

D. Module.CAN

Doc #2.0

User's Guide

D. SignT

Digital Signalprocessing Technology

Norbert Nölker & Adolf Klemenz GbR

Gelderner Straße 36

D - 47647 Kerken

phone +49 (0) 2833 / 570 977

fax +49 (0) 2833 / 33 28

email info@dsigt.de

www <http://www.dsigt.de>

D.Module.CAN Revision History

1.0 November 1999 Engineering Kit

2.0 February 2002 D.Module

1 INTRODUCTION	4
2 SPI INTERFACE	5
3 LAYOUT AND JUMPER SETTINGS	7
3.1 CAN Bus Connector.....	7
3.2 Bit-I/O Ports.....	8
3.3 SPI Interrupts.....	8
3.4 CAN Interrupts	9
4 POWER SUPPLY	10
5 CAN COMMUNICATIONS LIBRARY	11
5.1 Communications Protocol	11
5.2 Verifying SPI Communications.....	14
5.3 Including the CAN Library in your Projects	15
5.4 CAN Controller Interrupts.....	16
5.5 CAN Library Functions and Macros.....	17
5.5.1 Initialisation.....	17
5.5.2 Acceptance Filtering and Message Object Set-Up.....	18
5.5.3 Data Transfer Functions.....	19
5.5.4 Message Status and Control	20
5.5.5 Low Level Functions.....	20
5.5.6 Macros.....	21
6 SUPPORT SOFTWARE	22
6.1 CAN Terminal	22
6.1.1 Usage	22
6.1.2 Configuration Dialogue.....	23
6.1.3 Acceptance Filtering Dialogue	24
6.1.4 Communications Dialogue	25
6.1.5 Customising CAN Terminal	26

6.2 Remote.....	26
7 APPENDIX MECHANICS AND PINOUT.....	27
8 ELECTRICAL SPECIFICATIONS.....	29
Figure 2-1 Basic SPI Master Interface.....	5
Figure 2-2 SPI Interface to 82527	6
Figure 3-1 Location of Jumpers and Connectors.....	7
Figure 4-1 Power Supply Connections for 3.3 and 5V DSP systems.....	10
Figure 6-1 CAN Terminal Configuration Dialogue	23
Figure 6-2 Acceptance Filtering Dialogue.....	24
Figure 6-3 Communications Dialogue.....	25
Figure 7-1 Mechanical Dimensions	27
Table 2-1 SPI Communication Protocol	6
Table 3-1 SPI Interrupt Jumper Settings	8
Table 6-1 Pinout.....	28

1 Introduction

The D.Module.CAN is a daughter board for interfacing an Intel™ 82527 CAN Controller via SPI interface to a D.Module DSP Computer Module. This manual assumes you are already familiar with the CAN bus protocol. Further recommended reading is the Intel 82527 Serial Communications Controller Architectural Overview.

The SPI interface was chosen for the following reasons:

1. The CAN controller has a very slow bus interface. Typical access times range from 280 to 540 nsecs., depending on the register and the previous instruction. For many DSP applications, these access times will provide severe limitations, e.g. samples from a high speed A/D Converter may be lost because the CAN controller occupies the bus too long. The SPI interface, implemented in the D.Module's CPLD, on the contrary, allows zero wait state access.
2. Often several peripherals have to be connected to the external bus, each with it's own decoder and additional glue logic. The SPI interface can be implemented in the D.Module's CPLD without glue logic.
3. Only four connections between DSP board and CAN controller are required to establish the communication path which simplifies layout and system integration.
4. During the 280..540 nsecs, modern DSPs can execute a lot of useful instructions instead of waiting to finish a slow bus cycle. The SPI implementation allows to set up a complete transfer, e.g. write an entire message, in a few cycles. Communication with the CAN controller then takes place as an interrupt driven background task, allowing the DSP to proceed with useful operations.

The D.Module.CAN requires an SPI interface programmed into the DSP Module user-programmable CPLD. Suitable implementations and programming files are shipped with the support software.

2 SPI Interface

The SPI interface uses synchronous serial data transfers. The hardware is based on an 8 Bit shift register with parallel load/store. Data is shifted in and out on opposite clock edges. Clocks are generated by the 'master' device only, i.e. if the master wants to read data from the slave device, it has to provide the serial clock. In this application, the DSP is the master and the CAN controller is the slave device. Clocks are 'gated', i.e. the serial clock is only present during data transfers. Between data transfers the clock remains in it's idle state.

