

# MiniComDriver

## Dokumentation zum überarbeiteten Minicommander-Treiber

E.Bedau, K.Klein, B.Kiric

ICM-LowLevelGroup

Stand: 18.09.2004

## Inhaltsverzeichnis

Ausgangssituation.....	1
Redesign des Minicommander-Treibers.....	3
Integration des MiniComDrivers in das Framework.....	6
Wie besorgt man sich den Kontext des MainDispatchers?.....	9
Beseitigte Fehler/Schwachstellen.....	9
Redundante Funktionen und Variablen.....	9
Löschen der Default-termios-Einstellungen.....	10
Falsche Auswertung der Tastendrücke.....	10
Verallgemeinerte Initialisierung der Ser. Schnittstelle.....	12
Ausblick.....	13
MiniCommander-Emulator.....	13

## Ausgangssituation

Der ursprünglich zur Verfügung gestellte Treiber für den Minicommander weist eine enge Kopplung zwischen Empfangs- und Verarbeitungslogik auf. Das folgende Codefragment soll diesen Sachverhalt verdeutlichen.

```
//empfangt auf der seriellen Schnittstelle die Minicommanderpakete
//->Empfangslogik
int serReceive(void)
{
    ...
    while(1)
    {
        read(mcmd_fd, &data, 1);
        ...
        InterpretTelegram();
        ...
    }
} //end function serRecieve(void)

//interpretiert die empfangenen Datenpakete
//und leitet diese an Empfänger weiter
void InterpretTelegram(void)
{
    ...
    switch(telegram.cmdH)
```

```

{
    case 0x35: // Key Scan or Commander present
    if (telegram.cmdL == 0x35) // Key Scan
    {
        // Engineering Mode
        if ((telegram.dataL & 0x06) == 0x06) // Key C+ E pressed
            key35State = getTimestamp();
        else
            if (key35State != 0)
            {
                sendKeyToPh (0xf0ca); //F13
                keyCState = 0;
                keyEState = 0;
                key35State = 0;
            }

            if (telegram.dataH & 0x02) // Key A gedrückt
            {
                keyAState = getTimestamp();
            }
            else
            if (keyAState != 0)
            {
                if (getTimestamp() - keyAState < opt1)
                    sendKeyToPh (0xf0be); //F1
                else
                    sendKeyToPh (0xf0c4); //F7
                keyAState = 0;
            }

            ...
        }
        ...
    } //end switch(telegram.cmdH)
} //end InterpretTelegram(void)

//generiert und sendet Photon-Key-Events mit key als Parameter
void sendKeyToPh( long key) { ... }

```

Eine derart enge Kopplung kann bei Erweiterungen oder Änderungen Schwierigkeiten bereiten, da stets der Code der Minicommander-Applikation angepaßt werden müßte. Dies wiederum hat zur Folge, daß der gesamte Code jedes Mal neu getestet werden müßte.

Eine Trennung zwischen Empfangs- und Verarbeitungslogik kann hier Abhilfe schaffen. Der Empfangsteil des Minicommanders stellt die statische Komponente des System dar.

Lediglich in der Verarbeitung der Tastatureingaben kann es je nach Art der übergeordneten Komponenten unterschiedliche Ausprägungen geben. So wäre es beispielsweise denkbar, daß eine Komponente die Tastaturereignisse in Form von Konsolenausgaben haben möchte. Eine weitere Komponente hätte gerne die Tastatureingaben in Form von Photon-Key-Events. Und eine dritte Applikation möchte gerne die Minicommander-Eingaben in Form von MOST-Events präsentiert bekommen.

All diese unterschiedlichen Anforderungen lassen sich erheblich einfacher und übersichtlicher realisieren, wenn die Verarbeitungslogik als eigenständige Komponente angesehen wird.

Eine weitere Schwachstelle des bestehenden Minicommander-Codes ist die Verwendung globaler Variablen. Die Verwendung globaler Variablen erzeugt globale Abhängigkeiten zwischen den einzelnen Funktionen und erhöht somit die Komplexität der gesamten Applikation. Ziel eines jeden Entwurfs ist jedoch, die Komplexität so gering wie möglich zu halten, um flexibel gegenüber Erweiterungen, Änderungen etc. zu sein.

