# M I K A D O S

## U s e r ´ s   G u i d e

Dansk Data Elektronik ApS
15 March 1979

2825

Author: Rolf Molich

## Table of contents

# 1. Introduction

MIK and MIKADOS are modular multiprogrammed real-time operating systems for the ID-7000 microcomputer system manufactured by Dansk Data Elektronik ApS. The MIK system is intended for computer configurations without a disc, while the MIKADOS system is a disc oriented operating system. MIK and MIKADOS are compatible.

This manual describes the calling conventions for the MIK and MIKADOS modules and subroutines. The reader of this manual is expected to have a basic knowledge of INTEL 8080 programming.

The MIK system monitor described in chapter 2 is based on the MIK system devised and programmed by Bodil Schrøder, Institute of Datalogy, University of Copenhagen, in May 1975. The original MIK system is described in a report entitled 'MIK - et korutineorienteret styresystem til en mikrodatamat' available from the Institute of Datalogy.

Further information about the MIK and MIKADOS systems may be found in the 'MIKADOS Principles of Operation' manual which contains a detailed description of the internal mode of operation of the MIK and MIKADOS systems as well as a description of all the control blocks used by the systems.

## 2. Monitor

The system monitor controls process scheduling and synchronization. The strategy used in process scheduling is described in section 2.8. Processes may exchange information and achieve synchronization by communicating via messages or general semaphores as outlined in section 2.1 and 2.5.

A number of subroutines simplifying the use of the message system are an integrated part of most MIK systems. These subroutines are described in the 'Utility Programs' manual.

Note: the MIK subroutines described in this chapter should not be called from interrupt algorithms in MIK drivers.

### 2.1 Messages and message semaphores

A message may be used to transmit information from one process to another. Examples of messages are i/o requests where the message contains information about buffer address, buffer length, type of operation desired etc., or startup messages where the message contains information about the task that the new process should perform.

Message information is transmitted using a message buffer obtained from the system. After a message has been transmitted the receiving process must either return the message buffer to the system or use it in a new communication. The message buffer may be used to transmit an answer message from the receiver to the original sender. The system does not distinguish between 'original messages' and 'answer messages'.

Messages are sent to and received from message semaphores. Using message semaphores as a link in process communication has several advantages over direct process communication. It enables several identical processes to cooperate on a given task represented by one semaphore to which all requests for that task are directed. It also permits one process to receive messages of various kinds using different semaphores.

Message communication is implemented using only two operations: Send a message to a semaphore and Receive a message from a semaphore. These operations are described in the following sections.

Obtaining a message buffer from the system and returning a message buffer to the system are implemented using the above-mentioned operations. The system buffers are received from and sent to a semaphore named 'POOL'. Initially all message buffers are attached to POOL, which can be viewed as an inexhaustible ressource. Programmers should be very careful about returning message buffers to the system as experience has shown that very strange error patterns result if POOL is drained.

An exchange of messages between a user process that wants to perform an i/o operation and the corresponding i/o driver might look as follows:

```
user process                        i/o driver process


receive message buffer from POOL
insert information about i/o
   operation into message
send message to semaphore associated
   with i/o driver
                                    receive message from semaphore
                                       associated with driver
                                    perform i/o operation
                                    send message indicating
                                       completion of i/o operation
                                       to user process
receive answer message from
   i/o driver and examine
   resulting i/o status
send message buffer back to POOL
```

Although not required by the basic message handling system it has turned out to be practical to specifically assign two semaphores to each system process. The semaphores are named the main semaphore (also called the process or coroutine semaphore) and the auxiliary semaphore. The main semaphore is used to receive messages containing requests from other processes. The auxiliary semaphore is used for communication with other processes during the execution of a request. The address of the main and auxiliary semaphores for a process are found in the KOSEM and HJSEM fields of the process control block.

## 2.2 Message format

A specific message format is not enforced by the basic message handling subroutines. However, the following format has turned out to be practical and is used almost exclusively by system programs in all internal and external communications.

| | | |
|---|---|---|
| Byte | -2 - -1 | system information - not to be modified by user |
| | 0 - 1 | address of answer semaphore, i.e. the semaphore that should receive the response to this message |
| | 2 | request code identifying the type of operation desired |
| | 3 | (not assigned) |
| | 4 - 5 | buffer address |
| | 6 - 7 | buffer length in bytes |
| | 8 - 10 | (not assigned) |
| | 11 | (assigned only for answer messages) completion code |
| | 12 - 13 | (assigned only for answer messages) number of bytes processed |

The byte labelled '-2' is the first byte in the message and the byte pointed at when identifying a message.

Usually bytes 0 - 10 are left unchanged by a request processor. These bytes may be used by the originating process to identify a completed request from an answer message.

The standard message buffer length is 16 bytes of which 14 may be used to transmit information as shown above. Other message buffer lengths may be defined during system generation. The system will also accept variable length message buffers.

## 2.3 Send a message

| | |
|---|---|
| Registers on entry: | (B,C) address of message semaphore |
| | (H,L) address of message buffer |
| Calling sequence: | CALL SEND |
| Registers on exit: | same as on entry |

If one or more processes are waiting in the queue of the semaphore pointed out by (B,C), the message pointed at by (H,L) is given to the first (oldest) process in the queue. Otherwise the message is placed in the semaphore queue.

The calling process remains active unless a process with a higher priority was waiting as the first one in the semaphore queue.

## 2.4 Receive a message

| | |
|---|---|
| Registers on entry: | (B,C) address of message semaphore |
| Calling sequence: | CALL FÅBUF |
| Registers on exit: | (D,E) pointer to byte 0 in received message buffer |
| | (H,L) pointer to byte -2 in received message buffer |

If one or more messages are waiting in the queue of the semaphore pointed at by (B,C), the first (oldest) message is

given to the calling process, which then remains active. The address of the message is placed in the BSKED field of the process control block and returned in the (D,E) and (H,L) registers as specified above.
Otherwise the process enters the not ready state and is placed in the semaphore queue.

## 2.5 General semaphores

A general semaphore consists of an integer S which may assume the values 0, 1, 2, 3, ... and a queue of processes which may be empty. Two operations may be used to manipulate a general semaphore, 'signal' and 'wait'. These operations have the following effect:

```
signal:  if the process queue is empty:   S := S + 1
         otherwise the first process in the queue is activated

wait:    if S is 0: the process that executed the wait is
                    entered into the process queue. The
                    process is reactivated by a signal opera-
                    tion executed by another process
         otherwise: S := S - 1
```

S is always 0 when the process queue is not empty.

General semaphores may be used to achieve process synchronization and to limit the access to non-shareable ressources. The last case is illustrated by the following example which illustrates the use of a general semaphore to limit the access to a printer which may be used by one process at a time only:

```
process A:                      process B:

wait(PRINT)                     wait(PRINT)
output report A to printer      output report B to printer
signal(PRINT)                   signal(PRINT)
```

The semaphore PRINT is initialized to 1 so that the first process executing wait(PRINT) gets the printer first while the other process must wait.

