182.023 Systemnahe Programmierung
1. Test

16. April 2008

KNr. | MNr. | Zuname, Vorname

Ges.)(70) | 1.)(17) | 2.)(18) | 3.)(35) | Zusatzblätter:

Bitte verwenden Sie nur dokumentenechtes Schreibmaterial!

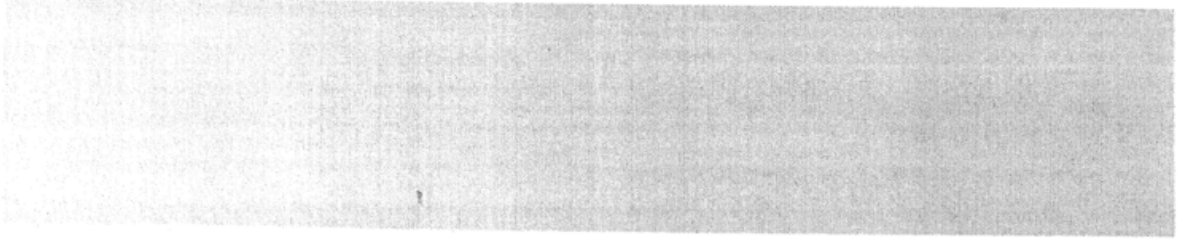
1 C und Makefile (17)

1.1 Funktionsparameter (4)

Schreiben Sie eine Funktion `mySum`, welche die Summe (Typ `int`) der Argumente a und b vom Typ `int` zurückliefert. Ein etwaiger Überlauf braucht nicht berücksichtigt werden.

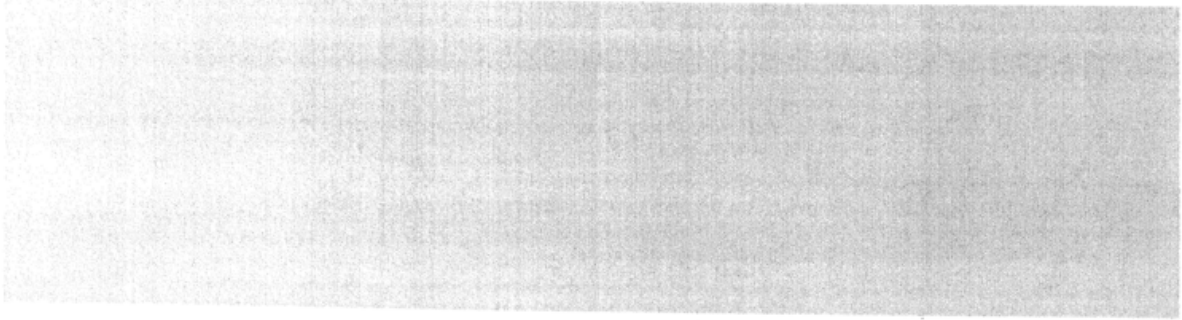
1.1.1 Werteparameter

Implementieren Sie die Funktion `mySum` derart, dass sie nur Wertparameter als Parameter benutzt.



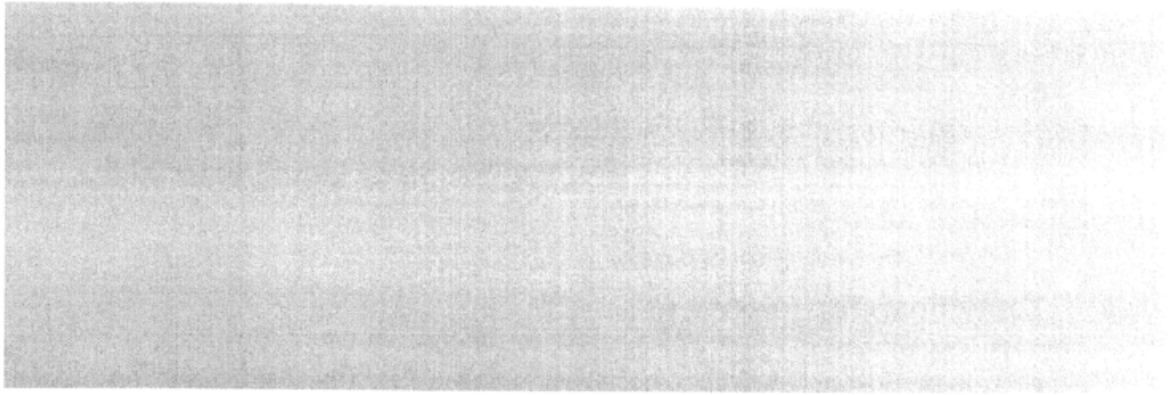
1.1.2 Variablenparameter

Implementieren Sie die Funktion `mySum` derart, dass das erste Argument a als Variablenparameter einerseits Input für den ersten Summanden und andererseits auch Rückgabe der Summe ist. Das zweite Argument b soll weiterhin als Wertparameter übergeben werden.

