

nesos

Finite State Machine Operating System

Reference Guide

nilsen elektronikk as

Jan Erik Nilsen

Aug. 16. 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

1	<i>LICENSE AGREEMENT / DISCLAIMER</i>	4
2	<i>Introduction</i>	5
3	<i>What is a Finite State Machine Operating System</i>	5
3.1	Process	6
3.2	The state variable	6
3.3	Initialisation state	6
3.4	STATE_APPSEL state	7
3.5	Message forwarding	7
3.6	Message saving	7
3.7	Maximum number of processes	7
4	<i>Communication between Processes</i>	8
4.1	Packets	8
5	<i>Time Control</i>	9
6	<i>Fatal Error Handling</i>	9
7	<i>Message Monitoring</i>	9
8	<i>Global nesos variables</i>	10
8.1	nesos.c	10
8.2	packet.c	10
8.3	timer.c	11
9	<i>Alignment</i>	11
10	<i>Message Priority</i>	11
11	<i>API – nesos.c</i>	12
11.1	nesosInit	12
11.2	nesosSetIMask	12
11.3	nesosSetOMask	12
11.4	nesosInstallProcess	12
11.5	nesosRun	13
11.6	nesosCyclic	13
11.7	packetPut	13
11.8	packetPutX	13
11.9	packetPutHiPri	13
11.10	packetPutHiPriX	14

11.11	nesosInPacketForwarding	14
11.12	nesosSaveInPacket	14
11.13	nesosSetSavedLimit	14
11.14	nesosInstallPutChar	15
11.15	nesosPanic	15
11.16	nesosEvent	15
11.17	nesosExeFunctionTable	16
12	<i>API - packet.c</i>	17
12.1	packetMakePool	17
12.2	packetAlloc	18
12.3	packetFree	18
12.4	packetUsedMax	18
12.5	packetNetsize	18
12.6	packetNumberOf	18
13	<i>API - timer.c</i>	19
13.1	Introduction	19
13.2	timerStart	19
13.3	timerStop	19
14	<i>An example</i>	20

1 LICENSE AGREEMENT / DISCLAIMER

Copyright © nilsen elektronikk as

The **nesos** (= Nilsen Elektronikk State machine Operating System) with documentation are properties of **nilsen elektronikk as**, Norway. The **nesos** OS 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 are 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.

2 Introduction

nesos (= Nilsen Elektronikk finite State machine Operating System) is a powerful and compact C-written FSMOS (= Finite State Machine Operating System) for embedded purposes. It makes it possible to use the techniques of concurrent programming, with a very small overhead both in time and space. **nesos** consists of a few C files that can be included with the application files or collected in a library and linked with the application program. **nesos** is completely processor independent. It can be used with any ANCI compiler. There is no need for assembly files or architecture dependent features.

3 What is a Finite State Machine Operating System

A system based on **nesos** consists of **nesos** and some processes. A process is a function that is called when a message packet addressed to that process is available. Always when a process function is called, there is an input packet and during execution, the process makes actions according to the contents of that packet (NB: See section Initialisation state). Changing the state of the process may be a part of an action. Every process has its own state variable and a unique process ID. The process ID is a number, used to route a message packet from a source to a destination process. A process may send message packets whenever it wants. The packets are used for synchronisation and information exchange.

Why use FSMOSs instead of pre-emptive Real-Time Operating Systems?

In some cases we prefer pre-emptive RTOS, but FSMOS has some interesting properties:

- FSMOSs can be made completely processor independent. I.e. completely in C with no architecture dependent stuff.
- All processes share work space (space for stack, parameters and local variables)
- As apposed to a pre-emptive RTOS, there are no concerns about atomic operations during process execution. A process function call is always completed until the next process function is called.
- For the same reasons, compilers making non-reentrant code can be used.
- There are fewer chances that a program based on FSMOS shall mess up the stack in case of bugs. A FSMOS program is easier to debug. Since no "clever" tricks are made to the stack space, debuggers usually works fine.
- There is a reason why FSMOS is widely used in LARGE Telecom systems. FSMOS based systems can be specified in SDL (Specification and Description Language), a CCITT recommendation.
- There are good tools and traditions for FSMOS based systems.

There are many books about SDL. One is:

"An Introduction to the CCITT SDL", Anders Rockström, Televerket Stockholm 1985.
ISBN 91-7810-321-5

3.1 Process

A process is a C function of type

```
void f(void);
```

A process function shall never block, it shall complete as soon as possible and never wait for input because it already has the necessary input. The reason why it executes is the input packet.