## 2.6 Signal to a semaphore

| | |
|---|---|
| Registers on entry: | (B,C) address of general semaphore |
| Calling sequence: | CALL SIGNL |
| Registers on exit: | same as on entry |

If one or more processes are waiting in the queue of the semaphore pointed at by (B,C), the first (oldest) process is activated.
Otherwise the semaphore value is increased by one.

The calling process remains active unless a process with a higher priority was waiting as the first one in the semaphore queue.

## 2.7 Wait for a semaphore

| | |
|---|---|
| Registers on entry: | (B,C) address of general semaphore |
| Calling sequence: | CALL WAIT |
| Registers on exit: | same as on entry |

If the general semaphore pointed at by (B,C) has a value greater than zero the value is decremented by one and the calling process remains active.
Otherwise the process enters the not ready state and is placed in the semaphore queue.

## 2.8 Process scheduling

Any process in the system may be in either of two states: 'ready' to execute program instructions or 'not ready', i.e. inhibited e.g. because it is waiting for a message or a signal to a general semaphore.

The ready processes are given access to the CPU, i.e. made active, according to their priority. The ready process with the highest priority (lowest numerical value of the PRIO field in the process control block) is always active. If several ready processes are found at the highest priority level, the CPU is multiplexed between these processes. First one process is made active. If this process is still active after a certain amount of time called the <u>time slice</u> has expired it is deactivated and the next process is made active etc. The first process is reactivated after all processes at that priority level have been activated. The time slice is defined during system generation. A typical time slice value is 250 ms.

The active process may lose control of the CPU, i.e. revert to the ready or not ready state in the following cases:

1) the process enters the not ready state following either a request to receive a message from a semaphore where no messages are waiting or a wait request to a general semaphore whose value is zero
2) the process has used its time slice
3) a process with a higher priority becomes ready because of an interrupt or because of action taken by this process

In case 2) the process will remain active if no other ready process exists with the same priority.

In case 3) the previously active process is placed first in the ready queue for its priority level and the unused part of the time slice is recorded in the process control block. Next time the particular priority level becomes the highest level with a ready process the interrupted process will be allowed to use the rest of its time slice.

## 2.9 Programming considerations

The MIK system provides each user with a stack of 60 bytes. A process may redefine its stack to another location which is then used until redefined or until the program terminates itself.

The user should be aware that a multiprogramming system makes certain demands on the size and use of the stack:

1) MIK driver interrupts require 12 stack words (24 bytes) on the stack in addition to the maximum program stack size

2) the contents of the 24 bytes above the current top of stack are always unpredictable. If the user wants to load (H,L) with the contents of the word just below the top of stack without affecting any registers he should <u>not</u> write:

```
    INX     SP
    INX     SP
    POP     H
    DCX     SP
    DCX     SP
    DCX     SP
    DCX     SP
```

as this makes the contents of the top of stack unpredictable. Instead he might write:

```
    PUSH    PSW
    LXI     H,4
    DAD     SP
    MOV     A,M
    INX     H
    MOV     H,M
    MOV     L,A
    POP     PSW
```

The first example would be correct if executed with the interrupt system disabled. The powerful synchronization operations of MIK permit the handling of most problems without disabling the interrupt system. It is generally not recommended to disable the interrupt system except in the following cases

- timing real time intervals of 1 ms or less
- executing a sequence of i/o instructions that may not be interrupted

# 3. I/O drivers

This chapter describes the calling conventions for the most common i/o drivers in the MIK system. The purpose of an i/o driver is to provide the user with a simple high-level interface to the input/output units attached to the computer system. The i/o drivers take care of all interrupt handling, i/o instructions and any device peculiarity in the communication with the external device controlled by the driver.

The i/o drivers use a standardized message format in their communications with user processes as described in section 2.2 with certain changes as described in detail in the following sections. Note that only the message bytes used by the drivers are mentioned.

All i/o request messages should be sent to the main semaphore associated with the i/o driver process for the device in question. The names of these semaphores are defined during system generation.

## 3.1 Clock driver

The clock driver updates the system clock. The system clock shows the current system time and date. The system clock is set and read by accessing the clock driver data area. The relevant part of this area has the following layout:

```
DATO:: DS      2       ;DAY OF MONTH IN ASCII ('01'-'31')
       DB      '.'
       DS      2       ;NUMBER OF MONTH IN ASCII ('01'-'12')
       DB      '.'
ÅR::   DS      4       ;NUMBER OF YEAR IN ASCII ('1978' ETC.)
;
TIDSP::DS      2       ;TIME OF DAY, HOUR IN ASCII ('00'-'23')
       DB      '.'
       DS      2       ;MINUTE IN ASCII ('00'-'59')
       DB      '.'
       DS      2       ;SECOND IN ASCII ('00'-'59')
```

The clock and calendar are in ASCII format. The system automatically increments the calendar by one day when advancing the clock from 23.59.59 to 00.00.00. Leap year support is not implemented, i.e. after February 28 the system will always advance the calendar to March 1. However, the system will accept that the clock is set manually to February 29 and will also advance the clock correctly after that day.

The clock and calendar must be set manually by the operator every time the system is restarted.

### 3.1.1 Measuring time intervals

The clock driver will also measure time intervals for user processes.

To have a time interval measured a user process must send a message to the clock semaphore (labelled SLEEP). This message must have the following format:

| | | |
|---|---|---|
| Byte | 0 - 1 | address of the semaphore to which the answer message should be sent |
| | 2 - 3 | number of system time units that should pass before the answer message is sent (16 bits unsigned) |

After the specified time period has elapsed an answer message is sent to the specified semaphore. The answer message is identical to the original message except that bytes 2 and 3 are zero.

The length of a system time unit is defined during system generation. User programs may determine the length of a system time unit by using the symbolic constant ENH2 defined by the SYMB macro. This constant has a value that is equal to the number of system time units in one second.

## 3.2 CRT console driver

The CRT console driver controls one or more CRT console terminals connected to an ID-7004 Asynchronous communication module or to an ID-7012 4-port Asynchronous communication module.

The driver was developed for the Mini Tec and TEC Model 70 line of terminals manufactured by TEC Inc., Tucson, Az., USA. With these terminals all operations described in the following sections are available. However, with minor changes the driver will support most asynchronous CRT terminals equipped with a cursor control facility.

### 3.2.1 Read a character string

Requests the driver to read one input line from the terminal to the buffer. The operator may edit the input string during the input operation.

Message format: as described in section 2.2. The request code is 1. Byte 3 of the message is used for control information. In the control information only bit 1 (weight 2) is used. If this bit is set then the operation is a binary read and no editing is supported. The operation is terminated as soon as the buffer is full or BREAK is pressed. If this bit is not set then the operation is a normal ASCII read and special character processing (editing) takes place as described below.

