Embedding it better...

µTasker

µTasker Document

TELNET

Table of Contents

1. Introduction

µTasker is an operating system designed especially for embedded applications where a tight control over resources is desired along with a high level of user comfort to produce efficient and highly deterministic code.

The operating system is integrated with TCP/IP stack and important embedded Internet services along-side device drivers and system specific project resources.

µTasker and its environment are essentially not hardware specific and can thus be moved between processor platforms with great ease and efficiency.

However the µTasker project setups are very hardware specific since they offer an optimal pre-defined (or a choice of pre-defined) configurations, taking it out of the league of “board support packages (BSP)” to a complete “project support package (PSP)”, a feature enabling projects to be greatly accelerated.

This document discusses the implementation and use of the TELNET protocol.

2. Overview of TELNET

To do...

3. Configuring for TELNET use

To do...

4. Sending TELNET frames

To do...

5. Receiving TELNET frames

To do...

6. TCP Windowing

Telnet is useful for realising serial / LAN converters. Data received from the serial interface is transmitted to a destination IP address and frames received from a remote IP address are sent out over the serial interface. Using this technique it is very easy to allow equipment with a standard serial port but not Ethernet capabilities to be made Internet capable – it can then be monitored or controlled at a distance and TELNET is very suited to this job.

Assume the serial interface is 115'200b/s and it is connected to a high speed Ethernet. Initially you would think that the Ethernet (or better TCP protocol over the Ethernet) would have no problems with the 115'200b/s data rate and of course this is true, but only when TCP is using active windowing.

Consider the non-windowing case where say 512 bytes are received from the serial interface, packed into a TCP frame and sent to the destination IP address. The 512 bytes are collected in about 40ms and so to achieve the throughput of 115'200b/s a TCP frame must be sent once every 40ms. The actual transmission of the TCP frame over a short distance 100M Ethernet connection will take about 52us and so is almost negligible. But we mustn't forget the fact that many TCP solutions use what is called Delayed ACK, which means that the receiving side doesn't acknowledge a received TCP frame immediately but instead will wait up to about 200ms, hoping to be able to piggy-back the ACK along with some data which the application may be sending back. If the ACK is sent after 200ms and the next TCP frame is not sent before the previous ACK has been received (non-windowing) then the throughput is limited to about 5 TCP frames a second but in our case we really need 25 frames a second to achieve the 115'200b/s serial throughput. If you try it you will see that the maximum serial speed which can be transmitted will be only around 19'600b/s without the serial interface having to use flow control to limit data. This also assumes a short distance with negligible transmission delay – a round trip time of zero, whereas over the Internet the ACKs may well be subject to even longer delays due to transmission distances. So in fact our fast Ethernet interface can only just handle 19'600b/s from the serial interface (on a good day) – what a let down…!!

So it is clear that a non-windowing solution is not much use for this application and so windowing support is required. It can be activated by setting #define SUPPORT_PEER_WINDOW in config.h and the maximum number of open windows is defined by WINDOWING_BUFFERS – 2 is in fact usually enough to unblock the situation described above and more will only really be noticeable when there are long transmission delay such as in the Internet. 4 are fairly generous and suggested as default value.

We have seen that windowing helps to achieve a high data throughput at the price of increased protocol complexity and a bit more RAM. We'll take a look at how the µTasker adds windowing to the TELNET protocol in a manor using as little additional resources as possible.

The following variables are added to the TCP_TX_BUFFER when windowing is enabled:

```
unsigned short usOpenBytes;     // Total bytes not yet acked
unsigned char  ucCwnd;          // congestion windows
unsigned char  ucOpenAcks;      // Total frames not yet acked
unsigned char  ucPutFrame;      // tx_window for next use
unsigned char  ucGetFrame;      // first outstanding tx_window
TCP_WINDOWS_LIST tx_window[WINDOWING_BUFFERS]; // window management
```

7. Congestion window - `ucCwnd`

When a TCP connection is established it is not good practice to bombard the destination with TCP frames just because we are working with windowing and don't wait for ACKs to each frame. We don't yet know what the characteristics of the connection are like, such as the round trip delay time. So we start carefully using a counter known as the congestion window and don't send more than one additional TCP frame before the first transmission is acknowledged. We then increase it when we see that these two TCP frames were acknowledged together. We continue increasing it until either the maximum number of windows is being sent or the number of outstanding frames without acknowledgements stops increasing (an indication of the round trip time, which will hardly pass 2 outstanding frames at a time on a network with small round trip delays).