A process function is associated with a `PROCESS_DESCRIPTOR`. <nesos.h>

```
typedef struct process_descriptor {
    int id; /* Process ID */
    char name[32]; /* Process name */
    void (*pp)(void); /* Process function ptr */
    int state; /* Process state */
    int entrystate; /* State at entry */
    int forwardflag; /* don't free inpp */
    PACKET *inpp; /* Input PACKET ptr */
    PACKET_DPTRS_LIM savedpp; /* Saved input packets */
} PROCESS_DESCRIPTOR;
```

The id is the unique identity number of a process. For debugging purposes, it may be given an ASCII name as well. A function pointer, pp, points to the process function. The process has a state variable included in the `PROCESS_DESCRIPTOR`. The current state is written to `entrystate` before the process function is called. A pointer to the input message packet is included as well. The input message packet is freed when the process function call returns, unless `forwardflag` is set. If the user has forwarded the input message packet to another user or to the save queue, then `forwardflag` is set.

3.2 The state variable

The state variable is a main issue in SDL. **nesos** uses the state variable both for controlling initialisation and execution. After a process is created by using

```
void nesosInstallProcess(PROCESS_DESCRIPTOR *pdp);
```

the state variable is assigned `STATE_INIT` (`= -1`) by **nesos**.

3.3 Initialisation state

All processes must have state `!= STATE_INIT`, until message packets are passed to any process. Processes with state `== STATE_INIT` are called, one by one, with `NULL` input `PACKET` pointer. Thus, the processes can make low level initialisations before they have to deal with messages.

When **nesos** has completed the initialisation call, and state `== STATE_INIT`, the process state is set to zero by **nesos**. When all process states are `!= STATE_INIT`, normal message passing starts.

3.4 STATE_APPSEL state

Rather than being a state, this state value can be used to restore the original state value. Before calling the application process function, **nesos** writes the current state to `entrystate`. If `state = STATE_APPSEL (= -2)` by the application code or `nesosExeFunctionTable` (See 11.17), the original state (`= entrystate`) is restored by **nesos**. The motivation is to provide adequate functionality for `nesosExeFunctionTable`.

3.5 Message forwarding

The input message packet can be forwarded to another user by updating the `processid.inpp->dst` accordingly (`processid` is the `PROCESS_DESCRIPTOR` of the a process), then calling

```
void nesosInPacketForwarding(void);
```

Finally, one of the following function calls must be made to transmit the packet:

```
packetPut(processid.inpp);  
packetPutX(processid.inpp);  
packetPutHiPri(processid.inpp);  
packetPutHiPriX(processid.inpp);
```

3.6 Message saving

If a particular input packet can not be processed in the current state, it can be saved for later. After state has been changed, the oldest message from the save queue appears as input message packet. I.e. saved messages have highest priority. To avoid too many packets from ending up in the save queue, a limit number can be set. The value 0 is infinite.

The two following functions are provided:

```
int nesosSaveInPacket(void);  
void nesosSetSavedLimit(unsigned int maxval);
```

3.7 Maximum number of processes

In `<nesos.h>`, the maximum number of processes can be defined:

```
#define MAX_PROCESSES 16
```

4 Communication between Processes

Inter process communication serves two purposes; synchronisation and information exchange. Packets are used for this purpose. An input packet is always available (except during initialisation). The information contained by the packet can be used for information transfer, decision-making, etc.

4.1 Packets

The packet consists of a Header and Data:

```
typedef struct packet {
    struct packet *next; /* ptr to next */
    int pindex; /* index to home pool */
    int src; /* source process ID */
    int dst; /* destination process ID */
    int signal; /* signal value/ID */
    unsigned int length; /* data length */
    unsigned int dmask; /* debug mask for tracing */
    char data[2]; /* data buffer */
} PACKET;
```

next	Packets are linked into a list while they are in the pool of free packers and while they are waiting to be popped and given to a destination. You should think and understand before using this one.
pindex	Used by packetFree to put a packet to the pool it belongs to. Never mess with this one.
src	The process id. of the Source process.
dst	The process id. of the Destination process.
signal	Packet type. For decision making purposes. Also a SDL issue.
length	Number of data bytes.
dmask	Debug Mask. Used for message monitoring. See section.
data	Data. Don't be confused by the [2]. This is for fooling the compiler only. There is no range checking in C, and the packet size is what you allocate.

Message passing is provided using pointers. First a packet is allocated from a pool. If the pool is empty, the user must either have a strategy for dealing with the situation or the pool must be increased. When a packet is allocated, **dst** and **signal** must be given values, By default, **dmask** and **length** are 0. **dmask** is useful during debugging, and may be assigned a value. If there are **data** to be transmitted, **length** must be given its size in bytes. The **src** is filled in by the packetPut function.

Important:

The application shall **not** free the input packet. The packet is freed by **nesos** when the process function returns (exception: Message forwarding. See 3.4). There are very few good reasons for calling packetFree(). One is if a packet is allocated, and then the application "regrets" and has to undo for one reason or another.

Message passing is very efficient due to the use of pointers. Several pools for different packet size are possible. See function `packetMakePool()`.