In an ASCII read the following input character codes receive special attention:

| | | |
|---|---|---|
| RETURN | - | line feed - carriage return is echoed back to the terminal. The input operation is terminated and the user process receives its answer message |
| ENTER | - | same as RETURN |
| ESC | - | same as RETURN except that completion code bit 4 is set (see section 3.2.8) |
| <- | - | (cursor left). The cursor and the internal buffer pointer are both moved one position left unless they are at the start of the line in which case no action is taken. The buffer contents are not altered. |
| -> | - | (cursor right). The cursor and the internal buffer pointer are both moved one position right unless they are past the end of the buffer in which case no action is taken. The buffer contents are not altered. |
| ↑ | - | (cursor up, insert character) all characters from and including the character at the cursor position and to and including the last but one character in the line are moved one position right. The last character in the line is deleted and a blank is inserted at the cursor position. The cursor is not moved. The operation is duplicated in the input buffer. |
| ↓ | - | (cursor down, delete character) all characters from and including the character to the right of the cursor position are moved one position left. The character at the cursor position is deleted and a blank is inserted as the last character in |

| | | |
|---|---|---|
| | | the line. The cursor is not moved. The operation is duplicated in the input buffer. |
| ERASE | - | all characters in the line from and including the character at the cursor position are deleted (replaced by blanks). The cursor is not moved. The operation is duplicated in the input buffer. |
| RUBOUT | - | all characters in the line are deleted (replaced by blanks). The cursor is moved to the start of the line. The operation is duplicated in the input buffer. |
| TAB | - | the cursor and the internal buffer pointer are advanced to the nearest relative buffer address divisible by 8. The buffer contents are not changed. |

If the user enters a control character (binary value < 48) other than the ones described above a BELL character is echoed (the terminal issues an audible sound). The buffer and screen contents are not modified.

Any attempt to move the cursor past the buffer limits will cause a BELL character to be echoed. The buffer and screen contents are not modified.

Note that before the input operation starts, the buffer is filled with blanks by the driver.

## 3.2.2 Output a character string

Requests the driver to output a character string from the buffer to the terminal.

Message format: as described in section 2.2. The request code is 2.

The driver automatically issues a line feed - carriage return sequence following the last character in the buffer unless the control sequence <S> is included at the start of the buffer (see below).

If the output buffer starts with a '<', the following characters to and including a terminating '>' are interpreted as control characters. Control character sequences are not printed. The control characters and their significance are:

| | |
|---|---|
| X | - erase screen |
| C | - move the cursor to the (x,y) address specified immediately after the C as XXYY where 01 <= XX <= 80 and 01 <= YY <= 24 (YY = 25 is permitted on some terminals) |
| S | - omit the final line feed - carriage return, i.e. leave the cursor in the position immediately after the last character output |
| B | - output blinking text (only on terminals having blinking text support) |
| P | - output protected text (only on terminals having protected text support) |

E - ignored (included to provide compatibility with other drivers)

Example: the output string '<XBC0402>Ready' will erase the screen and output the text 'Ready' in the 2nd line starting at character position 4. The text will blink. After the output operation the cursor will be located in the first position of line 3.

## 3.2.3 Update a character string

Requests the driver to output a character string from the buffer to the terminal (same operation as described in section 3.2.2). Subsequently the user may update part of or all the characters in the string just as in an input operation.

Message format: as described in section 2.2. The request code is 3. Byte 8 of the message is used to specify the number of characters counting from the start of the buffer that are not to be modified in the update operation.

## 3.2.4 Update screen in block mode

Requests the driver to output a character string from the buffer to the terminal (same operation as described in section 3.2.2). After the output operation is finished the cursor is moved back to the position it had before the output operation and the 'enter block mode' state is established (see section 3.2.5). This operation should be used when operating a terminal in block mode only.

Message format: as described in section 2.2. The request code is 4.

### 3.2.5 Enter block mode

Requests the driver to place the terminal in block mode. In block mode all input characters are echoed directly to the screen without being recorded in any buffer. When the user enters XMIT (ASCII code 02), RUBOUT (ASCII code 7F), ESC (ASCII code 1B), or ERASE (ASCII code 0C), a message is sent to a specified semaphore and the block mode is left.

Message format:

| | | |
|---|---|---|
| Byte | 2 | request code (5) |
| | 4 - 5 | address of the semaphore to which a message should be sent when the user enters XMIT, RUBOUT, ESC, or ERASE |

When the user enters XMIT, RUBOUT, ESC, or ERASE, a message is sent to the specified semaphore. The message contents are:

| | | |
|---|---|---|
| Byte | 0 - 1 | address of POOL |
| | 2 | request code (7) |
| | 10 | 02 if the user entered XMIT |
| | | 1B if the user entered ESC |
| | | 7F if the user entered RUBOUT |
| | | 0C if the user entered ERASE |

No direct answer is given to the 'Enter block mode' request. The user request message buffer is returned directly to POOL.

The driver may be reactivated while in block mode, i.e. while the user is entering data, e.g. by issuing an 'Update screen in block mode' request.

Note that when an 'Enter block mode' request is received by the driver any information about a previously defined Break semaphore is deleted (see section 3.2.7).

## 3.2.6 Input screen contents in block mode

Requests the driver to force an input operation of the current screen contents in block mode.

Message format: as described in section 2.2. The request code is 6.

The input buffer is not altered by the driver before the transmission starts. If the terminal transmits more characters than will fit into the buffer the driver ignores the surplus characters.

## 3.2.7 Define Break semaphore

Defines the address of a semaphore to which the driver should send a message if the Break key on the terminal is depressed. The Break key is any key or combination of keys which causes the terminal to issue a 02 code (control/B, ENTER or XMIT). The key labelled BREAK is not recognized by the driver as a Break key.

Message format:

| | | |
|---|---|---|
| Byte | 2 | request code (7) |
| | 4 - 5 | address of Break semaphore |
| | 9 - 10 | terminal identification |

No direct answer is given to the 'Define Break semaphore' request. The user request message buffer is returned directly to POOL.

When the Break key is depressed the following message is sent to the Break semaphore if one has been defined:

| | | |
|---|---|---|
| Byte | 0 - 1 | address of POOL |
| | 2 | request code (7) |
| | 9 - 10 | terminal identification |

When this message has been sent the information in the driver about the address of the Break semaphore is deleted. The Break semaphore must be redefined before the driver will issue a new break message.

The terminal identification word in the above messages may be used by a program that controls several terminals to determine from the Break message which terminal encountered a Break condition. The terminal identification word is not inspected or modified by the driver.

### 3.2.8 Answer message format

Unless otherwise specified above the driver at the completion of a user request will return the original message buffer to the semaphore whose address is specified in byte 0 and 1 of the original message. The answer message has the following format:

|  |  |  |
|---|---|---|
| Byte | 0 - 10 | same as in original message |
|  | 11 | resulting driver status (completion code) |
|  | 12 - 13 | (valid only for request codes 1, 3, and 6) number of characters read or updated. For ASCII input operations (request codes 1 and 3) the buffer length minus the number of trailing blanks is returned. If the input buffer contains only blanks the driver returns the cursor offset relative to the start of the line when RETURN was depressed (0 means cursor located at the leftmost position in the line). For binary input operations the actual number of characters read is returned. |

The bits in the completion code have the following significance:

|  |  |  |
|---|---|---|
| bit | 0 - | 1 if Break pressed during i/o operation (Break message has been sent if Break semaphore defined) |
|  | 2 - | error in message format (e.g. illegal request code) or illegal control sequence in output string |
|  | 4 - | 'Read a character string' operation terminated by ESC (as opposed to RETURN or ENTER) |

## 3.3 Printer drivers

The system currently supports the following printer devices:

- Logabax LX 180 AL printer, manufactured by Logabax, Arcueil Cedex, France (medium speed matrix printer)
- Data 100 Model 3400 with Centronics interface, manufactured by Data 100 Corp., Minneapolis, USA (medium to high speed chain printer)
- Qume Sprint Micro 3/45, manufactured by Qume Corp., Hayward (CA), USA (daisy wheel printer, used in text editing systems)

All printer drivers will accept requests for output operations the message format being the same as described in section 2.2. The request code is 2.