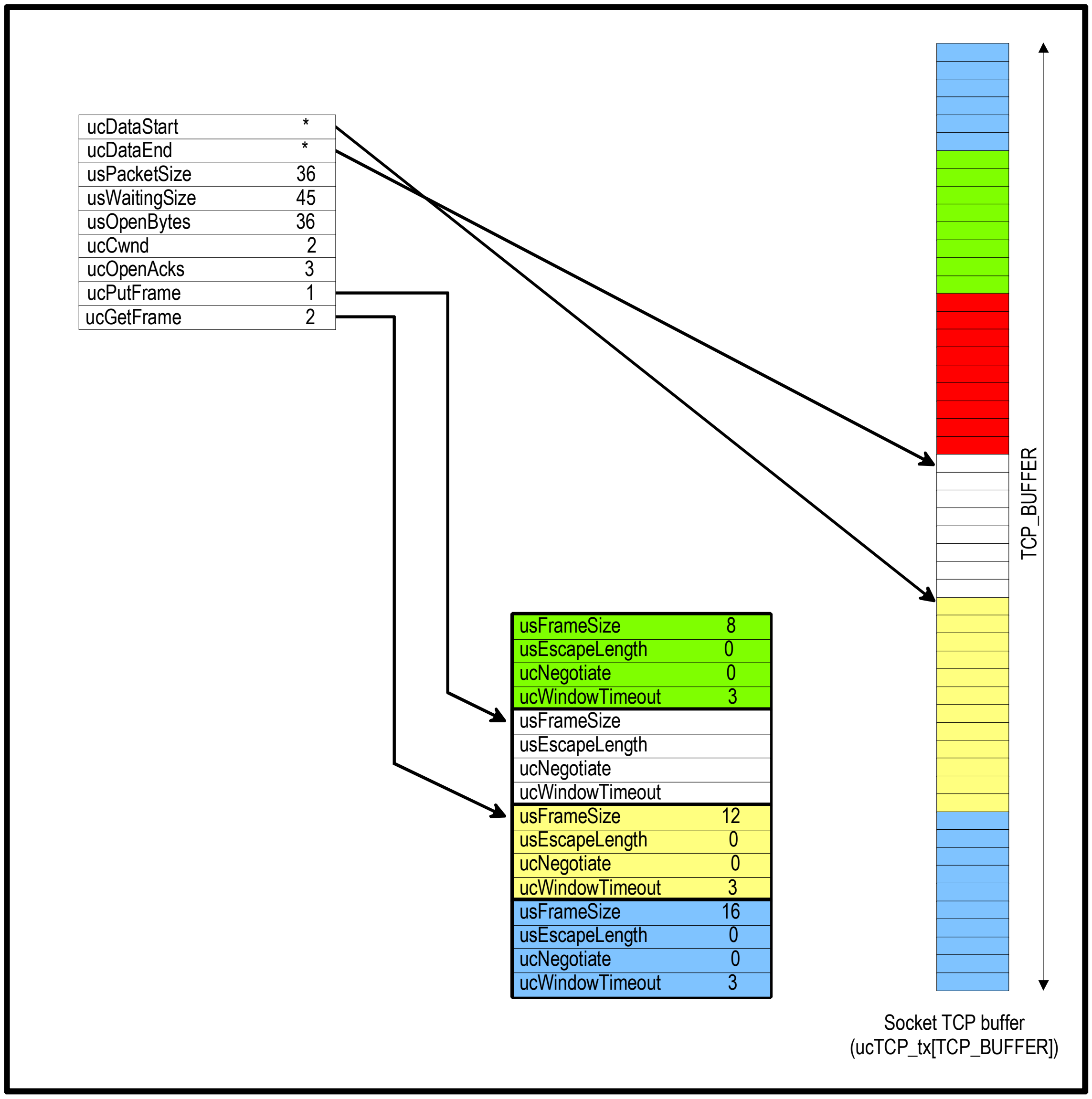
8. Window management

Just image if each transmitted frame were to be delivered without errors - it would make life so simple. We could just send the data to our heart's content and be sure that the destination will receive it, and then we could get on with what else we are doing and just trash any copies of the data which was transmitted since the receiver has it and we probably don't need it again.

Of course life is not as simple as that and we have to be capable of retransmitting any data which we detect as not arriving at its destination. This requires that we do not trash it as soon as it has been sent but keep a copy up to the point that we can be very sure that it has actually successfully arrived. Since we are sending the data in a number of windows it must always be possible to regenerate one or more TCP frames which don't seem to have arrived correctly at the destination.