5 Time Control

A timer is simply a packet waiting in a list of timer packets. It is put into the timer list by the *start* function, and may be removed any time by the *stop* function. When it times out, the timer packet is removed from the timer list and given to its destination process which is called due to the packet. Timers can provide time-out mechanisms or make a process sleep a given time.

6 Fatal Error Handling

A `void putchar(int ch)` function can be installed for printing fatal errors to a suitable channel. The function

```
void nesosInstallPutChar(void (*event_pchar)(int ch),
                        void (*panic_pchar)(int ch));
```

is available for that purpose. If some illegal conditions are detected by **nesos**, the `panic_pchar()` function is used to print an error message.

The function `void nesosPanic(char *fmt, ...)`; uses `panic_pchar()` and can be used by the application code whenever required.

7 Message Monitoring

A `void putchar(int ch)` function can be installed for printing useful messages to a suitable channel. The function

```
void nesosInstallPutChar(void (*event_pchar)(int ch),
                        void (*panic_pchar)(int ch));
```

is available for that purpose. Events detected by **nesos** are printed using the `event_pchar()` function. What kind of events is printed?

See section Packets. A packet has a **dmask** field. Timer packets always have `dmask = 0x8000`;

There are two functions for setting corresponding masks:

```
void nesosSetIMask(unsigned int imask);
void nesosSetOMask(unsigned int omask);
```

If an AND operation between `dmask` and `imask` results in a non-zero value, a message is printed when given to the `packetPut()` function. If an AND operation between `dmask` and `omask` results in a non-zero value, a message is printed when given to the destination process function. The function `void nesosEvent(char *fmt, ...)`; uses `event_pchar()` and can be used by the application code whenever required.

8 Global nesos variables

8.1 nesos.c

process_descr_tab	Table of pointers to PROCESS_DESCRIPTOR structs for installed processes. Size MAX_PROCESSES. Never modify this table.
process_num	The number of installed processes.
process_current	The process identity of the current process.
process_index	The index into process_descr_tab, pointing to the current process.
time	Incremented by nesosCyclic(), a function called periodically. The variable gives time in ticks since reset or last overflow.
nesos_imask	Set by nesosSetIMask(). See section Message Monitoring.
nesos_omask	Set by nesosSetOMask(). See section Message Monitoring.
nesos_dmask	Multipurpose debug mask.
process_packets_hipri	List pointers for High priority packets. Do not modify.
process_packets_nopri	List pointers for Low/Normal priority packets. Do not modify.
process_packets_timer	List pointers for Timer packets. Do not modify.
event_putchar	Function pointer. See section Message Monitoring.
panic_putchar	Function pointer. See section Message Monitoring.
nesos_error	Set when internal errors arise. Zero after reset. Can be cleared by user.
nesos_allinit	Zero after reset. Non-zero after all processes are initialized. See section The state variable

8.2 packet.c

```
#define PKDIFFSIZENUM 5 // defined in packet.h
```

```
PACKETPOOL ppolarray[PKDIFFSIZENUM];
int ppolnum;
```

ppolarray	A table of max length PKDIFFSIZENUM, containing PACKETPOOL structs for created packet pools. Never modify this table.
ppolnum	The number of packet pools.
nesos_allinit	Zero after reset. Non-zero after all processes are initialized. See section The state variable

```
typedef struct packetpool {
    PACKET *startptr; /* Head of pool list */
    char *poolmem; /* ptr to pool memory */
    unsigned int memsize; /* size of pool memory */
    unsigned int grossize; /* size (bytes) of a packet */
    unsigned int packetnum; /* number of packets */
    unsigned int netsize; /* net size of a packet */
    unsigned int usedmax; /* statistics: max pk used */
    unsigned int packetleft; /* packets left in pool */
} PACKETPOOL;
```

startptr	The packet pool is arranged into a single linked list of packets. This is the head pointer.
poolmem	Pointer to the memory area used for the packet pool.
memsize	Size of poolmem in bytes.
grossize	The total size (plus alignment size) of a packet.
packetnum	The total number of packets in the pool.
netsize	The net size (i.e. data, no header), including alignment size, of a packet.
usedmax	The maximum number of packets allocated. For statistics purposes.
packetleft	Packets free for allocation.

8.3 timer.c

```
PACKET    *timer_start;
```

Timers are arranged into a single linked list of packets. This is the head pointer.

9 Alignment

Some parts of **nesos** chop an area of memory into smaller suitable elements. The function `packetMakePool()`, chops a large memory area into packets. Some compilers and processor architecture demands that such structs must be properly aligned.

