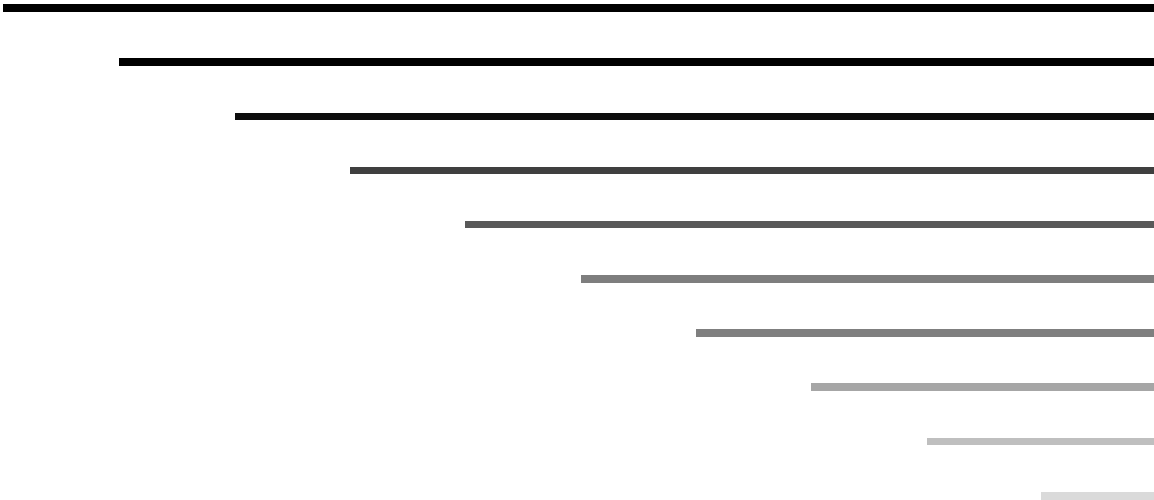
Embedding it better...

µTasker Document

µNetwork



Copyright © 2009 M.J.Butcher Consulting

Table of Contents

1. Introduction 3
2. Configuring to use µNetwork 3
3. Global Messages 5
4. Single Destination Addresses 8
5. Conclusion 11

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 µNetwork which is a simple but fast and efficient protocol used to enable µTasker projects to run as distributed tasks within a local network of devices. The protocol is integrated within the internal message transport system and is reliable in local networks due to fast automatic message repetitions if initial attempts could not be delivered. The operation is encapsulated and is therefore transparent to the application which doesn't specifically need to know whether a task communication partner is local or at a remote node, allowing for flexible and even dynamic distributed implementation. The µNetwork support is integrated in µTasker release from V1.3.

2. Configuring to use µNetwork

The define `SUPPORT_DISTRIBUTED_NODES` should be set in `config.h` to enable the µNetwork protocol.

There are various parameters in the same header file which can be set to suit the project.

```
#ifdef SUPPORT_DISTRIBUTED_NODES
    #define PROTOCOL_UNETWORK 0x754e  // uN - uNetwork protocol
    #define MAX_NETWORK_NODES 2
    #define UPROTOCOL_WITH_RETRANS
    #define MAX_STORED_MESSAGES       10
                                      // set TIMER_QUANTITY to at
                                      least this amount since each
                                      message needs a timer
    #define MAX_UPROT_MSG_SIZE        255
    #define UPROT_RETRANS_TIME        (CLOCK_LIMIT)(0.05 * SEC)
    #define MAX_UPROT_RETRANS_ATTEMPS 5
    #define GLOBAL_TIMER_TASK         // needs global timer to
                                      operate
    #define EXP_BACKOFF
    #define USE_TIMER_FUNCTION_EVENT  // global timer handles
                                      messages to itself as function
                                      event
#endif
```

The above example is a simple case where there are just 2 nodes in the network and `MAX_NETWORK_NODES` is thus set to 2. The protocol uses Ethernet for transmission and the Ethernet protocol type is defined to be 0x754e. This allows the frames to be distinguished

from other protocol types such as IP types (0x0800) and displayed as 'uN' (short for µNetwork) using Ethereal

UPROTOCOL_WITH_RETRANS is set because the operation should be reliable. If this is not set, the remaining configuration defines are no longer required but if frames were not to arrive at the destination this would be lost with no warning. Normally the reliable operation is important and so this will be used.

