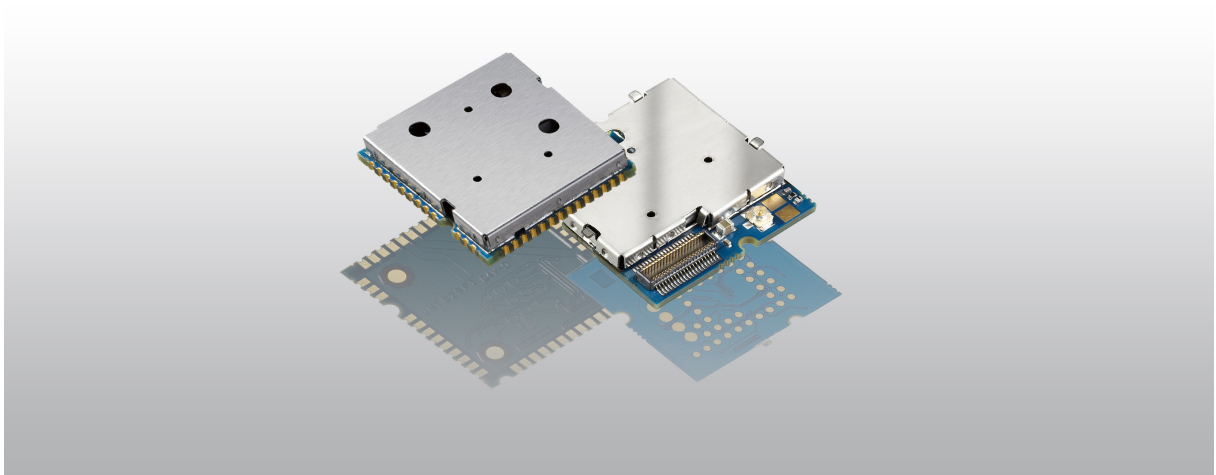


MUX 0710

LINUX IMPLEMENTATION EXAMPLE



~ Freedom of speech
for smart machines ~

FICHE RECAPITULATIVE / REVISION HISTORY

Ed	Date <i>Date</i>	Référence <i>Reference</i>	Pages modifiées / <i>Changed pages</i>	Observations <i>Comments</i>
1	24/08/2009	URD1– OTL 5635.1– 076 / 71 749		Création du document / <i>Document creation</i>
2				
3				
4				

SOMMAIRE / CONTENTS

FICHE RECAPITULATIVE / REVISION HISTORY	2
SOMMAIRE / CONTENTS	3
FIGURES LIST	4
1. Introduction	5
1.1. GSM 07.10 Multiplexer Protocol	5
1.2. Implemented Features.....	5
2. Applications Overview	5
2.1. Choice of implementation: Pseudo terminals	5
2.2. Instruction for use	6
3. Internal mechanisms description	8
3.1. Serial link management	8
3.1.1. Configuration: open_serialport	8
3.1.2. Hilo setup: initGeneric.....	9
3.2. Creation and configuration of the pseudo terminals	11
3.2.1. Pseudo terminals usage.....	11
3.2.2. Pseudo terminals configuration: open_pty.....	11
3.3. Data flow: The main loop	13
3.3.1. Interfaces	13
3.3.2. Exit condition	14
3.3.3. Flowchart	14
3.4. Handling 07.10 frames.....	16
3.4.1. 07.10 Frames exchange reminder	16
3.4.2. Handling incoming frames.....	18
3.4.3. Send frame	20
3.5. Virtual signals extension.....	21
3.5.1. Modem Status Command overview	21
3.5.2. Signal handling with pseudo terminals	22

FIGURES LIST

Figure 1: Functional Diagram.....	6
Figure 2: SerialSignals.....	7
Figure 3: Pseudo terminals layout	11
Figure 4: Dispatcher	15
Figure 5: Frame extraction	19
Figure 6: Messages exchanged between gsmMuxd and SerialSignals	23

1. Introduction

GSM 07.10 is a multiplexer protocol specified by ETSI. It operates between a MS and a TE and allows a number of simultaneous sessions over a normal serial asynchronous interface. The GSM 07.10 allows, for example, to send SMS from the TE while a data connection is in progress. The GSM 07.10 specification defines three operations modes: basic, advanced without error recovery and advanced with error recovery. The driver, based on the work of Tuukka Karvonen <http://developer.berlios.de/projects/gsmmux/>, has been enhanced and modified to suit the Sagem HiLo / HiLoNC modules which implement only a subset of the basic mode operations. That's why the driver includes only basic mode operations.

1.1. GSM 07.10 Multiplexer Protocol

A GSM 07.10 multiplexer implementation takes input data from multiple sources (e.g. virtual serial ports) and puts the data into frames before sending it over a physical serial link. It also reads data from the serial link, and extracts the user data from the frames then sends it to a specific receiver (e.g. a virtual serial port). This way GSM 07.10 protocol creates virtual channels that are called Data Link Connections (DLC). Frame headers include a DLC identifier that distinguishes the channel in question.

Following frame types are defined:

- Set Asynchronous Balanced Mode (SABM) command opens a DLC
- Disconnect (DISC) command closes a DLC
- Unnumbered Acknowledgment (UA) response acknowledges DISC or SABM frame
- Disconnected Mode (DM) response informs that the DLC is not opened
- Unnumbered Information with header check (UIH) transfers control commands on the control channel and data on other DLCs. FCS is calculated only from the frame headers.
- Unnumbered Information (UI) transfers control commands on the control channel and data on other DLCs. FCS is calculated from the whole frame.

There is only one DLC reserved for the multiplexer control information and it's called the control channel. Multiplexer session control information and modem signals can be transferred over the control channel in UI or UIH frames. Other DLCs are for transferring user data (e.g. AT commands and raw data).

1.2. Implemented Features

The driver does not support the features of the advanced GSM 07.10 modes and only a subset of the basic mode features is implemented. The driver understands all the frame types, but it can't send UI frames. So only UIH frames are used for data transfer. To provide an easy to understand code, command timeouts and retries are not implemented. The daemon sends in the beginning SABM frames to open all the DLCs needed. If a frame is lost, which is unusual, DLC remains closed and the daemon needs to be restarted to open the DLC.

The following control channel commands are implemented in the application:

- multiplexer close down
- test command
- modem status command
- non supported command response. The break signal octet in modem status command is not used.

2. Applications Overview

This part describes the GSM 07.10 multiplexer driver, the user interface to handle virtual modem signal and the instruction to build and use these applications.

2.1. Choice of implementation: Pseudo terminals

There are several ways to implement a GSM 07.10 multiplexer driver and the differences come from the virtual channel handling. With the GSM 07.10 mux, we create multiple logical channels that must be linked to virtual serial port in order to be connected to the existing applications (PPP stack, terminal, etc). With Linux system we can either create a kernel driver that includes GSM 07.10 mux protocol and provides regular serial port interfaces or a user-space daemon and pseudo

terminals as serial port interfaces. We prefer pseudo terminal implementation because, despite the lack of signals handling, it is easier to understand and more portable than kernel module.

Pseudo terminals have a master device, which the application sees and controls, and a slave device, which provides stdin, stderr, stdout for the server program that can run in user space (minicom, pppd, ...). The pseudo terminal driver in the kernel forwards the data between the master and slave devices. With pseudo terminals you can't control modem signals (RTS, DTR, etc) thus we have an external graphic application to interact with logical channel signals. We use Linux named pipe as communication mechanism between the two applications (cf.: Figure 1: Functional Diagram).