A value, `ALIGN` is defined for that purpose. `ALIGN` will make some **nesos** structs aligned to addresses starting at modulo 2^{ALIGN} (Example: `ALIGN == 4`, the structs start at addresses dividable by 16. I.e. the 4 least significant address bits are zero.)

```
#define ALIGN 4    // nesos.h
```

10 Message Priority

There are three different queues for packets with different priorities. High priority packets are handled first. When the higher priority queues are empty, the lower priority queues are checked. The three queues are:

- High priority packet queue.
- Normal/Low priority packet queue.
- Lowest priority timer packet queue. When a timer is stopped and the timer is not found, this queue is checked as well.

11 API – nesos.c

11.1 nesosInit

```
*===== nesosInit =====
*
* Purpose:      To be called first.
*
*
* Input:       none
* Return:      none
*
void nesosInit(void)
```

11.2 nesosSetIMask

```
*===== nesosSetIMask =====
*
* Purpose:      Set trace mask for input messages.
*
*
* Input:       mask
* Return:      none
*
void nesosSetIMask(unsigned int imask)
```

11.3 nesosSetOMask

```
*===== nesosSetOMask =====
*
* Purpose:      Set trace mask for output messages.
*
*
* Input:       mask
* Return:      none
*
void nesosSetOMask(unsigned int omask)
```

11.4 nesosInstallProcess

```
*===== nesosInstallProcess =====
*
* Purpose:      Install process.
*
*
* Input:       ptr to process descriptor containing
*              all necessary process information.
* Return:      none
*
void nesosInstallProcess(PROCESS_DESCRIPTOR *pdp)
```

11.5 nesosRun

```
*===== nesosRun =====
*
* Purpose:      The mail program loop must call this function
*               as often as possible.
*
* Input:       none
* Return:      0: No process to run, 1: Process ran
*
int nesosRun(void)
```

11.6 nesosCyclic

```
*===== nesosCyclic =====
*
* Purpose:      To be called every time tick. E.g. from a
*               timer interrupt handler.
*
* Input:       none
* Return:      none
*
void nesosCyclic(void)
```

11.7 packetPut

```
*===== packetPut =====
*
* Purpose:      Put packet from process to input list.
*               This is for Low/Normal Priority packets
*               PACKET.src = process_current;
*
* Input:       ptr to packet
* Return:      none
*
void packetPut(PACKET *pp)
```

11.8 packetPutX

```
*===== packetPut =====
*
* Purpose:      Put packet from External source to input list.
*               This is for Low/Normal Priority packets
*               User must set PACKET.src
*
* Input:       ptr to packet
* Return:      none
*
void packetPutX(PACKET *pp)
```

11.9 packetPutHiPri

```
*===== packetPutHiPri =====
*
* Purpose:      Put packet from process to input list.
*               This is for High Priority packets
*               PACKET.src = process_current;
*
* Input:       ptr to packet
* Return:      none
*
void packetPutHiPri(PACKET *pp)
```

11.10 packetPutHiPriX

```
*===== packetPutHiPri =====
*
* Purpose:      Put packet from External source to input list.
*               This is for High Priority packets
*               User must ser PACKET.src
*
* Input:       ptr to packet
* Return:      none
*
void packetPutHiPri(PACKET *pp)
```

11.11 nesosInPacketForwarding

```
*===== nesosInPacketForwarding =====
*
* Purpose:      Mark input packet as forwarded to avoid that nesos
*               frees the packet when the application exits.
*               NB: Don't forget to call packetPut() as well.
*
* Input:       none
* Return:      none
*
void nesosInPacketForwarding(void)
```

11.12 nesosSaveInPacket

```
/===== nesosSaveInPacket =====
*
* Purpose:      The input packet is put into a save list.
*               The number of saved packets is limited to a maximum.
*               The max can be set using nesosSetSavedLimit()
*               Default max value is Infinite (=0)
*
* Input:       ptr to PACKET_DPTRS_LIM (list handle)
*               packet ptr
* Output:      none
* Return:      1: Ok. 0: Failed because list is full
*
int nesosSaveInPacket(void)
```

11.13 nesosSetSavedLimit

```
/===== nesosSetSavedLimit =====
*
* Purpose:      Set maximum number of elements of a
*               save list
*
* Input:       max value
* Output:      none
* Return:      none
*
void nesosSetSavedLimit(unsigned int maxval)
```

11.14 nesosInstallPutChar

```
*===== nesosInstallPutChar =====  
*  
* Purpose:      Install character output functions for  
*               event messages and panic messages.  
*  
* Input:       ptr to event_put_char function  
*               ptr to panic_put_char function  
* Return:      none  
*  
void nesosInstallPutChar(void (*event_pchar)(int ch),  
                        void (*panic_pchar)(int ch))
```

11.15 nesosPanic

```
*===== nesosPanic =====  
*  
* Purpose:      A printf-like function. To be called when a  
*               non-reversible fault is detected.  
*  
* Input:       fmt          formatting string  
*  
* Return:      none  
*  
void nesosPanic(char *fmt, ...)
```