## **Redesign des Minicommander-Treibers**

Wie im vorhergehenden Abschnitt bereits erwähnt, soll die Minicommander-Treiberapplikation in die beiden Komponenten Empfangslogik und Verarbeitungslogik zerlegt werden.

Das nachfolgende Klassendiagramm (Abbildung 1) gibt einen Überblick über die einzelnen Komponenten.

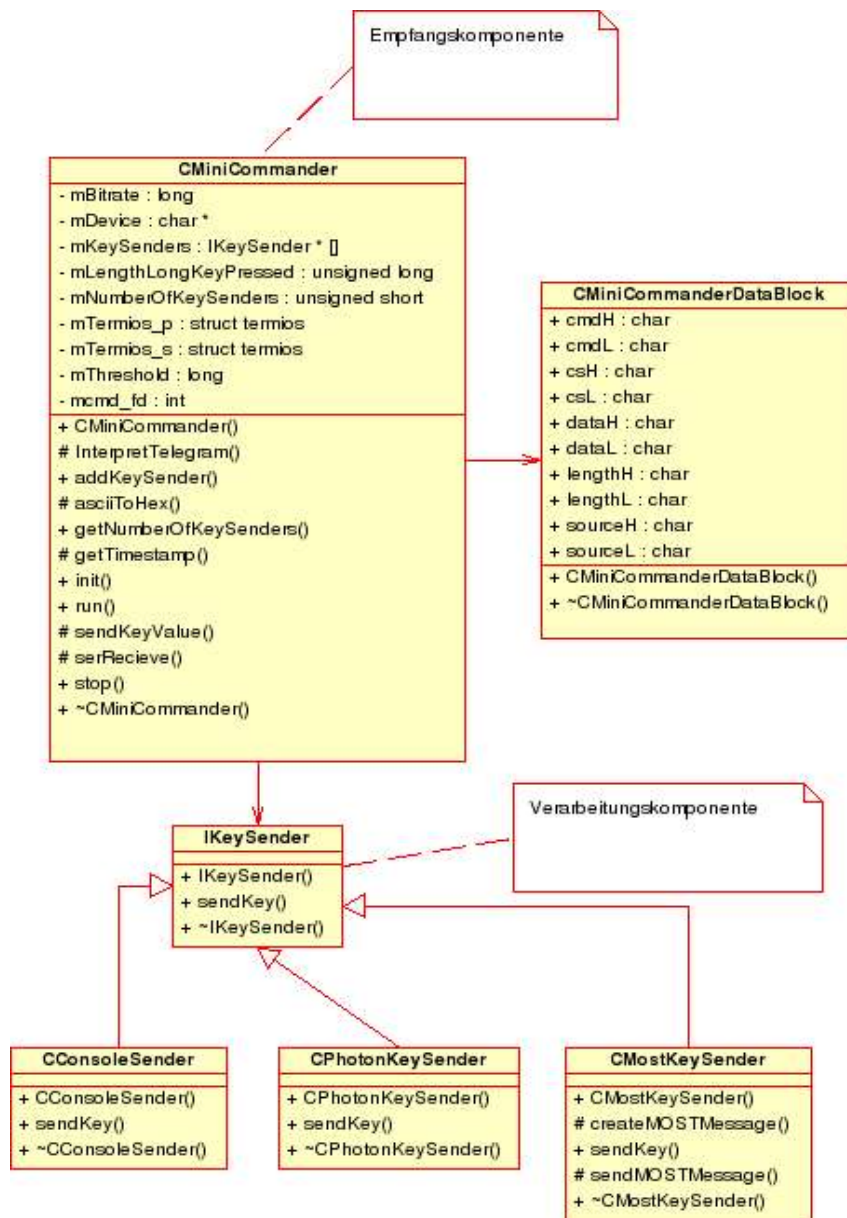


Abbildung 1: Klassendiagramm MiniComDriver

Die Methode sendKeyToPh wurde umbenannt in sendKey und ausgelagert in die abstrakte Klasse IKeySender.