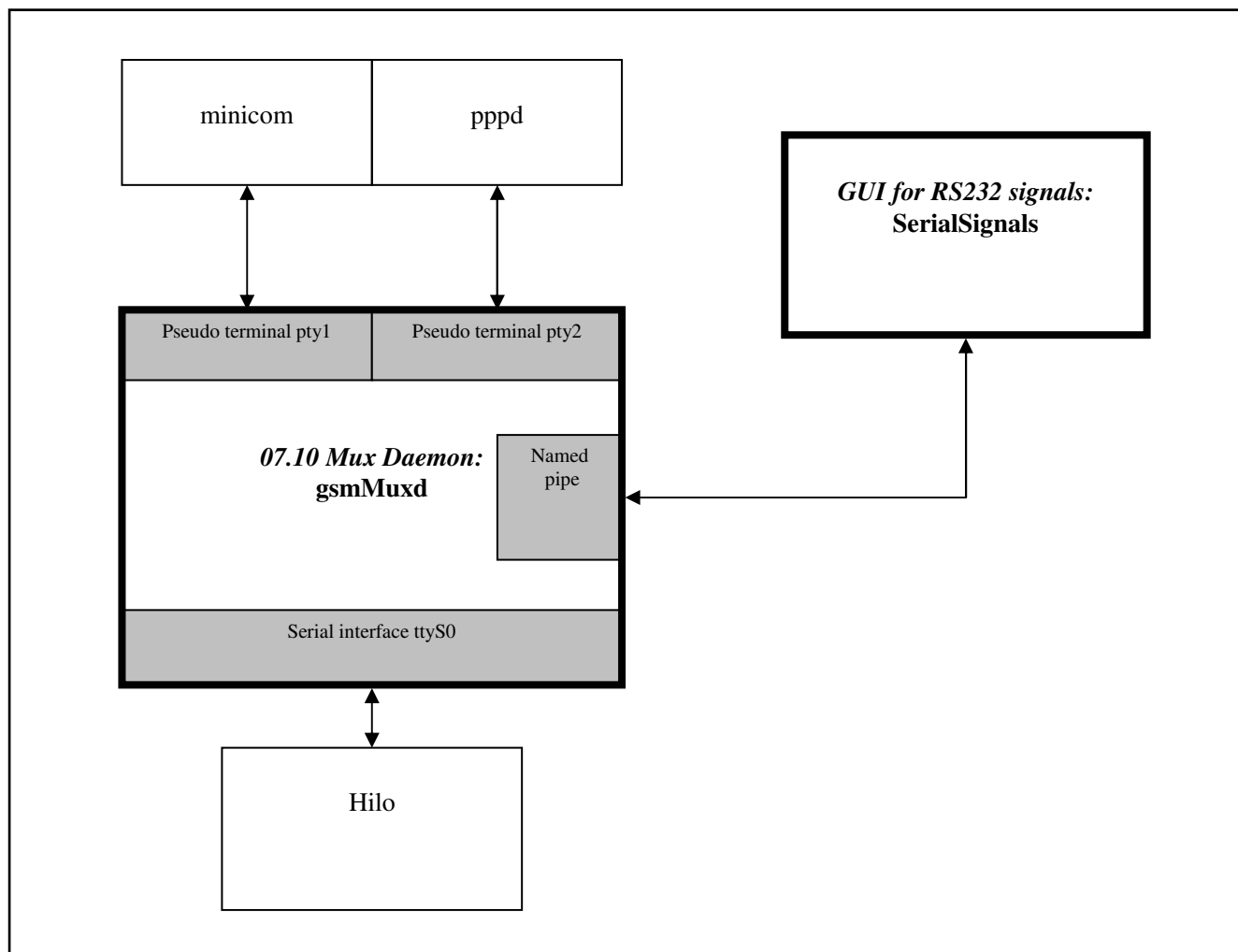


Figure 1: Functional Diagram

2.2. Instruction for use

First you need to extract and compile the driver:

- `tar -xvzf gsm0710.tar.gz`
- `cd gsmMuxd`
- `make`

The driver uses an external graphical application to monitor DLC modem signal. The SerialSignals application is a graphical application using QT 3.4.3 libraries. The installation of QT 3.4.3 is not covered by this document. Here are the instructions to compile this side application:

- `cd SerialSignals`
- `qmake SerialSignals.pro`
- `make`

Because of named pipe constraint, the GUI application must be started first. The application will wait for the mux driver to open the named pipe and the interface pops up (cf.: Figure 2: SerialSignals). This application relays modem status command for each DLC and the user can interact with virtual RTS, DTR and GSM 07.10 flow control (FC)

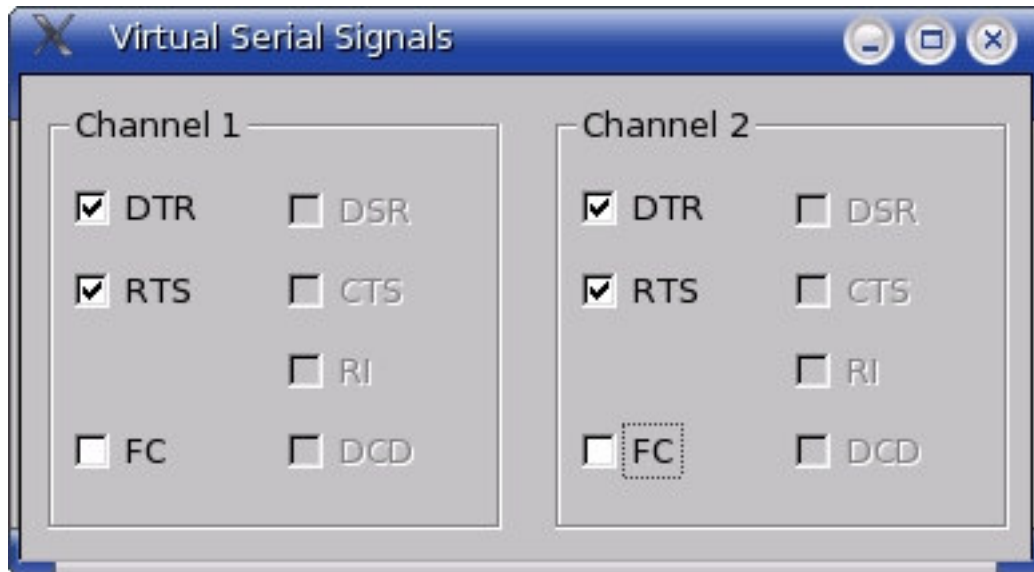


Figure 2: SerialSignals

Then you can start the daemon. First make sure that you have read/write permissions for the serial port where the Hilo is connected to. The driver command syntax is:

```
gsmMuxd [options] <pty1> <pty2> ...
```

<pty#> is pty devices (e.g. /dev/ptya0)

Options are:

- -p <serport> : Serial port device to connect to [/dev/modem]
- -f <framesize> : Maximum frame size [32]
- -d : Debug mode, don't fork
- -b <baudrate> : Module and MUX mode baud rate
- -P <PIN-code> : PIN code to fed to the modem
- -s <symlink-prefix> : Prefix for the symlinks of slave devices (e.g. /dev/mux)
- -w : Wait for daemon start-up success/failure
- -x : No signal management through pipes
- -h : Show this help message

So if you have connected your Sagem Hilo module to the first serial port, you could start the driver with command:

```
./gsmMuxd -p /dev/ttyS0 -w -f 64 -b 115200 -P <pin> -s /dev/HiloMux /dev/ptya1 /dev/ptya2
```

This will create two DLC that are accessed through pseudo terminals /dev/ptya1 and /dev/ptya2. The user side of these channels will be respectively /dev/HiloMux1 and /dev/HiloMux2.

