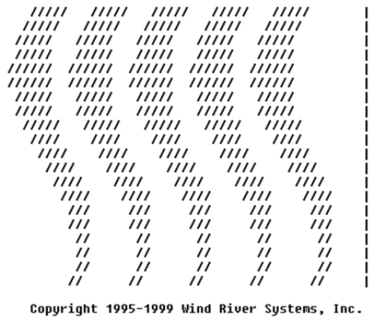
FYS 4220 – 2011 / #4

Real Time and Embedded Data Systems and Computing

Message based Process-Process Synchronization and Communication

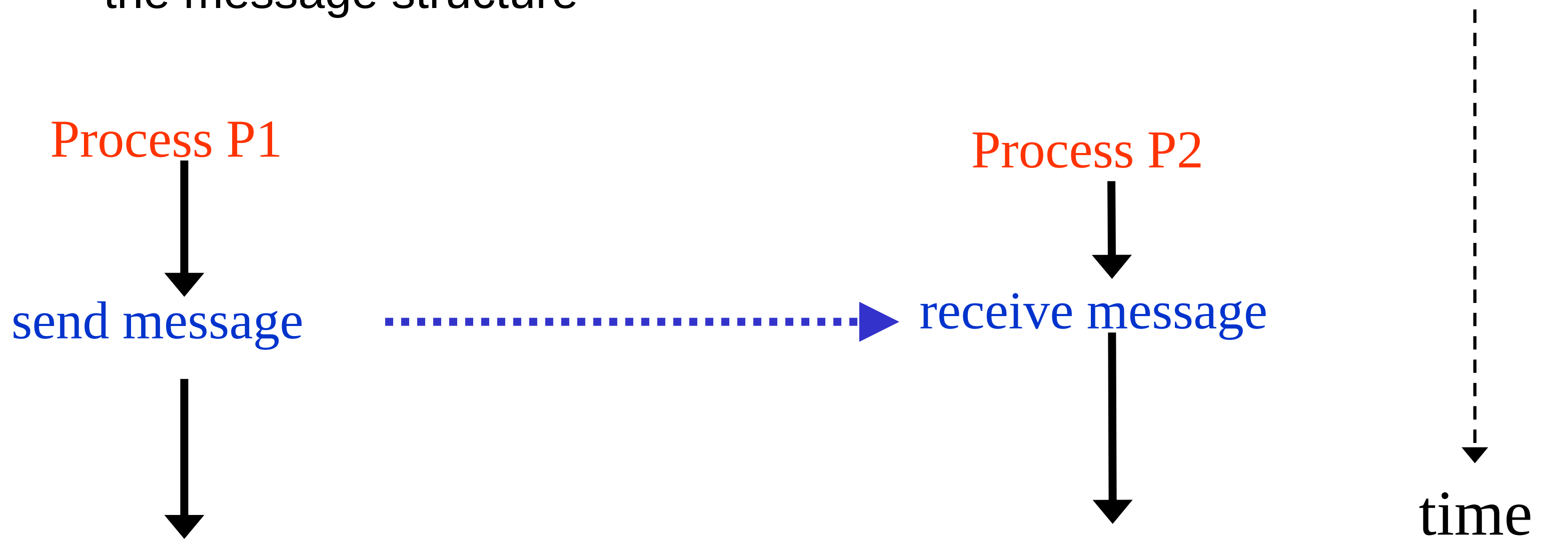


```
T O R N A D O

Development  System

Host  Based  Shell

Version  2.0.2

Copyright 1995-1999 Wind River Systems, Inc.
```

T.B. Skaali, Department of Physics, University of Oslo)

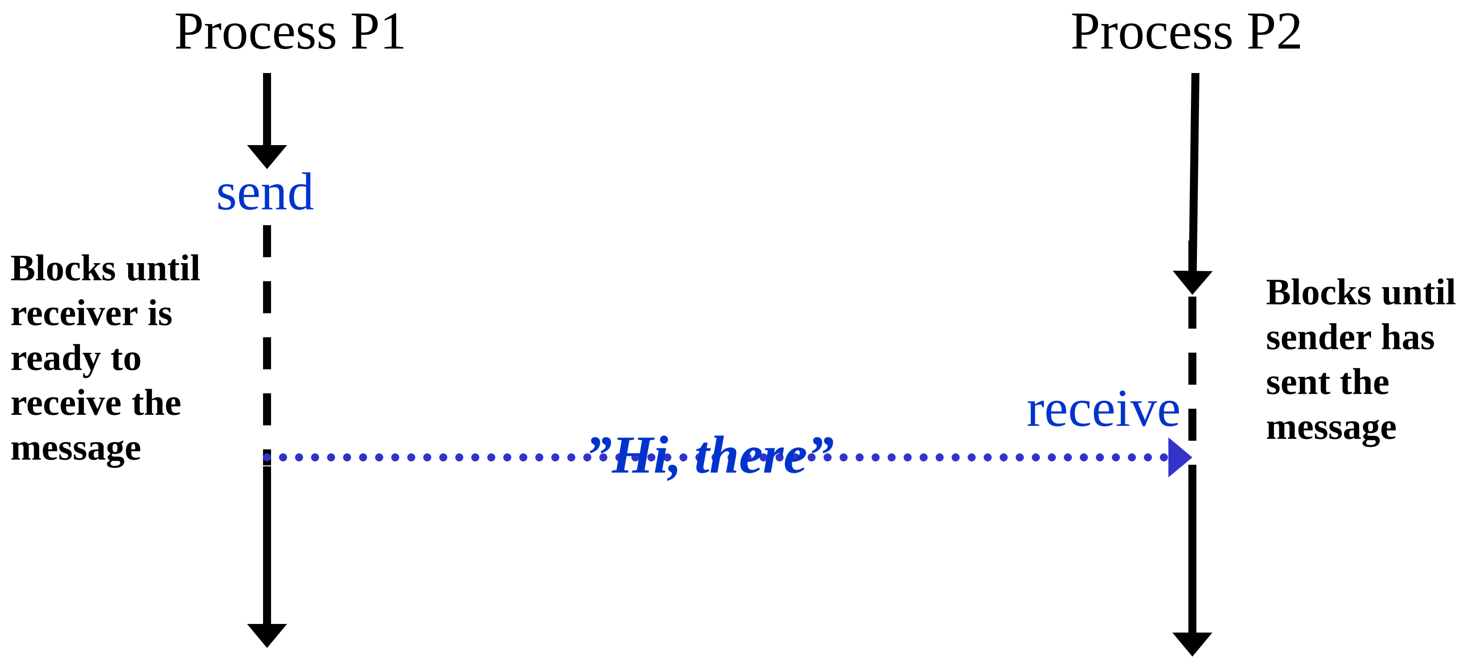
Process-to-process messages

- Use of a single construct for both synchronisation and transfer of data
- Three issues:
  - the model of message transfer, synchronous vs. asynchronous
  - the method of process naming
  - the message structure