The driver automatically issues a line feed - carriage return sequence following the last character in the buffer unless the control sequence <S> is included at the start of the buffer (see below).

If the output buffer starts with a '<', the following characters to and including a terminating '>' are interpreted as control characters. Control character sequences are not printed. The control characters and their significance are described below. Note that some drivers do not support all the control characters mentioned here. Unsupported and illegal control characters result in status bit 2 being set.

S - omit the final line feed - carriage return, i.e. print the first character of the next output line immediately after the last character output in this operation

N - omit the final line feed, i.e. print the first character of the next output line in the leftmost position of the line where the current buffer is output

B - output all characters in the buffer in italics

G - output all characters in the buffer as extended characters

E - eject page before outputting buffer contents

X - same as E

V - perform vertical tab as defined by VFU tape before outputting buffer contents

An answer message is returned to the semaphore specified in the request message as soon as the request has been processed. Byte 13 of the return message contains the resulting device status. The bits of the status word have the following significance:

0 - printer not ready (power off or out of paper)
1 - illegal request code
2 - illegal control sequence

The following sections describe the differences between the above standards and the actual device drivers.

## 3.3.1 Logabax printer driver

This driver works exactly as described above.

## 3.3.2 Data 100 model 3400

This driver supports the control characters E and X.

The driver will not report a 'power off' or 'out of paper' condition on the printer, i.e. status bit 0 cannot be set.

## 3.3.3 Qume model 3/45

This driver supports the control characters E, X, N, and S.

If bit 7 (the most significant bit) is set in a character in the output buffer, the character will be printed underlined by the driver.

The driver accepts a message that redefines the character distance, the line distance and the margin width. The message format is as described in section 2.2 with the following changes:

| | | |
|---|---|---|
| Byte | 2 | request code (8) |
| | 4 - 5 | new character distance in 1/120" (standard value is 10/120") |
| | 6 - 7 | new line distance in 1/96" (standard value is 16/96"); odd values are reduced by one |
| | 8 | new margin width in number of characters; measured using the character distance defined above (standard value is 30 characters) |

## 3.4 Disc drivers

The system currently supports the following disc devices:

- Sykes model 7000 floppy disc (250 Kbytes capacity)
- Pertec model 3400 disc (20 Mbytes capacity)

The disc drivers transfer information to and from the disc devices in multiples of 256 bytes called a sector. Note that this sector size does not have to be identical to the physical sector size for the disc drive.

The disc drivers accept two requests, read and write. The message format is:

| | | |
|---|---|---|
| Byte | 2 | request code (1 for read, 2 for write) |
| | 4 - 5 | buffer address |
| | 6 - 7 | buffer length; must be a multiple of 256 bytes |
| | 8 - 9 | disc identification (see section 5.4) |
| | 10 - 11 | starting track number |
| | 12 | starting sector number |

Bytes 10 - 12 of the message contain the track/sector address of the first sector to be read/written in the operation. A disc i/o operation may extend over more than one track. Track switching is handled automatically by the driver.

The first track on a disc has number 0. The first sector on a track has number 0.

When a data transfer has been completed the requesting message is returned to the answer semaphore specified in the message. The answer message is identical to the original message except for byte 13 which contains the resulting disc status. The status bits have the following significance:

| | | |
|---|---|---|
| bit 0 | - | illegal disc identification |
| bit 1 | - | transfer extends past last sector of disc drive |
| bit 2 | - | illegal track or sector no. in starting address for transfer |
| | | buffer length not multiple of 256 bytes |
| | | buffer length is zero |
| | | buffer length is >= 32 Kbytes (128 sectors) |
| bit 3 | - | illegal request code |
| bit 4 | - | attempt to write on a write protected disc |
| bit 5 | - | hard disc error |
| bit 6 | - | selected disc not ready |

More than one bit may be set.

The Pertec disc driver also supports an initialization operation (request code 3) used to format brand new discs. This operation is similar to a write operation except that the disc hardware does not compare the track/sector address in the sector header to the actual address before a write operation is performed. A disc cartridge must be formatted before it is used on the Pertec disc. An attempt to read or write a disc that has not been initialized will result in a hard disc error code. Floppy discs used on the Sykes disc are initialized by the manufacturer.

The disc characteristics as seen from the MIKADOS system are:

| | Sykes floppy disc | Pertec disc |
|---|---|---|
| No. of sectors/track | 26 | 48 |
| No. of tracks/platter | 37 | 400 |
| No. of platters/drive | 1 | 4 |
| Approx. no. of bytes/drive | 250K | 20M |

Disc characteristics, i/o port addresses and disc identifications are related to one another through the disc definition table, PLTAB, which is constructed during system generation.

### 3.4.1 Subroutine UDIND

The disc driver is accessed by the system and the system programs almost exclusively through subroutine UDIND. The calling conventions for this subroutine are given below.

Calling sequence:

```
(H,L) -> DCB area
(A)      bit 5-0 contain the request code (1, 2, or 3)
         bit 7=1 the read/write occurs to/from the address
                 contained in field NÆPST of the DCB
         bit 7=0 the read/write occurs to/from the buffer
                 area of the DCB  ( DCB+BUFF )
```

Before calling UDIND the user must place the track/sector address of the first sector to be read/written in the SEKT1 field, the number of sectors to be transferred in the SANT field, and a pointer to the disc identification in the PLBET field of the DCB.

```
       CALL    UDIND
```

After the call the a-register contains 0 if no error was detected; else (A) contains the error code (41, 42 or 44, see appendix 1).

## 4. Symbolic resource handling

The resource administration system makes it possible for cooperating sequential processes to establish a scheme for controlled access to serially reusable resources (e.g. line printers, main storage areas or information in a data base) by providing subroutines for the reservation and release of symbolic resources.

A resource may be reserved exclusively or non-exclusively. If a resource has been reserved exclusively no other process is allowed to reserve that resource before it has been released. If a resource has been reserved non-exclusively other processes are allowed non-exclusive reservations of the resource.

The resources are identified by resource names consisting of 10 ASCII characters. The first 8 characters are taken from main storage while the last 2 characters are taken from the process stack thus making it simple to perform resource reservations from reentrant subroutines.

The system does not establish any connection between a symbolic resource and an actual physical resource in the system. The resource administration does not provide any protection in case of a process that uses a resource which it has not reserved previously.

## 4.1 Reserve a resource

Calling sequence:

```
        PUSH    subparameters
        PUSH    2 last characters of resource name
        PUSH    address of first 8 characters of resource name
        CALL    RESRV
```

The subroutine reserves the specified resource. The reservation may be exclusive or non-exclusive.

