ) ' Bytefolge gespiegelt fuer LONG
print_using #LCD, "ac_mask:";ac_mask

run_task generate_frames ' erzeuge aufsteigende IDs

' zeige nun IDs der empfangenen Botschaften an
for ever = 0 to 0 step 0 ' Endlosschleife
  get #CAN, #0, #UFCI_IBU_FILL, 0, ibu_fill
  ' print #LCD, "<1Bh>A<0><2><0F0h>IBU_FILL:";ibu_fill;" ";
  if ibu_fill > 2 then ' wenn mindestens eine Message
    get #CAN, #0, 1, frameformat ' hole Frame-Info-Byte
    msg_len = frameformat bitand 1111b ' Laenge
    if frameformat bitand 80h = 0 then ' wenn Standard-Frame
      get #CAN, #0, CAN_ID11_LEN, r_id ' hole ID-Bytes
      r_id = byte_mirr ( r_id, 2 )
      r_id = r_id shr 5
    else ' sonst ist es extended frame
      get #CAN, #0, CAN_ID29_LEN, r_id ' und damit nicht von SLIO
      r_id = byte_mirr ( r_id, 4 )
      r_id = r_id shr 3
      if msg_len > 0 then ' wenn Daten
        get #CAN, #0, msg_len, data$ ' hole sie aus dem Puffer
      endif
    endif
    disable_tsw
    using "UH<8><8> 0 0 0 4 4" ' fuer ID Anzeige to
    print_using #LCD, "<1Bh>A<0><2><0F0h>ID rcvd:";r_id;
    enable_tsw

    if msg_len > 0 then ' wenn Daten
      get #CAN, #0, msg_len, data$ ' hole sie aus dem Puffer
    endif
  endif
  ' get #CAN, #0, #UFCI_CAN_STAT, 0, can_stat
  ' using "UH<2><2> 0 0 0 0 2" ' HEX-Format fuer ein Byte
  ' print_using #LCD, "<1Bh>A<1><1><0F0h>" ;can_stat;
next
END

'-----
' Erzeugt exteded Frames mit aufsteigender ID
'-----

TASK generate_frames
  BYTE ever
  WORD obu_free
  LONG t_id
  STRING msg$(13)

  using "UH<8><8> 0 0 0 4 4" ' fuer ID Anzeige im gesamten Prg
  t_id = 0AABB00h shl 3 ' extended Identifier
  for ever = 0 to 0 step 0 ' Endlosschleife

```

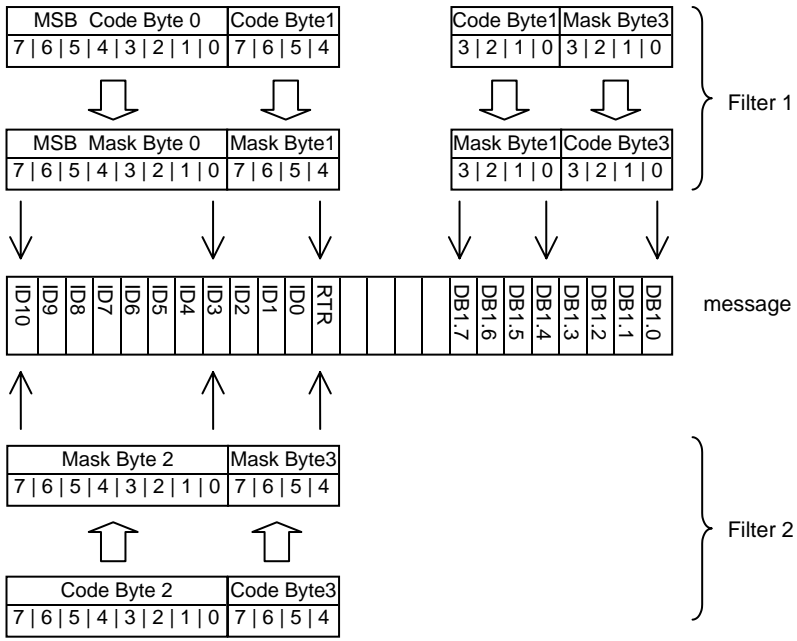
Device-Treiber

2

```
if obu_free > 13 then
' Frame-Info 80h = extended, 4 ID-Bytes, keine Daten
  msg$ = "<80h><0><0><0><0>"
  msg$ = ntos$ ( msg$, 1, -4, t_id ) ' ID high-Byte zuerst einb
  put #CAN, #0, msg$                ' sende Message im Standard-Frame
  print_using #LCD, "<1Bh>A<0><3><0F0h>ID sent: "; t_id shr 3;
                                     ' dies zaehlt um 1 im Byte 0 und 3
                                     ' wenn der Shift um 3 des ID
                                     ' bruecksichtigt wird
                                     ' naechste ID
  t_id = t_id + 08000008h
endif
wait_duration 50
next
END
```

Standard-Frame mit Dual-Filter-Konfiguration

Beim **Standard-Frame** werden alle ID-Bits inclusive RTR-Bit und dem ersten Datenbyte durch den ersten Access-Filter geleitet und mit dem gesetzten Code verglichen. Es können jedoch auch Botschaften mit weniger als 2 Datenbytes den Filter passieren. Ausserdem werden alle ID-Bits inclusive RTR-Bit durch den zweiten Access-Filter geleitet und mit dem gesetzten Code verglichen. Ist der Vergleich bei einem der beiden Filter erfolgreich, dann wird die CAN-Botschaft empfangen. Wenn das erste Datenbyte bei der Filterung keine Rolle spielen soll, dann werden die untersten 4 Bits der Filtermaske auf ‚don’t care‘ gesetzt. Dann arbeiten beide Filter gleich.



In der Anzeige des Beispielprogramms auf dem LCD ist deutlich zu sehen, wie beim Doppelfilter eine Gruppe alles herausfiltert, was als dritte HEX-Ziffer ein ‚E‘ hat, während der zweite Filter alles durchläßt, was als zweite HEX-Ziffer ein ‚A‘ hat. Die meiste Zeit sind Zahlen nach dem Muster ‚xxE0‘ auf dem LCD zu sehen, also E ist fest und die Positionen xx zählen durch (Filter1). Wenn jedoch als zweite Ziffer das A erscheint, wird die dritte Position mit dem E durchgezählt.

Device-Treiber

Programmbeispiel:

2

```
-----
' Name: CAN_Filter_SD.TIG
' Dual-Filter-Konfiguration
' Im Dual-Filter-Mode gibt es 2 Codes und 2 Masken mit 16 Bit
' Sendet Standard Frames mit verschiedenen IDs fuer Filtertest
' empfaengt gefiltert CAN-Botschaften und zeigt sie auf LCD an
' unterscheidet Standard und extended Frame
' Verbinde einen zweiten CAN-Tiger mit demselben Programm
-----
user var strict          ' unbedingte Var.deklaration
#include UFUNC3.INC      ' User Function Codes
#include DEFINE_A.INC    ' allg. Symbol-Definitionen
#include CAN.INC         ' CAN-Definitionen

BYTE frameformat, msg_len, can_stat
LONG ac_code, ac_mask
LONG r_id              ' Rx ID
STRING id$(4), msg$(13), data$(8)

-----
TASK MAIN
  BYTE ever            ' fuer Endlosschleife
  WORD ibu_fill       ' Eingangspufferfuellung

  install_device #LCD, "LCD1.TDD" ' LCD-Treiber installieren
  install_device #CAN, "CAN1_K1.TDD", & ' CAN-Treiber installieren
    "4E E0 4A E0 &      ' access code
    FF 1F F0 FF &      ' access mask
    10 45 &            ' bustim1, bustim2
    00 1A"%           ' dual filter mode, outctrl

  ' In den ersten beiden Bytes sind Code und Mask nun so gesetzt:
  ' 01001110111 RTR --data-- --data-- code (11 relevant Bits)
  ' 11111111000 1 1111          mask (0-Bits zaehlen, 1=don't care)
  ' es passieren also alle Botschaften, die folgende Bitmuster haben:
  ' ----ID---- RTR --data-- --data-- code (11 relevant Bits)
  ' xxxxxxx0111 x xxxxx
  ' empfangen wird alles, was im ID eine 7 im low Nibble hat
  ' hier sichtbar als E im 3. Nibble

  ' In den zweiten beiden Bytes sind Code und Mask nun so gesetzt:
  ' 01001010111 RTR --data-- --data-- code (11 relevant Bits)
  ' 11110000111 1 1111          mask (0-Bits zaehlen, 1=don't care)
  ' es passieren also alle Botschaften, die folgende Bitmuster haben:
  ' ----ID---- RTR --data-- --data-- code (11 relevant Bits)
  ' xxx0101xxxx x xxxxx
  ' empfangen wird alles, was im 2. Nibble ein A hat

  using "UH<8><8> 0 0 0 4 4"      ' fuer ID Anzeige im gesamten Prg
  get #CAN, #0, #UFUCI_CAN_CODE, 0, ac_code ' und access code lesen
  ac_code = byte_mirr ( ac_code, 4 ) ' Bytefolge gespiegelt fuer LONG
  print_using #LCD, "<1>ac_code: ";ac_code

  get #CAN, #0, #UFUCI_CAN_MASK, 0, ac_mask ' und access mask lesen
  ac_mask = byte_mirr ( ac_mask, 4 ) ' Bytefolge gespiegelt fuer LONG
  print_using #LCD, "ac_mask: ";ac_mask

  run_task generate_frames          ' erzeuge aufsteigende IDs
```

```

' zeige nun IDs der empfangenen Botschaften an
for ever = 0 to 0 step 0          ' Endlosschleife
  get #CAN, #0, #UFCI_IBU_FILL, 0, ibu_fill
  print #LCD, "<1Bh>A<0><2><0F0h>IBU_FILL:";ibu_fill;"    ";
  if ibu_fill > 2 then          ' wenn mindestens eine Message
    get #CAN, #0, 1, frameformat ' hole Frame-Info-Byte
    msg_len = frameformat bitand 1111b ' Laenge
    if frameformat bitand 80h = 0 then ' wenn Standard-Frame
      get #CAN, #0, CAN_ID11_LEN, r_id ' hole ID-Bytes
      r_id = byte_mirr ( r_id, 2 )
      disable_tsw
      using "UH<4><4>  0 0 0 0 4"
    else                          ' sonst ist es extended frame
      get #CAN, #0, CAN_ID29_LEN, r_id
      r_id = byte_mirr ( r_id, 4 )
      disable_tsw
      using "UH<8><8>  0 0 0 4 4"
    endif
    print_using #LCD, "<1Bh>A<0><2><0F0h>ID rcvd:";r_id;
    enable_tsw
    if msg_len > 0 then          ' wenn Daten
      get #CAN, #0, msg_len, data$ ' hole sie aus dem Puffer
    endif
  endif
next
END

'-----
' Erzeugt Standardframes mit aufsteigender ID
'-----

TASK generate_frames
  BYTE ever          ' fuer Endlosschleife
  WORD obu_free     ' Platz im Ausgangspuffer
  LONG t_id         ' Tx ID
  STRING msg$(13)

  t_id = 0          ' Standard Identifier
  for ever = 0 to 0 step 0 ' Endlosschleife
    get #CAN, #0, #UFCI_OBU_FREE, 0, obu_free
    if obu_free > 13 then
  ' Frame-Info 0 = standard, 2 ID-Bytes, keine Daten
    msg$ = "<0><0><0>"
    msg$ = ntos$ ( msg$, 1, -2, t_id ) ' ID high-Byte zuerst einb
    put #CAN, #0, msg$          ' sende Message im Standard-Frame
    disable_tsw
    using "UH<4><4>  0 0 0 0 4"
    print_using #LCD, "<1Bh>A<0><3><0F0h>ID sent:";t_id;
    enable_tsw

                                ' dies zaehlt t_id um 1 hoch
                                ' wenn der Shift um 5 des ID
                                ' bruecksichtigt wird
    t_id = t_id + 100000b      ' naechste ID
  endif
  wait_duration 30
next
END

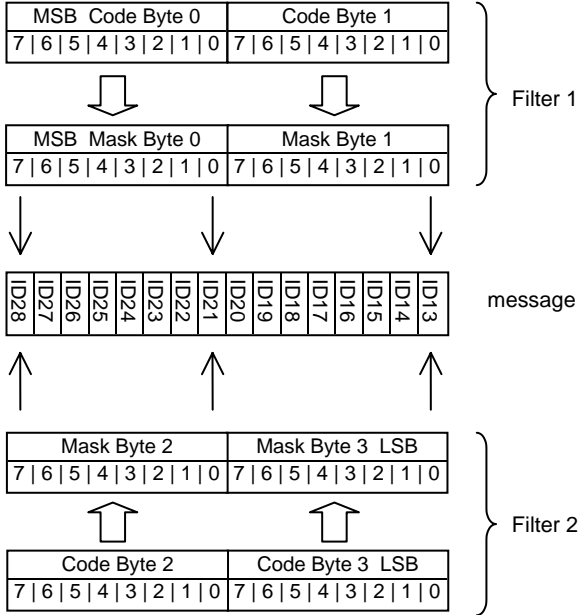
```

Device-Treiber

2

Extended Frame mit Dual-Filter-Konfiguration

Beim **Extended-Frame** arbeiten beide Filter gleich. Es werden die ersten beiden ID-Bytes durch die beiden Access-Filter geleitet und mit dem gesetzten Code verglichen. Es werden nur ID13...ID28 ausgewertet. Signalisiert der Vergleich bei einem der beiden Filter Akzeptanz, dann wird die CAN-Botschaft empfangen.



Device-Treiber

Programmbeispiel:

2

```
-----
' Name: CAN_Filter_ED.TIG
' Dual-Filter-Konfiguration
' Im Dual-Filter-Mode gibt es 2 Codes und 2 Masken mit 16 Bit.
' Sendet extended Frames mit verschiedenen IDs fuer Filtertest,
' empfaengt gefiltert CAN-Botschaften und zeigt sie auf LCD an.
' Unterscheidet Standard und extended Frame.
' Verbinde einen zweiten CAN-Tiger mit demselben Programm.
-----
user var strict                ' unbedingte Var.deklaration
#include UFUNC3.INC            ' User Function Codes
#include DEFINE_A.INC          ' allg. Symbol-Definitionen
#include CAN.INC                ' CAN-Definitionen

BYTE frameformat, msg_len, can_stat
LONG ac_code, ac_mask
LONG r_id                      ' Rx ID
STRING id$(4), msg$(13), data$(8)

-----
TASK MAIN
  BYTE ever                    ' fuer Endlosschleife
  WORD ibu_fill                ' Eingangspufferfuellung

  install_device #LCD, "LCD1.TDD" ' LCD-Treiber installieren
  install_device #CAN, "CAN1_K1.TDD", & ' CAN-Treiber installieren
    "10 55 10 AA &            ' access code
    FF 00 EF 00 &            ' access mask
    10 45 &                   ' bustim1, bustim2
    00 1A"%                   ' dual filter mode, outctrl

' In den ersten beiden Bytes sind Code und Mask nun so gesetzt:
' 22222222 21111111 111
' 87654321 09876543 21098765 43210Rxx RTR, 2x don't care
' 00010000 01010101          code (29 relevante Bits+RTR)
' 11111111 00000000          mask (0-Bits sind relevant)
'                               RTR und 2 nicht benutzte Bits
' es passieren also alle Botschaften, die folgende Bitmuster haben:
' xxxxxxxx 01010101
' empfangen wird alles, was im ID eine 55h im 2. Byte hat      r

' In den zweiten beiden Bytes sind Code und Mask nun so gesetzt:
' 22222222 21111111 111
' 87654321 09876543 21098765 43210Rxx RTR, 2x don't care
' 00010000 10101010          code (29 relevante Bits+RTR)
' 11101111 00000000          mask (0-Bits sind relevant)
'                               RTR und 2 nicht benutzte Bits
' es passieren also alle Botschaften, die folgende Bitmuster haben:
' xxxlxxxx 10101010
' empfangen wird, was mit 0AAh, 10AAh, 20AAh, etc. beginnt

  using "UH<8><8> 0 0 0 4 4" ' fuer ID Anzeige im gesamten Prg

  get #CAN, #0, #UFICI_CAN_CODE, 0, ac_code ' und access code lesen
  ac_code = byte_mirr ( ac_code, 4 ) ' Bytefolge gespiegelt fuer LONG
  print_using #LCD, "<1>ac_code: ";ac_code

  get #CAN, #0, #UFICI_CAN_MASK, 0, ac_mask ' und access mask lesen
```

```

print_using #LCD, "ac_mask: "; ac_mask

run_task generate_frames          ' erzeuge aufsteigende IDs

' zeige nun IDs der empfangenen Botschaften an
for ever = 0 to 0 step 0          ' Endlosschleife
  get #CAN, #0, #UFCI_IBU_FILL, 0, ibu_fill
  print #LCD, "<1Bh>A<0><2><0F0h>IBU_FILL: "; ibu_fill; "      ";
  if ibu_fill > 2 then            ' wenn mindestens eine Message
    get #CAN, #0, 1, frameformat ' hole Frame-Info-Byte
    msg_len = frameformat bitand 1111b ' Laenge
    if frameformat bitand 80h = 0 then ' wenn Standard-Frame
      get #CAN, #0, CAN_ID11_LEN, r_id ' hole ID-Bytes
      r_id = byte_mirr ( r_id, 2 )
    else                            ' sonst ist es extended frame
      get #CAN, #0, CAN_ID29_LEN, r_id ' und damit nicht von SLIO
      r_id = byte_mirr ( r_id, 4 )    ' ungeschiftet anzeigen
    endif
    print_using #LCD, "<1Bh>A<0><2><0F0h>ID rcvd: "; r_id;

    if msg_len > 0 then              ' wenn Daten
      get #CAN, #0, msg_len, data$   ' hole sie aus dem Puffer
    endif
  endif
next
END

'-----
' Erzeugt extended Frames mit aufsteigender ID
'-----
TASK generate_frames
  BYTE ever                          ' fuer Endlosschleife
  WORD obu_free                       ' Platz im Ausgangspuffer
  LONG t_id                           ' Tx ID
  STRING msg$(13)

  t_id = 10000020h                    ' extended Identifier
  for ever = 0 to 0 step 0            ' Endlosschleife
    get #CAN, #0, #UFCI_OBU_FREE, 0, obu_free
    if obu_free > 11 then
  ' Frame-Info 80h = extended, 4 ID-Bytes, keine Daten
    msg$ = "<80h><0><0><0><0>"
    msg$ = ntos$( msg$, 1, -4, t_id ) ' ID high-Byte zuerst einb
    put #CAN, #0, msg$                ' sende Message im Standard-Frame
    disable_tsw
    using "UH<8><8> 0 0 0 4 4"
    print_using #LCD, "<1Bh>A<0><3><0F0h>ID sent: "; t_id;
    enable_tsw

