Lightweight Requirements

Request for Widget byte-pipe-mode Test Fixture

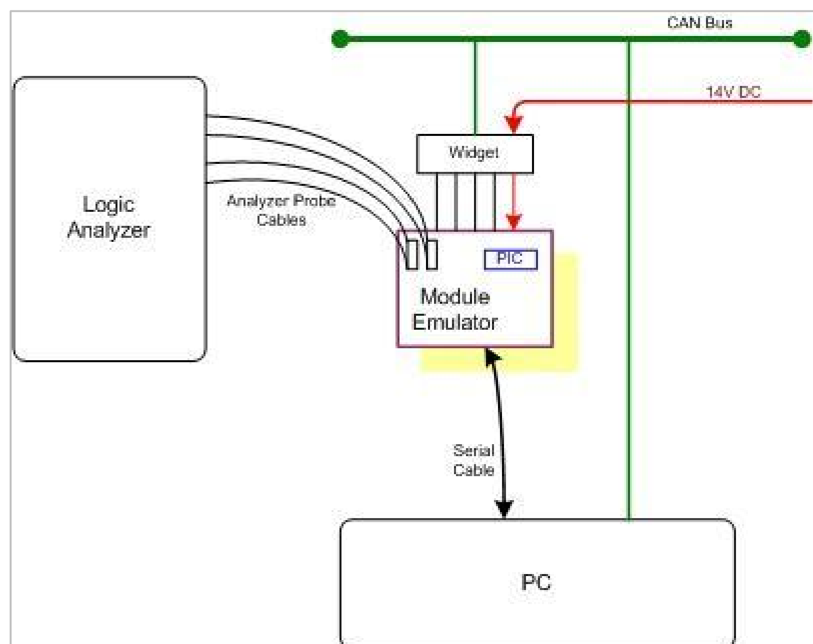
This page identifies the design goals and requirements for a module-widget-interface emulator to be used for final testing of the byte-pipe mode. The requirements are prioritized in order to simplify implementation decisions.

Design Goals

1. **Primary**: shake out the full-speed bidirectional operation.
2. **Secondary**: exercise the communication breakdowns most likely to occur.

Context of Device

The emulator is connected to the Widget just as a Module would be. The emulator gets its power from the unregulated, switched, power output of the Widget, providing the best testing fidelity.



There should be three interfaces on the module emulator (as the picture on the right shows):

1. Widget Socket
2. Analyzer Socket(s)
3. Serial I/F to plug to PC for controlling device and gathering status

The widget plugs into the emulator and connects to the CAN Bus and Unregulated DC Power.

The Logic Analyzer plugs into sockets on the emulator board. The sockets are simply taps on the Input Data bus, the Output Data bus, the two REQ lines and the two ACK lines.

The emulator is commanded from the PC via Serial and reports over Serial to the PC any data given to it by the Widget. The Serial interface should run at one of any standard baud rate between 19,200 and 115,200. The interface should use 8-data bits, no parity, and 1-stop bit.

The PC, acting as the IHU, will both command the Widget via CAN and will lurk on the CAN bus, capturing traffic sent by the Widget in order to verify that the transfer over the CAN bus worked without any data loss or corruption.

Prepared by Stephen M Moraco, kc0ftq@amsat.org

Core Requirements

For this emulator to be useful it must provide, at a minimum, the ability to run both the input and output pipes at what-ever speed the Widget can attain (measured at just under 90KB/Sec. running only one channel: in or out. NOTE: The 90KB/Sec. might be the sum of both channels or might be the highest each can go. We don't know, yet, which).

For at least one type of transfer test, the emulator must create predictable values so that the data arriving at the PC can be verified to be correct due to this predictability. (e.g., a specific byte value will only land in a specific location within the 8-byte can message (payload), least significant nibble is offset into packet, something to this effect.)

The emulator must be able to deliver traffic (CAN Packet content), which arrives from the Widget, to the PC via the serial port. By doing this the PC can verify that the traffic reported by the emulator is what it sent.

The PC (user at PC?) should be able to command the emulator to begin sending traffic. By not starting to send automatically, we have the ability to test input and output separately should we find a need.

The PC should be able to specify what type of traffic is to be sent if there is more than one type possible. (Maybe there is a use for more than one type of pattern to be sent?)

The PC should be able to specify the number of bytes of traffic to send. If the PC can specify the number of bytes to be sent, then a number that is not a multiple of eight could be used to test the short packet handling (just a thought).

Prepared by Stephen M Moraco, kc0ftq@amsat.org

Issue: identifying cause of data loss

If a loss of data occurs in the data reported from the emulator to the PC, we must be able to tell the difference between:

- loss due to bad handling of byte-pipe protocol (loss at the Widget interface), or
- loss in trying to report the data to the PC (loss over the serial interface) .

This should not mean anything more expensive than data coming from the Widget interface should be formatted so that the visual shape of the data speaks to the correctness. Maybe it's always eight columns wide, maybe it's addressed so that we can tell which packet in sequence this data came from, etc. This could mean there is some counter tracking the number of bytes since power-on or form the beginning of the packet, or well, you get the idea.

Issue: testing of protocol errors

Two errors need testing in each of the two directions. For IN (the Module -> Widget direction) there two are:

1. A 2nd through 8th byte transfer might not be REQuested causing an input time-out to occur. (The first byte had to be REQuested or we would not be in the INPUT mode, yet. Hence, failures can only occur before bytes 2-8...)
2. A 1st through 8th byte de-assert of REQ might not occur also causing an input time-out to occur.

