

proc

Real-Time Kernel

Reference Guide

nilsen elektronikk as

Jan Erik Nilsen

May 18. 2000

nilsen elektronikk as

Gml. Drammensvei 12B

N-1369 Stabekk

Norway

Tel: +47 67 58 31 62

Fax: +47 67 58 97 61

<http://www.nilsenelektronikk.no>

TABLE OF CONTENTS

LICENSE AGREEMENT / DISCLAIMER.....	4
INTRODUCTION.....	5
A PROCESS.....	5
COMMUNICATION BETWEEN PROCESSES	6
TIME CONTROL.....	6
THE PROCESS DESCRIPTOR (PID).....	7
PUBLIC VARIABLES	8
CRITICAL REGIONS.....	8
CONTEXT SWITCHING	8
EXTENDED CONTEX SWITCHING.....	9
SEMAPHORES	10
PRE-EMPTIVE RESCHEDULING / INTERRUPT HANDLING	11
DISABLE / ENABLE INTERRUPT	11
TARGET SPECIFIC FUNCTIONS	11
INTERRUPT FUNCTIONS	12
PC / 80x86 HW INTERRUPTS	13
procInstallTimerInterrupt.....	13
procUnInstallTimerInterrupt	13
procInitInterruptFunction	13
procInstallInterruptFunction.....	13
procUnInstallInterruptFunction	14
procInterruptFunctionInstalled	14
THE REAL-TIME KERNEL - proc.c	15
procInitialize	15
procCreateProcess	15
procTerminate	15
procReschedule	15
procChangePriority	16
procGetUsedWSSize	16
procSemInit.....	16
procSemSignal	17
procSemSignalCareful.....	17
procSemWait.....	17
CHANNEL FUNCTIONS - chan.c.....	18

Introduction	18
chanInit.....	18
chanOutChar.....	18
chanOutNB	19
chanInChar	19
chanInNB.....	19
Channel functions for interrupt handles.....	19
chanIfOutChar	19
chanIfInChar.....	20
TIMER FUNCTIONS - timer.c.....	21
Introduction	21
timerInit.....	21
timerStart.....	21
timerStop	22
timerWait.....	22
timerCheckList.....	22
QUEUE FUNCTIONS - queue.c.....	23
Introduction	23
queueInit	23
queuePut	23
queueGet.....	24
Queue functions for interrupt handles.....	24
queueIfPut.....	24
queueIfGet.....	24
PACKET FUNCTIONS - packet.c.....	25
Introduction	25
packetInitPool	26
packetAlloc	26
packetFree.....	26
packetInPool	27
packetsUsed	27
packetsLeft.....	27
packetsUsedMax	27
packetPut	27
packetGet	28
packetTo	28
packetFrom	28
Packet functions for interrupt handles.....	29
packetIfPut.....	29
packetIfGet.....	29
EXAMPLES.....	30
Use of PACKETS	30
Use of SEMAPHORES	34
Use of CHANnels.....	37

LICENSE AGREEMENT / DISCLAIMER

Copyright © nilsen elektronikk as

The **proc** RTOS with documentation are properties of **nilsen elektronikk as**, Norway. The **proc** RTOS is free software; you can use it, redistribute it and/or modify it under the terms below. By using, changing or redistributing the software, you accept the conditions below:

1. You are not allowed to remove or modify this copyright notice and License paragraphs, even if parts of the software is used.
2. The improvements and/or extensions you make **shall** be available for the community under **this** license, source code included. Improvements or extensions, including adaptations to new architectures, **shall** be reported and transmitted to Nilsen Elektronikk AS.
3. You must cause the modified files to carry prominent notices stating that you changed the files, what you did and the date of changes.
4. You may **not** distribute this software under another license without explicit permission from Nilsen Elektronikk AS, Norway.
5. This software is free, and distributed in the hope that it will be useful, but **WITHOUT ANY WARRANTY**; without even the implied warranty of **MERCHANTABILITY** or **FITNESS FOR A PARTICULAR PURPOSE**. You **SHALL NOT** use this software unless you accept to carry all risk and cost of defects or limitations.

The software is provided «as is» and without any express or implied warranties, including, without limitation, the implied warranties of merchantability and fitness for a particular purpose. The software should not be used within Life Support Systems. Life Support Systems are equipment intended to support or sustain life and whose failure to perform properly used in accordance with instructions provided can be reasonably expected to result in significant personal injury or death.

INTRODUCTION

proc is a powerful and compact C-written real-time kernel for embedded systems. It makes it possible to use the techniques of concurrent programming, with a very small overhead both in time and space. **proc** is a library to link with the application program. A real-time program usually run within one single program, but message based multiprocessor systems are supported as well. Apart from two functions, **proc** is processor independent. These two functions are context switching and building of a stack frame when a process is created. The context switching is the only assembly written module.

Features:

- The number of processes is only limited by the available memory.
- Priorities.
- Co-operative, pre-emptive and event/interrupt driven scheduling.
- Time-slicing is possible.
- Semaphores.
- Byte channels (ring buffer with i/o semaphores).
- Pointer queues. Same as channels, but the element is a pointer.
- Message packets. A packet is passed using pointer queues.
- Timers. A timer can be started and stopped. At time-out, a process waiting for channel input is waked. The number of timers is only limited by the available memory.
- Full source code is available.

A PROCESS

A process is a C function with one or more parameters. As a minimum, the function has a pointer to its Process Descriptor (explained later). Optional integer parameters can be given. Optional parameters are defined in the process creating function, and often used to give identity to different processes running the same code.

At system initialisation, one process is running; the main process. By function calls to the real-time kernel, the main process can create and terminate child processes. These processes are running in parallel with the main process.

At creation, a process is given a Process Descriptor and a Work Space. The work space provides space for local parameters, variables and stack. All variables declared local to the process function are allocated on the work space. Functions called by the process uses stack space on the work space of the process function. I.e. several processes can run the same code without interfering with each other's data. The work space must provide space for interrupts as well.

Global variables, i.e. variables declared outside functions, are available for any process. Any process can call any function. Any re-entrant function can be executed in parallel.

Each process has a priority between 0 and a maximum value, which can be redefined. 0 is the highest priority. A process is not switched in until all processes with higher priority are blocked.

COMMUNICATION BETWEEN PROCESSES

Inter process communication serves two purposes; synchronisation and information exchange. Semaphores are used for resource control and synchronisation. Channels and packets are used for synchronisation and communication.

A semaphore is an integer variable that, apart from initialisation, can be accessed only through two functions: *wait* and *signal*. The *wait* function is blocked if and while the variable is 0. If the variable is higher than 0, the variable is decrement. The *signal* function increments the variable. Blocked processes, waiting for a semaphore, are linked into a list.

A channel is a ring buffer of a given size, with two semaphores. The element size is byte. The initial value of the out-semaphore is 0, and the value of the in-semaphore is the size of the buffer. A consuming process must *wait* for the out-semaphore and *signal* to the in-semaphore. A producing process must *wait* for the in-semaphore and *signal* to the out-semaphore. Non-blocking input/output functions are available as well, and intended for use in interrupt handlers.