                                ' dies zaehlt um 1 im Byte 2 und 3
                                ' wenn der Shift um 3 des ID
                                ' bruecksichtigt wird
                                ' naechste ID
    t_id = t_id + 0008000h
    t_id = t_id bitand 0FFFFh        ' bleibe im Bereich Standardframe-ID
  endif
  wait_duration 50
next
END

```

Device-Treiber

2

Versenden von CAN-Botschaften

Der CAN-Device-Treiber unterstützt folgende Methoden des Versands:

Versenden einzelner Botschaften, die 0..8 Zeichen enthalten und deren Identifier nach Bedarf einzeln festgelegt wird. Jede CAN-Botschaft wird mit einer PUT- oder PRINT-Instruktion ausgegeben. Bei der Print-Instruktion ist zu beachten daß die Ausgabe formatiert wird und eventuell zusätzliche Byte (CR, LF) angehängt werden.

Versenden von Daten, die auch mehr als 8 Zeichen enthalten können. Der Device-Treiber macht soviele CAN-Datenpakete daraus, wie zum Versand der gesamten Datenmenge erforderlich ist und verwendet den Identifier, der am Anfang des Strings angegeben wurde. Die Daten mit einer einzigen PUT- oder PRINT-Instruktion in den Puffer übergeben.

Antworten auf ein ‚Remote Transmission Request‘, indem eine Botschaft speziell für diesen Zweck im Device-Treiber bereitgestellt wird. Die bereitgestellte Nachricht wird vom Treiber automatisch versandt, wenn eine RTR-Message empfangen wird.

Der CAN-Device-Treiber erwartet als Argument eine CAN-Botschaft im vorgegebenen Format. Das erste Byte wird als Frame-Format-Byte interpretiert. Je nach Frame-Format sind die nächsten 2 oder 4 Bytes der Identifier der Nachricht. Eine typische CAN-Ausgabe als Standard Frame hat folgendes Aussehen:

PUT #CAN, #0, “<Frame-Format><ID1><ID2>data”

<Frame-Format>	enthält Information, daß es ein Standard-Frame ist.
<ID1>	enthält die oberen Bits 3...10 des Identifiers.
<ID2>	enthält die unteren Bits 0...2 des Identifiers an den Bitpositionen 5, 6 und 7. Die übrigen Bits in diesem Byte sind bedeutungslos.
data	sind Datenbytes, die in der Botschaft übertragen werden. 0...8 Datenbytes sind möglich.

Bei 0..8 Datenbytes entsteht daraus eine CAN-Botschaft. Wenn mehr als 8 Datenbytes enthalten sind, dann verpackt der Device-Treiber Die Daten in mehrere CAN-Botschaften und verwendet den gleichen Identifier.

Aus

PUT #CAN, #0, “<Frame-Format><ID1><ID2>abcdefghijklmnopqrs”

werden folgende CAN-Botschaften:

“<Frame-Format><ID1><ID2>abcdefgh”

Device-Treiber

“<Frame-Format><ID1><ID2>ijklmnop”

“<Frame-Format><ID1><ID2>qrs”

Werden die Daten über die Sekundär-Adresse 1 verschickt, dann wird in der Botschaft das RTR-Bit gesetzt und dadurch ein ‚Remote Transmission Request‘ erzeugt.

Eine Einzel-Nachricht mit maximal 8 Datenbytes an Sekundär-Adresse 2 hinterlegt eine Antwort, die dann versandt wird, wenn der Device-Treiber selbst ein ‚Remote Transmission Request‘ empfängt.

Sek.-Adr.	Funktion
0	Normaler Datenversand
1	Datenversand mit ‚Remote Transmission Request‘
2	Hinterlegen einer Antwort-Botschaft, die dann versandt wird, wenn der Device-Treiber selbst ein ‚Remote Transmission Request‘ empfängt.

Ein einfaches Sendebispiel für **Standardframe**-CAN-Botschaften zeigt das folgende Programm.

Programmbeispiel:

```

-----
' Name: CAN_TXS.TIG
' sendet 'the quick brown fox' ueber den CAN-Bus in Standard-Frames
' Verbinde ein empfangendes CAN-Gerät, z.B. einen CAN-Tiger
-----
user var strict                ' unbedingte Var.deklaration
#include UFUNC3.INC            ' User Function Codes
#include DEFINE_A.INC          ' allg. Symbol-Definitionen
#include CAN.INC                ' CAN-Definitionen
-----

TASK MAIN
  BYTE ever, i_msg, can_stat
  WORD obu_free                ' Platz im Output-Puffer
  WORD t_id                    ' Transmit ID
  STRING data$, msg$(11)

  install_device #LCD, "LCD1.TDD" ' LCD-Treiber installieren
  install_device #CAN, "CAN1_K1.TDD", & ' CAN-Treiber installieren
    "50 A0 00 00 &                ' access code
    FF FF FF FF &                ' access mask
    10 45 &                        ' bustim1, bustim2
    08 1A"%                        ' single filter mode, outctrl

  data$ = "the quick brown fox jumps over the lazy dog"
  i_msg = 0                      ' Index fuer laufenden Text
  t_id = 155h shl 5              ' Standard Identifier

  for ever = 0 to 0 step 0      ' Endlosschleife
    get #CAN, #0, #UFCI_OBU_FREE, 0, obu_free
    print #LCD, "<1Bh>A<0><1><0F0h>OBU_FREE: "; obu_free, " ";
    if obu_free > 11 then
      msg$ = & ' Frame-Info 0 = standard, 2 ID-Bytes, data
      "<0><0><0>" + mid$ ( data$, i_msg, 8 )' Frame-Info, ID      frame i
      msg$ = ntos$ ( msg$, 1, -2, t_id ) ' ID high-Byte zuerst einb
      print #CAN, #0, msg$;          ' sende Message im Standard-Frame
      i_msg = i_msg + 1             ' String-Index eins weiter
      if i_msg > len(data$)-8 then ' Achtung Limit
        i_msg = 0
      endif
    endif
                             ' pruefe CAN Status
    get #CAN, #0, #UFCI_CAN_STAT, 0, can_stat
    using "UH<2><2> 0 0 0 2" ' HEX-Format fuer ein Byte
    print_using #LCD, "<1Bh>A<0><0><0F0h>CAN-State: "; can_stat;
    wait_duration 200
  next
END

```

Ein einfaches Sendebispiel für **extended Frame-CAN**-Botschaften zeigt das folgende Programm.

Programmbeispiel:

```

-----
' Name: CAN_TXE.TIG
' sendet 'the quick brown fox' ueber den CAN-Bus in extended Frames
' Verbinde ein empfangendes CAN-Gerät, z.B. einen CAN-Tiger
-----
user var strict                ' unbedingte Var.deklaration
#include UFUNC3.INC             ' User Function Codes
#include DEFINE_A.INC          ' allg. Symbol-Definitionen
#include CAN.INC                ' CAN-Definitionen
-----

TASK MAIN
  BYTE ever, i_msg, can_stat
  WORD obu_free                ' Platz im Output-Puffer
  LONG t_id                    ' extended ID 4 Bytes
  STRING data$, msg$(13)

  install_device #LCD, "LCD1.TDD" ' LCD-Treiber installieren
  install_device #CAN, "CAN1_K1.TDD", & ' CAN-Treiber installieren
    "50 A0 00 00 &                ' access code
    FF FF FF FF &                ' access mask
    10 45 &                        ' bustim1, bustim2
    08 1A"%                        ' single filter mode, outctrl

  data$ = "the quick brown fox jumps over the lazy dog"
  i_msg = 0                      ' Index fuer laufenden Text
  t_id = 01733F055h shl 3        ' extended Identifier

  for ever = 0 to 0 step 0      ' Endlosschleife
    get #CAN, #0, #UF0CI_OBU_FREE, 0, obu_free
    print #LCD, "<1Bh>A<0><1><0F0h>OBU_FREE:";obu_free;" ";
    if obu_free > 13 then
      msg$ = & ' Frame-Info 80h = extended, 4 ID-Bytes, data
      "<80h><0><0><0><0>" + mid$ ( data$, i_msg, 8 )
      msg$ = ntos$ ( msg$, 1, -4, t_id ) ' ID high-Byte zuerst einb
      print #CAN, #0, msg$;        ' sende Message im extended-Frame
      i_msg = i_msg + 1           ' String-Index eins weiter
      if i_msg > len(data$)-8 then ' Achtung Limit
        i_msg = 0
      endif
    endif
    ' pruefe CAN Status
    get #CAN, #0, #UF0CI_CAN_STAT, 0, can_stat
    using "UH<2><2> 0 0 0 0 2" ' HEX-Format fuer ein Byte
    print_using #LCD, "<1Bh>A<0><0><0F0h>CAN-State:";can_stat;
    wait_duration 200
  next
END

```


Empfangen von CAN-Botschaften

Der CAN-Device-Treiber empfängt CAN-Botschaften und legt diese im Empfangspuffer ab. Das Auslesen des Empfangspuffers bei dem CAN-Device-Treiber ist ein besonderer Vorgang und unterscheidet sich vom Auslesen anderer Puffer (z.B. des seriellen oder parallelen Treibers), da hier Botschaften im Puffer vorliegen, die außer den Daten weitere Informationen enthalten. Die Botschaften werden immer vollständig gelesen und entsprechend ihrem Messagetyp verarbeitet:

Zwei Lesemodi lesen unterschiedlich von den Sekundär-Adressen 0 und 1:

Sek.Adr.	
0	Die Bytes der CAN-Botschaften werden so gelesen, wie sie im Puffer vorliegen, inklusive Frame-Format- und ID-Bytes.
1	Nur Datenbytes werden gelesen. Frame-Format- und ID-Bytes werden ignoriert. Die Längeninformation von teilweise eingelesenen CAN-Botschaften wird im Puffer automatisch korrigiert.

Achtung: Von der Sekundär-Adresse 0 müssen die CAN-Botschaften vollständig gelesen werden, da sonst der nächste Lesevorgang nicht mit dem Frame-Info-Byte der nächsten CAN-Botschaft beginnt.

Über die Sekundär-Adresse 0 erfolgt das **Auslesen einzelner Botschaften**, die 0..8 Zeichen enthalten und deren Frameformatkennung und Identifier den Datenbytes vorangestellt sind. Zunächst wird das Frame-Info-Byte gelesen und festgestellt, ob es sich um einen ‚Standard-Frame‘ oder einen ‚extended Frame‘ handelt und wieviele Datenbytes enthalten sind. Dann werden die ID-Bytes gelesen, die den anwendungsspezifischen Botschaftstyp darstellen. Anschließend werden die Datenbytes eingelesen.

Das Beispielprogramm CAN_RX1.TIG liest die empfangenen Nachrichten aus dem Puffer, unterscheidet dabei Standardframes und extended Frames und zeigt diese in hexadezimaler Form an.

Programmbeispiel:

```

-----
' Name: CAN_RX1.TIG
' empfaengt ungefiltert CAN-Messages und zeigt sie auf LCD an
' unterscheidet Standard und extended Frame
' zeigt auch Status an
' Verbinde ein sendendes CAN-Geraet, z.B. einen CAN-Tiger
-----
user var strict          ' unbedingte Var.deklaration
#include UFUNC3.INC      ' User Function Codes
#include DEFINE A.INC    ' allg. Symbol-Definitionen
#include CAN.INC         ' CAN-Definitionen

-----
TASK MAIN
BYTE ever, frameformat, msg_len, can_stat
WORD ibu_fill          ' Eingangspufferfuellung
LONG r_id              ' empfangene ID
STRING msg$(8), data$(8)

install_device #LCD, "LCD1.TDD" ' LCD-Treiber installieren
install_device #CAN, "CAN1_K1.TDD", & ' CAN-Treiber installieren
"50 A0 00 00 &          ' access code
FF FF FF FF &          ' access mask
10 45 &                ' bustim1, bustim2
08 1A"%                ' single filter mode, outctrl

print #LCD, "<1Bh>A<0><0><0F0h>STAT LEN ID";

for ever = 0 to 0 step 0          ' Endlosschleife
get #CAN, #0, #UFICI_IBU_FILL, 0, ibu_fill
print #LCD, "<1Bh>A<0><3><0F0h>IBU_FILL: ";ibu_fill; " ";
if ibu_fill > 2 then              ' wenn mindestens eine Message
get #CAN, #0, 1, frameformat ' hole Frame-Info-Byte
msg_len = frameformat bitand 1111b ' Laenge
if frameformat bitand 80h = 0 then ' wenn Standard-Frame
get #CAN, #0, CAN_ID11_LEN, r_id ' hole ID-Bytes
r_id = byte_mirr ( r_id, 2 )      ' Bytefolge f. Tiger WORD
r_id = r_id shr 5                ' rechtsbuendig schieben
using "UH<8><3> 0 0 0 0 3"        ' fuer ID Anzeige
else                               ' sonst ist es extended frame
get #CAN, #0, CAN_ID29_LEN, r_id ' hole ID-Bytes
r_id = byte_mirr ( r_id, 4 )      ' low byte 1st in LONG
r_id = r_id shr 3                ' rechtsbuendig schieben
using "UH<8><8> 0 0 0 4 4"        ' fuer ID Anzeige
endif
print using #LCD, "<1Bh>A<9><1><0F0h>";r_id;

using "UH<1><1> 0 0 0 0 1"        ' zeige Laenge an
print using #LCD, "<1Bh>A<6><1><0F0h>";msg_len;
if msg_len > 0 then              ' wenn Daten
get #CAN, #0, msg_len, data$      ' hole sie und zeige an
msg$ = " "                        ' 8 Leerzeichen
msg$ = stos$ ( msg$, 0, data$, msg_len ) ' LCD-Feld vorbereiten
print #LCD, "<1Bh>A<0><2><0F0h>data: ";msg$;
else
print #LCD, ; " " ;
endif
endif

get #CAN, #0, #UFICI_CAN_STAT, 0, can_stat ' CAN-Status

```

```
    print_using #LCD, "<1Bh>A<1><1><0F0h>";can_stat;
next
END
```

Device-Treiber

Über die Sekundär-Adresse 1 erfolgt das **Auslesen von Daten** ohne Berücksichtigung der Frame-Format- und Identifier-Bytes. Der Device-Treiber liest nur die Datenbytes und ignoriert die Identifier. Unvollständig gelesene CAN-Botschaften behalten das Frameformat und ihre ID-Bytes, die Länge wird vom Treiber entsprechend korrigiert, so daß der nächste Lesevorgang wieder eine intakte CAN-Nachricht im Puffer vorfindet.

2

Programmbeispiel:

```

'-----
' Name: CAN_RX2.TIG
' empfaengt CAN-Daten und zeigt sie an, ignoriert IDs
' zeigt Daten als Text an (nur ASCII senden)
' zeigt auch Status an
' Verbinde ein sendendes CAN-Geraet, z.B. einen CAN-Tiger
'-----
user var strict                ' unbedingte Var.deklaration
#include UFUNC3.INC            ' User Function Codes
#include DEFINE_A.INC          ' allg. Symbol-Definitionen
#include CAN.INC                ' CAN-Definitionen
'-----

TASK MAIN
  BYTE ever, frameformat, msg_len, can_stat
  WORD ibu_fill                ' Ausgangspufferfuellung
  LONG r_id
  STRING id$(4), data$, line$

  install_device #LCD, "LCD1.TDD" ' LCD-Treiber installieren
  install_device #CAN, "CAN1_K1.TDD", & ' CAN-Treiber installieren
    "50 A0 00 00 &                ' access code
    FF FF FF FF &                  ' access mask
    10 45 &                          ' bustim1, bustim2
    08 1A"%                          ' single filter mode, outctrl

  print #LCD, "<1Bh>A<0><0><0F0h>STAT LEN ID";

  line$ = ""
  for ever = 0 to 0 step 0        ' Endlosschleife
    get #CAN, #0, #UFICI_IBU_FILL, 0, ibu_fill
    print #LCD, "<1Bh>A<0><3><0F0h>IBU_FILL: ";ibu_fill;";
    get #CAN, #1, 0, data$
    if data$ <> "" then
      line$ = line$ + data$
      if len(line$) > 20 then      ' wenn laenger als LCD-Zeile
        line$ = right$ ( line$, 20 )
      endif
      print #LCD, "<1Bh>A<0><2><0F0h>";line$;
    endif
    get #CAN, #0, #UFICI_CAN_STAT, 0, can_stat
    using "UH<2><2>  0 0 0 2" ' HEX-Format fuer ein Byte
    print_using #LCD, "<1Bh>A<1><1><0F0h>";can_stat;
  next
END

```

Empfang eines ‚Remote Transmission Request‘, führt dazu, daß eine Botschaft, die speziell für diesen Zweck im Device-Treiber bereitgestellt wurde, verschickt wird. Die empfangene CAN-Botschaft wird ansonsten so behandelt, wie eine CAN-Botschaft ohne ‚Remote Transmission Request‘.

Device-Treiber

Programmbeispiel:

2

```
'-----
' Name: CAN_RTR.TIG
' bereitet eine RTR-Message vor und sendet dann CAN-Nachrichten
' in einer Schleife.
' RTR und Schleifen-Nachrichten haben unterschiedliche ID.
' Verbinde ein CAN-Gerät, welches mit einer RTR-Message die Antwort
' abrufen, z.B. einen CAN-Tiger mit CAN_RTRS.TIG
'-----
user var strict          ' unbedingte Var.deklaration
#include UFUNC3.INC      ' User Function Codes
#include DEFINE A.INC    ' allg. Symbol-Definitionen
#include CAN.INC         ' CAN-Definitionen
'-----
TASK MAIN
  BYTE ever              ' Endlosschleife
  STRING rtr_msg$(13)

  install_device #LCD, "LCD1.TDD" ' LCD-Treiber installieren
  install_device #CAN, "CAN1_K1.TDD", & ' CAN-Treiber installieren
  "50 A0 00 00 &          ' access code
  FF FF FF FF &          ' access mask
  10 45 &                 ' bustim1, bustim2
  08 1A"%                 ' single filter mode, outctrl

  rtr_msg$ = "<0><0FFh><0E0h>RTR-resp" ' RTR Antwort als Standard-Frame
  put #CAN, #2, rtr_msg$           ' in Treiber hinterlegen
  print #LCD, "RTR-message prepared"

  for ever = 0 to 0 step 0        ' Endlosschleife
    wait duration 3000
    put #CAN, #0, "<0><0FFh><0C0h>abcdefgh"
    wait duration 3000
    put #CAN, #0, "<0><0FFh><080h>ijklmnop"
  next
END
```

Ein- und Ausgangspuffer

CAN-Botschaften bestehen aus einem Frame-Format-Byte, einem Identifier und maximal 8 Datenbytes. Im Falle eines ‚Standard-Frames‘ belegt der Identifier 2 Bytes. Beim ‚extended Frame‘ ist der Identifier 4 Bytes lang. Jede Botschaft wird zusammen mit dem Frame-Format-Byte und dem Identifier im Puffer abgelegt. Wenn eine Botschaft nicht mehr in den Puffer paßt, dann wartet beim Senden die PUT-Instruktion, bis wieder Platz im Puffer ist. Beim Empfangen wird die Botschaft verworfen und ein Overflow-Fehler registriert.

Anzahl Datenbytes	belegt im Puffer	
	Standard Frame	extended Frame
0	3	5
8	11	13

Anmerkung: wird mit einer einzigen PUT-Instruktion ein String in den Puffer übertragen, der mehr als 8 Datenbytes enthält, dann wird Platz für zusätzliche Identifier benötigt, da die Daten auf mehrere CAN-Botschaften verteilt werden.

Sowohl eingehende als auch gesendete Daten werden in einem Puffer zwischengespeichert. Größe, Füllstand oder verbleibender Platz der Ein- und Ausgangspuffer sowie die Version des Treibers können mit Hilfe der User-Function-Codes abgefragt werden.

Sowohl bei der Ausgabe als auch beim Empfang gilt ein Puffer als voll, sobald weniger als 13 Bytes frei sind. Eine CAN-Nachricht im Extended-Frame-Format ist 13 Bytes lang. Da keine halbe CAN-Nachrichten gespeichert werden können, gilt diese Grenze.

Device-Treiber

User-Function-Codes für Abfragen (Instruktion GET):

Nr	Symbol Prefix UFCI_	Beschreibung
1	UFCI_IBU_FILL	Füllstand des Eingangspuffers (Byte)
2	UFCI_IBU_FREE	freier Platz im Eingangspuffer (Byte)
3	UFCI_IBU_VOL	Größe des Eingangspuffers (Byte)
33	UFCI_OBU_FILL	Füllstand des Ausgangspuffers (Byte)
34	UFCI_OBU_FREE	freier Platz im Ausgangspuffer (Byte)
35	UFCI_OBU_VOL	Größe des Ausgangspuffers (Byte)

2

Wenn nicht genügend Platz im Ausgabepuffer ist und trotzdem ausgegeben wird, dann wartet die Instruktion PUT oder PRINT (und damit die ganze Task) solange, bis wieder Platz im Puffer ist. Dieses Warten kann verhindert werden, indem vor der Ausgabe der freie Platz im Puffer abgefragt wird.

Beispiel: gebe nur aus, wenn noch genügend freier Platz im Ausgangspuffers ist:

```
GET #CAN, #0, #UFCI_OBU_FREE, 0, wVarFree
IF wVarFree > (LEN(A$)) THEN
  PUT #CAN, #0, A$
ENDIF
```

Beispiel: prüfe, ob eine Nachricht im Eingangspuffers ist (die kürzeste mögliche Nachricht ist 3 bytes lang):