Note that if your Linux distribution have only /dev/ptmx device for pseudo terminals, you have to repeat this device in the parameters. For example, the previous configuration with ptmx interface will be:

```
./gsmMuxd -p /dev/ttyS0 -w -f 64 -b 115200 -P <pin> -s /dev/HiloMux /dev/ptmx /dev/ptmx
```

NOTE: These applications have been developed on Linux kernel release 2.6.11

3. Internal mechanisms description

3.1. Serial link management

3.1.1. Configuration: open_serialport

The serial port configuration is similar to any other applications communicating with the Hilo. The port is set to the speed given in application's parameters. The serial protocol is set to 8bits data, no parity and no stop bit. We set the flow control but note that this is only relevant for physical link, flow control must be set separately for each channel.

```
int open_serialport(char *dev)
{
    int fd;

    if(_debug)
        syslog(LOG_DEBUG, "is in %s\n" , __FUNCTION__);
    fd = open(dev, O_RDWR | O_NOCTTY | O_NDELAY);
    if (fd != -1)
    {
        int index = indexOfBaud(baudrate);
        if(_debug)
            syslog(LOG_DEBUG, "serial opened\n" );

        struct termios options;
        // The old way. Let's not change baud settings
        fcntl(fd, F_SETFL, 0);

        // get the parameters
        tcgetattr(fd, &options);

        // Set the baud rates
        cfsetispeed(&options, baud_bits[index]);
        cfsetospeed(&options, baud_bits[index]);

        // Enable the receiver and set local mode...
        options.c_cflag |= (CLOCAL | CREAD);

        // No parity (8N1):
        options.c_cflag &= ~PARENB;
        options.c_cflag &= ~CSTOPB;
        options.c_cflag &= ~CSIZE;
        options.c_cflag |= CS8;

        // enable hardware flow control (CNEW_RTCCTS)
        options.c_cflag |= CRTSCTS;

        // set raw input
        options.c_lflag &= ~(ICANON | ECHO | ECHOE | ISIG);
        options.c_iflag &= ~(INLCR | ICRNL | IGNCR);

        // set raw output
        options.c_oflag &= ~OPOST;
        options.c_oflag &= ~OLCUC;
        options.c_oflag &= ~ONLRET;
        options.c_oflag &= ~ONOCR;
        options.c_oflag &= ~OCRNL;

        // Set the new options for the port...
        tcsetattr(fd, TCSANOW, &options);
    }
    return fd;
}
```


3.1.2. Hilo setup: initGeneric

The Hilo does not need specific setup to operate in 07.10 mux mode. This part is used to check whether the communication with the Hilo is correctly established. Once the Hilo responds to AT commands, we send initialisation commands like PIN code, flow control. This function can be customized according to specific applications but the last command sent must be the CMUX command.

Parameters of the CMUX command are hard coded, we use default parameters except for the baud rate and the maximum size of mux packets.

```
int initGeneric()
{
    char mux_command[20] = "AT+CMUX=0\r\n";
    int baud = indexOfBaud(baudrate);
    // Setup the speed explicitly, if given
    sprintf(mux_command, "AT+CMUX=0,0,%d,%d\r\n", baud, max_frame_size);

    /**
     * Modem Init
     */
    if (!at_command(serial_fd, "AT\r\n", 10000))
    {
        if (_debug)
            syslog(LOG_DEBUG, "ERROR AT %d\r\n", __LINE__);
        return -1;
    }

    if (pin_code >= 0 && pin_code < 10000)
    {
        char pin_command[20];
        sprintf(pin_command, "AT+CPIN=\"%04d\"\r\n", pin_code);
        if (!at_command(serial_fd, pin_command, 20000))
        {
            if (_debug)
                syslog(LOG_DEBUG, "ERROR AT+CPIN %d\r\n", __LINE__);
        }
    }

    if (!at_command(serial_fd, "AT&K3\r\n", 10000))
    {
        if (_debug)
            syslog(LOG_DEBUG, "ERROR AT&K3 %d\r\n", __LINE__);
        return -1;
    }

    /**
     * Start mux operation
     */
    if (!at_command(serial_fd, mux_command, 10000))
    {
        syslog(LOG_ERR, "MUX mode doesn't function.\n");
        return -1;
    }
    return 0;
}
```

As we will see in a later section, there is a main loop to gather data from the physical serial link and dispatch the extracted information to a channel. This loop deals only with 07.10 data format thus we have a specific function to send and receive AT commands before mux establishment: `at_command`.

We use the Linux `select()` function in a loop to read data incoming from the Hilo. This function asks for a file descriptor, in our case the serial port, and permits to schedule the program execution waiting for an I/O event. Another function is used to retrieve the Hilo answer to the AT command -OK or ERROR.

```

int findInBuf(char* buf, int len, char* needle)
{
    int i;
    int needleMatchedPos=0;

    if (needle[0] == '\\0') {
        return 1;
    }

    for (i=0;i<len;i++) {
        if (needle[needleMatchedPos] == buf[i]) {
            needleMatchedPos++;
            if (needle[needleMatchedPos] == '\\0') {
                // Entire needle was found
                return 1;
            }
        } else {
            needleMatchedPos=0;
        }
    }
    return 0;
}

int at_command(int fd, char *cmd, int to)
{
    fd_set rfd;
    struct timeval timeout;
    unsigned char buf[1024];
    int sel, len, i;
    int returnCode = 0;
    int wrote = 0;

    // Write command to serial port
    wrote = write(fd, cmd, strlen(cmd));
    // Wait until all buffer written
    tcdrain(fd);
    sleep(1);

    for (i = 0; i < 100; i++)
    {
        FD_ZERO(&rfd);
        FD_SET(fd, &rfd);

        timeout.tv_sec = 0;
        timeout.tv_usec = to;

        //Wait for incoming data
        if ((sel = select(fd + 1, &rfd, NULL, NULL, &timeout)) > 0)
        {
            if (FD_ISSET(fd, &rfd))
            {
                memset(buf, 0, sizeof(buf));
                len = read(fd, buf, sizeof(buf));

                if (findInBuf(buf, len, "OK"))
                {
                    returnCode = 1;
                    break;
                }
                if (findInBuf(buf, len, "ERROR"))
                    break;
            }
        }
    }
}

```

3.2. Creation and configuration of the pseudo terminals

3.2.1. Pseudo terminals usage

The pseudo terminals are composed of two interfaces, a master and a slave. The master will be interfaced in the application with the serial link to send and receive data. The slave will be available for user application (e.g. terminal, pppd). All data sent to the master will be automatically transferred to the slave and all data sent to the slave will be automatically transferred to the master. This second side constitutes the virtual channel (cf.: Figure 3: Pseudo terminals layout).