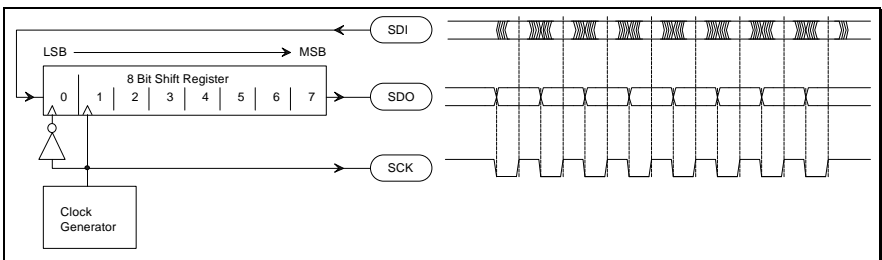


Figure 2-1 Basic SPI Master Interface

SPI interfaces are often configurable in terms of active clock edges and the clock's idle state. This implementation assumes the clock idle state to be 'high' and data to be shifted out on the falling clock edge, as shown in the diagram. The clock may be as high as 4.2 MHz. To simplify the design, we have chosen to operate the CAN controller from an 8 MHz crystal and fed it's CLKOUT signal to the CPLD. The CPLD then generates a 4 MHz serial clock from this master clock, resulting in almost the highest possible transmission rate, independent of the DSP's clock.

The suitable CPLD programming files are part of the D.Module.CAN support software. To program the CPLD, use the <p> command in Set-Up Mode, as described in the D.Module User's Guide and upload the programming file. For more information about the CPLD SPI implementation, please refer to the source files and the readme.txt file.

The SPI interface on the D.Module is interrupt driven. Each time an 8 Bit word is completely shifted in/out, an interrupt request is generated. Therefore, an Interrupt input of the D.Module has to be connected to the CPLD. On the D.Module.21065, D.Module.VC33 and D.Module.C6x01 Rev. 2, this can be configured in the Module Configuration Register. The D.Module.C31eco and first revision D.Module.C6x01 require an external connection as shown in the schematics.

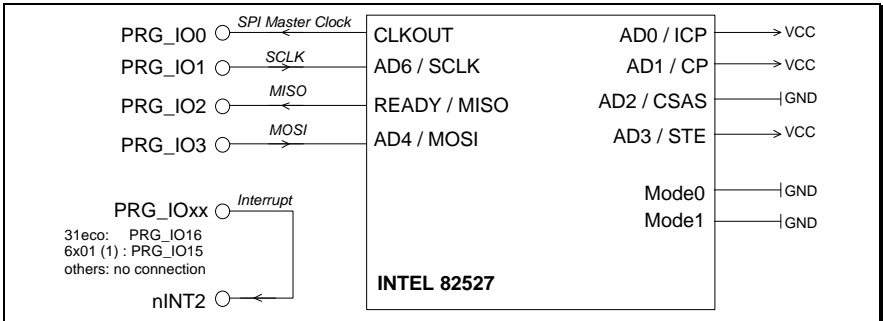


Figure 2-2 SPI Interface to 82527

SPI clocks are generated by the master simultaneously with shifting out data. If data should be read from the slave, the master must transmit some (don't care) data to generate the serial clock. The slave device ignores this data. This behaviour calls for a communication protocol to synchronise transfers. In the case of the 82527 CAN controller, two protocol bytes precede the actual data. These bytes determine the target register, the direction of the data transfer, and the number of data bytes to follow. Up to 15 data bytes can be transmitted in a single block.

Start Address	Serial Control Byte	Data Byte 1	...	Data Byte 15
0x00 .. 0xFE	Bit 3..0: data length (1..15) Bit 6..4: always 0 Bit 7: Dir (0-read 82527, 1-write 82527)	data written to successive addresses, starting at the specified start address		

Table 2-1 SPI Communication Protocol

3 Layout and Jumper Settings

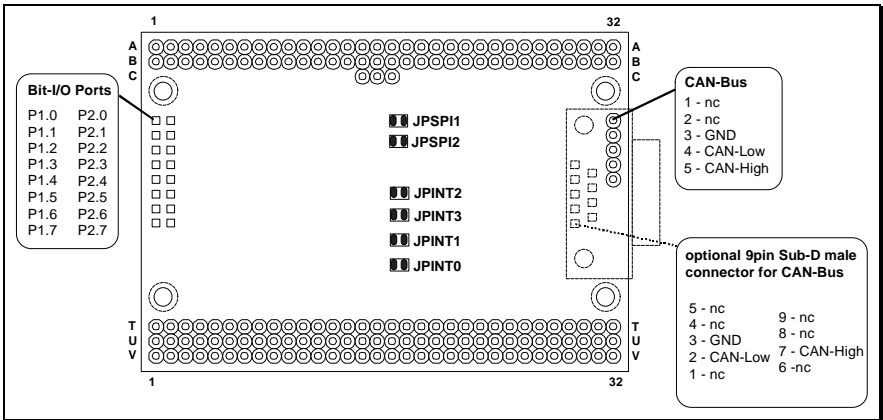


Figure 3-1 Location of Jumpers and Connectors

3.1 CAN Bus Connector

The CAN bus is accessible either via a 5-pin connector on the right of the module, or via an optionally mounted 9-pin male D-Sub connector. If the D-Sub connector is used, the pin assignment follows the recommendations of CIA DR-303-1. If the D.Module.CAN is mounted on the D.Base2 base board, the 5-pin header will route the CAN-Bus signals to the RJ45 connector on the base board. According to CIA DR-303-1, pin assignment is 1- CAN-High, 2 - CAN-Low, 3 - GND, pins 4..8 on the RJ45 are not connected.