```
GET #CAN, #0, #UFCI_IBU_FILL, 0, wVarFill
IF wVarFill > 2 THEN
  ` lies die CAN-Nachricht
ENDIF
```


Automatische Bitratenerkennung

Wird der Treiber im Modus ‚listen-only‘ installiert, dann versucht er, die Bitrate automatisch zu erkennen. Durch den Modus ‚listen-only‘ kann der CAN-Chip selbst nichts senden, und so werden auch nicht die sonst üblichen Fehlertelegramme erzeugt, solange die Bitrate noch nicht erkannt ist. Welche Bitraten erkannt werden können, wird durch eine Tabelle vorgegeben. Wird bei der Installation keine Tabelle übergeben, dann wird eine intern vorhandene Tabelle verwendet.

Um die Bitrate zu erkennen, sind folgende Voraussetzungen zu erfüllen:

- Es wird ein funktionierender Bus mit Datenverkehr vorausgesetzt, daß heißt, es müssen mindestens zwei aktive Teilnehmer vorhanden sein, die etwas senden.
- Die richtige Bitrate muß in der Tabelle enthalten sein.

Die Bitratenerkennung beginnt mit der ersten Einstellung aus der Tabelle, in der Regel die höchste mögliche Bitrate. Beim nächsten Datenpaket auf dem CAN-Bus tritt kein Empfangsfehler auf, wenn die Bitrate bereits stimmt. Tritt jedoch ein Empfangsfehler auf, dann schaltet der Treiber auf die nächste Bitrate der tabelle um, und wartet erneut auf ein CAN-Telegramm. Der Treiber wartet in jedem Fall bis genügend CAN-Telegramme entweder ein Erkennen der Bitrate ermöglicht haben, oder die Tabelle der möglichen Werte dreimal abgearbeitet ist. Wurde die Bitrate nicht erkannt, ist der CAN-Device-Treiber anschließend nicht installiert. Werden nur sehr selten CAN-Telegramme über den Bus geschickt und die richtige Bitrate befindet sich erst am Ende der Tabelle, dann dauert die Erkennung entsprechend lange. Wurde die Bitrate schließlich erkannt, verläßt der Device-Treiber den Modus ‚listen-only‘.

Device-Treiber

Die Tabelle enthält die Einstellungen für die Register ‚bustim0‘ und ‚bustim1‘ im CAN-CHIP. Für jede Einstellung werden also 2 Bytes benötigt. Mindestens 4 Bytes müssen in der Tabelle enthalten sein, sonst wird auf die interne Tabelle zurückgegriffen, die folgende Werte enthält:

2

bustim0	bustim1	Bitrate
0	43h	1 Mbit
0	5Ch	500 kBit
1	5Ch	250 kBit
3	5Ch	125 kBit
4	5Ch	100 kBit
9	5Ch	50 kBit
10h	19h	45.2 kBit
0Fh	7Fh	20 kBit
1Fh	7Fh	12.5 kBit

Programmbeispiel:

```

'-----
' Name: CAN_ABR.TIG
' sucht automatisch die richtige Bitrate aus vorgegebener Tabelle
' sonst wie CAN_RX1.TIG
' Verbinde mit einem CAN-Bus mit Sendeverkehr
'-----
user var strict           ' unbedingte Var.deklaration
#include UFUNC3.INC       ' User Function Codes
#include DEFINE_A.INC     ' allg. Symbol-Definitionen
#include CAN.INC          ' CAN-Definitionen
'-----

TASK MAIN
BYTE ever, frameformat, msg_len, can_stat
WORD ibu_fill           ' Eingangspufferfuellung
LONG r_id               ' empfangene ID
STRING msg$(8), data$(8)

install_device #LCD, "LCD1.TDD" ' LCD-Treiber installieren
print #LCD, "trying to find <10><13>CAN bitrate.<10><13>Please wait..."
install_device #CAN, "CAN1_K1.TDD", & ' CAN-Treiber installieren
"50 A0 00 00 & ' access code
FF FF FF FF & ' access mask
00 00 & ' bustim1, bustim2
0A 1A & ' single filter + listen only, outctrl
00 43 & ' 1 Mbit ab hier Tabelle mit Bytes
00 5C & ' 500 kbit fuer bustim0 und bustim1
01 5C & ' 250 kbit fuer automatische
03 5C & ' 125 kbit Bitratenerkennung
04 5C & ' 100 kbit
09 5C & ' 50 kbit
10 45 & ' 49 kbit for SLIO: TSYNC + TSEG1 + TSEG2 = 10
0F 7F & ' 25 kbit
1F 7F"% ' 12.5 kbit

print #LCD, "<1>STAT LEN ID";

for ever = 0 to 0 step 0 ' Endlosschleife
get #CAN, #0, #UFCI_IBU_FILL, 0, ibu_fill
print #LCD, "<1Bh>A<0><3><0F0h>IBU_FILL:";ibu fill," ";
if ibu_fill > 3 then ' wenn mindestens eine Message
get #CAN, #0, 1, frameformat ' welches Frame-Format?
msg_len = frameformat bitand 1111b
if frameformat bitand 80h = 0 then ' wenn Standard-Frame
get #CAN, #0, CAN_ID11_LEN, r_id ' hole ID-Bytes
r_id = byte_mirr ( r_id, 2 ) ' Bytefolge f. Tiger WORD
r_id = r_id shr 5 ' rechtsbueendig schieben
using "UH<8><3> 0 0 0 0 3" ' fuer ID Anzeige
else ' sonst ist es extended frame
get #CAN, #0, CAN_ID29_LEN, r_id ' hole ID-Bytes
r_id = byte_mirr ( r_id, 4 ) ' low byte 1st in LONG
r_id = r_id shr 3 ' rechtsbueendig schieben
using "UH<8><8> 0 0 0 4 4" ' fuer ID Anzeige
endif
print_using #LCD, "<1Bh>A<9><1><0F0h>";r_id;

using "UH<1><1> 0 0 0 0 1" ' zeige Laenge an
print_using #LCD, "<1Bh>A<6><1><0F0h>";msg_len;

```

Device-Treiber

2

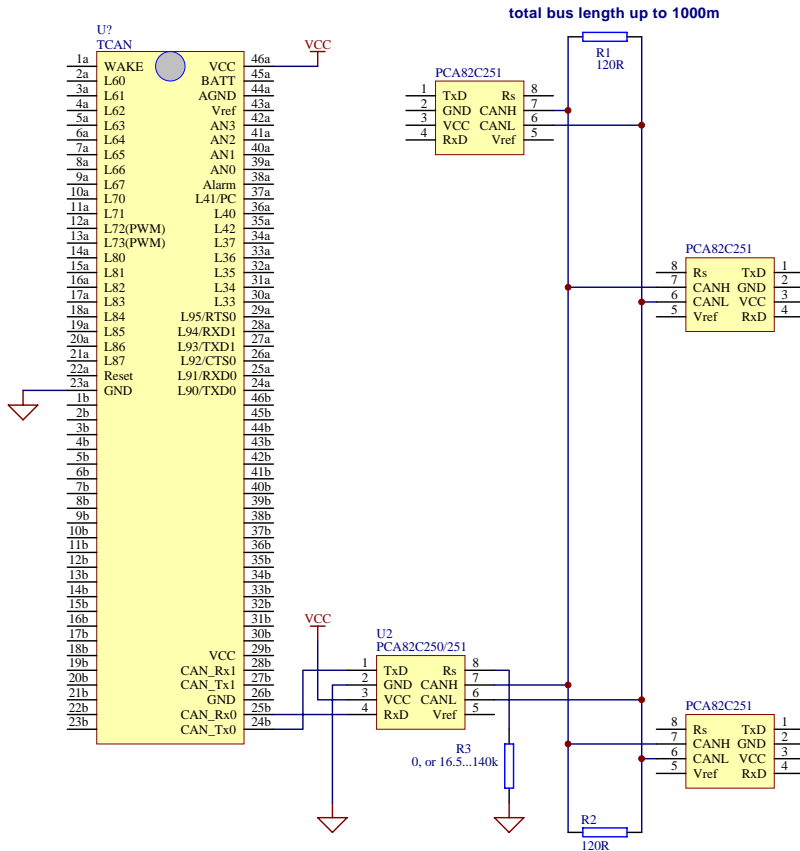
```
    get #CAN, #0, msg_len, data$      ' hole sie und zeige an
    msg$ = "          "              ' 8 Leerzeichen
    msg$ = stos$ ( msg$, 0, data$, msg_len )' LCD-Feld vorbereiten
    print #LCD, "<1Bh>A<0><2><0F0h>data:";msg$;
  else
    print #LCD, ;" RTR          ";
  endif
endif

get #CAN, #0, #UFCI_CAN_STAT, 0, can_stat ' CAN-Status
using "UH<2><2>  0 0 0 0 2" ' HEX-Format fuer ein Byte
print_using #LCD, "<1Bh>A<1><1><0F0h>";can_stat;
next
END
```

CAN-Bus Hardware-Anschlußbeispiel

In der Hardware sollte an jedem Leitungsende ein Abschlußwiderstand von 120 Ohm angebracht werden. Auf dem TCAN-Adapter befindet sich für diesen Zweck ein DIP-Schalter, der den Abschlußwiderstand zu- oder abschaltet.

2



Beachten Sie die Terminierung des Busses mit 120Ohm-Widerständen.

Eine kurze Einführung zu CAN

CAN ist eine Abkürzung für Controller Area Network. Ursprünglich wurde CAN als Kommunikationsprotokoll zum Informationsaustausch in Kraftfahrzeugen entwickelt. Mittlerweile ist CAN ebenso verbreitet in der Automatisierungstechnik und Haustechnik zu finden. Grundlage für den CAN-Bus ist eine Hardware, die den Anschluss an den CAN-Bus herstellt und den eigentlichen Nachrichtenversand und Nachrichtenempfang vornimmt, ähnlich einer UART bei der RS-232-Schnittstelle, jedoch bereits mit Prüfsummen, Fehlerkontrolle und Wiederholung der Nachrichten im Fehlerfall sowie Bus-Arbitrierung und Bus-Priorisierung. Es gibt eine Vielzahl von Herstellern, die CAN-Schnittstellen auf ihren Prozessoren implementiert haben, und es gibt externe CAN-Chips, die an Prozessoren angeschlossen werden können, die keine CAN-Schnittstelle ‚on-board‘ haben.

Auf dem CAN-Bus werden kompakte Datenpakete versandt, im folgenden CAN-Botschaften genannt. Eine Botschaft besteht aus Anwendersicht aus einem Identifier und 0 bis 8 Datenbytes. Es gibt zwei Varianten des Bit-Protokolls auf dem Bus gemäß CAN 2.0A mit **11-Bit-Identifier** und gemäß CAN 2.0B mit **29-Bit-Identifier**. Beide Varianten existieren nebeneinander, beide haben jeweils ihre Vor- und Nachteile. Moderne Chips unterstützen entweder CAN2.0B oder akzeptieren zumindest das Vorhandensein von 29-Bit-Identifiern auf dem Bus (CAN2.0B passiv).

Buszugriffe und Zugriffs-Prioritäten sind durch die CAN-Spezifikation festgelegt und werden vollständig von der CAN-Hardware bewältigt. Die Anwendungs-Software legt die CAN-Botschaft mit einem ‚Aufkleber‘ in den CAN Sende-Briefkasten. Der Aufkleber, Identifier genannt, ist jedoch kein Adress-Aufkleber, sondern eine Identifikation des Inhalts der CAN-Botschaft, z.B. ‚die Temperaturinformation von Sensor A‘, oder ‚die Stellinformation für Druckregler X‘. Jeder Busteilnehmer, für dessen Anwendung die Botschaft wichtig ist, wird darauf programmiert sein, sie aufzunehmen. Der Absender kann nicht feststellen, ob irgendein anderer Knoten die Botschaft aufgenommen hat.

Ein **Empfangsfilter** in der CAN-Hardware filtert die Botschaften nach bestimmten Kriterien vor, so dass nicht alle Botschaften zu der Anwendung gelangen. Im Empfangsteil liegen die größten Unterschiede bei den verschiedenen Implementationen von CAN-Hardware. Sowohl die Art der Filterung als auch die Anzahl der Botschaften, die im Empfangsbriefkasten gespeichert werden, sind sehr unterschiedlich. Es wird versucht, nur die Botschaften durch den Filter hindurchzulassen, welche für die Anwendung interessant sind.

Auf dem CAN-Bus können sogenannte ‚**Remote Transmission Requests**‘ verschickt werden. Damit werden die entsprechenden Busteilnehmer aufgefordert, mit einer spezifischen Botschaft zu antworten. So kann zum Beispiel die Aufforderung auf dem Bus erscheinen, die ‚Temperatur Kessel 2‘ zu melden. Die Anwendungen in den einzelnen CAN-Knoten legen fest, ob auf solche Sende-Aufforderungen geantwortet werden kann und welchen Inhalt die Antwort hat.

Die **Buszugriffe** finden in einem festen Zeitraster statt. Mit jedem Buszugriff synchronisieren sich alle Busteilnehmer. Die Zugriffe finden zur gleichen Zeit statt.

Der Ruhepegel auf dem Bus ist die ,1'. Dieser Pegel ist der nicht dominante. Eine ,1' kann durch eine ,0' überschrieben werden, daher der Ausdruck ,dominant' für die ,0'. Ein Buszugriff wird mit einer **dominanten ,0'** begonnen. Danach folgen die ,1' und ,0' Pegel des Identifiers, angefangen mit dem höchstwertigen Bit. Die niedriger priorisierten Busteilnehmer haben in den höherwertigen Bitstellen ,1'-Bits und können daher von den priorisierten Busteilnehmern mit einer ,0' überschrieben werden. Sobald ein Teilnehmer bei einem Buszugriff seine ,1' nicht platzieren konnte, bricht er den Buszugriff ab, um es später noch mal zu versuchen. Dieser erneute Versuch wird von der CAN-Hardware automatisch vorgenommen und braucht nicht in der Anwendung programmiert zu werden, die gar nichts davon weiß. Erst wenn ein Buszugriff über eine Anzahl Versuche hinweg nicht möglich ist, der Bus also scheinbar ständig von dominanteren Teilnehmern belegt ist, wird es der Anwendung möglich, über Abfrage von Fehlerregistern der CAN-Hardware diesen Zustand zu erkennen.

Hier noch einmal in einer Gegenüberstellung die prägnantesten Unterschiede zu den meisten anderen Netzen und Bussystemen:

Die meisten anderen Industrie-Bussysteme	CAN-Bus
Jeder Teilnehmer erhält eine Adresse und Nachrichten werden zusammen mit einer Zieladresse, manchmal auch einer Absenderadresse versehen.	Es gibt keine Adressen. Die Nachrichten sind mit einer Inhaltserklärung anstelle der Adresse versehen. Die Teilnehmer haben programmierbare Eingangsfiler, die bestimmte Nachrichten hindurchlassen.
Oft ist eine Empfangsbestätigung vorgesehen. Der Empfänger bestätigt dann den Korrekten Empfang der Sendung.	Die CAN-Hardware bestätigt am Ende eines Nachrichtenpaketes, dass dieses korrekt auf dem Bus erschienen ist (Acknowledge). Ob irgendein Teilnehmer die Nachricht auch aufgenommen hat, ist unbekannt.
Es existieren Regeln für den Buszugriff, so dass nie zwei Teilnehmer den Bus gleichzeitig benutzen.	Bei CAN können mehrere Teilnehmer gleichzeitig auf den Bus zugreifen. Im Laufe des Zugriffs verdrängt der priorisierteste Teilnehmer die anderen, die automatisch später noch mal auf den Bus zugreifen. Der Buszugriff wird vollständig von der CAN-Hardware bewältigt.

Device-Treiber

Besonderheiten des BASIC-Tiger[®]-CAN-Moduls

An das BASIC-Tiger[®]-CAN-Modul können andere Module oder Einheiten direkt angeschlossen werden, wenn es sich um kurze Entfernungen handelt, etwa auf der gleichen Platine oder zumindest im gleichen Gehäuse. Bei größeren Entfernungen ist ein externer Bustreiber erforderlich (z.B. PCA82C250).

Die CAN-Hardware wird durch den CAN-Device-Treiber unterstützt, der in Varianten vorliegt. Die Dateinamen haben folgende Bedeutung:

CAN_nn.TDD nn repräsentiert die Puffergröße
R1: 256 Byte
K1: 1 kByte
K8: 8 kByte

Bei der Installation können Parameter angegeben werden, die CAN-Botschaften-Filterungseigenschaften und das Bus-Timing festlegen. Zur Laufzeit können diese Parameter durch User-Function-Codes verändert werden.

Das BASIC-Tiger[®]-CAN-Modul unterstützt CAN2.0B, kann also Botschaften mit 29-Bit-Identifiern senden und empfangen. Botschaften werden in der regel zusammen mit dem Identifier in einem String vorbereitet und dann mit einer PUT- oder PRINT-Instruktion an den CAN-Treiber übergeben. Dieser puffert die Nachrichten, wenn nötig. Empfangene Botschaften werden im Empfangspuffer des Device-Treibers abgelegt, bis sie vom BASIC mit GET gelesen werden. Das spezielle Format der CAN-Botschaften ist in jedem Fall zu beachten.

2

Fehlersituationen

Im Folgenden werden einige Fehlersituationen aufgeführt, und es wird gezeigt, wie sie sich äußern.

Fehler-Erscheinung	Mögliche Ursache
<p>Auf dem Scope ist zu sehen: ein Teilnehmer sendet ständig und unablässig auf dem Bus, obwohl die Anwendung nur eine einzelne Nachricht absetzen wollte.</p>	<p>Der sendende Teilnehmer, besser: dessen CAN-Hardware, bekommt kein Acknowledge von einem anderen Busteilnehmer. Deswegen sendet die CAN-Hardware die Nachricht immer wieder erneut.</p> <p>Mögliche Gründe:</p> <p>Es befindet sich nur ein aktiver Teilnehmer am Bus. Die anderen sind entweder nicht vorhanden, ausgeschaltet oder nicht initialisiert.</p> <p>Die Bitrate dieses Teilnehmers stimmt nicht mit der Bitrate der anderen Busteilnehmer überein.</p>
<p>Botschaften, die sicher gesendet werden, kommen nicht an.</p>	<p>Es treten Empfangsfehler auf. Lassen Sie die Fehlerregister anzeigen, um Rückschlüsse auf den Fehler ziehen zu können.</p> <p>Sind die Fehlerregister in Ordnung, dann könnte es sein, daß die Filter die Identifier nicht passieren lassen.</p>
<p>Beim Senden werden sofort die Fehlerregister gesetzt.</p>	<p>Möglicherweise ist der Bus durch einen höher-priorisierten Teilnehmer ständig belegt (Überlastung), oder die Bitrate ist falsch.</p> <p>Ist etwa kein anderer Teilnehmer aktiv? Mindestens ein Busteilnehmer muß das ACK-Bit setzen.</p>

Literaturhinweise zu CAN

- 2**
- [1] Wolfgang Lawrenz: CAN Controller Area Network, Grundlagen und Praxis. Hüthig Verlag, 1994, ISBN 3-7785-2263-9
 - [2] Konrad Etschberger: CAN Controller Area Network, Grundlagen, Protokolle, Bausteine, Anwendungen. Verlag Hanser, 1994, ISBN 3-446-17596-2
 - [3] Bosch CAN Spezifikation Version 2.0 1991
 - [4] CiA: CAN in Automation e.V. Users Group, Am Weichselgarten 25, D-91058 Erlangen, Germany; Tel: +49 9131 601091, Fax: +49 9131 601092
 - [5] SJA1000 Datenbuch als PDF-datei im Internet:
<http://www-eu3.semiconductors.com/pip/SJA1000>
 - [6] P82C150 CAN-SLIO Datenbuch als PDF-datei im Internet:
<http://www-eu3.semiconductors.com/pip/P82C150>

In den Büchern finden Sie umfangreiche weitere Literaturangaben.

CAN-Board

Um die ersten Schritte in der CAN-Welt machen zu können, sind mindestens zwei, besser drei Busteilnehmer nötig. Sie haben also entweder

- sowieso schon Erfahrung mit CAN und andere Busteilnehmer greifbar. Sie kennen diese Geräte einigermaßen gut.
- Mindestens zwei TCAN-Module gekauft, einen Adapter für das Plug & Play Lab und eine andere Hardware-Plattform für das zweite TCAN-Modul aufbereitet.
- das TCAN-Entwicklungspaket gekauft, worin als weitere Busteilnehmer zwei Boards mit dem CAN-SLIO-Baustein von Philips enthalten sind.

2

In diesem Abschnitt finden Sie:

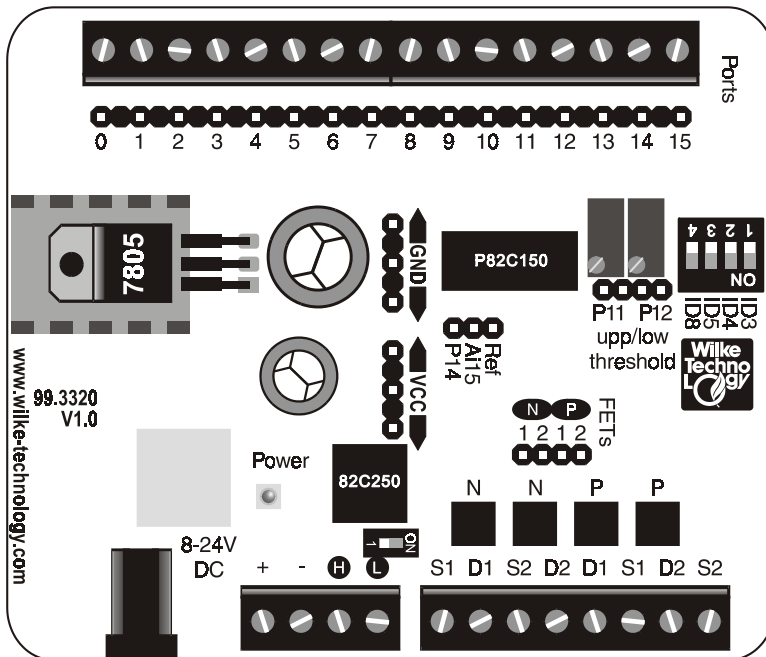
- CAN-SLIO-Board
- CAN-SLIO-Chip
- Identifier des SLIO
- Bitrate automatisch erkennen
- SLIO-Nachrichtenformat
- Statusbyte - Data-Byte 1
- SLIOs auf dem Bus finden
- Einige Besonderheiten für Interessierte
- Remote Frames
- Bit-Timing
- Oszillator und Kalibrierung
- Initialisierung
- Sign-On-Message
- Registerübersicht
- SLIO digitale Ein- und Ausgänge
- SLIO-Analog-Ausgänge
- Analog-Konfiguration
- Starten der A/D-Wandlung
- Zwei SLIOs an einem Bus

Device-Treiber

CAN-SLIO-Board

Das CAN-SLIO-Board ist eine I/O-Einheit, die über den CAN-Bus entfernt von der Steuereinheit seriell angebunden ist. Das Kernstück ist der CAN-SLIO-Chip (SLIO=Serial-Linked-I/O), der in den folgenden Abschnitten ausführlich beschrieben ist. Das Board enthält:

- ein Netzteil mit Spannungsregler
- den CAN-Treiber-Chip
- einen zuschaltbaren Abschlußwiderstand für den CAN-Bus
- einen DIP-Schalter, um den Identifier einzustellen
- Zwei Potis, um 2 analoge Spannungen einzustellen
- eine einfache Filterschaltung für die quasi-analogen Ausgänge
- 2 N-FETs und 2 P-FETs zur beliebigen Verwendung
- Schraubklemmen für 16 digitale I/O
- Schraubklemmen für den CAN-Bus
- Schraubklemmen für die Stromversorgung, falls die Buchse nicht verwendet wird



Technische Daten des CAN-SLIO-Boards:

Außenmaße / Gewicht:	ca.100 x 85 x 27 mm / ca. 75g
Stromversorgung	8V...24V / ca. 22 mA (unbelastete I/O)
Belastbarkeit der I/O-Pins	±4mA, alle Pins zusammen < 200mW
Reset	Power-ON-Reset auf dem Board durch R-C-Glied
Temperaturbereich	-40...+85°C
A/D-Eingang	Auflösung 6...7-Bit, bis zu 6 Kanäle über Analogschalter
D/A-Ausgänge	2 DPM (distributed pulse modulated) Auflösung 10-Bit Wiederholfrequenz 1024 Bitzeiten

Um das Board in Betrieb zu nehmen, schließen Sie es an einen CAN-Bus mit einem quartzkontrollierten Teilnehmer an (Schraubanschlüsse ‚H‘ und ‚L‘). Ein solcher Teilnehmer ist z.B. ein BASIC-Tiger[®]-CAN-Modul, welches auf dem CAN-Adapter des Plug & Play Labs Platz findet. Wenn sich das SLIO-Board am physikalischen Ende des Busses befindet, schalten Sie den DIP-Schalter neben der Schraubklemme auf ‚ON‘, um den Abschlußwiderstand zuzuschalten. Wenn sich das Board nicht am Bus-Ende befindet, schalten Sie den DIP-Schalter aus.

Verdrahten Sie weitere Hardware, die an I/O-Pins angeschlossen werden soll.

Stellen Sie an dem 4er-DIP-Schalter die gewünschten ID-Bits ein. Alle Busteilnehmer sollten einen eindeutigen Identifier haben, der nicht noch einmal im Bussystem vorkommt.

Laden Sie in das Modul TCAN-4/4 eines der Beispielprogramme. Das grundlegendste Beispielprogramm ist ‚SLIO_FIND1.TIG‘, welches das Finden des SLIO-Chips bewerkstelligt. Damit verbunden ist die automatische Bitratenerkennung und ständige Re-synchronisation des SLIO-Chips.

CAN-SLIO-Chip

Der CAN-SLIO-Chip P82C150 ist ein seriell angebundener I/O-Chip (SLIO=Serial-Linked-I/O), dessen Register über den CAN-Bus beschreibbar sind. Der SLIO-Chip unterstützt die Protokoll-Spezifikationen 2.0A und 2.0B (passive). Abstriche bei dem Bit-Timing bestehen aufgrund der automatischen Bitraten-Erkennung, sofern diese genutzt wird. In einem System mit P82C150-Knoten mit automatischer Bitraten-Erkennung muß sich mindestens ein aktiver CAN-Knoten mit Quarz befinden.

Auf dem CAN-SLIO-Board befindet sich ein SLIO-Chip, der die Bitrate selbst erkennen muß. Dies ist möglich im Bereich von Bitzeiten von 8µsec bis ca. 50µsec. Die folgenden Beispiele sind auf eine Bitzeit von ca. 22µsec eingestellt.

Identifizier des SLIO

Der CAN-SLIO verarbeitet 11-Bit-Identifizier. 29-Bit-Identifizier werden ignoriert. 4 Bits der Empfangsmaske werden nach dem Reset von den Portpins P0...P3 eingelesen. Diese Portpins sind danach als I/O-Pins verwendbar. Somit lassen sich bis zu 16 SLIO-Chips an einem Bus betreiben (bei Verwendung von externem Clock reduziert sich die Zahl auf 8). Jeder SLIO-Chip verwendet 2 Identifizier, die sich im untersten Bit unterscheiden. Die höhere Priorität ist für Nachrichtempfang reserviert (Bit ID.0=0).

ID.10	ID.9	ID.8	ID.7	ID.6	ID.5	ID.4	ID.3	ID.2	ID.1	ID.0
0	1	P3	1	0	P2	P1	P0	1	0	DIR

DIR 0: Nachrichten, die vom Host an den SLIO-CHIP gesendet werden (Schreibregister, Konfiguration).
 1: Nachrichten, die vom SLIO-CHIP an den Host gesendet werden (Leseregister, RTR).

P0, P1, P2, P3 Programmierbare ID-Bits, die nach dem Reset von den Pins P0...P3 gelesen werden.

Bitrate automatisch erkennen

Nach dem Einschalten oder einem Reset befindet der SLIO-Chip sich im Sleep-Modus. Der interne Oszillator ist gestoppt und alle Output-Treiber sind de-aktiviert. Beim ersten dominanten Bit auf dem Bus geht er in den ‚differential mode‘. Um den Chip auf dem CAN-Bus verfügbar zu machen, muß zunächst die Bitrate erkannt werden. Dazu benötigt der SLIO-Chip ein bestimmtes Bitmuster auf dem Bus. Eine bestimmte Nachricht mit festgelegtem Identifizier und festgelegten Datenbytes erzeugt dieses Bitmuster. Die gleiche Nachricht wird später verwendet, um den SLIO-Chip zu re-synchronisieren und damit aktiv zu halten. Spätestens nach 8000 Bitzeiten muß erneut eine re-synchronisation erfolgen, sonst geht der SLIO in wieder in einen

inaktiven Zustand über. Diese CAN-Nachricht wird in Tiger-BASIC® so zusammengesetzt:

```
calib_id = 0aah shl 5      ' ID der Kalibrier-Botschaften
calib$ = "<0><0><0><0aah><4>" ' spezieller String fuer Init-SLIO
calib$ = ntos$ ( calib$, 1, -2, calib_id ) ' ID high-Byte zuerst
```

Der Identifier ist AAh und die Datenbytes sind AAh und 4. Da der Identifier linksbündig in einem WORD bzw. 2 Bytes stehen muß, werden die 11 Bits um 5 nach links geschoben. In Tiger-BASIC® steht in einem WORD jedoch das low-Byte zuerst, in der Message wird aber das high-Byte zuerst benötigt. Die Funktion NTOS\$ dreht die Bytes beim Einbauen in den String durch die negativ angegebene Anzahl Bytes (-2). Diese Nachricht dient der Synchronisation, sie wird nicht von dem SLIO empfangen und sollte auch bei anderen Busteilnehmern nicht durch den Filter gelangen.

Die gewählte Bitrate sollte möglichst 10 Zeitsegmente enthalten oder dem nahe kommen, um die Bitratenerkennung zu erleichtern. Im Beispiel ist diese Bedingung erfüllt. Die Bitzeit sollte zwischen 8µsec und 50µsec liegen.

In den folgenden Beispielprogrammen ist der Filter der CAN-Hardware (Access-Code und Access-Mask) so eingestellt, daß möglichst nur die SLIO-Nachricht durchkommt. In einem System mit viel Datenverkehr würde dieses allererste einfache Beispielprogramm jede durchkommende Nachricht für eine SLIO-Nachricht halten.

Wenn die Bitrate erkannt ist, dann meldet sich der SLIO-Chip mit seiner ‚Sign-On‘-Nachricht. Diese hat das gleiche Format wie alle SLIO-Nachrichten und meldet den Inhalt des Daten-Input-Registers.

SLIO-Nachrichtenformat

Im Identifier jeder SLIO-Nachricht ist das Richtungsbit ‚DIR‘ auf 1 gesetzt, die Variablen Bits sind von den Ports P0...P3 eingelesen worden. Sowohl gesendete als auch empfangene CAN-Nachrichten enthalten 3 Datenbytes. Das erste Byte enthält die Registeradresse sowie Statusinformationen, die beiden anderen Bytes den Inhalt des angegebenen Registers. Nach jeder erfolgreich versendeten Nachricht verzögert der SLIO-Chip seine nächste eventuell anstehende Nachricht um 3 Bitzeiten, um niedriger priorisierten Knoten eine Sendemöglichkeit einzuräumen. Dies ist wichtig im Falle von Wackelkontakten an einem Pin, der Flankengetriggert eine CAN-Nachricht auslöst.

Die Statusinformationen im ersten Byte werden bei empfangenen Nachrichten ignoriert. Jede empfangene Nachricht wird durch eine neue Nachricht mit dem Inhalt des adressierten Registers bestätigt. (Ausnahme: siehe Analog-Konfiguration). Wenn der SLIO-Chip eine CAN-Nachricht empfangen hat, dann sendet er als Antwort eine Nachricht, die 4 Statusbits, die zuvor empfangene Registeradresse sowie den Inhalt

Device-Treiber

des Registers enthält. (Ausnahmen siehe A/D-Konfiguration). So werden Leseregister gelesen, und bei Schreibregistern wird der Schreibvorgang bestätigt.

Statusbyte - Data-Byte 1

Status				Register-Adresse			
RSTD	EW	BM1	BM0	A3	A2	A1	A0

RSTD	1: Sign-On-Nachricht 0: andere Nachrichten
EW	1: ‚error warning limit‘ (32) ist erreicht. Bit wird gesetzt, wenn der Receive Error Counter oder der Transmit Error Counter den Wert 32 überschritten hat. Immer 1 in der ‚Sign-On‘-Nachricht.
BM0, BM1	Busmode Statusbits (siehe unten).
A0...A3	Registeradresse. Bestimmt, welches interne SLIO-Register gelesen oder beschrieben wird.

Der Busmodus wird in den Statusbits BM0 und BM1 angezeigt:

Bus Mode	Bits		Reception Level		Transmission	
	BM1	BM0	recessive	dominant	Tx1	Tx0
0=differential	0	0	Rx0>Rx1	Rx0<Rx1	enabled	enabled
1=one wire Rx1	0	1	Rx1<REF	Rx1>REF	enabled	enabled
2=one wire Rx0	1	0	Rx0>REF	Rx0<REF	disabled	enabled
3=sleep	1	1	Rx0>REF and Rx1<REF	Rx0<REF and Rx1>REF	disabled	enabled

SLIOs auf dem Bus finden

Das Beispiel SLIO_FIND1.TIG geht davon aus, daß nur ein SLIO-Board angeschlossen ist. Es sollen erste Erkenntnisse über den SLIO gewonnen werden. Jede ankommende Nachricht wird für eine SLIO-Nachricht gehalten. Die Bytes werden in HEX so angezeigt, wie sie ankommen, so daß die oben erwähnten Bits leicht nachprüfbar sind. Die Adresse der SLIO ist an dem DIP-Schalter einstellbar. Nach dem Verstellen des DIP-Schalters muß das Board aus- und wieder eingeschaltet werden, denn der SLIO liest die ID-Bits nur beim Power-On bzw. nach einem Reset ein. Die Sign-On-Nachricht sieht z.B. so aus (alle DIP-Schalter aus):

03 50 A0 E0 80 00

03: Das Frame-Info-Byte zeigt an, daß es sich um eine Standard-Frame-Nachricht mit 3 Datenbytes handelt.

50 A0: der Identifier enthält nur die festen SLIO-ID-Bits, die variablen Bits sind ,0', da alle DIP-Schalter aus sind. Die 11 Bits sind rechtsbündig in den beiden Bytes enthalten: 0101 0000 101 – 0 0000 0000.

C0: RSTD=1 -> Sign-On-Nachricht, EW ist in Sign-On immer 1, BM=10 -> Busmodus ist ,Sleep-Mode'. Die Registeradresse ist 0, es ist also das Data-Input-Register.

80 00: Inhalt des Data-Input-Registers. Bei offenen Pins ist der Inhalt eher zufällig.

Verstellen Sie den DIP-Schalter und Schalten Sie das SLIO-Board aus und wieder ein. Lassen Sie dann SLIO_FIND1 nochmal laufen.

Debuggen: wenn Sie SLIO-Beispielprogramme im Debugmodus per Einzelschritt untersuchen, dann geht wahrscheinlich der SLIO in den Sleep-Modus, da die Kalibrierungsnachrichten zu lange ausbleiben. Einzelschritte sollten am Besten gezielt nach einem Breakpoint durchgeführt werden. Danach muß ganz neu gestartet werden, damit der SLIO neu gefunden wird.

Device-Treiber

Programmbeispiel:

2

```
-----
' Name: SLIO_FIND1.TIG
' Findet SLIO-Baustein(e) oder meldet Fehler.
' Verbinde CAN-SLIO-Board(s) mit unterschiedlichen Adressen
-----
user var strict          ' unbedingte Var.deklaration
#include UFUNC3.INC      ' User Function Codes
#include DEFINE_A.INC    ' allg. Symbol-Definitionen
#include CAN.INC         ' CAN-Definitionen
#include CANSLIO.INC     ' Definitionen zum CAN-SLIO-Chip

#define NOT_READY 0      ' SLIO-Status 'nicht bereit'
#define READY 1        ' SLIO-Status 'bereit'

'
'           3 210          ID-Bitpositionen
#define ID_SLIO1 00000000000b ' ID-Bit-Einstellung dieses SLIO

BYTE slio_stat
WORD slio_id, calib_id
LONG ac_code, ac_mask
STRING id$(4), calib$(5)
STRING slio1_dout$(6), slio1_doe$(6) ' data out, data out enable

-----
TASK MAIN
  BYTE ever
  WORD fi
  STRING id1$, c$

  install_device #LCD, "LCD1.TDD"

' warte auf SLIO-Code, aber filtere 'keep-alive'-Codes aus
' diese Bits stehen linksbueendig in den je 32-Bit Code u. Mask
' 01010000101 <- feste SLIO-Bits
' 00100111000 <- Filter: 1:don't care, 0: Bit muss stimmen

  install_device #CAN, "CAN1_K1.TDD", &
    "50 A0 00 00 &          ' access code
    27 1F FF FF &          ' access mask
    10 45 &                ' Bitzeit 20usec (moegl.:8...50usec)
    08 1A"%                ' single filter mode, outctrl

  calib_id = 0aah shl 5     ' ID der Kalibrier-Botschaften
  calib$ = "<0><0><0><0aah><4>" ' spezieller String fuer Init-SLIO
  calib$ = ntos$ ( calib$, 1, -2, calib_id )' ID high-Byte zuerst

-----
' Sendet Kalibrier-Botschaften und wartet auf
' (gefilterte) SLIO-Botschaften als Antwort
'
print #LCD, "trying to find SLIO";
slio_stat = NOT_READY
while slio_stat = NOT_READY ' pruefe SLIO-Status
  put #CAN, calib$          ' Kalibrierbotschaft auf den Bus
  wait_duration 50
  get #CAN, #0, #UFICI_IBU_FILL, 2, fi
  if fi > 5 then            ' wenn min 1 Botschaft da ist
    print #LCD, "<1bh>A<0><1><0f0h>"; ' Cursor setzen
```

```

    get #CAN, 1, c$
    using "UH<2><2> 0 0 0 0 2" ' und HEX anzeigen
    print using #LCD, asc(c$);" ";
    get #CAN, #0, #UFCE_IBU_FILL, 2, fi ' mehr?
  endwhile
  print #LCD, "<1bh>A<0><2><0f0h>SLIO found";
  slio_stat = READY
endif
endwhile

for ever = 0 to 0 step 0      ' Endlosschleife
next
END

```

Für die später folgenden Beispiele ist ein etwas komplizierteres Unterprogramm ‚find_slios‘ geschrieben worden, welches weitere Gegebenheiten berücksichtigt:

Es können mehrer SLIOs am Bus sein.

Es könnten noch Busteilnehmer vorhanden sein, die keine SLIO sind.

Es könnte gar keine SLIO da sein, aber das Programm soll weiterlaufen.

Die SLIOs brauchen regelmäßig Kalibrier-Bitmuster auf dem CAN-Bus, sonst gehen sie wieder in den Sleep-Modus und die Ausgänge werden inaktiv.

Das Unterprogramm wartet eine Sekunde, um allen eventuell vorhandenen SLIOs die Möglichkeit zu geben, Ihre Sign-On-Nachricht zu schicken, und von der Anwendung auch berücksichtigt zu werden. Andererseits wird nicht länger gewartet, denn es könnte sein, daß sich keine SLIO meldet. Dieses Beispiel meldet diese Tatsache und beendet das Main-Programm.

Die Empfangenen Nachrichten werden unersucht, ob sie auch von einer SLIO stammen. Kein anderer Busteilnehmer darf eine der 16 möglichen SLIO-IDs haben. Andere als SLIO-Nachrichten werden in der Erkennungsphase nur aus dem Puffer gelesen und verworfen.

Die Task ‚keep_alive‘ sendet die Kalibrier-Nachrichten, um die SLIOs aufzuwecken und sie anschließend aktiv zu halten. Spätestens nach 8000 Bitzeiten soll eine Kalibrier-Nachricht auf dem Bus erscheinen. Kein Teilnehmer empfängt sie, nur ein für die Re-synchronisation der SLIOs notwendiges Bitmuster wird dadurch erzeugt. Während der Erkennungsphase wird die Kalibrier-Nachricht etwas öfter gesendet.

Die Task ‚keep_alive‘ sendet auch Nachrichten aus sofern dies nötig ist. In diesem Beispiel wird es nicht nötig, da nur die Erkennung demonstriert werden soll. Die anderen Beispiele benutzen jedoch die Ausgabe der Task ‚keep_alive‘ über die Variable ‚can_out\$‘.

Wenn Sie dieses Beispiel jetzt einzeln verstehen, können Sie in den folgenden Beispielen dieses Unterprogramm als ‚Black-Box‘ betrachten.

Device-Treiber

Debuggen: wenn Sie SLIO-Beispielprogramme im Debugmodus per Einzelschritt untersuchen, dann geht wahrscheinlich der SLIO in den Sleep-Modus, da die Kalibrierungsnachrichten zu lange ausbleiben. Einzelschritte sollten am Besten gezielt nach einem Breakpoint durchgeführt werden. Danach muß ganz neu gestartet werden, damit der SLIO neu gefunden wird.

2

Programmbeispiel:

```

'-----
' Name: SLIO_FIND2.TIG
' Findet SLIO-Baustein(e) oder meldet Fehler.
' Verbinde CAN-SLIO-Board(s) mit unterschiedlichen Adressen
'-----
user var strict          ' unbedingte Var.deklaration
#include UFUNC3.INC      ' User Function Codes
#include DEFINE_A.INC    ' allg. Symbol-Definitionen
#include CAN.INC         ' CAN-Definitionen
#include CANSLIO.INC     ' Definitionen zum CAN-SLIO-Chip

#define ALIVETIME 50     ' warte ca. 2200 * 22usec      wait
#define SLIOSLOT 10000   ' msec Zeit, sich 'bereit' zu melden

'           3 210        ID-Bitpositionen
#define ID_SLIO1 00000000000b ' ID-Bit-Einstellung dieses SLIO

' globale Variable
BYTE no_of_slios        ' Zaehler SLIOs im System
WORD slio1_id, calib_id ' ID der SLIO und der Kalib.-Botsch.
LONG ac_code, ac_mask   ' Access-Code und Access-Mask
LONG slio_equip         ' ein Bit fuer jede SLIO
LONG alive_wait         ' 1/2 Wartezeit in 'keep alive'
STRING calib$(5)        ' Kalibrierungsbotschaft
STRING can_out$(13)     ' Sendebotschaft

'-----
TASK MAIN
  BYTE ever              ' Endlosschleife
  WORD fi                ' Pufferfuellung
  STRING c$

  install_device #LCD, "LCD1.TDD" ' LCD-Treiber installieren

  ' warte auf SLIO-Code, aber filtere 'keep-alive'-Codes aus
  ' diese Bits stehen linksbuendig in den je 32-Bit Code u. Mask
  ' 01010000101 <- feste SLIO-Bits (50Ah)
  ' 00100111000 <- Filter: 1:don't care, 0: Bit muss stimmen (271h)

  install_device #CAN, "CAN1_K1.TDD", &
    "50 A0 00 00 &      ' access code
    27 1F FF FF &      ' access mask
    10 45 &             ' Bitzeit 20usec (moegl.:8...50usec)
    08 1A"%             ' single filter mode, outctrl

  call find_slios        ' stellt Liste aller im System
                        ' befindlichen SLIOs zusammen
  if slio_equip = 0 then ' wenn keine SLIO gefunden
    goto no_slio_found  ' dann Programm abbrechen
  else
    print #LCD, "<1Bh>A<0><2><0F0h>";no_of_slios;" SLIOs found";
  endif

  for ever = 0 to 0 step 0 ' Endlosschleife
    print #LCD, "<1bh>A<0><3><0f0h>task main running"
  next

no_slio_found:

```

```

print #LCD, "<1>no SLIO found"
print #LCD, "program terminated"
END

-----
' Sendet Kalibrier-Botschaften und wartet auf
' (gefilterte) SLIO-Botschaften als Antwort
' Innerhalb einer gesetzten Zeit muessen alle SLIOs sich gemeldet
' haben. Die Filterung muss im Hauptprogramm gesetzt sein.
' Nicht-SLIO-Botschaften und extended Frames werden verworfen,
' d.h. lediglich aus dem Puffer geräumt.
-----

SUB find_sl ios ( )
    BYTE ever, i                ' Schleifenvariable
    WORD ibu_fill              ' Eingangspufferfuellung
    BYTE frameformat, msg_len  ' Frame-Format, Botschaftslaenge
    LONG r_id                  ' Empfangs-IDreceive ID
    LONG t
    STRING msg$(13), data$(8)  ' Botschaft und Daten

    sl io equip = 0            ' fange an mit 'keine SLIO'
    no_of_sl ios = 0
    cal ib id = 0aah shl 5     ' ID der Kalibrier-Botschaften
    cal ib $ = "<0><0><0><0aah><4>" ' spezieller String fuer Init-SLIO
    cal ib $ = ntos$ ( cal ib $, 1, -2, cal ib id ) ' ID high-Byte zuerst
    can_out $ = ""            ' muss initialisiert sein

    alive_wait = ALIVETIME/2  ' in der Initphase kuerzer
    run_task keep_alive       ' sendet Kalibrier-Botschaften
                                ' und haelt SLIOs synchronisiert

    print #LCD, "trying to find SLIOs";
    t = ticks()                ' Anfang des Zeitfensters
    while diff_ticks ( t ) < SLIOSLOT ' innerhalb des Zeitfensters
rx_cont:
    wait_duration 50
    get #CAN, #0, #UFCI_IBU_FILL, 0, ibu_fill
    if ibu_fill > 2 then        ' wenn mindestens eine Message
        get #CAN, #0, 1, frameformat ' hole Frame-Info-Byte
        msg_len = frameformat bitand 1111b ' Laenge
        if frameformat bitand 80h = 0 then ' wenn Standard-Frame
            get #CAN, #0, CAN_ID11_LEN, r_id ' hole ID-Bytes
            r_id = byte_mirr ( r_id, 2 )
            r_id = r_id shr 5

            ' wenn keine SLIO-Botschaft
        if r_id bitand SLIO_ID_IMASK <> SLIO_FIX_ID then ' aus Puffer
            if msg_len > 0 then ' wenn Daten: wegwerfen
                get #CAN, #0, msg_len, data$ ' hole sie aus dem Puffer
            endif
            goto rx_cont ' warte auf naechste Botschaft
        endif
    else ' sonst ist es extended frame
        get #CAN, #0, CAN_ID29_LEN, r_id ' und damit nicht von SLIO
        if msg_len > 0 then ' wenn Daten
            get #CAN, #0, msg_len, data$ ' hole sie aus dem Puffer
        endif
        goto rx_cont ' warte auf naechste Botschaft
    endif

-----
' Hier: es war SLIO-Botschaft. Finde Adresse heraus und

```

```

r_id = r_id bitand SLIO_ID_MASK      ' setzbare SLIO Adressbits
if bit ( r_id, 8 ) = 1 then          ' wenn P8 gesetzt, dann
  r_id = ( r_id bitand 111000b ) + 40h
endif
r_id = r_id shr 3                   ' schiebe untere 3 ID-Bits weg
set_bit slio Equip, r_id            ' setze Bitnummer in Liste
print #LCD, "<1Bh>A<0><1><0F0h>SLIO ";r_id;" found"
no_of_slis = no_of_slis + 1         ' zaehle die gefundene SLIO
                                     ' ignoriere hier Info in Daten
if msg_len > 0 then                ' wenn Daten
  get #CAN, #0, msg_len, data$      ' hole sie aus dem Puffer
  using "UH<2><2> 0.0.0.0.2"         ' Format HEX-Ausgabe
  for i = 0 to len ( data$ ) - 1    ' alle Bytes des Strings
    print_using #LCD, asc ( mid$ ( data$, i, 1 ) );' als HEX
  next
endif
endif
endwhile
alive_wait = ALIVETIME              ' normale Wartezeit 'keep_alive'
END

'-----
' Sendet Kalibrier-Botschaften, um die SLIO synchronisiert zu halten.
' Spaetestens alle 8000 Bitzeiten muss eine Kalibrier-Botschaften
' gesendet werden.
' Wenn 'can_out$' nicht leer ist, wird es gesendet und anschliessend
' wieder auf leer gesetzt.
'-----
TASK keep_alive
  BYTE ever

  for ever = 0 to 0 step 0           ' Endlosschleife
    put #CAN, calib$                 ' sende Kalibrier-Botschaft
    wait_duration alive_wait         ' warte max 4000 Bitzeiten
    if can_out$ <> "" then
      put #CAN, can_out$             ' Sendebotschaft
      can_out$ = ""                 ' leer = gesendet
    endif
    wait_duration alive_wait         ' warte max 4000 Bitzeiten
  next
END

```

Device-Treiber

Einige Besonderheiten für Interessierte

Um die nicht ganz unkomplizierte Thematik in der Einführung nicht unnötig zu erschweren, wurden einige Besonderheiten nicht an Ort und Stelle erwähnt.

2

Remote Frames

Remote-Frames können eingesetzt werden, um entfernt über den CAN-Bus Nachrichten hervorzurufen, sprich Werte abzufragen und Informationen zu erhalten. Obwohl die SLIO auch ohne gesetztes RTR-Bit antwortet, darf das Bit natürlich gesetzt sein. Empfangene Remote Frames müssen den DLC (data length code) jedoch auf 3 gesetzt haben, sonst werden sie ignoriert. Die Antwort enthält immer die Bytes des Daten-Input-Registers.

Bit-Timing

Die nominale Bitzeit des CAN-SLIO-Chips ist unterteilt in 10 Bitsegmente. Das Synchronisations-Segment und das Propagation-Time-Segment leiten das Bit ein. Es folgt mit 4 Segmenten Phase 1, an deren Ende das Bit abgetastet wird. Phase 2 ist ebenfalls 4 Segment lang. Diese Timingstruktur ist nicht veränderbar. Der Quarz-kontrollierte Host sollte auch so eingestellt werden, daß die Bitzeit in 10 Segmente zerlegt wird.

1 Bitzeit									
BT1	BT2	BT3	BT4	BT5	BT6	BT7	BT8	BT9	BT10
Sync	Prop	phase_1				phase_2			

Oszillator und Kalibrierung

Der SLIO-Chip hat einen internen R-C-Oszillator, der automatisch durch die CAN-Nachrichten auf dem Bus kalibriert wird. Während des Starts wird jede Nachricht verwendet, um die Bitzeit einzustellen. Zur genauen Kalibrierung ist ein bestimmtes Bitmuster auf dem Bus notwendig. Beispiel einer geeigneten Kalibrier-Nachricht (| = Stuffbit, die wichtigen Bits sind unterstrichen):

SOF	arbitration	control field	data byte 1	data byte 2	CRC field
0	000101010100	000 010	<u>1</u> 0101010	0000 0100	000 01011100000 0

Identifier ist AAh, zwei Datenbytes mit den Werten AAh, 4 sind in der Nachricht enthalten.

Initialisierung

In der Resetphase (RST=high) sind alle Ausgänge des CAN_SLIO hochohmig. An den Pins P0...P3 werden die 4 ID-Bits eingelesen. Die anderen ID-Bits sind unveränderbar festgelegt. Gemäß der CAN-Definition dürfen nicht zwei Knoten den gleichen Identifier verwenden. Ein CAN-SLIO hat eine der 16 möglichen Bitkombinationen.

Reset-Zustand:

Status-Bits	Identifier-Bits
RSTD = 1	ID.3 gleich P0
EW = 1	ID.4 gleich P1
BM1 = 0	ID.5 gleich P2
BM0 = 0	ID.8 gleich P3

Der SLIO-Chip muss mindestens 3 Nachrichten auf dem Bus empfangen haben, bevor der Bus-Mode gewechselt wird. Die erste Botschaft wird benutzt, um die Bitzeit zu messen, deswegen sollte die Nachricht eine Sequenz ‚010101‘ enthalten. Die 2. und die 3. Nachricht wird trotz korrektem Empfang nicht mit einem ACK bestätigt. Nach 3 korrekt empfangenen Nachrichten sendet der SLIO-Chip seine ‚sign-on‘-Nachricht.

Der CAN-SLIO-Chip wertet auch Nachrichten als korrekt empfangen, wenn diesen ein Error-Passive-Frame auf Grund des fehlenden ACK folgt. Diese Situation ist gegeben, wenn ein Host mit einem oder mehreren CAN-SLIOS zusammenarbeitet und die SLIOs noch nicht kalibriert sind.

Sign-On-Message

Diese spezielle Botschaft wird einmal von jedem CAN-SLIO versandt, nachdem der Chip kalibriert wurde. Damit wird die Betriebsbereitschaft des Knotens angezeigt.

Die Sign-On-Nachricht meldet den Inhalt des data-Input-Registers und kann von anderen Nachrichten durch das gesetzte Bit RSTD unterschieden werden:

RSTD = 1: Sign-On-Nachricht
RSTD = 0: andere Nachrichten

Anmerkung: In der Sign-On-Nachricht ist das EW-Bit gesetzt. Trotzdem wird der Status und die Error-Zähler zurückgesetzt auf 0.

Registerübersicht

Register Adresse	Registername	Funktion
0	Data Input	Enthält die Input-Pegel der Pins P0 bis P15
1	Event Positive Edge	Aktiviert bei gesetztem Bit das Absetzen einer Nachricht bei positiver Flanke an den Pins P0...P15
2	Event Negative Edge	Aktiviert bei gesetztem Bit das Absetzen einer Nachricht bei negativer Flanke an den Pins P0...P15
3	Data Output	Enthält die Output-Pegel der Pins P0 bis P15
4	Output Enable	Aktiviert bei gesetztem Bit die Ausgangstreiber an den Pins P0...P15
5	Analog Configuration	Bits 0...4: ohne Funktion Bits 5,6,7: Position des Anlogschalters Bits 8,9,10: Position der Monitoring-Schalter Bit 11: ohne Funktion Bits 12,13,14: Ergebnis der Analog-Komparatoren Bits 15: startet A/D-Wandlung, wenn gesetzt
6	DPM1	Distributed Pulse Modulation. Enthält den 10-Bit-Analogwert in den Bits 6...15. Quasi-Analog-Ausgang 1 am Pin P10
7	DPM2	Distributed Pulse Modulation. Enthält den 10-Bit-Analogwert in den Bits 6...15. Quasi-Analog-Ausgang 2 am Pin P4
8	A/D-Conversion	Enthält das 10-Bit-Ergebnis der A/D-Wandlung in den Bits 6...15

SLIO digitale Ein- und Ausgänge

Der SLIO-Chip stellt 16 Pins zur Verfügung, die als digitale Eingänge oder digitale Ausgänge arbeiten können. Einige Pins haben jedoch die Möglichkeit, andere Funktionen zu übernehmen, z.B. analoge Eingänge, analoge Ausgänge sowie das Einlesen der ID-Bits nach dem Reset. Einige Pins sind daher auf dem CAN-SLIO-Board besonders beschaltet und nicht einfach auf die Stiftleiste geführt:

- P0...P3** besitzen Pull-up-Widerstände von 47k und Pull-down-Widerstände von 4k7, um mit dem DIP-Schalter die ID-Bits einzustellen.
- P4, P10** sind mögliche quasi-analoge Ausgänge und über einfache R-C-Filter geführt.
- P15, P16** sind über einen 100k-Widerstand verbunden und an P15 befindet sich ein 3,3nF-Kondensator gegen Masse. Diese Beschaltung ist notwendig, um P15 als Analog-In zu verwenden.

Digitale Eingänge: Nach dem Reset oder Power-On sind alle Pins Eingänge mit ca. 500k Eingangswiderstand. Die Pegelzustände spiegeln sich im ‚Data-In‘-Register (Adresse 0) wieder. Das Register ‚Output-Enable‘ (Adresse 4) hat für jeden Portpin ein Bit, welches auf ‚0‘ steht und damit den Output-Treiber de-aktiviert.

Digitale Ausgänge: Das Register ‚Output-Enable‘ (Adresse 4) hat für jeden Portpin ein Bit, welches auf ‚1‘ den Output-Treiber aktiviert. Das Register muß explizit beschrieben werden, um Ausgangspins zu aktivieren. Das Bitmuster für alle aktivierten Ausgänge wird in das Register ‚Data-Output‘ (Adresse 3) geschrieben.

Events: CAN-Nachrichten werden automatisch versandt, wenn in den Registern ‚Event Positive Edge‘ und/oder ‚Event Negative Edge‘ das Bit für den gewünschten Pin gesetzt ist und die dazu passende Flanke auftritt.

Das folgende Beispielprogramm setzt alle Portpins zu Ausgängen und schaltet sie abwechselnd auf high und low. Um die SLIO(s) zu finden, wird das Unterprogramm ‚find_slios‘ zusammen mit der Task ‚keep_alive‘ verwendet, so wie unter ‚SLIOs auf dem Bus finden‘ beschrieben. Der Identifier der zuletzt gefundenen SLIO wird verwendet. Die Ports werden zu Ausgängen, indem in dem ‚Output-Enable‘-Register alle Bits auf ‚1‘ gesetzt werden. Dann wird in das ‚Data-Output‘-Register abwechselnd 0000 und FFFFh geschrieben. Da der SLIO mit einer Bestätigungsnachricht antwortet, wenn er eine Nachricht empfangen hat, gibt die Task ‚show_slio‘ diese SLIO-Nachrichten und den Identifier auf dem LCD in HEX aus. Fremde Nachrichten, dazu zählen automatisch extended Frames, werden von ‚show_slio‘ verworfen, es wird nur der Empfangspuffer bereinigt.

Device-Treiber

Debuggen: wenn Sie SLIO-Beispielprogramme im Debugmodus per Einzelschritt untersuchen, dann geht wahrscheinlich der SLIO in den Sleep-Modus, da die Kalibrierungsnachrichten zu lange ausbleiben. Einzelschritte sollten am Besten gezielt nach einem Breakpoint durchgeführt werden. Danach muß ganz neu gestartet werden, damit der SLIO neu gefunden wird.

2

Programmbeispiel:

```
-----
' Name:  SLIO_HIGH_LOW.TIG
' Blinkt alle Outputs des SLIO 'high-low'
' Verbinde CAN-SLIO-Board, alle DIP-Schalter aus
-----
user var strict          ' unbedingte Var.deklaration
#include UFUNC3.INC      ' User Function Codes
#include DEFINE_A.INC    ' allg. Symbol-Definitionen
#include CAN.INC         ' CAN-Definitionen
#include CANSLIO.INC     ' Definitionen zum CAN-SLIO-Chip

#define ALIVETIME 50     ' warte ca. 2500 * 20usec
#define SLIOSLOT 1000    ' msec Zeit, sich 'bereit' zu melden

#define NOT_READY 0     ' SLIO-Status 'nicht bereit'
#define READY 1        ' SLIO-Status 'bereit'

'           3 210          ID-Bitpositionen
#define ID_SLIO1 00000000000b ' ID-Bit-Einstellung dieses SLIO

' globale Variable
BYTE no_of_slis         ' Zaehler SLIOs im System
BYTE slio_stat         ' SLIO status
WORD slio1_id, calib_id ' ID der SLIO und der Kalib.-Botsch.
LONG ac_code, ac_mask  ' Access-Code und Access-Mask
LONG slio equip        ' ein Bit fuer jede SLIO
LONG alive_wait        ' 1/2 Wartezeit in 'keep_alive'
STRING calib$(5)       ' Kalibrierungsbotschaft
STRING slio1_dout$(6), slio1_doe$(6) ' data out, data out enable
STRING can_out$(13)    ' Sendebotschaft

-----
TASK MAIN
  BYTE ever            ' Endlosschleife
  WORD fi             ' Pufferfuellung

  install_device #LCD, "LCD1.TDD"

' warte auf SLIO-Code, aber filtere 'keep-alive'-Codes aus
' diese Bits stehen linksbuedig in den je 32-Bit Code u. Mask
' 01010000101 <- feste SLIO-Bits (50Ah)
' 00100111000 <- Filter: 1:don't care, 0: Bit muss stimmen (271h)

  install_device #CAN, "CAN1_K1.TDD", &
    "50 A0 00 00 &      ' access code
    27 1F FF FF &      ' access mask
    10 45 &            ' Bitzeit 20usec (moegl.:8...50usec)
    08 1A"%           ' single filter mode, outctrl