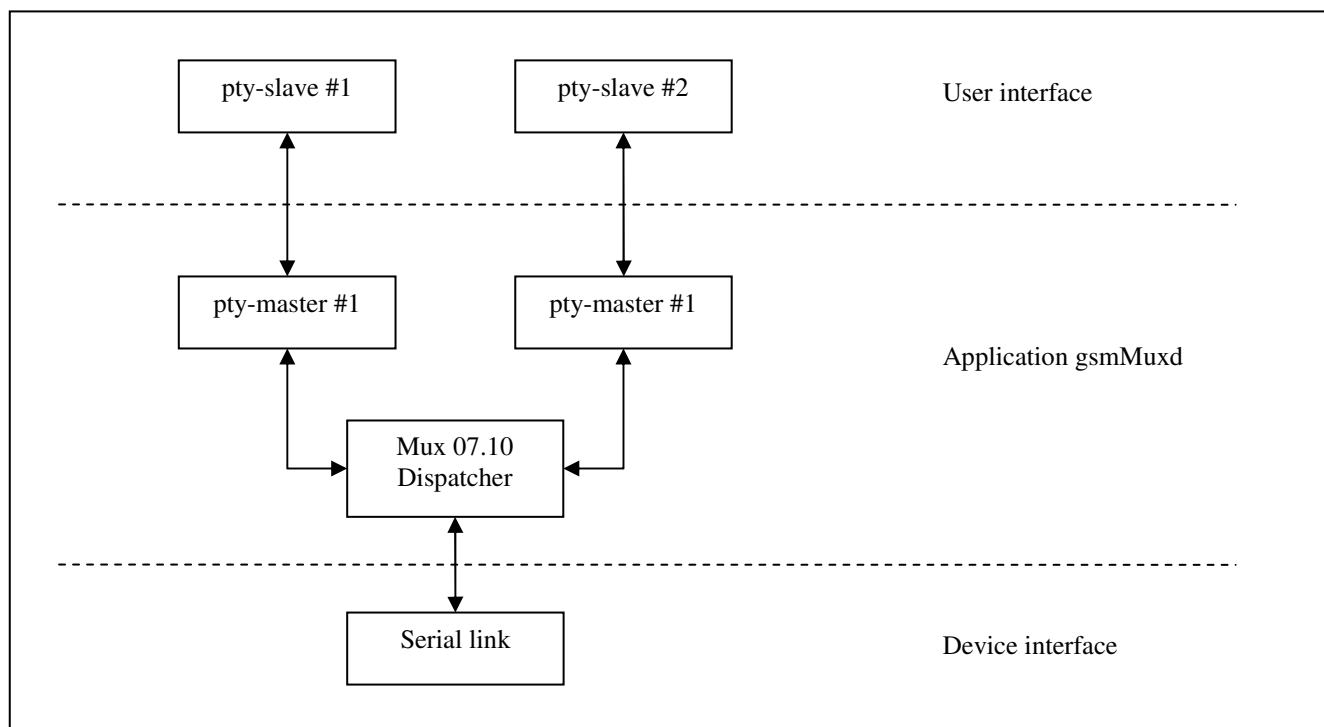


Figure 3: Pseudo terminals layout

3.2.2. Pseudo terminals configuration: open_pty

The number of virtual channel is get from the application parameters, once all the option have been parsed the remaining parameters are the pseudo terminal filename.

All the file descriptors corresponding to channels are stored in a global table: `dlci_fd[MAX_CHANNELS]`

The maximum number of virtual channel is set to 8 to match the Hilo specifications.

For each virtual channel we associate a structure to manage the status of the channel, either opened or closed and the v.24 virtual signals:

```

typedef struct Channel_Status {
    int opened;
    unsigned char v24_signals;
} Channel_Status;
  
```

The `v24_signal` variable is a bit field that contains the DTE side signals, as the channel is opened we assume that the DTR and CTS are active. The available signals are:

- `S_PIPE_FC` : Flow control - from 07.10 modem status command.
- `S_PIPE_DTR` : Ready to communicate - from 07.10 modem status command.
- `S_PIPE_RTS` : Ready to receive - from 07.10 modem status command.

```

for (t=optind;t<argc;t++)
{
    if((t-optind)>=MAX_CHANNELS) break;
    syslog(LOG_INFO, "Port %d : %s\n",t-optind,argv[t]);
    ptydev[t-optind]=argv[t];
}
numOfPorts = t-optind;

for (i = 0; i < numOfPorts; i++)
{
    remaining[i] = 0;
    if ((dlci_fd[i] = open_pty(ptydev[i], i)) < 0)
    {
        syslog(LOG_ERR, "Can't open %s. %s (%d).\n", ptydev[i], strerror(errno),
            errno);
        return -1;
    }
    else if (dlci_fd[i] > maxfd)
        maxfd = dlci_fd[i];
    cstatus[i].opened = 0;
    cstatus[i].v24_signals = S PIPE RTS | S PIPE DTR;
}

```

The pseudo terminal's configuration is similar to the physical serial port's configuration. Each pseudo terminal is configured as raw input and raw output.

```
int open_pty(char* devname, int idx) {
    struct termios options;
    int fd = open(devname, O_RDWR | O_NONBLOCK);
    char *symLinkName = createSymlinkName(idx);
    if (fd != -1) {
        if (symLinkName) {
            char* ptsSlaveName = ptsname(fd);

            // Create symbolic device name, e.g. /dev/mux0
            unlink(symLinkName);
            if (symlink(ptsSlaveName, symLinkName) != 0) {
                syslog(LOG_ERR, "Can't create symbolic link %s -> %s. %s (%d).\n",
                    symLinkName, ptsSlaveName, strerror(errno), errno);
            }
        }
        // get the parameters
        tcgetattr(fd, &options);
        // set raw input
        options.c_lflag &= ~(ICANON | ECHO | ECHOE | ISIG);
        options.c_iflag &= ~(INLCR | ICRNL | IGNCR);

        // set raw output
        options.c_oflag &= ~OPOST;
        options.c_oflag &= ~OLCUC;
        options.c_oflag &= ~ONLRET;
        options.c_oflag &= ~ONOCR;
        options.c_oflag &= ~OCRNL;
        tcsetattr(fd, TCSANOW, &options);

        if (strcmp(devname, "/dev/ptmx") == 0) {
            // Otherwise programs cannot access the pseudo terminals
            grantpt(fd);
            unlockpt(fd);
        }
    }
    free(symLinkName);
    return fd;
}
```

As the device name of the slave side of the pseudo terminal cannot be anticipated and is only known in the application, we provide a parameter to select a common prefix for all the created channels. The application will create a symbolic link between the slave side and a new device name.

3.3. Data flow: The main loop

3.3.1. Interfaces

The application deals with file descriptors to handle the physical serial port, the virtual channels and the pipe for virtual signals. We use Linux's select function to wake the application up each time data are present at one interface.

After initialization we have the following file descriptors:

- serial_fd : The physical serial port.
- dlc_i_fd : A table that gathers the virtual channels.
- pipe_fd : The pipe to handle virtual signals - This specific part will be discussed in another chapter.

