L3 Documentation

Marion Schalm, Jean Wolter, Michael Hohmuth

26.12.1995 (unfinished)

Contents

Chapter 1

L3 types

The following chapters will refer to various data types. They are defined in the header file *l3types.h*.

ByteT This type describes a single byte in gcc. It is a simple *unsigned char*.

DWordT This type describes a double word in gcc. It is a normal *int*.

TShortDS, TLongDS; This type describes a dataspace id. There are two types of id's, a short and a long version. The short version represents a dataspace from the standard pager. The other id stands for a dataspace from any pager.

```
typdef int TShortDS;

struct tagLongDS {
   ThreadT PagerID;
   TShortDS DSID;
   unsigned short MagicWord;
   unsigned short TaskNo;
};
typedef struct tagLongDS TLongDS;
```

TaskT This type describes a task id. A task id consists of different elements, the:

- the number of the task,
- the version,
- the chief of the task,
- and the address of task control block zero((Tlb0High << (10+14)) + Tlb0Low).

These elements are packed into an eight byte structure, which looks like the following (Caution!: The compiler allocates the bits from the right to the left side) :

```
struct tagBTask {
   unsigned Tlb0Low:14;
   unsigned TaskNo:10;
   unsigned Tlb0High:8;

   unsigned VerLow:14;
   unsigned ChiefNo:10;
   unsigned VerHigh:8;
};
typedef struct tagBTask TBTask;

struct tagSTask {
```

```
   int Low,High;
};
typedef struct tagSTask TSTask;

union tagTask{
   TBTask    B;
   TSTask    L;
   long long ID;
};
typedef union tagTask TaskT;
```

ThreadT This type describes a thread id. A task has at least one thread, but can also have more then one. A thread id looks similiar to a task id. It is also an eight byte structure with the following elements:

- the number of the thread,
- the version,
- the chief of the thread,
- the stationnumber,
- and the nesting level within the clan hierarchy

The structure looks like the following.

```
struct tagBThread {
   unsigned VerLow:12;
   unsigned ThreadNo:14;
   unsigned VerHigh:6;

   unsigned StationNo:12;
   unsigned ChiefNo:14;
   unsigned Depth:6;
};
typedef struct tagBThread TBThread;

struct tagSThread {
   int Low,High;
};
typedef struct tagSThread TSThread;

union tagThread{
   TBThread    B;
   TSThread    L;
   long long ID;
};
typedef union tagThread ThreadT;
```

TMessageDope This type defines a messagedope, a four byte item describing the structure of a message. For easier access to the single elements it is a union with a simple int and a structure containing five elements.

```
struct tagMessageDope
{
   unsigned Dummy:8;
   unsigned DWords:8;
   unsigned Strings:4;
   unsigned Flexpages:4;
```

```
   unsigned Dataspaces:8;
};
typedef struct tagMessageDope TMessageDope;

union tagMsgDope
{
   DWordT   DWord;
   TMessageDope MsgDope;
};
typedef union tagMsgDope MsgDopeT;
```

StrDopeT This type describes a string, a data area of bytes. It consists of a string part, the part to be sent, and a buffer part for an incoming string. Both parts are described by an address and the size of the specified area.

```
struct tagStrDope
{
    DWordT StrLength;
    DWordT StrAddress;
    DWordT StrBufSize;
    DWordT StrBufAddress;
};
typedef struct tagStrDope StrDopeT;
```

TFlexPage This type descibes a so-called flexpage, one or more pages of an address space, which are send or received within a message. The structure looks like the following:

```
struct tagFlexPage {
   DWordT MapCode;
   DWordT SndBase;
   DWordT SndFPage;
   DWordT RcvFPage;
};
typedef struct tagFlexPage TFlexPage;
```

Chapter 2

L3 IPC

2.1 IPC Overview

L3 tasks communicate with each other via messages. The communication is synchronous, i.e. sender and receiver have to be ready to exchange the message. If the receiver isn't ready, the sender has to wait until it is ready or the specified timeout is over and vice versa. The communication is direct, i.e. there are neither comunication channels nor links, only global thread and task identifiers.

L3 ipc is implemented with a single system call. The parameters select the operations to be executed. There are five operations:

- send a message to a certain thread (snd msg != nil msg, rcv msg == nil msg)
- receive a message from a certain thread (snd msg == nil msg, rcv msg != nil msg, closed wait)
- receive a message from any thread (snd msg == nil msg, rcv msg != nil msg, open wait)
- send and receive a message (rpc) to and from a certain thread (snd msg != nil msg, rcv msg != nil msg, closed wait)
- send a message to a certain thread and receive a message from any thread (snd msg != nil msg, rcv msg != nil msg, open wait)

A message is described by two so called *dopes*. The first dope describes the layout of the message area, the other a message to be sent (or a received message). The message to be sent can be the complete message area or a subset of it. So you can use one message area for both, for a send operation and for a receive operation.

Each message contains a direct string (mandatory) and may contain indirect strings (optional), flexpages (optinal) and dataspaces (optional). The parts of a message are strictly arranged. First comes the direct string, then the indirect strings and last but not least the dataspaces.

A direct string consists of a number of dwords. An indirect string hast two parts, one for a message to be sent and another for a message to be received. Both are described by an address of an area and the length of that area. If an indirect string hast to be sent, you fill the first part with it's address and length. If you want to receive an indirect string, you allocate a buffer and fill the second part with it's address and size.

under construction

The third part, the flexpage, is a page from the address space of the sender, which will be mapped into the receivers address space. It is described with four parameters. The sender specifies the map mode he allows (read only, read/write), the address and size of the page to be send and as an additional parameter the baseaddress in the receivers address space. The receiver specifies the address and size of the area, to which the page should be mapped. So we can describe a flexpage as a tupel of address and size like this: FlexPage (address, size).

It is possible, that the send page is bigger or smaller then the specified receive flexpage. Then the kernel has to calculate something like an effective flexpage for the final mapping. That means, the kernel has to divide the bigger flexpage into flexpages with the size of the smaller one and has to deceide, to which or which of this subflexpages should be mapt. For that he needs an additional parameter from the sender, the sendbase. This parameter provides the missing informations, if such a splitting occurs.

Consider the following situation: A pager provides different data spaces and a client maps a dataspace from that pager. The pager sends flexpages with different sizes, if a page fault occurs and the client accepts different sizes. The effectiv flexpages are calculated like this:

size of send flexpage = size of receive flexpage This is the trivial case. The page will be mapped to receive flexpage with size of receive flexpage.

size of send flexpage < size of receive flexpage This could be the normal situation. The client would accept more then the page causing the page fault, but the pager isn't able to deliver more then one page. Now the kernel has to decide, to which address in the receive flexpage the send flexpage should be mapped. The effectiv flexpages would be calculated like this:

- effective send flexpage = send flexpage
- effective receive flexpage =

```
( (rcvfpage AND -size of rcvfpage +
  (sndbase  AND (size of rcvfpage-1) AND -size of sndfpage),
  size of sndfpage )
```

The pager sends a flexpage (address, size), and a sendbase = page fault address. That means, the pager delivers at least the page causing the page fault, but if the client accepts a bigger flexpage, id asked to map it to the page fault address.

size of send flexpage > size of receive flexpage This would be the other situation. The pager is able to deliver more then one page, but the client accepts for instance only one.

The effectiv flexpages would be calculated like this:

- effective receive flexpage = receiver flexpage
- effective send flexpage =

```
( (sndfpage AND
  (rcvfpage AND (size of sndfpage-1) AND -size of rcvfpage),
  size of rcvfpage )
```

A flexpage is described by a structure consisting of the following elements:

the map code The map code describes the mapping operation to be performed by the kernel. You can map a page with the following modes:

- IPC_FPAGE_READ_ONLY

  The flex page is mapped read only.
- IPC_FPAGE_READ_WRITE

  The flex page is mapped read write.

the send base The send base is used by the kernel to calculate the final flex page, if the receive flex page is bigger then the send flex page. In the case of a page fault it is normally the page fault address.

the send flex page The send flex page describes the flex page to be send.

the receive flex page The receive flex page describes an area in which a process is willing to accept mappings.

And the last but not least elements are the dataspaces described by dataspace id's.

Direct and indirect strings are copied strictly, memory objects lazily. A message can have 257 dwords (direct string), 15 indirect strings, 15 flexpages and 255 dataspaces. Every message has at least two dwords, no matter if you use them or not. If you don't need the dwords, you have to insert a *dummy* dword for each missing dword.

That results from the special IPC design of L3. If you receive a message, the first two dwords are delivered in registers. Because a high proportion of messages are very short, e.g. acknowledgements from a server process, this leads to a performance gain. Another advantage is the usage of indirect strings. They help to avoid copy operations at user level. For further informations about L3 ipc see: “Improving IPC by Kernel Design” from Jochen Liedtke, GMD.

Let's summerize:

- L3 communication
  - is synchron,
  - is direct,
  - has timeouts,
  - and is implemented with a single system call.
- Messages
  - contain two *dopes* describing the message, one for the size of the message area, the other for the message to be sent,
  - have to have at least two dwords,
  - are strictly ordered (1. dwords, 2. strings, 3. flexpages, 4. dataspaces),
  - can be used for send and receive in one call (call(dest, &msg, &msg,-1,-1)),
  - should be as big as the expected reply, if used for receive.

Let's take a look at some examples.

Consider the following situation:

A string needs to be sent to a terminal device driver. The string is located in an array named *text*. The terminal driver requires an operation code and a string. The message could look like the following:

```
struct {
   TMessageDope message_size;
   TMessageDope message;
   int device_order;
   int dummy;
   TStringDope text;
} msg;
```

You can see, that you have two dwords althought you need only one. It is important to insert this dummy dword, because the ipc interface expects to find at least two dwords. You would fill the elements of the message as follows:

```
msg.message_size.DWord = ((2 dwords) - 2) + (1 indirect string)
msg.message            = message_size;
msg.device_order       = order_code_read;
msg.text.StrAddress    = text;
msg.text.StrLength     = strlen(text);
```

There are three things to notice:

- The dummy element is ignored.
- The number of dwords zero, althougth there are two dwords. But each message contains at least these two dwords. that why you decrement the number by two.

- The message dope has the same value as the message size dope, because you want to send the whole message area.

Another situation:
A client requests a read operation from a server. The server expects a read operation code, a number of bytes to read and replies with an errorcode and an indirect string containing the requested bytes. The message could look like the following:

```
struct {
   TMessageDope message_size;
   TMessageDope message;
   int order;
   int number;
   TStringDope buffer;
} msg;
```

As you can see, the message contains an indirect string, although you only want to send an order code and another dword. But you need to supply a message buffer for the reply of the server. You can use the same message buffer for both operations. You would fill the message elements like this:

```
char buffer[MAX_BUFFER_LENGTH];
msg.message_size.DWord = ((2 dwords) - 2) + (1 indirect string);
msg.message            = ((2 dwords) - 2);
msg.device_order       = order_code_read;
msg.text.StrBufAddress = buffer;
msg.text.StrLength     = MAX_BUFFER_LENGTH;
```

As you can see, the dopes arn't equal. The message area is bigger then the message to be sent. You send only two dwords, but the message area is big enough for the incoming reply, so you can use one message area for both, for the send operation and for the receive operation.

Remark: For compatibility reasons L3 supports the old V2 ipc calls. It emulates V2 ipc with two special dwords, the so called *emulation dope* and the *emulation token* 2.2.4.

2.2 IPC Parameters

2.2.1 Parameters of IPC

send/receive timeout
- 0 <= n <= 2**30 : n ms timeout
- n = -1 : never

destination/source id
- thread id (V3 type!)
- intr id
- nil id (0)

address of message to be sent
- linear address of message to be sent (dword aligned!)
- nil address (-1), no message will be sent

address of receive buffer
- linear address of message buffer (dword aligned!) for receiving a message, may be identical with snd vector address, IPC includes receive operation
- nil address (-1), IPC does not include receive operation

closed option
- closed (0) : if IPC includes receive operation, a message will be accepted solely from the specified dest
- open (2) : if IPC includes receive operation, messages from anyone are accepted

received message dope `result code`
`+ (dwords     << 8 )`

```
+ (strings    << 16)
+ (flexpages  << 20)
+ (dataspaces << 24)
```

2.2.2 Message structure

message structure:
```
DWORD message size dope
DWORD message send dope
DWORD direct_part[max dwords+2]
STRINGDOPE indirect_part_option[max strings]
DATASPACE dataspace_part_option[max dataspaces]
```

message size dope =
```
  (max dwords     << 8)
+ (max strings    << 16)
+ (max dataspaces << 24)
```

max dwords 0 <= n < 256, size of direct part is (n+2)*4

max strings 0 <= n < 256, length of indirect part is n*4*4

max dataspaces 0 <= n < 256, length of dataspace part is n*4

msg snd dope =
```
  dummy
+ (dwords << 8)
+ (strings << 16)
+ (dataspaces << 24)
```

dummy 0 <= n < 256 ignored

dwords 0 <= n < 256, message to be sent has n+2 (!) dwords in direct part

strings 0 <= n < 256, message to be sent has n strings in indirect part

dataspaces 0 <= n < 256, message to be sent has n dataspaces in dataspace part

StringDope
```
DWORD string length
DWORD string address
DWORD string buffer size
DWORD string buffer address
```

string length/string addr describes string to be sent, ignored if string not to be sent.

string buffer string buffer addr/ describes buffer for receiving a string, ignored if no corresponding string received

relation between size and send dope For size and send dope of the same message structure must hold:

```
send dwords     <= size max dwords   AND
send strings    <= size max strings  AND
send dataspaces <= size max dataspaces
```

message buffer for send and receive If a message structure is used for both, send and receive operation, it must hold:

```
max(dwords,      expected dwords in reply)     <= max dwords AND
max(strings,     expected strings in reply)    <= max dwords AND
max(dataspaces,  expected dataspaces in reply) <= max dwords
```

receive only buffer If a msg structure is used as a receive buffer only in an IPC, the contents of the snd dope is irrelevant.

resend a received message Simply storing the received message dope (EAX) in the receive buffer's send dope and received dword0 (EBX) and 1 (ECX) in the receive buffer's direct part (the dword part) leads to a sendable message structure.

2.2.3 Results

received msg dope = `result code`

```
+ (dwords     <<  8)
+ (strings    << 16)
+ (dataspaces << 24)
```

result code `0b000   ok, direct`

```
0b001   ok, redirected from inner clan
0b010   ok, redirected from outer clan
0x10    send timeout
0x18    receive timeout
0x20    dest not existent
0x28    source not existent
0x30    erroneous (e.g. send to myself)
0x40    dwords overflow
0x48    dataspaces overflow
0x50    strings overflow
0x58    string overflow
0x60    dwords overflow at rcv
0x68    dataspaces overflow at rcv
0x70    strings overflow at rcv
0x78    string overflow at rcv
```

dwords 0 <= n < 256 , n+2 (!) dwords in direct msg part are received; dword 0 and 1 are contained in EBX and ECX, not (yet) in the message buffer!

strings 0 <= n < 256 , n indirect strings are received

dataspaces 0 <= n < 256 , n dataspaces are received

2.2.4 V3 - V2 IPC

How to send a message to a thread, which uses the V2 IPC:

V2 ipc uses a message vector describing the message to be send/received. This vector allows to order the single parts of the message. You can compose two messages with the same components, but different structure, eg:

```
     V2                        V2                    V3

DWORD     svcode      DWORD     svcode        DWORD     svcode
dataspace ds          dataspace ds            DWORD     dummy
string    name        DWORD     dummy         string    name
DWORD     dummy       string    name          dataspace ds
```

This isn't possible with V3 ipc. There is only one order, first part is the direct part, followed by the strings, if any, and last the dataspaces, if any. If you want to send a message to a thread, which uses the V2 ipc and depends on the correct order of the message components, you have to add two integers to the direct part of the message. The first integer "v2_emu_dope" describes the V2 structure of the message (beginning on the right site with the first element) and the second is something like a token for the V2 emulation.

```
v2_emu_dope
0x12345678
```

The v2_emu_dope consists of the following elements:

```
  00      INT       (2 bits)
1101      String    (4 bits)
  10      DATASPACE (2 bits)
  11      nil       (2 bits)



For instance: TEXT,INT,INT,INT,DATASPACE :


11 11 11 11 11 11 11 11 11 11 10 00 00 00 11 01
                              -- -- -- -- -----
                               |  |  |  |   +------- 1. elem, string
                               |  |  |  +----------- 2. elem, INT
                               |  |  +-------------- 3. elem, INT
                               |  +----------------- 4. elem, INT
                               +-------------------- 5. elem, DS
```

The direct part of the V3 message looks like the following:

```
int1, int2, int3, 0xfffff80d, 0x12345678
```

2.2.5 V2 versus V3 ID-s

how to construct a V3 thread id out of a V2 thread id:

```
v3.high = 0x00043000 ;

v3.low  = ((v2.low  & 0x00003fff) << 12) +
          ((v2.high & 0x00000fff)      ) +
          ((v2.high & 0x0003f000) << 18)
```

2.3 IPC C-Library

This is a first attempt to write a c-library for the l3 ipc system interface. It was done in a straigth forward manner and needs some future optimisations to utilisize the special features of the l3 ipc, like short messages.