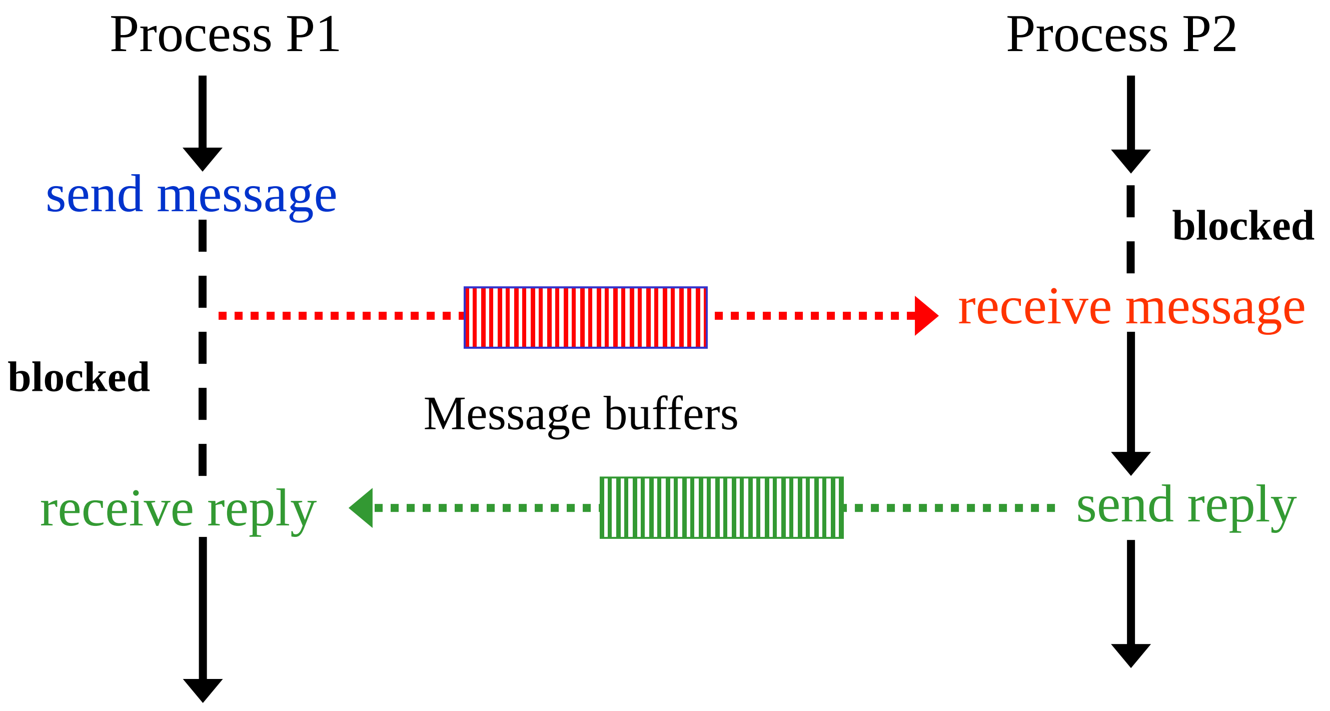
Synchronous message handling

- No buffering, space for only one message, blocking
    - Known as a rendezvous in Ada
  - Analogy: a telephone connection, or the HTTP protocol
  - A send-receive pair is both data transfer and synchronization
    - But how can receiver process know that there is a message for her?


Process P1
Process P2
send
receive
Blocks until receiver is ready to receive the message
Blocks until sender has sent the message
"Hi, there"

Buffered communication

- Buffer space required
  - Buffer write/read operations
  - Blocking occurs when buffer is full for write or empty for read
    - However, POSIX Asynchronous I/O is non-blocking!

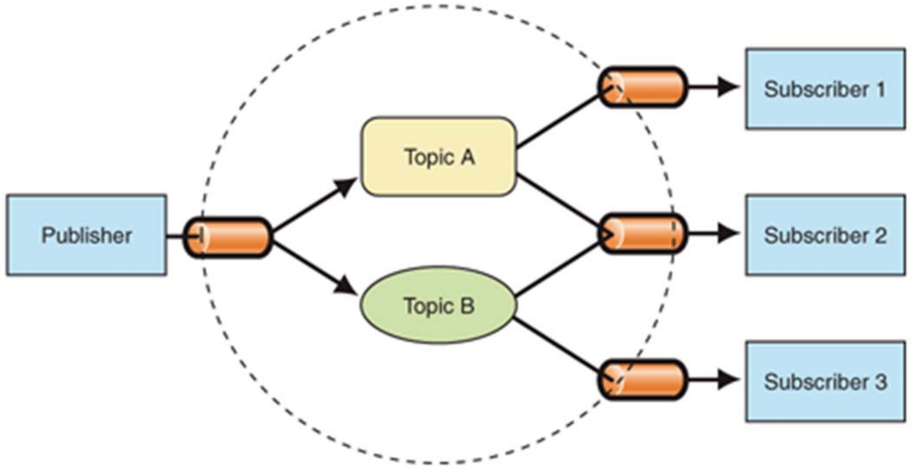


Asynchronous communication

- For example, an application may need to notify another that an event has occurred, but does not need to wait for a response.
- Another example occurs in publish/subscribe distributed systems, where an application "publishes" information for any number of clients to read. (A Real-Time example is the CERN ALICE High Level Trigger system comprising ~1000 processors)
- In both these examples it would not make sense for the sender of the information to have to wait if, for example, one of the recipients had crashed.
- However, the communication may include a reply message, in this case the initial sender may want to read the answer when it suits her.

Publisher/subscribers

- Searching the web for this method, one sees that it is mainly described in the context of general, distributed systems, for instance in servicing customers. http://msdn2.microsoft.com/en-us/library/ms978603.aspx
- However, it is also very interesting for distributed data acquisition / processing configurations



Some problems with asynchronous communication

- Potentially infinite buffers are needed to store unread messages
- The programs get more complex, in particular when signals (to be described in a later lecture) are used to notify the receiver
- It is probably more difficult to prove the correctness of a system
- What to do with messages which seems to be never read?
  - In the IEEE 1596 Scalable Coherent Interface one can specify ”time-of-death” for a message

UNIVERSITY OF OSLO

Process naming

- Two distinct sub-issues
  - direction versus indirection
  - symmetry
- With direct naming, the sender explicitly names the receiver:

  send <message> to <process-name>

- With indirect naming, the sender names an intermediate entity (e.g. a channel, mailbox, link or pipe):

  send <message> to <message queue>/channel/mailbox

(ref. B&W)

Process naming (contd)

- A naming scheme is symmetric if both sender and receiver name each other (directly or indirectly)

  send <message> to <process-name>
  wait <message> from <process-name>

  send <message> to <mailbox>
  wait <message> from <mailbox>
- It is asymmetric if the receiver names no specific source but accepts messages from any process

  wait <message>
- Asymmetric naming fits the client-server paradigm
- With indirect the intermediary could have:
  - a many-to-one structure
  - a many-to-many structure
  - a one-to-one structure
  - a one-to-many **(as in publisher/subscribers)**

(ref B&W)

”Message queues”

- The generic name Message Queues is a software-engineering component used for interprocess communication or inter-thread communication within the same process. It uses a queue for messaging – the passing of control or of content
- Message queues provide an asynchronous communications protocol, meaning that the sender and receiver of the message do not need to interact with the message queue <u>at the same time</u>. Messages placed onto the queue are stored until the receiver retrieves them.