To mount the D-Sub connector, the 5-pin connector has to be removed. It is a "press-fit" type connector and can easily be dismounted by holding each pin individually with small pliers on the bottom side of the module and applying some pressure to it.

The CAN bus is driven by a ISO/DIS 11898 compatible P82C250 transceiver which allows data rates up to 1 MBit/s and a maximum of 110 bus nodes.

3.2 Bit-I/O Ports

The Intel AN82527 CAN controller provides 16 individually configurable Bit-I/O signals, grouped in two 8 bit wide ports. These ports are configurable via the P1CONF and P2CONF registers, and are read and written via P1IN, P1OUT, resp. P2IN, P2OUT registers. The corresponding signals are accessible on a 2*8 pin connector on the left of the module. Please note that these ports use open drain drivers: 10 K Pull-up resistors on the module provide the high level. Source current is therefore limited to 260 μ A to stay within TTL limits. If a port is configured as output and driven low, the maximum sink current is 1.6 mA

3.3 SPI Interrupts

To notify the DSP if a SPI transfer is completed, interrupts are used. This interrupt is generated in the DSP module's User-CPLD. Most D.Module DSP boards allow to route this signal to a DSP interrupt input via the Module Configuration Register. Some older designs do however require an external connection:

DSP Module	JPSP11	JPSP12
D.Module.C31eco	open	closed
D.Module.C6x01, Rev. 1	closed	open
all others	open	open

Table 3-1 SPI Interrupt Jumper Settings

The D.Module.CAN support software expects nINT2 to be used for SPI interrupts. Please make the appropriate changes to the software drivers if a different signal should be used.

3.4 CAN Interrupts

The CAN Controller can be instructed to generate an interrupt to the DSP in case of various events: Reception of a message, transmission of a message, reception of a remote frame, error conditions etc. You may select one of the four external DSP interrupts by closing one of the solder jumpers JPINT0...JPINT3.

Note that one interrupt is already in use for SPI communications. Typically, this is DSP interrupt nINT2. Make sure not to use this signal for CAN interrupts, unless you changed the SPI interrupt to a different signal.

4 Power Supply

The CAN controller and the CAN bus transceiver require an operating voltage of 5V. Since most DSP designs now work from 3.3V, an extra power supply is required for the D.Module.CAN.

VCC (+5V) is fed to the D.Module.CAN via pin C17, labelled +AVCC. GND (0V) must be provided via pin C16 (AGND). This GND signal is internally connected to the DSP board GND on pin A32/B1.

If the module is mounted on the D.Base base board, please use the following connections:

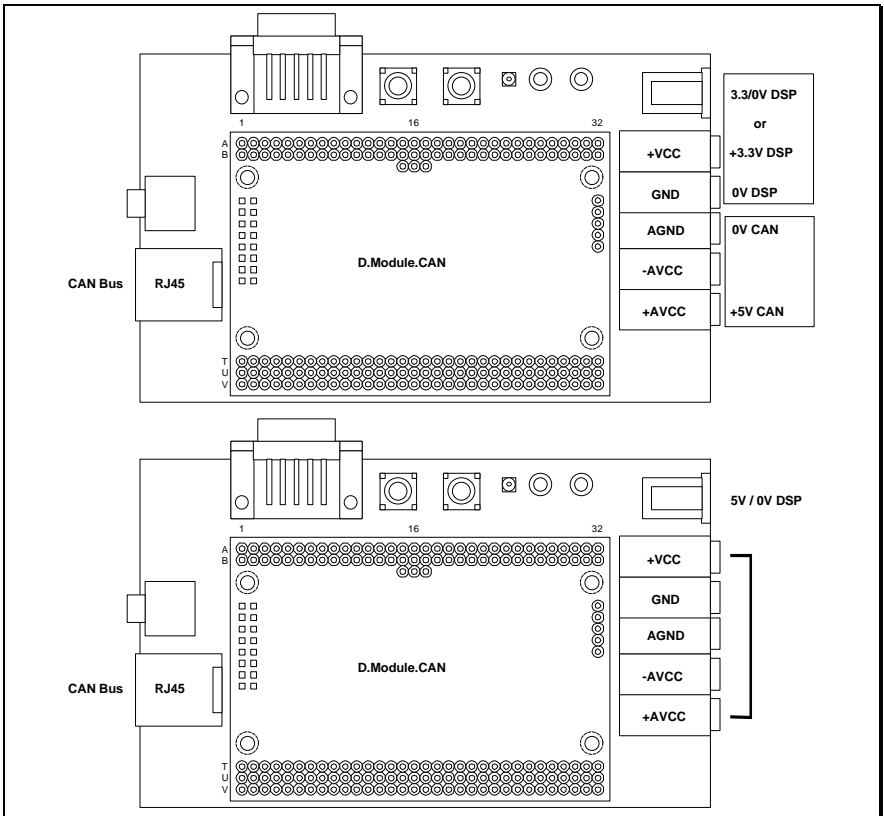


Figure 4-1 Power Supply Connections for 3.3 and 5V DSP systems

5 CAN Communications Library

The CAN communications library (`can_lib.c` and `can_lib.h`) provides basic functions for initialisation, defining acceptance filtering masks, setting up message objects, transmitting and receiving data, as well as low level functions for direct access to the 82527 registers and memory.