2.3.1 Parameters

This chapter describes briefly the parameters of the IPC calls of the C library. All IPC calls have at least a *msg* and a *timeout* parameter, some have an additional *thread* parameter.

msg The parameter *msg* represents the message to be sent. It should be the linear address of a structure containing at least the messagesizedope, messagedope and the direct part. 2.2.2

timeout The parameter *timeout* specifies the time, which the client will wait, if the server isn't ready. The value is given in milliseconds and means:

- 0 <= *timeout* <= 2**30 : *timeout* ms timeout
- *timeout* == -1 : never

thread The parameter *thread* is the id of the sending/receiving thread. It can be:

- thread id (V3 type!)
- intr id
- nil id (0)

return value The return value describes the result of the requested operation. If IPC includes a receive operation and a message was received, the return value is the messagedope of the received message. If the operation was successful, the lower bits are zero (*result code* & 0xf8 == 0), otherwise the lower eight bits represent the error code of the operation2.2.3. If the message was redirected on any clan border, the lower two bits are not zero.

There are different categories of errors.

Thread specific errors
- Thread don't exists (destination not existent/source not existent)
- Thread not ready (send timeout/receive timeout)

Message buffer specific errors
- Message buffer not big enough (miscellaneous overflows)

2.3.2 Call

*int Call(ThreadT thread, void *msg, void *reply, int send_timeout, int receive_timeout);*

This function implements the normal RPC system call. The client process sends a message to a server process with thread id *destination_id* and waits for a reply from the server.

Parameters:

msg The parameter *msg* represents the message to be sent. It should be the address of a structure containing at least the messagesizedope, messagedope and the direct part. 2.2.2

reply The parameter *reply* is the linear address of a message buffer 2.2.2 for the received message. It should be at least as big as the expected reply. Remark: You can use the same message buffer for *msg* and *reply* 2.2.2.

send_timeout The parameter *send_timeout* specifies the time, which the client will wait, if the server isn't ready 2.3.1.

receive_timeout The parameter *receive_timeout* specifies the time, which the client will wait for the reply of the Server 2.3.1.

thread The parameter *thread* is the id of the receiving thread 2.3.1.

return value The return value contains the messagedope of the received reply, if the call was succesful, otherwise the lower eight bit describes the error 2.3.1, 2.2.3.

2.3.3 ReplyAndWait

*int ReplyAndWait(ThreadT thread, void *reply, ThreadT *client, void *order, int send_timeout, int receive_timeout);*

This function sends a reply to client *thread* and waits for the next order from any client. It is designed for the implementation of a server, which typically sends a message to a distinct thread and waits for a message from any thread.

Parameters:

reply The parameter *reply* represents the message to be sent. It should be the address of a structure containing at least the messagesizedope, messagedope and the direct part. ref id="Message-structure"//

order The parameter *order* is the linear address of a message buffer 2.2.2 for the received message. It should be at least as big as the expected order.

Remark: You can use the same message buffer for *reply* and *order* 2.2.2.

timeout The parameter *timeout* specifies the time, which the calle will wait, if the server isn't ready 2.3.1.

thread The parameter *thread* is the id of the receiving thread 2.3.1.

return value The return value contains the messagedope of the received order, if the call was succesful, otherwise the lower eight bit describes the error 2.3.1, 2.2.3.

2.3.4 Send

*int Send(ThreadT thread, void *msg, int timeout);*

This functions sends the message *msg* to the thread *thread*. If the thread isn't ready to receive the message, the sender waits *timout* milliseconds 2.3.1.

Parameters:

msg The parameter *msg* represents the message to be sent. It should be the address of a structure containing at least the messagesizedope, messagedope and the direct part. 2.2.2

timeout The parameter *timeout* specifies the time, which the caller will wait, if the destination isn't ready 2.3.1.

thread The parameter *thread* is the id of the receiving thread 2.3.1.

return value The return value is zero, if the call was succesful, otherwise the lower eight bit describes the error 2.3.1, 2.2.3.

2.3.5 Receive

*int Receive(ThreadT thread, void *msg, int timeout);*

This function waits for a message from thread *thread*. If no message arrives within *timeout* milliseconds 2.3.1, it returns.

Parameters:

msg The parameter *msg* represents the message buffer for the incoming message. It should be the address of a structure containing at least the messagesizedope, messagedope and the direct part and should be at least as big as the expected message 2.2.2.

timeout The parameter *timeout* specifies the time, which the caller will wait for a message 2.3.1.

thread The parameter *thread* is the id of the sending thread 2.3.1.

return value The return value contains the messagedope of the received message, if the call was succesful, otherwise the lower eight bit describes the error 2.3.1, 2.2.3.

2.3.6 Wait

*int Wait(ThreadT *thread, void *msg, int timeout);*

This function waits for a message from any thread. If no message arrives within *timeout* milliseconds 2.3.1, it returns.

Parameters:

msg The parameter *msg* represents the message buffer for the incoming message. It should be the address of a structure containing at least the messagesizedope, messagedope and the direct part and should be at least as big as the expected message 2.2.2.

timeout The parameter *timeout* specifies the time, which the caller will wait for a message 2.3.1.

thread The parameter *thread* is a pointer to the bufffer that will contain the id of the sending thread 2.3.1.

return value The return value contains the messagedope of the received message, if the call was succesful, otherwise the lower eight bit describes the error 2.3.1, 2.2.3.

2.4 Kernel Interface

System Call: IPC
Invocation: INT 50h

Input
- EAX address of receive buffer + closed option
- EBX send timeout
- ECX address of message to be send
- EDX destination id low
- ESI destination id high
- EDI receive timeout

Output:
- EAX received msg dope
- EBX received dword0
- ECX received dword1
- EDX source id low
- ESI source id high
- EDI undefined
- EBP undefined
- DS,ES linear space
- FS,GS undefined

Remark: Undefined means, there is no garantie for the contents of that register. If it has to remaim, it has to be saved before IPC invocation.

Chapter 3

Virtual Memory Management

The following sections describe L3's virtual memory management facilities: address spaces, data spaces and memory managers, and their interaction.

3.1 Overview

3.1.1 Address Spaces

An address space (aka *virtual* address space) defines the set of valid virtual addresses that any thread executing within the task owning the address space is allowed to reference. An address space is owned by exactly one task.

Creation and Destruction

An address space is created when the task is created, and destroyed when the task is destroyed. When a new task is created, its address space contains only one valid memory range, namely a mapping of its standard data space (see below (3.1.3)). If a task needs to establish other mappings after it has been started, it can only do so with a **DSMapRegion** system call.

Memory Range Establishment

The semantics associated with a valid range of virtual memory are provided by memory managers. When a new memory range is established in an address space, a thread id of a manager providing those semantics needs to be specified. No kernel calls exist for a task to directly affect the semantics associated with its memory ranges. A task has such control only by virtue of choosing memory managers providing the desired functionality or by communicating directly with its memory managers to direct their actions.

In order to establish a new range of virtual memory in an address space, the task needs to specify a *map vector*, a data structure containing the specific details of the memory range (placement, size, alignment, access mode, data space to use and data space offset and size (the *map region*)). There exist two helper functions **DSFillMapVectorExt** and **DSFillMapVectorStd** which assist in building a map vector.

DSMapRegion is the system call to establish a new range of virtual memory in an address space. This call communicates the map vector address and protection attributes to the kernel. After specifying a map vector in a **DSMapRegion** system call, the vector remains under kernel control until the region is unmapped and may not be recycled for other purposes.

Memory ranges can be made invalid (i.e., can be unmapped) by calling **DSMapRegion** specifying the appropriate map vector and access mode NO_ACCESS.

3.1.2 Data Spaces

A data space is an abstract memory object which can be mapped into the address space of tasks. The semantics associated with a data space are provided by the memory manager that backs it. Data spaces can be of variable sizes up to 4 GB.

From the kernel's perspective, data spaces are uniquely identified by a <user_task, memory_manager, data_space_number> triple, where the data space number is an identifier which is never interpreted by the kernel. It is the manager's duty to decide whether the same data space number in two different user tasks address the same memory object or not (in the standard memory manager, it does not; see section 3.1.3).

Data spaces can be manipulated by means provided by the memory manager, and of course by modifying memory ranges mapped from them.

3.1.3 Memory Managers

A memory manager (sometimes also called a *pager*) provides the semantics associated with the act of referencing memory regions which have mapped data spaces backed by it by providing (or denying) access to pages of data that have been referenced by user tasks.

Every thread can be a memory manager for every other thread. There is no special initialization necessary to become a memory manager, and the kernel provides no special access protection for them (as with for all L3 IPC). Once the id of some thread has been specified in a **DSMapRegion** system call, the kernel starts treating the thread as the memory manager for the region (and send page fault messages to it, which, of course, can be ignored or refusingly replied to by the thread).

Basic Manipulation of Address Spaces

Manipulation of a task's address space takes the following basic form:

- A task establishes a new memory range, specifying a memory manager and data space number for backing that range (**DSMapRegion**). This does not involve the memory manager.
- A thread of the task attempts to reference a portion of this memory range (most likely simply by touching it). Since that portion does not yet exist in memory, the thread takes a page fault. The kernel sends a message requesting the page for the desired access *on behalf of the faulting thread* (i.e., with the thread as the sender) to the associated memory manager.
- The memory manager analyzes the request and responds *directly to the thread* (for which the kernel will have set up a closed receive operation). It can respond either refusingly, raising a memory failure exception in the faulting thread, or affirmatively by supplying a flexpage or several flexpages (memory ranges in the manager's address space) which the kernel maps into the faulting task's address space; after that, the kernel restarts the thread that caused the page fault.

  Of course, memory ranges specified by the memory manager will most probably be memory backed by other memory managers. It is the memory manager's responsibility to ensure that memory referenced by a returned flexpage currently resides in main memory (perhaps by touching the respective physical memory pages and thereby invoking other memory managers) so that the faulting task will not immediately fault again.
- The kernel is free to forget the flexpage mapping at any time (perhaps when evicting the page from main memory) which may result in making clients send page fault messages requesting previously supplied pages again.
- The memory manager may also forcibly remove a flexpage mapping established earlier (**FlushFPage**), for instance, when the manager is low on memory and wants to recycle memory regions previously committed as flexpages. This may also make clients send page fault requests for pages invalidated like that.
- The client task dispels the memory range (unmaps it) with **DSMapRegion**, access mode NO_ACCESS. Again, this does not involve the memory manager.

The kernel should be viewed as using main memory as a (directly accessible) cache for the contents of the various data spaces.

If a thread requests write access for a page and the memory manager permits it, then it is certain that the page has been modified. This way it is possible for a memory manager to keep track of modified pages.

The Standard Memory Manager

The standard memory manager pages data spaces backed by it against the system's paging space, the *background*. It maintains an ownership relation between its data spaces and tasks: Only the owning task can map any specific data space at a time.

Special Properties of the Standard Memory Manager.

There are several ways the standard memory manager is special or different from other memory managers:

- Data spaces backed by the standard memory manager can be manipulated through system calls. Other memory managers need to have their data spaces manipulated through IPC via L3 messages. (Currently, the standard way to do this is with the General Pager Protocol, described in a later section (6.3).)
- Data spaces backed by the standard memory manager can be transferred (and change ownership) with normal L3 messages.
- The standard data space specified on task creation (**SVBeginTask**) must be backed by the default memory manager (because only such data spaces can be transferred with Supervisor protocol messages).

  The Supervisor maps the specified standard data space into the first 32 MB (address range 0–0x2000000) of the newly created task's address space.

Manipulating Data Spaces Backed by the Standard Memory Manager.

Standard memory manager data spaces are created by copying a special data space called *nilspace* (**DSCopy**). Nilspace is special in that it does not own any pages. By copying nilspace the new data space gets an unique identifier. If created in this way, a data space still has no pages allocated from the background.

There are several possibilities to populate a data space with pages allocated from the background:

- Write to a page in a memory range mapped to a data space backed by the standard memory manager for the first time.
- Use any of **DSCopy**, **DSCopyPages** or **CopyPagesInside** to populate till then unused pages with copies of already existing pages.

3.2 C Bindings for System Calls Related to Data Spaces

The following sections describe the C interface for manipulating data spaces. All calls except the system call **DSMapRegion** and the helper function **DSFillMapVectorExt** can only manipulate data spaces backed by the standard memory manager.

XXX need to clean up the following sections...

3.2.1 Fill Map Vector

This function is provided twice, ones for the filling of a map vector for a data space provided by the standard pager and the other for map vectors for data spaces provided by external pagers. The difference are the task identifiers of both the external pager and the task or thread, which wants to map the data space.

The Map Vector filled ones is used for both mapping and unmapping of the given region of the data space.

```
void DSFillMapVectorStd(TMapVector *m_v, int lower_bound, int upper_bound,
DataSpaceT ds, int size, ByteT align, int offset)

void DSFillMapVectorExt(TMapVector *m_v, int lower_bound, int upper_bound,
                     DataSpaceT ds, TTask pager_id, TTask my_task, int size,
                     Byte align, int offset)
```

For supporting the map function there exists a function for filling the map vector.
Parameters:

m_v The parameter represents the address of the map vector for this map region. The vector is filled before the first mapping and will be unchanged until the unmapping of the region.

lower bound The parameter lower bound defines the lower bound of the mapping address in the address space of the mapping thread.

upper bound The parameter upper bound defines the upper bound of the mapping address in the address space of the mapping thread. The kernel tries to find a start address inside the interval specified by lower and upper bound. If both addresses ar identical the kernel maps the region at this address.

pager_id The parameter pager_id specifies the requested pager. If this parameter has the value 0 the standard pager is chosen.

size The parameter size specifies the size of the map region. The lower four bytes of size should be 0. Otherwise the size will be/ rounded to the next bigger value.

align The parameter align specifies the requested alignment. It can be 0 default alignment should be used. 3..31 use alignment 2**align other values reserved

my_task The parameter my_task specifies the mapping task. Only the lower byte is used for the operation.

size The parameter size specifies the size of the map region. Beginning at data space base + offset size bytes will be mapped into the address space of the task.

offset The parameter offset specifies the beginning of the maop region inside the data space.

3.2.2 Map Region

```
void *DSMapRegion(TMapVector *map_vector, Byte access, int *size)
```

The function MapRegion forces the kernel to map or to unmap a region of a data space specified in the map_vector.
Parameters:

map_vector The parameter map_vector specifies the map vector for the mapping operation. All significant values are specified in this vector. Note: You can't use one map vector for different regions at the same time!

access The parameter access speciefies the wanted access to the map region. access can be one of:

1. 0 NO_ACCESS
2. 1 READ_ONLY
3. 2 NO_CHANGE
4. 3 READ_WRITE

By giving NO_ACCESS the mapped region will be unmapped.

size The parameter size is an integer variable, where the size of the mapped region is given back if the operation succeeds.

return value The return value is an address in the address space of the mapping task, where to the map region is mapped.

3.2.3 Copy Data

```
void DSCopy(DATASPACE source, DATASPACE *destination)
```

Parameter:

source The parameter source represents the identifier of the data space which should be copied by this operation. If this data space id is nilspace a new data space is created.

destination The parameter destination represents the identifier of the destination data space. If source is nilspace, destination will be the identifier of a new data space.

3.2.4 Next Data Space

```
DATASPACE DSNext(DATASPACE data_space)
```

This function provides the Identifier of the next data space of the calling task. If the identifier is 0 for nilspace, the function provides the identifier of the standard data space of the task. Calling the function with the identifier of the last data space of the task, the function provides nilspace.
Parameter:

data_space The parameter data space describes the identifier of the data space whose next data space (the one which has the next index) should be provided.

return value The return value is the identifier of the next data space. This is either the id of the standard data space, if the input was nilspace, or nilspace, if the input was the id of the last data space of the task.

3.2.5 Nilspace

```
DATASPACE DSNilspace()
```

This function provides the data space identifier of nilspace. In general, this is 0.

return value The return value is the data space id of nilspace, 0.

3.2.6 Associated Data Space

```
DATASPACE DSAssociatedDataspace(void *address)
```

This function provides the data space identifier for a given address.

address The parameter address is an address inside the data space associated with this address.

return value The return value of this function is the identifier of the associated data space or nilspace, if there is no data space mapped at this address.

3.2.7 Copy Pages

```
void DSCopyPages(DATASPACE source, DATASPACE destination, int from_page,
               int to_page, int pages)
```

This function copies a number of pages from one datspace to another.

source data space The parameter source data space represents the id of the data space whose pages will be copied.

destination data space The parameter destination data space represents the identifier of the destination to which the pages will be copied.

from page The parameter from page specifies the start page inside the source data space, from which n pages will be copied.

to page The parameter to page specifies the start page in the destination data space, where the pages will be copied to.

pages The parameter pages specifies the number of pages, which will be copied.

3.2.8 Pages

```
int DSPages(DATASPACE data_space)
```

This function provides the number of pages of the given data space.

data space id The parameter data space id specifies the data space.

3.2.9 Delete Pages

```
void DSDeletePages(DATASPACE data_space, int from_page, int to_page, int
                   pages)
```

This function deletes a number of pages of the given data space. The pages won't really deleted, they will only be unmapped.

data space id The parameter data space id represents the data space, from which a number of pages should be deleted.

from page The parameter from page specifies the start page, from which the pages should be deleted.

to page The parameter to page specifies the end page, up to which the pages should be deleted.

pages The parameter pages specifies the number of pages to be deleted.

