AT90CAN128 drivers*

Aseem Rastogi
ari12003@student.mdh.se

Nils Brynedal Ignell
nbl08001@student.mdh.se

ABSTRACT

The report details the various drivers for the AT90CAN128. These drivers are used in various boards depending upon the board requirements. Code for some of these drivers was available from the Vasa Project. However, various changes have been done to all parts of the code from the Vasa Project. All these drivers have been tested and used during the project and have been shown to work well.

1. INTRODUCTION

The document gives the details of the various drivers for AT90CAN, used in the Naiad AUV. The drivers are used for the various boards.
Details about drivers for the following communication standards: Serial Peripheral Interface (SPI), Universal Asynchronous Receiver/Transmitter (UART) and Controller Area Network (CAN).
The code for these drivers are available at the Naiad AUV GitHub page [**?**].

2. IMPLEMENTATION

The following sections describe the drivers of various modules of AT90CAN128 used in the Naiad AUV project.

2.1 CAN Drivers

The AT90CAN128 consists of 1 CAN module. The CAN module has 15 Message Objects (MObs) indexed from 0 to 14. Each MOb can be configured for either receiving (Rx mode) or transmitting (Tx mode). For the drivers in the Naiad AUV, MObs 0 to 12 are used for receiving and 13 is used for transmitting. Out of the Rx MObs only 0 to 9 can be configured by the user. For the Naiad AUV, each board containing the AT90CAN128 can be in different modes such as Bootloader mode, Simulation mode or Normal mode. The CAN module for different boards should operate differently in different modes, for example in Simulation mode the INS Board should not send any CAN messages. The Board_And_Mode_Defs define how each board should react in each mode.
The drivers primarily consist of following functions:

1. **Initialization (Can_Init):** This is used to initialize the CAN module. It takes the baud rate and board name as input. The mode is initialized to Normal mode. The initialization sets the baud rate, clears all MObs and enables CAN. It also sets the MObs 10, 11, 12 to receive Bootloader start message, Mode message and Status request message respectively. These are the messages that all the boards should receive.

2. **Set Filter & Mask (Can_Set_MOb_ID_MASK) :** This is used to configure the RX MObs from 0 to 9. This takes as inputs the MOb number and the filter and mask. For AT90CAN128 the MOb filter and mask is cleared once a message is received in that register and the user has to re-configure it. To avoid that hassle the filter and mask for each MOb is stored in an array and the MOb is automatically reconfigured with the filter and mask once a message is received in it.

3. **Software Buffers :** The CAN drivers have RX and TX buffers of 16 messages each. The buffer is a ring buffer, and has pointers pointing to the read location and write location. The buffer pointers can have values from 0 to 31 even though the buffers are from 0 to 15. This is done to distinguish the full buffer from an empty buffer. When the buffer is empty both read and write will have same pointer location while in case of full they will differ by 16. In order to access the buffer, the buffer pointer mod 16 is used.

4. **CAN Interrupt :** The CAN interrupt occurs whenever a message is transmitted or received in the MObs. The Interrupt iterates over all MObs and checks if a message is received or transmitted.
   For the receive interrupt it checks if the message is a Mode message and if so, the mode is changed to the mode defined in the message, if the message is a Reboot message, the Reboot function is called and if the message is to start the bootloader the boolean variable is set to True. Then the message is put in the message software buffer if the mode allows the CAN module to receive the messages. Otherwise the message is discarded. Then the RX Mobs filter and mask are reset

*This report was written during the fall of 2013 in an advanced level project course at Mälardalen University, Sweden.

by reading the values from the arrays.
For the transmit interrupt, when the message has been transmitted the message is removed from the software buffer and the next message is put in the RX register for transmitting.

5. **Reading CAN Messages (Can_Get) :** This is used to get the messages received. The function takes as input the time it should wait for the message and returns a message and a boolean variable to tell if the message is received. A value of -1 can be passed in "waiting time" to make the funtion wait infinitely till a message arrives. The function iterates through all the messages in the software buffers and returns the highest priority message. Also the returned message is removed from the software buffer. This function also calls the switch_to_bootloader message after the mode is set to bootloader and the mode message has been read by the user.
6. **Transmitting CAN Messages (Can_Send) :** This function takes as input the message to send over the CAN bus. If the current mode for that board allows for a message to be sent the message is put into the TX software buffer, otherwise the message is discarded.

2.2 SPI Drivers

The AT90CAN128 has an 8 bit SPI module. The SPI drivers primarily has following functions:

1. **Initialization (Init) :** The Init function takes as input the Clock divisor to use for the SPI clock, the clock mode, a boolean variable to tell if the device is master or slave, a boolean to tell if the most significant bit is transmitted first and a boolean to tell if the Slave Select pin is used or not. The function disables the SPI device, configures it with the values passed and enables it.
2. **SPI transmit and receive :** There are three functions to transmit and receive SPI data depending on whether only transmit, only receive or both transmit and receive is required. The SPI_IO function takes as input a byte which it transmits on the SPI and waits for the reply. It calls the master or slave IO functions depending on whether the AT90CAN128 is master or slave. The reply is then returned.
   The WriteSPI is used to transmit 1 byte of data. This function calls SPI_IO internally. Whatever is received during that SPI transmit is discarded.
   The ReadSPI is used to read a byte of SPI data. This function also calls SPI_IO internally. A temp value of 0 is transmitted and the received value is returned.