Message handling is provided by pointer queues. Pointer queues are like channels, but the element size is pointer. First a packet is allocated from a semaphore-equipped pool. Then the packet is written and passed to another process via a pointer queue. The receiving process reads the content and gives the packet back to the pool. The message passing is very efficient due to the use of pointers. Several pools for different packet size are possible. In the rear of the manual, an example shows remote message passing.

TIME CONTROL

A timer is an object waiting in a list of timers. It is put into the list by the *start* function, and may be removed any time by the *stop* function. When it times out, the timer is removed from the list, and a given value is put into a channel. A process waiting for input from the channel, wakes up. Timers can provide time-out mechanisms or make a process sleep a given time.

THE PROCESS DESCRIPTOR (PID)

A process descriptor describes every process in the system. The process descriptor is defined in **proc.h** and has the following format:

```

/*
   PID struct
   -----*/
typedef struct pid {
    struct pid *next;    /* next pid on list          */
    char      *stack;    /* stack of process when not running */
    char      *ws;      /* work space ptr              */
    int       wssize;    /* work space size             */
    int       age;      /* awake age                   */
    int       sleep;    /* sleeping if != 0            */
    int       priority; /* priority. 0 is highest      */
    struct pid *sema;    /* ptr to next pid on semaphore list */
} PID;

```

- The **next** field in the pid points to the next pid on a circular list of all pids, running or sleeping.
- The **stack** field holds the stack pointer of a process that is not currently running. The continue address for the process is stored on the stack.
- The field **ws** is a pointer to the workspace area. The high address part of the workspace contains the process stack and parameters given to the process. The user is free to use the low area for local heap etc.
- The field **wssize** contains the workspace size in bytes.
- The **age** field is used for pre-emptive rescheduling. The **age** of the current pid is incremented by an assembly-written function called by the system timer interrupt handler. If the age of a process becomes equal to the global variable **proc_maxage**, the process is rescheduled.
- The **sleep** field is used for semaphores. If **sleep** is different from zero, the process will sleep at the first rescheduling, until **sleep** is cleared.
- The priority field contains the priority of the process. **0** is highest, and **PROC_PRISIZE - 1** is the lowest.
- The **sema** field points to the next pid on a semaphore list. The list acts like a FIFO, and the oldest semaphore is at the front of the list.

Never mess with the pid. Inconsistent data will blow up the system

PUBLIC VARIABLES

proc_curpid	(PID *) is a pointer to the pid of the currently running process. Should not be altered.
proc_nodisp	(char) disallows rescheduling if not zero. Intended for critical regions.
proc_maxage	(int) gives the maximum age of a running process until the process is rescheduled 'by force' (if not inside a critical region). If the value of proc_maxage is -1, the system is non-pre-emptive.
proc_pattern	(char) The workspace is initialised with a byte value when the process is created. The default value is 0. proc_pattern may be given any other value. procGetUsedWSSize() uses proc_pattern to find how much workspace is used. If you want to change proc_pattern , do it before procInitialize() is called.
proc_pritab	(PID *proc_pritab[PROC_PRISIZE]) is an array of PID pointers, one per priority. Explained later. Should not be altered.
proc_at_next	(void (*proc_at_next)(void)) Pointer to a function to be called after rescheduling if the pointer is not NULL. Used to implement extensions to the context switching. Explained later.

CRITICAL REGIONS

The interrupt rescheduler will only perform its task if the public variable **proc_nodisp** is zero. It is thus possible for a process to increment **proc_nodisp** before entering a critical region where dispatches would be fatal. When the region is exited, **proc_nodisp** must be decrement again. **proc_nodisp** should not be set to a constant value, just incremented and decrement. Some microprocessor architectures are not able to do this operation atomically. In these cases, functions are used.

The macros **BEGIN_CRITICAL_REGION()** and **END_CRITICAL_REGION()** increments and decrements **proc_nodisp**, respectively, in an atomic operation. For very small critical regions, the interrupt flag may be used.

CONTEXT SWITCHING

Under PUBLIC VARIABLES, **proc_curpid** and **proc_pritab** is described. **proc_pritab** is a table of PID pointers, one per priority. These pointers points to **PROC_PRISIZE** number of different single linked circular lists. Active as well as sleeping processes are all kept in the circular lists, the sleeping processes with the **sleep** attribute TRUE. **proc_pritab[0]** points to the list with the processes of the highest priority, and **proc_pritab[PROC_PRISIZE - 1]** points to the list with the processes of the lowest priority. If none processes at a given priority exists, the respective **proc_pritab** pointer is NULL.

proc_curpid is a pointer to the current process at the current priority level. This pointer is used by **procReschedule()** during context switching, which is carried out this way:

- **proc_nodisp** is incremented to flag critical region.
- The stack pointer of the current process is saved into **proc_curpid->stack**.
- The function **procAnyNextProcess** is called until it returns a non-zero value.
- In case of positive return value, the same process continues. Then **proc_nodisp** is decremented to flag end of critical region, process age is set to zero and return made.
- moves **proc_curpid** to the next run-able process.
- In case of negative return value, another process is ready to continue. Then:
 - The context is pushed.
 - If function pointer **proc_at_next** is not NULL, the function is called
 - The value from **proc_curpid->stack** is moved to the stack pointer.
- **proc_nodisp** is decremented to flag end of critical region and process age is set to zero.
- The context is popped and return made.

Two functions are provided for inserting a PID into the system, and removing a PID from the system. These functions are **procInsertProcess** and **procRemoveProcess**, respectively.

ProcInitialize and **procCreateProcess** use **procInsertProcess**, and **procRemoveProcess** is used by **procTerminate**. Both are used by **procChangePriority**, which changes the priority of a process by first removing it and then inserting it at the new priority level.

EXTENDED CONTEX SWITCHING

You may have pointers, variables etc. you want to change when the next process is switched in. In the following example, the Real-Time Kernel is used together with a screen manager with windows assigned to processes. Output is directed to the window pointed to by **currw**.

```
WINDOW *currw;      /* pointer to current window */
```

In our example, we have 8 windows:

```
WINDOW *wpa[8];    /* ptrs to window descriptors */
```

The 'main' process is the one, which calls **procInitialize**. The pid of the 'main' process has no pointer to work space nor given work space size. In this example, the 'main' process owns the 'main' (background) window, **wpa[0]**.

```
procInitialize(&pid[MAIN], 3);
```

The 7 processes is assigned to **wpa[1]... wpa[7]**. The index corresponds with an optional parameter **i** given to the processes. **i** serves two purposes:

- The process is given an identity parameter.
- The parameter is given to the process by the stack frame. The parameter can be found at the highest address in the work space.

```
for (i = 1; i < 8; i++)
  procCreateProcess(&pid[i], procp[i], 3,&(ws[i][0]), WSSIZ, 1, i);
```

The following function is assigned to the global function pointer **proc_at_next**.

```
proc_at_next = afterContexSwitching;

/*===== afterContexSwitching =====
*
* Purpose:      Point currw to the window which belongs to
*               the process switched to.
*               proc_curpid points to the pid of the process
*
*/
void afterContexSwitching(void)
{
    static int last = 0;
    int      i;

    wpa[last] = currw;
    if (proc_curpid->wssize == 0) i = 0;
    else i = *(int *) (proc_curpid->ws + proc_curpid->wssize - 2);
    if (i < 0 || i >= 8) i = 0;
    if (i != last) {
        windowCursorSave();
        currw = wpa[last = i];
        windowCursorRestore();
    }
}
```

The main process is recognised by the work space size, which is 0. Reading the optional parameter from the bottom of the work space identifies the other processes. The pointer to the current window is first saved at the 'last' position of the window array, then the pointer is loaded from the current position, and finally, 'last' is given the current index.