MAX_STORED_MESSAGES defines buffer space for keeping backups of this many messages of maximum size MAX_UPROT_MSG_SIZE. The backup of each transmitted frame is kept until an acknowledgment has been received, after which is it invalidated (destroyed) and the buffer space is again free to back up further frames. MAX_UPROT_MSG_SIZE also defines the largest frame content to be transmitted and is set in the example to 255, which is the maximum size of internal messages.

UPROT_RETRANS_TIME is the time within which an acknowledge to a transmitted frame is expected- In a LAN this can be quite short to ensure that lost frames are retransmitted quickly to avoid system delays.

MAX_UPROT_RETRANS_ATTEMPS defines the maximum of times a frame will be repeated before giving up. Normally the maximum repetition count will only be reached if there is a drastic failure in the network (such as a disconnected cable) and a system event is generated to inform of the occurrence.

EXP_BACKOFF defines that retransmission attempts use exponential back-off. Each time a particular frame needs to be transmitted, the timeout period doubles. This is recommended since lost frames due to noise or interference will usually be correct after the first attempt, whereas non response due to temporarily occupied receiver processor may require longer periods. A short repetition delay may cause apparent message loss since the number of repetitions is quickly used up – using a back-off algorithm allow more time without having to repeat as much.

GLOBAL_TIMER_TASK enables the global timer task which is responsible for monitoring the transmission timeouts. There are two further configuration defines associated with GLOBAL_TIMER_TASK use, these are

```
#define TIMER_QUANTITY 10      // the number of global timers
                                    required

#define GLOBAL_HARDWARE_TIMER  // Global timer task supports
                                    hardware resolution
```

Here the maximum quantity of timers managed by the global timer task can be set up as well as whether it works with software or hardware tasks. If short accurate time outs are important (for example when reaction time even by one repetition is to be optimised) then it is advisable to use the hardware timer support and quite short timeout values.

USE_TIMER_FUNCTION_EVENT enables events in the global timer task to be handled directly by a function rather than generating a system event to an owner task. This should be activated when using the µNetwork since it makes use of this for efficiency reasons.

Further to the basic configuration of the protocol it is necessary to decide how the nodes in the network are addressed. This part could well be application specific especially when the network is operating together with IP, in which case each node will usually require use of its normal MAC address. In cases where the network is not shared with other devices and the actual network address is not critical it is possible to use the same values as the node number for communication.

In TaskConfig.h it is recommended to add the network setup as follows.

```
#ifdef OPSYS_CONFIG   // this is only set in the hardware module
CONFIG_LIMIT OurConfigNr = DEFAULT_NODE_NUMBER;
// in single node system this default value can be fixed

#ifdef SUPPORT_DISTRIBUTED_NODES
    NETWORK_LIMIT OurNetworkNumber = 0;
// this value must be set (to non-zero) on startup individually for
each node in the network
    unsigned char OurNetworkExtension = 0;
    const unsigned char ucNodeMac[MAX_NETWORK_NODES][MAC_LENGTH] = {
        {0x00, 0x00, 0x00, 0x00, 0x00, 0x01},
        {0x00, 0x00, 0x00, 0x00, 0x00, 0x02}
    };
#endif
…
…
#endif
```

Here a very simple case is shown where the MAC addresses of the nodes in the network are the same as the node addresses themselves. One card configures OurConfigNr to be 1 and the other configures it to be 2. `ucNodeMac[]` can be in FLASH since the addresses are fixed. If fixed MAC addresses were to be available, these could also be set here but would require the project to be recompiled for each configuration which only makes sense in 'one-off' systems. It could be necessary to first dynamically configure these in RAM before using them and may be later retrieving the values from the parameter system. Here the designed can decide what is best for the particular project and we will stay with the simple version for further discussion.

