RIJKSUNIVERSITEIT TE GRONINGEN
MATHEMATISCH INSTITUUT

GUTS

manuals

Wim Bronsvoort

TW-200 deel II

Contents.

GUTS user manual.

This user manual is an adapted version of the DEIMOS user manual. DEIMOS is a single-user, multi-tasking system for the PDP11, developed by B.Gilmore at the University of Edinburgh.

How to log in.

To log into the system must press any key on the keyboard of the console of which one wants to make use.

The system asks the user to type his name by printing:

name:

and his password by printing:

passwd:

Both name and password should be ended by a carriage return or a line feed.

The user process assigned to a user, who is admitted to the system, will ask for commands by the prompt 'command:'. The user can however type ahead of the prompt.

Commands.

All commands given to the system are interpreted by the command language interpreter (cli).

The input to the cli has two forms, either:

- -a standard program name, possibly followed by parameters, or
- -a file name, followed by 'stream definitions'.

The list of standard programs, which are located in the subsystem, is searched first. If the name is found, the program is executed.

If the name is not the name of a standard program, it must be the name of a file. A file name consists of:

- -user defined name (1 letter followed by 0-5 letters or digits)
- -user defined extension (1 letter)
- -identification of the owner of the file (2 letters).

The user defined name is written first, followed by a dot and the extension. If no extension is set, the extension 'o' is set by the cli to indicate an object file. If no owner identification is set, it is checked whether the user owns a file with that name; if not, it is checked whether the system manager, i.e. the owner of all files which are intended for general use, owns a file with that name. If an owner identification is set by the user himself, only the fully specified file is searched for. The owner identification has to be typed between square brackets (e.g. list.o[wb]).

If the name is neither the name of a standard program nor the name of a file, the message 'program not found' is printed.

A standard program name.

The following programs are located in the subsystem. The name is followed by the parameters, which are octal numbers except for file names which should be typed in their normal format.

1) rd
   read location, the program will ask for the address(es)
2) st
   store into location, the program will ask for the address(es) and contents
3) dump begin-address end-address
   dump the specified area of the virtual memory
4) svc service par1 par2 par3 par4 par5
   isssue a supervisor call with the specified parameters
5) delete filename
   delete the specified file
6) rename old-name new-name
   rename the file from the old name to the new name
7) permit filename owner-permission other's-permission
   set the access permissions of the file to the specified values;
   for every access mode there is one bit, if set the user has permission to connect the file in that mode
   bit 0: read unshared
   bit 1: read shared
   bit 2: read and write unshared
   bit 3: read and write shared
   an example: if the access permission of a user is 3, that user is allowed to connect the file in read unshared and in read shared mode
8) discon filename
   disconnect the specified file
9) logout
   stop the user process

A file name.

If a file name is specified, that file is loaded and the program contained in it is entered.

The 'stream definitions' are input in the form:

<input 1>,<input 2>/<output 1>,<output 2>

<input 1> etc, each represent a 'stream definition'. The definitions can be separated by a space too.

A 'stream definition' is of the form:

.tt -either input or output from your own terminal
.la -either input or output from the system console
.di -either input or output from the diablo printer
<any other devices on the system, specified in the same way>
<file name>

The streams defined are linked to the input and output streams available to an IMP program, for example, if the program 'fred' is run with the following command:

fred f1,f2/o1

then input stream two, used by calling 'select input(2)' is mapped to the file 'f2'. A 'select output(1)' will send output to the file 'o1'.

If a field is left blank, as with <output 2> in the above example, the stream is mapped to 'null' - which causes an 'end of file' signal on input and on output, all the output is thrown away.

Input stream zero and output stream zero are always mapped to the users terminal.

Further examples.

prog .tt	-uses one input stream '.tt', no output stream (apart from zero) is defined
prog file	-will read from a file called 'file' on stream one
prog /file	-no input streams defined (apart from zero), will write to a file called 'file' after a 'select output(1)'.

Note: when a program tries to write into a non-existing file, a file with the specified name is created.

System routines.

There are a number of system routines, i.e. non-standard routines usable by IMP programs, in the subsystem:

-%systemroutine svc (%record (%integer service, par1, par2, par3, par4, par5) %name mes); issues a supervisor call with the specified parameters

-%systemintegerfn owner; gives identification of the user

-%systemroutine command (%string (80) s); puts the string into the command buffer from which the command interpreter tries to read characters before it starts reading from the user console; this offers the possibility of issueing commands from programs

-%systemstring (12) %fn strname (%integer inout,stream); gives the name of the device or file linked to the specified stream

-%systemroutine setstrm (%integer inout, stream, %string (12) name); links the specified device or file to the specified stream

-%systemroutine setinr (%string (11) s); sets the input request message to s

-%systemroutine specout and %systemroutine normout; used to set whether or not output of a buffer for a device should be done at a line feed; this offers the possibility of speeding up output, because the process is put asleep only when the buffer for the device is full or the "prompt character" (k'100000') is sent.

To use one of them one has to include a "%systemroutinespec" for the routine in the program.

The editor.

The editor is a PDP11 version of the Edinburgh Compatible Context Editor.

For general information on the editor, see the user guide of the Edinburgh Editor by H. Dewar.