SEMAPHORES

Semaphore operations are implemented. A semaphore has a value and a pointer to a list of processes waiting for a semaphore.

```
typedef struct sem {
    struct pid *ph; /* head ptr - the next waiting process */
    struct pid *pt; /* tail ptr - the last waiting process */
    int value; /* semaphore value */
} SEMAPHORE;
```

PRE-EMPTIVE RESCHEDULING / INTERRUPT HANDLING

In some cases, it has been found convenient to use installation / uninstallation functions for the timer interrupt function for handling pre-emptive rescheduling. This is done in the 8086 implementation. See the section «**PC / 80x86 HW interrupts**»

In other implementations, however, **procRescheduleTimerInt** is an entry point for a cyclic timer interrupt. A pointer, **procTimerIntFuncPtr**, is provided for entering a function with user tasks which should be done every timer. If some kind of interrupt handler wants control given to a process waiting for an event from that handler, a jump to the entry point **procRescheduleInside** will do. A global variable **forcereschedule** (defined in the assembly written context switch module), is used to flag «reschedule-as-soon-as-possible» if non-zero.

Such matters are architecture dependent, and better explained in the assembly file containing the architecture specific stuff.

DISABLE / ENABLE INTERRUPT

Two functions are defined, usually as macros:

```
disable_interrupt();  
enable_interrupt;
```

These functions are used in some of the **proc** functions. Only functions for interrupt handles should be used in an interrupt function, otherwise the interrupt flag may be corrupted.

The **68k** processor has interrupt levels rather than a single interrupt flag. In this case,
`int sr;`

is used to save the interrupt level and must exist as a local variable. *proc.c* and *proc.h* explains how.

TARGET SPECIFIC FUNCTIONS

Most of the real-time kernel is target independent. With an exception for the installation of pre-emptive rescheduling, the function interface for the user is uniform. However, different processors has different stack frames, therefore, some code is target specific. The most used function, **procReschedule**, is assembly-written, and uses another C-written function, **procAnyNextProcess**, to find the pid of the next run-able process. The assembly-written functions are contained in a file named **procxx.asm**, where **xx** denotes the name of the target processor. When a process is created, a stack frame is build. This target specific function is found in a file named **stkfxx.c**. This file may contain important architecture specific information, and should be read carefully.

INTERRUPT FUNCTIONS

How interrupts are handled, depends on the given architecture. In some cases, an assembly file with interrupt handles, is enclosed. The file contains explanations as well, and should be carefully read. The example below is from the AVR.

```
/*  
;   The installed function must be a normal C-function returning  
;   an int. If the return value is non-zero, rescheduling will  
;   be made.  
;   The intvects are defined in io8515.h, iom103.h (supported by IAR)  
*/  
  
void ihInstallHandle(int (*intfunc)(void), int intvect);
```

When interrupt occurs, the handle pushes registers and calls the installed function. If the function returns 0, registers are popped and return from interrupt is made. Otherwise, a JMP is made to an entry point (**procRescheduleInside**) defined in the assembly written context switch code.

PC / 80x86 HW INTERRUPTS

The timer interrupt is used for pre-emptive rescheduling. In «**proc86.asm**», functions are defined for installing, uninstalling and handling timer interrupt:

procInstallTimerInterrupt

```
*
* Purpose:      Install the timerintfunc to be called each
*               time 8253 generates an interrupt.
*               timerintfunc provides pre-emptive rescheduling.
*
void procInstallTimerInterrupt(void)
```

procUnInstallTimerInterrupt

```
*
* Purpose:      Uninstall the timerintfunc.
*               To be done before leaving the program.
*
void procUnInstallTimerInterrupt(void)
```

The following functions, defined in «**iftab.asm**», may be used for installing, uninstalling and checking functions handles for hardware interrupts at vector 9..15. The installed function must be of type: `int func(void);`

When an interrupt handler is installed, the old vector value is saved in a buried table together with a pointer to the function to be called. Then the respective vector is pointed to a socket function. When an interrupt occurs, the respective socket function is entered. The registers are pushed, DS is loaded with DGROUP, and the installed function is called by the pointer. When the installed function returns, the global variable **forcereschedule** (defined in **proc86.asm**) is set TRUE if the return value is non-zero. Then a jump is made to the entry point **procRescheduleInside** (**proc86.asm**) and **forcereschedule** is checked. If zero, return from interrupt is done. If non-zero, rescheduling is done first.

procInitInterruptFunction

```
*
* Purpose:      Clears the whole interrupt table.
*               To be done first.
*
void procInitInterruptFunction(void)
```

procInstallInterruptFunction

```
*
* Purpose:      Installs a function to be called each
*               time an external interrupt happens.
*               The function is of type
*               int func(void);
*               returning non-zero if rescheduling is required.
*
*               An interrupt number must be specified. Range 1..15
*               3 INT 0Bh COM2-4
```

```
*          4 INT 0Ch  COM1-3
*          5 INT 0Dh  Printer Controller 2
*          7 INT 0Fh  Printer Controller 1
*          8 INT 70h  CMOS real-time clock interrupt
*          9 INT 71h  Replaces bus IRQ2
*         10 INT 72h  Not used, extention slot IRQ
*         11 INT 73h  Not used, extention slot IRQ
*         12 INT 74h  Not used, extention slot IRQ
*         15 INT 77h  Not used, extention slot IRQ
*
*          When an interrupt at the specified vector happens,
*          the registers are pushed, and the function is called.
*          If the function returns non-zero, a jmp is made to *
*          procRescheduleInside (proc86.asm)
*
*  Input:      interrupt number. Valid numbers listed above.
*             fptr  function
*  Output:     intfunctab
*
```

```
void procInstallInterruptFunction(int inum, int (*fptr)(void))
```

procUnInstallInterruptFunction

```
*
*  Purpose:    Un-installs the installed interrupt function and
*             re-installs the old function.
*             Then clears the table slot.
*             To be done before leaving the program.
*
*  Input:      interrupt number, see previous function
*
```

```
void procUnInstallInterruptFunction(int inum)
```

procInterruptFunctionInstalled

```
*
*  Purpose:    Returns TRUE if interrupt function given by
*             interrupt number is installed, otherwise FALSE.
*
*  Input:      interrupt number, see previous function
*  Return:     see above
*
```

```
int procInterruptFunctionInstalled(int inum)
```