These file descriptors are registered in a container and passed to select function to be monitored. If there is no data available the select function will automatically wake up the process after one second:

```
FD_ZERO(&rfd);
FD_SET(serial_fd, &rfd);
FD_SET(pipe_fd[1], &rfd);
for (i = 0; i < numOfPorts; i++)
    FD_SET(ussp_fd[i], &rfd);

timeout.tv_sec = 1;
timeout.tv_usec = 0;

sel = select(maxfd + 1, &rfd, NULL, NULL, &timeout);
```

3.3.2. Exit condition

This main loop is also used to control 07.10 mux deactivation. As the application is supposed to run in background mode, the end of multiplexed mode is triggered by Linux signals. The signal SIGPIPE will immediately close the application and the following signals can be used to deactivate the mux before closing the application: SIGINT, SIGTERM, SIGUSR1, SIGKILL.

There are two variables checked to exit the loop:

- **terminateCount** : Concerns the virtual channels. This global is set each time a channel is opened or closed. When the user requests a close of the mux mode, all the channels are correctly disconnected.
- **terminate** : At start up this global is used to validate the mux establishment. Inside the loop, this variable is set by the user with signal or by the application if the mux mode is closed by the Hilo.

3.3.3. Flowchart

The next figure (Figure 4: Dispatcher) describes how the incoming data are driven through the main loop. Note that this mechanism could have been implemented as separate automata with an environment that supports threads.

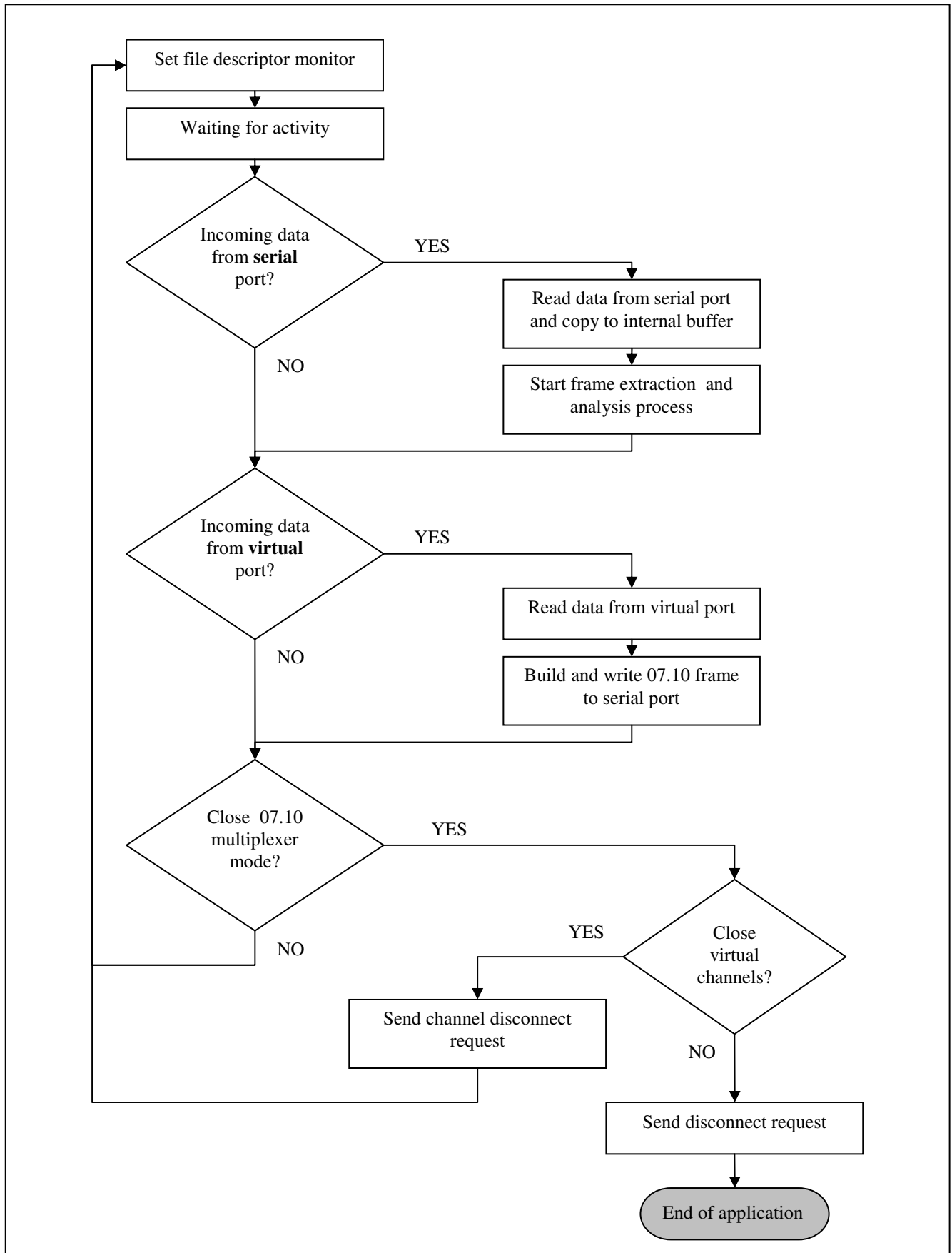


Figure 4: Dispatcher

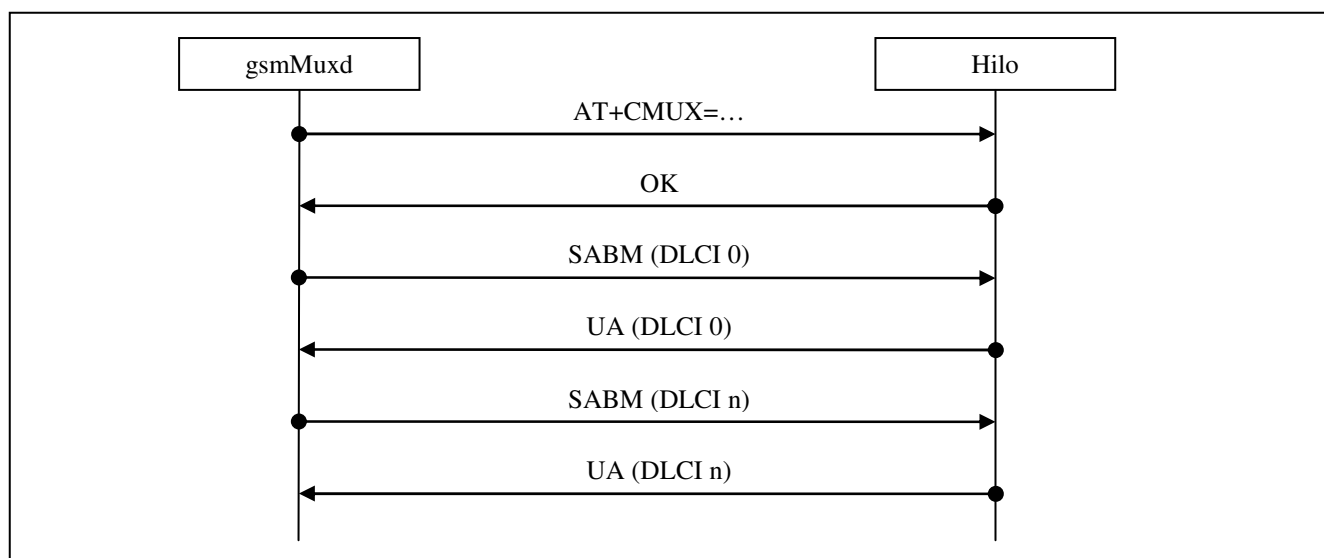
3.4. Handling 07.10 frames

3.4.1. 07.10 Frames exchange reminder

This section explains how frames are used to establish a multiplexer mode with two channels, data and command exchange and multiplexer deactivation.