Eine CMiniCommander-Instanz kann nun mehrere (max. 10) konkrete IKeySender-Objekte bedienen. Diese müssen mit Hilfe der Methode addKeySender der CMiniCommander-Instanz bekanntgemacht werden.

Empfängt die CMiniCommander-Instanz ein Datenpaket (CminiCommanderDataBlock) an der seriellen Schnittstelle, wird anschließend die private Methode sendKeyValue aufgerufen, die das Datenpaket an alle registrierten IKeySender-Objekte weiterleitet.

```

void CMiniCommander::sendKeyValue( long keyValue )
{
    for( int i=0; i < mNumberOfKeySenders; i++ )
    {
        mKeySenders[i]->sendKey( keyValue );
    }
}

//end method sendKeyValue( long keyValue )

```

Dieses Design ermöglicht es, neue Verarbeitungskomponenten nachträglich, ohne großen Aufwand (sogar zur Laufzeit) in die bestehende Applikation einzufügen.

Ein Hauptprogramm für eine Minicommander-Applikation könnte wie folgt aussehen:

```

#include <cstdlib>
#include <stdio.h>

#include "PhotonKeySender.h"
#include "ConsoleSender.h"
#include "MiniCommander.h"

int main(int argc, char *argv[]) {

    CMiniCommander myCommander("/dev/ser1", 500, 2, 57600 );
    CPhotonKeySender phKeySender; //erzeuge Photon-Key-Sender
    CConsoleSender consoleSender; //erzeuge ConsolenSender

    //Verarbeitungskomponenten an Empfangskomponente uebergeben
    myCommander.addKeySender( consoleSender );
    myCommander.addKeySender( phKeySender );

    //pruefe, ob Verarbeitungskomponenten vorhanden
    if( myCommander.getNumberOfKeySenders() > 0 )
    {
        //starte seriellen Empfang
        myCommander.run();
    }
    //no key-senders available -> no recieptient for minicommander-key-events
    else
    {
        printf("Minicommander-Driver has no Key-Recievers!\n");
        return -1;
    }

    return EXIT_SUCCESS;
}

//end main(...)

```

## Integration des MiniComDrivers in das Framework

In der nächsten Stufe der MiniComDriver-Implementierung soll der Treiber des Minicommanders in das Framework integriert werden. Die Tastatureingaben sollen in Form von MOST-Nachrichten an die übergeordneten Applikation gesendet werden.

Die Nachrichten werden zu diesem Zweck in die Warteschlange des MainDispatchers gestellt.

Dieser leitet sie dann an die HMI weiter.

Zur Realisierung dieser Vorgehensweise muß zum einen eine weitere KeySender-Klasse implementiert werden, welche die Tastatureingaben in MOST-Events umwandelt und an den MainDispatcher-schickt. Hierfür wird für den Minicommanders ein Pseudo-MOST-Device definiert. Die Definition dieses Pseudo-MOST-Devices muß Angaben über die Device-Id, die zur Verfügung gestellten Funktionen, Daten und Datentypen enthalten. Nachfolgend ist die Header-Datei abgebildet, welche diese Definition enthält.

```
#ifndef _LOWLEVEL_EVENTS_MINICOM_H
#define _LOWLEVEL_EVENTS_MINICOM_H

#define DEVICE_ID                0x0002        //we are a non most device
#define FBLOCK_ID                0xF0          //suppl. spec. FBLOCK_ID
#define INSTANCE_ID              0x01
#define FUNCTION_ID_BUTTON_SHORT 0x0F00
#define FUNCTION_ID_BUTTON_LONG  0x0F01
#define FUNCTION_ID_WHEEL        0x0F02
#define FUNCTION_ID_WHEEL_PUSHED 0x0F03
#define FUNCTION_ID_WHEEL_TURN_PUSHED 0x0F04

#define OP_TYPE_GET              1 //get

#define DATA_BUTTON_A          0x41 //ASCII-Code for A
#define DATA_BUTTON_B          0x42 //ASCII-Code for B
#define DATA_BUTTON_C          0x43 //ASCII-Code for C
#define DATA_BUTTON_D          0x44 //ASCII-Code for D
#define DATA_BUTTON_E          0x45 //ASCII-Code for E
#define DATA_BUTTON_F          0x46 //ASCII-Code for F
#define DATA_WHEEL_DIR_RIGHT   0x02
#define DATA_WHEEL_DIR_LEFT    0x04
#define DATA_WHEEL_PRESSED_SHORT 0x08
#define DATA_WHEEL_PRESSED_LONG 0x10
```

```
#endif
```