11.16 nesosEvent

```
*===== nesosEvent =====  
*  
* Purpose:      A printf-like function. To print an event.  
*  
* Input:       fmt          formatting string  
*  
* Return:      none  
*  
void nesosEvent(char *fmt, ...)
```

11.17 nesosExeFunctionTable

```

typedef struct function_id {
    int  sn;          /* signal / number value      */
    int  st;          /* suggested next state / state */
    void (*fp)(void); /* function ptr              */
} FUNCTION_ID;

/*===== nesosExeFunctionTable =====
*
* Purpose:      This function is given an array of FUNCTION_ID.
*               The array is terminated with a NULL function ptr.
*               The FUNCTION_ID table is described above.
*               The array is scanned for equal state and signal,
*               When found, the corresponding function is called.
*
*               The purpose is to provide a convenient method for the
*               user to call a function corresponding to the state
*               and signal value of the input message.
*
* Input:        ptr to array of FUNCTION_ID
* Return:       0 = Not found. The default function is executed.
*               1 = Found. The corresponding function is executed.
*               -1 = Bad FUNCTION_ID table. State not found.
*/
int nesosExeFunctionTable(FUNCTION_ID *fip)

```

The FUNCTION_ID table structure is:

```

<state_A_header>
<state_A_signal_i> (repeated for every different signal i of state A)
<state_B_header>
<state_B_signal_j> (repeated for every different signal j of state B)
(... repeated for every different state)
<terminator>

```

Definitions:

```

<state_S_header> = { sn, st, fp },
sn = Number of different signals in state S
st = State value
fp = Default function to call if no signal value is found

```

```

<state_S_signal_J> = { sn, st, fp },
sn = Signal value (J)
st = Next state. If next state depends on input etc., st = STATE_APPSEL (=2). Then, if not
    changed by the application code, the original state (entrystate) is restored by nesos.
fp = Function ptr corresponding to current state S and signal J

```

```

<terminator> = { 0, 0, NULL }

```

12 API - packet.c

12.1 packetMakePool

```
*===== packetMakePool =====
*
* Purpose:      To be done first.
*               This function must be called a number of times to
*               build message pools of different sizes.
*
*               Each time this function is called, a new area of
*               memory, and the size of that memory is given.
*               The net size of a packet is given as well.
*               The net size MUST be different and increasing.
*               Net size of zero is allowed.
*               A maximum of PKDIFFSIZENUM different net
*               packet sizes can be configured.
*               (defined in packet.h)
*               The FIRST time, pindex MUST be 0.
*               The NEXT time(s), pindex MUST be 1,2,3 and so on.
*               The maximum pindex value is PKDIFFSIZENUM - 1
*
*               The memory size (m) for n number of packets:
*               m = n * (sizeof(PACKET_HEAD) + packet_net_size) +
*               alignment space if ALIGN is defined.
*
* Input:        ptr to pool memory
*               size of pool memory      (bytes)
*               net size of data packet (bytes)
*               pindex, see above
* Return:       Ok: number of pools, Error: -1
*
int packetMakePool(char *ppool, unsigned int ppsize,
                  unsigned int netsize, int pindex)
```

12.2 packetAlloc

```
/*===== packetAlloc =====
*
* Purpose:      Alloc a Packet from the pool. A packet of size equal or
*               greater than minsize is returned. If the pool is empty,
*               NULL is returned.
*
* Input:        minimum net size of packet
* Output:       none
* Return:       ptr to Packet, see above
*
*
PACKET *packetAlloc(unsigned int minsize)
```

12.3 packetFree

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

12.4 packetUsedMax

```
/*===== packetUsedMax =====
*
* Purpose:      Returns the maximum number of packets in use
*
*/
unsigned int packetUsedMax(int pindex)
```

12.5 packetNetsize

```
/*===== packetNetsize =====
*
* Purpose:      Returns the net size of packet
*
*/
unsigned int packetNetsize(int pindex)
```

12.6 packetNumberOf

```
/*===== packetNumberOf =====
*
* Purpose:      Returns total number of packets
*
*/
unsigned int packetNumberOf(int pindex)
```

13 API - timer.c

13.1 Introduction

A timer is a hidden object, which after a given time will send a message packet to a given destination and then disappear. A waiting process will then be called. The signal value of the packet header can be used to distinguish the timer packet from other packets. The number of active timers is only limited by the memory space. For more information, see the function descriptions below.

The timers are inserted to the list according to the expiration time. Those who expire first are at the front of the list. The timer value of the first one is the number of ticks until it expires. The timer value of the next timers is relative to the previous. This way, only the first timer has to be decrement. When decrement to zero, all timers with zero are removed from the list and put into the process input list.