The command for calling the editor has three possible forms as illustrated below:

```
e /test          -to produce a file called 'test'
e test           -to inspect an existing file called 'test'
e test1/test2    -to produce a new file called 'test2'
                   from an existing file called 'test1'.
```

Notes:
1) The editor prompts '>' when it is ready to accept a command and ':' when it expects a line of input (command: get).

2) This version of the editor uses a 'window' on the file, this will not normally be apparent, but it does mean that the command 'm-*' will not necessarily return right to the top of the file.

Running the IMP compiler.

The IMP compiler (provided by P.S.Robertson) is a three pass compiler to which a fourth pass, a linking phase, and a fifth pass, which creates the object file, are automatically called for %begin .. %endofprogram programs.

For general information on the IMP language, see the IMP reference manual by P.S.Robertson of the University of Edinburgh.

There are four main ways of calling it:

```
imp a/b      -which compiles source file 'a' to object 'b'
imp a/,l     -which only does one pass and creates a listing file 'l'
               (note: 'l' can be either a filename or '.tt')
imp a/b,l    -which creates both an object file and a listing file
imp a,.tt/b -this form is used to change the stack size of the
               compiled program, see the section on the linker.
```

Notes:

1) The program 'impe' may be run to avoid the linking and object file generating phases for %externalroutine files. In this case, the file 'b' above is the third pass output.
2) The first pass automatically uses a file of '%specs' called 'prims.i[sy]'.
3) The first pass creates a temporary output file 'p1.s' for use with the second pass. The second pass creates 'p2a.s' and 'p2b.s', the third pass 'oy.s' and the fourth pass 'p4.s'.
4) The output from the third pass 'oy.s' is useful. It can be used as input to the program 'recode' if it is necessary to de-compile the compiler output.

The linker.

The linker is normally run automatically as the fourth pass to the IMP compiler. It can be run separately by typing one of the two following forms of command to the command language interpreter:

```
link a/b                    or
link a,.tt/b
```

Both of the above commands take the file 'a', which must be the output from the third pass of the compiler and create a file 'b', which can be used as input for the fifth pass of the compiler ('putl') to generate the object file.

The second form of the command overrides the standard stack size, the linker prompts for the new stack size as follows:

stack:

The desired stack size (in octal bytes), excluding the gla (initialized data area), should be typed in (default: 14000).

If a program has external references, the linker will attempt to satisfy them using 'lib000.o[sy]'. For more information on libraries, see the appropriate section.

If a second output stream is specified, a linker output map is given on this stream. A sample linker output map is given below:

```
CODE: 040000  GLA: 140020
  XDEF: 040000  #GO
  XREF: PACK
FILE: rtfile.o[sy]
  CODE: 043162  GLA: 140276
  XDEF: 043162  PACK
  XDEF: 043652  UNPACK

TOTALS: CODE = 004162  GLA/STACK = 014422
```

The base address for the program's code (040000) and gla (140020) are printed. The 'XREF' refers to 'external references' from that section of code (in this case they are all declared by the system). The 'XDEF' is an 'external definition', in this case '#GO' which is the entry point for the main program.

The 'FILE:' indicates that the linker has loaded an object file , it specifies its name and the start address of its code (043162) and gla (140276) sections. The list beneath that contains the 'external definitions', in this case entry points that the file contains.

The last line gives the overall code length of the program (004162) and the total size of the gla and the declared stack.

If any references are left undeclared, the linker will list them along with an 'UNDEFINED REFERENCE' message.

If the linker attempts to load a file that contains an entry point that has already been loaded, the message '*DOUBLE DEF' is output and the linker stops.

Library manipulation.

When the linker finds an external reference in a program file it will attempt to satisfy the reference by searching the library 'lib000.o[sy]', loading object files as necessary.

There are four programs currently available to the user to manipulate libraries, they are:

1) newlib -creates a new library file
2) crlibs -creates a new library file containing the definitions for the system routines
3) insert -inserts the entries of an object file into a library
4) index -lists the contents of a library.

Note: Each of the programs will manipulate libraries other than 'lib000.o[sy]', but at present the linker will not link them.

1) newlib
The command for using 'newlib' is as follows:

```
newlib /lib
```

This command will create a new library file 'lib'.

2) crlibs
The form of this command is as follows:

```
crlibs /lib
```

It will create a new library file 'lib', containing the definitions of the system routines.

3) insert
The command for using 'insert' is as follows:

```
insert test.o,lib1/lib2
```

The command will add the contents of the file called test.o into the library called 'lib1', creating a new library called 'lib2'.

4) index
The form of this command is as follows:

```
index lib
```

This command will print out all the entries in the library file 'lib'.

Available programs.

Besides the already mentioned user programs, there are available the following ones:

t -transfers data from input stream 1 to output stream 1 if a file or device is specified for this stream, otherwise to output stream 0 (the terminal)

files -gives a list of all files, with length in blocks, own access permission, other's access permission, date of creation, date of last access, number of accesses and offer/accept status, on output stream 1 if a file or device is specified for this stream, otherwise to output stream 0 (the terminal)

bytes -dumps input from input stream 1 in octal bytes on output stream 1

words -dumps input from input stream 1 in octal words on output stream 1

This manual is divided into 3 sections, describing the supervisor calls which can be performed by:

1) user processes only
2) both user and supervisor processes
3) supervisor processes only.