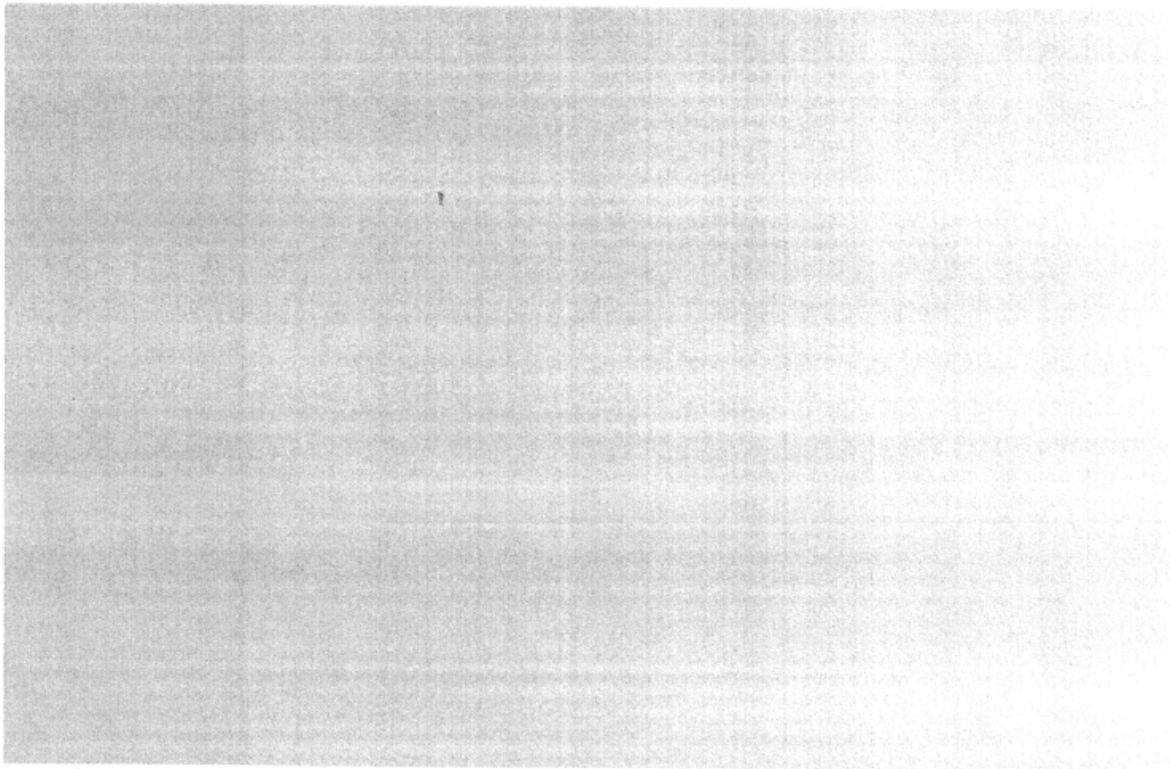


1.2 Strukturierte Datentypen (7)

Deklarieren Sie eine Struktur `ListSW` mit folgenden Elementen: eine Zeichenkette `Name` mit `MAXNAME` echten Zeichen und `Preis` vom Typ `float`. Weiters enthält die Struktur ein Element namens `Next` welches ein Zeiger auf eine Struktur vom Typ `ListSW` ist. Es darf zur Deklaration kein `typedef` verwendet werden!



Definieren Sie weiters drei Variablen für die folgenden drei Softwareprodukte ("OpenOffice" mit Preis 0, "Kirk Office" mit Preis 524,–, und "Manson Office" mit Preis 345,–). Neben der Zuweisung von Name und Preis in allen drei Variablen, sind die drei Elemente weiters als Liste zu verlinken (früher genannte Produkte sollen näher dem Ende der Liste stehen), wobei die zusätzliche Pointer-Variable `list` als Start der List zu definieren und zuzuweisen ist. Vergessen Sie nicht, die Liste korrekt abzuschließen.