3. Global Messages

Global messages have the following characteristics:

- they are received by all nodes in the network
- they are not acknowledged at the protocol level and are thus potentially unreliable
- they are neither backed up in memory nor repeated (due to the fact that they do not expect an acknowledge)
- Global messages are broadcast frames at the Ethernet level

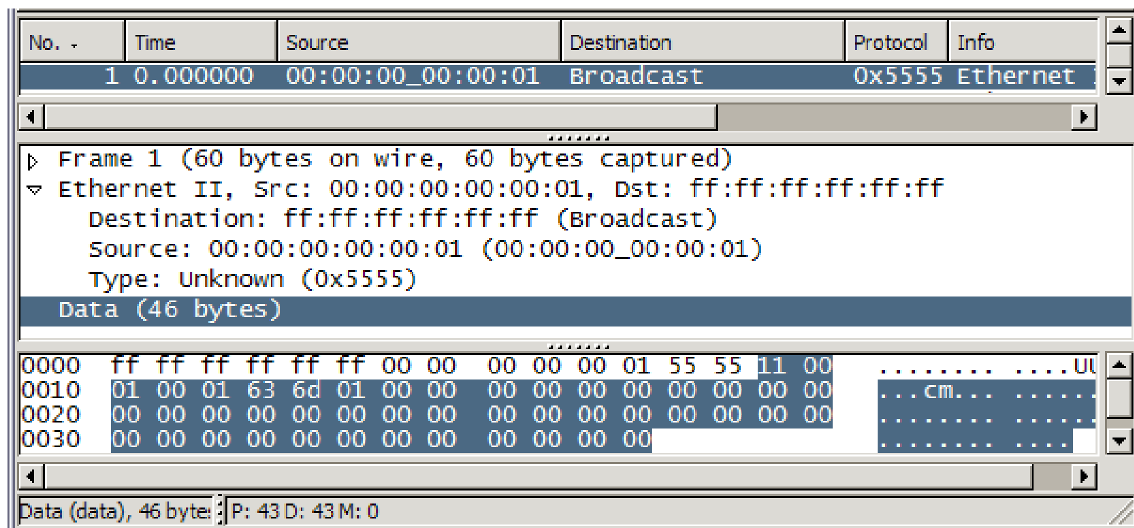
They are however very useful for system configuration or various non-critical synchronisation purposes. It should be remembered that in a normal LAN the level of lost frames due to corruption is normally extremely low – the higher risk of frame loss is overrun of input buffers at the nodes themselves and this depends on network activity (mainly broadcast activity) and the MAC receive resources at the processor (processor and possibly project setup dependent).

The following is an example of a simple global message being sent to all devices on the network using the following routine.

```
void fnSendMessageG(unsigned char ucMessageType)
{
    unsigned char ucMessage[HEADER_LENGTH + 1];         // reserve space for message

    ucMessage[MSG_DESTINATION_NODE] = GLOBAL_MESSAGE;   // global message
    ucMessage[MSG_SOURCE_NODE]      = ucNodeAddress;    // own node
    ucMessage[MSG_DESTINATION_TASK] = CONFIG_TASK;      // destination task
    ucMessage[MSG_SOURCE_TASK]      = MASTER_TASK;      // own task
    ucMessage[MSG_CONTENT_LENGTH]   = 1;                // length of data
    ucMessage[MSG_CONTENT_COMMAND]  = ucMessageType;    // the message type

    fnWrite (INTERNAL_ROUTE, ucMessage,  (HEADER_LENGTH + 1));// send message via
                                                        internal transport system to
                                                        destination
}
```



This Ethereal recording of the transmitted frame on the network shows the protocol type 0x754e (uN) which is unknown to Ethereal but we recognise it as a µNetwork frame.

The broadcast destination address is seen to be ff:ff:ff:ff:ff:ff and our node address is the same as the source address 00:00:00:00:00:01. The minimum Ethernet frame size of 60 bytes has been respected although the data content doesn't fill all of the 46 data bytes. The unused bytes a have been set to zero.

The data content is constructed from two parts. First a µNetwork header and then the µTasker internal data format. This can be illustrated as follows:

µNetwork header: 0x42 0x00 0x01

> 0x42 (displayed as B for Broadcast) means that the frame doesn't require an acknowledgement (it is a broadcast)
> 0x00 is a sequence number of the transmitted frame. A zero is the first transmission and causes the receiver to expect the next frame to have a 0x01 sequence number.
> 0x01 is the sender's node address (in this case it is equal to the source Ethernet address but must not be)

µTasker internal message: 0x00 0x01 0x63 0x6d 0x01 0x00 [the rest are filling zeros]

> The µTasker header indicates that the destination node is 0x00 (define GLOBAL_MESSAGE), the source node is 0x01, the destination task has the name 'c' and the source task has the name 'm' [in the project where this recording was taken from there are tasks 'configuration' and 'master']. The data content length is 1 byte and the byte has the value 0x00 – in the project which this was taken from this is defined as the request for configuration information which all other devices then return to the master. The master can then check that all expected slaves have answered, do a reality check on the returned data and then continue as required. It is a good example of such uses for a global broadcast message.