The first digit of a section number in this manual indicates to which of these categories the service described in that section belongs.

Supervisor calls are performed by placing the parameters in the registers r0-r5 and issuing the svc instruction. Supervisor processes can also make use of the send primitive, which is performed when the emt 2 instruction is issued.

To get the parameters into the registers user processes can make use of the system routine "svc" after including the following statements:

```
%recordformat message(%integer service,par1,par2,par3,par4,par5)
%systemroutinespec svc(%record (message) %name mes)
```

Supervisor processes can make use of the external routines "svc" and "send" after including the statements:

```
%recordformat message(%byteinteger id,source, %c
                      %integer par1,par2,par3,par4,par5)
%externalroutinespec svc(%record (message) %name mes)
%externalroutinespec send(%record (message) %name mes)
```

Supervisor processes have to fill in the source of a call themselves.

In the description of the parameters for a call, these will be numbered 0-5. If a parameter is divided into two parts, each occupying one byte, this is indicated by inserting a slash into the desciption of the parameter. The part left of the slash should be put into the left byte of the parameter (note: when declaring two byte integers within a record in IMP, the first one will be in the right byte of a word and the second one in the left byte).

Parameter 0 always contains the service number in the right byte and for supervisor processes the source of the call in the left byte. This source will be omitted in the description.

The registers r0-r5 will contain the reply to a call when control is returned to a process after an svc instruction. This reply is put into the specified record when using one of the "svc" routines. Supervisor processes can receive the reply to a call issued by a "send" by making use of the receive primitive, which is performed by issuing the emt 3 instruction. This can be done by making use of the externalroutine "receive" after including the statements:

```
%recordformat reply(%byteinteger id,service, %c
                    %integer par1,par2,par3,par4,par5)
%externalroutine receive(%record (reply) %name rep)
```

In the description of the reply to a call the words it consists of will be numbered 0-5 again. Word 0 always contains the number of the service sending the reply in the right byte and the number of the service having asked for the service in the left byte (remember that one supervisor process can handle several services). This word will be omitted in the description. Word 1 always contains a flag: if this flag equals zero, the service was executed successfully, otherwise it indicates the reason why it could not be executed successfully.

Blocks always consist of 512 bytes.

1.1) File services.

The name of a file must be expressed in RADIX 50 notation (appendix A). Parameter 1 contains the RADIX 50 equivalent of the first 3 characters of the file name, parameter 2 the RADIX 50 equivalent of the second 3 characters of the file name and parameter 3 the RADIX 50 equivalent of the extension character and the two owner id characters (in that order). For the connect, disconnect and accept file services the owner id should be specified, for the other services the system (re)sets them to the id of the user requesting the service.

1.1.1) Connect file.

call: 0) 40
1,2,3) name of the file
4) mode (2 bits) / for segment 0 and 1: startblock within segment (bits 6-3), segment (bits 2-0)
5) length in blocks of the part of the file to be connected (bits 14-9), startblock within file of the part to be connected (bits 8-0)

mode: 0=read unshared
1=read shared
2=read and write unshared
3=read and write shared

startblock within segment: 0-15
length: 0=whole file
N=N blocks (1<=N<=63)
startblock within file: 0-511

If a connect is requested for a file which is already connected and the new connect can be executed successfully, the part of the file already connected is first disconnected. The mode must not be changed.

reply: 1) flag
2) virtual address of (the part of) the file connected
3) length of (the part of) the file connected in bytes
4) total length of the file in blocks

flag: 1=file does not exist or no access permission is granted
2=startblock within file does not exist
3=file already connected in another virtual memory in a conflicting mode
4=part of the virtual memory required for connecting (the part of) the file is already in use
5=the mode for a connect of a file, which is already connected, must not be changed
6=user is not allowed to connect one of his base files

1.1.2) Disconnect file.

call: 0) 41
1,2,3) name of the file

reply: 1) flag

flag: 1=file not connected
2=user is not allowed to disconnect one of his base files

1.1.3) Create file.

call: 0) 42
1,2,3) name of the file
4) length

length: 0=either 1/2 the largest unused area or the entire second largest unused area, whichever is largest
-1=the largest unused area
M=M blocks
The maximum length is always 250 blocks.

The owner gets permission for all access modes, others get no permission at all.

reply: 1) flag
2) length in blocks

flag: 1=file already exists
2=length<0 or length>maximum size
3=not enough space for requested length
4=no more free space
5=directory overflow

1.1.4) Close file.

call: 0) 43
1,2,3) name of the file
4) length in blocks

length: should be less than or equal to the length allocated when the file was created

A file can be closed only once. If it is not closed, it is deleted at a compaction of the disk.

reply: 1) flag

flag: 1=file does not exist
2=file is connected in some virtual memory
3=file already closed
4=more blocks requested than allocated
5=negative length not allowed

1.1.5) Delete file.

call: 0) 44
1,2,3) name of the file

reply: 1) flag

flag: 1=file does not exist
2=file is connected in some virtual memory

1.1.6) Rename file.