3.2.10 Delete, Exists

```
DATASPACE DSDelete(DATASPACE data_space)
```

This function deletes the specified data space. The data space hasn't been mapped, but it can be mapped.

data space id The parameter specifies the data space to be deleted.

return value The return value is no data space, if the operation succeeded.

3.2.11 Exists

```
TBool DSExists(DATASPACE data_space)
```

This function traces whether the given data space exists or not.

data space id The parameter specifies the data space to be checked.

return value The return value is 1 if the data space exists, 0 otherwise.

3.2.12 Move

```
void DSMove(DATASPACE *source, DATASPACE *destination)
```

This function moves a data space to another. The source data space is deleted after moving. If the destination data space doesn't exist, it will be created.

source This parameter represents the source data space, which will be moved.

destination This parameter represents the destination data space, which will be existing after the moving.

3.2.13 Next

```
DATASPACE DSNext(DATASPACE data_space)
```

This function provides the identifier of the following data space of the given data space of one task. Every data space of a task has got an index, which is used to get the next data space.

data space The parameter specifies the identifier of the data space, whose follower is searched.

return value The return value is the identifier of the follower of the given data space.

3.2.14 Next Page

```
int DSNextPage(DATASPACE data_space, int pageno)
```

This function provides the next used page of a given data space. There is a page sequence in every data space. If the page number -1, the function provides the first used page of the data space. There can be gaps in the page sequence.

data space This parameter specifies the data space, whose pages are searched.

pageno This parameter represents the number of the page, whose following page is searched. If it is -1, the first page is searched.

return value The return value is the number of the following page. If there are no more pages, the return value is -1.

3.2.15 Dataspaces

```
int DSDataspaces(TTask task)
```

This function provides the total number of all (named and unnamed) data spaces of the given task. The maximum number is 16383.

task This parameter specifies the task, whose number of data spaces is searched.

return value The return value is the total number of all data spaces of task.

3.2.16 Copy Pages

```
void DSCopyPagesInside(DATASPACE data_space, int from_page, int to_page,
                       int pages)
```

This function copies a number of pages inside one data space. The copying starts at the page from page and copies n pages to the page to page. In this case lazy copying is used.

data space The parameter data space specifies the data space, where the copying will take place.

from page The parameter from page defines the start page of the copy operation.

to page The parameter to page specifies the start page of the destination region, where the pages are copied to.

pages The parameter pages specifies the number of pages, which will be copied.

3.2.17 Index

`int DSIndex(DATASPACE data_space)`

This function provides the index of a data space inside the sequence of data spaces of one task.

data space The parameter data space specifies the data space identifier of the data space, whose index is searched.

return value The return value is the index of the given data space. The index is part of the data space identifier.

3.2.18 Storage

`int DSStorage(DATASPACE data_space)`

This function provides the storage, used by the given data space. The storage size is given in bytes, but page aligned, i.e. the result can be divided by page size 4096 without any rest.

data space This parameter specifies the data space, whose size should be provided.

return value The return value is the used storage in bytes.

3.2.19 Type

`int DSType(DATASPACE data_space)`

This function provides the type of the given data space, if there was a type number assigned to the data space.

data_space The parameter specifies the data space, whose type should be provided.

return value The return value is the type of the data space.

3.3 Low-level Kernel Interface

Data spaces can be mapped into the address space of a task. The part of the data space which is mapped is called a map region. The conditions for the mapping of a region is specified in a so called map vector. There must be a map vector for each mapped region. This vector is filled by the mapping task, but controlled by the kernel. Don't use one map vector twice at the same time! Different regions of one data space can be mapped at the same time.

If a data space is mapped without errors the application can use it inside the requested size. By mapping a region of the data space the task gets the part of the data space which is described by the values for offset and size specified in the map vector. The offset is the beginning of the map window relativly to the beginning of the data space. If the task exceeds the borders of the mapped region, an error is created by the kernel. Otherwise the task can use every address inside the map region. If the page doesn't reside in the real memory, the kernel will create a page fault, which is handled by the pager which created the data space. In general this will be the standard pager. The mapping task won't see the page fault.

The map vector is a structur like the following:

```
struct {
```

```
    int lower_bound;
    union {
        DataSpaceIdT Long;
        DataSpaceOldIdT Short;
    } ds;
    int upper_bound;
    int size;
    int offset;
} TMapVector;
```

TDataSpaceId is a 32bit long identifier either consisting of the V2 - data space id and three integers set to 0 or consisting of four parts, if the data space is provided by the standard pager:

```
struct {
    int ds;
    int reserve[3];
} DataSpaceOldIdT;

struct {
    int magic_wx;
    int ds_no;
    int thread_low;
    int thread_high;
} DataSpaceIdT;
```

Chapter 4

L3 System Calls

Chapter 5

SV protocol

5.1 SV Overview

The task supervisor is responsible for the management of the task system. He creates tasks and threads and holds all informations about the task system. If you want to know something about the tasksystem or want to create a task or a thread, you have to communicate with the supervisor. The next sections describes the supervisor protocol and the C library. Caution! the current supervisor communicates with the old fashioned V2 ipc. You have to use the special V2 emulation (V2 emu dope and emu token).

5.2 Supervisorprotocol

begin **Purpose:** A new process is created.

Parameters:
- TEXT task name
- INT attributes
- instruction pointer
- stack pointer
- stdds {optional parameter}

attribut bits:

1..7 privilege This field specifies the requested privilege of the new process (0=user, 1=privileged, 2=supervisor, 3=kernel). The effective privilege is the minimum of the requested privilege and the privilege of the requesting process.

8..15 associated intr This field is relevant for hardware driver processes. It contains the number of the hardware interrupt (33..255), which should be linked to the new process. A value of 0 specifies 'no interrupt to be associated'.

16 resident
- = 0 The new process should be pageable and persistent. The creator must also be pageable.
- = 1 The new process should be resident. All pages allocated in the dataspace are loaded into main memory. They will never be swapped out. Resident processes are not persistent.

17..18 io permit
- = 00 The new process will have no access to IO ports. (IOPL=0 and io permission bitmap empty.)
- = 01 The new process may access selected IO ports (IOPL=0, non empty io permission bitmap). This mode is not yet implemented!
- = 11 The new process may access all IO ports and may disable interrupts (IOPL=3).

24 thread
- = 0 The new process should be autonomous, i.e a task with own address space and dataspaces.
- = 1 The new process should be a thread of the same task.

25 undeletable
- = 0 The new process may be deleted by its parent or a privileged task.
- = 1 The new process should be undeletable by others. It may delete itself or may be deleted by shutdown (if resident).

Instruction pointer and Stack pointer *Instruction pointer* and *stack pointer* specify the initial values for the EIP and ESP registers of the new process. CS is loaded with 'espace' (base=0, size=16MB, use32, executable, readable). All other segment registers are loaded with 'dspace' (base=0,size=32MB, use32, readable, writable).

Dataspace The optional dataspace message chapter is needed iff the thread bit in the attribute dword is reset, i.e. a stdds is needed only when creating a new task.

rename **Purpose:** The calling task or thread is renamed. (Caution! It gets an new version number)

Parameters: TEXT new task name

end **Purpose:** The process 'task' is deleted. The end message is only allowed if:

```
     task = order task
COR  (task < order task CAND task is deletable)
COR  (order task < supervisor) CAND NOT (order task < task)
CAND task is deletable)
```

Parameters: TASK task

halt **Purpose:** The process 'task' is halted, i.e. an error 'halt from terminal' is induced. The end message is only allowed if

```
     task = order task
COR  task < order task
COR  (order task < supervisor) CAND NOT (order task < task))
```

Parameters: TASK task

halt **Parameters:** TASK task

Remark Not yet implemented.

system catalogue **Purpose:** The system catalogue space is delivered.

Parameters: TASK task

Result: DATASPACE system catalogue space

father **Purpose:** The father task of *task* is delivered.

Parameters: TASK task

Result: TASK father of task

son **Purpose:** The first child task of *task* is delivered.

Parameters: TASK task

Result: TASK son of task. If there isn't any child task, it return *nil task*.

brother **Purpose:** The first brother task of *task* is delivered.

Parameters: TASK task

Result: TASK brother of task. If there isn't a brother task, it return *nil task*. It is possible to get nil task, although there are other tasks with the same parent. If you want to get all brothers of task x, get the father of task x, the son of this task and call *brother(actual brother)* starting with this *child task* until you get *nil task*.

name **Purpose:** The name of task *task* is delivered.

Parameters: TASK task

Result: TEXT name of task

task **Purpose:** The task id of task *name of task* is delivered.

Parameters: TEXT name of task

Result: TASK task. If there isn't a task with name *name of task*, nil task is delivered

exists task **Purpose:** Check for the existence of task *name of task* and return the result.

Parameters: TEXT task name

Result: BOOL flag. Caution: The supervisor return 0 for *true* and 1 for *false*.

set station name **Purpose:** The name of the station is set to *new name of station*.

Parameters: TEXT new name of station

Precondition: privilege (order task) >= system privilege

get station name **Purpose:** The name of the station is delivered.

Parameters: None

Result: TEXT name of station

supervisor version **Purpose:** The version number of the supervisor is delivered.

Parameters: NONE

Result: INT actual supervisor version

change supervisor **Parameters:**

- TEXT dummy
- INT password
- INT dummy
- INT dummy
- DATASPACE new stdds

5.3 SV C-Library

This chapter describes the library functions which provide an interface to the supervisor protocol. The function automatically transform the given identifiers into the format required by the supervisor and transform the results back to the format expected by the callee. The default identifier format is the version three format. If you want to use the version two format you have to include a `#define V2_IPC` before including `svprot.h`.

5.3.1 Parameters of SV C-Library

thread The parameter *thread* is the id of a thread in the default format, for which an operation is requested. If a thread id is delivered, it is an id in the same format.

name The parameter name is a normal zero terminated string representing the name of a thread or a station name.

max length The parameter max length describes the maximum length of a delivered string. The buffer for the deliverd string should be at least as big as *max length + 1*.

return value If the call was successful, the functions return the value *L3_SV_ACK*, otherwise *L3_SV_NACK*. If you try to rename yourself you can get a return value of *L3_SV_ERR_NACK*. That means, that you have to wait a certain time before you can request that operation again[1].

other parameters 5.2

[1] That is implemented to prevent code sequences like `while(1) SVRename(myself)` resulting in an undeletable thread.

5.3.2 Functions of SVCL

void SVProtInit(void); This function initializes the supervisorprotocol. It has to be the first call of this library.

int SVFather(ThreadT thread,ThreadT *father); This function delivers the *father* of thread *thread*.

int SVBrother(ThreadT thread,ThreadT *brother); This function delivers the first *brother* of thread *thread*. If there isn't a brother thread, it returns *nil thread*. It is possible to get nil thread, although there are other threads with the same parent. If you want to get all brothers of thread x, get the father of thread x, the son of this thread and call *brother(actual brother)* starting with this *child thread* until you get *nil thread*.

int SVSon(ThreadT thread,ThreadT *son); This function delivers the first child thread of thread *thread*. If there isn't any child, it delivers *nil thread*.

int SVEnd(ThreadT thread); `thread = order thread`

```
COR  (thread < order thread CAND thread is deletable)
COR  (order thread < supervisor) CAND NOT (order thread < thread)
CAND thread is deletable)
```

int SVName(ThreadT thread, char *name, int maxlen); This function delivers the zero terminated name of thread *thread*. The buffer *name* has to be at least as big as *max length + 1*.

int SVThread(char *name, ThreadT *thread); This funtions delivers the thread id of the thread *thread*. If the thread doesn't exist, it returns *nil thread*.

int SVRename(char *newname); This function renames the calling thread. The calling thread get's the new name *newname* and a new version number. This call isn't posible at all times. If a thread renames itself, it has to wait for a distinct time before it can rename itself again. The call returns *i don't no* , if this happens.

int SVHalt(ThreadT thread); The process 'thread' is halted, i.e. an error 'halt from terminal' is induced. The end message is only allowed if

```
     thread = order thread
COR  thread < order thread
COR  (order thread < supervisor) CAND NOT (order thread < thread))
```

int SVCatalog(ThreadT thread, TDataSpace *ds); todo

int SVExistsThread(char *name); todo

int SVGetStationName(char *name, int maxlength); todo

int SVSetStationName(char *name); todo

int SVLessThen(char *name, char *precursor); todo (unklar)

int SVBeginThread(char *name, int attrib, int ip, int sp, ThreadT *newthread); This function creates a new thread with the name *name* 5.2. The id of the new thread is returned in `*newthread`.

int SVBeginTask(char *name, int attrib, int ip, int sp, DATASPACE ds, ThreadT *firstthread); This function creates a new task with the name *name* 5.2. The id of the first thread of the new task is returned in `*firstthread`.

Chapter 6

External Pager Protocol

6.1 Overview

The following chapters describe the general pager protocol.

In L3 there is a standard pager which is part of the kernel and handles all data spaces (3.1.2). But if there is a need for the handling of external filesystems like a UNIX file system, rather use an external pager.

The task of an external pager is to provide an external memory or a part of an external memory as one or more linear addressed data spaces, which can be mapped into the address space of other users.

It is possible to use different external pagers. One of the pagers can handle a hard disk and provide it as data space, one can handle a floppy disk, and the next can provide files as data spaces, which can be mapped into the address spaces of som e UNIX users.

But all pagers use the same protocol, called the general pager protocol, which is described in the following chapters.

The pager distinguishes between services, used

- by the users of the pager
- by the kernel.

To distinguish the origin of the message there are two protocols:

1. the PageFaultProtocol
2. the General Pager Protocol

The messages sent to the pager are of the following form:

```
typedef struct {
    TMsgDope size;
    TMsgDope msg;
    DWord codeword1;
    DWord codeword2;
  }
```

In some cases there can be one or more additional parameters.

The results are given back by the pager with the two code words.

6.2 Page Fault Protocol

The service used by the kernel is the handling of some page faults, which the kernel ascertains. The kernel recognizes which pager is the owner of a page or a data space when a page fault occurs. In this case the kernel sends a message to the pager fourcing it to provide the page where the page fault occurs. The pager answers with the address of this page inside its own address space.

The page fault protocol has a special protocol identifier, like every L3 protocol. It is defined like follows:

```
#define pfp      0x30000000
```

The ix86 distinguishes between two kinds of access, read access and write access. Therefor the page fault procol knows about two access modes:

ReadAccess forces the pager to provide the page, where the page fault occured, the requested page is given back for read access. The pager can decide, whether the page is given back for write access, if the page was modified earlier.

WriteAccess the same with write access

To supply the necessary information to resolve the page fault the kernel sends two dwords describing the page fault.

dword 1 PFP + data space id

dword 2 pfaddress + access (pfaddress & ~31)

The pager tries to find the necessary pages and sends a reply consisting of two dwords and one or more flex pages. The dwords are the same like in the page fault message. The pager simply replace the access with the result of its own operation.

result code word 1 pfp + data space id

result code word 2 pfaddress + result (pfaddress & ~31)

- 0x00000 - ACK
- 0x00001 - illegal data space id
- 0x00010 - illegal address
- 0x100rw - illegal access (rw proposed)

There is a header file named pagefault.h which defines some structures and constants necessary to implement the external pager protocol.

6.3 General Pager Protocol

All other services of the pager indicated by the 2 in the code word are used by applications of the pager. These are in general the same services as the standard pager provides (see section 3.1.2).

The use of the functions is a bit different, that's why they will be explained in the following chapter.

The external pagers provide named and unnamed data spaces. Data spaces are created by copying an existing data space. By this the user will get an identifier of the data space. This is a 32 bit long identifier, which holds the task identifier of the called pager, the task number of the calling thread and an index, which is processed by the pager.

An existing data space can be named by the user by an operation of the pager. An already named data space can be renamed in the same way. The name of an existing data space can be determined by another operation.