The subparameter is a 16-bit word. The subparameter bits have the following significance:

```
bit 0  -  0 non-exclusive reservation wanted
          1 exclusive reservation wanted

bit 1  -  0 return with error code if resource cannot be
            reserved
          1 wait until resource available
```

The first character of the resource name must be alphanumeric (ASCII character value between 21 and 7E).

On exit from the routine the a-register contains a result code which may assume the following values:

```
0  -  reservation ok
1  -  resource reserved by some one else
3  -  the first character of the resource name is illegal
4  -  no resource element available (the number of resource
      elements is defined during system generation)
```

## 4.2 Release a resource

Calling sequence:

```
        PUSH    2 last characters of resource name
        PUSH    address of first 8 characters of resource name
        CALL    RELSE
```

The subroutine releases the specified resource.

On exit from the routine the a-register contains a result code which may assume the following values:

| | | |
|---|---|---|
| 0 | - | resource released |
| 2 | - | the resource has not been reserved |

## 5. General remarks about the file system

The MIKADOS file system allows the user to easily access disc storage on his ID-7000 computer system.

This chapter contains an introduction to the MIKADOS file system, describing the general disc layout and calling conventions for the file system subroutines. The following chapter contains an exact description of the formats of the various file system calls.

### 5.1 File system structure

The information that can be handled by the MIKADOS system is stored on disc drives. There may be one or more disc drives in a system and disc drives of different types may be intermixed.

A disc drive contains one or more physical discs. The discs some of which may be removable are used for storing information. The physical discs are mapped onto logical discs during system generation. A logical disc is a disc seen from the programmer's point of view. Most often there is a direct relationship between physical and logical discs, but an installation may choose to subdivide a physical disc into several logical discs. Currently it is not possible to combine physical discs into one logical disc.

A logical disc (hereafter referred to as 'a disc') consists of a file catalog and a file area. The maximum number of files that can be accomodated in the file catalog is specified during system generation. A special scheme (hashing) ensures fast file lookup even if the catalog is nearly full.

A file consists of a primary file which is the original file constructed by the create operation plus 0 to 25 extents. All extents are of the same size as the original file. Any attempt to extend a file past the 25th extent is rejected.

When specifying a file size the user must be aware that a sector always consists of 256 bytes as seen from the MIKADOS system even though the hardware manuals may say otherwise. Further the user should note that the first 32 bytes of the first sector of the primary file and of any extent are used for system information and not accessible to the user. Except for its effect on the useable file size this restriction is transparent to the user.

## 5.2 Calling file system subroutines

The file system functions are implemented as subroutines. Most of the subroutines are reentrant. The reentrant routines are normally located in the resident system area, while the non-reentrant routines are linked into the programs that use them. In the current MIKADOS version, the following routines are not reentrant: CREAT, RENAM and PURGE.

Each file system subroutine returns an error code in the a-register. The error code is zero if the operation was completed succesfully. A complete list of the error codes and their significance appears in appendix 1.

## 5.3 The Data Control Block (DCB)

In all calls of the file system a DCB (Data Control Block) must be specified. The DCB is a data area containing 48 + n * 256 bytes, where 1 <= n <= 20. The DCB consists of a pointer area of 48 bytes followed by a buffer area of n * 256 bytes used for packing and unpacking records from disc sectors. If the DCB contains more than 48 + 256 bytes, the surplus is used for enlarging the buffer area. A large buffer area generally means fewer disc accesses and consequently faster file processing. Note however that using a DCB larger than the primary file size for the file being processed does not give any advantage.

The user of the file system does not have to know anything about the detailed contents of the DCB. The DCB layout is described in the 'MIKADOS, Principles of Operation' manual.

## 5.4 File identification

A file is identified by a file name, a file type code and a disc identification. The file name is a string of 8 characters the first of which must be a printable ASCII character (hex code 21 through 7E). The file name is supplemented by a file type code which must be an alphanumeric character. The file type code is used together with the file name to uniquely identify a file on a disc, i.e. several files may have the same file name but different type codes (e.g. source file, relocatable file and program file). The user may select his type codes freely. Certain systems programs (e.g. editor and assembler) assume that certain file types are assigned specific type codes:

| | |
|---|---|
| K | - source text |

| | |
|---|---|
| R | - relocatable program |
| digit | - linked program for region 'digit' |

The disc identification is used to point out the logical disc where the file is located. The disc identification always has the form 'Px' where x is an alphanumeric character. The legal disc identifications vary among installations and are defined at system generation time.

Note that two files with the same name and type may exist on different discs in the same system.

## 5.5 Nullfiles

The user may use the file catalog for storing limited amounts of information (16 bits at a time) about a string which conforms to the above specifications for file names. These catalog elements are called nullfiles. A nullfile does not occupy space in the file area of the disc. Only the following operations are permitted on nullfiles: Create, Purge and Open file (returns the nullfile information).

Example: Nullfiles are used by the linker program to convert operating system entry point names to actual main storage addresses.

## 5.6 Fixed and variable length record files

The file system supports two different record structures within files: fixed length records and variable length records. A file should not contain records of both types.

In a fixed length record file the record length is defined when the file is opened. All records in the file have the same length. Record boundaries are not recorded in the file but are determined using only the specified record length. The main advantage of this record structure is that it permits direct and fast access to any record in the file using the position file subroutine. In special applications the user may take advantage of the fact that the record length may vary from one open operation to the next, and that the record length may be altered in the DCB while the file is open. These features should be used with great care.

In a variable length record file each record carries information about its own length. The length indicators require a 2-byte overhead per record in the file. Variable length record files must be read or written sequentially, i.e. access to record no. n is possible only after reading the preceding n-1 records. The main advantage of this file type is a better utilization of disc space because no record occupies unneccessary space. The 'record length' parameter needed by certain file system subroutines should be set to any nonzero value when operating on variable length record files.

## 6. File system calls

This section describes the exact format of the file system calls. The reader is expected to be familiar with the general information about the file system contained in chapter 5.

Note that a file must always be opened before it can be read from or written upon. A file can be opened implicitly using CREAT or explicitly using OPEN. A file must be closed after use. If these rules are not obeyed, information from the file may be lost.

### 6.1 Create file

Calling sequence:

```
        PUSH    n  -  DCB size is  48 + n * 256 bytes
        PUSH    disc identification
        PUSH    number of sectors in primary file
        PUSH    record length (must be greater than zero)
        PUSH    file type in 8 most significant bits
        PUSH    address of file name
        PUSH    DCB address
        CALL    CREAT
```

The subroutine creates a new primary file of the given size, name and type on the specified disc. If the operation completes succesfully the new file will be opened exclusively (reading and writing permitted). Positioning, reading and writing of the file may take place immediately after the call.

The DCB must be closed when the call is made.

If the number of sectors in the primary file is set to 0, the file system will create a nullfile, i.e. a catalog element only. In this case the record length parameter is inserted as the nullfile information. The nullfile value may be inspected by opening the nullfile.

## 6.2 Extend a file

Calling sequence:

```
        PUSH   DCB address
        CALL   CREXT
```

The subroutine creates a new extent to the file specified by the DCB. The extent will be of the same size as the primary file.

The DCB must be opened exclusively before the call is made. Any attempt to extend a file past the 25th extent is rejected.