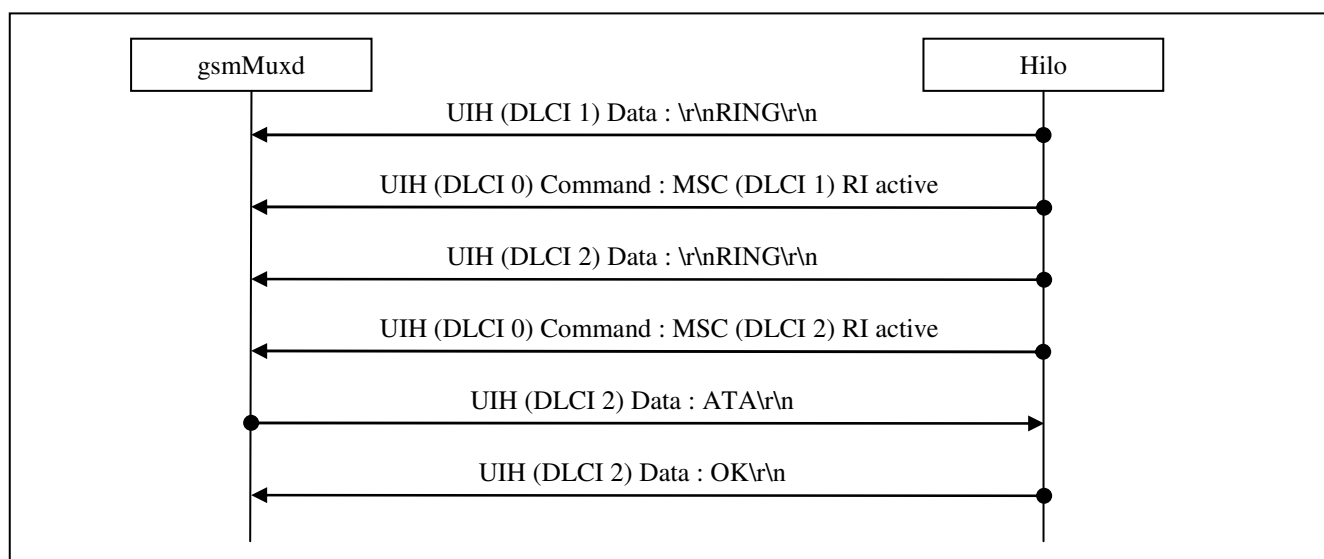
Channel activation

Channel activation is performed after the OK answer of AT command CMUX. Then the first channel to be opened is the control channel (DLCI 0). We request a channel by sending a SABM frame, the number of the channel corresponds to the address field. The Hilo will return a UA frame or DM frame with the same address field in case of, respectively, success or failure. This process is repeated for each data channel.



Data and command transport

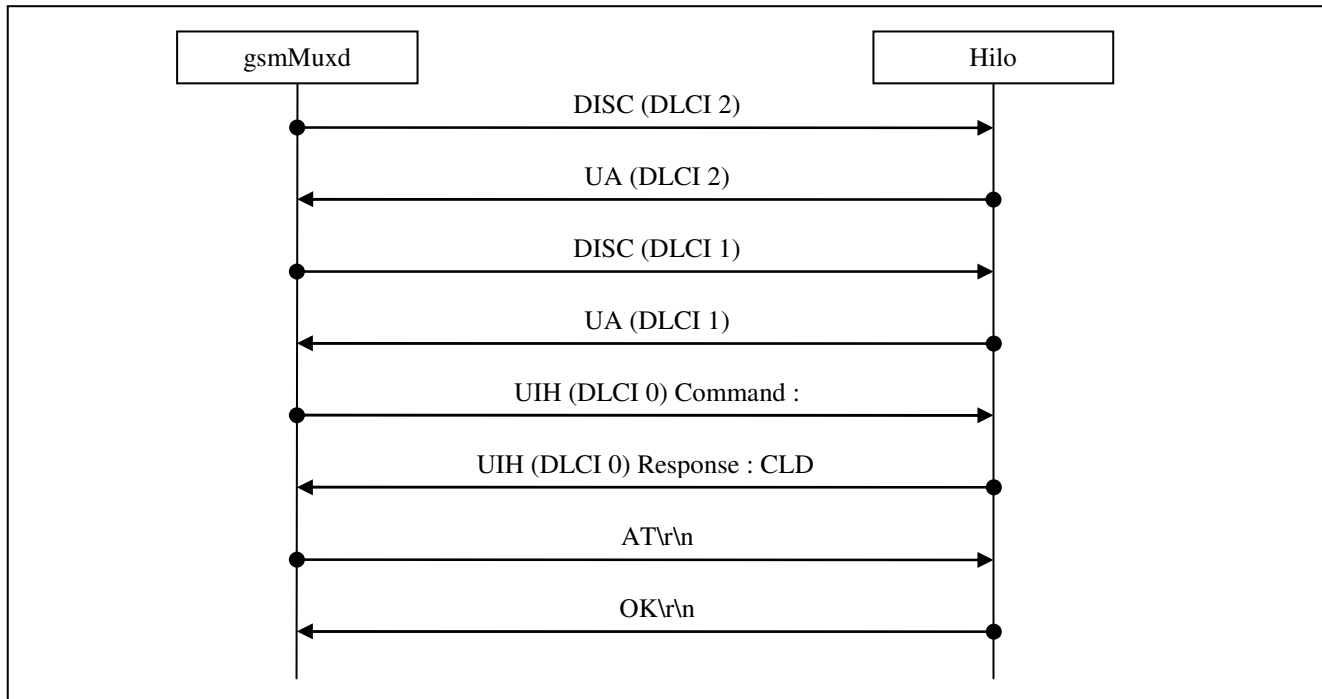
Data are transmitted through UI(H) frames, the address field corresponds to the selected virtual channels. The UI(H) frames addressed to the control channel are used to send commands either for the multiplexer management or for a specific channel, in this case the command structure contains a field to identify the channel. For the following example, two data channels are opened and a voice call is received. We receive the RING indication for both channels and the virtual RI v.24 signal is raised. We answer on channel 2.



Channel deactivation

The data channels are closed by sending a DISC frame to the corresponding channel number, the command is acknowledged with an UA frame.

For the control channel, there is two ways to close the channel, either with a DISC frame or by using the multiplexer close down operation (CLD) contained in an UI(H) frame.



Frame format

As a reminder the 07.10 frame has the following structure:

FLAG	ADDRESS	CONTROL	LENGTH	MESSAGE	FCS	FLAG
0xF9	1 octet	1 octet	1 or 2 octets	Defined by length field	1 octet	0xF9

The FCS calculation includes the ADDRESS, CONTROL and LENGTH fields but in the case of an UI frame, the calculation will also includes the MESSAGE field.