Other operations are similiar to the operations of the standard pager like

```
#define GPP          0x20000000
```

Functions

- 1 CopyDataSpace
- 2 DataSpace
- 3 SetDSName

- 4 GetDSName
- 5 Pages
- 6 Type
- 7 Storage
- 8 Delete
- 9 NextPage
- A DeletePages
- B CopyPages

The code words are formed as follows:

code word 1 GPP + data space identifier

code word 2 function + additional info

The additional information and parameters depend on the function. It is described below.
Description of Functionsions

CopyDataSpace copies a data space from source to destination.

code word 1 GPP + data space identifier of the source data space. If this is 0, a new data space is created.

code word 2 CopyDataSpace

result code word 1
- 0 - ok + data space id
- 1 - illegal data space id

DataSpace provides the data space identifier of the data space associated with the given name

code word 1 GPP

code word 2 DataSpaces

additional StrDopeT name of the data space

result code word 1
- 0 (ok) + data space identifier
- 1 - illegal name

SetDSName sets the specified name to the given data space.

code word 1 GPP + data space identifier

code word 2 SetDSName

additional StrDopeT name of the data space

result code word 1
- 0 (ok)
- 1 - illegal name
- 2 - illegal data space id

GetDSName determines the specified name to the given data space.

code word 1 GPP + data space identifier

code word 2 GetDSName

result code word 1
- 0 (ok) - Name is stored at the given address
- 1 - illegal data space id

additional StrDopeT name of the data space

DeleteDataSpace deletes the data space with the given identifier

code word 1 GPP + data space identifier

code word 2 DeleteDataSpace

result code word 1
- 0 - ok
- 1 - illegal data space id

Pages determines the total number of the used pages of the data space with the given identifier

code word 1 GPP + data space identifier

code word 2 Pages

result code word 1
- 0 - ok
- 1 - illegal data space id

result code word 2 total number of pages

Index provides the number of the data space specified by the given identifier inside the data spaces sequence provided by the external pager

code word 1 GPP + data space identifier

code word 2 Index

result code word 1
- 0 - ok
- 1 - illegal data space identifier

result code word 2 index of the data space

DeletePages deletes the specified pages of the data space

code word 1 GPP + data space identifier

code word 2 Delete Pages + number of the pages to be deleted

additional parameter> from page , to page

result code word 1
- 0 - ok
- 1 - illegal data space id
- 2 - illegal pages
- 3 - illegal access

CopyPages copies the specified pages of the given data space to the given pages of the destination data space. The pages are copied in it's indata space sequence. If the number of pages of both intervalls isn't identical, the number of pages of the source data space is copied.

code word 1 GPP + source data space identifier

code word 2 CopyPages + destination data space identifier

additional parameter DWord from_page, to page inside the source

additional parameter DWord from_page, to_page inside the destination

result code word 1
- 0 - ok
- 1 - illegal data space id
- 2 - illegal pages
- 3 - illegal access

NextPage provides the number of the page following the page with the given number inside the data space

code word 1 GPP + data space identifier

code word 2 NextPage + number of the page whose follower should be provided, if this is -1, the first used page should be provided

result code word 1
- 0 - ok
- 1 - illegal data space id
- 2 - illegal page

result code word 2 number of the next used page, it is -1,if there are no more pages

6.4 C - Library

The functions of the gpp are provided in a library.

CopyDataSpace `void PCopyDataSpace(DataSpaceId source, DataSpaceIdT *dest)`

Parameter:

source This parameter specifies the data space to copy. If this is 0, a new data space is created.

dest This parameter specifies the identifier of the new created data space.

DataSpace `DtaSpaceIdT PDataSpace(char * name)`

Parameter:

name This parameter is the name of the data space.

result The result of this function is either the data space identifier, if the the data space associated with name exists, or 0, if it doesn't exist.

SetDSName `void PSetDSName(DataSpaceId ds, char *name)`

Parameter:

ds This parameter specifies the data space to be named.

name This parameter specifies the name, which is given to the data space.

GetDSName `void PSetDSName(DataSpaceId ds, char *name)`

Parameter:

ds This parameter specifies the data space, whose name should be traced.

name This parameter holds the name after the operation. If the data space was unnamed, it is NIL.

Pages `int PPages(DataSpaceId ds)`

Parameter:

ds This parameter specifies the data space, whose number of pages should be provided.

result The result of this function is either the number of the used pages of the given data space or -1, if the data space doesn't exist.

Type `int PType(DataSpaceId ds)`

Parameter:

ds This parameter specifies the data space, whose type is traced.

result The result of the function is either the type of the data space or -1 if the data space doesn't exist.

Storage `int PStorage(DataSpaceId ds)`

Parameter:

ds This parameter specifies the data space, whose storage size has to be provided.

result The result of this function is either the size of the used storage of the data space or -1 if the dat space doesn't exist.

Delete `void PDelete(DataSpaceId ds)`

Parameter:

ds This parameter specifies the data space to be deleted.

NextPage `int PNextPage(DataSpaceId ds, int start_page)`

Parameter:

ds This parameter specifies the data space.

start_page This parameter specifies the number of the page, whose follower should be provided. Is this -1, the first used page of the data space is provided.

result The result of the function is either the number of the next used page or -1 if there are no more used pages.

DeletePages `void PDeletePages(DataSpaceId ds, int from_page, int to_page, int pages)`

Parameter:

ds This parameter specifies the data space, whose pages have to be deleted.

from_page This parameter specifies the first page to be deleted.

to_page This parameter specifies the last page to be deleted.

pages This parameter specifies the total number of pages to be deleted.

CopyPages
```
void PCopyPages(DataSpaceId source, DataSpaceIdT dest, int from_page,
                int to_page, int pages)
```

Parameter:

source This parameter specifies the data space, whose pages have to be copied.

dest This parameter specifies the identifier of the data space, where the pages are copied to.

from_page This parameter specifies the first page to be copied.

to_page This parameter specifies the last page to be copied.

pages This parameter specifies the total number of pages to be copied.

Chapter 7

General Driver Protocol

7.1 Preliminaries

7.1.1 Purpose of this Document

This document is addressed to system programmers. It specifies the recommended L3 conventions for communicating with IO drivers as well as for writing and installing them.

Important concepts of L3 are flexibility and extendability. In the field of hardware drivers this means that privileged users may replace and add IO drivers. (The only exception is the backing store (hard disk) driver, which belongs to the kernel itself.) So the protocols specified in this document are not enforcible, but they are strongly recommended to all people implementing driver software.

It is recommended to obey the conventions specified here not only in hardware drivers but even when designing higher level software (if possible and senseful).

7.1.2 Related Documents

7.1.3 Method of Description

7.2 The General Driver Protocol

Usually hardware drivers in L3 are realized as a set of processes. Whether a driver consists of one, two or even more processes depends on the type of device. The General Driver Protocol (GDP) defines the communication between drivers and user or system processes. It is strongly recommended that all drivers use this protocol. If possible it should be used also by "high level drivers".

7.2.1 Orders and Replies

The method to communicate with drivers is the usual L3 inter process communication. So orders are sent and results are received as messages. This chapter defines the various message types used by the GDP.

An order message always consists of an order code followed by additional parameters. In the same way a reply message starts with a reply code and may be followed by additional parameters. In general the additional parameters are objects of various types (e.g. blocks, strings, characters, bitmaps ...), coordinates (block numbers, screen positions ...) and mode specifiers (colour, font ...). The GDP has the protocol type number 1. So all GDP order codes have the form

The same convention holds for GDP specific reply codes.

7.2.2 Message Codes

In GDP the message code is always the first element of a message. This section defines the codes used for orders, interrogations and displays. The message code contains the addressed subchannel, the message type, the desired function and the type of the involved object:

01 GDP identification

subchannel If a driver process distributes its orders to other processes or handles more than one physical or logical device, this field specifies the addressed subchannel. Its range is 0...255. The default subchannel is 0.

object type This field specifies the type of object involved. Its range is 0...4095. The concrete semantics of the requested operation depends on the combination of the object type and the function specified by the IO-bits.

function

reply The execution of an order is answered by a "reply". The same holds for an interrogation concerning an executable order. The reply code is followed by an object (specified in the type field) and an additional error status information referring to the acknowledged operation. The structure of the error status object is described in the object type table.

alternate in request, alternate out request Whenever an order or an interrogation is not executable, the driver has to reply an alternate in/out request. In the reply message the subchannel, object type and further parameters have to be as close at the original order or interrogation as possible, but the alternate request must specify an executable order.

in order An in order always requests the driver to send data (usually after performing an input operation) or the actual settings of control or status parameters. The specified object type determines the type of operation. The driver has to operate in synchrounous mode, i.e. has to reply after executing the order.

out order An out order sends data for output or control information to the driver. The specified object type determines the type of operation. The driver has to operate in synchrounous mode, i.e. has to reply after executing the order.

early in order Here one has to differentiate between driver controlled devices (e.g. disks, tapes) and devices controlled by external events (e.g. keyboards, RS232 and LAN receivers).

1. early in at driver controlled devices

   The driver is free to handle the order in synchronuous way (like an 'in order') or in the early mode. In the second case the driver has to reply immmediately a time delay after which the data is supposed to be available. Following this reply the driver will initiate the data input operation. Some time later it will receive a new early in order concerning to the same object. If the data is available now, the driver has to send the data as reply, otherwise a new supposed time delay.

2. early in at devices controlled by external events

   If the requested data is available within the driver, it has to send the data as reply, otherwise it must send a nil reply (without waiting for data).

early out order The driver is free to handle the order in synchronuous way (like an 'out order') or in the early mode. In the second case the driver has to answer an immediate 'early ack'. In contrast to synchronuous mode the data output is initiated after the early ack. Thus a write error may occurr later. If this happens, the reply to the next order will signal an error in the error status field.

Attention: There is no strict relationship between the error ack and the preceding early out operations. All between the last synchronuous operation and the error reply must be regarded as faulty.

interrogate in order, interrogate out order The driver is asked whether it could execute the specified order or not (and why not). An interrogation will never change the drivers state nor really read or write data.

7.3 Objects

7.3.1 Object Types

Object types are grouped in a hierarchical fashion. The 12-bit object type specifier is structured as follows:

Currently four device classes are known in this list:

- Stream IO devices, which simply read and/or write byte strings. Examples for such devices are RS232 or LAN Controller.
- Block IO devices, which can read and write sequences of blocks to specified addresses. Examples are disks and tapes.
- Keyboard devices, including mouses, joysticks...
- Display devices, which can show alphanumerical and/or graphical data. These are e.g. video screen, printers and plotters.

7.3.2 Object Type Table

This table shows the tree of all actually defined object types with its codings. Codings not contained within this table are reserved for further extensions.

- 0x0.. general
  - (...)
    - 0x000 nil object
    - 0x001 time delay
  - 0x01. error handling
    - 0x010 error status
    - 0x011 error report
    - 0x012 substitution report
  - 0x02. broadcast
    - 0x020 object
  - 0x03. system
    - 0x030 restart
- 0x1.. stream io
  - 0x10. data objects
    - 0x100 byte string
    - 0x101 datagram
  - 0x11. RS232
    - 0x110 transfer mode specifier
    - 0x111 control line specifier
    - 0x112 break event
    - 0x113 out flow control specifier
    - 0x114 in flow control specifier
  - 0x12. lan
    - 0x120 address
- 0x2.. block IO

  - 0x20. data objects
    * 0x200 block sequence
    * 0x201 block sequence ptr
  - 0x21. Floppy Disk
    * 0x210 transfer mode specifier
    * 0x211 media type specifier
    * 0x212 format
    * 0x213 format record
    * 0x214 deleted data mark
  - 0x22. Hard Disk
    * 0x220 transfer mode specifier
    * 0x221 format
    * 0x222 format record
    * 0x223 partition table
  - 0x23. streamer
- 0x3.. input
  - 0x30. keyboard
    * 0x300 scan ascii
    * 0x301 key
    * 0x302 shift state
    * 0x303 make break
    * 0x304 hot key
    * 0x305 diacritical flag
  - 0x31. mouse
- 0x4.. display
  - 0x40. alpha 8
    * 0x400 attr, char string
    * 0x401 attr, char string, glue
    * 0x402 (char,attr) string
    * 0x403 (char,attr) string, glue
    * 0x404 char filler
    * 0x405 (char, attr) scroller
    * 0x406 update string
    * 0x407 attribute mode
  - 0x44. graphics
  - 0x45. speaker
    * 0x450 beep
    * 0x451 click
    * 0x452 tone
  - 0x48. screen
    * 0x480 descriptor
    * 0x481 memory
    * 0x482 page selector

    * 0x483 write access
    * 0x484 palette descriptor
    * 0x485 palette selector
    * 0x486 hardware
    * 0x487 screen plane selector
    * 0x488 screen ega palette selector
    * 0x489 screen vga color register
    * 0x48b acivate screen
  - 0x49. window
    * 0x490 size
  - 0x4a. cursor/mouse
    * 0x4A0 descriptor
  - 0x4b. font

- 0x5.. raw
  - 0x50. via port
    * 0x500 byte
    * 0x501 word
    * 0x502 dword
    * 0x503 qword
    * 0x504 byte string
    * 0x505 word string
    * 0x506 dword string
    * 0x507 qword string

7.3.3 Object Description

The format (representation) of an object may depend on the message type. So in the detailed object description the format is marked by I, O, R or any combination of them:

I the representation within an 'in order', 'interrogate in order' or 'alternate in order request'

O the representation within an 'out order', 'interrogate out order' or 'alternate out order request'

R the representation within a reply.

If any object format is not given for one of these cases, this object type cannot be used in this context.

0x000 Nil Object

Format:

- (IO)R format: STRUCT ()

This object type specifies "no object". It is used e.g in simple ack or nak replies, which do not include any object.

When used in conjunction with orders or alternate order requests it has a special meaning (see 7.4).

0x001 Time Delay

Format:

- R format: INT

This specifies a supposed time period (in milliseconds) after which the operation should be retried. This reply may come only as a result to an early in order.

0x010 Error status

Format:

- R format: INT error status
- I format: STRUCT ()

The error status always describes the result of the last operation. It is only valid after executing an order. The error status consists of 8 error bits and an error probability descriptor k:

Error bits:

Error probability:

Remark: GDP requires that each reply message contains the current error status as an additional object, not specified in the object type field of the GDP code. So the error status type 0x010 will never be used within GDP messages. It is listed in this section for sstematic reasons only.

0x011 Error Report

Format:

- OR format: STRING (TASK driver proces id, INT gdp order code, first word of gdp message, error status, device specific 0, device specific 1, device specific 2)
- I format: STRUCT ()

0x012 Substitution Report

Format:

- OR format: STRING (TASK driver proces id, INT object type, old object id, substituted object id)
- I format: STRUCT ()

0x020 Broadcast Object

Format:

- OI format: STRUCT ()

The 'exec in' function of this object signals the start of a broadcast sequence and 'exec out' function signals the end. Remark: This object is only used for internal communication between the terminal process and its manager.

0x030 System Restart

Format:

- R format: STRUCT ()

Signals a system restart.

Remark: This object is only used for internal communication between the terminal process and its manager.

0x100 Byte String

Format:

- OR format: ROW string length BYTE string
- I format: INT max string length

The bytes are part of a sequential stream but do not form a closed object. So a byte string may be broken into smaller pieces or concatenated with other strings by the driver without changing the semantics. If a byte string should be read by an 'exec in', the driver has to wait until at least one byte is received. When using an 'exec early in' it has to respond immediately.

0x101 Datagram

Format:

- O format: ROW string length BYTE string
- I format: INT max string length

A datagram is a string of bytes, which forms a closed semantical object. In contrast to byte strings, datagrams must always be handled as undivisible objects. If a datagram should be read by an 'exec in', the driver has to wait until a datagram is received. When using an 'exec early in' it has to respond immediately.

0x110 RS232 Transfer Mode Specifier

Format:

- OR format: STRUCT (INT baud rate, data bits, stop bits, parity)
- I format: STRUCT ()

The baud rate is specified in bit per seconds. Parity is coded as

0 no parity

1 odd parity

3 even parity

5 high parity

7 low parity

0x111 RS232 Control Line Specifier

Format:

- OR format: STRUCT (INT line mask bits, line value bits)
- I format: STRUCT ()

The line numbering refers to RS232. All 1-bits in *line mask bits AND line value bits* specify lines with 1-polarity, all 1-bits in *line mask bits AND NOT line value bits* specify lines with 0-polarity. All bits not set in line mask bits are undefined (reading) or not affected (writing).

0x112 Break Event

Format:

- OR format: STRUCT ()

No data representation of break event is needed.

0x113 RS232 Output Flow Control Specifier

Format:

- OR format: STRUCT (INT line mask bits, line value bits, xon code, xoff code)

The first two parameters may be used to implement output flow control based on signal lines (CTS, DCD,...). They specify the state of the incoming control lines required for output. The line numbering refers to RS232. All 1-bits in *line mask bits AND line value bits* specify lines with 1-polarity required, all 1-bits in *line mask bits AND NOT line value bits* specify lines with 0-polarity required. Otherwise the output will be blocked.

Xon code and xoff code may be used to implement XON/XOFF or similar output flow protocols. An incoming xoff will block the output channel, while the next incoming xon will unblock it. Initially after sending a flow control specifier the channel will be in the xon state. The value 0 is reserved.

xon code = xoff = 0 specifies "no xon/xoff protocol". If an xon/xoff protocol is switched on, these codes will not be transfered by the corresponding input channel but only to control the output.

0x114 RS232 Input Flow Control Specifier

Format:

- OR format: STRUCT (INT line mask bits, line value bits, buffer size, buffer limit, reserved 1, reserve 2, reserved 3)

The first two parameters specify the state of the incoming control lines required for input. The line numbering refers to RS232. All 1-bits in line mask bits AND line value bits specify lines with 1-polarity required, all 1-bits in line mask bits AND NOT line value bits specify lines with 0-polarity required. Otherwise incoming data will be ignored. The third parameter specifies the size of the driver's input buffer for incoming bytes. 'buffer limit' specifies a limit for collecting input, i. e., if input received, the driver waits 10 ms, until no more input received or buffer limit is reached. Generally the initial value for buffer limit is buffer half. The last 3 parameters are reserved and must be 0 !

0x200 Block Sequence

Format:

- OR format: STRUCT (INT block no, ROW block seq length BYTE blocks)
- I format: STRUCT (INT block no, INT block seq length)

This is the basic data object for all magnetic devices. The length of a block sequence must always be the actual block size (specified in the transfer mode) or a multiple of this value.

