BeagleBone Black to CAN interface*

Nils Brynedal Ignell
nbl08001@student.mdh.se

ABSTRACT

Several BeagleBone Black boards were used in the system for various tasks (the Mission control or the Sensor fusion for example). Each board needs to communicate over the CAN bus, something that is not supported directly by the hardware. This problem was solved by using the Generic CAN controller which is a piece of hardware that is used in many roles throughout the project. The BeagleBone Black communicates with the Generic CAN controller over UART. The Generic CAN controller receives the CAN messages from the BeagleBone Black via the UART bus and sends them on the CAN bus. When a CAN message is received from the CAN bus, the Generic CAN controller will send it over UART to the BeagleBone Black.
This approach enables the BeagleBone Black to send and receive CAN messages, something vital for the success of the Naiad AUV project because of the central role of the BeagleBone Black boards and the CAN bus in the design of the system.

1. INTRODUCTION

Most higher level computations run on BeagleBone Black boards (abbreviated "BBB" in this report), therefore these boards must be connected to the CAN bus of the Naiad Autonomous Underwater Vehicle (AUV). Research in the beginning of the project found that there was a *Cape* (an extension board to the BBB) available, that could connect a BBB to the CAN bus. However, further investigation revealed that this cape was facing issues and may not work. For this reason, it was decided that each BeagleBone Black would be connected to the CAN bus using a board custom built during the project, the Generic CAN controller.

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

The Generic CAN controller is a small (approx. 76 by 40 millimetres) electronic board that has an AT90CAN128 microcontroller [1], an MCP2551 CAN transceiver [2] and all the peripheral circuitry needed for these, as well as its own power supply. The Generic CAN controller is used throughout the project for several tasks. It can be connected to the CAN bus and has two UART buses as well as an SPI bus.
Hence there were two possible ways to connect the BBB to the Generic CAN controller: via UART or SPI.
UART was chosen for two reasons: In the Vasa project, a protocol for sending CAN messages over UART was already implemented for an AT90CAN128 microcontroller (the same that is used on the the Generic CAN controller) which was used as a starting point. Furthermore, UART allows the communication in one direction to be truly independent of that in the other direction. Using SPI would most definitely be possible but would require the communication protocol between the BBB and the Generic CAN controller to be more complicated and would have to be implemented from scratch.
The protocol used from the Vasa project was modified to better fit the requirements of the Naiad project.

Whenever the BBB needs to send a CAN message the following steps are done: The message is converted into a string of bytes. The message is then sent over UART to the Generic CAN controller that in turn will convert this string of bytes back to a CAN message. The message is then sent on the CAN bus.
When the Generic CAN controller receives a CAN message the same process will be done but in the opposite direction.

2. IMPLEMENTATION

The implementation has the following parts:

- *The UART drivers for the AT90CAN128 microcontroller*
  This part is described in the AT90CAN128 drivers report.
- *The CAN drivers for the AT90CAN128*
  This part is described in the AT90CAN128 drivers report.
- *The UART drivers for BeagleBone Black*
- *Conversion between CAN message and strings of bytes*
  This is implemented in the Can_Utils project.
- *CAN interface code for the BeagleBone Black*
  This is implemented in the BBB_Can project.
- *The main program on the AT90CAN128*
  This is implemented in the AT90CAN_Uart_To_Can project.

The hardware filters in the AT90CAN128 microcontroller are set to receive all CAN messages sent on the CAN bus, i.e. no filtering will be used. The reasoning for this is the following; If not all messages would be forwarded to the BBB, then the information about which messages that are to be forwarded would need to sent from the BBB to the Generic CAN controller. This could most definitely be done but it would make the communication protocol much more complicated.

2.1 Protocol

The highest baud rate that was possible to achieve between the BBB and the AT90CAN128 microcontroller was 115200 baud. Both BBB and the AT90CAN128 microcontroller can send at higher speeds but 115200 baud was the highest baud rate at which both of them could run. This relatively low baud rate (compared to the 250 kBaud used for the CAN bus, though the CAN protocol has a significant amount of overhead) means that the BBB to CAN link is the "bottle neck" of the whole CAN bus system. For this reason the protocol for sending CAN messages over UART was made as simple as possible to maximize performance, i.e. number of CAN messages that can be sent per second.

There are mainly four pieces of information in each CAN message:

- Whether the message uses extended (29 bits) or normal (11 bits) message IDs
- The length (number of bytes) of the payload
- The message ID
- The payload of the message

Each message is encoded in the form of a header of five bytes followed by the payload of zero to eight bytes. The first byte of the header contains information regarding whether or not the message uses an extended message ID and the length of the message. Its four least significant bits contain the length of the message. The fifth least significant bit is set to 1 if the message is extended and 0 if it is not.
The following four bytes of the header represent the message ID.
After the header, the payload bytes (if any) follow.
No "start of message" or "end of message" flags are used, to minimize the overhead even further.

Please note: This protocol is dependent on the byte sequence between the two devices being precisely correct. Even one extra, one less, or (in some cases) a corrupted byte will cause an offset in the receiver buffer, and the two devices will become unsynchronized and the whole communication will be compromised. The same will happen if one or several bytes are lost e.g. due to a receive buffer overflowing.