Command/response bit management

The address field contains the command/response (C/R) information but the operation conveyed by the UI(H) frames also contains a command/response information in the field type. We have the following behaviour for the address field's C/R management with SABM, UA, DM, DISC:

Command/response	Direction			C/R value
Command	DTE	→	DCE	1
	DCE	→	DTE	0
Response	DTE	→	DCE	0
	DCE	→	DTE	1

As the UI(H) frames can transport either operation for the control channel or data for virtual channels, the C/R management is simplified: The C/R value is based on the frame direction. In the case of operation, the type field includes also a C/R bit

to distinguish commands and response, the frame direction is ignored. The following table show this two C/R bit management:

Command/response	Direction			Address C/R	type C/R
Command	DTE	→	DCE	1	1
	DCE	→	DTE	0	1
Response	DTE	→	DCE	1	0
	DCE	→	DTE	0	0

Poll/Final bit management

This bit present in the control field is use to solicit a response frame from the other station at the earliest opportunity. In our 07.10 implementation, we send each frames with this bit set except for the UI(H) frames.

3.4.2. Handling incoming frames

Internal buffer: GSM0710 Buffer

The application uses a specific circular buffer to store incoming frames. This buffer is arbitrary sized to store 2 048 bytes.

```
typedef struct GSM0710_Buffer {
    unsigned char data[GSM0710_BUFFER_SIZE];
    unsigned char *readp;
    unsigned char *writep;
    unsigned char *endp;
    int flag_found; // set if last character read was flag
    unsigned long received_count;
    unsigned long dropped_count;
} GSM0710_Buffer;
```

This buffer is also used for statistic purposes: Amount of received frames and amount of dropped frames. Each data bloc received from the serial port is sent to this buffer with the function: `gsm0710_buffer_write`.

Frame extraction: GSM0710 Frame

The function to extract frames from the internal buffer is `gsm0710_buffer_get_frame`. This function detects the start/stop flags (0xF9) and stores the available data in a structure. This function will also calculate and verify the Frame Checking Sequence (FCS) in order to return only valid frames.

The fields extracted are:

- The channel address (Control channel or data channels).
- The type of frame received (SABM, UA, UIH, ...).
- The length of the 07.10 message.
- The message or data payload.

```
typedef struct GSM0710_Frame {
    unsigned char channel;
    unsigned char cr;
    unsigned char control;
    int data_length;
    unsigned char *data;
} GSM0710_Frame;
```

To detect a valid frame inside the buffer we parse each byte in a decoder. The frame extraction and the FCS calculation are performed simultaneously by analysing each byte from the first received FLAG to the end flag according to the length field. A frame will be discarded if the FCS calculations fails or if the end of frame flag is not present at its expected place. When a frame is discarded the function calls itself again to provide the next valid frame to the first caller. The figure 5 describes the flowchart of the frame extraction.

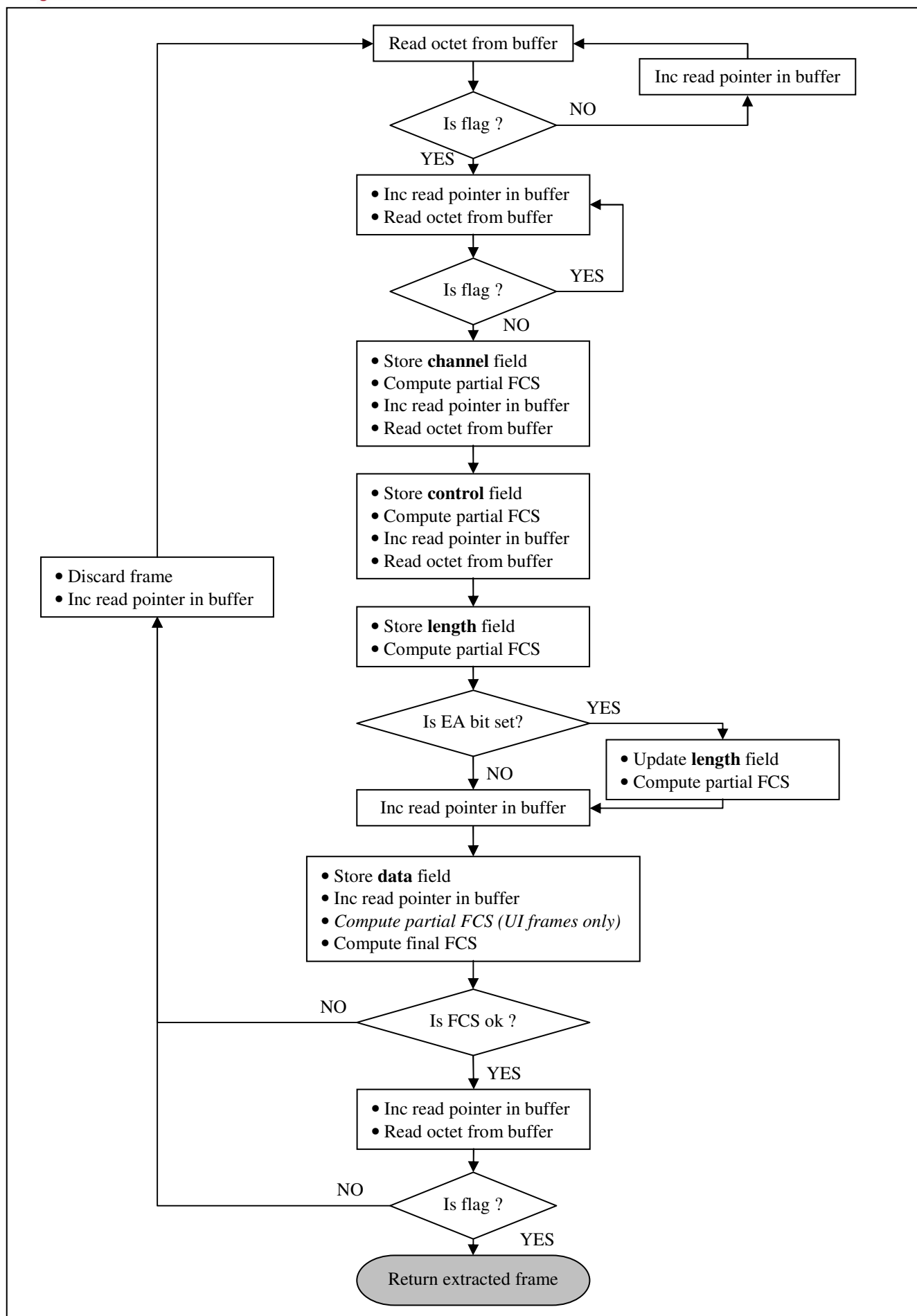


Figure 5: Frame extraction

The FCS calculation is performed as define in the 3GPP specification:

- CRC table is hard coded
- Initial value of FCS is: 0xFF
- Partial FCS is: $FCS = \text{crctable}[FCS \wedge \text{current_data_octet}]$
- Final FCS is: $FCS = \text{crctable}[FCS \wedge \text{received_FCS}]$
- FCS check is: $FCS == 0xCF$

Frame analysis: extract frames

The frame analysis is managed by the function `extract_frames`. As shown in the main loop flowchart, this function is trigged each time data are read from the serial port. The function implements a loop where all the available frames in the buffer are parsed.

The first step is to define whether the frame type is UI(H). Whether the UI(H) is addressed to the control channel or a data channel, the information contains respectively command control for multiplexer management or data information for virtual channels (AT commands or raw data). For data information, the information is directly sent to the corresponding virtual channel. For control channel, another parser is used to handle the command: `handle_command`.