0x200 Block Sequence Ptr

Format:

- OI format: STRUCT (INT block no, block seq length, memory address)
- R format: -

This type of object is used for paging within the kernel and the backing store drivers only.

0x210 Floppy Disk Transfer Mode Specifier

Format:

- OR format: STRUCT (INT tracks, sectors per track, sector size)
- I format: STRUCT ()

The sector size is given in bytes.

0x211 Floppy Disk Media Type Specifier

Format:

- OR format: INT media type
- I format: STRUCT ()

0 undefined media

1 3.5“ or 5.25” diskette, double sided, single track

2 3.5“ or 5.25” diskette, double sided, double track

3 5.25“ diskette, double sided, high density ”

4 3.5“ diskette, double sided, high density ”

“Undefined media” in an out operation means: Try to determine the media actually in the floppy disk drive and take it as new media type. If this succeeds, the answer has to be an “ack nil” and the new media type can be requested by an in operation. Else (no readable diskette in the drive) the answer has to be an alternate out request.

0x300 Scan Ascii

Format:

- I format: STRUCT ()
- R format: INT scan ascii

The reply has the following Structure *STRUCT (BYTE ascii, scan, reseved1, reserved2)* [scan ascii]

0x301 Key

Format:

- I format: STRUCT ()
- R format: INT key

The reply has the following Structure STRUCT (BYTE ascii, scan, shift state, extended shift state) [key]

0x302 Shift State

Format:

- I format: STRUCT ()
- R format: INT state

The reply has the following Structure STRUCT (BYTE shift state, extended shift state, reseved1, reserved2) [state]

0x303 Make Break

Format:

- I format: STRUCT ()
- R format: STRUCT (INT make break, INT key)

The reply has the following Structure STRUCT (BYTE make break code, reseved1, reserved2, reserved3) [make break] STRUCT (BYTE ascii, scan, shift state, extended shift state) [key] Every make break code is sent, if it is detected. If no valid scan ascii could be derived from this make break code, the scan byte and the asci byte are set to 0xff.

0x304 Hot Key

Format:

- R format: INT hot key

Signals, that a hot key occured. Remark: This object is only used for internal communication between the terminal process and its manager.

0x305 Diacritical Flag

Format:

- OR format: BOOL flag
- I format: STRUCT ()

Sets a flag in the keyboard manager, which controls the consideration of diacritical chars.

0x400 Alpha8 (Attr, Char String)

Format:

- O format: STRUCT (INT CONST x, y, attr, ROW n BYTE char string)

0x401 Alpha8 (Attr, Char String, glue)

Format:

- O format: STRUCT (INT CONST x, y, attr, ROW n BYTE char string)

0x402 Alpha8 ((Char,Attr) String)

Format:

- O format: STRUCT (INT CONST x, y, ROW n STRUCT (BYTE char,attr) string)

0x403 Alpha8 ((Char,Attr) String, Glue)

Format:

- O format: STRUCT (INT CONST x, y, ROW n STRUCT (BYTE char,attr) string)

0x404 Alpha8 Char Filler

Format:

- O format: STRUCT (INT CONST left upper x, left upper y, width, height, fill char)

The specified rectangle is filled by the fill char without changing the attributes.

0x405 Alpha8 (Char,Attr) Scroller

Format:

- O format: STRUCT (INT CONST left upper x, left upper y, width, height, scroll lines, fill char, fill attr)

A positive scroll line value means "scroll up", a negative one "scroll down", zero has no effect. A scroller can be used to fill a rectangular region by specifying 'scroll lines' large enough.

0x406 Alpha8 Update String

Format:

- O format: STRUCT (INT CONST x, y, attr, ROW n CSCWORD update string)

This is a special object which can only be used in the context of the GDP/CSC. It is described in the CSC chapter.

0x407 Attribute Mode

Format:

- O format: STRUCT (INT CONST mode)

Sets the background colour for the attributes with bit 7. If mode = 0, then background of the attributes with bit 7 is intensiv, and if mode = 1, it is blinking.

0x450 Beep

Format:

- O format: STRUCT ()

Sounds a beep.

0x451 Click

Format:

- O format: STRUCT ()

Sounds a mouse click.

0x452 Tone

Format:

- O format: INT frequency

Turns on the loudspeaker in the given frequency.

0x480 Screen Descriptor

Format:

- O format: STRUCT (INT width, height, char width, char height, mode, INT screen memory size, screen memory address)

Video mode and active page are defined. The alphanumeric modes are characterized by 'char width = char height = 1'. The 'mode' can be

0 even bytes: char, odd bytes: attribute

1 1 bit pixel

2 2 bit pixel

3 2 plane pixel

4 4 plane pixel

0x481 Screen Memory

Format:

- O Format: DATASPACE

Outputting a screen memory includes a complete refresh of the hardware screen from the screen space. From now on all screen modifications are also made within the screen space (usually an alias space).

0x482 Screen Page

Format:

- O format: STRUCT (INT page base address, page size)

0x483 Screen Write Access

Format:

- O format: INT screen write access address

By this a process can signal to the driver that it wants to write to the (aliased) screen space. Usually the driver will protect the displayed part of the screen space from writing. In such a case the user process may request direct write access, if its software wants to modify the screen memory by its own. If the driver acknowledges this request, it has removed the write protection from the addressed 4K page for some milliseconds. Afterwards the driver will analyze the changes by itself and update the hardware screen. (This is a nil action, if the screen space is directly mapped to hardware.)

0x484 Screen Palette Descriptor

Format:

- O format: STRUCT (INT palette no, reg no, reg value)

This is used to defines colour palettes. Available palettes as well as palette registers are numbered from 0. The overscan register (if available) has the reg no -1.

0x485 Screen Palette Selector

Format:

- O format: INT palette no

Selecting the active palette.

0x486 Screen Hardware

Format:

- I format: STRUCT ()
- R format: STRUCT (INT display type, available modes, std mode)

display type:
available modes:

0x487 Screen Plane Selector

Format:

- OR format: INT planes

```
planes (bit 15..8):  select plane for read (0..3)
planes (bit 7..0) :  select planes for write (0..15)
                     IF bit (selected planes for write, n)
                        THEN select plane 'n'
                     FI
```

Remark: This object is only used for internal communication between the display manager of the CONSOLE and its driver.

0x488 Screen EGA Palette Selector

Format:

- O format: INT palette selector

```
palette selector:
      (bit 31..16):     don't care
      (bit 15..8 ):     register to set
                        0..15  <=>   palette register no
                        0x 11  <=>   overscan register
      (bit 7..0 ):      value to set
```

0x489 Screen VGA Color Register

Format:

- O format: STRUCT (INT color to set, color descriptor)

```
color to set:           0..255
color descriptor :
      (bit 31..24):     don't care
      (bit 23..16):     blue value     (0..63)
      (bit 15..8 ):     green value    (0..63)
      (bit 7..0 ):      red value      (0..63)
```

0x48a Screen VGA Register

Format:

- O format: STRUCT (INT mode, ROW n BYTE portvalues)
- I format: INT mode
- R format: ROW n BYTE portvalues

```
mode:  0..3:  read/write  dac-register
          0:    0.. 63    (n = 192)
          1:   64..127    (n = 192)
          2:  128..191    (n = 192)
```

```
 3:  192..255    (n = 192)

4:   read/write  attribut controller registers,
                 graphik controller registers,
                 sequencer registers;
                 (n  =  37)
```

Remark: This object is only used for internal communication between the display manager of the CONSOLE and its driver.

0x48b Screen Activate

Format:

- OR format: BOOL screen on or off
- I format: STRUCT ()

0x490 Window Size

Format:

- OR format: STRUCT (INT x origin, y origin, x size, y size)
- I format: STRUCT ()

0x4A0 Cursor Descriptor

Format:

- O format: STRUCT (INT cursor no, cursor shape, cursor mode)

Cursor number 0 is the alphanumeric cursor, 1 identifies the mouse. The following cursor shapes are defined:

```
0            no visible cursor
n            defined by bitmap n (0 < n < 65536)
0x1ttbb      line cursor (bottom line = bb, top line = tt)
```

The following cursor modes are defined:

```
0            overwriting
1            overwriting, blinking
2            reversing
3            reversing, blinking
```

0x500 Via Port Byte, 0x501 Via Port Word, 0x502 Via Port Dword

Format:

- O format: STRUCT (INT port, INT value)
- R format: STRUCT (INT value)
- I format: STRUCT ()

0x504 Via Port Byte String, 0x505 Via Port Word String, 0x506 Via Port Dword String

Format:

- O format: STRUCT (INT port, ROW string length BYTE/WORD/DWORD)
- R format: STRUCT (ROW string length BYTE/WORD/DWORD)
- I format: STRUCT (INT string length)

7.4 Open And Close

In or out orders containing the nil object have a special meaning. "In nil" is defined as open, "out nil" as close. "Out nil" is also used as interrogation reply to denote a nonexistent operation.

Open Order (In Nil) 0x0008 The open order signals the driver that the ordering task will start from scratch.

Close Order (Out Nil) 0x0009 The driver is enforced to do the necessary close operations. After closing all orders except open have to be rejected by the driver.

Open Requested (Alternate In Requested Nil) 0x0004 By this reply the driver signals to the ordering process that the device is closed and a new open is necessary. Closing can be done by the ordering task (close order) as well by the driver itself. The latter should happen, if the environment changes, e.g. system switched off, terminal switched off, terminal changed, diskette changed... After closing all orders except open have to be rejected by the driver.

Close Requested (Alternate Out Requested Nil) 0x0005 This operation has to be proposed by the driver, if an order or interrogation is not executable and there is no real executable alternative at all.

7.5 Drivers and Process Linking

Most drivers will obey the Link Protocol (see ???) to prevent against unautorized access. The drivers reactions in this context are defined by the following rules:

1. The driver starts without a linked partner.
2. When receiving a link message from the supervisor, usually the driver accepts the process specified by this message as its new 'linked partner'. If the driver wants to reject this partner, it will immediately send an unlink message to the supervisor.
3. When receiving an unlink message from the supervisor, the driver sets its 'linked partner' to nil process.
4. If an unlink message or a new link message comes from the supervisor (and will be accepted by the driver) while a reply to a GDP order of the old partner is yet pending, this operation has to be aborted and an 'invalid partner' message has to be sent to the old partner.
5. Whenever a GDP message comes from a different process than the driver's 'linked partner', the drived will skip the received order and response with an 'invalid partner' message.

Remark: In most cases a new partner will require a new open operation.

On the other side the processes calling the driver (e.g. as sysin or sysout partner) should use the following algorithm:

```
call (partner, gdp order, result, reply) ;
WHILE result = prc not existing COR reply = invalid partner REP
      partner := partner (link no) ;
      call (partner, gdp order, result, reply)
PER
```

Of course 'partner' should be initialized to 'nilprc'.

Chapter 8

Libraries

There are some libraries implemented to make the live of the programmer a little bit easier. These libraries are not directly associated with system functionality like system calls, therefore they are listed in a seperate chapter.

The mentioned libraries handle

- Terminal IO (in a very simple manner),
- Debug output to the associated debug screen,
- Some debugging support like stack checking and a core dump facility.

8.1 L3 Terminal Handling

The L3 terminal functionality is provided at a very low level. You can write a number of arbitrary characters to the screen and you can read a distinct number of characters from the keyboard. This is done by two functions,

- TermWrite(char *text, int count);
- TermRead(char *text, int max);
- TermPrint(char *text) (#define TermPrint(x) TermWrite((x), _strlen(x)))

These functions handle the protcol used to access the terminal. If you attach your task to the terminal, the terminal manager tries to communicate with your task to establish the terminal connection. Your task has to respond with a given sequence. If it doesn't do so, the terminal manager rejects the requested connection. Therefore you have to call `TermRead` or `TermWrite` before trying to attach your task to the terminal.

A simple *Hello World* program could look like follows.

```
#include <l3/l3term>

void main(void)
{
   TermPrint("Hello, World!\n");
}
```

8.2 Debug Output

There is a special task responsible for managing the debug terminal. The library function `void HercPrint(char *text);` provides access to this task. To use this function you have to do two things:

- Enable debug output

  Normally the responsible task doesn't put anything to the debug terminal. To to enable this function you execute the following steps:

  1. enter the kernel using *Scroll Lock*
  2. press *T+* to enable tracing
  3. press *g* to leave the kernel debugger

- Initialize debug output

  A call to `HercInit()` will initalize the debug output. It will mainly ask the kernel for the thread id of the log task, which is responsible for the debug terminal.

A simple *Hello World* program could look like follows.

```
#include <l3/herclog.h>

void main(void)
{
   HercInit();
   HercPrint("Hello, World!\n");
}
```

The log task doesn't understand a normal `'\n'` so you have to use `''\n\r''`.

8.3 Server Library

There is one library providing support for *"server"* development, but it isn't restricted to this area. This library provides:

- default exception handling routines

  L3 mirrors the processor exception, therefore Every thread has to do its own exception handling. The default exception handling routines try to contact a distinct task given at initialization time (default *"core"*) and provide it with the necessary information to create a core task. If a thread tries to create a core dump it puts a message to the debug screen and waits for the core task. You switch to the core task and issue a ccd(/"<name of thread trying to create a core dump>"). After a successful dump you will find a file named <name of core dumped tread>.core.

- stack check

  The libary will set up a fire wall at the bottom of the stack resulting in a stack exception if the stack overflows.

- *"code check"*

  The library will set the code segment to read only to allow detection of write access to the code segment. The library creates a core dump if this happens.

- argument handling

  The library provides the normal calling convention for main, `main(int argc, char **argv)`. It take the parameters given by runx, prepares a character array and calls main.

Normaly it is enough to call `DefaultInit()` to do all the necessary initialization. `DefaultInit()` does the following things:

- Initialize Supervisor Protocol (call `SVProtInit()`)
- Initialize Debug Output (call `HercInit()`)
- Initialize default exception handling

- Initialize core dump (call `CoreInit(‘‘core’’)`)
- Initialize stack and code check (call `InitStackCheck()` and `SetTextReadOnly()`)
- Set up `MainThreadIDV2`, a variable used by l3term

If this doesn't meet your requirements, you have to do the initialization by yourself.

Chapter 9

Kernel Debugger

under construction

This chapter tries to describe the kernel debugger functionality and interface.

9.1 Entering and Leaving the Kernel Debugger

There are several methods to enter kernel debugger. You can

- execute a L3 system call (SysDebug)
- raise a kernel error
- press scroll lock on keyboard

For the last method you have to have a debug version of the L3 kernel.

You can leave the kernel debugger by pressing 'g' (go).

9.2 Entering Commands

The kernel debugger provides several commands. They usually consist of a letter and some additional parameter. If a command consists of only one letter, the command is executed by pressing the letter on the keyboard.

If there are parameters for the command, you have to type the full number of characters, space or enter to execute the command. For instance the command 'show thread control block' (t) can have no parameter and a thread number. The thread number can be a short one or a long one. So depending on the parameter you would have to type:

- no parameter: t[<space> — <enter>]
- a thread number: t5[<space> — <enter>]
- a long thread number: t12345

9.3 Tracing some Kernel Information

The kernel provides mechanisms for continously displaying of kernel events like page faults, messages, idle times and other things.

L+/L- Enable/Disable autolog. This funtion provides continous information about kernel events on a statistical base.

P+/P- Enable/Disable page fault display. This function provides a continous display of page faults.

T+/T- Enable/Disable tracing. (I don't know, what this function really does, but you have to enable tracing to send output to the hercules screen.)

9.4 Information about kernel modules and addresses

The kernel debugger provides information about the L3 kernel modules, versions and associated addresses. You can use this information to set breakpoint base adresses or to find the associated address and module for a given address.

a Show all modules with name, offset and version. The output has the format:

<offset> <modul name> <date>,<version>

a<address> Look for module and the associated address within that modul for the given address <address>. The output has the format:

<offset> <modul name> <date>,<version>

9.5 Breakpoints

The kernel debugger allows to set *one* (only one) breakpoint to an arbitrary address. There are three different kinds of breakpoints,

- instruction breakpoints
- write access breakpoints
- (read/write) access breakpoints

You can combine a breakpoint with an additional condition or restrict it to a distinct thread.

For ease of use you can use breakpoint offsets to set a breakpoint within a module. If there is an offset in use, it will be displayed while entering a breakpoint.

b<enter> Show current breakpoint if there is one.

bb<break point address> Set breakpoint base. This address serves as a base for all following breakpoints and allows easier handling of breakpoints within kernel modules. If you want to set a breakpoint within the kernel, you have to add 0xf0000000 to the output of command *m* or an address of kernel listings.

bi<breakpoint address> Set an instruction breakpoint. The breakpoint is triggered before the instruction is executed.

bw, ba Set a write/access breakpoint. The breakpoint is triggered after the instruction is executed. Breakpoints are attributed by size (1 (byte), 2 (word), 4 (dword)). The default size is byte.