THE REAL-TIME KERNEL - proc.c**procInitialize**

```

*
* Purpose:      Initializes process system, must be called
*               before anything else. Inserts the current ``main''
*               process into the system.
*               NB: If the priority value is out of range, system
*               will blow up.
*
* Input:        ptr to pid
*               priority of this process
* Output:       none
* Return:       none
*
void procInitialize(PID *pid, int priority)

```

procCreateProcess

```

*
* Purpose:      Initializes the pid and the stack frame,
*               Inserts the pid into the system.
*               NB: If the priority value is out of range, system
*               will blow up.
*
* Input:        see below
* Output:       none
* Return:       none
*
void procCreateProcess(
    PID *pid,                /* process descriptor      */
    void (*func)(),          /* process function        */
    int priority,           /* priority                 */
    char *ws,               /* work space (stack etc.) */
    int wssize,             /* work space size [Bytes] */
    int nparam,            /* # of parameters to process */
    ...)                   /* parameters to process   */

```

procTerminate

```

*
* Purpose:      Terminates i.e. destroys the given process.
*
*               NB: NEVER ATTEMPT TO TERMINATE THE LAST PROCESS.
*
* Input:        ptr to pid
* Output:       none
* Return:       see proc.h
*
int procTerminate(PID *pid)

```

procReschedule

```

*
* Purpose:      Finds the next process to start. The global
*               proc_curpid is changed to point to the next process
*               to start. This function is assembly implemented.

```

```
*          and calls the C-function procNextProcess()
*
*   Input:      none
*   Output:     proc_curpid
*   Return:     none
*
void procReschedule(void)
```

procChangePriority

```
*
*   Purpose:    Changes the priority of the given process.
*               The change works after the first rescheduling.
*               NB: If the priority value is out of range, system
*               will blow up.
*
*   Input:      ptr to pid
*               new priority
*   Output:     none
*   Return:     none
*
void procChangePriority(PID *pid, int priority)
```

procGetUsedWSSize

```
*
*   Purpose:    Returns the size of the work space which is used.
*               This is done by finding the largest continuous
*               block with pattern equal to that of proc_pattern.
*               The size of that block is subtracted from the
*               work space size and returned.
*
*   Input:      ptr to pid
*   Output:     none
*   Return:     see above
*
int procGetUsedWSSize(PID *pid)
```

procSemInit

```
*
*   Purpose:    Initializes a semaphore. Semaphores must
*               be initialized prior to use.
*
*   Input:      ptr to semaphore, initial value
*   Output:     none
*   Return:     none
*
void procSemInit(SEMAPHORE *sp, int value)
```

procSemSignal

```
*
* Purpose:      Increment semaphore value. If negative
*               or zero then take oldest process from
*               head of semaphore list and activate it.
*               NB: procSemSignal will never reschedule.
*
* Input:       ptr to semaphore
* Output:      none
* Return:      none
*
void procSemSignal(SEMAPHORE *sp)
```

procSemSignalCareful

```
*
* Purpose:      Same as procSemSignal, but interrupt is
*               NOT disabled during the processing.
*               Intended for use in interrupt handles.
*
* Input:       ptr to semaphore
* Output:      none
* Return:      Waiting process was made runnable: 1
*               Otherwise: 0
*
int procSemSignalCareful(SEMAPHORE *sp)
```

procSemWait

```
*
* Purpose:      Decrement semaphore value. If negative then
*               put current process on tail of semaphore list,
*               suspend process and reschedule
*
* Input:       ptr to semaphore
* Output:      none
* Return:      none
*
void procSemWait(SEMAPHORE *sp)
```

CHANNEL FUNCTIONS - chan.c**Introduction**

Channels are byte-wide, and can be used for passing characters, events or data of any type between processes. A receiving process will block until data is available, while a sending process will block when the channel FIFO is full.

The source code can be compiled with or without **USE_MALLOC** defined. If compiled without **USE_MALLOC** defined, the user is responsible of setting the FIFO pointer of the channel structure to point to an area of sufficient size. If compiled with **USE_MALLOC** defined, the FIFO pointer must either be **NULL**, in this case memory is allocated, or point to an area of sufficient size. Memory is allocated using the function `char *procMalloc(int size)`, which must be user-supplied if **USE_MALLOC** is defined.

The channel structure and type:

```
struct chan_type {
    SEMAPHORE    used;          /* semaphore */
    SEMAPHORE    left;         /* semaphore */
    int          ii,oi,size;    /* indexes, size */
    unsigned char *fifo;       /* queue fifo */
};

typedef struct chan_type    CHAN;
```

chanInit

```
*
* Purpose:      Initialize an character channel.
*               Must be done before a channel is used.
*               NB: Read the introduction above.
*
* Input:        ptr to channel
*               size of channel [bytes]
* Output:       none
* Return:       If OK: 1, if malloc problem: 0.
*
int chanInit(CHAN *cp, int size)
```

chanOutChar

```
*
* Purpose:      Put a byte into a channel.
*               If the channel is full, the function
*               will block until space is available.
*
* Input:        ptr to chan
*               byte
* Output:       none
* Return:       none
*
void chanOutChar(CHAN *cp, int byte)
```

chanOutNB

```
*
* Purpose:      Put a byte buffer of a given size
*               into a channel.
*               If the channel is full, the function
*               will block until space is available.
*
* Input:        ptr to chan
*               ptr to byte buffer
*               number of bytes to transmit
* Output:       none
* Return:       none
*
*
void chanOutNB(CHAN *cp, char *msg, int num)
```

chanInChar

```
*
* Purpose:      Block until a byte can be
*               received from a channel.
*
* Input:        ptr to chan
* Output:       none
* Return:       byte
*
*
int chanInChar(CHAN *cp)
```

chanInNB

```
*
* Purpose:      Block until a string of bytes can be
*               received from a channel.
*
* Input:        ptr to chan
*               ptr to byte buffer
*               number of bytes to receive
* Output:       byte buffer
* Return:       none
*
*
void chanInNB(CHAN *cp, char *msg, int num)
```

Channel functions for interrupt handles

Channels can be used for inter-process communication. For communication between an interrupt handler and a process, however, a function must never be allowed to block. Interrupt handlers should use the following functions:

chanIfOutChar

```
*
* Purpose:      If the channel is not full then
*               put a byte into the channel.
*               If the channel was empty, return 2.
*               If the channel was not full, return 1.
*               If the channel was full, return 0.
```

```
*
*           The interrupt is not enabled nor disabled during
*           the execution. The user is suggested to disable
*           the interrupt before use.
*
*           The function is intended for interrupt-driven
*           receive handles.
*
*   Input:   ptr to chan
*           byte
*   Output:  none
*   Return:  Full: 0, Not full: 1 Empty: 2.
*
int chanIfOutChar(CHAN *cp, int byte)
```

chanIfInChar

```
*
*   Purpose: If the channel is not empty then output a
*           byte from the channel to the user.
*           If the channel was empty, return 0.
*           If the channel was not full, return 1.
*           If the channel was full, return 2.
*
*           The interrupt is not enabled nor disabled during
*           the execution. The user is suggested to disable
*           the interrupt before use.
*
*           The function is intended for interrupt-driven
*           transmit handles.
*
*   Input:   ptr to chan
*           ptr to unsigned char
*   Output:  byte from the channel
*   Return:  Empty: 0, Not full: 1 Full: 2.
*
int chanIfInChar(CHAN *cp, unsigned char *byte)
```

TIMER FUNCTIONS - timer.c**Introduction**

A timer is a hidden object which after a given time will put a given byte value into a channel, and then disappear. A process waiting for channel input will then wake up. The given byte value can be used to distinguish the time-out event from other events which also may wake up the process. The number of active timers is only limited by the memory space. For more information, see the function descriptions below.