call: 0) 45
1,2,right byte 3) old name
left byte 3,4,5) new name

The names are in a format different from the other calls. They consist of 5 bytes: the first byte contains the right byte of the RADIX 50 equivalent of the first 3 characters, the second byte the left byte, the third byte the right byte of the RADIX 50 equivalent of the second 3 characters, the fourth byte the left byte, and the last byte the RADIX 50 equivalent of the extension character.

reply: 1) flag

flag: 1=old file does not exist
2=file with new name already exists
3=file is connected in some virtual memory

1.1.7) Set access permissions of file.

call: 0) 46
1,2,3) name of the file
4) access permissions others (4 bits) /
access permissions owner (4 bits)

access permissions: for every access mode there is one bit, if set the user has permission to connect the file in that mode
bit 0: read unshared
bit 1: read shared
bit 2: read and write unshared
bit 3: read and write shared

An example: if the access permission of a user is 3, that user is allowed to connect the file in read unshared and in read shared mode.

reply: 1) flag

flag: 1=file does not exist or is offered to another user

1.1.8) Get file names and information.

call: 0) 47
1,2,3) name of the file into which names and information should be written

reply: 1) flag

flag: 1=file does not exist
2=file is too short

The file will contain for every file belonging to the user the following information:
-the name in RADIX 50 format (3 words)
-the length in blocks
-other's access permissions / owners's access permissions
-date of creation

-date of last access
-number of accesses
-offer/accept status.

Dates are in the following format:
bit 14-10: month (1-12)
bit 9-5: day (1-31)
bit 4-0: year-72.

At the end of the list there is a zero.

1.1.9) Offer file.

call: 0) 48
1,2,3) name of the file
4) id of new owner in RADIX 50 format

The file cannot be used any more until it is accepted or the offer is revoked. If parameter 4 equals zero, the previous offer is revoked and the owner gets permission for all modes and others no permission at all.

reply: 1) flag

flag: 1=file does not exist
2=file is connected in some virtual memory

1.1.10) Accept file.

call: 0) 49
1,2,3) name of the file

reply: 1) flag

flag: 1=file does not exist or is not offered

1.2) Console (user terminal) services.

Every console has its own number, which is also the service number of most services of that console. For a "put output" request however the service number equals the console number plus one, because output requests are directed via the scheduler. If an illegal service number is set, flag=3 is returned.

1.2.1) Open.

call: 0) console number
1) 0
2) 0=ascii mode
1=raw mode
3) 0=no echo
1=echo

reply: 1) flag

flag: 1=device already opened

1.2.2) Get input.

call: 0) console number

1) 1
2) block number (0-15) in segment 0 of buffer
3) maximum number of bytes

reply: 1) flag
2) number of bytes
(0 -> end of input: generated by typing eot (=control+d)
at the beginning of a line)

flag: 1=device not opened by user
2=illegal buffer area

1.2.3) Put output.

call: 0) console number + 1
1) <not used>
2) block number (0-15) in segment 1 of buffer
3) number of bytes

reply: 1) flag

flag: 1=device not opened by user
2=illegal buffer area

1.2.3) Set input request message.

call: 0) console number
1) number of characters (0-8) / 2
2-5) characters

reply: 1) flag

flag: 1=device not opened by user

1.2.4) Change mode.

call: 0) console number
1) 3
2) 0=ascii mode
1=raw mode
3) 0=no echo
1=echo

reply: 1) flag

flag: 1=device not opened by user

1.2.5) Get mode.

call: 0) console number
1) 4

reply: 1) flag
2) 0=ascii mode
1=raw mode
3) 0=no echo
1=echo

flag: 1=device not opened by user

1.2.6) Close.

call: 0) console number
1) 5

reply: 1) flag

flag: 1=device not opened by user

1.3) Recovery services.

1.3.1) Set recovery registers.

call: 0) 60
1) 0
2) new r5
3) new sp
4) new pc

reply: no parameters used

1.3.2) Reset process state.

call: 0) 60
1) 1
2) error number

reply: 1) error number
2) old r5
3) old sp
4) old pc
5) new r5

The sp and pc registers are set to their new values too.

This service is called on behalf of the user by the kernel when an error occurs and by a console handler when the user types an escape (error number -1). It may also be called by the user himself.

1.4) Logout.

call: 0) 61

a reply is not sent

2.1) Clock services.

The replies for the first two services contain the current time, the request for the last service contains the new time. These times are in the following format:

2) msec
3) min/sec
4) day/hour
5) year/month

with:

0<=msec<1000
0<=sec<60
0<=hour<24
1<=day<=maximum day in month
1<=month<=12
0<=year<100

2.1.1) Reply after a certain interval.

call: 0) 0
1) 0
2) number of milliseconds (msec)
3) number of seconds (sec)

0<=msec<1000
0<=sec<25
not (msec=0 and sec=0)

reply: 1) flag
2-5) current time

flag: 1=illegal request

2.1.2) Get time.

call: 0) 0
1) 1

reply: 1) <not used>
2-5) current time

2.1.3) Set time.

call: 0) 0
1) 2
2-5) new time

reply: 1) flag

flag: 1=only system processes and the system manager are
allowed to set the time
2=illegal time

3.1) Disk services.