5.1 Communications Protocol

Data transfer to/from the CAN controller is interrupt driven. The SPI interface, implemented in the D.Module's CPLD, will assert `nINT2` every time an 8 Bit word is shifted in/out completely. This allows to set up complete transfer packets and perform SPI communication as a background task.

The CAN communications library uses two communication buffers, one holding the data to be send to the 82527, the other holding the received data. The first entry in the transmit buffer defines the total count of data to follow. This way, multiple transfers can be set up in a single packet. If, for example, data should be written to a transmit message object, the following steps are required (the example assumes message object 1 should transmit 4 bytes of data using a standard identifier format):

1. notify the CAN controller that the data in this message object is going to be updated. This will prevent the CAN controller from accessing this message object. If required, reset the Interrupt Pending Flag of this message object.
2. Update the message object's Data Length Code and Data Buffer.
3. notify the CAN controller that update is finished. Now the CAN controller is allowed to access this message object again.

Using the 82527 SPI communication protocol described in the previous section, each of these steps requires it's own transfer:

1. Start Address: Message Object 1 Control 0 Register (0x10)
 Serial Control Byte: write 2 bytes to 82527 (0x82)
 Data: reset IntPnd in Control 0 register (0xFD)
 set CPUUpd and NewDat in Control 1 register (0xFA)
2. Start Address: Message Object 1 Mess.Conf Register (0x16)
 Serial Control Byte: write (1 + data count) bytes (in this example : 0x85)
 Data: Mess.Conf (in this example: 0x48)
 Data 0
 ...
 Data 3
3. Start Address: Message Object 1 Control 1 Register (0x11)
 Serial Control Byte: write 1 byte to 82527 (0x81)
 Data: reset CPUUpd (0xF7)

These three steps can be combined to a single transfer packet. The transmit buffer `can_tx` will contain:

```
can_tx[0] = 14;           /* a total of 14 bytes should be transmitted */
can_tx[1] = 0x10;       /* start address */
can_tx[2] = 0x82;       /* serial control byte */
can_tx[3] = 0xFD;       /* -> Control 0 register */
can_tx[4] = 0xFA;       /* -> Control 1 register */
can_tx[5] = 0x16;       /* start address */
can_tx[6] = 0x85;       /* serial control byte */
can_tx[7] = 0x48;       /* -> DLC, transmit, std. format */
can_tx[8] = ... ;       /* -> Data 0 */
can_tx[9] = ... ;       /* -> Data 1 */
can_tx[10] = ... ;      /* -> Data 2 */
can_tx[11] = ... ;     /* -> Data 3 */
can_tx[12] = 0x11;      /* start address */
can_tx[13] = 0x81;      /* serial control byte */
can_tx[14] = 0xF7;      /* -> Control 1 register */
```

After setting up this transfer packet, `can_tx[1]` is written to the SPI interface and

interrupt `nINT2` is enabled. Macro `CAN_START_TRANSFER`, declared in `can_lib.h`, should be used for this purpose. Now the DSP may continue processing other tasks. The interrupt service routine will take care of transfers to/from the CAN controller. If the entire buffer is transmitted, the interrupt service routine disables `nINT2` interrupts itself.

Most of these tasks are encapsulated in the higher level functions defined in `can_lib.c`.

Before the DSP may transmit another packet or read the results of a read transfer, it has to check if transfer is completed. This is done by reading the `nINT2` Interrupt Enable Bit: Remember, this bit has been set by macro `CAN_START_TRANSFER` and is cleared by the interrupt service function as soon as transfer is complete. You may use macro `CAN_WAIT_READY` for busy-polling the `nINT2` Enable Bit.

Using the higher level functions defined in `can_lib.c`, updating the data of a message object and sending it reduces to:

```
can_define_data (1, CAN_DEF_LEN(4) | CAN_DEF_TX, &data, SEND);
... /* do something useful */

CAN_WAIT_READY;          /* make sure previous transfer is complete */
can_define_data (...);   /* next transfer to CAN Controller */
```

To read the results of a CAN Controller read operation (e.g. read a message), send the desired command by using function `can_get_message()`. If transfer is complete, the message can be retrieved from the can communications receive buffer `can_rx`, starting at `can_rx[3]`:

```
can_get_message (15);          /* read message object 15 */

CAN_WAIT_READY;
msg_id = can_rx[3] << 24;
msg_id |= (can_rx[4] << 16);
msg_id |= (can_rx[5] << 8);
msg_id |= can_rx[6];          /* Identifier */
msg_conf = can_rx[7];        /* message configuration */
msg_data[0] = can_rx[8];     /* data */
```

```
msg_data[1] = can_rx[9];
```

```
...
```

Depending on your application and protocol, reading the message identifier and/or message configuration may not be required.