' ID dieses SLIO: 0=standard frame
```

```

' sliol_id = (SLIO_FIX_ID bitor ID_SLIO1) shl 5 ' SLIO-Identifizier
' sliol_id wird letzte gefundene SLIO
calib_id = 0aah shl 5 ' ID der Kalibrier-Botschaften
can_out$ = "" ' muss initialisiert sein

call find_sl ios ' stellt Liste aller im System
' befindlichen SLIOs zusammen
if slio_equip = 0 then ' wenn keine SLIO gefunden
goto no_slio_found ' dann Programm abbrechen
else
print #LCD, "<1Bh>A<0><2><0F0h>";no_of_sl ios;" SLIOs found";
endif
wait_duration 1500
print #LCD, "<1>";

run_task show_slio ' zeigt an, was die SLIO(s) senden

sliol_doe$ = "<0><0><0><4><0FFh><0FFh>" ' SLIO_OUT_ENABLE
sliol_doe$ = ntos$ ( sliol_doe$, 1, -2, sliol_id)'ID high-Byte zuerst
can_out$ = sliol_doe$
wait_duration 100
for ever = 0 to 0 step 0 ' Endlosschleife -----
sliol_dout$ = "<0><0><0><3><0FFh><0FFh>" ' SLIO_DATA_OUT
sliol_dout$ = ntos$ ( sliol_dout$, 1, -2, sliol_id )' ID high-Byte
can_out$ = sliol_dout$
wait_duration 100
sliol_dout$ = "<0><0><0><3><0><0>" ' SLIO_DATA_OUT SLIO_DATA_OUT
sliol_dout$ = ntos$ ( sliol_dout$, 1, -2, sliol_id )' ID einbauen
can_out$ = sliol_dout$
wait_duration 100
next ' Endlosschleife -----

no_slio_found: ' keine SLIO gefunden
stop_task keep_alive
print #LCD, "<1>no SLIO found"
print #LCD, "program terminated"
END

'-----
' Zeigt Inhalte der SLIO-Botschaften an
' Zeile 1 auf LCD: ID
' Zeile 2 auf LCD: Datenbytes in HEX
'-----

TASK show_slio
BYTE ever, i ' Schleifenvariable
WORD ibu_fill ' Eingangspufferfuellung
BYTE frameformat, msg_len ' Frame-Format, Botschaftslaenge
LONG r_id ' Empfangs-ID
STRING msg$(13), data$(8) ' Botschaft und Daten

for ever = 0 to 0 step 0 ' Endlosschleife
rx_cont:
get #CAN, #0, #UFICI_IBU_FILL, 0, ibu_fill
if ibu_fill > 2 then ' wenn mindestens eine Message
get #CAN, #0, 1, frameformat ' hole Frame-Info-Byte
msg_len = frameformat bitand 1111b ' Laenge
if frameformat bitand 80h = 0 then ' wenn Standard-Frame
get #CAN, #0, CAN_ID11_LEN, r_id ' hole ID-Bytes
r_id = byte_mirr ( r_id, 2 )

```

```

                                ' wenn keine SLIO-Botschaft
if r_id bitand SLIO_ID_IMASK <> SLIO_FIX_ID then ' aus Puffer
  if msg_len > 0 then           ' wenn Daten: wegwerfen
    get #CAN, #0, msg_len, data$ ' hole sie aus dem Puffer
  endif
  goto rx_cont                  ' warte auf naechste Botschaft
endif
else                             ' sonst ist es extended frame
  get #CAN, #0, CAN_ID29_LEN, r_id ' und damit nicht von SLIO
  if msg_len > 0 then           ' wenn Daten
    get #CAN, #0, msg_len, data$ ' hole sie aus dem Puffer
  endif
  goto rx_cont                  ' warte auf naechste Botschaft
endif
                                ' -----
                                ' Hier: es war SLIO-Botschaft
using "UH<3><3>  0 0 0 0 3"      ' fuer ID Anzeige
print_using #LCD, "<1Bh>A<0><1><0F0h> ID:";r_id ' ID des SLIO
print #LCD, "                    <13>DATA:"; ' loesche Datenanz.
if msg_len > 0 then           ' wenn Daten
  get #CAN, #0, msg_len, data$ ' hole sie und zeige an
  using "UH<2><2>  0.0.0.0.2"   ' Format HEX-Ausgabe
  for i = 0 to len ( data$ ) - 1 ' alle Bytes des Strings
    print_using #LCD, asc ( mid$ ( data$, i, 1 ) ); ' als HEX
  next
endif
endif
next
END

' -----
' Sendet Kalibrier-Botschaften und wartet auf
' (gefilterte) SLIO-Botschaften als Antwort
' Innerhalb einer gesetzten Zeit muessen alle SLIOs sich gemeldet
' haben. Die Filterung muss im Hauptprogramm gesetzt sein.
' Nicht-SLIO-Botschaften und extended Frames werden verworfen,
' d.h. lediglich aus dem Puffer geräumt.
' -----

SUB find_sl ios ( )
  BYTE ever, i                  ' Schleifenvariable
  WORD  ibu_fill                ' Eingangspufferfuellung
  BYTE frameformat, msg_len    ' Frame-Format, Botschaftslaenge
  LONG  r_id                    ' Empfangs-IDreceive ID
  LONG  t
  STRING msg$(13), data$(8)    ' Botschaft und Daten

  sl io_equip = 0               ' fange an mit 'keine SLIO'
  no_of_sl ios = 0
  calib_id = 0aah shl 5         ' ID der Kalibrier-Botschaften
  calib$ = "<0><0><0><0aah><4>"   ' spezieller String fuer Init-SLIO
  calib$ = ntos$ ( calib$, 1, -2, calib_id ) ' ID high-Byte zuerst

  alive_wait = ALIVETIME/2     ' in der Initphase kuerzer
  run_task keep_alive          ' sendet Kalibrier-Botschaften
                                ' und haelt SLIOs synchronisiert

  print #LCD, "trying to find SLIOs";
  t = ticks()                  ' Anfang des Zeitfensters
  while diff_ticks ( t ) < SLIOSLOT ' innerhalb des Zeitfensters
find_cont:
  wait_duration 50

```

```

if ibu_fill > 2 then                                ' wenn mindestens eine Message
get #CAN, #0, 1, frameformat                        ' hole Frame-Info-Byte
msg_len = frameformat bitand 1111b                 ' Laenge
if frameformat bitand 80h = 0 then                  ' wenn Standard-Frame
get #CAN, #0, CAN_ID11_LEN, r_id                   ' hole ID-Bytes
r_id = byte_mirr ( r_id, 2 )
r_id = r_id shr 5

if r_id bitand SLIO_ID_IMASK <> SLIO_FIX_ID then    ' wenn keine SLIO-Botschaft
if msg_len > 0 then                                ' wenn Daten: wegwerfen
get #CAN, #0, msg_len, data$                       ' hole sie aus dem Puffer
endif
goto find_cont                                     ' warte auf naechste Botschaft
endif
else
get #CAN, #0, CAN_ID29_LEN, r_id                   ' sonst ist es extended frame
if msg_len > 0 then                                ' und damit nicht von SLIO
get #CAN, #0, msg_len, data$                       ' wenn Daten
endif
goto find_cont                                     ' hole sie aus dem Puffer
endif
goto find_cont                                     ' warte auf naechste Botschaft
endif

'-----
' Hier: es war SLIO-Botschaft. Finde Adresse heraus und
' trage in Bit-Liste ein, verwende die Adresse im Testprogramm

slio1_id = ( r_id bitand 7FEh) shl 5                ' nehme diese Adress f. Test
r_id = r_id bitand SLIO_ID_MASK                    ' maskiere DIR-Bit weg
if bit ( r_id, 8 ) = 1 then                         ' setzbare SLIO Adressbits
r_id = ( r_id bitand 111000b ) + 40h               ' wenn P8 gesetzt, dann
endif
r_id = r_id shr 3                                  ' schiebe untere 3 ID-Bist weg
set_bit slio equip, r_id                           ' setze Bitnummer in Liste
print #LCD, "<lBh>A<0><l><0F0h>SLIO ";r_id;" found"
no_of_slios = no_of_slios + 1                      ' zaehle die gefundene SLIO
                                                    ' ignoriere hier Info in Daten
if msg_len > 0 then                                ' wenn Daten
get #CAN, #0, msg_len, data$                       ' hole sie aus dem Puffer
using "UH<2><2> 0.0.0.0.2"                          ' Format HEX-Ausgabe
for i = 0 to len ( data$ ) - 1                     ' alle Bytes des Strings
print_using #LCD, asc ( mid$ ( data$, i, 1 ) );    ' als HEX
next
endif
endif
endwhile
alive_wait = ALIVETIME                             ' normale Wartezeit 'keep_alive'
END

'-----
' Sendet Kalibrier-Botschaften, um die SLIO synchronisiert zu halten.
' Spaetestens alle 8000 Bitzeiten muss eine Kalibrier-Botschaften
' gesendet werden.
'-----

TASK keep_alive
BYTE ever

for ever = 0 to 0 step 0                            ' Endlosschleife
put #CAN, calib$                                    ' sende Kalibrier-Botschaft
wait_duration alive_wait                            ' warte max 4000 Bitzeiten

```

Device-Treiber

```
    put #CAN, can_out$           ' Sendebotschaft
    can_out$ = ""              ' leer = gesendet
endif
    wait_duration alive_wait    ' warte max 4000 Bitzeiten
next
END
```

2

SLIO-Analog-Ausgänge

Der SLIO-Chip stellt zwei quasi-analoge Ausgänge zur Verfügung. Auf dem CAN-SLIO-Board sind diese Ausgänge über einfache R-C-Filter geführt.

Der analoge Wert wird erzeugt, indem über einen festen Zeitraum mehr oder weniger ,1'-Bits generiert werden (DMP=Distributed Pulse Modulation). Integriert man die ,1'-Bits erhält man für wenige ,1'-Bits pro Zeiteinheit eine niedrige Spannung, für viele ,1'-Bits pro Zeiteinheit eine hohe Spannung (0...5V). Bei der Integration muß ein Kompromiß geschlossen werden zwischen Reaktionsgeschwindigkeit der analogen Spannung und Welligkeit. Die Erzeugung eines Analogwertes ist nach 1024 Bitzeiten abgeschlossen und wird anschließend wiederholt. Grundlage für die Anzahl der ,1'-Bits ist der Inhalt des Registers ,DPM1' für Pin P10 oder ,DMP2' für Pin P4. Im Register befindet sich in einem WORD linksbündig der 10-Bit-Wert für die DPM-Ausgabe.

Die Ausgangstreiber der Pins, die zur Analogausgabe verwendet werden, müssen im Register ,Output-Enable' aktiviert werden.

Das Beispielprogramm erzeugt auf dem Pin P10 eine Sägezahnkurve, indem aufsteigende Analogwerte ausgegeben werden. Die SLIO(s) werden mit Hilfe des Unterprogrammes ,**find_slios**' zusammen mit der Task ,**keep_alive**' gefunden, so wie unter ,SLIOs auf dem Bus finden' beschrieben. Danach übernimmt die Hauptschleife des Programms die Aufgabe, die SLIOs zu re-kalibrieren, da hier sowieso eine schnelle und konstant sich wiederholende Ausgabe stattfindet. Die Task ,**show_slio**' zeigt die ankommenden SLIO-Nachrichten und den Identifier auf dem LCD in HEX zur Kontrolle an. Fremde Nachrichten, dazu zählen automatisch extended Frames, werden von ,show_slio' verworfen, es wird nur der Empfangspuffer bereinigt.

Debuggen: wenn Sie SLIO-Beispielprogramme im Debugmodus per Einzelschritt untersuchen, dann geht wahrscheinlich der SLIO in den Sleep-Modus, da die Kalibrierungsnachrichten zu lange ausbleiben. Einzelschritte sollten am Besten gezielt nach einem Breakpoint durchgeführt werden. Danach muß ganz neu gestartet werden, damit der SLIO neu gefunden wird.

Device-Treiber

Programmbeispiel:

2

```
'-----
' Name: SLIO DPM1.TIG
' erzeugt DPM-Outputs des SLIO (digital->analog)
' Verbinde CAN-SLIO-Board, alle DIP-Schalter aus
' messe am Pin P10
'-----
user var strict          ' unbedingte Var.deklaration
#include UFUNC3.INC      ' User Function Codes
#include DEFINE_A.INC    ' allg. Symbol-Definitionen
#include CAN.INC         ' CAN-Definitionen
#include CANSLIO.INC     ' Definitionen zum CAN-SLIO-Chip

#define ALIVETIME 50     ' warte ca. 2500 * 20usec
#define SLIOSLOT 1000    ' msec Zeit, sich 'bereit' zu melden

#define NOT_READY 0     ' SLIO-Status 'nicht bereit'
#define READY 1        ' SLIO-Status 'bereit'

'           3 210      ID-Bitpositionen
#define ID_SLIO1 00000000000b ' ID-Bit-Einstellung dieses SLIO

BYTE no_of_sl ios      ' Zaehler SLIOs im System
BYTE sl io_stat        ' SLIO status
WORD sl io1_id, cal ib_id ' ID der SLIO und der Kalib.-Botsch.
LONG ac_code, ac_mask  ' Access-Code und Access-Mask
LONG sl io_equip       ' ein Bit fuer jede SLIO
LONG alive_wait        ' 1/2 Wartezeit in 'keep_alive'
STRING calib$(5)       ' Kalibrierungsbotschaft
STRING sl io1_dpml$(6), sl io1_doe$(6) ' DPML out, data out enable
STRING sl io1_dout$
STRING can_out$(13)    ' Sendebotschaft

'-----
TASK MAIN
  BYTE ever            ' Endlosschleife
  WORD fi              ' Pufferfuellung
  WORD value, tmp      ' Analogwert
  LONG t               ' 'keep_alive' Zeit
  STRING tmp$(6)

  install_device #LCD, "LCD1.TDD" ' LCD-Treiber installieren

' warte auf SLIO-Code, aber filtere 'keep-alive'-Codes aus
' diese Bits stehen linksbuendig in den je 32-Bit Code u. Mask
' 01010000101 <- feste SLIO-Bits (50Ah)
' 00100111000 <- Filter: 1:don't care, 0: Bit muss stimmen (271h)

  install_device #CAN, "CAN1_K1.TDD", &
    "50 A0 00 00 &      ' access code
    27 1F FF FF &      ' access mask
    10 45 &             ' Bitzeit 20usec (moegl.:8...50usec)
    08 1A"%             ' single filter mode, outctrl

' ID dieses SLIO: 0=standard frame
' + 2 ID-Bytes um 5 geschiftet
sl io1_id = (SLIO_FIX_ID bitor ID_SLIO1) shl 5 ' SLIO-Identifizier
calib_id = 0aah shl 5 ' ID der Kalibrier-Botschaften
can_out$ = ""         ' muss initialisiert sein
```

```

call find_sl ios          ' stellt Liste aller im System
                          ' befindlichen SLIOs zusammen
if sl io_ equip = 0 then  ' wenn keine SLIO gefunden
  goto no_sl io_ found    ' dann Programm abbrechen
else
  print #LCD, "<1Bh>A<0><2><0F0h>";no_of_sl ios;" SLIOs found";
endif
wait_ duration 1500
print #LCD, "<1>";

run_ task show_sl io      ' zeigt an, was die SLIO(s) senden

sl io_ do_ e$ = "<0><0><0><4><0FFh><0FFh>" ' SLIO_OUT_ENABLE
sl io_ do_ e$ = ntos$ ( sl io_ do_ e$, 1, -2, sl io_ id)' ID high-Byte zuerst
can_ out$ = sl io_ do_ e$
wait_ duration 1000

sl io_ do_ ut$ = "<0><0><0><3><0FFh><0FFh>" ' SLIO_DATA_OUT
sl io_ do_ ut$ = ntos$ ( sl io_ do_ ut$, 1, -2, sl io_ id)' ID high-Byte
can_ out$ = sl io_ do_ ut$
wait_ duration 1000

sl io_ do_ ut$ = "<0><0><0><3><0><0>" ' SLIO_DATA_OUT  SLIO_DATA_OUT
sl io_ do_ ut$ = ntos$ ( sl io_ do_ ut$, 1, -2, sl io_ id)' ID einbauen
can_ out$ = sl io_ do_ ut$
wait_ duration 1000

sl io_ dp_ m1$ = "<0><0><0><6><0><0>" ' SLIO_DPM1
sl io_ dp_ m1$ = ntos$ ( sl io_ dp_ m1$, 1, -2, sl io_ id)' ID high-Byte
value = 0
t = ticks()
stop_ task keep_ alive    ' Hauptschleife uebernimmt das jetzt
for ever = 0 to 0 step 0  ' Endlosschleife -----
  tmp = value shl 5       ' 10-Bit rechtbuendig in WORD
  tmp$ = ntos$ ( sl io_ dp_ m1$, 4, -2, tmp)' value high-Byte zuerst
  can_ out$ = tmp$
  put #CAN, tmp$
  value = modulo_ inc ( value, 0, 7FFh, 7Fh ) ' inc step 7Fh
  if diff_ ticks ( t ) > 50 then           ' Wert haengt von Bitzeit ab
    put #CAN, calib$                       ' sende Kalibrier-Botschaft
    t = ticks()
  endif
next                                       ' Endlosschleife -----

no_sl io_ found:                          ' keine SLIO gefunden
  stop_ task keep_ alive
  print #LCD, "<1>no SLIO found"
  print #LCD, "program terminated"
END

'-----
' Zeigt Inhalte der SLIO-Botschaften an
' Zeile 1 auf LCD: ID
' Zeile 2 auf LCD: Datenbytes in HEX
'-----

TASK show_sl io
  BYTE ever, i          ' Schleifenvariable
  WORD ibu_ fill        ' Eingangspufferfuellung
  BYTE frameformat, msg_ len  ' Frame-Format, Botschaftslaenge
  LONG r_ id            ' Empfangs-IDreceive ID

```

```

for ever = 0 to 0 step 0          ' Endlosschleife
rx_cont:
  get #CAN, #0, #UFCI_IBU_FILL, 0, ibu_fill
  if ibu_fill > 2 then           ' wenn mindestens eine Message
    get #CAN, #0, 1, frameformat ' hole Frame-Info-Byte
    msg_len = frameformat bitand 1111b ' Laenge
    if frameformat bitand 80h = 0 then ' wenn Standard-Frame
      get #CAN, #0, CAN_ID11_LEN, r_id ' hole ID-Bytes
      r_id = byte_mirr ( r_id, 2 )
      r_id = r_id shr 5
      ' wenn keine SLIO-Botschaft
    if r_id bitand SLIO_ID_IMASK <> SLIO_FIX_ID then ' aus Puffer
      if msg_len > 0 then ' wenn Daten: wegwerfen
        get #CAN, #0, msg_len, data$ ' hole sie aus dem Puffer
        endif
        goto rx_cont ' warte auf naechste Botschaft
      endif
    else ' sonst ist es extended frame
      get #CAN, #0, CAN_ID29_LEN, r_id ' und damit nicht von SLIO
      if msg_len > 0 then ' wenn Daten
        get #CAN, #0, msg_len, data$ ' hole sie aus dem Puffer
        endif
        goto rx_cont ' warte auf naechste Botschaft
      endif
      ' -----
      ' Hier: es war SLIO-Botschaft
      fuer ID Anzeige
      print using #LCD, "<1Bh>A<0><1><0F0h> ID:";r_id ' ID des SLIO
      print #LCD, " <13>DATA:"; ' loesche Datenanz.
      if msg_len > 0 then ' wenn Daten
        get #CAN, #0, msg_len, data$ ' hole sie und zeige an
        using "UH<2><2> 0.0.0.0.2" ' Format HEX-Ausgabe
        for i = 0 to len ( data$ ) - 1 ' alle Bytes des Strings
          print_using #LCD, asc ( mid$ ( data$, i, 1 ) ); ' als HEX
        next
      endif
    endif
  next
END