You enter a breakpoint like follows:

ba/bw[<size> — <space>]*at: [<breakpunkt base>+]*<breakpoint address>

b- Disable Breakpoint.

bt<thread id> Restrict breakpoint to a distinct thread with id <thread id>. To remove restriction enter *bt0*.

bx Add a condition to a breakpoint. The breakpoint is only triggered if the condition is true.

bx<size>monit:<address><space>[<value1>,<value2><space>]

Conditions are related to contents of memory allowing monitoring of a distinct address. They are attributed with size. The breakpoint is triggered if the memory contains a value within or out of a specified range.

It depends on the values given to *bx* if the meaning is within or out of.

value1 < value2 The breakpoint is triggered if for the content of memory holds

<value1> <= <content of memory> <= <value2>.

value1 > value2 The breakpoint is triggered if for the contents of memory holds

<value2> <= <content of memory> —— <content of memory> <= <value1>

9.6 Informationen about threads (tcb's, register contents, stack contents, ...)

The kernel debugger provides mechanisms to examine the state of threads. It supplies information like thread state, contents of wait queues, register contents, stack contents and more.

While visiting the kernel stack you can change the visible area by virtally changing the esp register using cursor keys. You can virtually set register contents to the state just before kernel entry using the <enter> key. The debugger executes a popa.

t Show TCB of current thread.

t<thread id> Show TCB of thread <thread id>

9.7 Inspecting Mappings

To fill up its own virtual memory a task has to map dataspaces. You can inspect these mappings using the kernel debugger. There should be at least three mappings, the mapping of the standard dataspace and two mappings supplied by the kernel.

m Show mappings of task of current thread. Thsi command provides information about the memory mode of the task, about io permissions and current mappings. The display looks like follows:

```
task:<task no>             (<number> Threads, Root:<i don't know>)
mode:<mem mode, io mode>                             Limit:<number>
                           maphw base:<number>
                           hwmap max :<number>  wrlck max:<number>
                           prsent lnk:
regions

[number] : [<start>,<end>] <map mode> ds:<number>,<number>,<number> mvec:<address>
...
```

Values of entries:

mem mode Memory mode can have the following values:

- resident
- pageable

io mode Every task has its own io priviledges. That can be:

- unlimited io
- <no entry> - no io priviledges

map mode Dataspaces can be mapped with two different modes, read write and read only. There is an additional mode, which is specified for kernel supplied mappings. There are no map vectors for standard dataspace and kernel supplied mappings.

- rw - read write
- ro - read only
- XX - kernel area

m<thread id> Show mappings of task of thread <thread id>.

Cursor Keys Scroll through mappings.

9.8 Inspecting Memory

There are several mechanisms to inspect memory. You can inspect address spaces, search for hex and ascii strings, change memory contents and calculate the associated physical address for a given virtual address.

There are two major modes for inspecting memory, disabled task switching and enabled task switching. While task switching is disabled you can only inspect virtual addresses, which are currently in physical memory. To inspect any virtual address or any dataspace you have to enable task switching to allow the pager to transfer the necessary information from disk to memory.

i/I Enter inspecting memory. Using i you enter memory inspection with disabled task switching, using I with enabled task switching.

Commands for inspecting memory:

- **i** Specify a thread id. Format: <thread id>i<enter>
- **l** Specify a length to be displayed. Format: <length>l<enter>
- **p** Specify an address display starts with. Format: <address>p<enter>
- **<enter>** Redisplay block.
- **+** Go forward in steps of length l.
- **-** Go backward in steps of length l.
- **:** Set base address. Format: <address>:<enter>
- **Cursor Keys** Move within the displayed block. Once you have reached the desired position you can change contents of memory entering arbitrary values. By pressing <enter> you go back to normal memory inspection.
- **x** Search for memory contents. You can search for hex or asci strings.
  - **xh<hex value>{<space><hex value>}*<enter>** Search for hex string <hex value>{<space><hex value>}*
  - **xc<ascii string><enter>** Search for ascii string.
  - **<enter>** Repeat last search operation.
- **T** Invoke nested kernel debugger, for instance to inspect a tcb without leaving memory inspection. You can leave the nested kernel debug using key *h*.
- **t** Invoke spvertest. You can leave spvertest with the key *h*.

Additional commands if task switch is enabled (**I**).

- **s** Specify a dataspace id. Format: <dataspace id>s<enter>
- **k** Load a block from backing storage. ???
- **m** Write a block to backing storage ???
- **q** Access task root ???

Chapter 10

Development for L3/L4 (under construction)

10.1 Programming Style

10.1.1 Names and Identifier

If you choose names for your modules, identifiers, etc, try to use meaningful names. To achive a consistency in use of names, we encourage you to follow the following rules.

The name of a module or compilation unit is created like this:

- the name of a unit in C is: Interfacename_Functionname
- the name of a unit in C++ is: Classname_Funktionname

If you have to create name for an identifier, you should use meaningful names. To seperate words in a name, start every single word with a capital letter, for instance SetAccessRight.

To distinguish between different types of names, use the following suffixes:

C for a constant, for example `const FamDridC=234;`

T for a type, for example `typedef unsigned int DataSpaceT;`

T for a class, for example `class DisplayT;`

Ptr for a pointer, for example `typedef int *NumberPtrT;`

Variables of minor importance may be short and are written in lower cases.

If there are conflicts of names, add an additional suffix to your name to resolve the conflict. Use:

P for a parameter of a function, for example `void Fkt(DummyT TestP);`

V for a local variable, for example `DummyT CounterV;`

G for a global variable, for example `DummyT SemaG;`

Use L3 specific prefixes to avoid pullution of name space, for instance: *L3_SV_ACK (l3, sv-protokoll, acknowledge)*. Maybe we should focus on L3_.

Avoid the use of the following pre/suffixes. They are reserved for libc. (If we use L3_ as our prefix, there should be no problem.)

pre_/_suffix header file using name pre_/_suffix(is, to ctype.h)d_ dirent.hE errno.hL_, F_, O_, S_ fcnstl.h_MAX limits.hLC_ locale.hpw_ pwd.hsa_, SA_ signal.hmem, str, wcs string.hst_, S_ stat.htms_ times.hc_, V, I, O, T, C termios.h_t type definitions

10.1.2 Conventions for function declaration

If you declare a function, write a short description of your function above your declaration. Use the following structure for this description:

```
/*
  * Description of parameter
  * (optional) pre- and postcondition
  * (optional) Errorcodes
  */
```

If you want to declare a function, that is used to assert a structure, use the following conventions for nameing assert functions:

- **pre_assert** for precondition
- **post_assert** for postcondition

There should be no side effect in your assert function.

10.1.3 Headers

In header files you have to cope with two problem:

- multiple inclusion of headers
- force the user to use the appropriate compiler
- cope with name mangeling of C++ compilers

Use the following constructs to cope with these problems:

1. Avoiding problems with multiple inclusision of header files

   Use the folling construct:

   ```
   #if !defined _headerfilename without extension_H
   #define _headerfilename without extension_H
   /* body of header */
   #endif
   ```

2. Force the user to use the appropriate compiler

   Use the folling construct:

   ```
   #if !defined __cplusplus
   #error C++ compiler required
   #endif
   ```

3. Cope with name mangeling of C++ compilers

   If you write a library, that shall be used with C and C++, you have to cope with name mangeling. A C++ compiler appends type information to function names to allow type safe linking. You have to switch name mangeling off using the folling link directive:

```
#if defined __cplusplus

extern ''C'' {

#endif

/* body of header */

#if defined __cplusplus

extern ''C'' {

#endif
```

10.1.4 Encapsulation of debug code

We enforce you to use

```
#ifndef NDEBUG

/* debug code */

#endif /* NDEBUG */
```

Debug code must be read only!

to encapsulate debug code and to allow generation of debug free code.

10.1.5 CVS related things

Every Module should contain a line describing the version and date of last change, user etc. You can achive this by inserting the folling lines into your file:

- in a header file: `/* $Id:  develop.sgml,v 1.4 1996/03/21 09:46:14 jw5 Exp $ */`
- in a C/C++ file: `static char rcs_id=$id$;`

The information is then automagically written to your file every time it changes.

10.1.6 Library depencies

There should be no circular depency between libraries. For instance if you use only functions of libl3sys.a, you need only libl3util.a and libl3sys.a to generate you programm. There should be no linker sequence like: `-L l3/l3/lib -ll3util -ll3prot -ll3sys -ll3util`

10.1.7 Miscellaneous Conventions

- Indentation as standard in emacs ("{" on a new line)
- Use // for linecomment in C++
- Error values:

– No Error: 0
– Error : != 0

10.2 Tools (under construction)

The development for our project is done on various unix plattforms, for instance ultrix, freebsd and linux. We are using tools which are available for free, so that everyone wanting to take part in our project can do so. Another advantage is the possibility to set up a cross compile environment for these tools. Your os assignement will be done on the same base.

The development for our project is done in a combination of C, C++ and assemby language. We have choosen this languages because of they are highly available and they are the languages typically used for os projects.

This chapter contains a short description of the tools we use. If you want to know more about it, you will need to have a look at the mentioned references for each tool.

10.2.1 gcc, g++

The C/C++ compiler we use is gcc. It is used to compile C/C++ code to assembly language or objectcode or to generate complete programms using several other tools like ld. The main options used to invoke gcc are:

-o specify the name of the output file

-c compile only, don't call ld

-S compile to assembly language (useful to check code generated for asm statement)

-E stop after preprocessing (useful to check macro expansion)

-Wall enable all (nearly all) warnings

-Wstrict-prototyps warn if a function is declared or defined without a prototype

-Wmissing-prototype warn if a global function is defined without a prototype

-L specify where to look for libraries

-l specify library to link with

-I specify where to look for include files

For a complete documentation of gcc and especially the options of gcc look at the appropriate info page. Aditionally there is a chapter coping with the asm statement in the l3-documentation.

10.2.2 ld

The GNU linker ld is used to combine a number of object and archive files to create an executable file. It is normaly invoked as the last step in program generation. The main options are:

-o specify the name of the output file

-L specify where to look for libraries

-l specify library to link with

-e specify the entry point of the executable programm (usually you have to preceed a C function with an underscore)

-static generate a statically linked binary

10.2.3 ar

10.2.4 make

10.2.5 objdump

10.2.6 nm

10.2.7 cvs

To allow more then one person to work on our project we use a revision control system named *cvs*. In contrast to other systems it doesn't use file locking. The first developer committing a change in his file doesn't anything about changes other developers have made locally. If one of these other developers tries to check in his changes he will get an error message and has to use cvs commands to bring in his changes.

For a short walk through have a look at the sample session in the info page of cvs.

The most interesting cvs commands are:

checkout The very first thing you have to do is creating of your own working copy of the repository. For this, you use the checkout command:

```
cvs checkout l3
```

This will create a directory named l3 containing your copy of the source files.

add If you create a new file belonging to the project you have to add it to the repository. You use the `add` command of cvs to do so.

```
cvs add foo.c
```

This will add the file foo.c to the repository. But you have to commit the file to allow other programmers to work with the newly added file.

update If other developers have made changes to files in the repository you have to incorporate these changes into your own working copy. Cvs tries to detect differences between the repository and your working copy and to merge in the changes into your file to provide an up to date version of the project. You can do this using `update`.

```
cvs update
```

If you have changed the file locally it is possible, that there are conflicts between your version and the repository version. Cvs will detect this and generate a file containing the conflicting parts of both versions. You have to clean up the conficts before checking your new version.

commit To add your changes to the repository you have to `commit` your changes using

```
cvs commit
```

Cvs tries to integrate your changes and will abort commiting if there are conflicts. If there are conflicts you have to resolve it before trying to commit again.

diff To check out differences between a locally changed file and the repository or two versions in the repository you can use the `diff` command:

```
cvs diff foo.c
```

to get a diff between the last and you current version or

```
cvs diff -r1.11 -r1.12 foo.c
```

to get a diff between two versions.

log To get information about the history of a repository file you can use the command `log`. It provides the history of changes including the commit messages.

```
cvs log svprot.c

RCS file:
Working file: svprot.c
```

```
head: 1.17
branch:
locks: strict
access list:
symbolic names:
description:

revision 1.17
date: 1996/01/25 14:08:35;  author: mh1;  state: Exp;  lines: +8 -9
SendSVMsg(): specify all scratch registers to the asm() statement
instead of trying to safe some of them ourselves; also, use a local
symbol ("0:") instead of a named symbol in order to avoid clashes with
the compiler's symbol name space

...
```

status To find out the current state of a file you can use `status`. It figures out the current state and displays one of the folling states:

- Up to date
  The file is identical with the latest revision in the repository.
- Locally modified
  You have edited the file, and not yet committed your changes.
- Needing update
  Someone else has committed a newer revision to the repository.
- Needing merge
  Someone else have committed a newer revision to the repository, and you have also made modifications to the file.

```
$ cvs status svprot.c
File: svprot.c          Status: Needs Merge
   Working revision:    1.14
   Repository revision: 1.17
   Sticky Tag:          (none)
   Sticky Date:         (none)
   Sticky Options:      (none)
```

10.2.8 emacs

10.2.9 elan scripts

There are several (two at the moment) useful scripts written in ELAN, the built in language of L3. They provide some useful facilities for programmers allowing an easier development for L3.

runx

Runx is a collection of routines providing methods for handling a.out binaries. You can run binaries, create special dataspace used to create driver tasks at startup an similar things. Before using the scripts you have to insert them in the environment of your task using the L3 command `insert(''<file name>'')`. The following part presents a list of available routines, their parameters and an example for their usage.

runx(file name[, task name [,param string])] Runx creates a task using an a.out binary. If you provide a *task name*, the new task will be named *task name* otherwise the name of the binary is taken to name the new task. If a task with the same name already exists runx will fail. If you provide a *param string* runx will supply this string for the newly created task. In conjunction with the startup code supplied by libl3serv.a you can use the normal C declaration `main(int argc, char **argv)` for the main function, otherwise you don't have any parameters for the main function.

runxp(file name, param string) This function is does the same like runx. It is only provided to allow a *file name* and a *param string* as parameter which wouldn't be possible with runx.

dumpx(file name) Dumpx provides some useful information about a.out binaries, like segment sizes, start addresses and other useful things.

xtods(file name [, dataspace name)] Xtods creates a special dataspace used by L3 to create driver tasks. It contains all necessary information (stack pointer, program entry point) allowing th *hardware configurator* to run the driver at L3 startup. All you have to do is to create a subdirectory `''/driver/<driver name>''` within the hardware configurator, copy the dataspace to this directory using the name *stdds* and reboot L3. If the system comes up again there will be a new task named `''<driver name>''`. This will be your new driver.

sethw([file name, io base, io extend, irq)] Drivers need access to special resources like interrupt lines and io space. You have to assign these parameters to the dataspace of your driver before copying it to the hardware configurator. L3 will check for conflicts and deny the operation if there is one.

core dump facility

To allow a minimal amount of debugging there is a core dump facility consisting of an ELAN script and some procedures within libl3serv. If an exception occurs the library catches it and tries to create a core dump. To do this it waits for a request from a distinct task. You can specify this task at initialization time of the library (`InitCore(task name)`). After receiving the request the library replies with information necessary to create the core dump, for instance register and memory contents. The request is sent after starting a script called `create core dump(task name)` and if the reply comes in the script generates a core dump named `<task name>.core`. Any existing core file with the same name will be overwritten.

10.2.10 How to build an run a program using these tools

There are several steps involved in making and running a L3 programs. This section tries to explain these step using a simple example and a stepwise walk through. The steps are:

- Writing a Makefile
- Writing your program
- Compiling and linking your program
- Transferring your program to L3
- Creating and preparing a L3 task
- Running your program
- Creating a core dump
- Analyse a core dump

Writing a Makefile

If you write a program consisting of more than one source and header file, it seems to be a good idea to use *make* to build your libraries and executable image. Make automates the process of building your library or executable and tries to find the minimal set of operations necessary to do so. A short introduction (and a lot of more information) can be found in the make *info* file.

A make file in general consists of rules and targets. "A 'rule' appears in the makefile and says when and how to remake certain files, called the rule's 'targets' (most often only one per rule). It lists the other files that are the 'dependencies' of the target, and"commands“ to use to create or update the target" *[make info]* .

Assuming our program consists of a main program and several L3 libraries our Makefile could look like follows.

```
#                            -*-makefile-*-
CPPFLAGS=       -I../include  \
                -I../../../../l3/lib/libl3prot/include \
                -I../../../../l3/lib/libl3sys/include \
                -I../../../../l3/lib/libl3util/include \

include ../../../../Makeconf

HELLO_SRCS=     hello.c
HELLO_OBJS=     $(HELLO_SRCS:.c=.o)

HELLO_LDFLAGS=  -static -g -L ../.. -L ../../../../l3/lib ../../servcrt0.o
HELLO_LIBS=     -lserv -ll3prot -ll3sys -ll3util

hello:  $(HELLO_OBJS)
        $(LD)   -o hello $(HELLO_LDFLAGS) $(HELLO_OBJS) $(HELLO_LIBS)

clean::
        $(RM) -rf $(HELLO_OBJS) *~

include $(HELLO_SRCS:.c=.d)
```

The very first line is a comment telling emacs that this file is a make file. Emacs switches to a special mode in this case allowing a more comfortable editing.

The second line defines a *variable* .