Extent switching is performed automatically by the file system. The distinction between records in the primary file and records in extent files is completely transparent to the user except for an increase in access time.

The extend operation has no influence on the record position parameters in the DCB.

## 6.3 Rename a file

Calling sequence:

```
PUSH    n  -  DCB size is  48 + n * 256 bytes
PUSH    disc identification
PUSH    file type in 8 most significant bits
PUSH    address of new file name
PUSH    address of current file name
PUSH    DCB address
CALL    RENAM
```

The subroutine changes the the name of the file specified by 'current name' to 'new name'. The file type is not changed.

The DCB and the file must be closed when the operation is initiated. The DCB and the file will be closed when the operation is completed.

Renaming a file does not change the file contents.

## 6.4 Purge a file

Calling sequence:

```
PUSH    n  -  DCB size is  48 + n * 256  bytes
PUSH    disc identification
PUSH    file type in 8 most significant bits
PUSH    address of file name
PUSH    DCB address
CALL    PURGE
```

The subroutine purges the specified file including all extents. The space occupied by the file is not released until the disc is squished.

The DCB and the file must be closed when the operation is initiated. The DCB will be closed when the operation is complete.

## 6.5 Open a file

Calling sequence:

```
        PUSH    n  -  DCB size is  48 + n * 256  bytes
        PUSH    disc identification
        PUSH    subparameters (described below)
        PUSH    record length
        PUSH    file type in 8 most significant bits
        PUSH    address of file name
        PUSH    DCB address
        CALL    OPEN
```

The subroutine initializes the specified DCB with control information about the specified file. Positioning, reading and writing of the file may take place immediately after the call.

The DCB and the file must be closed when the operation is initiated.

If the record length is set to zero the system will use the record length specified when the file was created.

The subparameter is a 16-bit word. The subparameter bits have the following significance:

```
bit 0:   1  -  the file is to be opened exclusively, i.e.
               reading and writing must be permitted. No
               other users can access a the file that is
               opened exclusively
         0  -  the file is to be opened non-exclusively,
               i.e. reading only is permitted. Other users
               may read the file simultaneously
```

Bits 1 to 15 of the subparameter word are currently not used. They should be set to zero.

A file that has just been opened is positioned to record no. 1.

If the specified file name corresponds to a nullfile no OPEN operation takes place, i.e. the nullfile and the DCB are still closed after successful completion of the operation. The 16 information bits associated with the nullfile are saved in the DCB, field SEKT1. If a nullfile is opened, OPEN returns a result code of -1 in the a-register.

## 6.6 Position file

Calling sequence:

```
      PUSH    record number
      PUSH    DCB address
      CALL    POSN
```

The subroutine changes the information in the DCB so that the next read or write operation will cause the fixed length record with the specified record number to be input or output.

The DCB must be opened before the call is made.

Before the operation takes place the subroutine checks that the first character in the record specified lies within the file boundaries. If this turns out not to be the case the operation is rejected and an error code is returned. The subroutine does not check whether the whole record lies within the file boundaries. This is checked by the read/write routines.

The first record in a file is record number 1.

## 6.7 Read a fixed length record

Calling sequence:

```
      PUSH   buffer address
      PUSH   DCB address
      CALL   READ
```

The subroutine reads the next fixed length record from the file specified by the DCB. The record length used is the one specified in the OPEN operation or if no record length was specified in the OPEN operation the record length specified when the file was created. The buffer length must be greater than or equal to the record length. After the record is read the DCB is positioned to the next record.

The DCB must be open when the operation is initiated.

If the record to be read lies partly or completely outside the file boundaries an error code is returned and the DCB is not changed.

## 6.8 Write a fixed length record

Calling sequence:

```
PUSH    buffer address
PUSH    DCB address
CALL    WRITE
```

The subroutine writes the next fixed length record onto the file specified by the DCB. The record length used is the one specified in the OPEN operation or if no record length was specified in the OPEN operation the record length specified when the file was created. The buffer length must be greater than or equal to the record length. After the record is written the DCB is positioned to the next record.

The DCB must be open when the operation is initiated.

If the record to be written lies partly or completely outside the file boundaries an error code is returned and the DCB is not changed. Note that this permits the user program to issue an extend file command followed by a renewed write operation.

The write operation is effectively an update operation i.e. existing records before and after the one written are not modified by the write operation.

## 6.9 Read a variable length record

Calling sequence:

```
PUSH     buffer address
PUSH     DCB address
CALL     READV
```

The subroutine reads the next variable length record from the file specified by the DCB into the buffer. The length of the record that has been read is returned in the b-register. If b = 0 then the end-of-file mark has been reached. Note that the buffer must have a length of at least 78 characters, i.e. the largest variable record length permitted + 1.

The DCB must be open when this operation is initiated.

## 6.10 Write a variable length record

Calling sequence:

```
PUSH     buffer address
PUSH     DCB address
CALL     WRITV
```

The subroutine writes a variable length record from the buffer onto the file specified by the DCB. The record length must be specified in the first byte of the buffer. The record contents start in the second byte of the buffer. If a buffer length of zero is specified an end-of-file mark is written in the file.

The length of a variable length record must not exceed 77 bytes.

## 6.11 Close a file

Calling sequence:

```
    PUSH    DCB address
    CALL    CLOSE
```

The subroutine closes the file specified by the DCB. The DCB is released.

# 7. Main storage administration

The main storage administration controls the ID-7000 RAM memory used for data areas and for execution of nonresident programs.

The main storage administration consists of two parts:

- the data area administration (DAA), which consists of two subroutines, ALLOC and DELOC, which are used to dynamically allocate and deallocate data areas of specified sizes in the dynamic data area. System as well as user processes may use this facility

- the program area administration (PAA), which is a process that starts and stops other processes, reads programs into main storage and controls swapping. Messages to the PAA should be sent to the semaphore labelled HLSEM

## 7.1 Memory layout

The ID-7000 memory layout under MIK is shown below:

```
  address                      contents

   FFFF
                               debugger
   F000
                            program regions
                   (execution of non-resident programs)
approx. 4000
                  dynamic data area (administered by DAA)
approx. 3D00
                         MIK system data area
   3000
                              MIK system
      0
```

The program execution area, which is administered by PAA, is divided into a number of regions, called R1, R2, R3 etc. Each region may contain one program. Several programs may share one region by using swapping (see section 7.7). A program cannot occupy more than one region. Regions currently may not overlap. The starting addresses and sizes of the regions are defined during system generation.

The dynamic data area, which is administered by DAA, is dynamically subdivided into a number of data areas as requested by system or user processes as needs arise. The DAA keeps track of the unused parts of the dynamic data area. The size of the dynamic data area is defined during system generation. The usual size is approximately 750 bytes.

## 7.2 Allocate a main storage area

Calling sequence:

```
        LXI     H,number of bytes in requested main storage
                  area ( 4 <= (H,L) <= 2047 )
        CALL    ALLOC
```

On exit the (D,E) register contains the address of the first byte in the allocated data area (lowest address) while (H,L) contains the length of the allocated data area. Note that to avoid fragmentation of the dynamic data area, the DAA may allocate up to 3 more bytes than were requested.

If (D,E) = 0 then a data area of the desired size is not currently available.