3.1.1) Read from the disk to a specified block in core.

call: 0) 1
1) drive number / 0
2) disk address (in blocks)
3) start address in core (in blocks)
4) length (in blocks)
5) marker

The marker can be used by the caller to identify different requests.

reply: 1) flag
2) marker

flag: 1=no permission to use disk

3.1.2) Write to the disk from a specified block in core.

call: 0) 1
1) drive number / 1
2) disk address (in blocks)
3) start address in core (in blocks)
4) length (in blocks)
5) marker

reply: 1) flag
2) marker

flag: 1=no permission to use disk

3.1.3) Read from the disk to a specified address in core.

call: 0) 1
1) drive number / 2
2) disk address (in blocks)
3) start address in core
4) length (in blocks)
5) marker

reply: 1) flag
2) marker

flag: 1=no permission to use disk

3.1.4) Write to the disk from a specified address in core.

call: 0) 1
1) drive number / 3
2) disk address (in blocks)
3) start address in core
4) length (in blocks)
5) marker

reply: 1) flag
2) marker

flag: 1=no permission to use disk

3.2) Core management.

3.2.1) Get core.

call: 0) 2
1) 0
2) <not used>
3) length in blocks

reply: 1) flag
2) start block
3) length in blocks

flag: 1=no permission to call core manager

2=no free area large enough available

3.2.2) Release core.

call: 0) 2
1) 1
2) start block
3) length in blocks

reply: 1) flag

flag: 1=no permission to call core manager

3.3) Login.

call: 0) 4
1) <not used>
2) <not used>
3) address of bytearray in which name and password are stored
4) index in this array where name starts (0 origin indexing)
5) index in this array where password starts

Name and password are strings: the first byte contains the number of characters.

reply: 1) flag
2) service number assigned to user process

flag: 1=illegal name or password
2=user is already logged in
3=base file error
4=too many users on the system

This service is intended to be called by the console handlers.
If a user process calls it, flag=1 is returned.

Format of messages sent to the scheduler.

The scheduler is called from a number of other supervisor processes. The messages sent are not supervisor calls, but indications to the scheduler that a certain event has occurred. See the description of the system. The messages are identified by their identification number, which ranges from 0-7 and which is filled in by the sender of the message. A "put output" request for a console is directed via the scheduler. For the format of this request, see the system reference manual.

For the description of the messages the parameters are numbered 0-5, as in the description of the supervisor calls.

0) Clock.

See description of clock supervisor calls.

1) Disk.

See description of disk supervisor calls.

2) Connect file.

0) 3/2
1) service number of user process which requested connect

3) Disconnect file.

0) 3/3

4) Handler input.

0) 3/4
1) service number of user of console / service number of console
2) block number (0-15) in segment 0 of buffer
3) maximum number of bytes

5) Handler output.

0) 3/5
1) service number of user of console
2) block number (0-15) in segment 1 of buffer
3) number of bytes

6) Kernel.

0) 3/6

This is a call originating from the kernel, not from a supervisor process.

7) Logout.

0) 3/7
1) service number of user process which has logged out

How to load and start the system.

GUTS was developed on the RT11 system and is available on an RT11 compatible disk. RT11 is a single user operating system developed by DEC for the PDP11 series of computers. See the RT11 system manuals.

The object code of GUTS is contained in an RT11 file called 'guts.sav'. How this file is generated is described in the chapter 'how to generate a new system'. To load GUTS, one has to bootstrap RT11 in the normal way and to type the command:

```
get guts
```

GUTS can then be started manually at address 40 (octal). To restart the system without reloading it, one can use the same address.

There is a dump routine in the kernel which can be started manually at address 44 (octal). After starting the routine one has to set the begin address of the dump area into the switches, press the continue switch, set the end address of the dump area into the switches and press the continue switch again. The specified area will then be dumped onto the system console (the decwriter).

To bootstrap RT11 again, one can use a routine located at address 50 (octal). This routine should be started manually too.

After starting GUTS, the system manager, i.e. the user with name 'system', who owns all system files and files for general use, should log into the system and set the date and time by calling the program 'settim', which will prompt for the day, month, etc. and give their ranges. If this is not done, the creation dates of files will not be correct.

If it is necessary to compact the disk, i.e. copy all used blocks to the begining of the disk, one must restart RT11 and use the compress service of the peripheral interchange program:

```
r pip
*rk0:/s
*control+c
```

One can of course make use of other services of RT11 too, e.g. to delete files.

How to introduce a new user to the system.

To introduce a new user to the system, the system manager should change the password file 'passwd.p[sy]'. This can be done by the program 'newusr' in the following way:

```
command:newusr passwd.p/newpas.p
fred flintstone ff
```

The system manager has to start the program 'newusr' and type a line containing the name of the new user ('fred'), his password ('flintstone') and his identification ('ff'). The name and password consist of a maximum of 20 arbitrary characters, the identification consists of two letters. Both name and identification should be unique, i.e. not already be in the system. If one of them is not unique, an error message is printed. If no password is given, the system manager should type a '*'. The base files and a scratch file are created for the new user. The old password file should be deleted ('delete passwd.p') and a rename should be done on the new one ('rename newpas.p passwd.p).

How to generate a new system.

The current version of GUTS consists of 2 assembly programs and 8 IMP programs. The assembly programs are maintained on the RT11 system, the IMP programs on GUTS itself. The programs are linked on RT11. To read more about RT11 than can be treated here, see the RT11 system manuals.