5.2 Verifying SPI Communications

Security relevant applications may regularly verify data transmissions to/from the CAN controller by verifying the contents of the receive buffer `can_rx[1]` and `can_rx[2]`. These bytes should read `0xAA` and `0x55` after each transfer. The 82527 indicates a loss of SPI protocol synchronisation if one or both of these bytes contain values other than `0xAA` and `0x55`. A loss of synchronisation may occur if a new transmission packet is send before the previous packet is processed completely, or a heavily loaded system fails to process the SPI interrupt `nINT2` properly. If synchronisation is lost, the last transfer has failed and must be repeated after a call to function `can_reinit()`. `Can_reinit()` will write a sequence of sixteen `0xFF` bytes to the 82527 Serial Reset Address which re-synchronises the SPI communication. A typical program with SPI verification is:

```
can_define_data (...);  
...  
CAN_WAIT_READY;  
if ( (can_rx[1] != 0xAA) || (can_rx[2] != 0x55) )  
{  
    can_reinit ();  
}
```

Examples how to use this feature can be found in the CAN Terminal program `can_term.c`.

5.3 Including the CAN Library in your Projects

To use the CAN Communications Library, copy file `can_lib.c` and `can_lib.h` to your project directory and include `can_lib.c` to your project. Each program module referring to CAN library functions or macros must include file `can_lib.h`:

```
#include "can_lib.h"
```

5.4 CAN Controller Interrupts

To notify the main program about a reception or transmission of a CAN message, or about the occurrence of bus errors, the CAN controller may generate an interrupt request to the DSP. For each message object, interrupts can be enabled individually. Three approaches exist to process interrupts generated by the 82527:

1. The 82527 interrupt output is connected to a DSP interrupt input. In this scenario, the program flow of the DSP will be interrupted by the CAN controller. Be careful if the associated interrupt service function should send or receive data to/from the CAN controller ! Some DSPs do not allow nesting of interrupts. If you try to read data from the CAN controller during an interrupt service routine, make sure that nINT2 (SPI) interrupts will be processed properly. We recommend that the interrupt service function should do nothing but setting a global flag which notifies the main program about the CAN Controller's interrupt request.
2. The 82527 interrupt output is connected to a Bit-Input Port of the DSP system. This port is regularly sampled by the main program. If low, an interrupt is pending.
3. The Can Controller's interrupt output is left unconnected. Instead, the main program regularly reads the 82527 Interrupt Register. This register indicates the source of the interrupt request, a 0-value indicates no interrupt is pending.

In the two sample programs provided with the software, configuration 1.) is used for the CAN Terminal (`can_term.c`) and configuration 3.) is used in program `remote.c`.

Please note that only the highest prioritised interrupt is reflected in the CAN controller's Interrupt Register. If, for example, message 1 has generated the interrupt and message 15 receives data too before the interrupt is processed, the interrupt register will read 0x02 !

5.5 CAN Library Functions and Macros

5.5.1 Initialisation

void init_can (can_config_struct can_config)

This function resets and initialises the 82527 CAN controller. Initialisation parameters are passed in can_config. This structure is defined in can_lib.h:

```
typedef struct
{
    char bit_timing0; /* bit timing register 0 @ 0x3F */
    char bit_timing1; /* bit timing register 1 @ 0x4F */
    char p1_conf; /* Port 1 configuration register @ 0x9F */
    char p2_conf; /* Port 2 configuration register @ 0xAF */
    char low_speed; /* TRUE: 82527 ISO low speed interface enabled */
} can_config_struct;
```

For a complete description of bit timing calculation and port configuration please refer to the Intel 82527 Serial Communications Controller Architectural Overview.

The hardware reset of the 82527 is performed via the D.Module nRESOUT output. Init_can () does not use interrupt driven transfers, so you may call it in an early stage of system initialisation when interrupts are still disabled.

Init_can () declares all message objects as invalid and sets all global masks to '111..11'. Error interrupts are enabled.

void can_reinit (void)

Re-initialises SPI communications by writing to the 82527 Serial Reset Address. Use this function only to recover from SPI errors. SPI errors are indicated by can_rx[1] != 0xAA or can_rx[2] != 0x55 after a transfer is completed.

5.5.2 Acceptance Filtering and Message Object Set-Up

```
void can_define_masks ( unsigned int g_mask_std,  
                        unsigned int g_mask_ext,  
                        unsigned int msg15_mask, char msg15_format)
```

This function defines the global masks for acceptance filtering. Masks are assumed right justified, i.e. valid entries for a standard mask are 0x000..0x7FF, extended masks range from 0x00000000 .. 0x1FFFFFFF. Message 15 has only one global mask, either standard or extended format. This format is specified in the last parameter. Use 's' for standard or 'e' for an extended mask.