The channel pointer of the timer structure must point to an initialised CHAN struct before use.

The timer structure and type:

```
struct timer_type {          /* Event timer type */
    struct timer_type *next; /* ptr to next          */
    struct chan_type  *chan; /* ptr to CHANnel   */
    int               timer; /* timer value      */
    int               x;    /* signal value     */
};

typedef struct timer_type  TIMER;
```

timerInit

```
*
* Purpose:      To be done first, once and for all.
*
* Input:        none
* Output:       none
* Return:       none
*
*
void timerInit(void)
```

timerStart

```
*
* Purpose:      A timer is started, or re-started if
*               it already exists. The timer will
*               exist until it is deleted by timerStop
*               or until it has timed out. If it exists
*               when the time elapses, the function
*               checkEventTimerList(); will delete the
*               timer and send a value x to the target CHANnel.
*
*               The timers are elements of a simple linked
*               list. The first elements time is absolute,
*               and the next element(s) time is relative
*               to the current. At insertion, the time of
*               the current, and the next, element may be
*               adjusted. At deletion, the time of the
*               following element must be adjusted.
*               The function checkEventTimerList() will
*               decrement the first timer, and then remove
```


QUEUE FUNCTIONS - queue.c

Introduction

Queues are pointer-wide, and can be used for passing pointers between processes. A receiving process will block until data is available, while a sending process will block when the queue is full.

The source code can be compiled with or without `USE_MALLOC` defined. If compiled without `USE_MALLOC` defined, the user is responsible of setting the FIFO pointer of the queue structure to point to an area of sufficient size. If compiled with `USE_MALLOC` defined, the FIFO pointer must either be `NULL`, in this case memory is allocated, or point to an area of sufficient size.

The queue structure and type:

```
struct queue_type {          /* queue type */
    SEMAPHORE used;         /* semaphore */
    SEMAPHORE left;        /* semaphore */
    int      ii,oi,size;    /* indexes, size */
    char *   *fifo;        /* queue fifo */
};

typedef struct queue_type   QUEUE;
```

queueInit

```
*
* Purpose:      Initialize a queue of void pointers.
*               Must be done before a queue is used.
*
* Input:        ptr to queue, queue length (# of elements)
* Output:       none
* Return:       If OK: 1, if malloc problem: 0.
*
*
int queueInit(QUEUE *qp, int size)
```

queuePut

```
*
* Purpose:      Put a pointer into a queue
*               If the queue is full, the function
*               will block until space is available.
*
* Input:        ptr to queue, pointer x
* Output:       none
* Return:       none
*
*
void queuePut(QUEUE *qp, char *x)
```

queueGet

```

*
* Purpose:      Blocks until a pointer can be taken
*               from a queue.
*
* Input:       ptr to queue
* Output:      none
* Return:      pointer
*
*
char *queueGet(QUEUE *qp)

```

Queue functions for interrupt handles

Queues can be used for inter-process communication. For communication between an interrupt handler and a process, however, a function must never be allowed to block. Interrupt handlers should use the following functions:

queueIfPut

```

*
* Purpose:      If the queue is not full then
*               put a pointer into the queue.
*               If the queue was empty, return 2.
*               If the queue was not full, return 1.
*               If the queue was full, return 0.
*
*               The interrupt is not enabled nor disabled during
*               the execution. The user is suggested to disable
*               the interrupt before use.
*
*               The function is intended for interrupt-driven
*               receive handles.
*
* Input:       ptr to queue
*               ptr to x
* Output:      none
* Return:      Full: 0, Not full: 1 Empty: 2.
*
int queueIfPut(QUEUE *qp, char *x)

```

queueIfGet

```

*
* Purpose:      If the queue is not empty then output a
*               pointer from the queue to the user.
*               If the queue was empty, return 0.
*               If the queue was not full, return 1.
*               If the queue was full, return 2.
*
*               The interrupt is not enabled nor disabled during
*               the execution. The user is suggested to disable
*               the interrupt before use.
*               The function is intended for interrupt-driven
*               transmit handles.
*
* Input:       ptr to queue
*               ptr to pointer x
* Output:      pointer from the queue
* Return:      Empty: 0, Not full: 1 Full: 2.
*
int queueIfGet(QUEUE *qp, char **xp)

```

PACKET FUNCTIONS - packet.c

Introduction

A packet is a structure of a buffer with a given size, a length and a type parameter. Packets can contain any kind of data, and be sent between processes. Packets are passed through pointers using queue functions, and are thus efficient.

A packet pool is the home of free packets. A packet pool is organised as a single linked list of packets. Packets are allocated from the front of the list. Free packets are inserted at the front of the list.

The source code can be compiled with or without `USE_MALLOC` defined. If compiled without `USE_MALLOC` defined, the user must give the function `packetInitPool` a valid pointer to a pool buffer area of sufficient size. If compiled with `USE_MALLOC` defined, the pool buffer pointer must either be `NULL`, in this case memory is allocated, or point to an area of sufficient size.

The user can make as many packet pools she wants, and the packet size can be different. A packet contains a pointer to its home pool, and can thus access its packet handle. As all members of a pool handle are covered by functions, there is no need for the user to access the handle directly. A packet pool handle has the following struct:

```
struct packetpool {
    int      packetsize;      /* size (bytes) of a packet */
    int      packetnum;      /* number of packets      */
    int      packetmsgsize;  /* net size of a packet   */
    int      packetusedmax;  /* statistics: max pk used */
    struct packet *packetstart; /* ptr to the 1th packet */
    SEMAPHORE packetleft;    /* pool empty semaphore   */
};

typedef struct packetpool PACKETPOOL;
```

The packet structure and type is showed below. NB: The size of the data buffer is not limited by the size given in the struct. (The size, 2, is a trick to fool the compiler.) The size is given to the function `packetInitPool`.

```
struct packet {
    struct packet *next;      /* ptr to next           */
    struct packetpool *ppool; /* ptr to home pool     */
    int      kind;          /* packet kind          */
    int      length;       /* data length         */
    char     data[2];      /* data buffer         */
};

typedef struct packet PACKET;
```