' -----
' Sendet Kalibrier-Botschaften und wartet auf
' (gefilterte) SLIO-Botschaften als Antwort
' Innerhalb einer gesetzten Zeit muessen alle SLIOs sich gemeldet
' haben. Die Filterung muss im Hauptprogramm gesetzt sein.
' Nicht-SLIO-Botschaften und extended Frames werden verworfen,
' d.h. lediglich aus dem Puffer geräumt.
' -----

SUB find_sl ios ( )
  BYTE ever, i ' Schleifenvariable
  WORD ibu_fill ' Eingangspufferfuellung
  BYTE frameformat, msg_len ' Frame-Format, Botschaftslaenge
  LONG r_id ' Empfangs-IDreceive ID
  LONG t
  STRING msg$(13), data$(8) ' Botschaft und Daten

  slio_equip = 0 ' fange an mit 'keine SLIO'
  no_of_sl ios = 0
  calib_id = 0aah shl 5 ' ID der Kalibrier-Botschaften

```

```

calib$ = ntos$ ( calib$, 1, -2, calib_id ) ' ID high-Byte zuerst

alive_wait = ALIVETIME/2           ' in der Initphase kurzerzer
run_task keep_alive                 ' sendet Kalibrier-Botschaften
                                   ' und haelt SLIOs synchronisiert

print #LCD, "trying to find SLIOs";
t = ticks()                         ' Anfang des Zeitfensters
while diff_ticks ( t ) < SLIOSLOT ' innerhalb des Zeitfensters
find_cont:
  wait_duration 50
  get #CAN, #0, #UFCl_IBU_FILL, 0, ibu_fill
  if ibu_fill > 2 then               ' wenn mindestens eine Message
    get #CAN, #0, 1, frameformat    ' hole Frame-Info-Byte
    msg_len = frameformat bitand 1111b ' Laenge
    if frameformat bitand 80h = 0 then ' wenn Standard-Frame
      get #CAN, #0, CAN_ID11_LEN, r_id ' hole ID-Bytes
      r_id = byte_mirr ( r_id, 2 )
      r_id = r_id shr 5

      if r_id bitand SLIO_ID_IMASK <> SLIO_FIX_ID then ' wenn keine SLIO-Botschaft
        if msg_len > 0 then ' wenn Daten: wegwerfen
          get #CAN, #0, msg_len, data$ ' hole sie aus dem Puffer
          endif
          goto find_cont ' warte auf naechste Botschaft
        endif
      else ' sonst ist es extended frame
        get #CAN, #0, CAN_ID29_LEN, r_id ' und damit nicht von SLIO
        if msg_len > 0 then ' wenn Daten
          get #CAN, #0, msg_len, data$ ' hole sie aus dem Puffer
          endif
          goto find_cont ' warte auf naechste Botschaft
        endif
      endif
    endif
  -----
  ' Hier: es war SLIO-Botschaft. Finde Adresse heraus und
  ' trage in Bit-Liste ein, verwende die Adresse im Testprogramm

  sliol_id = ( r_id bitand 7FEh ) shl 5 ' nehme diese Adress f. Test
                                           ' maskiere DIR-Bit weg
  r_id = r_id bitand SLIO_ID_MASK ' setzbare SLIO Adressbits
  if bit ( r_id, 8 ) = 1 then ' wenn P8 gesetzt, dann
    r_id = ( r_id bitand 111000b ) + 40h ' erzeuge 4-Bit-Adr.
  endif
  r_id = r_id shr 3 ' schiebe untere 3 ID-Bist weg
  set_bit slio equip, r_id ' setze Bitnummer in Liste
  print #LCD, "<1Bh>A<0><1><0F0h>SLIO ";r_id;" found"
  no_of_slios = no_of_slios + 1 ' zaehle die gefundene SLIO
                                   ' ignoriere hier Info in Daten
                                   ' wenn Daten
  if msg_len > 0 then
    get #CAN, #0, msg_len, data$ ' hole sie aus dem Puffer
    using "UH<2><2> 0.0.0.0.2" ' Format HEX-Ausgabe
    for i = 0 to len ( data$ ) - 1 ' alle Bytes des Strings
      print_using #LCD, asc ( mid$ ( data$, i, 1 ) ); ' als HEX
    next
  endif
endif
endwhile
alive_wait = ALIVETIME           ' normale Wartezeit 'keep_alive'
END

```

Device-Treiber

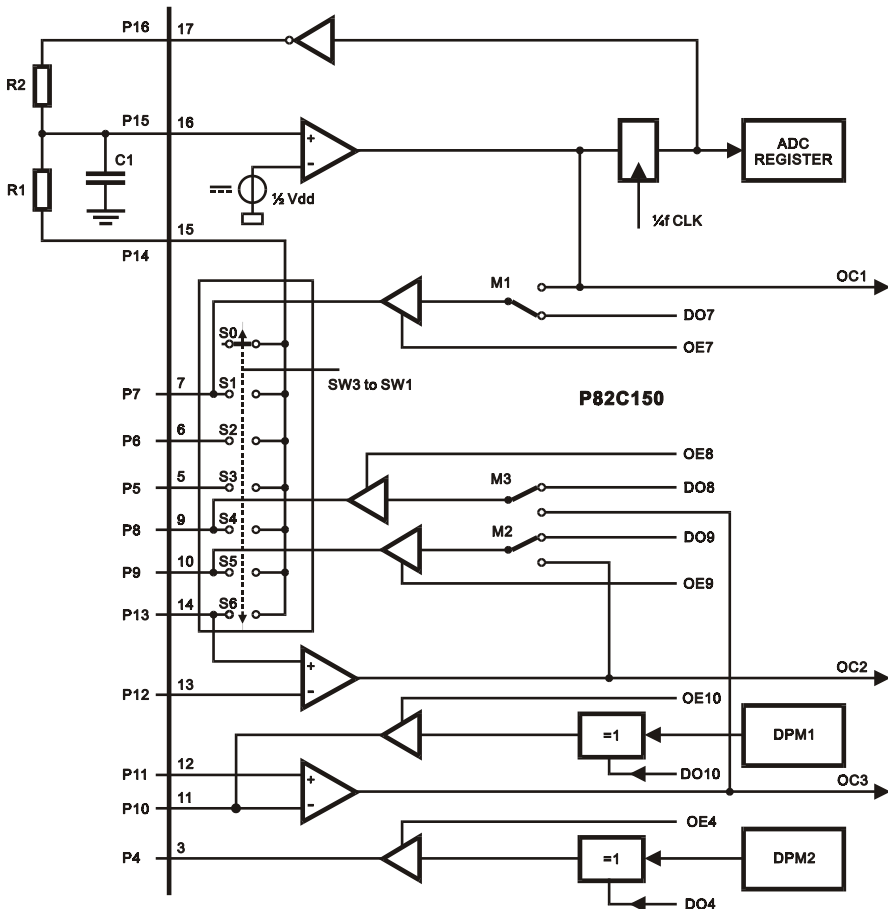
2

```
' Sendet Kalibrier-Botschaften, um die SLIO synchronisiert zu halten.
' Spaetestens alle 8000 Bitzeiten muss eine Kalibrier-Botschaften
' gesendet werden.
-----
TASK keep_alive
  BYTE ever
  LONG t

  t = ticks()
  for ever = 0 to 0 step 0
    if diff_ticks ( t ) > 50 then
      put #CAN, calib$
      t = ticks()
    endif
    if can_out$ <> "" then
      put #CAN, can_out$
      can_out$ = ""
    endif
  next
END
```

Analog-Konfiguration

Der SLIO-Chip stellt einen recht komplexen Aufbau zur Erfassung von analogen Eingangssignalen zur Verfügung. Es wird eine Auflösung von 7..8 Bit erreicht. Die Wandelrate ist genauso hoch wie die Ausgaberate bei der quasi-analogen Ausgabe. Eine Wandlung benötigt 1024 Bitzeiten.



Der A/D-Wandler im oberen Teil der Abbildung verwendet den Pin P15 als Analog-Eingang sowie P16 als Feedback-Ausgang. Die Eingangsbeschaltung besteht aus R1 und R2 (je 100k) und C1 (3,3nF). Auf dem CAN-SLIO-Board steht am Pin Ai15 ein solcher Eingang zur Verfügung. Der Pin P15 muß offen bleiben, wenn Ai15 verwendet wird.

Wird mehr als nur ein Analogeingang benötigt, dann können die Pins P5, P6 P7, P8, P9 und P13 über einen Analogschalter auf den Analogeingang geschaltet werden. Der Ausgang des Analogschalters P14 wird dazu auf den Pin Ai15 gebrückt.

Device-Treiber

Anmerkung: der Pin Vref auf der gleichen 3-poligen Stiftleiste hat nichts mit dem Analogteil des SLIO zu tun. Vref gehört zu der CAN-Schnittstelle.

Eine weitere Möglichkeit des Analogteils besteht darin, mit Hilfe von Komparatoren nur das Über- oder Unterschreiten eines Schwellwertes anzuzeigen und, wenn gewünscht, auch gleichzeitig auf einem Ausgangspin auszugeben. Der Aufbau kann so konfiguriert werden, daß eine CAN-Nachricht erzeugt wird, wenn der Schwellwert über- oder unterschritten wird, oder daß eine selbsttätige Regelung läuft, ohne daß CAN-Nachrichten erzeugt werden. Folgende Komparatoren stehen zur Verfügung:

2

Eingangspins	Ausgang	schaltbar auf Ausgangspin	Bemerkung
P15(+)	OC1	P7	(-) an interner Vref=1/2 VCC (2.5V) Verwendet den Eingang des ADC
P12(+), P13(-)	OC2	P8	
P11(+), P10(-)	OC3	P9	P10 ist auch analog-Ausgang DMP1, der dann nicht mehr verfügbar ist.

Die Ausgänge der Komparatoren können im Register ‚Analog Configuration‘ gelesen werden (Bits 12, 13 und 14). Ob diese Signale auf Ausgänge durchgeschaltet werden, wird in den Bits 8, 9 und 10 des Registers ‚Analog Configuration‘ eingestellt. Außerdem müssen dazu natürlich die Ausgangstreiber der betroffenen Pins aktiviert sein. CAN-Nachrichten werden dann automatisch erzeugt, wenn in den Registern ‚Event Positive Edge‘ und/oder ‚Event Negative Edge‘ die entsprechenden Bits für P8, P9 und P10 gesetzt sind.

Das ‚Analog Configuration‘-Register (Adresse 5):

ADC	OC3	OC2	OC1	0	M3	M2	M1	S3	S2	S1	0	0	0	0	0
-----	-----	-----	-----	---	----	----	----	----	----	----	---	---	---	---	---

Bit(s)			Funktion
0...4			ohne Funktion
S3	S2	S1	Position des Analogschalters
0	0	0	kein Schalter geschlossen
0	0	1	P7 an P14
0	1	0	P6 an P14
0	1	1	P5 an P14
1	0	0	P8 an P14
1	0	1	P9 an P14
1	1	0	P13 an P14
1	1	1	reserviert
			Position der Monitoringschalter
M1			1: OC1 an P7
M2			1: OC3 an P9
M3			1: OC2 an P8
11			ohne Funktion
			Ergebnis der Analog-Komparatoren
OC1			1: P15 > Vref-intern (2.5V)
OC2			1: P13 > P12
OC3			1: P11 > P10
ADC			1: startet A/D-Wandlung

Device-Treiber

Es ergeben sich einige Regeln (in jedem Fall Kurzschlußgefahr):

- Wenn der Anlogschalter gesetzt ist, darf P14 nicht Ausgang sein.
- Wenn M1 gesetzt ist, d.h. OC1 auf P7 geschaltet ist, darf P7 kein Ausgang sein.
- Wenn M2 gesetzt ist, d.h. OC3 auf P9 geschaltet ist, darf P9 kein Ausgang sein.
- Wenn M3 gesetzt ist, d.h. OC2 auf P8 geschaltet ist, darf P8 kein Ausgang sein.

2

Starten der A/D-Wandlung

Eine Wandlung benötigt 1024 Bitzeiten und liefert eine Genauigkeit von 7...8 Bit.

Der A/D-Wandler wird gestartet, wenn

- das Register ‚A/D-Conversion‘ beschrieben wird. Die Antwort erfolgt nach der Conversion automatisch.
- das Register ‚Analog Configuration‘ mit gesetztem Bit ‚ADC‘ beschrieben wird. Die Antwort erfolgt nach der Conversion automatisch.

Das Beispielprogramm liest auf dem Pin P15 gemessene Analogwerte. Die SLIO(s) werden mit Hilfe des Unterprogrammes ‚**find_slios**‘ zusammen mit der Task ‚**keep_alive**‘ gefunden, so wie unter ‚SLIOs auf dem Bus finden‘ beschrieben. Die Task ‚**show_slcio**‘ zeigt die ankommenden SLIO-Nachrichten und den Identifier auf dem LCD in HEX zur Kontrolle an. Fremde Nachrichten, dazu zählen automatisch extended Frames, werden von ‚show_slcio‘ verworfen, es wird nur der Empfangspuffer bereinigt.

Debuggen: wenn Sie SLIO-Beispielprogramme im Debugmodus per Einzelschritt untersuchen, dann geht wahrscheinlich der SLIO in den Sleep-Modus, da die Kalibrierungsnachrichten zu lange ausbleiben. Einzelschritte sollten am Besten gezielt nach einem Breakpoint durchgeführt werden. Danach muß ganz neu gestartet werden, damit der SLIO neu gefunden wird.

Programmbeispiel:

```

'-----
' Name: SLIO_ANA1.TIG
' Liest Analogeingang des SLIO an P15 und zeigt den Wert als
' HEX-Zahl an (0...7FFh).
' Verbinde CAN-SLIO-Board
'-----
user var strict          ' unbedingte Var.deklaration
#include UFUNC3.INC      ' User Function Codes
#include DEFINE_A.INC    ' allg. Symbol-Definitionen
#include CAN.INC         ' CAN-Definitionen
#include CANSLIO.INC     ' Definitionen zum CAN-SLIO-Chip

#define ALIVETIME 50     ' warte ca. 2500 * 20usec
#define SLIOSLOT 1000    ' msec Zeit, sich 'bereit' zu melden

#define NOT_READY 0     ' SLIO-Status 'nicht bereit'
#define READY 1        ' SLIO-Status 'bereit'

'           3 210       ID-Bitpositionen
#define ID_SLIO1 00000000000b ' ID-Bit-Einstellung dieses SLIO

' globale Variable
BYTE no_of_sl ios      ' Zaehler SLIOs im System
BYTE sl io_stat        ' SLIO status
WORD sl io1_id, cal ib_id ' ID der SLIO und der Kalib.-Botsch.
LONG ac_code, ac_mask  ' Access-Code und Access-Mask
LONG sl io equip       ' ein Bit fuer jede SLIO
LONG alive_wait        ' 1/2 Wartezeit in 'keep alive'
WORD acfg              ' Analog-Konfigurationsbits
STRING calib$(5)       ' Kalibrierungsbotschaft
STRING sl io1_doe$(6)  ' data out enable
STRING sl io1_acfg$(6), sl io1_ain$(6) ' analog config, analog in
STRING can_out$(13k)   ' Sendebotschaft zur SLIO

'-----
TASK MAIN
  BYTE ever            ' Endlosschleife
  WORD fi              ' Pufferfuellung
  STRING tmp$(6)

  install_device #LCD, "LCD1.TDD"

' warte auf SLIO-Code, aber filtere 'keep-alive'-Codes aus
' diese Bits stehen linksbuendig in den je 32-Bit Code u. Mask
' 01010000101 <- feste SLIO-Bits (50Ah)
' 00100111000 <- Filter: 1:don't care, 0: Bit muss stimmen (271h)

  install_device #CAN, "CAN1_K1.TDD", &
    "50 A0 00 00 &      ' access code
    27 1F FF FF &      ' access mask
    10 45 &             ' Bitzeit 20usec (moegl.:8...50usec)
    08 1A"%            ' single filter mode, outctrl

' ID dieses SLIO: 0=standard frame
' + 2 ID-Bytes um 5 geschiftet
sl io1_id = (SLIO_FIX_ID bitor ID_SLIO1) shl 5 ' SLIO-Identifizier
calib_id = 0aah shl 5 ' ID der Kalibrier-Botschaften
can_out$ = "" ' muss initialisiert sein

```

```

call find_sl ios          ' stellt Liste aller im System
                          ' befindlichen SLIOs zusammen
if sl io_ equip = 0 then  ' wenn keine SLIO gefunden
  goto no_sl io_ found   ' dann Programm abbrechen
else
  print #LCD, "<1Bh>A<0><2><0F0h>";no_of_sl ios;" SLIOs found";
endif
wait_ duration 1500
print #LCD, "<1>";

run_ task show_sl io     ' zeigt an, was die SLIO(s) senden

sl io1_ doe$ = "<0><0><0><4><0><0>" ' SLIO_OUT_ENABLE alles Inputs
sl io1_ doe$ = ntos$ ( sl io1_ doe$, 1, -2, sl io1_ id)'ID high-Byte zuerst
can_ out$ = sl io1_ doe$ ' ausgeben (lassen)
wait_ duration 100
                          ' bereite die Analogkonfiguration vor
sl io1_ acfg$ = "<0><0><0><5><0><0>" ' SLIO_OUT_ENABLE alles Inputs
sl io1_ acfg$ = ntos$ ( sl io1_ acfg$, 1, -2, sl io1_ id)'ID high-Byte erst
'      aooo-mmmsss-----
acfg = 100000000000000000b  ' nur 'start conversion bit' gesetzt
                          ' ooo = 0, mmm = 0: kein Comparator
                          ' sss = 0, kein Analogschalter
for ever = 0 to 0 step 0  ' Endlosschleife -----
  tmp$ = ntos$ ( sl io1_ acfg$, 4, -2, acfg)' Konfig.wort einfüegen
  can_ out$ = tmp$        ' ausgeben (lassen), startete A/D
  wait_ duration 1000
next                      ' Endlosschleife -----

no_sl io_ found:         ' keine SLIO gefunden
  stop_ task keep_ alive
  print #LCD, "<1>no SLIO found"
  print #LCD, "program terminated"
END

-----
' Zeigt Inhalte der SLIO-Botschaften an
' Zeile 1 auf LCD: ID
' Zeile 2 auf LCD: Datenbytes in HEX
-----

TASK show_sl io
  BYTE ever, i           ' Schleifenvariable
  BYTE frameformat, msg_ len ' Frame-Format, Botschaftslaenge
  WORD ibu_ fill         ' Eingangspufferfuellung
  WORD value             ' Analogwert
  LONG r_ id             ' Empfangs-ID
  STRING msg$(13), data$(8) ' Botschaft und Daten

  for ever = 0 to 0 step 0 ' Endlosschleife
rx_ cont:
  get #CAN, #0, #UFCI_IBU_FILL, 0, ibu_ fill
  if ibu_ fill > 2 then    ' wenn mindestens eine Message
    get #CAN, #0, 1, frameformat ' hole Frame-Info-Byte
    msg_ len = frameformat bitand 1111b ' Laenge
    if frameformat bitand 80h = 0 then ' wenn Standard-Frame
      get #CAN, #0, CAN_ID11_LEN, r_ id ' hole ID-Bytes
      r_ id = byte_mirr ( r_ id, 2 )
      r_ id = r_ id shr 5
                          ' wenn keine SLIO-Botschaft

```

```

        if msg_len > 0 then          ' wenn Daten: wegwerfen
            get #CAN, #0, msg_len, data$ ' hole sie aus dem Puffer
        endif
        goto rx_cont                ' warte auf naechste Botschaft
    endif
else                                ' sonst ist es extended frame
    get #CAN, #0, CAN_ID29_LEN, r_id ' und damit nicht von SLIO
    if msg_len > 0 then              ' wenn Daten
        get #CAN, #0, msg_len, data$ ' hole sie aus dem Puffer
    endif
    goto rx_cont                    ' warte auf naechste Botschaft
endif
' -----
' Hier: es war SLIO-Botschaft
using "UH<3><3>  0 0 0 0 3"         ' fuer ID Anzeige
print_using #LCD, "<1Bh>A<0><1><0F0h> ID:";r_id ' ID des SLIO
if msg_len = 3 then                ' wenn Daten
    get #CAN, #0, msg_len, data$   ' hole sie und zeige an
    if nfrows ( data$, 0, 1 ) = 8 then ' wenn Analogregister if anal
        value = nfrows ( data$, 1, 2 )
        value = byte_mirr ( value, 2 ) shr 5
        using "UH<3><3>  0.0.0.0.3" ' Format HEX-Ausgabe
        print_using #LCD, " P15:";value;
    endif
endif ' msg_len
endif ' ibu_fill
next
END