1.3 Makefile (6)

Der Sourcecode des Programmes "Prog" besteht aus den beiden Modulen `mod1.c` und `mod2.c`, inklusive der Headerfiles `mod1.h` und `mod2.h`.

Gegeben sei folgendes Makefile:

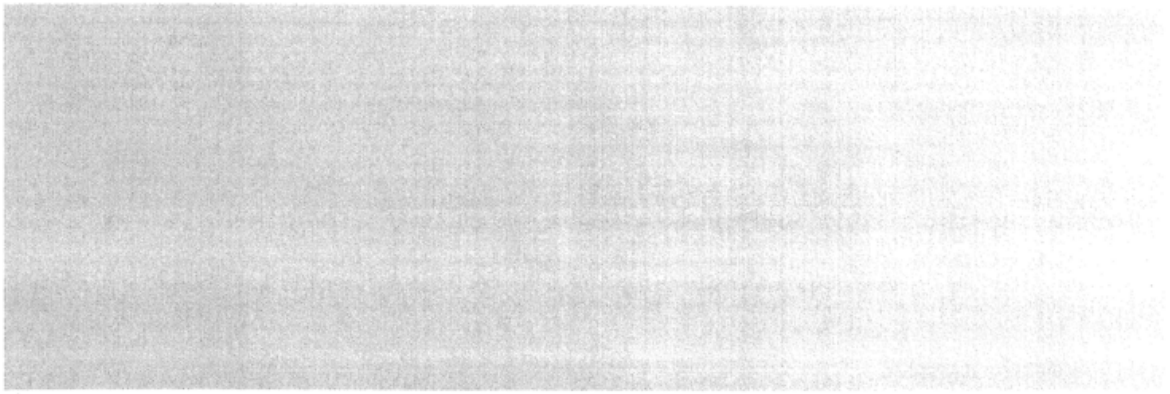
```
all: Prog

Prog: mod1.o mod2.o
    gcc -o Prog mod1.o mod2.o     # cmd1

mod1.o: mod1.c mod1.h mod2.h
    gcc -c mod1.c                 # cmd2

mod2.o: mod2.c mod2.h
    gcc -c mod2.c                 # cmd3
```

Ergänzen Sie zu dem Makefile eine korrekte Regel für das Target "clean":

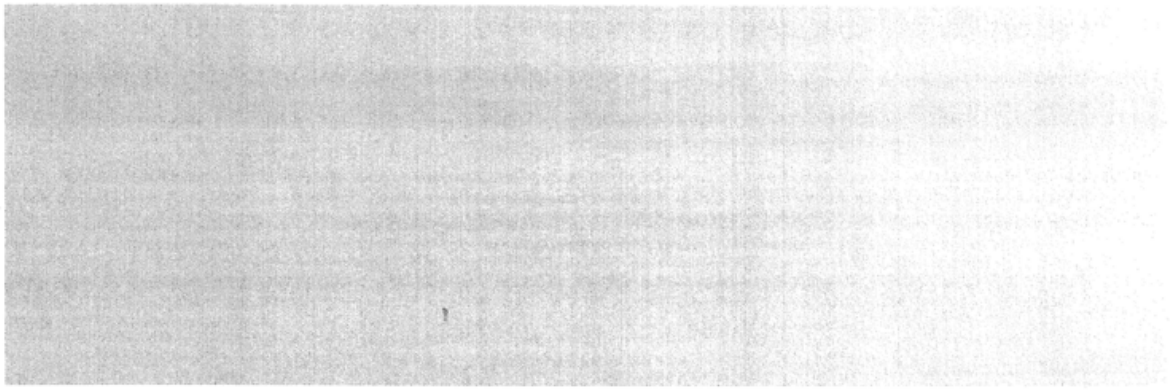
`clean:`

Angenommen, nachdem ein "make all" aufgerufen wurde, wird die Datei `mod2.h` editiert. Welche Aktionen (cmd1, cmd2, cmd3) werden bei einem erneuten "make all" aufgerufen (Reihenfolge der Shellkommando-Aufrufe beachten)? Schreiben Sie "keine", falls kein Shellkommando aufgerufen wird.