Variables are used to simplify make files and are referenced using the character `'$'`. This variable is a special variable and contains flags for the C compiler. In this case it is set to the include directory allowing the compiler to find specified include files. The next line will add some more switches to this variable. It includes a file named `Makeconf` containing some general rules like how to create object files, dependancies and some other things.

The third line specifies a set of C files which are necessary for our executable. In this case it is only one file, but normally you will have several files. The next line takes these names and replaces the .c at the end of the name with a '.o' resulting in a variable which contains all necessary object files.

The next lines contain some more definitions. The only option worth to diskuss is `-static`. You have to specify *static* to tell the linker that it should generate a static linked image rather then a dynamic linked one. This is necessary because of our current ELAN scripts can't handle shared libraries.

And finally there is our first goal named *hello*. It is the deault goal if there is no parameter givven to *make*. The target has one depency, $(HELLO_OBJS) meaning that it has to be rebuild if the target doesn't exists or one of the object files is changed. In that case the next line is executed calling ld to generate the executable image. If there is an object file missing it is build using de fault rules specified by special *built in rules* or rules from *Makeconf*.

The include statement at the end of our make file includes some '.d' files. There is a rule telling make how to create these files. It will call the compiler with a special option causing the compiler to generate a list of all files the current file depends on. The output is redirected to the '.d' file and included. This allows us to rebuild our programm if some include files have changed.

Last but not least an example for building a library. Assuming there is a variable called LIBOBJS containing all necessary objects to build the library. Then the following target produces a library with the same name like the target.

```
libserv.a:  $(LIBOBJS)
$(AR) rv $@ $^
```

```
$(RANLIB) $@
```

In this example the variables '@' and '^' are a variables defined by make. '@' contains the name of the target and '^' contains the list of dependencies. So make would call `ar` and `ranlib` to generate the library.

If you need more informations there are a lot of make files in the source tree of our L3 project and there is the *gnu info file* on gnu make.

Writing your program

I think, there is not much to say about this. You have to have a problem, some ideas about a solution and some skills in programming C/C++. Just to show, how a short L3 program could look like, I will add a (very) small and simple program. The only thing it will do is to create a new task, send a request and wait for an answer. The newly created task in turn will wait for a request and send an answer.

```
#include <l3/util.h>
#include <l3/ipc.h>
#include <l3/syscalls.h>
#include <l3/l3term.h>
#include <l3/herclog.h>

#include <l3/l3term.h>
#include <l3/svprot.h>

#include <l3/servutil.h>
#include <l3/definthandler.h>

extern int _astack;

struct {
  MsgDopeT Size, Msg;
  int d1, d2, d3;
} Msg;

int TermPrint(char *text);
int TermPrint(char *text)
{
  return TERMwrite(text, _strlen(text));
}

void TestLoop(void)
{
  int ret;
  ThreadT Client;

  DefaultInit();
  HercInit();

  for(;;)
    {
      Msg.Size.DWord = MSGDOPE(1,0,0,0);
      LogStr(MySelfID(), "Waiting for Order");
      ret = Wait(&Client, &Msg, NEVER);
      if (IPCError(ret))
{
  LogRet(MySelfID(), "IPCError ", ret);
}
```

```
      while (!IPCError(ret))
{
  LogStr(MySelfID(), "Order received");
  Msg.Size.DWord = Msg.Msg.DWord = MSGDOPE(1,0,0,0);
  LogStr(MySelfID(), "Sending Reply");
  ret = ReplyAndWait(Client, &Msg, &Client, &Msg, 0, NEVER);
  if (IPCError(ret))
    LogRet(MySelfID(), "IPCError in ReplyAndWait", ret);
}
    }
}
void main(int argc, char **argv)
{
  char buffer[180];
  int result, i;
  DataSpaceT stdds;
  ThreadT Son;

  DefaultInit();
  HercInit();

  CopyStdds(&stdds);
  result = SVBeginTask("SonTask", 0, (int)TestLoop, _astack,
        stdds, &Son);
  if (result == L3_SV_ACK)
    {
      Msg.Size.DWord = Msg.Msg.DWord = MSGDOPE(1,0,0,0);
      Msg.d1 = Msg.d2 = Msg.d3 = 0xaffe;

      for (i=0; i<5; i++)
{
  LogStr(MySelfID(), "Calling Son");
  result = Call(Son, &Msg, &Msg, NEVER,NEVER);
  if (IPCError(result))
    {
      LogRet(MySelfID(), "IPCError in Call", result);
    }
  else
    {
      LogStr(MySelfID(), "IPC call ok");
    }
}
    }
}
```

Compiling and linking your program

The easiest way to compile and link your program is to write a good Makefile as explained earlier. If you have one the only thing you have to do is to call `make -k`. Make will execute the necessary things to build the requested target and will either finish with a successfully build target or with a bunch of error messages.

If you decide to compile you program without using make have a look at the description of the necessary tools. *(to be continued)*

Transferring your program to L3

After building your image you have to transfer it to L3 to be able to run and test it. There are several posibilities to do that depending on the configuration you are using. You can transfer your program using:

the network If your development plattform is connected to a L3 station via network you can use ftp to transfer your program to L3. Normally every L3 station configured for network use has a task named anonymous which is accessible through ftp. You connect to the L3 station and use the task name (in this case 'anonymous') as user name. There is normally no station password set so you kann simply enter anything you want. After the login procedure you can use the normal ftp command to transfer your images (don't forget to switch to binary transfer mode befor transferring binary images).

the floppy disks You can simply copy the image to a floppy disk formatted for MSDOS. After that you put the floppy into the L3 station, open `ARCHIVE A` and copy the image into the destination task.

a available MSDOS partition on your hard disk You can copy the image to your DOS partition, boot L3, open `ARCHIVE C` and copy it to your destination task.

Creating and preparing a L3 task

In L3 every task manages its own file system, so you have to create a task, copy the necessary files to this task and test your program.

To create a task you activate the *session manager (????)* using <ALT> <PRTSCR> go to `OBJ ROOT` (or any other task) and press <ALT> <E>. L3 will request a task name, create the new task and activate it.

Now you have a new task with nothing in it (except some system files necessary for L3DOS). You have to transfer your image and two other files to this task using ftp or disk and the L3 copy command. The 'other' files are *runx* and *core.elan*. These two files provide the functionality described earlier. To be able to use the functionality you have to insert it in your environment using the L3 command `insert` (`insert(''runx'');insert(''core.elan'')`).

Running your program

- runx("image name","task name")

Creating a core dump

- ccd("name of task trying to create a core dump");

Analyse a core dump

- gdb image_name core_dump_name

10.3 Inline Assembler

In some situations we need a possibilty to generate a distinct sequence of assembler code, for instance to provide an interface for L3 system calls or to access registers or memory in a direct way.

It is obvious that you have to use assembler to provide kernel interfaces. But why should a normal application use assembler directly?

Consider a pager which need to page in a distinct page provided by another pager. The most obvious way to do this is to access the page with the appropriate operation (read operation for read access and write operation for write access). But it has to do in without changing the contents of that page, therefore it has to do something like:

```
/*
```

```
* execute a write access to a page <page> to ensure the page is
* available for write
*/
*page = *page + 0;
```

But nearly every compiler will throw away this statement because it does nothing. So the only way is to use assembly language. The following provides an introduction to inline assembler in gcc (excerpt from info). For a more detailed documentation look at gcc.info and gas.info.

10.3.1 Syntax

Currently we develop for the intel platform, so you probably expect an assembler which uses intel syntax. The code to access the page would look like:

```
push   eax
mov    eax,    _page_address
add    [eax],  0
```

But for several reasons that's not true. The following sections provide the intel specific information based on the gas info files.

AT&T Syntax versus Intel Syntax

In order to maintain compatibility with the output of 'gcc', 'as' supports AT&T System V/386 assembler syntax. This is quite different from Intel syntax. We mention these differences because almost all 80386 documents used only Intel syntax. Notable differences between the two syntaxes are:

- AT&T immediate operands are preceded by '$'; Intel immediate operands are undelimited (Intel 'push 4' is AT&T 'pushl $4'). AT&T register operands are preceded by '%'; Intel register operands are undelimited. AT&T absolute (as opposed to PC relative) jump/call operands are prefixed by '*'; they are undelimited in Intel syntax.
- AT&T and Intel syntax use the opposite order for source and destination operands. Intel 'add eax, 4' is 'addl $4, %eax'. The 'source, dest' convention is maintained for compatibility with previous Unix assemblers.
- In AT&T syntax the size of memory operands is determined from the last character of the opcode name. Opcode suffixes of 'b', 'w', and 'l' specify byte (8-bit), word (16-bit), and long (32-bit) memory references. Intel syntax accomplishes this by prefixes memory operands (*not* the opcodes themselves) with 'byte ptr', 'word ptr', and 'dword ptr'. Thus, Intel 'mov al, byte ptr FOO' is 'movb FOO, %al' in AT&T syntax.
- Immediate form long jumps and calls are 'lcall/ljmp $SECTION, $OFFSET' in AT&T syntax; the Intel syntax is 'call/jmp far SECTION:OFFSET'. Also, the far return instruction is 'lret $STACK-ADJUST' in AT&T syntax; Intel syntax is 'ret far STACK-ADJUST'.
- The AT&T assembler does not provide support for multiple section programs. Unix style systems expect all programs to be single sections.

Opcode Naming

Opcode names are suffixed with one character modifiers which specify the size of operands. The letters 'b', 'w', and 'l' specify byte, word, and long operands. If no suffix is specified by an instruction and it contains no memory operands then 'as' tries to fill in the missing suffix based on the destination register operand (the last one by convention). Thus, 'mov %ax, %bx' is equivalent to 'movw %ax, %bx'; also, 'mov $1, %bx' is equivalent to 'movw $1, %bx'. Note that this is incompatible with the AT&T Unix assembler which assumes that a missing opcode suffix implies long operand size. (This incompatibility does not affect compiler output since compilers always explicitly specify the opcode suffix.)

Almost all opcodes have the same names in AT&T and Intel format. There are a few exceptions. The sign extend and zero extend instructions need two sizes to specify them. They need a size to sign/zero extend *from* and a size to zero extend *to*. This is accomplished by using two opcode suffixes in AT&T syntax. Base names for sign extend and zero extend are 'movs...' and 'movz...' in AT&T syntax ('movsx' and 'movzx' in Intel syntax). The opcode suffixes are tacked on to this base name, the *from* suffix before the *to* suffix. Thus, 'movsbl %al, %edx' is AT&T syntax for "move sign extend *from* %al *to* %edx." Possible suffixes, thus, are 'bl' (from byte to long), 'bw' (from byte to word), and 'wl' (from word to long).

The Intel-syntax conversion instructions

- 'cbw' – sign-extend byte in '%al' to word in '%ax',
- 'cwde' – sign-extend word in '%ax' to long in '%eax',
- 'cwd' – sign-extend word in '%ax' to long in '%dx:%ax',
- 'cdq' – sign-extend dword in '%eax' to quad in '%edx:%eax',

are called 'cbtw', 'cwtl', 'cwtd', and 'cltd' in AT&T naming. 'as' accepts either naming for these instructions.

Far call/jump instructions are 'lcall' and 'ljmp' in AT&T syntax, but are 'call far' and 'jump far' in Intel convention.

Register Naming

Register operands are always prefixes with '%'. The 80386 registers consist of

- the 8 32-bit registers '%eax' (the accumulator), '%ebx', '%ecx', '%edx', '%edi', '%esi', '%ebp' (the frame pointer), and '%esp' (the stack pointer).
- the 8 16-bit low-ends of these: '%ax', '%bx', '%cx', '%dx', '%di', '%si', '%bp', and '%sp'.
- the 8 8-bit registers: '%ah', '%al', '%bh', '%bl', '%ch', '%cl', '%dh', and '%dl' (These are the high-bytes and low-bytes of '%ax', '%bx', '%cx', and '%dx')
- the 6 section registers '%cs' (code section), '%ds' (data section), '%ss' (stack section), '%es', '%fs', and '%gs'.
- the 3 processor control registers '%cr0', '%cr2', and '%cr3'.
- the 6 debug registers '%db0', '%db1', '%db2', '%db3', '%db6', and '%db7'.
- the 2 test registers '%tr6' and '%tr7'.
- the 8 floating point register stack '%st' or equivalently '%st(0)', '%st(1)', '%st(2)', '%st(3)', '%st(4)', '%st(5)', '%st(6)', and '%st(7)'.

Opcode Prefixes

Opcode prefixes are used to modify the following opcode. They are used to repeat string instructions, to provide section overrides, to perform bus lock operations, and to give operand and address size (16-bit operands are specified in an instruction by prefixing what would normally be 32-bit operands with a "operand size" opcode prefix). Opcode prefixes are usually given as single-line instructions with no operands, and must directly precede the instruction they act upon. For example, the 'scas' (scan string) instruction is repeated with:

```
repne
scas
```

Here is a list of opcode prefixes:

- Section override prefixes 'cs', 'ds', 'ss', 'es', 'fs', 'gs'. These are automatically added by specifying using the SECTION:MEMORY-OPERAND form for memory references.

- Operand/Address size prefixes 'data16' and 'addr16' change 32-bit operands/addresses into 16-bit operands/addresses. Note that 16-bit addressing modes (i.e. 8086 and 80286 addressing modes) are not supported (yet).
- The bus lock prefix 'lock' inhibits interrupts during execution of the instruction it precedes. (This is only valid with certain instructions; see a 80386 manual for details).
- The wait for coprocessor prefix 'wait' waits for the coprocessor to complete the current instruction. This should never be needed for the 80386/80387 combination.
- The 'rep', 'repe', and 'repne' prefixes are added to string instructions to make them repeat '%ecx' times.

Memory References

An Intel syntax indirect memory reference of the form

```
SECTION:[BASE + INDEX*SCALE + DISP]
```

is translated into the AT&T syntax

```
SECTION:DISP(BASE, INDEX, SCALE)
```

where BASE and INDEX are the optional 32-bit base and index registers, DISP is the optional displacement, and SCALE, taking the values 1, 2, 4, and 8, multiplies INDEX to calculate the address of the operand. If no SCALE is specified, SCALE is taken to be 1. SECTION specifies the optional section register for the memory operand, and may override the default section register (see a 80386 manual for section register defaults). Note that section overrides in AT&T syntax *must* have be preceded by a '%'. If you specify a section override which coincides with the default section register, 'as' will *not* output any section register override prefixes to assemble the given instruction. Thus, section overrides can be specified to emphasize which section register is used for a given memory operand.

Here are some examples of Intel and AT&T style memory references:

- AT&T: '-4(%ebp)', Intel: '[ebp - 4]'

  BASE is '%ebp'; DISP is '-4'. SECTION is missing, and the default section is used ('%ss' for addressing with '%ebp' as the base register). INDEX, SCALE are both missing.
- AT&T: 'foo(,%eax,4)', Intel: '[foo + eax*4]'

  INDEX is '%eax' (scaled by a SCALE 4); DISP is 'foo'. All other fields are missing. The section register here defaults to '%ds'.
- AT&T: 'foo(,1)'; Intel '[foo]' This uses the value pointed to by 'foo' as a memory operand. Note that BASE and INDEX are both missing, but there is only *one* ','. This is a syntactic exception.
- AT&T: '%gs:foo'; Intel 'gs:foo' This selects the contents of the variable 'foo' with section register SECTION being '%gs'.

Absolute (as opposed to PC relative) call and jump operands must be prefixed with '*'. If no '*' is specified, 'as' will always choose PC relative addressing for jump/call labels.

Any instruction that has a memory operand *must* specify its size (byte, word, or long) with an opcode suffix ('b', 'w', or 'l', respectively).

Handling of Jump Instructions

Jump instructions are always optimized to use the smallest possible displacements. This is accomplished by using byte (8-bit) displacement jumps whenever the target is sufficiently close. If a byte displacement is insufficient a long (32-bit) displacement is used. We do not support word (16-bit) displacement jumps (i.e. prefixing the jump instruction with the 'addr16' opcode prefix), since the 80386 insists upon masking '%eip' to 16 bits after the word displacement is added.

Note that the 'jcxz', 'jecxz', 'loop', 'loopz', 'loope', 'loopnz' and 'loopne' instructions only come in byte displacements, so that it is possible that use of these instructions ('gcc' does not use them) will cause the assembler to print an error message (and generate incorrect code). The AT&T 80386 assembler tries to get around this problem by expanding 'jcxz foo' to

```
            jcxz cx_zero
            jmp cx_nonzero
cx_zero:    jmp foo
cx_nonzero:
```

Floating Point

All 80387 floating point types except packed BCD are supported. (BCD support may be added without much difficulty). These data types are 16-, 32-, and 64- bit integers, and single (32-bit), double (64-bit), and extended (80-bit) precision floating point. Each supported type has an opcode suffix and a constructor associated with it. Opcode suffixes specify operand's data types. Constructors build these data types into memory.

- Floating point constructors are '.float' or '.single', '.double', and '.tfloat' for 32-, 64-, and 80-bit formats. These correspond to opcode suffixes 's', 'l', and 't'. 't' stands for temporary real, and that the 80387 only supports this format via the 'fldt' (load temporary real to stack top) and 'fstpt' (store temporary real and pop stack) instructions.
- Integer constructors are '.word', '.long' or '.int', and '.quad' for the 16-, 32-, and 64-bit integer formats. The corresponding opcode suffixes are 's' (single), 'l' (long), and 'q' (quad). As with the temporary real format the 64-bit 'q' format is only present in the 'fildq' (load quad integer to stack top) and 'fistpq' (store quad integer and pop stack) instructions.