Since conversions between CAN messages and bytes would be done on many different devices (the BeagleBone Black, the Generic CAN controller and also in the Simulator), this functionality was put into the *Can_Utils* project, in order to be shared.
The Can_Utils project has three functions: *Message_To_Bytes*, *Bytes_To_Message_Header* and *Bytes_To_Message_Data*.
When converting bytes to a CAN message, the length of the message is unknown and therefore the conversion has to be done in two steps. In the first step, the bytes of the header (which is of constant length) are read and converted to the message header, from which the length of the payload data can be retrieved. In the second step, as many bytes as the payload size are read and converted to message data.
This is the reason why two separate conversion procedures are provided: *Bytes_To_Message_Header* and *Bytes_To_Message_Data*.
When converting a CAN message to bytes, the length of the message is known, and consequently the whole conversion can be done in one procedure, *Message_To_Bytes*.

2.2 Interface code for the BeagleBone Black

The BBB_Can project uses the UartWrapper project to send and receive data on the UART. BBB_Can provides three procedures: *Init*, *Send* and *Get*.
Init simply initiates the UartWrapper.
The *Send* procedure takes a CAN message as an argument, calls the *Message_To_Bytes* procedure in *CAN_Utils* to get a string of bytes representing the CAN message and writes these bytes to the UART send buffer using UartWrapper.
The *Get* function has three out parameters: *msg* (the CAN message received), *bMsgReceived* (a boolean value set true if a CAN message was received) and *bUARTChecksumOK* (which is obsolete and should not be read).

The *Get* procedure reads all the data in the UART receive buffer and puts it in a separate software buffer. The reason for this separate software buffer is that the UartWrapper does not provide a function for getting the number of bytes in the receive buffer without reading all available bytes.
If there is enough data in the software buffer, the header of the message is read and converted to a CAN message. Given this, the number of bytes in the message payload is known and the remaining bytes are read from the software buffer. If

not all payload bytes have been put into the software buffer, more data is read from the UART receive buffer until enough bytes can be read. The procedure will then return with the received CAN message and *bMsgReceived* set true.
If not enough data for a full header has been received, the procedure will return with *bMsgReceived* set false, *msg* will then be undefined and shall not be read.

2.3 Main program running on the Generic CAN controller

The main program running on the Generic CAN controller is called AT90CAN_Uart_To_Can and has two main procedures:

- *Handle_Uart* reads CAN messages from the UART receive buffer and writes them to the CAN send buffer.
- *Handle_Can* reads CAN messages from the CAN receive buffer and writes them on to the UART send buffer.

Since this program is single threaded it will alternate between these procedures.
As mentioned, the protocol for the communication over UART is sensitive to an overflow in the receive buffer. For this reason, the Handle_Uart procedure has been prioritized and will handle all CAN messages in the buffer and only return when no more messages have been received. Handle_Can will only handle one message, if any message is present in the CAN receive buffer, and then return.
This way any overflow in BBB to CAN subsystem will only be in one of the following:

1. *The send buffer of the BBB*
   If the BBB writes to its send buffer at a higher rate than the UART connection can send.
2. *The CAN send buffer*
   If the main program tries to send messages at a higher rate than it can send, for example during to high CAN bus utilization.
3. *The CAN receive buffer*
   If CAN messages are received from the CAN bus at a higher rate than what the main program can read. For example if the BBB is sending CAN messages at a high rate the Handle_Can will not get enough time to empty the CAN receive buffer.

The first case is easily avoided by making sure the BBB does not send CAN messages at a faster rate than the BBB to CAN subsystem is capable sending. The second and third cases will only cause individual CAN messages to be lost rather than compromising the communication protocol between the BBB and the Generic CAN controller.
This way the BBB to CAN communication subsystem can be considered fail-safe.

3. RESULT

The results from the BeagleBone Black to CAN subsystem have been very good. The subsystem has been tested and used extensively throughout the course of the project and has worked well.
The potential problem with the protocol being offset (too few or to many bytes being sent) has not been found to occur at any time. Nor have there been any cases of CAN messages being lost in the BBB to CAN subsystem.
There exists no known case of data being corrupted during the transmission over UART.

4. CONCLUSION

As mentioned in Section 3, the implementation of the BeagleBone Black to CAN communication has worked well. However, there are ways the current design can be improved.

4.1 Future work

As mentioned in the previous sections, the protocol for the communication over UART is based on the sequence of bytes being correct at any time. One could implement some form of ”start of message” and ”end of message” flags to signal the start and end of messages. This would ensure that if one message is sent incorrectly, with too many or too few bytes, the receiver would be able to find the ”start of message” and ”end of message” flags and regain synchronization.
Since any byte value can be sent in the payloads of the CAN messages, there exists no pattern of bytes that can be assumed not to occur in the data that will be sent over UART. Therefore some form of byte stuffing has to be implemented in order to handle this.
Unfortunately a change such as this would make the protocol less efficient.

To improve the communication speed, the UART communication could be done via SPI instead. This would require rewriting some of the AT90CAN_Uart_To_Can code as well as implementing SPI on the BBB.

The protocol could be be optimized further by only sending the message ID in two bytes instead of four if the message ID is not extended.

Tools like UPPAAL could be used to prove correctness and/or simulate the BBB to CAN subsystem and other nodes on the CAN bus.

5. REFERENCES

[1] At90can32/64/128 datasheet. `http://www.atmel.com/Images/doc7682.pdf`. Accessed 2013-12-28.
[2] Mcp2551 datasheet. `https://www.sparkfun.com/datasheets/DevTools/Arduino/MCP2551.pdf`. Accessed 2013-12-28.