Angenommen, nachdem ein "make all" aufgerufen wurde, wird mit einem Hexeditor die Datei `Prog` editiert. Welche Aktionen werden bei einem erneuten "make all" aufgerufen (Reihenfolge der Shellkommando-Aufrufe beachten)? Schreiben Sie "keine", falls kein Shellkommando aufgerufen wird.



Angenommen, es wurden ein "make all" und ein "make clean" aufgerufen. Welche Aktionen werden bei einem erneuten "make all" aufgerufen (Reihenfolge der Shellkommando-Aufrufe beachten)? Schreiben Sie "keine", falls kein Shellkommando aufgerufen wird.



2 Argumentbehandlung (18)

Implementieren Sie das Programm *gerade*, welches überprüft ob die als Argument übergebe Zahl gerade ist. Die Zahl kann dem Programm als Dezimalzahl, Oktalzahl oder als Hexadezimalzahl übergeben werden. Das Programm *gerade* besitzt die folgende Aufrufsyntax:

```
gerade [-b "dezimal" | "oktal" | "hexadezimal"] zahl
```

`-b [ "dezimal" | "oktal" | "hexadezimal"]` Nimmt das Zahlenformat für die nachfolgende Zahl entgegen. Fehlt diese Option, so ist die übergebene Zahl als Dezimalzahl zu interpretieren.

`zahl` Die zu untersuchende Zahl. Ist diese keine gültige Zahl in dem spezifizierten Zahlensystem so ist eine Usage-Meldung auszugeben.

Beispiele für gültige Aufrufe:

```
./gerade -b "dezimal" 123
./gerade -b "hexadezimal" 345F
```

Beispiele für ungültige Aufrufe:

```
./gerade -b "dezimal" 12FF
./gerade -b "dezimal" -b "oktal" 100
./gerade -b "dezimal"
```

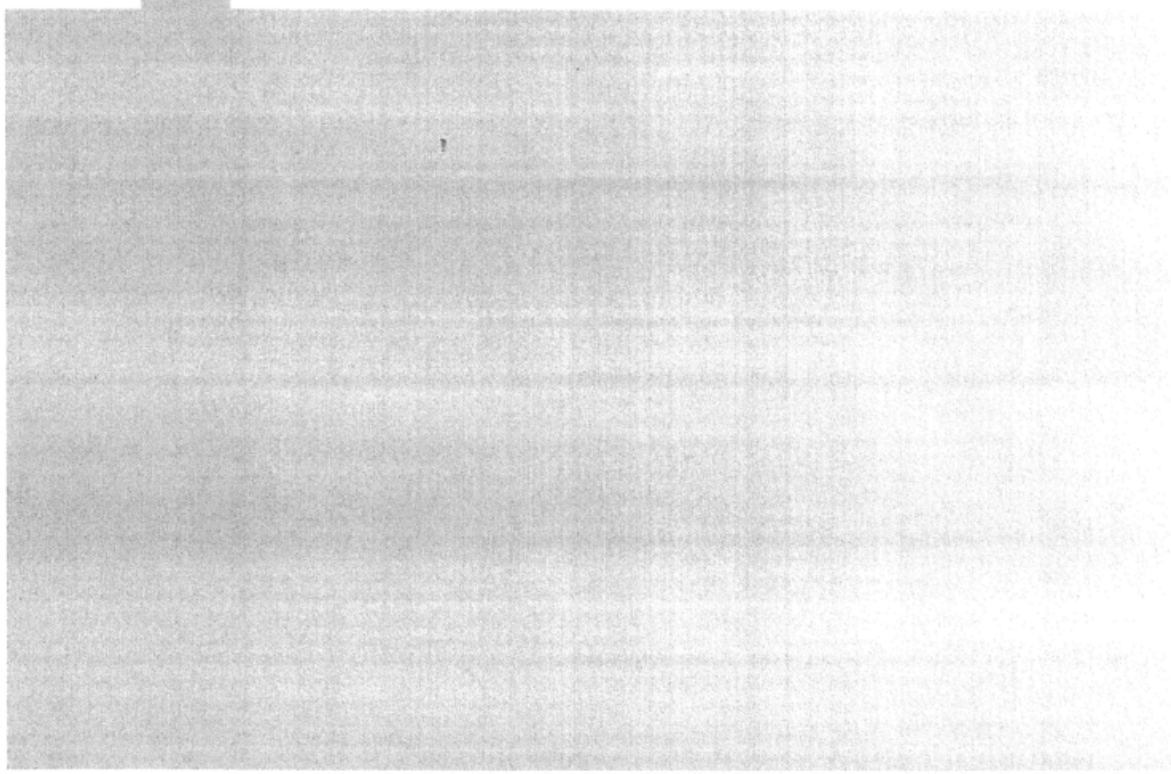
Implementieren Sie die Argumentbehandlung für die oben erklärten Optionen. Bei der Implementierung sind folgende Punkte zu beachten:

- Die Anzahl und das Format der Argumente sind zu überprüfen. Verwenden Sie dazu die Funktion `int getopt(int argc,char *const argv[],const char *optstr)`.
- Bei mehrmaliger Verwendung von Argumenten ist die bereits vorhandene (d.h. nicht zu implementierende) Funktion `void usage(void)` aufzurufen.
- Bei ungültigen Optionen bzw. ungültigen Kombinationen der Optionen ist die Funktion `void usage(void)` zu verwenden.
- Verwenden Sie stets den Pointer `optarg`, um auf die Optionsargumente zuzugreifen.
- Vergessen Sie nicht die Fehlerbehandlung bei Funktionsaufrufen (z.B. bei der Konvertierung von Argumenten mit `strtol`)!
- Geben Sie am Ende des Programms die Meldung `"Zahl gerade"` oder `"Zahl ungerade"` aus.

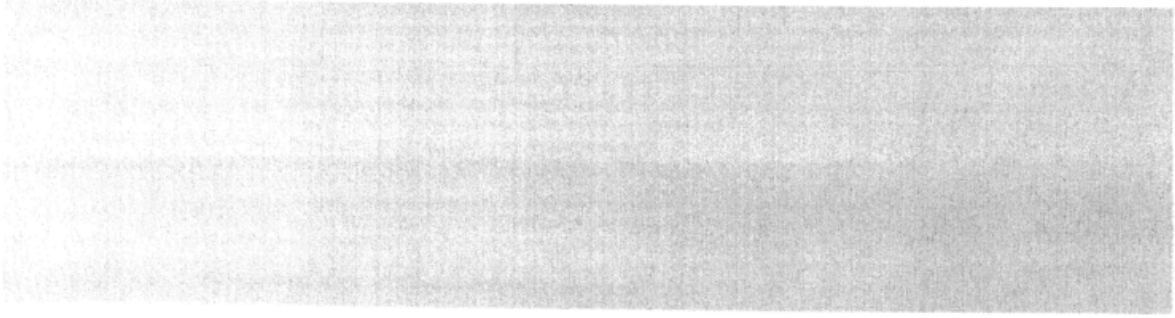
```
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <assert.h>
```

```
#include <errno.h>

#define EXIT_FAILURE  -1

void usage(void) {
    (void) fprintf(stderr,"Usage: gerade [-b {\"dezimal\" | \"oktal\" | \"hexadezimal\"}] zah
    exit(EXIT_FAILURE);
}


int main(int argc, char **argv) { /* Deklaration und Initialisierung

of variables */
```

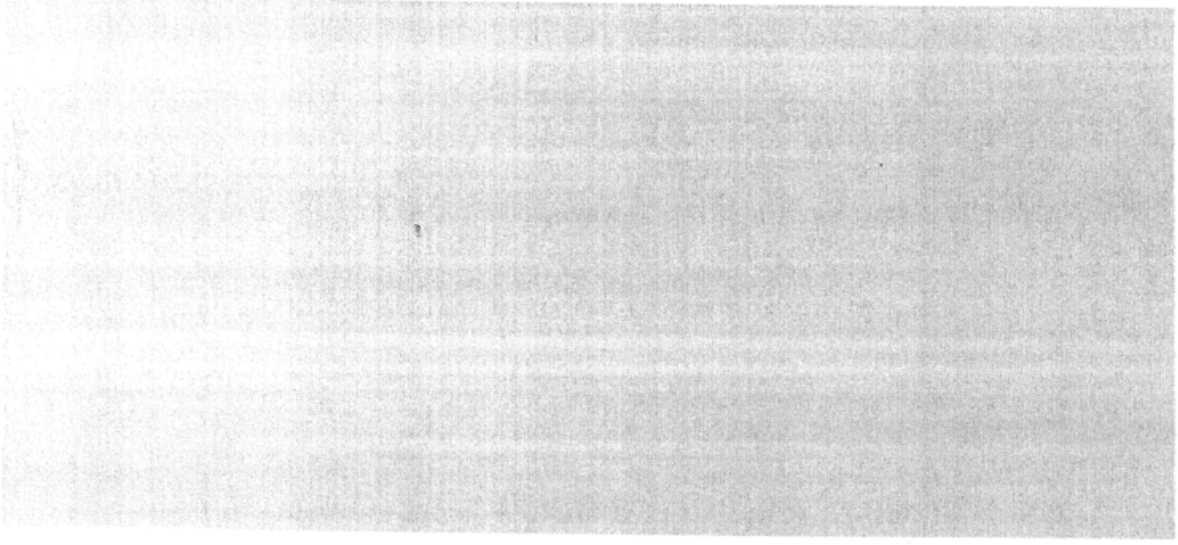
```
    while ((        =getopt(                        ))              ) {

    switch (        ) {

    case        :
```



```
    case '?':
      usage();
      break;
    default:
      assert(0);
    }
}
/* Korrekte Anzahl der Argument pruefen */
```