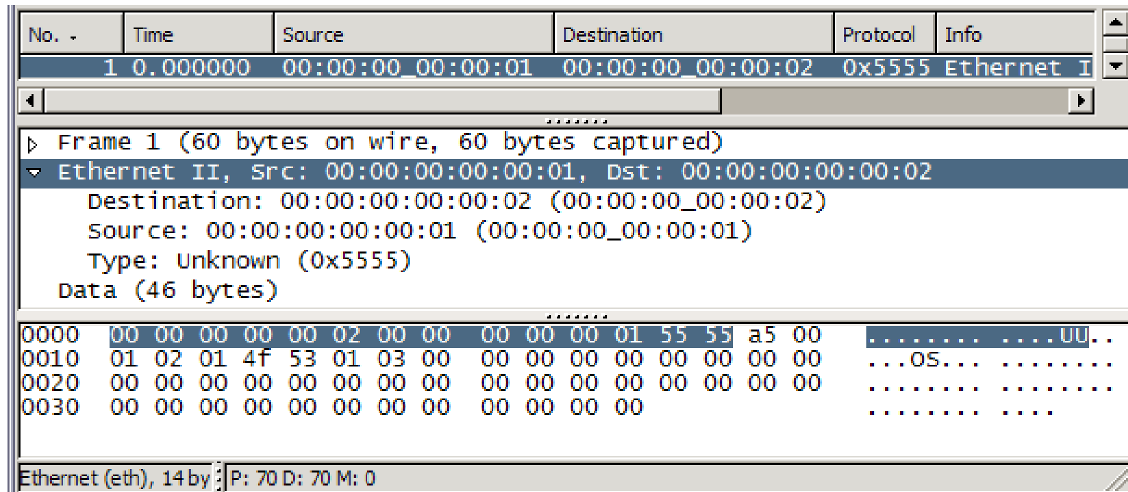
The transmission of a global message is very simple and it is obvious that the µNetwork support has recognised that the message is to be routed to the network rather than stay on the local board (it stays on the local board when the source and destination node addresses are identical). The message has been packed into a global µNetwork frame and sent as an Ethernet broadcast frame.

The Ethernet broadcast frame is received by all devices on the network and is recognised as a µNetwork protocol frame and handled by the receiver function in µNetwork.c. Using the routine information in the µTasker message header the message part of the frame is then transmitted internally (queues) to the destination task, which wakes the task and can read the message from its input queue in a compatible manor to when reading any internal message.

4. Single Destination Addresses

Most frames sent between distributed nodes will be to a specific destination task. Such frames have the following characteristics:

- they are received by only the destination node in the network
- they are acknowledged at the protocol level and repeated if no acknowledge is received within a defined time window. These frames are thus reliable.
- they are backed up in memory and there is a limit to the number of outstanding frame acknowledgements (the limit can be set to ensure that a certain application always has adequate buffer space)
- Single destination messages are sent to a unique destination address at the Ethernet level



This Ethereal show a recording is a single destination frame. Notice that the destination address is the MAC address of the destination device 00:00:00:00:00:02. The

The µNetwork header is 0x4d 0x00 0x01

0x4d means that the frame requires an acknowledgement (displayed as M for Message)
0x00 is the sequence number of the transmitted frame. A zero is the first transmission and causes the receiver to expect the next frame to have a 0x01 sequence number
0x01 is the sender's node address (in this case it is equal to the source Ethernet address but must not be)

µTasker internal message: 0x02 0x01 0x4f 0x53 0x01 0x03 [the rest are filling zeros]

The µTasker header indicates that the destination node is 0x02, the source node is 0x01, the destination task has the name 'O' and the source task has the name 'S' [these are examples of tasks in the project from which the recording was made]. The data content length is 1 byte and the byte has the value 0x03 – an event in the project used for the recording.