13.2 timerStart

```
*===== timerStart =====
*
* Purpose:      A timer is started, or re-started if it already
*               excists. The timer will excist until it is deleted by
*               timerStop or until it has timed out. If it excists
*               when the time elapses, the function checkEventTimerList()
*               will pop it from the list and transmit it to destination.
*
*               The timers are ordinary packets arranged in 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() decrements the first timer,
*               and then remove all timers with zero time.
*
* Input:        source, destination and signal,
*               time [ticks]
* Output:       none
* Return:       none
*
void timerStart(int signal, unsigned long time)
```

13.3 timerStop

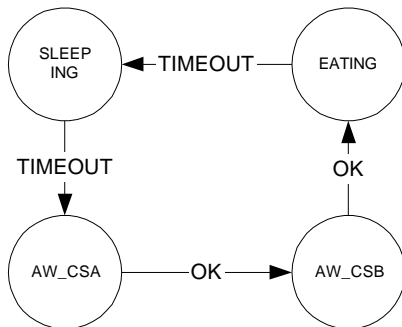
```
*===== timerStop =====
*
* Purpose:      A timer, started using timerStart(),
*               is deleted.
*
* Input:        source, destination and signal
* Output:       none
* Return:       If the timer does not excist
*               return 0, otherwise 1
*
*
*
int timerStop(int signal)
```

14 An example

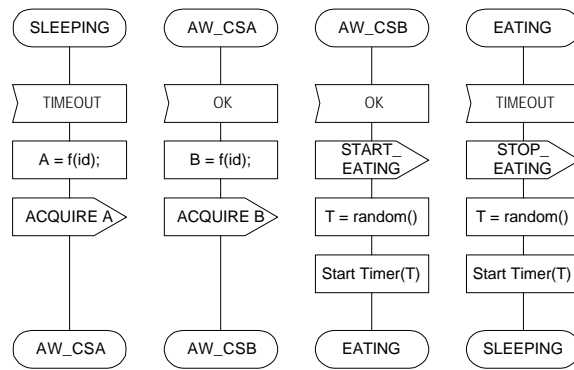
A number of philosophers are sitting around a round table. Each philosopher owns a disc, but they must share chopstick with their neighbours. Before eating, a philosopher must acquire two chopsticks, the one at the left hand and the one at the right hand. I.e. when a philosopher eats, his neighbour can not. A philosopher is either sleeping, waiting for chopsticks to be ready or eating. A deadlock situation could occur if all philosophers grabbed the left (or right) chopsticks then began waiting for the other one. Therefore, they grab different chopsticks first.

Find the model of a dining philosopher below. His four states are: Sleeping, Awaiting_ChopStick_A (AW_CSA), Awaiting_ChopStick_B (AW_CSB), and Eating. The START_EATING / STOP_EATING messages from the philosophers are sent to a display process. This process has only one state and calls a Display function

State diagram

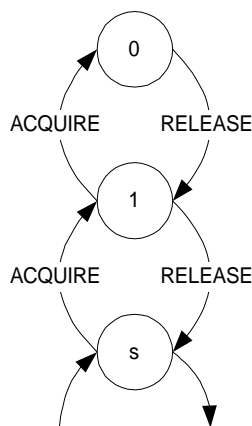


SDL diagrams

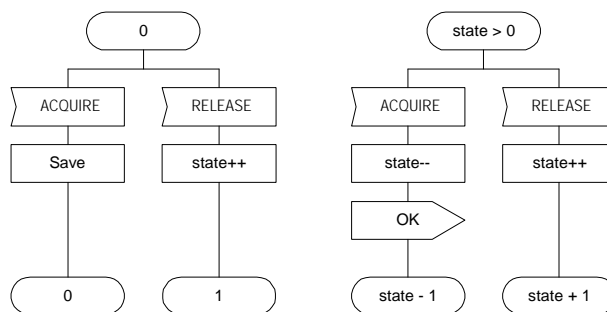


The model of a chopstick is much like any other resource model. The number of resources is one; therefore, the initial state is 1. Acquire requests are saved when all resources are reserved (See 3.6 Message saving)