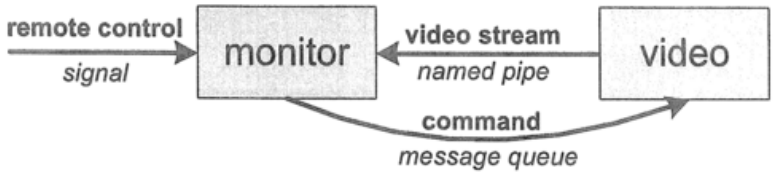
```
/* Testen ob gerade Zahl */
```



```
return 0; }
```

3 Message Queue, Named Pipe, Signalbehandlung (35)

Implementieren Sie die Kommunikation zwischen einem Videorecorder und einem Fernseher mittels einer Message Queue und einer Named Pipe. Mittels einer Fernbedienung können Kommandos über den Fernseher an den Videorecorder gesendet werden. Folgendes Bild veranschaulicht den Kommunikationsablauf:



Es gibt zwei Prozesse, der erste erfüllt die Funktion eines Fernsehers (*monitor*), der zweite die eines Videorecorders (*video*). Der Prozess *video* liest von einem Medium Videodaten und macht diese dem *monitor* Prozess über eine Named Pipe als video stream verfügbar. Der *monitor* Prozess liest die Daten von der Named Pipe und zeigt diese an. Weiters implementiert der *monitor* eine Signalbehandlungsroutine, mit der der Benutzer das Abspielen des video streams anhalten oder fortsetzen kann. Dazu wird vom *monitor* Prozess das Benutzersignal *SIGUSR1* abgefangen und in der Form eines Kommandos an den *video* Prozess weitergeleitet. Das Kommando wird in eine Nachricht eingebettet, die vom *monitor* Prozess mittels eine Message Queue zum *video* Prozess übertragen wird.

Benutzen Sie das Programmgerüst und implementieren Sie die beiden Prozesse *video* und *monitor*. Beachten Sie folgende Punkte:

- Tragen Sie Ihre Lösung in die grauen Felder ein. Nicht jedes Feld muss beschrieben werden.
- Die Datei *library.h* enthält alle gemeinsamen Definitionen. Weiters stellt sie die beiden Funktionen `read_char_from_tape` und `write_char_to_screen` zum Lesen und Schreiben der entsprechenden Geräte bereit.
- Der Videorecorder kann entweder angehalten oder beim Abspielen eines streams sein.
- Nach dem Programmstart ist der Videorecorder in angehaltenem Zustand.
- Der aktuelle Zustand des *video* Prozesses kommt an drei Stellen vor: in den beiden `video_command_t` Variablen des *monitor* und des *video* Prozesses, die zum Senden und Empfangen eines Kommandos dienen und in der `video_state` Variable, die den Zustand des *video* Prozesses speichert.
- Der *video* Prozess legt die Named Pipe an. Wird der *monitor* Prozess vor dem *video* Prozess gestartet, terminiert dieser, weil noch keine Named Pipe vorhanden ist.
- Die Message Queue wird neu angelegt oder – falls schon eine Message Queue im System vorhanden ist – wird diese benutzt.
- Der *video* Prozess ist verantwortlich, dass die Ressourcen korrekt dem System zurückgegeben werden.
- Nur der Benutzer hat Zugriffsrechte auf die Ressourcen.
- In der Signalbehandlungsroutine darf nicht gewartet werden.
- Vergessen Sie nicht die Verständnisfragen am Ende zu beantworten!