VxWorks message queues

- Modern real-time applications are constructed as a set of independent but cooperating tasks. While semaphores provide a high-speed mechanism for the synchronization and interlocking of tasks, often a higher-level mechanism is necessary to allow cooperating tasks to communicate with each other. In VxWorks, the primary intertask communication mechanism within a single CPU is *message queues.* This is also seen in WindView traces.
- Message queues allow a variable number of messages, each of variable length, to be queued. Any task or Interrupt Service Routine (ISR – more about that in a later lecture) can send messages to a message queue. Any task can receive messages from a message queue. Multiple tasks can send to and receive from the same message queue. Full-duplex communication between two tasks generally requires two message queues, one for each direction

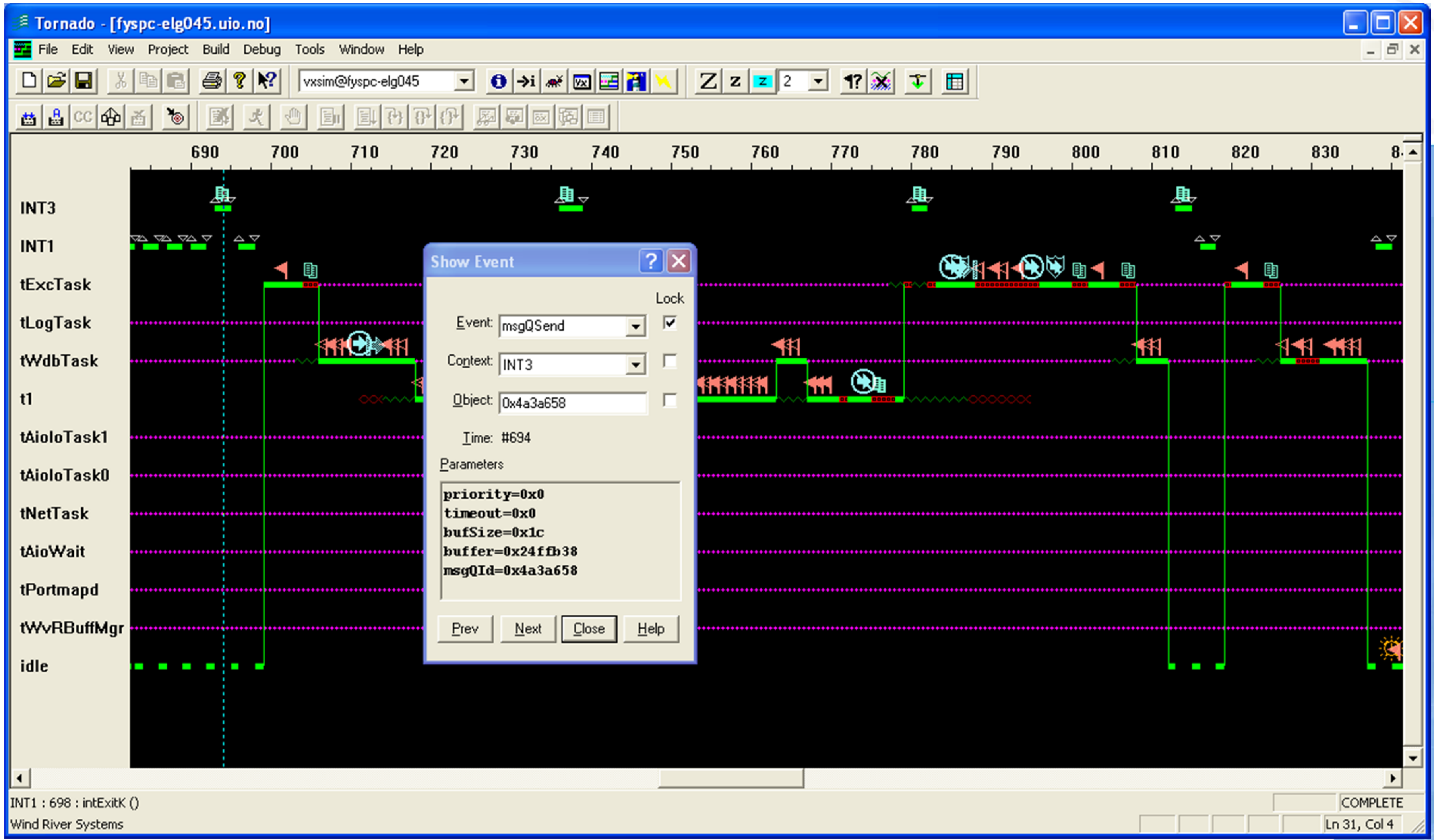
UNIVERSITY OF OSLO

”Hello World” – intertask communication

```
#include <iostream.h>

using namespace std;

void main()
{
   cout << "Hello World!" << endl;
   cout << "Welcome to C++ Programming" << endl;
}
```

WindView trace of ”Hello World”, see next page, note the message queue operations

WindView trace of Hello World



Event: msgQSend ✓
Context: INT3
Object: 0x4a3a658
Time: #694
Parameters

```
priority=0x0
timeout=0x0
bufSize=0x1c
buffer=0x24ffb38
msgQId=0x4a3a658
```

INT1 : 698 : intExitK ()

Ln 31, Col 4

VxWorks Wind and POSIX message queues

- There are two message-queue subroutine libraries in VxWorks
- The first of these, **msgQLib**, provides Wind message queues, designed expressly for VxWorks;
- the second, **mqPxLib**, is compatible with the POSIX standard (1003.1b) for real-time extensions

VxWorks Wind message queues

- Wind message queues are created and deleted with the routines shown below. This library provides messages that are queued in FIFO order, with a single exception: there are two priority levels, and messages marked as high priority are attached to the head of the queue.
  - ***msgQCreate*( )** Allocate and initialize a message queue
  - ***msgQDelete*( )** Terminate and free a message queue.
  - ***msgQSend*( )** Send a message to a message queue
  - ***msgQReceive*( )** Receive a message from a message queue

VxWorks Wind create message queue

- A message queue is created with ***msgQCreate*( )**. Its parameters specify the maximum number of messages that can be queued in the message queue and the maximum length in bytes of each message. Enough buffer space is preallocated for the specified number and length of messages

```
MSG_Q_ID msgQCreate
    (
    int maxMsgs,           /* max messages that can be queued */
    int maxMsgLength,      /* max bytes in a message */
    int options            /* message queue options */
    )

    options:
    MSG_Q_FIFO (0x00)       queue pended tasks in FIFO order
    MSG_Q_PRIORITY (0x01)   queue pended tasks in priority order
```

UNIVERSITY
OF OSLO

VxWorks Wind msg send message

- A task or ISR sends a message to a message queue with ***msgQSend*( )**. If no tasks are waiting for messages on that queue, the message is added to the queue's buffer of messages. If any tasks are already waiting for a message from that message queue, the message is immediately delivered to the first waiting task

```
STATUS msgQSend
   (
   MSG_Q_ID msgQId, /* message queue on which to send */
   char * buffer,   /* message to send */
   UINT nBytes,     /* length of message */
   int timeout,     /* ticks to wait */
   int priority     /* MSG_PRI_NORMAL or MSG_PRI_URGENT */
   )
```

VxWorks Wind msg send priorities

- The ***msgQSend*( )** function allows specification of the priority of the message as either

  normal (**MSG_PRI_NORMAL**) or

  urgent (**MSG_PRI_URGENT**).