Zum anderen benötigt die MiniComDriver-Applikation den Kontext des MainDispatchers, um die EventQueue im Shared-Memory bedienen zu können. Aus diesem Grunde wurde eine Klasse CMiniComDriver entworfen, die von IRunnable erbt und die entsprechenden Methoden implementiert. Auf diese Weise kann eine CMiniComDriver-Instanz von der Admin-Komponente erzeugt und dem Prozess-Starter übergeben werden. Der Prozess-Starter erzeugt für jede Komponente einen eigenen Thread. Auf diese Weise werden alle System-Komponenten samt Kontext, auch der MainDispatcher, erzeugt. Für den Minicommander wurde auf eine Implementierung des CComponent-Interfaces verzichtet, da keine Queues benötigt werden. Der Minicommander soll ja lediglich andere Komponenten mit Tastatureingaben versorgen. Umgekehrt ist dies nicht erforderlich. Das nachfolgende Klassendiagramm (Abbildung 2) soll das Zusammenwirken der einzelnen Komponenten noch einmal verdeutlichen.

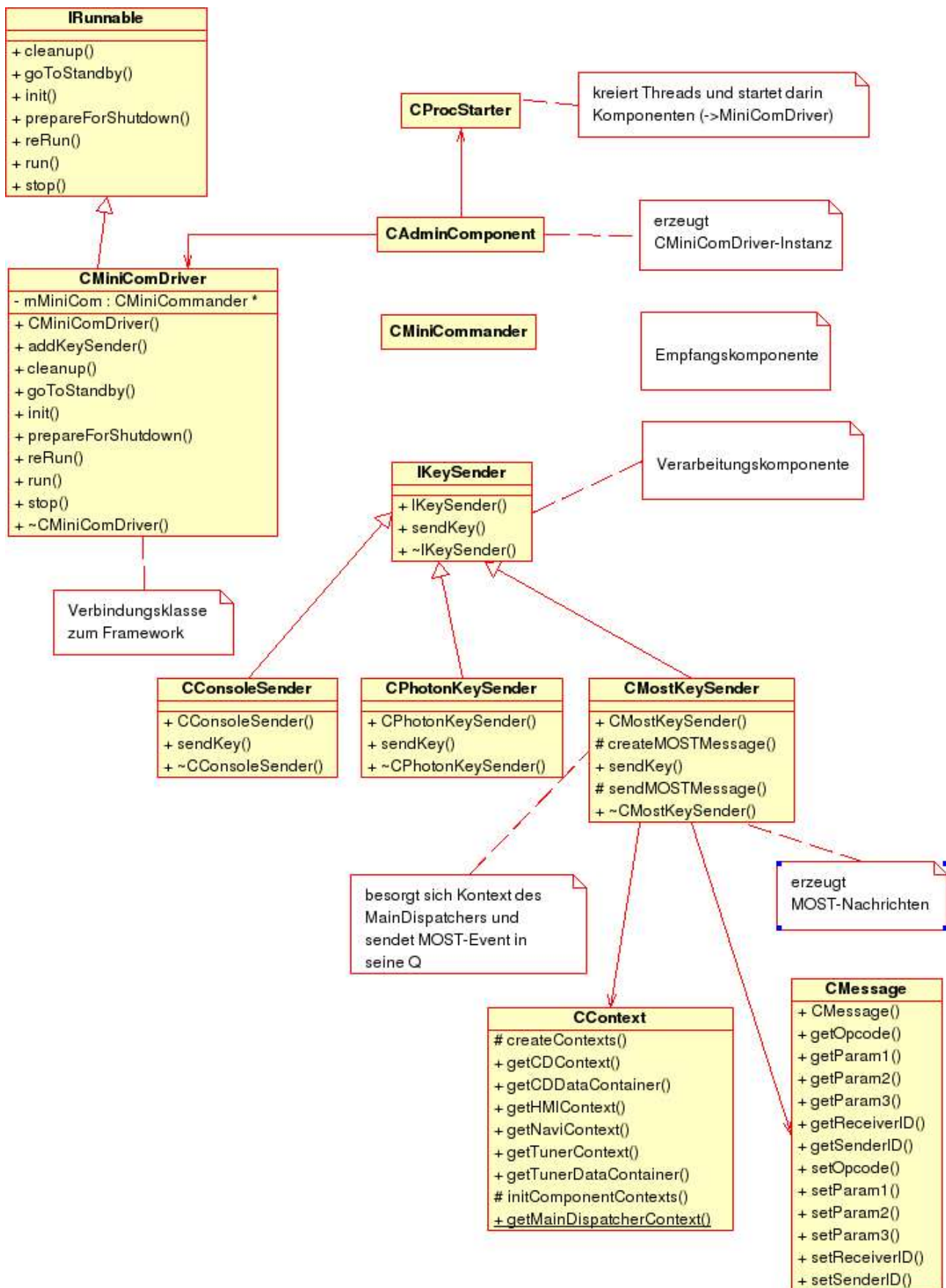


Abbildung 2: Klassendiagramm MiniComDriver integriert in Framework



Nachfolgend ist ein Codeausschnitt aus der Admin-Komponente (CAdminComponent) zu sehen, welcher die Erzeugung einer CMiniComDriver-Instanz skizziert.

```
...
void CAdminComponent::run(void) {
    cout << "Admin: running..." << endl;
    cout << "Admin: Initializing Component contexts..." << endl;
    initComponentsContexts();
    cout << "Admin: contexts initialized." << endl;

    //erzeuge MiniCommander-Driver-Instanz
    CMiniComDriver miniCom("/dev/ser1", 500, 2, 57600);

    //add console key sender -> for debugging
    //add MOST for MostSender