The only actual difference between sending a global and a single destination address is the define of the destination node. If the destination node is not defined as GLOBAL_MESSAGE it is for a specific node.

For example by using the following line, the global message routine now sends the message to the specific node

```
ucMessage[MSG_DESTINATION_NODE] = NODE_WHERE_O_IS; // specific destination node
```

where `NODE_WHERE_O_IS` is defined as 2 since the task 'O' is on this node. Should later the position of the task 'O' be moved to a different node, one define change will change all routing appropriately. If the task 'O' is local to the sending node it will result in the transmission of a local message rather than one which uses µNetwork so even moving the task back to node 1 will not result in any problems. The µNetwork ensures that the correct route is made and this is transparent to the application.

Should there be no acknowledge from the destination, µNetwork will repeat the message up to the maximum defined attempts. If the frame still doesn't get acknowledged a local interrupt event UNETWORK_FRAME_LOSS will be sent to a task defined to receive such warnings:

For example, by defining (in TaskConfig.h)

```
#define UNETWORK_MASTER                    MY_MONITOR_TASK
```

the task MY_MONITOR_TASK will receive such network events and can decide what action is to be taken there.

If more flexibility is required so that the destination task can also be dynamically modified, using

```
UTASK_TASK network_monitor_task = MY_MONITOR_TASK;

#define UNETWORK_MASTER                    network_monitor_task
```

will allow the task to be changed as required.

```
No.   Time        Source              Destination         Protocol  Info
    5 0.000503    00:00:00_00:00:02   00:00:00_00:00:01   0x5555    Ethernet I

▷ Frame 5 (60 bytes on wire, 60 bytes captured)
▽ Ethernet II, Src: 00:00:00:00:00:02, Dst: 00:00:00:00:00:01
    Destination: 00:00:00:00:00:01 (00:00:00_00:00:01)
    Source: 00:00:00:00:00:02 (00:00:00_00:00:02)
    Type: Unknown (0x5555)
  Data (46 bytes)

0000  00 00 00 00 00 01 00 00  00 00 00 02 55 55 ff 00   ........ ....UU..
0010  02 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00   ........ ........
0020  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00   ........ ........
0030  00 00 00 00 00 00 00 00  00 00 00 00               ........ ....

Data (data), 46 bytes  P: 70 D: 70 M: 0
```

This Ethereal recording shows the acknowledgement to the previous frame. There is no µTasker message content but only the following µNetwork header

0x41 0x00 0x02 [plus zero padding to the minimum Ethernet frame length]

0x41 is an acknowledge (displayed as A for Ack)

0x00 is the sequence number which is being acknowledged

0x02 is the node address of the device sending the acknowledge

The sequence number is important because it allows an acknowledge to be matched to a transmitted frame and thus allows the transmitter to be sure exactly which frame has been acknowledged. Further, it allows the receiver to be sure that it is not handling repetitions as new frames – this can otherwise happen should an acknowledge frame not arrive.

The transmitter doesn't have to wait for a frame to be acknowledged before continuing with further message transmissions. Only the number of outstanding acknowledges is limited to the value `MAX_STORED_MESSAGES`. Should there be this many open acknowledges and still more messages be sent, the messages will be transmitted but will not have be able to repeat – the µTasker header is marked with 0x11 to indicate that the frame is not to be acknowledged. This simply means that the space for an adequate number of backed up messages should be foreseen in any practical system. If ever a non-global message is sent with the un-acknowledge attribute it is an indication of too few buffers, but not necessarily a network error since it will very probably still be correctly received.

5. Conclusion

This document has introduced the µNetwork protocol which is available as part of the µTasker project package. Based on network communication via Ethernet it allows communication in a distributed system in a fast, efficient and reliable manor.

Queuing, routing and reliable message delivery is taken care of in the µNetwork protocol layer and the application doesn't need to know where the destination task is located – the message will automatically be delivered locally (internal routing) or externally (network routing) depending on the exact location in the distributed system.

Modifications:
- V0.01 7.4.2007 First version

- V0.02 4.9.2007 Changed message coding to improve monitoring / debugging and added exponential retransmission back-off.

-V0.03 15.1.2009 Reformatted and conclusion added.