Dont use `sizeof(PACKET)`, it makes no sense. The following definition can be found in `packet.h`:

```
#define PACKET_HEAD_SIZE() (sizeof(PACKET) - (2 * sizeof(char)))
```

The memory size, **PKPOOLSIZE**, sufficient for a pool containing **PKNUM** number of packets, each of net size **PKMSGSIZE** follows. NB: If **PKMSGSIZE** is an odd number, the compiler may need extra space for alignments.

```
PKPOOLSIZE = sizeof(PACKETPOOL) +
              PKNUM * (PKMSGSIZE + PACKET_HEAD_SIZE())
```

The following structure applies for the two functions **packetFrom** and **packetTo**. These functions implement packet fifo's using single linked list with end-pointer. In some cases it can be wise to put a packet into such an fifo and then use an event channel to inform a receiving process.

```
struct packet_dptrs {
    struct packet *front;      /* front ptr      */
    struct packet *back;      /* end ptr       */
};
```

```
typedef struct packet_dptrs PACKET_DPTRS;
```

packetInitPool

```
*
* Purpose:      To be done first.
*               An area of memory and its size is given.
*               The net size of a packet is given as well.
*               The first part of the pool memory is used for
*               a pool handle. The memory size (m) needed for a
*               n number of packets is:
*               m = sizeof(PACKETPOOL) +
*                   n * (PACKET_HEAD_SIZE() + msgsize)
*
* Input:        ptr to pool buffer
*               size of pool buffer      (bytes)
*               net size of data packet (bytes)
* Output:       none
* Return:       pointer to packet pool struct
PACKETPOOL *packetInitPool(char *ppool, int ppsize, int msgsize)
```

packetAlloc

```
*
* Purpose:      Alloc a Packet from a pool. If the
*               pool is empty, then block until a packet
*               is available.
*
* Input:        pointer to pool
* Output:       none
* Return:       ptr to Packet
*
PACKET *packetAlloc(PACKETPOOL *h) *
```

packetFree

```
*
* Purpose:      Free a Packet to the pool.
*
* Input:        ptr to Packet
* Output:       none
* Return:       none
*
void packetFree(PACKET *pp)
```

packetInPool

```
*
* Purpose:      If the packet is found in the home pool,
*               return(1) else return(0);
*
* Input:        ptr to Packet
* Output:       none
* Return:       Found? 1 else 0
*
*
int packetInPool(PACKET *pp)
```

packetsUsed

```
*
* Purpose:      Returns number of packets in use.
*
*
int packetsUsed(PACKETPOOL *h)
```

packetsLeft

```
*
* Purpose:      Returns number of packets left in a pool.
*
*
int packetsLeft(PACKETPOOL *h)
```

packetsUsedMax

```
*
* Purpose:      Returns the maximum number of packets which has
*               been in use. For statistics and control purpose.
*
*
int packetsUsedMax(PACKETPOOL *h)
```

packetPut

```
*
* Purpose:      A packet, allocated and then written to,
*               is put into a queue. If the queue is full,
*               this function blocks.
*
* Input:        queue ptr
*               packet ptr
* Output:       none
* Return:       none
*
void packetPut(Queue *qp, PACKET *pp) *
```

packetGet

```
*
* Purpose:      Return a packet from the packet queue.
*               If the queue is empty, this function blocks.
*
* Input:        queue ptr
* Output:       none
* Return:       packet ptr
*
PACKET *packetGet(Queue *qp)
```

packetTo

```
*
* Purpose:      A packet, allocated from the pool,
*               is put into the end of a linked list
*               using an end ptr.
*
* Input:        ptr to PACKET_DPTRS
*               packet ptr
* Output:       none
* Return:       none (never fails)
*
void packetTo(PACKET_DPTRS *dpp, PACKET *pp)
```

packetFrom

```
*
* Purpose:      A packet is removed from the front a linked list.
*               If the list was empty, NULL is returned.
*
* Input:        ptr to PACKET_DPTRS
* Output:       none
* Return:       ptr to packet. NULL if empty list.
*
PACKET *packetFrom(PACKET_DPTRS *dpp)
```

Packet functions for interrupt handles

Packet functions can be used for inter-process communication. For communication between an interrupt handler and a process, however, a function must never be allowed to block. Interrupt handlers should use the following functions:

packetIfPut

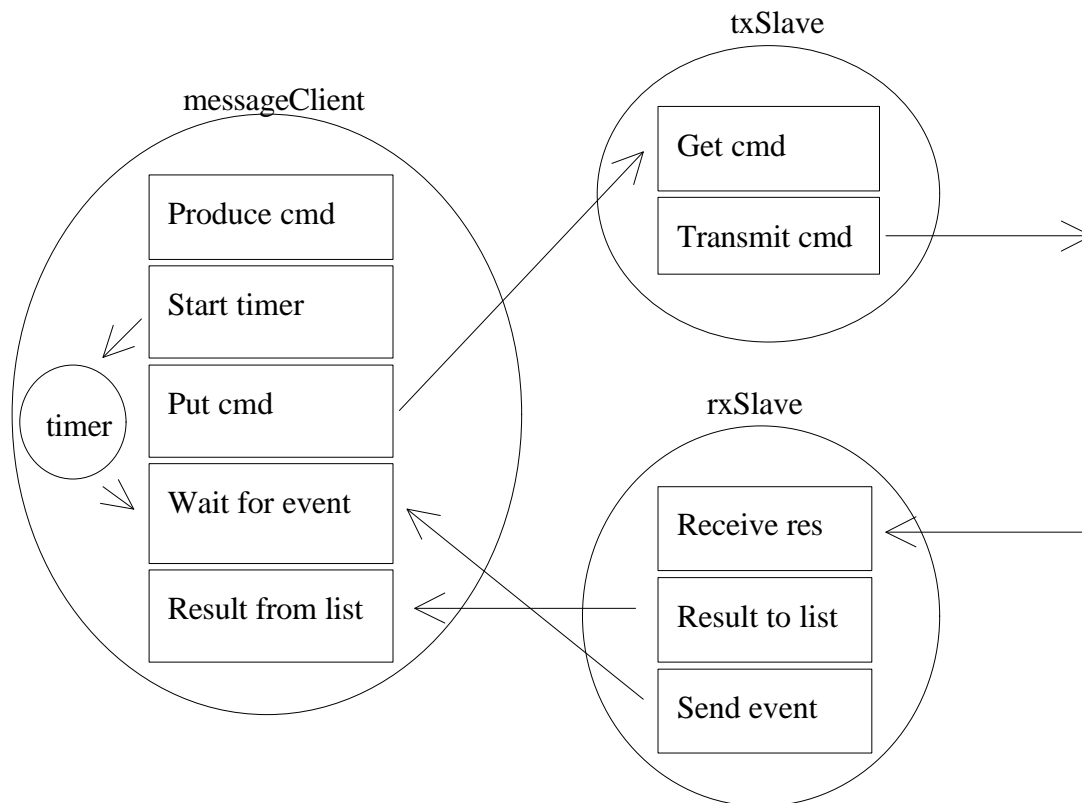
```
*
* Purpose:      If the packet queue is not full then
*               put a pointer into the queue.
*               If the queue was empty, return 2.
*               If the queue was not full, return 1.
*               If the queue was full, return 0.
*
*               The interrupt is not enabled nor disabled during
*               the execution. The user is suggested to disable
*               the interrupt before use.
*
*               The function is intended for interrupt-driven
*               receive handles.
*
* Input:        queue ptr
*               packet ptr
* Output:       none
* Return:       Full: 0, Not full: 1 Empty: 2.
*
int packetIfPut(Queue *qp, Packet *pp)
```