State diagram



SDL diagrams



```

/***** MODULE INFO *****/
/*
** File name   : df.c
*/
/* AUTHOR      : Jan Erik Nilsen           */
/* VERSION     : 1.0                       */
/* DATE        : Thu May 04 14:01:12 2000  */
/*
* Compiler    : Microsoft cl /AL /Gs /W4
* Library     :
* Contents    : The classic Dining Philosophers Problem of Dijkstra
*               solved by semaphores.
*               Test program for nesos
*
*/
#include <math.h>
#include <stdio.h>
#include <string.h>
#include <conio.h>
#include <stdlib.h>
#include <dos.h>

#include "nesos.h"
#include "timer.h"
#include "packet.h"

extern int          process_current;
extern unsigned int nesos_dmask;
extern int          nesos_error;
void dfDisplayStatus(int id, int awake);
void dfDisplayProcess(void);
void dfPhilosopherProcess(void);
void dfChopstickProcess(void);
double expdistrand(double mean);

/* ---- Process IDs ----- */
#define PHNUM      5          /* Number of Philosophers */
#define PH0_PID    0          /* identity of first philosopher */
#define CS0_PID    (PH0_PID+PHNUM) /* identity of first chopstick */
#define DSP_PID    (CS0_PID+PHNUM) /* identity of display process */

/* ---- Signal definitions ----- */
#define S_PH_TIMER 1          /* Philosopher timer */
#define S_DISPLAY  2          /* Display state message */
#define S_ACQUIRE  3          /* Acquire chopstick */
#define S_RELEASE  4          /* Release chopstick */
#define S_OK        5          /* Chopstick granted */

PROCESS_DESCRIPTOR  phpid[PHNUM]; /* dfPhilosopherProcess */

/* ---- Philosopher state ----- */
#define PH_SLEEPING 0          /* Sleeping */
#define PH_AW_CSA   1          /* Awaiting Chopstick A */
#define PH_AW_CSB   2          /* Awaiting Chopstick B */
#define PH_EATING   3          /* Eating */

PROCESS_DESCRIPTOR  cspid[PHNUM]; /* dfChopstickProcess */
PROCESS_DESCRIPTOR  dpid;         /* dfDisplayProcess */

#define PKDIFFNUM  4
typedef struct {
    unsigned int nsize;
    unsigned int msize;
} PKCONF;

static const PKCONF pkstab[PKDIFFNUM] = {
    { 0, 256}, { 4, 256}, {16, 1024}, {64, 1024}};

void interrupt (*orig_clock)();
void interrupt myClock();
void myPutch(int ch);

```

```

void main(int argc, char *argv[])
{
    int    i, ch;
    char   *cp;

    argv = argv; argc = argc;

    printf("Dining Philosophers.  q for Quit\n\n");
    nesosInit();
    nesosInstallPutChar(myPutch, myPutch);
/*
    nesosSetIMask(0xFFFF);
    nesosSetOMask(0xFFFF);
    nesos_dmask = 0xFFFF;
*/
    for (i = 0; i < PKDIFFNUM; i++) {
        if ((cp = (char *)malloc(pkstab[i].msize)) == NULL) {
            printf("Memory allocaction error\n");
            exit(4);
        }
        packetMakePool(cp, pkstab[i].msize, pkstab[i].nsize, i);
    }
    if (nesos_error) exit(4);

    orig_clock = getvect(8); /* timer int */
    setvect(8, myClock);

    dpid.id = DSP_PID;
    strcpy(&dpid.name[0], "DPID");
    dpid.pp = dfDisplayProcess;
    nesosInstallProcess(&dpid);

    for (i = 0; i < PHNUM; i++) {
        phpid[i].id = PH0_PID + i;
        strcpy(&phpid[i].name[0], "PPID0");
        phpid[i].name[4] = (char)(i + '0');
        phpid[i].pp = dfPhilosopherProcess;
        nesosInstallProcess(&phpid[i]);

        cspid[i].id = CS0_PID + i;
        strcpy(&cspid[i].name[0], "CPID0");
        cspid[i].name[4] = (char)(i + '0');
        cspid[i].pp = dfChopstickProcess;
        nesosInstallProcess(&cspid[i]);
    }

    for (i = 0;; i++) {
        nesosRun();
        if (kbhit()) {
            ch = getch();
            if (ch == 'q') break;
            if (ch == 't') timerPrint();
            if (ch == 0) {
                ch = getch() | 0x100;
                if (ch == 0x175) break; /* Ctrl + End */
            }
        }
    }

    setvect(8, orig_clock); /* restore old timer handler */
    printf("\n");
    for (i = 0;; i++) {
        if (packetUsedMax(i) == 0xFFFF) break;
        printf("Pk up=%u nz=%u\n", packetUsedMax(i), packetNetsize(i));
    }
    exit(0);
}