  Normal priority messages are added to the tail of the list of queued messages, while urgent priority messages are added to the head of the list

VxWorks Wind receive message

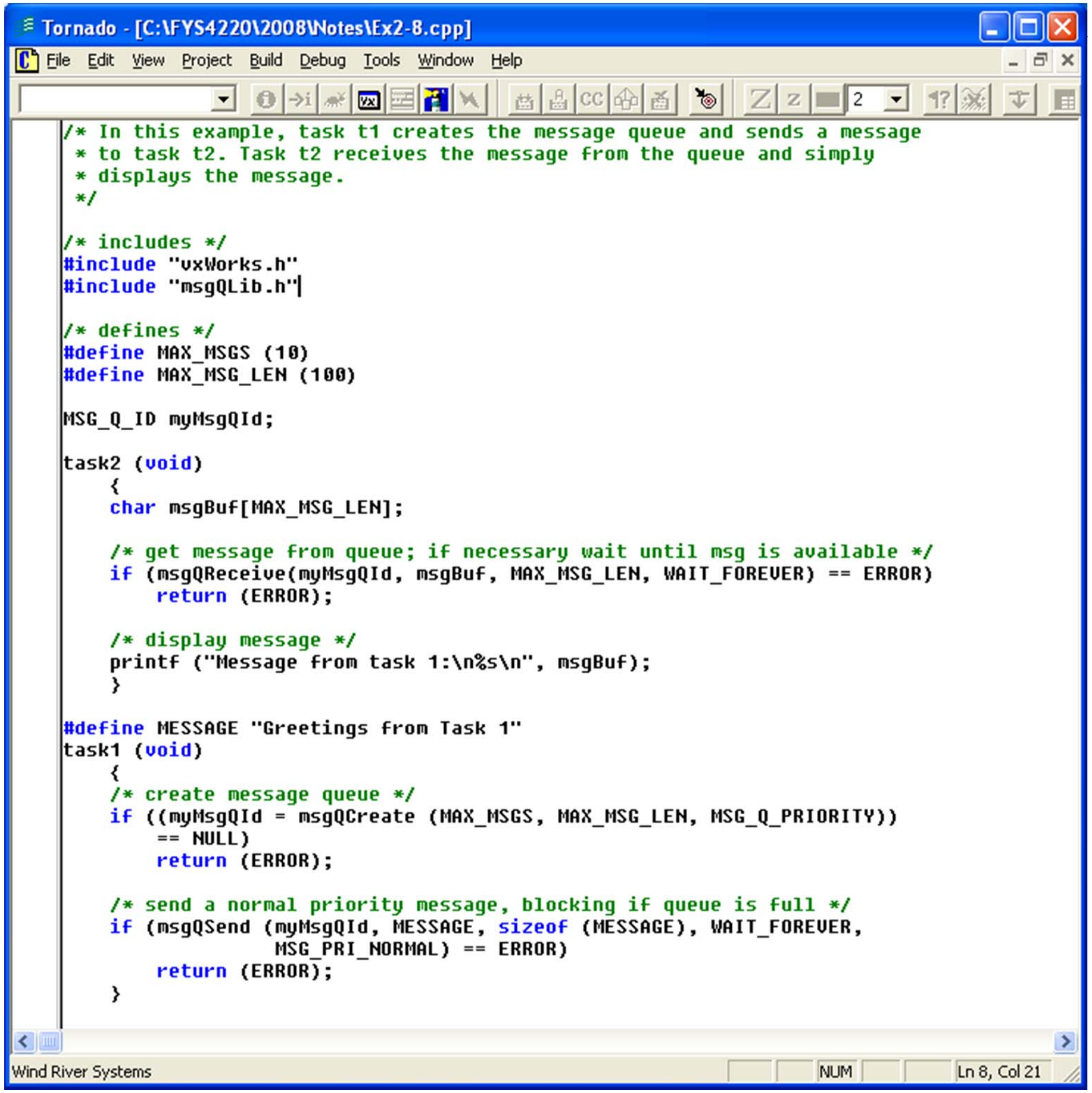
- A task receives a message from a message queue with ***msgQReceive*( )**. If messages are already available in the message queue's buffer, the first message is immediately dequeued and returned to the caller. If no messages are available, then the calling task blocks and is added to a queue of tasks waiting for messages. This queue of waiting tasks can be ordered either by task priority or FIFO, as specified in an option parameter when the queue is created

```
int msgQReceive
   (
   MSG_Q_ID msgQId, /* message queue from which to receive */
   char * buffer,   /* buffer to receive message */
   UINT maxNBytes,  /* length of buffer */
   int timeout      /* ticks to wait,
                       or NO_WAIT (0) or WAIT_FOREVER (-1) */
   )
```

VxWorks Wind msg timeouts

- Both ***msgQSend*( )** and ***msgQReceive*( )** take timeout parameters. When sending a message, the timeout specifies how many ticks to wait for buffer space to become available, if no space is available to queue the message. When receiving a message, the timeout specifies how many ticks to wait for a message to become available, if no message is immediately available. As with semaphores, the value of the timeout parameter can have the special values of **NO_WAIT** (0), meaning always return immediately, or **WAIT_FOREVER** (-1), meaning never time out the routine