2.3 UART drivers

The UART drivers provide the ability to send and receive data over the two UART buses that the AT90CAN128 has. For each of the two buses there is one send buffer that holds data that is to be sent and one receive buffer that holds data that is yet to be read. The buffers are implemented as circular buffers.
The UART drivers are implemented in the AVR.AT90CAN128.USART package that provides the following functions and procedures:

- **Init** initiates the UART by setting the registers controlling the baud rate and UART interrupts.
- **Put** writes one byte to the send buffer and checks whether UART transmission is already ongoing, if not, transmission of the first byte is started.
- **Write** uses **Put** to write a a string of data to the buffer.
- **Space_Available** returns the amount of space left.
- **Data_Available** returns the number of bytes present in the receive buffer.
- **Get_Char** reads a byte from the receive buffer (if the buffer is not empty).
- **Read** reads a string of bytes from the receive buffer.
- **Flush_Receive_Buffer** empties a receive buffer.

Each function takes a variable of the type *BAUDTYPE* as argument. *BAUDTYPE* is used to represent which UART bus that a function call refers to.
Four UART interrupts are used: *USART0 Data Register Empty*, *USART1 Data Register Empty*, *USART0 Receive Complete* and *USART1 Receive Complete*.

2.3.1 USART Data Register Empty interrupts

These interrupts occur when transmission of a byte has been completed. When this happens, the Interrupt Service Routine (ISR) checks whether the send buffer is non-empty (if there is more data to send). If so, a byte is read from the send buffer and transmission of this byte is started.

2.3.2 Receive Complete interrupts

These interrupts occur when a byte has been received. The ISR will fetch the byte by reading the *USART0 Data Register* (or *USART1 Data Register* depending on which bus caused the interrupt). If the receive buffer is not full the ISR will put this byte into the receive buffer.
If the receive buffer is full the byte can not be stored in the receive buffer and will be lost. Because of this it is important that the application software reads the receive buffer often enough, particularly in applications where large quantities of data is received over UART.

3. RESULT

The drivers were tested individually and also as a part of testing the individual board software. All drivers are working well. The subsections below describe the results of each driver more thoroughly.

3.1 CAN drivers

The CAN drivers were tested thoroughly. The following functionalities were tested:

1. Basic send and receive functionality was tested. A message was sent over CAN bus and a message was received.

2. Received multiple messages in the buffer and checked if the Can_get function returned the highest priority message first.

3. Overflow the send and receive buffers and observed that the messages were discarded after the buffers were full.

4. Sent the Mode message to change the mode of the board and saw how the send and receive functionality changed depending on the mode.

5. Sent the reboot message and the board rebooted.

Apart from these tests the CAN drivers were also tested exhaustively throughout the project by other modules which were using the CAN. The CAN drivers seem to be pretty stable.

3.2 SPI drivers

The SPI drivers were tested by sending and receiving messages to the ADS1255, Analog to Digital Converter (ADC) chip. Messages were sent to write to the registers of the ADC and the value was read back. Also the SPI drivers were tested to receive the Digital value from ADC which is of 24 bits. Both the tests worked as expected.
The SPI drivers were not tested when more than one slave device is connected to the master or when the AT90CAN acts as a slave.

3.3 UART drivers

The UART drivers have been tested and used extensively throughout the project and work well. There was one error that was detected quite some time after the UART code was implemented:
If a Receive Complete interrupt occurred when the receive buffer was full, the ISR will not do anything. This meant that the *USART0 Data Register* (or *USART1 Data Register*) would not be read.
When the next byte was received it would too be put in this register and since this register had not been read, an error occurred and the AT90CAN128 would reset itself.
This error was removed by fetching the received byte regardless whether the receive buffer was full or not.

4. CONCLUSION

As mentioned in Section 3, the drivers work well. Future projects should use these drivers instead of implementing their own drivers from scratch. Hopefully, the work of this project will save time for future projects.

4.1 Future work

Future work on these drivers may include further testing. As mentioned in Section 3.3, the UART drivers worked well under normal conditions (when the receive buffer did not overflow). But under abnormal conditions (when the receive buffer overflowed) the actual behavior of the UART drivers was not what was expected. The whole system cashed instead of there "just" being a few bytes lost in the UART communication. Also as mentioned in Section 3.2 the SPI drivers need to be tested more thoroughly.
What can be learned from this is that tests of a subsystem such as drivers of this kind should not be "nice". Tests should not only involve normal working condition, but should also try to "break" the subsystem and see how it breaks. Specifications on a subsystem should not only include the system's behavior under normal working conditions but also how the subsystem should behave when put under conditions it cannot handle (such as when the receive buffer is not read by the application software causing the it to overflow).

It might also be worth to try to prove the correctness of the drivers using methods such as UPPAAL.

5. REFERENCES