    miniCom.init( CONSOLE | MOST );

    CTunerComponent tuner(CContext::getTunerContext());
    CCDComponent cd(CContext::getCDContext());
    CNavicomponent navi(CContext::getNaviContext());
    CHmiComponent hmi(CContext::getHMIContext());
    CMainDispatcher mdsp(CContext::getMainDispatcherContext());

    mStarter.addProcess(&mdsp);

    //wir moechten unmittelbar nach dem MainDispatcher gestartet werden
    mStarter.addProcess(&miniCom);

    mStarter.addProcess(&tuner);
    mStarter.addProcess(&cd);
    mStarter.addProcess(&navi);
    mStarter.addProcess(&hmi);
    mStarter.startAll();
}
```

### ***Wie besorgt man sich den Kontext des MainDispatchers?***

Nachdem der MostKeySender die empfangenen Daten in ein MOST-Event konvertiert hat, erfolgt dessen Übertragung. Hierfür ruft die MOSTKeySender-Instanz ihre interne Methode sendMOSTMessage auf. Diese Methode sendet das MOST-Event in die Queue des MainDispatchers. Um an die Queue des MainDispatchers zu gelangen, besorgt sich die Methode einfach dessen Kontext. (siehe Codefragment)

```
//returns 0 for success, otherwise 0
void CMostKeySender::sendMOSTMessage()
{
    CContext::getMainDispatcherContext().getNormalQueue().add(mMostMsg, false);
} //end method sendMOSTMessage( CMessage mostMsg)
```

## **Beseitigte Fehler/Schwachstellen**

### ***Redundante Funktionen und Variablen***

Während des Redesigns der MiniCommander-Applikation wurde eine redundante Methode (unsigned char GetByteMcmd(void)) identifiziert. Da diese lediglich in einem speziellen

Testmodus des ursprünglichen Autors Verwendung fand, entschlossen wir uns, sie zu beseitigen.