For OUT (the Widget -> Module direction), the two are:

1. A 1st through 8th byte may not be ACKnowledged causing an output time-out to occur.
2. The ACK may not be de-asserted from the 1st through 8th byte transfer attempt also causing an output time-out to occur.

Prepared by Stephen M Moraco, kc0ftq@amsat.org

Commanding via the Serial I/F

A small set of commands facilitate rigorous testing:

- Two configure commands setup the condition to be tested:
  - **Send <contentType><#bytes>**: defines the number of bytes to be sent to the IHU (PC)
  - **Fail <Type><byte#><packet#>**: defines the failure condition to be induced at specified location during the next transfers
- Two action commands: cause the emulator to do something:
  - **Go<When>**: starts the sending of data bytes to the widget as configured with a previous invocation of the **Send** configure command.
  - **Reset**: aborts the sending currently in progress. It also removes any fails or sends which have not yet taken place.

A typical session would consist of one or both configure commands followed by the 'go' action command. The configure commands define the test while the 'go' executes it.

Command parsing made simple

To simplify the parsing of the commands via serial the minimum of ASCII chars are used to describe the command and its parameters. The command parsing should be case insensitive. The following is a suggestion for making the commands easy to parse.

- The commands are to be single letter: (s)end, (f)ail, (g)o, and (r)eset.
- Fixed width parameters appear before those of arbitrary length when reading from right to left.
- **<contentType>** is a single letter: (i)ncrementing, (z)eros, or (o)nes. Where (i) means the byte value being sent is constantly being incremented starting at zero and wrapping back to zero from 255. The first eight values are placed in the first packet, next eight in the 2nd packet, and so on. (z) means the packet is filled with zeros, and (o) means all ones.
- **<#bytes>** are the number of bytes to be sent to the IHU (decimal: 1-32768)
- **<Type>** is one of four codes identifying the point in the handshake protocol, which is to fail. (r) REQ assert, (s) REQ de-assert, (a) ACK assert, and (b) ACK de-assert. Since we are case-insensitive I simply chose the next letter after (r) for REQ and (a) for ACK. So r+1 equals 's' which becomes REQ de-assert and a+1 = 'b' which becomes ACK de-assert.
- **<byte#>** is the value 0-7, which is the offset within the 8-byte CAN payload.
- **<packet#>** is a packet sequence number (decimal: 1-4096) indicating which packet in the stream should be affected. NOTE: 4096 is 32768 (from above) / 8 (number of bytes in the largest CAN packet.)
- **<When>** is either (n)ow, or following first (r)eceived packet.

Prepared by Stephen M Moraco, kc0ftq@amsat.org

Serial command examples

Given the above description, a sample set of commands is can now be presented.

Example test #1:

In this test, we setup to fail the output-side write of the 5th byte. We also specify a short final packet caused by the Module failing to send the final bytes.

Fb625

(f) Fail (b) ACK de-assert after byte 6 in packet 25

Si34

(s) Send (i) incrementing bytes, 34 of them (meaning 4 8-byte packets followed by a short 2-byte packet, which will cause the input pipe-timer to expire. The timer expiration will then cause a 2-byte short packet to be sent to the PC playing IHU.)

Gn

We are setup, execute the test (n)ow.

Example test #2:

In this 2nd test we see both the inbound and outbound channels will fail when they get to the right point. After both failure points are set up we see the request to send 40 bytes of zeroes. With no other influence, the output side will fail causing an output-timer expiration and reset of the output side. There should be no effect on the progress of the input side, which should later fail after packet 995.

Fa2995

(f) Fail (a) ACK assert after byte 2 in packet 995 # INPUT failure

Fs54

(f) Fail (s) REQ de-assert after byte 5 in packet 4 # OUTPUT failure

Sz40

(s) Send (z) zero filled bytes, 40 of them (meaning 5 8-byte packets will be sent.)

Gr

We are setup, execute the test only after (r)eceiving the first packet from the IHU.

Prepared by Stephen M Moraco, *kc0ftq@amsat.org*

Extra Credit

With a desire to go further than the minimum required, the following ideas could be next things to implement:

- Add a new Command: **(a)nswer <count><contentType>** – with the 'answer' action, the emulator replies to the pipe data sent to the widget, via CAN, as if it were a command of some sort.
  - **<count>** is 1-9 or 'r' for random (0-9)
  - **<contentType>** is (i)ncrementing, (z)eros, (o)nes, or (e)cho [where echo is a reply consisting of the packet received echoed back to the sender]
  - Answer would be used in place of (s)end when setting up tests

- Wire one of the remaining two unused digital output bits to either the PIC reset or an input bit, which the PIC code will check to see if a reset is being requested. In response to the reset, do the equivalent of the (r)eset command; busting out of the current test effort if one is in progress and clearing any pending failures.

By now, you have a sense of what we are trying to test. You have your own ideas of what would help make testing easier. Feel free to add those you feel would be of the greatest benefit.

Beyond Extra Credit

Still looking for more to do? ;-)

- If you happen to have a library implementation of Intel Hex reading/writing routines (*do not write it just for this project*!) then:
  - Instead of identifying the data pattern by a letter, accept the definition of the data to be sent to the IHU in the form of the contents of an Intel hex file. (This might require adding a 'download data' command to accept the hex file.)
  - Instead of dumping plain text to the Serial port, dump the data in Intel Hex format. This would provide a checksum for each output line thereby providing a means to verify that no data was lost during the write to the serial port. Each line could be the data of one packet and the address could be the packet sequence number from, say, when the module emulator was last powered up.

Prepared by Stephen M Moraco, kc0ftq@amsat.org