If (D,E) = -1 then the value of (H,L) on entry was illegal (less than 4 or greater than 2047)

The (A) and (B,C) registers are not changed by the routine.

## 7.3 Deallocate a main storage area

Calling sequence:

```
        LXI     D,start of main storage area
        LXI     H,length of main storage area in bytes
        CALL    DELOC
```

On exit the (H,L) register contains 0 if the operation was completed without error, and 1 if the (H,L) register on entry to DELOC contained an illegal value (< 4 or > 2047) or if the user has attempted to release a main storage area that was not acquired using ALLOC.

The (A), (B,C) and (D,E) registers are not changed by the routine.

## 7.4 Start a process

When a process wants to initiate a new process to execute a program it should send a message to the PAA with the following contents:

| | | |
|---|---|---|
| Byte | 0 - 1 | address of answer semaphore |
| | 2 | request code (40) |
| | 3 - 10 | name of the file which contains the program module in executable form; this file usually has been produced by the linker |
| | 11 | second character of region identification for the region to which the program has been linked and where it should be executed (specify 'M' for region 'RM') |
| | 12 | second character of disc identification for disc containing program module file (specify 'N' for disc 'PN') |
| | 13 | priority of new process (see section 2.8 for information about process scheduling) |

When the PAA has found an available process and allocated it for the execution of the program, an answer message is sent to the answer semaphore specified in the request. The answer message format is

| | | |
|---|---|---|
| Byte | 0 - 2 | same as in request message |
| | 3 - 4 | address of the process control block of the new process allocated to execute the program |
| | 5 - 6 | address of the main semaphore of the new process allocated to execute the program. The parent process may use this semaphore to transmit a message to the new process |
| | 13 | error code |

The requested program is read into the memory area to which it has been relocated by the linker. Execution starts in the address specified during relocation.

The error codes are listed in appendix 1. Note that the PAA may return file system error codes resulting from erroneous file, region, or disc specifications.

## 7.5 Terminate a process

When a process wants to terminate itself it should send a message to the PAA with the following contents:

| | | |
|---|---|---|
| Byte | 0 - 1 | address of answer semaphore; normally a process is not able to wait for the answer so the address of POOL should be specified |
| | 2 | request code (41) |
| | 3 - 4 | address of process control block for this process |

After a process has terminated itself, it should issue a receive message call to the semaphore DEAD, i.e.

```
        LXI     B,DEAD
        CALL    FÅBUF
```

This call enters the process in the queue of available processes in the system.

When a 'Terminate process' message is received by the PAA, the main storage region used by the process is released immediately for use by other processes. The message therefore should not be sent using instructions in the program region as the instructions after the SEND may have been overlaid before execution of the FÅBUF calling sequence can be completed.

Termination of a process, i.e. transmission of a message as described above followed by a queue up to the semaphore DEAD may be accomplished simply by calling the reentrant system subroutine EXIT. The calling sequence is

```
        CALL    EXIT
```

The call has no parameters. No return occurs.

A program should not terminate itself before it has received an answer message to all its messages, closed all of its files, released all of its symbolic ressources and deallocated any dynamic memory for which it is responsible.

Note that no process is allowed to terminate another process using this operation.

## 7.6 Swapping

Swapping means that a program in a main storage region is temporarily stopped and copied into a disc file to make space for another program that uses the same region. The old program is said to be swapped in favor of the new program.

Program B is swapped in favor of program A under the following conditions:

1) Program A has a higher priority than program B, and program B has indicated that swapping of it is permitted.

   Swapping is permitted/prohibited by sending an appropriate message to the PAA (see section 7.7).

2) Program B has informed the PAA that it is waiting for a message and that its presence in main storage is not required while it is waiting.

   Swapping is permitted by waiting for a message using a call of `FÅBFX instead of FÅBUF (see section 7.8). Note that if a program is waiting for the completion of an i/o operation, swapping should be permitted only if the i/o buffer is located in the dynamic data area, i.e. outside the program region. This is taken care of by the system if the i/o message is sent using a call of SENDX instead of SEND (see section 7.8). Swapping is prohibited by FÅBFX as soon as a message is received.

When a new process is started swapping of the corresponding program is prohibited.

## 7.7 Change swap status for a process

If a program wants to change its swap status it should send a message with the following contents to the PAA:

| | | |
|---|---|---|
| Byte | 0 - 1 | address of answer semaphore |
| | 2 | request code (42, 43, or 44) |
| | 3 - 4 | address of the process control block for the process affected by the operation |

The operations are:

Permit swapping (request code 42). After this message the program associated with the process may be swapped in favor of another process with a higher priority that uses the same region. While the program is swapped the process is stopped. The answer message is sent immediately

Release region temporarily (request code 43). After this message the program associated with the process may be swapped in favor of any other process using the same region. The process is not stopped during a swap, i.e. the program must send this request and continue execution in a code area outside its own region (e.g. routine FÅBFX, see section 7.8). The answer message is sent immediately.

Prohibit swapping (request code 44). After this message the program associated with the process can not be swapped. If the program is swapped at the time this message is issued it is read back into memory as soon as its priority and region availability permit. The answer message is sent when the program is in memory.

## 7.8 Subroutines SENDX and FÅBFX

These subroutines are used instead of SEND and FÅBUF when the user wants to perform an i/o operation during which the program may be swapped.

The SENDX subroutine works exactly as the SEND subroutine except that before sending the message it copies the i/o buffer (address in message byte 4-5, length in message byte 6-7) into a buffer located in the dynamic data area. Then the buffer pointer in the message is changed to point at the buffer copy in the dynamic data area and the message is forwarded to its destination. During the i/o operation the program may now release its region (see below).

The FÅBFX subroutine works exactly as the FÅBUF subroutine except that before calling FÅBUF it releases the program region associated with the process (see section 7.7). After a message has arrived for the process at the specified semaphore, a 'Prohibit swapping' request is issued. When the program has been brought back into memory by the PAA, the subroutine checks to see if the message is an i/o message. If this is the case the i/o buffer is copied from the dynamic data area back into program memory and the dynamic data area is deallocated.

# 8. Operator communication

The operator communication module of the MIKADOS operating system enables the console operator to define output units for list and error messages and to initiate the execution of named programs. The operator may pass parameters to the programs initiated from the console.

Before entering a command, the operator must press the key labelled 'XMIT' or 'ENTER' (on terminals where neither of these keys exist, the operator may use a control/B keystroke instead). The MIKADOS system responds by showing a '>' as shown below:

```
>
```

The operator may now enter the desired command.

The legal commands are LI, FE, and RU.

The LI command defines the list output unit. Only two forms of this command are permitted, namely

```
>LI,D       use console terminal as list output unit
>LI,P       use system printer as list output unit
```

The FE command defines the error output unit. Only two forms of this command are permitted, namely

```
>FE,D       output error messages on console terminal
>FE,P       output error messages on system printer
```

At system startup time the list output unit is the printer while error messages are output on the console terminal.