Assembly programs.

The assembly programs are:
- -the kernel (kernel.mac)
- -the heap file (heap.mac)

The kernel is the part of the system responsible for the basic synchronisation, the heap file only contains a heap, i.e. a data area for operations on strings performed by supervisor processes.

These programs can be changed by using the RT11 version of the Edinburgh editor. This editor is started by giving the command:

```
r editor
```

RT11 prompts by '*', after which one can give the new file, followed by '=', followed by the old file (e.g. *kernel.new=kernel.mac). Now one can edit the file and afterwards close it by ':c'. One can rename 'kernel.new' into 'kernel.mac' by the program 'pip':

```
r pip
*kernel.mac=kernel.new/r
```

When the renaming is finished, one gets a '*' again. To leave 'pip' one has to type 'control+c'.

After changing the source file, one has to assemble it using the macro assembler. This is done by typing:

```
r macro
*kernel=kernel
```

which assembles the file 'kernel.mac' into the object file 'kernel.obj' (in general: '*a.x=b.y' assembles the file 'b.y' into the object file 'a.x'). To get a listing file, one should type:

```
r macro
*kernel,kernel=kernel
```

which produces a listing file 'kernel.lst' too. When the program is assembled, one gets a '*' again. To leave the assembler one has to type 'control+c'.

IMP programs.

Most of the system is written in IMP. At the moment there are 8 programs:
- -the clock handler (clock.i[sy])
- -the disk driver (disk.i[sy])
- -the core manager (core.i[sy])
- -the file system (filsys.i[sy])
- -the console handler (conhan.i[sy])
- -the scheduler (schedu.i[sy])
- -some external routines (syser.i[sy])
- -the initialization program (init.i[sy])

To change one of them, one should edit the source on GUTS using the Edinburgh Editor. The program should be compiled by the 'impe' compiler, which only performs the first three passes of the IMP compiler:

```
impe disk.i / disk.o
```

The file 'disk.o' now contains the output of the third pass of the compiler. To be able to link it with the object files generated by the RT11 macro assembler, the file is converted into RT11 macro format by the program 'mgeny':

```
mgeny disk.o / disk.m
```

The file 'disk.m' contains octal representations of the instructions and the initialized data area (glap) and global definitions for external definitions and external references. As an example, consider the file 'disk.m' in appendix B, which is produced when compiling the current version of the disk driver.

To generate a new system one has to switch to RT11. Before assembling the file 'disk.m', one has to edit it using the version of the editor on RT11:

```
r editor
*disk.mac=disk.msy
```

The RT11 file name 'disk.msy' is derived from the GUTS file name 'disk.m[sy]'. The following things have to be changed for all programs except the console handler:

- -the name of the code part has to be put behind the '.csect' statement on the first line
- -the name of the stack start label has to be put behind the '.globl' statement on the third line
- -the stack size, which is determined by the statement '.=.+200' at the end of the program, should be changed if it is different from 200 (octal)
- -the name of the stack start label has to be put behind the '.=.+...' statement.

The name of the code part and the stack start label are used by the initialization part and so are fixed. The stack size depends on the variables declared. Appendix C contains a list of programs with the name of the code part, the name of the stack start label and the stack size. How the changes should be made is shown in appendix D.

A macro file generated for the console handler should be changed in a different way. This is due to the fact that the code for this program is shared by a number of processes and a copy of the initialized data area (glap) has to be made for every process. The macro file has to be changed in the following way:

- -the name of the code part ('conhan') has to be put behind the '.csect' statement on the first line
- -the name of the glap label ('congla') has to be put behind the '.globl' statement on the third line
- -the instructions 'mov #glaentry,r1' and '011104' at the beginning of the code have to be deleted
- -before the statement 'glabase:glaentry' the statement 'congla:glabase' has to be included
- -the '.=.+200' statement has to be deleted (every handler process gets its own data area).

How these changes should be made is shown in appendix E.

After having made the changes, the file(s) must be assembled by the macro assembler. This can be done as follows:

```
r macro
*disk=disk
etc.
```

This assembles the file 'disk.mac' into the object file 'disk.obj'. The object file name is fixed, see appendix C.

For the linkage of the programs a so called batch program is available. This program ('gutstt.bat') combines the object files and calls the linker, which produces the file 'guts.sav' and gives a load map of the system. The program is started in the following way:

```
.assign tt:log,lst
.load ba,log
.r batch
*gutstt
```

See appendix F, which also shows a load map. From the load map one can see whether there are any undefined externals or multiple definitions (the multiple definition of '$go' is not important). The high limit determines the system size; if it exceeds the current system size in the initialization program, in blocks of 512 bytes, this constant has to be changed and another system has to be generated.

Adding a new service to the system.

If a new service is to be added to the system one must change the initialization program so that the service number is converted into the number of the process which handles that service. See appendix G for a table containing the service numbers currently in use.

Adding a new process to the system.

If a new process is to be added to the system one has to do two things:

- -choose a name for the code part, a name for the stack start label and a process number and change the initialization program so that a process table entry is initialized for the process
- -edit the file 'gutstt.bat' under RT11, so that the new object file is linked to the rest of the system, i.e. include the name of the object file into the list which is given in the file.