These masks define which messages are received by the CAN controller. A '1' Bit means a 'must match', i.e. the message object's arbitration field bit and the received identifiers bit for that position must match, otherwise the message is rejected. A '0' value means 'don't care'. The resulting mask for message 15 is the logical AND of the global mask and the message 15 mask. You may for example set the global standard mask to 0x7FF. Only messages with exactly matching identifiers will be received by message object 0 to 14. If you set the message 15 mask to 0x700, any other message with matching upper 3 identifier bits will be stored in message object 15.

```
void can_define_msg (char msg, unsigned int ident, char config, int int_en)
```

Define a message object. Parameter msg is the desired message object (0 .. 15). Parameter ident is written to the arbitration field of the message object. The identifier is assumed right justified, i.e. 0x000..0x7FF for standard format and 0x00000000..0x1FFFFFFF for extended format. Parameter config is written to the message object's MessConf register. You do not need to specify the data length code at this time, but direction (receive or transmit) and message format (standard or extended) are required. You may use the CAN_DEF_xxx constants defined in can_lib.h. The last parameter, int_en, should be set to TRUE if you want the CAN controller to generate an interrupt upon reception or transmission of this message object. Depending on the direction specified in parameter config, receive or transmit interrupt is selected.

This function also declares the object as valid.

5.5.3 Data Transfer Functions

void can_define_data (char msg, char config, char *data, int send)

This function is used to write data to a message object's data field. It also configures the data length code, message format and direction. Parameter msg is the desired message object (0..14). Config is written to the message object's MessConf register. Use the CAN_DEF_xxx constants defined in can_lib.h to specify data length code, direction and format (of course, this function is used for transmit objects only). Parameter *data is a pointer to the data to be copied to the message object's data field.

If parameter send is TRUE, the TxRqst bit of the message is set and the message will be send. If the message is intended to answer remote requests only, or if it should be transmitted later, set send to FALSE. This function also clears the IntPnd bit and sets the NewDat bit of the message object.

void can_get_message (char msg)

This function is called to read an entire message object, including arbitration field, message configuration and data field. Parameter msg specifies the desired message object. After transfer is completed (check with CAN_WAIT_READY for example), the receive buffer can_rx contains the message object as it is stored in the CAN controller, starting at can_rx[3]:

can_rx[3] - Arbitration 0
can_rx[4] - Arbitration 1
can_rx[5] - Arbitration 2
can_rx[6] - Arbitration 3
can_rx[7] - Mess.Conf
can_rx[8] .. can_rx[15] - Data 0 .. Data 7

This function clears the IntPnd Bit and resets the NewDat bit of the message object.

5.5.4 Message Status and Control

void can_set_msg_control (char msg, int control)

Write to a message object's control registers. This function allows to set or reset individual message control bits independent of writing or reading data to or from the message object. The upper eight bits of parameter control (bit 15..8) are written to Control 0, the lower eight bits (7..0) are written to Control 1. You may use the definitions `CAN_DEF_xxx_Set` or `CAN_DEF_xxx_Reset` defined in `can_lib.h`. Please note that combining these parameters requires an AND operation, not OR as usual. For example, to reset the `IntPnd` and `RmtPnd` bits of message object 1, call:

```
can_set_msg_control (1, CAN_DEF_IntPnd_Reset & CAN_DEF_RmtPnd_Reset);
```

void can_get_msg_control (char msg)

This function is used to read the status of a message object. After transfer is complete (check for example using macro `CAN_WAIT_READY`), the message object's Control 0 register can be retrieved from `can_rx[3]` and Control 1 from `can_rx[4]`.

5.5.5 Low Level Functions

void can_read_reg (char reg)

Read a register. Valid registers are 0 to 0xFE. Note that reading the serial reset register 0xFF will cause the SPI communication to fail. After transfer is completed (check with macro `CAN_WAIT_READY`), the register's data can be read from `can_rx[3]`.

void can_write_reg (char reg, char data)

Write data to a register.

5.5.6 Macros

Several macros are defined in `can_lib.h` to start transfer, check if transfer is complete and read or write special registers:

`CAN_START_TRANSFER` writes `can_tx[1]` to the SPI and enables `nINT2` interrupts. This macro is required if you want to write your own CAN transfer functions.

`CAN_WAIT_READY` polls the state of the `nINT2` Interrupt Enable Bit. It is used to wait for a transfer to finish and must be called before data is retrieved from the receive buffer `can_rx`, or before any new transfer is initiated.

`CAN_TXRQST(msg)` sets the Transmit Request bit of the specified message object. If direction is transmit, the CAN controller will send the message, if direction is receive, a remote request frame will be send.

Some other macros are defined for reading or writing commonly accessed registers, based on functions `can_read_reg ()` and `can_write_reg ()`:

<code>CAN_READ_CONTROL</code>	read the control register 0x00
<code>CAN_READ_STATUS</code>	read the status register 0x01
<code>CAN_READ_INT</code>	read the interrupt register 0x5F
<code>CAN_READ_PORT1</code>	read port 1 input register 0xBF
<code>CAN_READ_PORT2</code>	read port 2 input register 0xCF
<code>CAN_WRITE_CONTROL(data)</code>	write data to control register 0x00
<code>CAN_WRITE_STATUS(data)</code>	write data to status register 0x01
<code>CAN_WRITE_PORT1(data)</code>	write data to port 1 output register 0xDF
<code>CAN_WRITE_PORT2(data)</code>	write data to port 2 output register 0xEF

6 Support Software

6.1 CAN Terminal

The CAN Terminal program `can_term.c` is an interactive tool for configuring the CAN controller and sending and receiving messages via the CAN bus. It is also useful for debugging and diagnostic purposes. Please note that CAN Terminal however cannot replace a protocol analyser or similar tools !