The RU command instructs the system to load into memory and start execution of a named program. Optionally the command may contain a parameter string to be passed to the program. The format of the RU command is

```
>RU,programname(:Pn(:region))(,parameterstring)
```

| | |
|---|---|
| where programname | is the name of the program to be executed |
| Pn | (optional) is the disc identification (see section 5.4) of the disc where the linked program module can be found. If this parameter is not specified the system assumes that the program module should be loaded from the system disc (P1) |
| region | (optional) is the number of the memory region where the program is to be loaded. If this parameter is not specified the system assumes that the program is to be loaded into the memory region that was assigned to this terminal during system generation. |
| parameterstring | (optional) is a string that is passed unchanged to the new program using a dynamic memory area. |

Example: the operator command

```
>RU,ASM,PROGR,L,X
```

initiates execution of the assembler and asks it to assemble the program PROGR and to produce a listing (L) and a cross-reference listing (X).

## Appendix 1. MIK system error codes

The following is a complete list of the error codes produced by the MIK system.

The origin of the error codes is as follows:

| | | |
|---|---|---|
| code | 0 | : (many routines; indicates no error) |
| | 1 through 24 | : file system |
| | 30 | : resource management |
| | 40 through 52 | : UDIND (error in disc access) |
| | 61 through 69 | : program area administration (PAA) |

| error code | explanation |
|---|---|
| 0 | no error |
| 1 | a file with the specified name does not exist |
| 2 | a file with the specified name already exists |
| 3 | no more room on disc |
| 4 | illegal record length |
| 5 | the file is being used by another user |
| 6 | the specified DCB is not open |
| 8 | attempt to extend a file more than 60 times |
| 9 | the file name is illegal (the first character is not printable, see section 5.4) |
| 10 | illegal disc identification |
| 11 | attempt to position file to non-existent record |
| 12 | attempt to read or write a record that lies partly or completely outside the file boundaries |
| 13 | the file has not been opened for writing |
| 14 | the catalog on the disc is full |
| 15 | illegal DCB length |
| 16 | illegal number of sectors in file |
| 17 | illegal file type |

| | |
|---|---|
| 18 | the file name has not been reserved (returned by CLOSE if file not open or DCB bombed) |
| 19 | error in variable record length indicator (record length > 77 or error in file format) |
| 30 | no resource element available |
| 40 | disc drive not ready |
| 42 | hard error on disc |
| 44 | disc drive is write protected |
| 48 | illegal track/sector no. or illegal buffer length |
| 50 | transfer extends past last sector of disc drive |
| 52 | illegal disc identification |
| 61 | no process control block available |
| 62 | illegal priority |
| 63 | illegal main storage limits for program |
| 64 | illegal entry point for program |
| 65 | illegal load module file structure (number of extensions not zero, number of sectors does not correspond to main storage limits specified) |

## Appendix 2. MIKADOS Bootloader

The MIKADOS bootloader is a small stand alone program, which resides in a 1 k PROM.

The bootloader may be used to:

- load an absolute program from a disc file to memory

- load and execute an absolute program from a disc file

- compare the contents of an absolute program file with memory and report any mismatch

- store an absolute program from memory to a disc file

The bootloader uses less than 130 (hex) bytes of RAM memory for working storage. This memory area should not be the same as the memory area used by the processed program.

If the bootloader is entered at the standard entry point, the work area will be placed ending in the RAM byte with the highest memory address.

If the bootloader is entered at the standard entry point + 10 (hex), the work area will be placed ending in the RAM byte pointed to by the stack pointer when the bootloader is entered.

In ID-7000 systems the standard entry point address usually is E000. The bootloader is entered using the debugger, e.g. by typing the debugger command  E000<0XJ

In SPC/1 systems the bootloader is entered automatically at the standard entry point when power is turned on. In these systems the bootloader cannot be entered at other addresses than the standard entry point. The size of the work area in this case is less than 530 (hex) bytes.

The bootloader identifies itself after initiation and asks for the name of the file to be processed. The user may respond with up to 9 characters. Input is terminated by entering RETURN. The first 8 characters input (including trailing blanks) are the name of the file to be loaded. The 9th character, which must be a digit, is the file type. If no file type is input, 0 (zero) is assumed. The user may correct input errors before pressing RETURN by backspacing the cursor by entering Control-H (backspace). If an illegal control character is entered the bootloader is restarted.

The bootloader asks for the function to be performed after the user has entered the file name. The user answers

L - load absolute program from specified file to memory area specified during program linking. After the whole program has been loaded, control is transferred to the debugger.

G - same as L except that after the whole program has been loaded control is transferred to the start of execution address of the loaded program.

C - compare absolute program in specified file with memory area specified during program linking. If no mismatch is found the message "FILE AND MEMORY IDENTICAL" is output and control is transferred to the debugger. If a mismatch is found control is transferred to the debugger with the (HL) register pointing to the byte where a mismatch was detected and (A) showing the expected contents of the byte. It is not possible to continue the comparison.

S - store absolute program from memory to specified file.
The program limits specified during linking of the original program are used as the limits of the memory area whose contents is transferred to the file. The limits as well as the start of execution address are not changed. After the memory contents have been stored control is transferred to the debugger.
The Load command may be used in conjunction with the Store command to make minor changes ("patches") in an absolute program.

The bootloader expects the disc inserted in drive no. 1 to contain the program file. If drive no. 1 is not ready, drive no. 2 is used instead.

In SPC/1 systems the bootloader normally does not identify itself after initiation but proceeds immediately to perform a load-and-go (G) command on a program file named MIKM of type 0 (zero). The normal interactive sequence, as specified above, is followed by the bootloader if the user presses the BREAK key on the terminal during power up. In this case the bootloader will identify itself and request program name, etc., as soon as the user releases the BREAK key.

Five easy steps to start up your ID-7000 computer:

1) Power up the system beginning with the computer

2) Insert the system disc delivered with the system in disc drive no. 1 (usually the lowest physically placed drive)

3) On the computer front panel, switch to STOP, press RESET (RES), press Debug Call (DC), press RESET (RES), switch to RUN. The debugger should output a ´>´ on the terminal.

4) Press ESC. The debugger should output ´R´. Now enter E000<0XJ (Note: 0 is a zero)

5) The bootloader should respond by outputting its identification. The user may then enter the system program file name (written on the system disc) followed by RETURN and a G command.
In the following example user responses are shown underlined:

```
MIKADOS BOOTLOADER (7080 DISC)
File name MIK
Load/Go/Compare/Store G
```

In the above example the absolute program file MIK of type 0 will be loaded from the system disc and executed.

# Software problem report

Type: ___ program error

___ suggested program change

___ documentation error

___ suggested change in documentation

Date ________

Company: ______________________________

Address: ______________________________

______________________________

______________________________

Attention: ______________________________ Phone: ________

Program or manual title: ______________________________

Version/edition: ______________________________

Description: ______________________________

______________________________

______________________________

______________________________

______________________________

______________________________

All error reports and suggestions should be put forward in writing.

All suggestions become the property of Dansk Data Elektronik ApS (DDE). DDE may accept, modify or reject suggestions without giving any reason.

Before filing an error report the user should attempt to isolate the error, i.e. to write a small program (30 lines or less), which demonstrates the error. A listing of this program should be enclosed.

DDE will register the report, add its comments on the bottom of this page, and return a photocopy to the user.

For internal DDE use: Received ____________. Error number ________

DDE comments ______________________________

______________________________