' -----
' Sendet Kalibrier-Botschaften und wartet auf
' (gefilterte) SLIO-Botschaften als Antwort
' Innerhalb einer gesetzten Zeit muessen alle SLIOs sich gemeldet
' haben. Die Filterung muss im Hauptprogramm gesetzt sein.
' Nicht-SLIO-Botschaften und extended Frames werden verworfen,
' d.h. lediglich aus dem Puffer geräumt.
' -----
SUB find_sl ios (
    BYTE ever, i                    ' Schleifenvariable
    WORD ibu_fill                   ' Eingangspufferfuellung
    BYTE frameformat, msg_len      ' Frame-Format, Botschaftslaenge
    LONG r_id                       ' Empfangs-IDreceive ID
    LONG t
    STRING msg$(13), data$(8)      ' Botschaft und Daten

    sl io_equip = 0                 ' fange an mit 'keine SLIO'
    no_of_sl ios = 0
    calib_id = 0aah shl 5           ' ID der Kalibrier-Botschaften
    calib$ = "<0><0><0><0aah><4>"    ' spezieller String fuer Init-SLIO
    calib$ = ntos$ ( calib$, 1, -2, calib_id ) ' ID high-Byte zuerst

    alive_wait = ALIVETIME/2      ' in der Initphase kuerzer
    run_task keep_alive           ' sendet Kalibrier-Botschaften
    ' und haelt SLIOs synchronisiert

    print #LCD, "trying to find SLIOs";
    t = ticks()                    ' Anfang des Zeitfensters
    while diff_ticks ( t ) < SLIOSLOT ' innerhalb des Zeitfensters
find_cont:
        wait_duration 50
        get #CAN, #0, #UFCI_IBU_FILL, 0, ibu_fill

```

```

get #CAN, #0, 1, frameformat      ' hole Frame-Info-Byte
msg_len = frameformat bitand 1111b ' Laenge
if frameformat bitand 80h = 0 then ' wenn Standard-Frame
  get #CAN, #0, CAN_ID11_LEN, r_id ' hole ID-Bytes
  r_id = byte_mirr ( r_id, 2 )
  r_id = r_id shr 5

  ' wenn keine SLIO-Botschaft
  if r_id bitand SLIO_ID_IMASK <> SLIO_FIX_ID then ' aus Puffer
    if msg_len > 0 then ' wenn Daten: wegwerfen
      get #CAN, #0, msg_len, data$ ' hole sie aus dem Puffer
    endif
    goto find_cont ' warte auf naechste Botschaft
  endif
else ' sonst ist es extended frame
  get #CAN, #0, CAN_ID29_LEN, r_id ' und damit nicht von SLIO
  if msg_len > 0 then ' wenn Daten
    get #CAN, #0, msg_len, data$ ' hole sie aus dem Puffer
  endif
  goto find_cont ' warte auf naechste Botschaft
endif

-----
' Hier: es war SLIO-Botschaft. Finde Adresse heraus und
' trage in Bit-Liste ein, verwende die Adresse im Testprogramm

slio1_id = ( r_id bitand 7FEh) shl 5 ' nehme diese Adress f. Test
' maskiere DIR-Bit weg
r_id = r_id bitand SLIO_ID_MASK ' setzbare SLIO Adressbits
if bit ( r_id, 8 ) = 1 then ' wenn P8 gesetzt, dann
  r_id = ( r_id bitand 111000b ) + 40h
endif
r_id = r_id shr 3 ' schiebe untere 3 ID-Bist weg
set bit slio equip, r_id ' setze Bitnummer in Liste
print #LCD, "<1Bh>A<0><1><0F0h>SLIO ";r_id;" found"
no_of_slios = no_of_slios + 1 ' zaehle die gefundene SLIO
' ignoriere hier Info in Daten
if msg_len > 0 then ' wenn Daten
  get #CAN, #0, msg_len, data$ ' hole sie aus dem Puffer
  using "UH<2><2> 0.0.0.0.2" ' Format HEX-Ausgabe
  for i = 0 to len ( data$ ) - 1 ' alle Bytes des Strings
    print_using #LCD, asc ( mid$ ( data$, i, 1 ) );' als HEX
  next
endif
endif
endwhile
alive_wait = ALIVETIME ' normale Wartezeit 'keep_alive'
END

-----
' Sendet Kalibrier-Botschaften, um die SLIO synchronisiert zu halten.
' Spaetestens alle 8000 Bitzeiten muss eine Kalibrier-Botschaften
' gesendet werden.
-----

TASK keep_alive
  BYTE ever

  for ever = 0 to 0 step 0 ' Endlosschleife
    put #CAN, calib$ ' sende Kalibrier-Botschaft
    wait_duration alive wait ' warte max 4000 Bitzeiten
    if can_out$ <> "" then

```

```
    can_out$ = ""           ' leer = gesendet
  endif
  wait_duration alive_wait  ' warte max 4000 Bitzeiten
next
END
```

Device-Treiber

Zwei SLIOs an einem Bus

Das folgende Beispiel ist eine Kombination aus SLIO_DPM1.TIG und SLIO_HIGH_LOW.TIG. Es erzeugt DPM-Outputs auf SLIO1 (digital->analog) und digitale Rechteck-Ausgabe auf SLIO2. Stellen Sie auf einem CAN-SLIO-Board alle DIP-Schalter auf OFF, auf dem anderen CAN-SLIO-Board nur DIP-Schalter ID3 auf ON.

2

Programmbeispiel:

```

'-----
' Name: SLIO 2.TIG
' 2 SLIOs im System
' erzeugt DPM-Outputs auf SLIO1 (digital->analog)
' erzeugt digitale Ausgabe auf SLIO2 (Rechteck)
' Verbinde
' 1 CAN-SLIO-Board, alle DIP-Schalter aus
' 1 CAN-SLIO-Board, nur DIP-Schalter ID3 an
' messe am Pin P10
'-----
user var strict          ' unbedingte Var.deklaration
#include UFUN3.INC       ' User Function Codes
#include DEFINE_A.INC   ' allg. Symbol-Definitionen
#include CAN.INC        ' CAN-Definitionen
#include CANSLIO.INC    ' Definitionen zum CAN-SLIO-Chip

#define ALIVETIME 50    ' warte ca. 2500 * 20usec
#define SLIOSLOT 1000  ' msec Zeit, sich 'bereit' zu melden

#define NOT_READY 0    ' SLIO-Status 'nicht bereit'
#define READY 1       ' SLIO-Status 'bereit'

'                   3 210          ID-Bitpositionen
#define ID_SLIO1 00000000000b     ' ID-Bit-Einstellung des SLIO1
#define ID_SLIO2 00000001000b     ' ID-Bit-Einstellung des SLIO2

BYTE no_of_slios          ' Zaehler SLIOs im System
BYTE slio_12_found       ' Flag ueber gefundene SLIOs 1 und 2
BYTE slio_stat           ' SLIO status
WORD slio1_id, slio2_id, calib_id ' ID der SLIOs und der Kalib.-Botsch.
LONG ac_code, ac_mask    ' Access-Code und Access-Mask
LONG slio_equip         ' ein Bit fuer jede SLIO
LONG alive_wait         ' 1/2 Wartezeit in 'keep_alive'
STRING calib$(5)        ' Kalibrierungsbotschaft
STRING slio1_dpml$(6), slio1_doe$(6) ' DPM1 out, data out enable
STRING slio2_doe$(6)    ' data out enable SLIO2
STRING slio1_dout$
STRING slio2_dout$
STRING can_out$(13)     ' Sendebotschaft

'-----
TASK MAIN
BYTE ever              ' Endlosschleife
WORD fi               ' Pufferfuellung
WORD value, tmp       ' Analogwert
LONG t               ' 'keep_alive' Zeit
    LONG digi        ' fuer digitalen Ausgang
STRING tmp$(6)

install_device #LCD, "LCD1.TDD" ' LCD-Treiber installieren

' warte auf SLIO-Code, aber filtere 'keep-alive'-Codes aus
' diese Bits stehen linksbuendig in den je 32-Bit Code u. Mask
' 01010000101 <- feste SLIO-Bits (50Ah)
' 00100111000 <- Filter: 1:don't care, 0: Bit muss stimmen (271h)

install_device #CAN, "CAN1_K1.TDD", &
    "50 A0 00 00 &          ' access code

```

```

10 45 &                                ' Bitzeit 20usec (moegl.:8...50usec)
08 1A"%                                  ' single filter mode, outctrl

                                        ' ID dieses SLIO: 0=standard frame
                                        ' + 2 ID-Bytes um 5 geschiftet
calib_id = 0aah shl 5                    ' ID der Kalibrier-Botschaften
can_out$ = ""                             ' muss initialisiert sein

call find_sl ios                          ' stellt Liste aller im System
                                        ' befindlichen SLIOs zusammen
if sl io equip = 0 then                    ' wenn keine SLIO gefunden
    goto no_sl io_found                    ' dann Programm abbrechen
else
    print #LCD, "<1Bh>A<0><2><0F0h>";no_of_sl ios;" SLIOs found";
endif
if sl io equip bitand 000000011b <> 3 then ' wenn SLIO1/2 n. gefunden
    print #LCD, "<1Bh>A<0><3><0F0h>SLIO1/2 not found";
endif
wait_duration 1500
print #LCD, "<1>";

sl io1_id = (SLIO_FIX_ID bitor ID_SLIO1) shl 5 ' SLIO1-Identifizier
sl io2_id = (SLIO_FIX_ID bitor ID_SLIO2) shl 5 ' SLIO2-Identifizier
run_task show_sl io                        ' zeigt an, was die SLIO(s) senden

sl io1_doe$ = "<0><0><0><4><0FFh><0FFh>" ' SLIO1_OUT_ENABLE
sl io1_doe$ = ntos$ ( sl io1_doe$, 1, -2, sl io1_id)'ID high-Byte zuerst
can_out$ = sl io1_doe$
wait_duration 1000

sl io1_dout$ = "<0><0><0><3><0FFh><0FFh>" ' SLIO1_DATA_OUT
sl io1_dout$ = ntos$ ( sl io1_dout$, 1, -2, sl io1_id)' ID high-Byte
can_out$ = sl io1_dout$
wait_duration 1000
sl io1_dout$ = "<0><0><0><3><0><0>" ' SLIO1_DATA_OUT
sl io1_dout$ = ntos$ ( sl io1_dout$, 1, -2, sl io1_id)' ID einbauen
can_out$ = sl io1_dout$
wait_duration 1000

sl io2_doe$ = "<0><0><0><4><0FFh><0FFh>" ' SLIO2_OUT_ENABLE
sl io2_doe$ = ntos$ ( sl io2_doe$, 1, -2, sl io2_id)'ID high-Byte zuerst
can_out$ = sl io2_doe$
sl io2_dout$ = "<0><0><0><3><0FFh><0FFh>" ' SLIO2_DATA_OUT
sl io2_dout$ = ntos$ ( sl io2_dout$, 1, -2, sl io2_id)' ID einbauen
wait_duration 1000

sl io1_dpml$ = "<0><0><0><6><0><0>" ' SLIO1_DPM1
sl io1_dpml$ = ntos$ ( sl io1_dpml$, 1, -2, sl io1_id)' ID high-Byte
value = 0
digi = 0                                  ' Wert fuer digitalen Ausgang
t = ticks()
stop_task keep_alive                       ' Hauptschleife uebernimmt das jetzt
for ever = 0 to 0 step 0                    ' Endlosschleife -----
    tmp = value shl 5                       ' 10-Bit rechtbuendig in WORD
    tmp$ = ntos$ ( sl io1_dpml$, 4, -2, tmp )' value high-Byte zuerst
    put #CAN, tmp$
    tmp$ = ntos$ ( sl io2_dout$, 4, -2, digi )' value high-Byte zuerst
    put #CAN, tmp$
    digi = digi + 1                          ' dig. Ausgangswert aendern
    value = modulo_inc ( value, 0, 7FFh, 7Fh ) ' inc step 7Fh

```

```

        put #CAN, calib$                ' sende Kalibrier-Botschaft
        t = ticks()
    endif
next                                     ' Endlosschleife -----

no_slcio_found:                          ' keine SLIO gefunden
    stop_task keep_alive
    print #LCD, "<l>no SLIO found"
    print #LCD, "program terminated"
END

'-----
' Zeigt Inhalte der SLIO-Botschaften an
' Zeile 1 auf LCD: ID
' Zeile 2 auf LCD: Datenbytes in HEX
'-----

TASK show_slcio
    BYTE ever, i                        ' Schleifenvariable
    WORD ibu_fill                       ' Eingangspufferfuellung
    BYTE frameformat, msg_len          ' Frame-Format, Botschaftslaenge
    LONG r_id                           ' Empfangs-IDreceive ID
    STRING msg$(13), data$(8)         ' Botschaft und Daten

    for ever = 0 to 0 step 0           ' Endlosschleife
rx_cont:
    get #CAN, #0, #UFCl_IBU_FILL, 0, ibu_fill
    if ibu_fill > 2 then               ' wenn mindestens eine Message
        get #CAN, #0, 1, frameformat  ' hole Frame-Info-Byte
        msg_len = frameformat bitand 1111b ' Laenge
        if frameformat bitand 80h = 0 then ' wenn Standard-Frame
            get #CAN, #0, CAN_ID11_LEN, r_id ' hole ID-Bytes
            r_id = byte_mirr ( r_id, 2 )
            r_id = r_id shr 5

            ' wenn keine SLIO-Botschaft
            if r_id bitand SLIO_ID_IMASK <> SLIO_FIX_ID then ' aus Puffer
                if msg_len > 0 then ' wenn Daten: wegwerfen
                    get #CAN, #0, msg_len, data$ ' hole sie aus dem Puffer
                endif
                goto rx_cont ' warte auf naechste Botschaft
            endif
        else ' sonst ist es extended frame
            get #CAN, #0, CAN_ID29_LEN, r_id ' und damit nicht von SLIO
            if msg_len > 0 then ' wenn Daten
                get #CAN, #0, msg_len, data$ ' hole sie aus dem Puffer
            endif
            goto rx_cont ' warte auf naechste Botschaft
        endif
        '-----
        ' Hier: es war SLIO-Botschaft
        using "UH<3><3> 0 0 0 0 3" ' fuer ID Anzeige
        print_using #LCD, "<l>1h>A<0><l><0F0h> ID:";r_id ' ID des SLIO
        print #LCD, " <l3>DATA:"; ' loesche Datenanz.
        if msg_len > 0 then ' wenn Daten
            get #CAN, #0, msg_len, data$ ' hole sie und zeige an
            using "UH<2><2> 0.0.0.0.2" ' Format HEX-Ausgabe
            for i = 0 to len ( data$ ) - 1 ' alle Bytes des Strings
                print_using #LCD, asc ( mid$ ( data$, i, 1 ) );' als HEX
            next
        endif
    endif
endif

```

END

```

-----
' Sendet Kalibrier-Botschaften und wartet auf
' (gefilterte) SLIO-Botschaften als Antwort
' Innerhalb einer gesetzten Zeit muessen alle SLIOs sich gemeldet
' haben. Die Filterung muss im Hauptprogramm gesetzt sein.
' Nicht-SLIO-Botschaften und extended Frames werden verworfen,
' d.h. lediglich aus dem Puffer geräumt.
-----
SUB find_sl ios ( )
  BYTE ever, i                ' Schleifenvariable
  WORD ibu_fill               ' Eingangspufferfuellung
  BYTE frameformat, msg_len   ' Frame-Format, Botschaftslaenge
  LONG r_id                   ' Empfangs-IDreceive ID
  LONG t
  STRING msg$(13), data$(8)   ' Botschaft und Daten

  sl io_equip = 0              ' fange an mit 'keine SLIO'
  no_of_sl ios = 0
  calib_id = 0aah shl 5        ' ID der Kalibrier-Botschaften
  calib$ = "<0><0><0><0aah><4>" ' spezieller String fuer Init-SLIO
  calib$ = ntos$( calib$, 1, -2, calib_id )' ID high-Byte zuerst

  alive_wait = ALIVETIME/2    ' in der Initphase kuerzer
  run_task keep_alive         ' sendet Kalibrier-Botschaften
                              ' und haelt SLIOs synchronisiert

  print #LCD, "trying to find SLIOs";
  t = ticks()                  ' Anfang des Zeitfensters
  while diff_ticks ( t ) < SLIOSLOT ' innerhalb des Zeitfensters
  find_cont:
    wait_duration 50
    get #CAN, #0, #UFCI_IBU_FILL, 0, ibu_fill
    if ibu_fill > 2 then        ' wenn mindestens eine Message
      get #CAN, #0, 1, frameformat' hole Frame-Info-Byte
      msg_len = frameformat bitand 1111b ' Laenge
      if frameformat bitand 80h = 0 then ' wenn Standard-Frame
        get #CAN, #0, CAN_ID11_LEN, r_id ' hole ID-Bytes
        r_id = byte_mirr ( r_id, 2 )
        r_id = r_id shr 5
                              ' wenn keine SLIO-Botschaft
      if r_id bitand SLIO_ID_IMASK <> SLIO_FIX_ID then ' aus Puffer
        if msg_len > 0 then    ' wenn Daten: wegwerfen
          get #CAN, #0, msg_len, data$' hole sie aus dem Puffer
        endif
        goto find_cont        ' warte auf naechste Botschaft
      endif
    else                        ' sonst ist es extended frame
      get #CAN, #0, CAN_ID29_LEN, r_id' und damit nicht von SLIO
      if msg_len > 0 then      ' wenn Daten
        get #CAN, #0, msg_len, data$ ' hole sie aus dem Puffer
      endif
      goto find_cont          ' warte auf naechste Botschaft
    endif
  -----
  ' Hier: es war SLIO-Botschaft. Finde Adresse heraus und
  ' trage in Bit-Liste ein, verwende die Adresse im Testprogramm

  sl io1_id = ( r_id bitand 7FEh) shl 5 ' nehme diese Adress f. Test

```

```

r_id = r_id bitand SLIO_ID_MASK      ' setzbare SLIO Adressbits
if bit ( r_id, 8 ) = 1 then          ' wenn P8 gesetzt, dann
  r_id = ( r_id bitand 111000b ) + 40h ' erzeuge 4-Bit-Adr.
endif
r_id = r_id shr 3                    ' schiebe untere 3 ID-Bist weg
set_bit slio equip, r_id            ' setze Bitnummer in Liste
'
  print #LCD, "<1Bh>A<0><1><0F0h>SLIO ";r_id;" found"
  using "UH<1><1>  0 0 0 0 1"        ' fuer slio equip Anzeige
  print_using #LCD, "<1Bh>A<0><1><0F0h>SLIO equip ";slio equip
  no_of_slis = no_of_slis + 1        ' zaehle die gefundene SLIO
'                                     ' ignoriere hier Info in Daten
                                     ' wenn Daten
if msg_len > 0 then
  get #CAN, #0, msg_len, data$       ' hole sie aus dem Puffer
'                                     ' Format HEX-Ausgabe
  using "UH<2><2>  0.0.0.0.2"
  for i = 0 to len ( data$ ) - 1     ' alle Bytes des Strings
    print_using #LCD, asc ( mid$ ( data$, i, 1 ) );' als HEX
  next
endif
endif
endwhile
alive_wait = ALIVETIME              ' normale Wartezeit 'keep_alive'
END

'-----
' Sendet Kalibrier-Botschaften, um die SLIO synchronisiert zu halten.
' Spaetestens alle 8000 Bitzeiten muss eine Kalibrier-Botschaften
' gesendet werden.
'-----
TASK keep_alive
BYTE ever
LONG t

t = ticks()
for ever = 0 to 0 step 0              ' Endlosschleife
  if diff_ticks ( t ) > 50 then      ' Wert haengt von Bitzeit ab
    put #CAN, calib$                 ' sende Kalibrier-Botschaft
    t = ticks()
  endif
  if can_out$ <> "" then
    put #CAN, can_out$               ' Sendebotschaft
    can_out$ = ""                   ' leer = gesendet
  endif
next
END

```