One can then generate the system in the normal way. See appendix G for a table containing the process numbers currently in use.

Changing the subsystem.

The source for the subsystem, which is written in assembly language, is in the RT11 file 'comi.mac'. One can edit it under RT11. After editing it, it should be assembled and linked:

```
r macro
*comi=comi (assuming the source is in 'comi.mac')
*control+c
r link
*nseg7.osy=comi
*control+c
```

Now the current version of GUTS has to be started and access permission has to be given to the file 'nseg7.o[sy]' by the system manager. If the address of the label 'strtpr' is changed, compared to the previous version, the value of the constant 'strtpr' in the initialization program, which equals 160000 (octal) + the address of the label 'strtpr', has to be changed and a new subsystem has to be generated. To use the new subsystem one has to do a rename under RT11:

```
r pip
*seg7.osy=nseg7.osy/r
```

It is advisable to save the previous version of 'seg7.osy' before this is done by:

```
r pip
*oseg7.osy=seg7.osy/r
```

Now one can restart the system.

Appendix A

RADIX 50 CONVERSION

character	ASCII octal equivalent	RADIX-50 equivalent
space	40	0
A-Z	101-132	1-32
$	44	33
.	56	34
UNUSED		35
0-9	60-71	36-47

The maximum RADIX-50 value is, thus,

47*50 X 50 + 47*50 + 47 = 174777

The following table provides a convenient means of translating between the ASCII character set and its RADIX-50 equivalents. For example, given the ascii string X2B, the RADIX-50 equivalent is (arithmetic is performed in octal):

```
X = 113000
2 = 002400
B = 000002
    ------
X2B = 115402
```

single char. or first char.		second character		third character	
A	003100	A	000050	A	000001
B	006200	B	000120	B	000002
C	011300	C	000170	C	000003
D	014400	D	000240	D	000004
E	017500	E	000310	E	000005
F	022600	F	000360	F	000006
G	025700	G	000430	G	000007
H	031000	H	000500	H	000010
I	034100	I	000550	I	000011
J	037200	J	000620	J	000012
K	042300	K	000670	K	000013
L	045400	L	000740	L	000014
M	050500	M	001010	M	000015
N	053600	N	001060	N	000016
O	056700	O	001130	O	000017
P	062000	P	001200	P	000020
Q	065100	Q	001250	Q	000021
R	070200	R	001320	R	000022
S	073300	S	001370	S	000023
T	076400	T	001440	T	000024
U	101500	U	001510	U	000025
V	104600	V	001560	V	000026
W	107700	W	001630	W	000027
X	113000	X	001700	X	000030
Y	116100	Y	001750	Y	000031
Z	121200	Z	002020	Z	000032
$	124300	$	002070	$	000033
.	127400	.	002140	.	000034
UNUSED	132500	UNUSED	002210	UNUSED	000035
0	135600	0	002260	0	000036
1	140700	1	002330	1	000037
2	144000	2	002400	2	000040
3	147100	3	002450	3	000041
4	152200	4	002520	4	000042
5	155300	5	002570	5	000043
6	160400	6	002640	6	000044
7	163500	7	002710	7	000045
8	166600	8	002760	8	000046
9	171700	9	003030	9	000047

Appendix B

```
.CSECT
.MCALL .REGDEF
.REGDEF
.GLOBL

START:
MOV #GLAENTRY,R1
CODEBASE:
011104 mov (R1),R4          ← gla
010605 mov R6,R5            ← begin
010514 mov R6,(R4)          ← lnb
062706 add #-50,R6
177730
010665 mov R6,-10(R5)       recieve(mes)
177770
012746 mov #-14,-(SP)
177754
060516 add r5,(sp)
016401 mov 46(R4),R1
000046
004431 jsr r4,@(R1)+
122765 cmpb #106,-14(r5)    if mes_id ≥ lownser then flag :1 and ->setrep
000106
177754
101005 bhi
012765 mov #1,-42(r5)
000001
177736
000167 jmp setrep
000454
116500 movb -14(r5),r0      com=command(mes-service&1)
177756
042700 bic #-2,r0
177776
006300 asl r0
060400 add r4,r0
016065 mov 62(r0),-42(r5)
000062
177734
136527 bitb -12(r5),#2      if mes_service&2=0 start
177756
000002
001020 bne
016501 mov -16(r5),r1       com=(mes_start//8&8-60)!com
177762
006700 sxt r0
071027 div R0,#10
000010
042700 bic #^c60,r0
177717
050065 bis r0,-44(r5)       ba=mes_start*512
177734
016500 mov -16(r5),r0
177762
072027 ash r0,#11
000011
010065 mov r0,-46(r5)
177732
000403 br
```

```
177740
060516
016401
000050
004431
000167
177200
016506
177770
004767
000056
000000
000000
000000
000000
000000
000000
000000
000000
000000
000000
000000
000000
000000
000000
000000
000000
017600
000000
013707
160012

.GLOBL HEAP
.GLOBL $GO
.GLOBL RECEIVE
.GLOBL SEND
.GLOBL WAITSEMA
.GLOBL PRINT
.GLOBL POCTAL
.GLOBL PRINTSYM

GLAENTRY:GLABASE
GLABASE:
000000
001003
000000
000000
000000
CODEBASE+000000
HEAP
GLABASE+000206
177777
```

```
104000
000207
000000
000000
000000
GLABASE+000040
$GO:CODEBASE+000000
GLABASE+000000
RECEIVE
SEND
WAITSEMA
PRINT
POCTAL
PRINTSYM
000505
000503
020047
020040
020040
020040
020040
025040
025052
062040
071551
020153
067157
073440
064562
062564
070040
067562
062564
072143
025040
025052
020047
020040
020040
020040
020040
025040
025052
020052
020040
020040
064544
065563
062440
071162
071157
020040
020040
020040
025052
025052
 =. +200
STACK:
 =. -< -START&77>+100
 END
```

Apendix C.