packetIfGet

```
*
* Purpose:      If the packet queue is not empty then output a
*               pointer from the queue to the user.
*               If the queue was empty, return 0.
*               If the queue was not full, return 1.
*               If the queue was full, return 2.
*
*               The interrupt is not enabled nor disabled during
*               the execution. The user is suggested to disable
*               the interrupt before use.
*
*               The function is intended for interrupt-driven
*               transmit handles.
*
* Input:        queue ptr
*               ptr to packet pointer
* Output:       packet pointer
* Return:       Empty: 0, Not full: 1 Full: 2.
*
int packetIfGet(Queue *qp, Packet **ppp)
```

EXAMPLES**Use of PACKETS**

The following example is about packet handling. The scenario is:



A message client sends a Remote Procedure Call (RPC) to a server. When the server gets a command, it computes and returns a result. The communication with the server is not failing safe. Packets may be lost or corrupted. The message client is blocked until a new command is produced. A timer is started and the command packet is sent to the transmit slave, which is blocked until the packet arrives. The transmit slave transmits the packet to the remote server. The receive slave is blocked until a result arrives from the remote slave. If the result is fine, the result packet is put into a FIFO and event = 0 is given the client. If the result is corrupted, event = 1 is given. If the answer is lost, the timer gives an event = 2 to the client. The client is blocked until an event arrives. If the event = 0, the result is taken from the FIFO, otherwise the event value is used to give an error message to the consumer of the result. If the event is different from 2, the timer is stopped.

Include files

```
#include <stdlib.h>
#include <stdio.h>
```

```
#include "proc.h"
#include "chan.h"
#include "queue.h"
#include "timer.h"
#include "packet.h"
```

The three processes

```
void messageClient(PID *pid);
void txSlave(PID *pid);
void rxSlave(PID *pid);
```

transmits a RPC to a server which makes a result

```
void transmitCommand(PACKET *pp);
```

receives a result from the server. Returns 0 if OK, 1 if CRC error

```
int receiveResult(PACKET *pp);
```

produces a RPC command message

```
void produceCommand(PACKET *pp);
```

consumes the result from the RPC server

```
void consumeResult(PACKET *pp);
```

```
#define PKPOOLSIZE 1024  size of packet pool in bytes
#define PKMSGSIZE 64    net size of a packet
#define WSSIZ      256   work space size for a process
#define MSGQSIZE  4     size (elements) of a fifo buffer
#define ECHNSIZE  4     size (bytes) of the event channel buffer
```

```
PACKETPOOL    *polp;           ptr to the packet pool object
char          pkpool[PKPOOLSIZE]; memory space for the packet pool
PID           mpid;           main Process ID
PID           cpid;           client ..
PID           txpid;          txSlave ..
PID           rxpid;          rxSlave ..
char          cws[WSSIZ];     client work space
char          txws[WSSIZ];     txSlave ..
char          rxws[WSSIZ];     rxSlave ..
CHAN          echan;          event channel object
unsigned char ecbuf[ECHNSIZE]; buffer space for event channel
char          *totxqbuf[MSGQSIZE]; buffer space for packet queue
QUEUE        totxq;           packet queue object
PACKET_DPTRS  reslist;        result list pointers
```

```

void main(void)
{
    procInitialize(&mpid, 4);           main process, priority = 4
    proc_maxage = 10;                 pre-emption after 10 ticks
    timerInit();                      initialize the timer system
    arrange packets out of the packet pool memory space
    polp = packetInitPool(&pkpool[0], PKPOOLSIZE, PKMSGSIZE);
    totxq.fifo = totxqbuf;            mount buffer space for the packet queue
    queueInit(&totxq, MSGQSIZE);      initialise the packet queue object
    echan.fifo = ecbuf;               mount buffer space for the event channel
    chanInit(&echan, ECHNSIZE);       initialise the event channel object
    procCreateProcess(&cpid, messageClient, 3, &cws[0], WSSIZ, 0);
    procCreateProcess(&txpid, txSlave, 3, &txws[0], WSSIZ, 0);
    procCreateProcess(&rxpid, rxSlave, 3, &rxws[0], WSSIZ, 0);
    for (;;) {
        procReschedule();             do nothing
    }
}

void messageClient(PID *pid)
{
    PACKET *pp;                       pointer to a packet
    int status;
    TIMER tim;                         timer object

    tim.chan = &echan;                the event channel is assigned to the timer
    for (;;) {
        pp = packetAlloc(polp);        allocate a new packet from packet pool
        produceCommand(pp);
        packetPut(&totxq, pp);         send command packet to txSlave
        timerStart(&tim, 50, 2);       start timer for 50 ticks, signal 2 at timeout
        status = chanInChar(&echan);   block until result or timeout
        switch (status) {
            case 0: timerStop(&tim);    no error nor time-out, stop active timer
                    pp = packetFrom(&reslist); get the result
                    consumeResult(pp);      consume the result
                    packetFree(pp);         recycle the packet
                    printf("Success\n");
                    break;
            case 1: timerStop(&tim);    CRC error, no timeout
                    printf("CRC error\n");
                    break;
            case 2: printf("Time-out\n");
        }
    }
}

```

```
void txSlave(PID *pid)
{
    PACKET *pp;

    for (;;) {
        pp = packetGet(&totxq);    block until command received
        transmitCommand(pp);      send the packet to the remote server
        packetFree(pp);           recycle the packet
    }
}

void rxSlave(PID *pid)
{
    PACKET *pp;
    int status = 0;

    for (;;) {
        if (status == 0)
            pp = packetAlloc(polp);    allocate a new packet from packet pool
        status = receiveResult(pp);    block until result message arrived
        if (status == 0)
            packetTo(&reslist, pp);    if no error
        packetTo(&reslist, pp);        put the result on the result list
        chanOutChar(&echan, status);    wake-up the client process
    }
}
```

Use of SEMAPHOREs

The first example is use of a semaphore for printer reservation. The initial semaphore value is 1. The **printerOpen** function uses **procSemWait**, which decrements the semaphore value. If the value is 0 before decrementing, the calling process is blocked until another process has released the printer. When the **printerClose** function is called, the **procSemSignal** function increments the semaphore value.