Neben nicht-benötigten Funktionen, konnten auch unbenutzte Variablen beseitigt werden. Dabei fielen drei Variablen (`pthread_mutex_t mutex`, `unsigned short mcmdRecBufRd`, `unsigned char mcmdRecievedTelegr[5]`) und eine Enum weg.

Des weiteren konnte die Anzahl der globalen Variablen (ehemals 21) auf 9 private Klassenvariablen und eine Zahl von lokalen Variablen reduziert werden.

## **Löschen der Default-termios-Einstellungen**

Bei der Initialisierung der seriellen Schnittstelle wurden die aktuellen Einstellungen der seriellen Schnittstelle ohne Sicherung einfach überschrieben. Im Falle eines Abbruchs der Applikation müßte das System neu gestartet werden, da nur auf diese Weise die Konfiguration seriellen Schnittstelle restauriert werden kann.

Durch die Einführung der beiden Klassenvariablen `mTermios_s` (zum Sichern) und `mTermios_p` (aktuelle Einstellungen) wurde dieses Problem beseitigt. Wird die Applikation vorzeitig beendet, erfolgt die Rücksicherung der Termios-Einstellungen.

**Problem: Wie kann man einen Signal-Handler innerhalb einer Klasse verwenden?**

Der Signal-Handler möchte als callback eine globale Methode haben. In einer globalen Methode hat man aber keinen Zugriff auf Klasseninstanzen. Jedenfalls nicht, ohne den objektorientierten Ansatz zu verletzen.

## **Falsche Auswertung der Tastendrücke**

Bei der Analyse des Programmcodes, welcher für die Interpretation der gesendeten Tastatureingaben verantwortlich ist, wurden Unstimmigkeiten bezüglich der Prüfung der Eingaben festgestellt. So wurde zum Beispiel ein Druck der Taste A stets als Tastendruck auf F interpretiert. Laut Kurzpflichtenheft Mini-Commander Version 1.00 ist ein Datenpaket folgendermaßen aufgebaut (hier ist ein Key-Scan-Result-Paket dargestellt):

<i><b>STX</b></i>	<i><b>Length [1..0]</b></i>		<i><b>Source [1..0]</b></i>		<i><b>CMD [1..0]</b></i>		<i><b>Data [1..0]</b></i>		<i><b>CS [1..0]</b></i>		<i><b>ETX</b></i>
0x02	0x30	0x42	0x46	0x30	0x35	0x35	0xXX	0xXX	0xXX	0xXX	0x0D

Die einzelnen Tasten werden in den Datenbytes wie folgt codiert:

$$\text{Data}[0] = 0x30 + (D*8) + (C*4) + (B*2) + (A*1)$$

$$\text{Data}[1] = 0x30 + (\text{Incr}*4) + (F*2) + (E*1)$$

Ein Druck wird durch eine 1 beschrieben. Das Loslassen ist mit 0 definiert.

Wurde die Taste A gedrückt, ergeben sich folgendes Datenbytes:

$$\text{Data}[0] = 0x30 + (0*8) + (0*4) + (0*2) + (1*1) = 0x31$$

$$\text{Data}[1] = 0x30 + (0*4) + (0*2) + (0*1) = 0x30$$

Beim Loslassen der Taste A haben beide Datenbytes den Wert 0x30.

$$\text{Data}[0] = 0x30 + (0*8) + (0*4) + (0*2) + (0*1) = 0x30$$

$$\text{Data}[1] = 0x30 + (0*4) + (0*2) + (0*1) = 0x30$$

Im ursprünglichen Code sah die Prüfung für die Taste A folgendermaßen aus:

```
...  
if (telegram.dataH & 0x02) // Key A pressed  
{ ... }
```

Das Highbyte ist jedoch beim Druck auf Taste A gleich 0x30. Folglich ist die oben genannte Bedingung nicht erfüllt und es wird nicht A erkannt.