LIST OF SYSTEM SOURCE FILES.

macro files

kernel.mac
heap.mac

imp files

file	code name	stack start label	stack size
clock.i	clock	clostk	300
disk.i	disk	disstk	200
core.i	core	corstk	500
filsys.i	filsys	filstk	4000
conhan.i	conhan		
schedu.i	schedu	schstk	1000
syser.i			
init.i	init	initstk	200

Appendix D

```
.R EDITOR

EDIT 3.1 >
*DISK.MAC=DISK.MSY

>R0I. DISK
.CSECT DISK^
>M3R0I. DISSTK.P
.GLOBL DISSTK^
>M0
LINE >132. CHOPPED !
LINE >132. CHOPPED !
 **END**
>M-7
.=.+200
>F.2M0.S 200.P
.=.+200^
>MG
:DISSTK:
STACK:
>:C
```

Appendix E

```
.R EDITOR

EDIT 3.1 >
*CONHAN.MAC=CONHAN.MSY

>R0I  CONHAN.P
.CSECT CONHAN^
>M3R0I  CONGLA.
.GLOBL CONGLA^
>M3
MOV #GLAENTRY,R1
>K
CODEBASE:
>M
011104
>K
010605
>F.:.
GLAENTRY^:GLABASE
>G
:CONGLA:GLABASE
>M0
LINE >132.  CHOPPED !
 **END**
>M-6
.=.+200
>K
STACK:
>:C
```

Appendix F

```
.ASSIGN TT:LOG,LST
.LOAD BA,LOG
.R BATCH
*GUTSTT
MULT DEF OF $GO
MULT DEF OF $GO
MULT DEF OF $GO
MULT DEF OF $GO
MULT DEF OF $GO
MULT DEF OF $GO
MULT DEF OF $GO
RT-11 LINK    V04-04      LOAD MAP
GUTS  .SAV                30-NOV-78

SECTION ADDR    SIZE    ENTRY   ADDR    ENTRY   ADDR    ENTRY   ADDR

. ABS.  000000  011700  ASLEEP  000032  ERRORE  000440  IOTEP   000442
                        POWEP   000444  EMTEP   000446  CLOCKE  000450
                        DISKEP  000452  CINEP   000454  COUTEP  000456
                        KERNEX  000460  PROCES  000520  APD     000522
                        RUNPRO  000524  PROCTA  000726  READYQ  003326
                        BEGRDY  003366  ENDRDY  003370  SUPPAR  003374
                        SUPPDR  003414  USERAP  003434  FIND    005000
                        COUNT   005126
HEAPFI  011700  000200  HEAP    011700
DISK    012100  001400  $GO     013064  DISSTK  013430
CLOCK   013500  004100  CLOSTK  017506
CORE    017600  002600  CORSTK  022306
FILSYS  022400  034000  SEGMEN  045410  FILE    046756  BLOCK   050604
                        FILSTK  056346
CONHAN  056400  007600  CONGLA  065546
SCHEDU  066200  014500  SCHSTK  102654
SYSERS  102700  001500  RECEIV  104102  SEND    104110  SVC     104116
                        WAITSE  104124  SIGNAL  104132  PRINTS  104140
                        PRINT   104146  POCTAL  104156
INIT    104400  003300  STARTU  106774  AVAILA  106776  CONDES  107050
                        INITST  107600

TRANSFER ADDRESS = 007624
HIGH LIMIT = 107700
$JOB/RT11
        TTYIO
```

```
R PIP
OBJ.OBJ=KERNEL.OBJ,HEAP.OBJ,DISK.OBJ,CLOCK.OBJ,CORE.OBJ
OBJ.OBJ=OBJ.OBJ,FILSYS.OBJ,CONHAN.OBJ,SCHEDU.OBJ,SYSER.OBJ
OBJ.OBJ=OBJ.OBJ,INIT.OBJ

R LINK
GUTS,TT:=OBJ.OBJ


R PIP
OBJ.OBJ/D

$EOJ

END BATCH
```

Appendix G

OCTAL PROCESS NUMBERS IN USE

```
    0 - clock handler
    1 - disk driver
    2 - core manager
    3 - file system
10-15 - console handlers
   30 - scheduler
31-37 - user processes
   47 - idle
```

SERVICE NUMBERS IN USE

```
    0 - clock
    1 - disk
    2 - core management
    3 - scheduler
    4 - login
10-21 - consoles
40-49 - file services
   60 - recover
   61 - logout
70-76 - users
```

SEMAPHORES IN USE

```
    1 - disk
10-15 - console handlers
```