Example 2.8 VxWorks Programmer's Guide

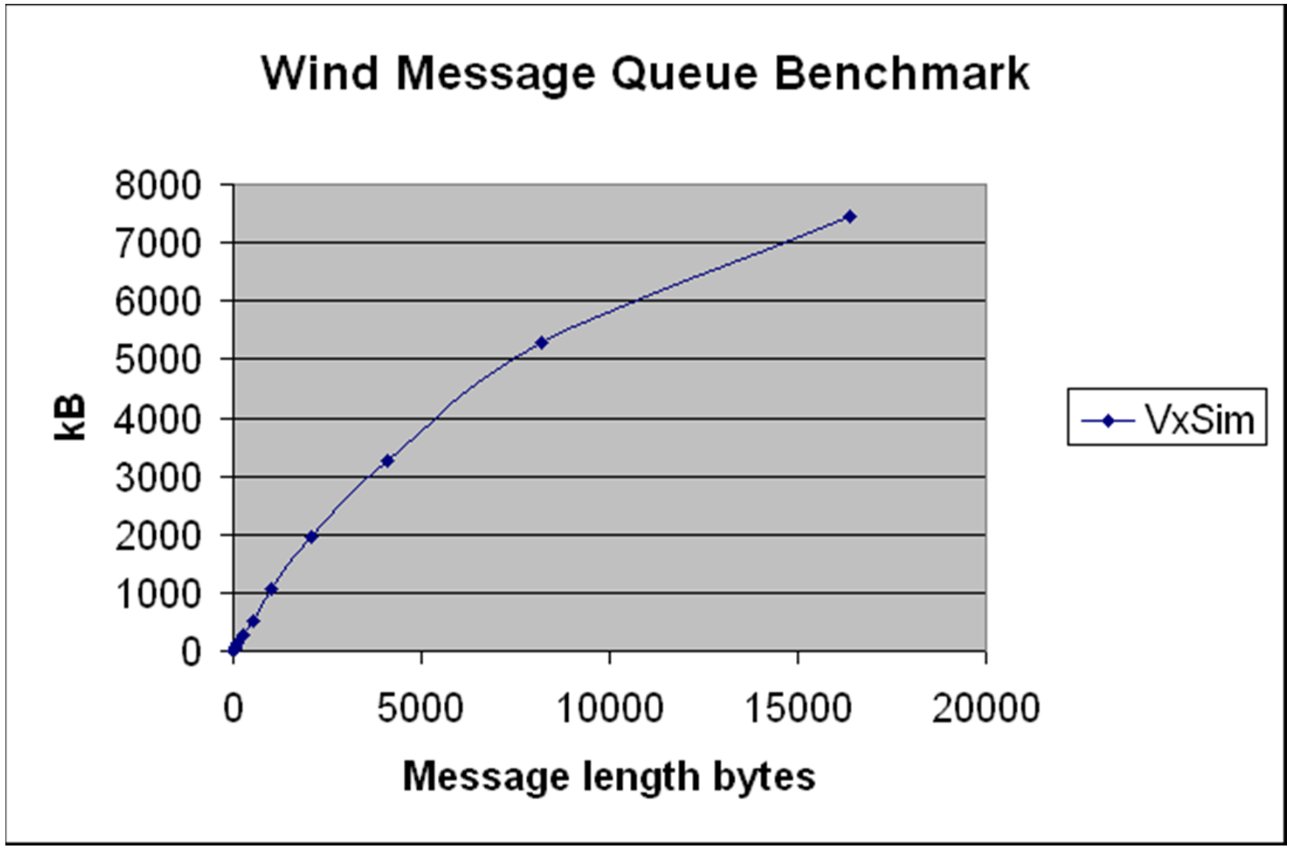

Tornado - [C:\FYS4220\2008\Notes\Ex2-8.cpp]
File Edit View Project Build Debug Tools Window Help
Wind River Systems
NUM
Ln 8, Col 21

```
/* In this example, task t1 creates the message queue and sends a message
 * to task t2. Task t2 receives the message from the queue and simply
 * displays the message.
 */

/* includes */
#include "vxWorks.h"
#include "msgQLib.h"

/* defines */
#define MAX_MSGS (10)
#define MAX_MSG_LEN (100)

MSG_Q_ID myMsgQId;

task2 (void)
    {
    char msgBuf[MAX_MSG_LEN];

    /* get message from queue; if necessary wait until msg is available */
    if (msgQReceive(myMsgQId, msgBuf, MAX_MSG_LEN, WAIT_FOREVER) == ERROR)
        return (ERROR);

    /* display message */
    printf ("Message from task 1:\n%s\n", msgBuf);
    }

#define MESSAGE "Greetings from Task 1"
task1 (void)
    {
    /* create message queue */
    if ((myMsgQId = msgQCreate (MAX_MSGS, MAX_MSG_LEN, MSG_Q_PRIORITY))
        == NULL)
        return (ERROR);

    /* send a normal priority message, blocking if queue is full */
    if (msgQSend (myMsgQId, MESSAGE, sizeof (MESSAGE), WAIT_FOREVER,
                  MSG_PRI_NORMAL) == ERROR)
        return (ERROR);
    }
```

UNIVERSITY OF OSLO

Wind message queue benchmark

- Below is the result of a simple message queue benchmark in kB/sec for VxSim on a DELL D420 laptop with 1.20 GHz Intel CPU



POSIX message queues *(text: ref B&W)*

- POSIX supports asynchronous, indirect message passing through the notion of message queues
- A message queue can have many readers and many writers
- Priority may be associated with the queue
- Intended for communication between processes (not threads)
- Message queues have attributes which indicate their maximum size, the size of each message, the number of messages currently queued etc.
- An attribute object is used to set the queue attributes when the queue is created

POSIX message queues

- Message queues are given a name when they are created
- To gain access to the queue, requires an `mq_open` name
- `mq_open` is used to both create and open an already existing queue (also `mq_close` and `mq_unlink`)
- Sending and receiving messages is done via `mq_send` and `mq_receive`
- Data is read/written from/to a character buffer.
- If the buffer is full or empty, the sending/receiving process is blocked unless the attribute O_NONBLOCK has been set for the queue (in which case an error return is given)
- If senders and receivers are waiting when a message queue becomes unblocked, it is not specified which one is woken up unless the priority scheduling option is specified

POSIX message queues

- A process can also indicate that a signal should be sent to it when an empty queue receives a message and there are no waiting receivers
- In this way, a process can continue executing whilst waiting for messages to arrive or one or more message queues. This concept, named Asynchronous I/O or non-blocking I/O, will be discussed in a later lecture.
- It is also possible for a process to wait for a signal to arrive; this allows the equivalent of selective waiting to be implemented
- If the process is multi-threaded, each thread is considered to be a potential sender/receiver in its own right

VxWorks POSIX message queues

- These routines are similar to Wind message queues, except that POSIX message queues provide named queues and messages with a range of priorities. A process can also indicate that a signal should be sent to it when an empty queue receives a message and there are no waiting receivers. Signals will be covered in a later lecture.
- Before a set of tasks can communicate through a POSIX message queue, one of the tasks must create the message queue by calling ***mq_open*( )** with the **O_CREAT** flag set. Once a message queue is created, other tasks can open that queue by name to send and receive messages on it. Only the first task opens the queue with the **O_CREAT** flag; subsequent tasks can open the queue for receiving only (**O_RDONLY**), sending only (**O_WRONLY**), or both sending and receiving (**O_RDWR**)

VxWorks POSIX mq routines

- ***mqPxLibInit*( )** Initialize the POSIX message queue library (non-POSIX)

  ***mq_open*( )** Open a message queue.

  ***mq_close*( )** Close a message queue.

  ***mq_unlink*( )** Remove a message queue.

  ***mq_send*( )** Send a message to a queue.

  ***mq_receive*( )** Get a message from a queue.

  ***mq_notify*( )** Signal a task that a message is waiting on a queue.

  ***mq_setattr*( )** Set a queue attribute.

  ***mq_getattr*( )** Get a queue attribute.

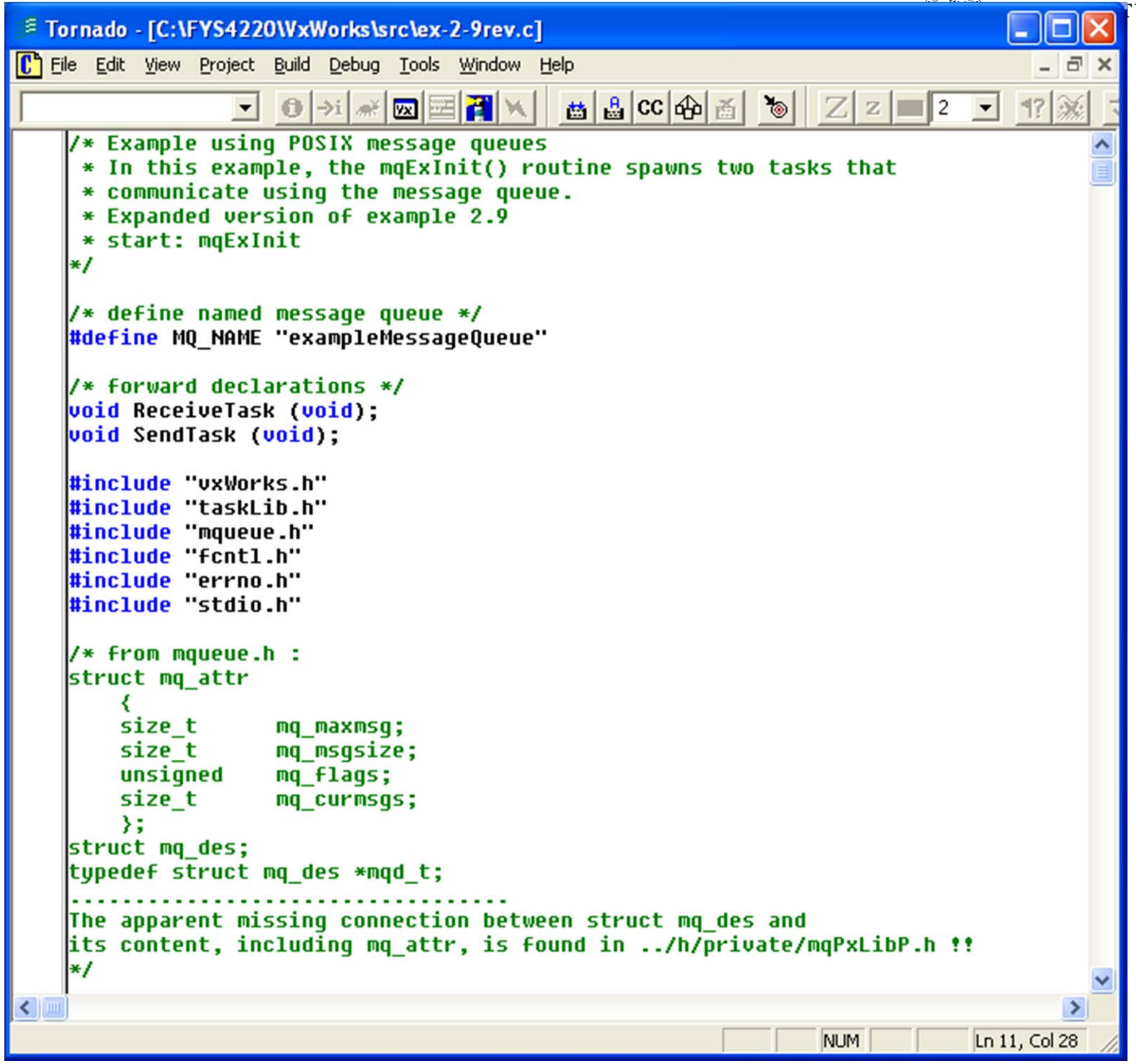
VxWorks POSIX mq_send( )