Register to register operations do not require opcode suffixes, so that 'fst %st, %st(1)' is equivalent to 'fstl %st, %st(1)'.

Since the 80387 automatically synchronizes with the 80386 'fwait' instructions are almost never needed (this is not the case for the 80286/80287 and 8086/8087 combinations). Therefore, 'as' suppresses the 'fwait' instruction whenever it is implicitly selected by one of the 'fn...' instructions. For example, 'fsave' and 'fnsave' are treated identically. In general, all the 'fn...' instructions are made equivalent to 'f...' instructions. If 'fwait' is desired it must be explicitly coded.

Notes

There is some trickery concerning the 'mul' and 'imul' instructions that deserves mention. The 16-, 32-, and 64-bit expanding multiplies (base opcode '0xf6'; extension 4 for 'mul' and 5 for 'imul') can be output only in the one operand form. Thus, 'imul %ebx, %eax' does *not* select the expanding multiply; the expanding multiply would clobber the '%edx' register, and this would confuse 'gcc' output. Use 'imul %ebx' to get the 64-bit product in '%edx:%eax'.

We have added a two operand form of 'imul' when the first operand is an immediate mode expression and the second operand is a register. This is just a shorthand, so that, multiplying '%eax' by 69, for example, can be done with 'imul $69, %eax' rather than 'imul $69, %eax, %eax'.

So the assembly sequence would look like:

```
pushl   %eax
movl    _page_address, %eax
addb    $0,            (%eax)
popl    %eax
```

10.3.2 Assembler Instructions with C Expression Operands

To implement this sequence within a C program you have to use the *asm* statement. This Statement ist described in the following section.

The asm Statement

In an assembler instruction using 'asm', you can now specify the operands of the instruction using C expressions. This means no more guessing which registers or memory locations will contain the data you want to use.

You must specify an assembler instruction template much like what appears in a machine description, plus an operand constraint string for each operand.

For example, here is how to use the 68881's 'fsinx' instruction:

```
asm ("fsinx %1,%0" : "=f" (result) : "f" (angle));
```

Here 'angle' is the C expression for the input operand while 'result' is that of the output operand. Each has "'f"' as its operand constraint, saying that a floating point register is required. The '=' in '=f' indicates that the operand is an output; all output operands' constraints must use '='. The constraints use the same language used in the machine description (see 10.3.3).

Each operand is described by an operand-constraint string followed by the C expression in parentheses. A colon separates the assembler template from the first output operand, and another separates the last output operand from the first input, if any. Commas separate output operands and separate inputs. The total number of operands is limited to ten or to the maximum number of operands in any instruction pattern in the machine description, whichever is greater.

If there are no output operands, and there are input operands, then there must be two consecutive colons surrounding the place where the output operands would go.

Output operand expressions must be lvalues; the compiler can check this. The input operands need not be lvalues. The compiler cannot check whether the operands have data types that are reasonable for the instruction being executed. It does not parse the assembler instruction template and does not know what it means, or whether it is valid assembler input. The extended 'asm' feature is most often used for machine instructions that the compiler itself does not know exist.

The output operands must be write-only; GNU CC will assume that the values in these operands before the instruction are dead and need not be generated. Extended asm does not support input-output or read-write operands. For this reason, the constraint character '+', which indicates such an operand, may not be used.

When the assembler instruction has a read-write operand, or an operand in which only some of the bits are to be changed, you must logically split its function into two separate operands, one input operand and one write-only output operand. The connection between them is expressed by constraints which say they need to be in the same location when the instruction executes. You can use the same C expression for both operands, or different expressions. For example, here we write the (fictitious) 'combine' instruction with 'bar' as its read-only source operand and 'foo' as its read-write destination:

```
asm ("combine %2,%0" : "=r" (foo) : "0" (foo), "g" (bar));
```

The constraint "'0"' for operand 1 says that it must occupy the same location as operand 0. A digit in constraint is allowed only in an input operand, and it must refer to an output operand.

Only a digit in the constraint can guarantee that one operand will be in the same place as another. The mere fact that 'foo' is the value of both operands is not enough to guarantee that they will be in the same place in the generated assembler code. The following would not work:

```
asm ("combine %2,%0" : "=r" (foo) : "r" (foo), "g" (bar));
```

Various optimizations or reloading could cause operands 0 and 1 to be in different registers; GNU CC knows no reason not to do so. For example, the compiler might find a copy of the value of 'foo' in one register and use it for operand 1, but generate the output operand 0 in a different register (copying it afterward to 'foo''s own address). Of course, since the register for operand 1 is not even mentioned in the assembler code, the result will not work, but GNU CC can't tell that.

Some instructions clobber specific hard registers. To describe this, write a third colon after the input operands, followed by the names of the clobbered hard registers (given as strings). Here is a realistic example for the Vax:

```
asm volatile ("movc3 %0,%1,%2"
              : /* no outputs */
              : "g" (from), "g" (to), "g" (count)
```

```
                : "r0", "r1", "r2", "r3", "r4", "r5");
```

If you refer to a particular hardware register from the assembler code, then you will probably have to list the register after the third colon to tell the compiler that the register's value is modified. In many assemblers, the register names begin with '%'; to produce one '%' in the assembler code, you must write '%%' in the input.

If your assembler instruction can alter the condition code register, add 'cc' to the list of clobbered registers. GNU CC on some machines represents the condition codes as a specific hardware register; 'cc' serves to name this register. On other machines, the condition code is handled differently, and specifying 'cc' has no effect. But it is valid no matter what the machine.

If your assembler instruction modifies memory in an unpredictable fashion, add 'memory' to the list of clobbered registers. This will cause GNU CC to not keep memory values cached in registers across the assembler instruction.

You can put multiple assembler instructions together in a single 'asm' template, separated either with newlines (written as '\n') or with semicolons if the assembler allows such semicolons. The GNU assembler allows semicolons and all Unix assemblers seem to do so. The input operands are guaranteed not to use any of the clobbered registers, and neither will the output operands' addresses, so you can read and write the clobbered registers as many times as you like. Here is an example of multiple instructions in a template; it assumes that the subroutine '_foo' accepts arguments in registers 9 and 10:

```
asm ("movl %0,r9;movl %1,r10;call _foo"
     : /* no outputs */
     : "g" (from), "g" (to)
     : "r9", "r10");
```

Unless an output operand has the '&' constraint modifier, GNU CC may allocate it in the same register as an unrelated input operand, on the assumption that the inputs are consumed before the outputs are produced. This assumption may be false if the assembler code actually consists of more than one instruction. In such a case, use '&' for each output operand that may not overlap an input. *Note Modifiers::.

If you want to test the condition code produced by an assembler instruction, you must include a branch and a label in the 'asm' construct, as follows:

```
asm ("clr %0;frob %1;beq 0f;mov #1,%0;0:"
     : "g" (result)
     : "g" (input));
```

This assumes your assembler supports local labels, as the GNU assembler and most Unix assemblers do.

Speaking of labels, jumps from one 'asm' to another are not supported. The compiler's optimizers do not know about these jumps, and therefore they cannot take account of them when deciding how to optimize.

Usually the most convenient way to use these 'asm' instructions is to encapsulate them in macros that look like functions. For example,

```
#define sin(x)       \
({ double __value, __arg = (x);   \
   asm ("fsinx %1,%0": "=f" (__value): "f" (__arg));  \
   __value; })
```

Here the variable '__arg' is used to make sure that the instruction operates on a proper 'double' value, and to accept only those arguments 'x' which can convert automatically to a 'double'.

Another way to make sure the instruction operates on the correct data type is to use a cast in the 'asm'. This is different from using a variable '__arg' in that it converts more different types. For example, if the desired type were 'int', casting the argument to 'int' would accept a pointer with no complaint, while assigning the argument to an 'int' variable named '__arg' would warn about using a pointer unless the caller explicitly casts it.

If an 'asm' has output operands, GNU CC assumes for optimization purposes that the instruction has no side effects except to change the output operands. This does not mean that instructions with a side effect cannot be used, but you must be careful, because the compiler may eliminate them if the output operands aren't used, or move them out of loops, or replace two with one if they constitute a common subexpression. Also, if your instruction does have a side effect on a variable that otherwise appears not to change, the old value of the variable may be reused later if it happens to be found in a register.

You can prevent an 'asm' instruction from being deleted, moved significantly, or combined, by writing the keyword 'volatile' after the 'asm'. For example:

```
#define set_priority(x)  \
asm volatile ("set_priority %0": /* no outputs */ : "g" (x))
```

An instruction without output operands will not be deleted or moved significantly, regardless, unless it is unreachable.

Note that even a volatile 'asm' instruction can be moved in ways that appear insignificant to the compiler, such as across jump instructions. You can't expect a sequence of volatile 'asm' instructions to remain perfectly consecutive. If you want consecutive output, use a single 'asm'.

It is a natural idea to look for a way to give access to the condition code left by the assembler instruction. However, when we attempted to implement this, we found no way to make it work reliably. The problem is that output operands might need reloading, which would result in additional following "store" instructions. On most machines, these instructions would alter the condition code before there was time to test it. This problem doesn't arise for ordinary "test" and "compare" instructions because they don't have any output operands.

If you are writing a header file that should be includable in ANSI C programs, write '__asm__' instead of 'asm'.

An Example

The the code for the pager example would look like:

```
void *page_address;
/* some C code, initializing the global variable page_address */
asm("pushl   %%eax                          \n\t"
    "movl    _page_address,   %%eax        \n\t"
    "addb    $0,               (%%eax)      \n\t"
    "popl    %%eax                          \n\t"
);
```

The main differences are:

- use two "%" instead of one
- finish the line with "\n\t"

10.3.3 Constraints

Using simple constraints our example would look like:

```
void *page_address;
/* some C code, initializing the variable page_address */
asm("pushl   %%eax              \n\t"
    "movl    %0,    %%eax       \n\t"
    "addb    $0,    (%%eax)     \n\t"
    "popl    %%eax              \n\t"
    : /* no output */
    : "g" (page_address)
);
```

If you specify that the register eax is clobbered by the assembler statements you can leave out push and pop.

```
void *page_address;
```

```
/* some C code, initializing the variable page_address */
asm("movl    %0,  %%eax           \n\t"
    "addb    $0,  (%%eax)         \n\t"
    : /* no output */
    : "g" (page_address)
    : "eax"
);
```

And last but not least using machine specific constraints and creating a function the code would look like:

```
void foo(void *page_address)
{
    asm("addb  $0, (%%eax)        \n\t"
        : /* no output */
        : "a" (page_address)
);
```

The result would look like (assuming optimization):

```
.globl _foo
_foo:
        movl 4(%esp),%eax
#APP
        addb $0,     (%eax)

#NO_APP
        ret
```

10.3.4 A more complex example

Consider you have to implement a L3 system call for instance GetIDT(), a system call which provides the current interrupt descriptor table for the calling thread.

The system call is selected through a number in register *al*, invoked through *int 7* and expects its parameter in *ebx* and *ecx*. The results are returned in *ebx*, *ecx* and *edx*.

The code looks like the following:

```
IDTEntryT *GetIdt(int *SizeP)
{
  int type;
  IDTEntryT *idt;

  asm ("int  $7\n "
                    /* return values:                  */
      : "=b" (type),  /*    type of idt        <- ebx   */
        "=c" (idt),   /*    address of idt     <- ecx   */
        "=d" (*SizeP) /*    size of idt        <- edx   */
                    /* input values:                   */
       : "a" ((ByteT)SPECIAL_OP),
                    /*    syscall number      -> a    */
         "0" (SO_SET_IDT),/* opcode  set/getidt -> ebx  */
         "1" (-1)     /* -1 (no idt = getidt)   -> ecx  */
       );
  return idt;
}
```

The resulting code looks like this:

```
_GetIdt:
```

```
        pushl %ebx
        xorl %ebx,%ebx
        movl $-1,%ecx
        movb $52,%al
#APP
        int  $7
#NO_APP
        movl 8(%esp),%eax
        movl %edx,(%eax)
        movl %ecx,%eax
        popl %ebx
        ret
```

todo

Chapter 11

Interface To L3 TCP And UDP Functionality (under construction)

11.1 Preliminaries

11.1.1 Purpose of this Document

This document is addressed to people using the built-in TCP/IP functionality provided by **TCP Manager, UDP Manager and Name Server** L3 tasks.

The C-Interface for these managers was created as a basis for the development of a **socket server** for the UNIX emulation project on top of L3. However, they allow usage of networking functions (but without the socket interface).

The functions are still in a pre-release state, since only rudimentary testing was undertaken. In addition, the documentation is currently incomplete and awaits testers.

11.1.2 Related Documents

There is a document describing the network implementation of L3 (in Elan).

11.2 TCP Manager functions

The information contained here is basically from tcpifc.h.

11.2.1 Datastructures

`typedef struct { int index; int state; ThreadT thread; } TCPChannelT;` TCPChannelT contains the thread owning the connection, an index into a table managed by TCP Manager and the state of the connection (see below).

`typedef struct { int IP_Address; WordT Port; } TCPSocketT;` This structure (6 bytes) identifies uniquely the communication peer. IP_Address (4 Bytes) can be set using the L3_IP_Adress macro (see on top). Port contains the remote port.

Possible connection states

- L3_TCP_CLOSED
- L3_TCP_LISTEN
- L3_TCP_SYNSENT
- L3_TCP_SYNRECEIVED
- L3_TCP_ESTABLISHED

- L3_TCP_FINWAIT
- L3_TCP_FINWAIT2
- L3_TCP_CLOSEWAIT
- L3_TCP_CLOSING
- L3_TCP_LASTACK
- L3_TCP_TIMEWAIT

11.2.2 Macros

Some macros are defined in netmacros.h for easier access to the functions provided here.

- L3_Bytes2DWord(b1,b2,b3,b4) ((b1)<<24 — (b2)<<16 — (b3)<<8 —(b4))
- L3_IP_ADDRESS(b1,b2,b3,b4) (L3_Bytes2DWord(b4,b3,b2,b1))
- L3_Words2DWord(b1,b2) ((b1)<<16 —(b2))
- L3_SwapBytes(w) (((w) & 0xFF) << 8 — ((w) & 0xFF00) >> 8)
- L3_SwapDWords(dw) (((dw) & 0xFFFF) << 16 — ((dw) & 0xFFFF0000) >> 16)

11.2.3 TCPRead

```
int TCPRead (TCPChannelT * Channel, int Mode, char *Buffer, int BufSize, int
RecvTimeOut);
```

TCPRead reads data from an existing TCP connection. This can be either blocking or nonblocking, depending on *mode*. { L3_TCP_BLOCKING,L3_TCP_NONBLOCKING }

Notes

1. A connection must exist.
2. If the read is blocking (e.g., for normal server operation), one can specify a timeout after which the call comes back (recommended during testing).
3. You must provide a buffer. If it is too small, multiple reads should get the remaining data.
4. TCPRead sets the connection state in TCPChannelT variable.

11.2.4 TCPWrite

```
int TCPWrite (TCPChannelT * C, char *SendMsg, int MessageLength, char *Buffer, int
BufSize, int flags);
```

Writes to an existing TCP connection. The message to be send must be placed in SendMsg.

If an error occured, the error message is placed in Buffer (? needs testing).

Flags can be:

- *L3_TCP_NON*
- *L3_TCP_FIN* Sender is done, wants to close.
- *L3_TCP_SYN* Used by TCPOpen.
- *L3_TCP_RST* Connection reset.
- *L3_TCP_PSH* Disable Nagle algorithm.
- *L3_TCP_ACK* Used by both Open und Write.
- *L3_TCP_URG* Urgent flag.

It is not clear to what extend the current L3 implementation supports these flags.

11.2.5 TCPOpen

`int TCPOpen (TCPSocketT RSock, int Own_Port, int mode, int timer, TCPChannelT * Channel);`

<u>**Remarks**</u>

- RSock contains the peer IP address and the remote port.
- Own_Port can be either -1 (automatic port selection), [1024..8999] or, if privileged [1..1023].
- Open can be either passive or active.
- Active open: SYN-Datagram is sent, do an waitack afterwards to wait for the TCP 3-way handshake to complete. Remote port and IP address must be specified in this case.
- Passive open: Just say that we are ready to accept connections on `Own_Port`. Remote_IP and port should be 0.
- Timer specifies the time interval before giving up connection setup (? -test this)

11.2.6 Other functions

`int TCPWaitAck (TCPChannelT * ChannelPtr);`

MUST be called after an active open.

```
int TCPClose (TCPChannelT * Channel);
int TCPAbort (TCPChannelT *Channel);
int printChannel (TCPChannelT * Ch);
int TCPAbort(TCPChannelT * Channel);
int TCPGetSocket(TCPChannelT * Channel, TCPSocketT * MySockPtr, TCPSocketT * RSockPtr);
```

Fills Remote Socket and Own Socket.

`int TCPInitTCPServerThreadID (void);`

This function must be called before anything else.

11.3 UDP Manager Interface

Information from udpifc.h

11.4 Name Server Interface

Information from nsifc.h