CAN Terminal communicates via RS232 connection and a terminal program. The terminal must support ANSI Escape sequences. Suitable terminal emulation modes are ANSI, VT52, VT100 etc. Line parameters are: 9600 Baud, 8 databits, 1 stopbit, no parity and Xon/Xoff flow control.

CAN Terminal also assumes the CAN controller's interrupt output connected to the D.Module's `nINT0` input. If your system requires a different configuration, please refer to 6.1.5 Customising CAN Terminal.

6.1.1 Usage

CAN Terminal is controlled via an RS232 terminal or terminal program. All numeric inputs are expected as hexa-decimal. Input is finished by pressing either the `<ENTER>` or `<TAB>` key. To cancel an input, press `<ESC>`. ANSI and VT52 terminal modes allow to use the `<BACKSPACE>` key to delete entries. To navigate between the screen elements use the `<TAB>` key. to activate a button, press `<ENTER>`. CAN Terminal uses three user dialogues for configuration, acceptance filtering and communications.

6.1.2 Configuration Dialogue

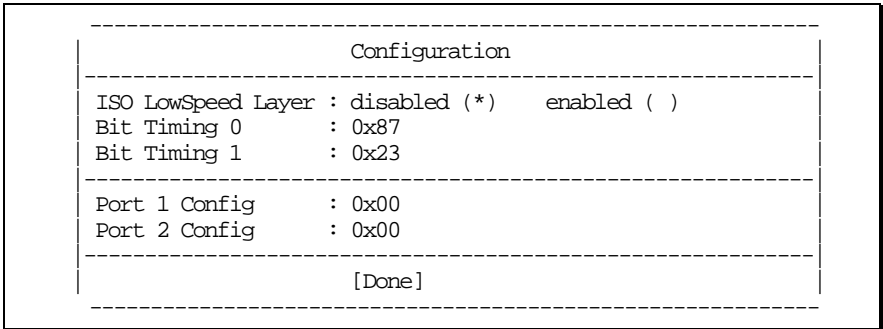


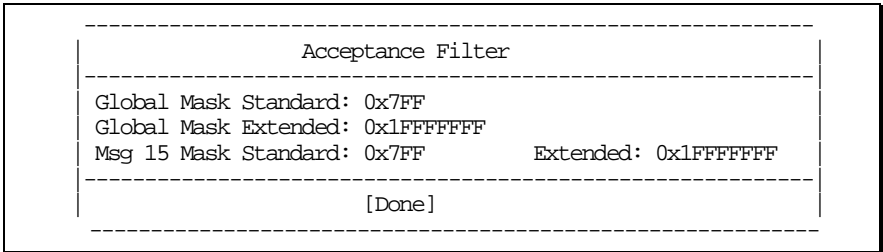
Figure 6-1 CAN Terminal Configuration Dialogue

The Configuration screen is used for the basic setup of the CAN controller. Default settings are:

- use external bus transceiver (ISO LowSpeed Interface disabled)
- 125 kBit (Bit Timing Registers)
- Ports configured as inputs

For a complete reference how to setup bus configuration and ports, please refer to the Intel 82527 Serial Communications Controller Architectural Overview. If all items are configured according to your requirements, select the [Done] button and press <ENTER>. This will invoke the CAN Library function `init_can()` and display the next dialogue.

6.1.3 Acceptance Filtering Dialogue



The image shows a rectangular dialog box with a dashed border. Inside the box, the text is as follows:

```
-----  
                          Acceptance Filter  
-----  
Global Mask Standard: 0x7FF  
Global Mask Extended: 0x1FFFFFFF  
Msg 15 Mask Standard: 0x7FF           Extended: 0x1FFFFFFF  
-----  
                          [Done]  
-----
```

Figure 6-2 Acceptance Filtering Dialogue

This dialogue defines the global masks for acceptance filtering. Valid entries are 0x000..0x7FF for standard format masks and 0x00000000..0x1FFFFFFF for extended format masks. If all entries are configured, select the [Done] button and press <ENTER>. This will call CAN Library function `can_define_masks()`.

6.1.4 Communications Dialogue

```

-----
                        CAN Communications
-----
Status: 0x18
-----
Message 15      Format : Std(*) Xtd( )
Arb: 0x100      Len: 3   Data: 0x 12 12 34
-----
Message 1       Format : Std(*) Xtd( )
                Direct.: Rcv(*) Xmt( )
Arb: 0x200      Len: 1   Data: 0x 47
[Send Msg1]
-----
Port1 in: 0x55 out: 0x55  Port2 in: 0xAA out: 0xAA
-----
                [Configuration]      [Acceptance Filter]
-----

```

Figure 6-3 Communications Dialogue

This is the main screen for receiving and transmitting messages and access to the 82527 I/O ports. Message 15 is used to display incoming messages, Message 1 can be configured for either receive or transmit. To send a remote request frame, configure message 1 as 'receive', select the [Send Msg1] button and press <ENTER>. The status register display is updated each time a message is received or in case of bus errors or warnings. The Ports are updated each time the corresponding item is selected. To move back to the configuration dialogue select the [Configuration] button on bottom of the screen and press enter, to change Acceptance Filtering masks select [Acceptance Filtering].