- To put messages on a queue, use ***mq_send*( )**. If a task attempts to put a message on the queue when the queue is full, the task blocks until some other task reads a message from the queue, making space available. To avoid blocking on ***mq_send*( )**, set **O_NONBLOCK** when you open the message queue. In that case, when the queue is full, ***mq_send*( )** returns -1 and sets **errno** to **EAGAIN** instead of pending, allowing you to try again or take other action as appropriate
- One of the arguments to ***mq_send*( )** specifies a message priority. Priorities range from 0 (lowest priority) to 31 (highest priority).

VxWorks POSIX mq_receive( )

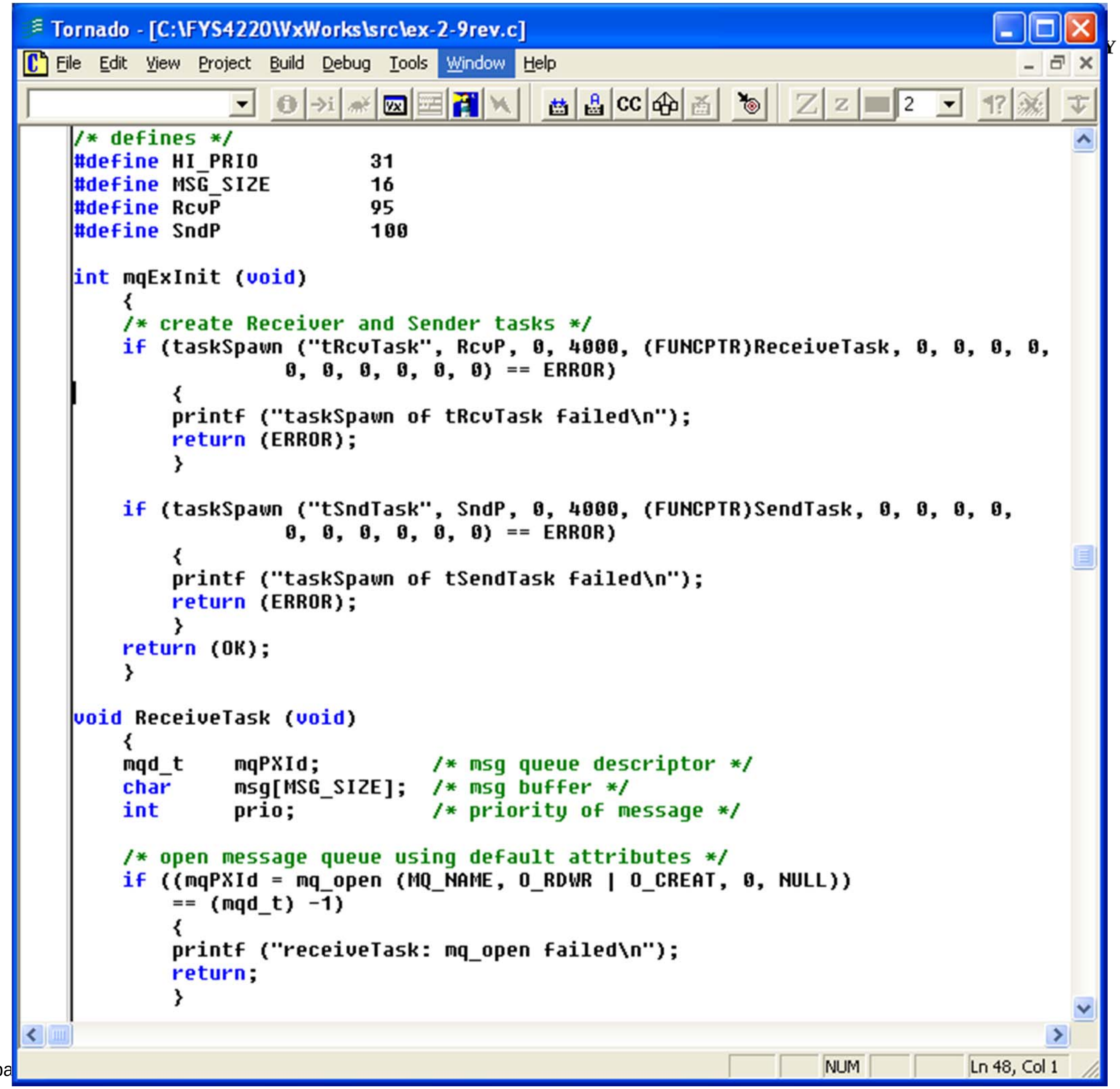
- When a task receives a message using ***mq_receive*( )**, the task receives the highest-priority message currently on the queue. Among multiple messages with the same priority, the first message placed on the queue is the first received (FIFO order). If the queue is empty, the task blocks until a message is placed on the queue. To avoid pending on ***mq_receive*( )**, open the message queue with **O_NONBLOCK**; in that case, when a task attempts to read from an empty queue, ***mq_receive*( )** returns -1 and sets **errno** to **EAGAIN**

Example 2.9 VxWorks Programmer's Guide – p1


Tornado - [C:\FYS4220\VxWorks\src\ex-2-9rev.c]
File Edit View Project Build Debug Tools Window Help
NUM
Ln 11, Col 28