An ACK matching a transmitted window, or series of transmitted windows, received a certain time after sending them, is a sign the data has successfully arrived and that we no longer need to monitor them. When there is no ACK from a transmitted window or series of transmitted windows after a longer delay it is an indication that data has been corrupted or lost underway and it is our job to retransmit, a number of times if necessary, until they are either finally received or it becomes clear that the connection has broken down.

The following illustration shows the case where three TCP frames have been transmitted and are awaiting corresponding ACKs. In addition there are some bytes waiting to be sent but cannot presently be sent using windowing since the congestion window is not open wide enough. It should always be possible to regenerate any of these frames or any number of them if necessary. It must also be possible to recognise that one or more of them have been successfully acknowledged so that the memory space can be freed for use by subsequent frames.


ucDataStart *
ucDataEnd *
usPacketSize 36
usWaitingSize 45
usOpenBytes 36
ucCwnd 2
ucOpenAcks 3
ucPutFrame 1
ucGetFrame 2
usFrameSize 8
usEscapeLength 0
ucNegotiate 0
ucWindowTimeout 3
usFrameSize
usEscapeLength
ucNegotiate
ucWindowTimeout
usFrameSize 12
usEscapeLength 0
ucNegotiate 0
ucWindowTimeout 3
usFrameSize 16
usEscapeLength 0
ucNegotiate 0
ucWindowTimeout 3
TCP_BUFFER
Socket TCP buffer
(ucTCP_tx[TCP_BUFFER])

9. ACKed frames

When windowed TCP frames are acknowledged then they can be either acknowledged individually, in which case the ACKs to each frame arrive individually some time later and when each ACK is received, the corresponding windowed segment can be trashed since it has obviously arrived at the destination. This is performed by simply incrementing `ucGetFrame` by one and updating the buffer block appropriately.

However it is more common that there is not an acknowledgement received for each TCP window frame sent but rather there is one ACK received for at least two TCP window frames. The reason is that the delayed ACK is started on reception of the first TCP windows frame and as soon as the second TCP frame window is received by the destination it sends a single ACK for the two received frames. Therefore it is possible to trash as many TCP windows frames as signalled by the received ACK. Again this is easily performed by incrementing `ucGetFrame` as many times as necessary and updating the buffer block appropriately.

10. Out of order ACKs

It is not excluded that TCP frames arrive out of order and so also an ACK can arrive out of order. Assume that one ACK was sent to acknowledge the first and second TCP window frames, followed by a third to acknowledge the third TCP window frame. Somewhere out in the Internet the second ACK frame overtook the first and so arrived first.

This frame actually acts as an acknowledgement for all three windows and so all of our backed up data is trashed and we are already happy. Then the initial ACK is received but is silently discarded as there is no outstanding data waiting to be acknowledged.

Therefore such out of order acknowledgments do not cause any difficultly at all….

11. Lost ACKs

Lost ACKs can be due to several reasons. Here is a list of possibilities which is not absolutely complete but will give you the general idea:

- The ACK arrived but is corrupted and so ignored.
- The ACK got lost underway and never arrives at our receiver.
- The ACK is never sent by the remote side due to a software error there.
- The ACK is never sent by the remote side since our original data never arrived or arrived with an error.

In our example case we have sent three TCP windows frames and are expecting between one and three ACK frames from the remote side. One, two or three of these could never arrive, indicating that some or all of our data was possibly never received at the other side. We must be able to recover from each of these possibilities and so we will analyse them now to show that we can indeed do so and the TCP connection will remain reliable, although throughput will obviously be reduced for a short time.

The µTasker demo project included support for making tests of the TCP implementation and this was used to verify and analyse the real reaction to loss of message. The support is activated in config.h using the `#define TCP_TEST`. The project requires also TELNET support with `#define USE_TELNET`, which will automatically activate the necessary windowing support.

The tests were performed first in the simulator environment and the Ethernet frames recorded with Ethereal for analysis and also for playback to analyse and verify the code operation.

The same tests can be performed on any supported target hardware for comparison and confirmation that the reaction and performance is equivalent to the results with the simulator.

The test is performed in the menu "TCP Test", using the command "window". This command accepts a parameter which defines a corrupted message to be sent in the future. "window 0" means perform the test with no errors. "window 1" means corrupt the next TCP frame which is sent. "window 2" means corrupt the second TCP frame which is sent, etc.

After a short delay to allow any present TELNET activity to terminate the test program sends a number of short text messages one after the other at intervals of 50ms. This feeds the TCP buffer with ...to do...

12. Conclusion

This document is in progress and has not been officially released.

Modifications:
- V0.1 21.4.2010 – provisional version for the documentation page