3.1 library.h (5)

```
#include ... /* All functions from the SysNah cheat sheet are available. */

/* Names and permissions for message queue and named pipe */

#define PIPENAME "pipe"

#define PIPEPERM

#define MQKEY    9895

#define MQPERM

/* Message type for monitor to video message*/

#define VIDEOCOMMANDMSGTYPE 12

/* Possible states for the video process */

#define STOP 0

#define PLAY 1

/* Message structure for monitor to video message*/

struct video_command_t

{

};

/* Display an error message and exit */
void BailOut(const char *szMessage);

/* Read a byte from the video tape */
char read_char_from_tape( void );

/* Write a byte to the framebuffer */
void write_char_to_screen( char c );
```

3.2 monitor.c (12)

```
#include "library.h"

/* Definition section of global variables */
[redacted]

/* Signal handler for SIGUSR1 signal*/

void remote_control( int sig )

{

  /* set message type of command message*/
[redacted]

  /* toggle video stream command play/stop */
[redacted]

  /* send message to video process */

  if( msgsnd([redacted], [redacted], [redacted], \
                [redacted] ) < [redacted] )

  {
        BailOut("Cannot send command.\n");
  }
}

/* Main program */
int main( int argc, char *args[] )
{
  /* Local variables*/
[redacted]
```

```
    /* Initialize video command state */

    /* Open the pipe to read the stream */
    if( (        =fopen(          ,          )) ==          )
    {
        BailOut("Cannot open named pipe.\n");
    }

    /* Allocate a message queue */
    if( (mqid=msgget(          ,          )) <    )
    {
        BailOut("Cannot access message queue.\n");
    }
    /* Register signal handler for the SIGUSR1 signal */
    (void) signal(          ,          );
    /* Read from the stream and forward the received data */
    while( (          =(        ) fgetc(          )) !=        )
    {
        write_char_to_screen(          );
    }
    /* Close the pipe */
    if( fclose(          ) ==        )
    {
        pipe =          ;
        BailOut("Cannot close named pipe\n");
    }
    pipe =          ;
    /* End of program */
    return          ;
}
```

3.3 video.c (14)

```
#include "library.h"

/* Definition section of global variables */
[redacted]

static int terminate = 0;

/* Signal handler for the SIGTERM signal */

void terminator(int sig)

{

    terminate = [redacted];

}

/* Main program */

int main( int argc, char *args[] )

{

  int video_state = [redacted];


  /* Allocate named pipe */

  if( mkfifo([redacted], [redacted]) < 0 )

  {
    BailOut("Cannot create pipe.\n");
  }


  /* Allocate a message queue */

  if( ([redacted]=msgget([redacted], [redacted])) < [redacted] )

  {
    BailOut("Cannot access message queue.\n");
  }
```

```
/* Open the pipe to write to the stream */
if( (          = fopen(              ,       )) ==                  )
{
  BailOut("Cannot open named pipe.\n");
}
/* Register signal handler for the SIGTERM signal */
signal(               ,                  );
/* Main program loop */
while( !terminate )
{
  /* If the video stream is set to play, write a char to the video stream */
  if( video_state ==             )
  {
      if( fputc(                                    ,         ) ==         )
      {
        BailOut("Cannot write to pipe.\n");
      }
  }
  /* If a command is received from the message queue, change the video's state */
  if( !(msgrcv(           ,             ,                   , \
                                 ,                   ) <       ))
  {

  }
}
```

/****   Turn page one last time!   ***/

```
/* Delete the message queue */

if( msgctl(          ,                    ,                                    ) <      )
{
    BailOut("Cannot remove message queue.\n");
}

/* Close the pipe */

if( fclose(                              ) ==      )
{
    pipe =                    ;
    BailOut("Cannot close named pipe\n");
}
pipe =                    ;


/* Remove named pipe */

if( remove(PIPENAME) !=       )
{
    BailOut("cannot remove pipe\n");
}

/* End of program */

return                    ;
}
```

3.4 Verständnisfragen (4)

(a) Auf welche Art synchronisiert sich der *monitor* Prozess zur Named Pipe?

- ○ Blocking read
- ○ Non-blocking read
- ○ Busy waiting

(b) Auf welche Art synchronisiert sich der *video* Prozess zur Message Queue?

- ○ Blocking read
- ○ Non-blocking read
- ○ Busy waiting