Changing a message's direction, format, arbitration field or data length code will call CAN Library function `can_define_msg()`. Modifying data will call `can_define_data()`, but without sending the message immediately. Transmission is requested only by activating the [Send Msg1] button, which relates to macro `CAN_TXRQST`.

Please note that display update may be much slower than the actual CAN bus traffic. This may result in losing incoming messages. Select appropriate acceptance filtering to avoid buffer overrun.

6.1.5 Customising CAN Terminal

If you want to use CAN Terminal as a diagnostic program in your application, customisation may be required. Following are some hints for common customisations:

1. To change the default bus configuration, change the corresponding initialisers in `struct config_menu_struct cfg_menu = {...};`
2. To change the default acceptance filtering masks, make the required changes in `struct filter_menu_struct filter_menu = {...};`
3. CAN Terminal expects the 82527 interrupt output connected to the D.Module's nINT0 input. To use a different interrupt input, change the installation (and de-installation) of the interrupt handler in function `communications()`. This interrupt handler only sets a global flag which is polled by function `communications()` regularly. If you connect the interrupt to an I/O port for polling, or if you want to use polling of the CAN controller's interrupt register instead, make the required changes at the end of function `communications(): /* read CAN Controller interrupt state */`

To re-build the modified `can_term` program, use the supplied batch file `built.bat`.

6.2 Remote

`Remote.c` is a simple CAN application, showing how to initialise the CAN controller, how to send and receive messages, how to actively respond to remote frames and how to use the CAN controller's interrupts in polling mode. `Remote.c` uses message object 1 as an active transmit object. It is configured to standard identifier 0x100 and will transmit a 3 byte message, the 'time' once per second. Message 15 is configured to receive standard frames with ID 0x101. The data received will pre-set the time transmitted by message 1. Message object 2 is intended to answer remote request frames (standard identifier 0x200) only. Each time a remote request was answered, message 2 generates an interrupt and the program increments the data (1 byte) of message 2.

To re-build `remote.c`, use the supplied batch file `build.bat`.

Pin	A	B	C	T	U	V
1	nc	GND		nc	nc	nc
2	nc	PRG_IO0		nc	nc	nc
3	nc	PRG_IO1		nc	nINT0	nc
4	nc	PRG_IO2		nc	nINT1	nc
5	nc	PRG_IO3		nc	nc	nc
6	nc	PRG_IO4		nc	nc	nc
7	nc	nc		nc	nRESOUT	nc
8	nc	nc		nc	nc	nc
9	nc	nc		nc	nc	nc
10	nc	nc		nc	nc	nc
11	nc	nc		nc	nc	nc
12	nc	nc		nc	nc	nc
13	nc	nc		nc	nc	nc
14	nc	nc		nc	nc	nc
15	nc	nc	nc	nc	nc	nc
16	nc	nc	GND	nc	nc	nc
17	nc	PRG_IO15	VCC	nc	nc	nc
18	nc	PRG_IO16		nc	nc	nc
19	nINT2	nc		nc	nc	nc
20	nINT3	nc		nc	nc	nc
21	nc	nc		nc	nc	nc
22	nc	nc		nc	nc	nc
23	nc	nc		nc	nc	nc
24	nc	nc		nc	nc	nc
25	nc	nc		nc	nc	nc
26	nc	nc		nc	nc	nc
27	nc	nc		nc	nc	nc
28	nc	nc		nc	nc	nc
29	nc	nc		nc	nc	nc
30	nc	nc		nc	nc	nc
31	nc	nc		nc	nc	nc
32	GND	nc	nc	nc	nc	nc

Table 7-1 Pinout

PRG_IO0: Master Clock from AN 82527 (8 MHz)

PRG_IO1: SCK serial clock

PRG_IO2: MISO master in, slave out

PRG_IO3: MOSI master out, slave in

PRG_IO4: AN82527 Chip Select, active low

PRG_IO15: SPI Interrupt Request on D.Module.C6x01, Rev. 1, active low

PRG_IO16: SPI Interrupt Request on D.Module.C31eco, active low

8 Electrical Specifications

Recommended Operating Conditions

Supply Voltage VCC	5V +/- 10%
Operating Temperature	-40 .. +85°C
High Level Input Voltage	min. 2V, max. 5.5V
Low Level Input Voltage	min. -0.5V, max. 0.8V

CAN

supports CAN Specification 2.0 part A and B

Bus interface according to ISO/DIS 11898

Bit-I/O

16, individually configurable as input or output, open-drain,

max. sink current 1.6mA, max. source current 260uA,

10K pull-up resistors on-board