```

```

/*===== dfDisplayStatus =====
*
* Purpose:      Display Philosopher status.
*
* Input:       personal identity [0..(PHNUM-1)]
*              awake TRUE/FALSE
*/
void dfDisplayStatus(int id, int awake)
{
    #define LEDSTR_SIZE ((PHNUM * 4) + 4)
    static char      ledstr[LEDSTR_SIZE];
    static int       init = 0;
    static unsigned int counter = 0;
    int i;

    if (!init) {
        for (i = 0; i < LEDSTR_SIZE; i++) ledstr[i] = ' ';
        for (i = 0; i < PHNUM; i++)      ledstr[i*4] = '-';
        ledstr[(i*4) + 1] = 0;
        init = 1;
    }
    if (id < 0 || id >= PHNUM) printf("ERROR id = %d\n", id);
    else {
        ledstr[id*4] = (char)((awake)? 219 : '-');
        printf("%7u:   %s\r", counter++, &ledstr[0]);
    }
}

/*===== dfPhilosopherProcess =====
*
* Purpose:      There are PHNUM dining philosophers sharing the
*              same code, but using different process descr.
*
*/
void dfPhilosopherProcess(void)
{
    void sendPacketToDisplay(int on);
    void sendPacketToChopstick(int dst, int sig);

    int    csa, csb;
    PACKET *pp;
    int    index = process_current - PH0_PID;

    if (phpid[index].state == STATE_INIT) { /* initialisation state */
        timerStart(S_PH_TIMER, (unsigned long)expdistrand(40.0));
        return;
    }

    if ((index & 1) == 0) {
        csa = (index+1) % PHNUM;  csb = index;
    }
    else {
        csa = index;  csb = (index+1) % PHNUM;
    }

    switch (phpid[index].state) {
    case PH_SLEEPING:
        sendPacketToChopstick(CS0_PID + csa, S_ACQUIRE);
        phpid[index].state = PH_AW_CSA;
        break;

    case PH_AW_CSA:
        sendPacketToChopstick(CS0_PID + csb, S_ACQUIRE);
        phpid[index].state = PH_AW_CSB;
        break;

    case PH_AW_CSB:
        timerStart(S_PH_TIMER, (unsigned long)expdistrand(40.0));

```

```

    sendPacketToDisplay(1);
    phpid[index].state = PH_EATING;
    break;

case PH_EATING:
    timerStart(S_PH_TIMER, (unsigned long)expdistrand(20.0));
    sendPacketToChopstick(CS0_PID + csa, S_RELEASE);
    sendPacketToChopstick(CS0_PID + csb, S_RELEASE);
    sendPacketToDisplay(0);
    phpid[index].state = PH_SLEEPING;
    break;

    default: nesosPanic("No such state");
}
}

void sendPacketToDisplay(int on)
{
    PACKET *pp;

    pp = packetAlloc(2);
    if (pp == NULL)
        nesosPanic("pp == NULL");
    else {
        pp->dst = DSP_PID;
        pp->signal = S_DISPLAY;
        pp->dmask = 1;
        pp->data[0] = (char)(process_current - PH0_PID); /* identity */
        pp->data[1] = (char)on; /* "on" */
        pp->length = 2;
        packetPut(pp);
    }
}

void sendPacketToChopstick(int dst, int sig)
{
    PACKET *pp;

    pp = packetAlloc(0);
    if (pp == NULL)
        nesosPanic("pp == NULL");
    else {
        pp->dst = dst;
        pp->signal = sig;
        pp->dmask = 1;
        pp->length = 0;
        packetPut(pp);
    }
}

/*===== dfChopstickProcess =====
*
* Purpose:      There are PHNUM chopsticks sharing the
*               same code, but using different process descr.
*
*/
void dfChopstickProcess(void)
{
    PACKET *pp;
    int index = process_current - CS0_PID;

    if (cspid[index].state == STATE_INIT) { /* initialisation state */
        cspid[index].state = 1; /* Number of resources */
        return;
    }

    if (cspid[index].inpp->signal == S_RELEASE) cspid[index].state++;
    else if (cspid[index].inpp->signal == S_ACQUIRE) {

```

```

    if (cspid[index].state == 0) nesosSaveInPacket();
    else {
        pp = packetAlloc(0);
        if (pp == NULL)
            nesosPanic("pp == NULL");
        else {
            pp->dst = cspid[index].inpp->src;
            pp->signal = S_OK;
            pp->dmask = 1;
            pp->length = 0;
            packetPut(pp);
        }
        cspid[index].state--;
    }
}
else nesosPanic("Unknown signal");
}

/*===== dfDisplayProcess =====
*
* Purpose:      This process is waiting for a status message
*               packet from a Dining Philosopher.
*               A system dependent display function is called,
*               and then the packet is released.
*
* Input:       ptr to pid
*/
void dfDisplayProcess(void)
{
    if (dpid.state == STATE_INIT) { // initialisation state
        return;
    }
    dfDisplayStatus(dpid.inpp->data[0], dpid.inpp->data[1]);
}

/*===== expdistrand =====
*
* Purpose:      The function computes a Random
*               Negative Exponential distributed value.
*               This is an estimate for the time between
*               two independent events, e.g. visitors.
*
* Input:       mean time [system ticks]
* Return:      random time [system ticks]
*/
double expdistrand(double mean)
{
    return(-mean * log(1.0 - ((double)rand() / 32768.0)));
}

void myPutch(int ch)
{
    printf("%c", ch);
}

void interrupt myClock()
{
    (*orig_clock)();
    nesosCyclic();
}

```