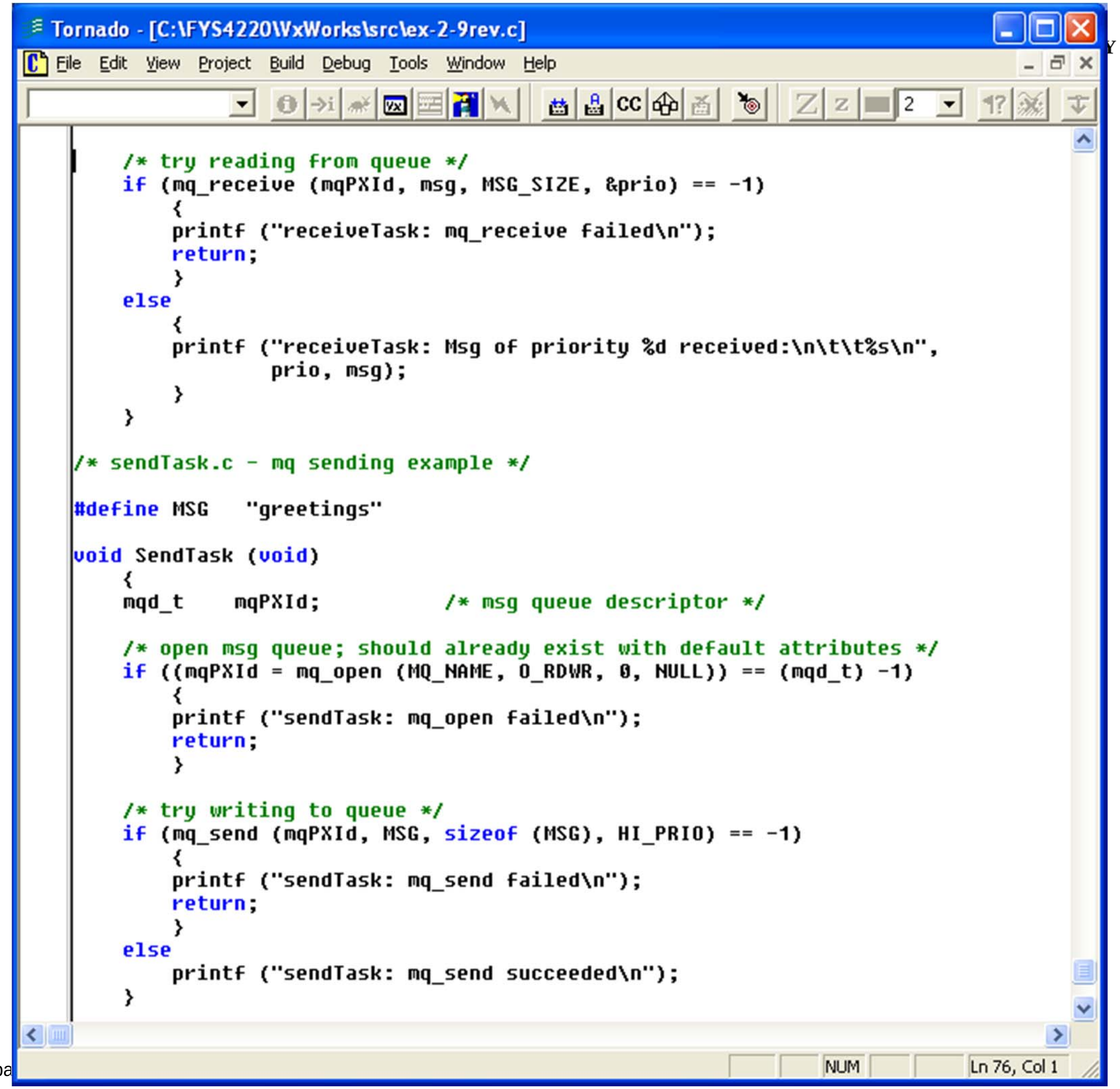
```c
/* Example using POSIX message queues
 * In this example, the mqExInit() routine spawns two tasks that
 * communicate using the message queue.
 * Expanded version of example 2.9
 * start: mqExInit
*/

/* define named message queue */
#define MQ_NAME "exampleMessageQueue"

/* forward declarations */
void ReceiveTask (void);
void SendTask (void);

#include "vxWorks.h"
#include "taskLib.h"
#include "mqueue.h"
#include "fcntl.h"
#include "errno.h"
#include "stdio.h"

/* from mqueue.h :
struct mq_attr
    {
    size_t      mq_maxmsg;
    size_t      mq_msgsize;
    unsigned    mq_flags;
    size_t      mq_curmsgs;
    };
struct mq_des;
typedef struct mq_des *mqd_t;
...................................
The apparent missing connection between struct mq_des and
its content, including mq_attr, is found in ../h/private/mqPxLibP.h !!
*/
```

Example 2.9 VxWorks Programmer's Guide – p2


Tornado - [C:\FYS4220\VxWorks\src\ex-2-9rev.c]
File Edit View Project Build Debug Tools Window Help
NUM
Ln 48, Col 1

```c
/* defines */
#define HI_PRIO         31
#define MSG_SIZE        16
#define RcvP            95
#define SndP            100

int mqExInit (void)
    {
    /* create Receiver and Sender tasks */
    if (taskSpawn ("tRcvTask", RcvP, 0, 4000, (FUNCPTR)ReceiveTask, 0, 0, 0, 0,
                    0, 0, 0, 0, 0, 0) == ERROR)
        {
        printf ("taskSpawn of tRcvTask failed\n");
        return (ERROR);
        }

    if (taskSpawn ("tSndTask", SndP, 0, 4000, (FUNCPTR)SendTask, 0, 0, 0, 0,
                    0, 0, 0, 0, 0, 0) == ERROR)
        {
        printf ("taskSpawn of tSendTask failed\n");
        return (ERROR);
        }
    return (OK);
    }

void ReceiveTask (void)
    {
    mqd_t    mqPXId;         /* msg queue descriptor */
    char     msg[MSG_SIZE];  /* msg buffer */
    int      prio;           /* priority of message */

    /* open message queue using default attributes */
    if ((mqPXId = mq_open (MQ_NAME, O_RDWR | O_CREAT, 0, NULL))
        == (mqd_t) -1)
        {
        printf ("receiveTask: mq_open failed\n");
        return;
        }
```

Example 2.9 VxWorks Programmer's Guide – p3


Tornado - [C:\FYS4220\VxWorks\src\ex-2-9rev.c]
File Edit View Project Build Debug Tools Window Help
NUM
Ln 76, Col 1

```
    /* try reading from queue */
    if (mq_receive (mqPXId, msg, MSG_SIZE, &prio) == -1)
        {
        printf ("receiveTask: mq_receive failed\n");
        return;
        }
    else
        {
        printf ("receiveTask: Msg of priority %d received:\n\t\t%s\n",
                prio, msg);
        }
    }

/* sendTask.c - mq sending example */

#define MSG   "greetings"

void SendTask (void)
    {
    mqd_t        mqPXId;          /* msg queue descriptor */

    /* open msg queue; should already exist with default attributes */
    if ((mqPXId = mq_open (MQ_NAME, O_RDWR, 0, NULL)) == (mqd_t) -1)
        {
        printf ("sendTask: mq_open failed\n");
        return;
        }

    /* try writing to queue */
    if (mq_send (mqPXId, MSG, sizeof (MSG), HI_PRIO) == -1)
        {
        printf ("sendTask: mq_send failed\n");
        return;
        }
    else
        printf ("sendTask: mq_send succeeded\n");
    }
```