Das Highbyte erfüllt diese Bedingung bei gedrückter F -Taste (Berechnung siehe oben).

Richtigerweise muß für die Taste A hier das Lowbyte auf folgende Bedingung geprüft werden:

```
...  
if (telegram.dataL & 0x01) // Key A pressed  
{ ... }
```

## Verallgemeinerte Initialisierung der Ser. Schnittstelle

Die Initialisierung der seriellen Schnittstelle erfolgt im wesentlichen in zwei Schritten:

- Beschaffen eines File-Descriptors auf die Gerätedatei der seriellen Schnittstelle

```
...
mcmd_fd = open('/dev/ser1', O_RDWR, 0);
...
```

- Beschaffen einer termios-Struktur. Anpassung der termios-Struktur an die vereinbarten Übertragungsparameter.

```
...if( mcmd_fd != -1 )
{
    struct termios mTermios_p;
    memset (&mTermios_p, 0x00, sizeof (mTermios_p));
    tcgetattr( mcmd_fd, &mTermios_s );    //get act. Termios-settings

    // set serial device parameters
    mTermios_p.c_iflag &= ~( ICRNL | INLCR | ISTRIP | IXOFF | IXON );    //input flags
    mTermios_p.c_oflag &= ~( OPOST );    //output flags
    //control flags
    mTermios_p.c_cc[VMIN] = 1;    //no blocking read
    mTermios_p.c_cc[VTIME] = 0;    //no blocking read
#ifdef IHFOW    //not defined in POSIX -> QNX
    mTermios_p.c_cflag &= ~ IHFLOW;    //no flow control
#endif
#ifdef OHFLOW    //not defined in POSIX -> QNX
    mTermios_p.c_cflag &= ~ OHFLOW;    //no flow control
#endif
#ifdef CRTSCTS    //not defined in POSIX -> LINUX
    mTermios_p.c_cflag &= ~ CRTSCTS;    //no flow control
#endif
    mTermios_p.c_cflag &= ~( PARENB );    //no parity
    mTermios_p.c_lflag &= ~( ECHO | ICANON | ECHOE | ECHOK | ECHONL | IEXTEN );
    // baud, 8N1
    mTermios_p.c_cflag |= ( CREAD );
    mTermios_p.c_cflag |= (CS8 | CSIZE);
    cfsetispeed(&mTermios_p, speed);    //set DTE speed
    cfsetospeed(&mTermios_p, speed);    //set DCE speed
    tcsetattr(mcmd_fd, TCSADRAIN, &mTermios_p);    //apply all changes
}...
```

In der ursprünglichen Version des Minicommander-Codes wurden zur Deaktivierung der Flußkontrolle die Flags OHFLOW und IHFLOW deaktiviert.

Diese Flags sind jedoch nicht im POSIX-Standard definiert, weshalb es beim Compilieren unter einem anderen Betriebssystem, als QNX, zu einem Fehler kommt.

Der Name zur Deaktivierung der Flußkontrolle kann von Betriebssystem zu Betriebssystem variieren. Unter Linux ist beispielsweise das CSRTSCTS-Flag hierfür verantwortlich. Um seinen Code möglichst portabel zu halten, sollten diese Betriebssystem-spezifischen Abhängigkeiten in `#ifdef ... #endif` – Anweisungen verpackt werden. Somit kann zur Compile-Zeit entschieden werden, welches Flag für welches Betriebssystem definiert ist.

## Ausblick

### ***MiniCommander-Emulator***

Für Testzwecke ist geplant, im weiteren Verlauf dieses Semesters einen MiniCommander-Emulator zu liefern, der sowohl unter Linux, als auch unter QNX (evtl. auch Windows in Kombination mit cygwin) einsatzfähig ist. Hierfür werden lediglich ein serielles Kabel (gekreuzt) sowie zwei serielle Schnittstellen benötigt.