The second example is use of semaphore in a blink function. Suppose a green Light Emitting Diode is available. When the **blinkGreenLED** function is called, the LED is switched on for 55 ms, then off for 55 ms. If **blinkGreenLED** is called twice, the LED blinks twice. To avoid meaningless accumulation of blinks, the counter (semaphore) value is limited to 4.

```
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include "proc.h"
#include "timer.h"

void xProcess(PID *pid, int identity_number);
double expdistrand(double mean);

void printerInit(void);           functions for the printer example
void printerOpen(void);
void printerClose(void);
void printerWrite(char *str);

void blinkGreenLED(void);        functions for the blink example
void blinkProcess(PID *pid);
void greenLED(int on);          assumed to be external

#define WSSIZ          256

PID          mpid;              main Process ID
PID          blinkpid;         blinkProcess ID
PID          xpid[2];          xProcess ID, two pieces
char         blinkws[WSSIZ];   blinkProcess work space
char         xws[2][WSSIZ];    xProcess work space, two pieces

void main(void)
{
    procInitialize(&mpid, 4);
    proc_maxage = 10;
    timerInit();
    printerInit();
}
```

Two instances of xProcess is running, individual PID and work space is needed. The two instances are given an unique identity number as optional parameter.

```

procCreateProcess(&blinkpid, blinkProcess, 5, &blinkws[0],
    WSSIZ, 0);
procCreateProcess(&xpid[0], xProcess, 3, &xws[0][0],
    WSSIZ, 1, 0);
procCreateProcess(&xpid[1], xProcess, 3, &xws[1][0],
    WSSIZ, 1, 1);
for (;;) {
    procReschedule();    do nothing
}
}

```

```

void xProcess(PID *pid, int identity_number)
{
    int t;
    char buf[48];

    for (;;) {
        t = (int)expdistrand(250.0);    random negexp distributed number
        timerWait(t);
        printerOpen();
        sprintf(&buf[0], "Process %d printing 1th line\n",
            identity_number);
        printerWrite(&buf[0]);    the two printed lines are never interrupted
        sprintf(&buf[0], "Process %d printing 2nd line\n",
            identity_number);
        printerWrite(&buf[0]);
        blinkGreenLED();    make one blink
        printerClose();
    }
}

```

Assume system clock frequency of 18.2 ticks per sec. The input parameter is mean time in second until an event. The negexp distribution is known from discrete event systems.

```

double expdistrand(double mean)
{
    double u = (double)rand() / 32768.0;
    return(-mean * 18.2 * log(1.0 - u));
}

```

```

/* ----- printer.c ----- */

```

```

SEMAPHORE    prnsem;

```

```

void printerInit(void)
{

```

```
    procSemInit(&prnsem, 1);
/*  printerHardwareInit(); */
}

void printerOpen(void)
{
    procSemWait(&prnsem);
}

void printerClose(void)
{
    procSemSignal(&prnsem);
}

void printerWrite(char *str)
{
    /* send string to a printer process using channel or packet */
}

/* ----- blink.c ----- */

SEMAPHORE blinksem;

void blinkGreenLED(void)
{
    procSemSignal(&blinksem);
}

void blinkProcess(PID *pid)
{
    procSemInit(&blinksem, 0);
    for (;;) {
        greenLED(0);           switch the loght off
        timerWait(1);         wait 55 ms
        procSemWait(&blinksem); block until semaphore is signaled
        greenLED(1);          switch the loght on
        timerWait(1);         wait 55 ms
        if (blinksem.value > 4) blinksem.value = 4;    limit counter
    }
}
```

Use of CHANnels

The following example shows the use of channels in an UART module. The **comGetch** blocks until the interrupt handler **comHandler** has put a byte into the receive channel. If the receive channel is full, the **comHandle** must not block, but set an error flag. The **comPutch** function may block if the transmit channel is full. When a transmitter-empty interrupt occurs, the transmit channel is checked. If a byte is available, it is put into the transmitter, else the transmitter is disabled.

For this reason, the non-blocking channel functions **chanIfOutChar** and **chanIfInChar** returns:

chanIfOutChar returns if the channel was:

empty	2	Perhaps one high priority process is blocked waiting for input. It could be right to reschedule now.
not full	1	There was space for the byte.
full	0	There was not space for the byte. The byte is lost. Flag error!

chanIfInChar returns if the channel was:

full	2	Perhaps one high priority process is blocked because the channel was full. It could be right to reschedule now.
not empty	1	There was remaining byte(s) in the channel.
empty	0	There was no remaining bytes in the channel. It could be right to disable the consuming device.

If the return value from either of these functions is 2, the interrupt program calling **comHandle** could make a jump to an entry point for rescheduling from interrupt.

```
#include <stdlib.h>
#include "proc.h"
#include "chan.h"

/* ----- comport.h ----- */

#define UART_STATUS      (*(unsigned char *)0xF020)
#define TX_EMPTY        0x80
#define RX_FULL         0x40
#define COM_RXQ_FULL    0x20
#define UART_ERRFLAG    0x07
#define UART_CMD        (*(unsigned char *)0xF021)
#define TX_ENABLE       0x01
#define UART_RXREG      (*(unsigned char *)0xF022)
#define UART_TXREG      (*(unsigned char *)0xF023)

#define COM_RX_FIFO_SIZE 128
#define COM_TX_FIFO_SIZE 128
```

```
void comInit(void);
int comKbhit(void);
int comGetch(void);
void comPutch(int ch);
int comHandler(void);

/* ----- comport.c ----- */

CHAN          comRXChan;
unsigned char comRXFifo[COM_RX_FIFO_SIZE];
CHAN          comTXChan;
unsigned char comTXFifo[COM_TX_FIFO_SIZE];
char          comError;

void comInit(void)
{
    comRXChan.fifo = &comRXFifo[0];
    chanInit(&comRXChan, COM_RX_FIFO_SIZE);
    comTXChan.fifo = &comTXFifo[0];
    chanInit(&comTXChan, COM_TX_FIFO_SIZE);
}

int comKbhit(void)          returns number of bytes waiting in the channel
{
    return(comRXChan.used.value);
}

int comGetch(void)          blocks until a byte is available
{
    return(chanInChar(&comRXChan));
}

void comPutch(int ch)      blocks until the channel is not full
{
    chanOutChar(&comTXChan, ch);
    disable_interrupt();
    UART_CMD |= TX_ENABLE;    enable the UART transmitter
    enable_interrupt();
}

int comHandler(void)       called at UART interrupt
{
    unsigned char ch = UART_STATUS;
    int          r = 0;
}
```

```
disable_interrupt();
if (ch & RX_FULL) {
    comError |= (ch & UART_ERRFLAG);
    if ((r = chanIfOutChar(&comRXChan, UART_RXREG)) == 0)
        comError |= COM_RXQ_FULL;
}
else if (ch & TX_EMPTY) {
    if ((r = chanIfInChar(&comTXChan, &UART_TXREG)) == 0)
        UART_CMD &= ~TX_ENABLE;
}
return(r == 2);          returns 1 if rescheduling desirable, else 0
}
```