VxWorks demo p 1 of 4

```
/* VxWorks Wind and POSIX message queues */


#include        "vxWorks.h"
#include        "msgQLib.h"
#include        "taskLib.h"
#include        "time.h"
#include        "stdio.h"

#include        "POSIXerrno.c“              /* Skaali routine */

/* Number of messages in queue, length of message */
#define         MSG_MAX                     4
#define         MSG_LEN                     16

MSG_Q_ID        myMsgQId;

STATUS          show_OBJ_error (int errn)
{
                if (errn == S_objLib_OBJ_ID_ERROR) printf("OBJ_ID_ERROR\n");
                if (errn == S_objLib_OBJ_UNAVAILABLE) printf("OBJ_UNAVAILABLE\n");
                if (errn == S_objLib_OBJ_DELETED) printf("OBJ_DELETED\n");
                if (errn == S_objLib_OBJ_TIMEOUT) printf("OBJ_TIMEOUT\n");
                if (errn == S_objLib_OBJ_NO_METHOD) printf("OBJ_NO_METHOD\n");
                return (OK);
}
```

VxWorks demo p 2 of 4

```
/* --- WIND message queue example --- */

STATUS          msqWIND (void)
{
                int msgl, loop;
                char            msgBuf[MSG_LEN];
                msgl = MSG_LEN;                                     /* max bytes in a message */

                /* create message queue */
                if ((myMsgQId = msgQCreate (MSG_MAX, MSG_LEN, MSG_Q_PRIORITY)) == NULL)
                                return (ERROR);

                /* fill message queue */
                for (loop = 0; loop<MSG_MAX; loop++) {
                   if (msgQSend (myMsgQId, msgBuf, msgl,
                                                WAIT_FOREVER, MSG_PRI_NORMAL) == ERROR)
                                                                return (errno);
                }
                printf("fill message queue OK, try one more message in NO_WAIT mode\n");
                if (msgQSend (myMsgQId, msgBuf, msgl,
                                                NO_WAIT, MSG_PRI_NORMAL) == ERROR) {
                                show_OBJ_error (errno);
                                return (errno);
                }

                msgQDelete (myMsgQId);
                return (OK);
}
```

VxWorks demo p 3 of 4

```
/* --- POSIX message queue example --- */
#include            "mqueue.h"
#include            "fcntl.h"
#define             MSQ_NAME        "POSIXmsq"
#define             MODE                            0
#define             MSGPRI                          31

STATUS msqPOSIX()
{
                    mqd_t mq_descr;                 /* msg queue descriptor */
                    mode_t mode=(mode_t)MODE;       /* mode, dummy */
                    struct mq_attr ma;              /* queue attributes */
                    int loop;
                    char  msgBuf[MSG_LEN];

                    /* set the required message queue attributes */
                    ma.mq_flags = 0;
                    ma.mq_flags  = O_NONBLOCK;      /* Immediate  return */
                    ma.mq_maxmsg = MSG_MAX;
                    ma.mq_msgsize = MSG_LEN;        /* max bytes in a message */

                    /* Create and open message queues */
                    if ((mq_descr = mq_open (MSQ_NAME, O_RDWR|O_CREAT, mode, &ma))
                                    == (mqd_t)-1) {
                                    return(errno);
                    }
                    /* fill message queue */
                    for (loop = 0; loop<MSG_MAX; loop++)
                      if (mq_send (mq_descr, msgBuf, ma.mq_msgsize, MSGPRI) != OK) {
                                    printf("\nmq_send error %d", errno);
                                    mq_unlink (MSQ_NAME);
                    }

                    printf("fill message queue OK, try one more message in NONBLOCK mode\n");
                    if (mq_send (mq_descr, msgBuf, ma.mq_msgsize, MSGPRI) != OK) {
                                    printPOSIXerrorcode (errno);
                                    mq_unlink (MSQ_NAME);
                                    return (errno);
                    }
                    return (OK);
}
```

VxWorks demo p 4 of 4

Run the message queue demo tasks:

```
-> msqPOSIX
fill message queue OK, try one more message in NONBLOCK mode
POSIX error EAGAIN - No more processes
value = 2 = 0x2

-> msqWIND
fill message queue OK, try one more message in NO_WAIT mode
OBJ_UNAVAILABLE
value = 3997698 = 0x3d0002
->
```

Note different error return codes from Wind and POSIX *send-message* when message buffer is full!

Summary of VxWorks message queue features

Message Queue Feature Comparison

Feature	Wind Message Queues	POSIX Message Queues
Message Priority Levels	1	32
Blocked Task Queues	FIFO or priority-based	Priority-based
Receive with Timeout	Optional	Not available
Task Notification	Not available	Optional (one task)
Close/Unlink Semantics	No	Yes

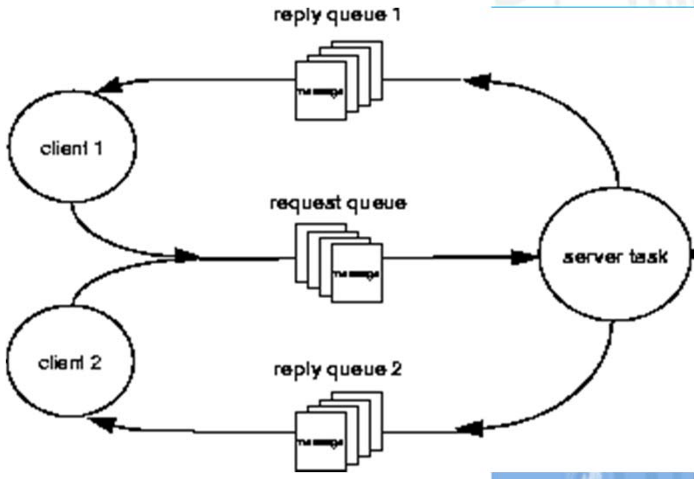
Another important feature of POSIX message queues is, of course, portability: if you are migrating to VxWorks from another 1003.1b-compliant system, using POSIX message queues enables you to leave that part of the code unchanged, reducing the porting effort.

UNIVERSITY OF OSLO

Servers and Clients with Message Queues

Real-time systems are often structured using a *client-server* model of tasks. In this model, server tasks accept requests from client tasks to perform some service, and usually return a reply. The requests and replies are usually made in the form of intertask –(process) messages. In VxWorks, message queues or pipes are a natural way to implement this.

For example, client-server communications might be implemented as shown in figure. Each server task creates a message queue to receive request messages from clients. Each client task creates a message queue to receive reply messages from servers. Each request message includes a field containing the **msgQId** of the client's reply message queue. A server task's "main loop" consists of reading request messages from its request message queue, performing the request, and sending a reply to the client's reply message queue.



Other InterProcessCommunication facilities

- Pipes- VxWorks supports pipe devices
  - *status* = pipeDevCreate ("*/pipe/name*", *max_msgs*, *max_length*);
  - Tasks can use standard I/O routines to open, read and write pipes, and invoke *ioctl* routines
- Sockets
  - A socket is an endpoint for communications between tasks; data is sent from one socket to another.
  - VxWorks supports the Internet protocols TCP and UDP
- Remote Procedure calls
  - Remote Procedure Calls (RPC) is a facility that allows a process on one machine to call a procedure that is executed by another process on either the same machine or a remote machine. Internally, RPC uses sockets as the underlying communication mechanism. Thus with RPC, VxWorks tasks and host system processes can invoke routines that execute on other VxWorks or host machines, in any combination.

- Some IPC methods will be discussed further in the lecture on Real-Time facilities and I/O. IPC is a key issue in Computer science, and for those who are interested, a few zillion papers are available on this subject