Others Frames types than UI(H) deal with channels establishment:

- UA : This is a positive response to a channel creation (control channel and data channel) or a positive answer to a data channel disconnection. The status of each virtual channel is stored in the global structure `Channel_Status`, its field "open" is set once we receive an UA frame.
- DM : This is a negative response to a channel creation. In the case of the control channel the application will be terminated. Nothing needs to be returned.
- DISC : This is a request for a channel disconnection. We close the channel according to the address field and the application returns an UA frame to acknowledge the request. If the channel is already closed we return a DM frame.
- SABM : This is a request for a channel connection, we must try open the asked channel and return an UA or DM frame. Nonetheless this is for future purpose, the Hilo works as a responder in the channel creation.

Only a subset of the commands carried by the UI(H) frames are available in the Hilo and this simple application implements the most usual:

- CLD : This is the equivalent of the DISC command, the multiplexer mode is closed.
- MSC : This command indicates the modification of virtual v.24 signals for a given port. The states of the signals are transferred to the `SerialSignals` application through the dedicated pipe (cf.: 3.6 Virtual signals extension). When the MSC is detected we must acknowledge the reception by sending an MSC that contains the received v.24 signals.
- NSC : This message is use to inform the other side the multiplexer (DTE or DCE) that the previous receive command is not supported. There is no answer to this command.

An operation must have the C/R flag set to be treated, if the flag is not set this is a response to a previous command.

3.4.3. Send frame

Frame creation: write frames

The frames are sent by the function `write_frames`. This is function handle the 07.10 frame structure, the message field is considered as a buffer thus control frames (e.g.: UI(H) that contains MSC message) must be build before the call to this function. The function receives the following parameters:

- Channel address
- Command/Response bit
- Frame type with Poll/Final bit
- Message length
- Message

The function returns the length of the message actually sent. This length will be inferior or equal to the maximum frame size defined at multiplexer activation - Parameter N1 from the AT command CMUX.

The frame is sent in three parts:

- prefix: FLAG, ADDRESS, CONTROL, LENGTH fields. These fields are use for FCS calculation.
- message: MESSAGE filed. This message is truncated if needed.
- postfix: FCS, FLAG.

Address field is a combination of the channel address and the C/R bit, the Extended Address (E/A) bit is always set:

1	2	3	4	5	6	7	8
E/A	C/R	DLCI					

The control field is a combination of the frame type and the poll/final bit. The poll/final bit is set for all frames, except for UI(H) frames:

1	2	3	4	5	6	7	8
Frame type (LSB)				P/F	Frame type (MSB)		

The length field can be one or two octet length according to the length value:

1	2	3	4	5	6	7	8	1	2	3	4	5	6	7	8
E/A	Length < 128							Length ≥ 128							

The FCS is calculated over the prefix part, as soon as this table has been correctly set up. We use code provided by the 3GPP specification to calculate the FCS:

```
unsigned char make_fcs(const unsigned char *input, int count) {
    unsigned char fcs = 0xFF;
    int i;
    for (i = 0; i < count; i++) {
        fcs = r_crc4table[fcs^input[i]];
    }
    return (0xFF-fcs);
}
```

The three parts are sent through the serial link with Linux write function.

Incoming data from virtual ports

The data coming from virtual ports are read directly in the main loop and the length of these data can be greater than the maximum frame size previously defined. The write_frame function will build the frame according to the maximum allowed and returns the amount of data really transferred, we check this value in order to send the remaining data.

All the data coming from virtual ports are encapsulated in UI(H) frame with the poll/final bit unset.

3.5. Virtual signals extension

3.5.1. Modem Status Command overview

With basic implementation, the most useful operation on virtual channels is the Modem Status Command (MSC). This message is use by both side of the multiplexer (DTE and DCE) to notify each other of the status of their own V.24 control signals. The DTE will notify the Hilo about RTS and DTR signals for a channel and the Hilo will send a MSC that contains CTS, DSR, DCD and RI. This message is also used to convey break signals but this option is not implemented in this application.

The MSC also embeds a specific field to manage a flow control in the 07.10 protocol layer. This field, called Flow Control (FC), is set when one side of the multiplexer is enable to receive frames.

The MSC command has the following structure:

Command	Length	DLCI	V.24 signals
1 octet	1 octet	1 octet	1 octet

The command field octet has the following format:

1	2	3	4	5	6	7	8
E/A	C/R	0	0	0	1	1	1

The C/R bit will be set for a command and unset for a response. The E/A bit is always set.

The Length field is always set to one:

1	2	3	4	5	6	7	8
E/A	1	0	0	0	0	0	0

The DLCI field contains the channel number associated to the signals:

1	2	3	4	5	6	7	8
E/A	1	DLCI					

The V.24 signals contain the signals. These signal are common for both side of the multiplexer then the meaning of the signal must be interpreted according to the sender. For a MSC sent by the DTE, RTC and RTR are associated to DTR and RTS, other bits are ignored. For a MSC sent by the DCE, RTC and RTR are associated to DSR and CTS, IC and DV are associated to RI and DCD. The FC bit is independent from the sender and has always the same meaning:

	1	2	3	4	5	6	7	8
07.10 signals	E/A	FC	RTC	RTR	0	0	IC	DV
DTE	E/A	FC	DTR	RTS	0	0	0	1
DCE	E/A	FC	DSR	CTS	0	0	RI	DCD

3.5.2. Signal handling with pseudo terminals

As explain in the application overview, the limitation of pseudo terminals is the signal management. Because the V.24 signals are not handled by the terminal we manage the signals from another application. This application is especially use to display V.24 signals coming from the Hilo (RI and DCD) but the user can also modify PC side signals to demonstrate how signals are sent.

The multiplexer application use two named pipes to communicate with the external application:

- /tmp/msc_out : Send MSC receive from the Hilo (FC, CTS, DSR, RI and DCD)
- /tmp/msc_in : Relay MSC from the PC side to the Hilo (FC, RTS and DTR)

To simplify the MSC transfer, the two octets message transmitted in these pipes is similar to the fields DLCI and V.24 signals from MSC frames:

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
DLCI								FC	RTC	RTR	x	x	RI	DCD	x

The following code example shows how the pipes are handled. The function mkfifo will create the named pipe. The named pipe is considered as a file so it needs to be create only on time, that's why the EEXIST result (already exist) is not an error.

```
/* Pipe creation */
ret = mkfifo(PIPE_MSC_OUT, 0666);
if((ret == -1) && (errno != EEXIST))
{
    syslog(LOG_ALERT, "Can't create %s. %s (%d).\n", PIPE_MSC_OUT,
    strerror(errno), errno);
    return -1;
}
ret = mkfifo(PIPE_MSC_IN, 0666);
if((ret == -1) && (errno != EEXIST))
{
    syslog(LOG_ALERT, "Can't create %s. %s (%d).\n", PIPE_MSC_IN,
    strerror(errno), errno);
    return -1;
}
/* Open pipes */
if((pipe_fd[0] = open(PIPE_MSC_OUT, O_WRONLY | O_NONBLOCK)) < 0)
{
    syslog(LOG_ALERT, "Can't open %s. %s (%d).\n", PIPE_MSC_OUT,
    strerror(errno), errno);
    return -1;
}
if((pipe_fd[1] = open(PIPE_MSC_IN, O_RDONLY | O_NONBLOCK)) < 0)
{
    syslog(LOG_ALERT, "Can't open %s. %s (%d).\n", PIPE_MSC_IN,
    strerror(errno), errno);
}
```

The figure 6 illustrates the data exchange between the Hilo, the multiplexer application and the signal extension.

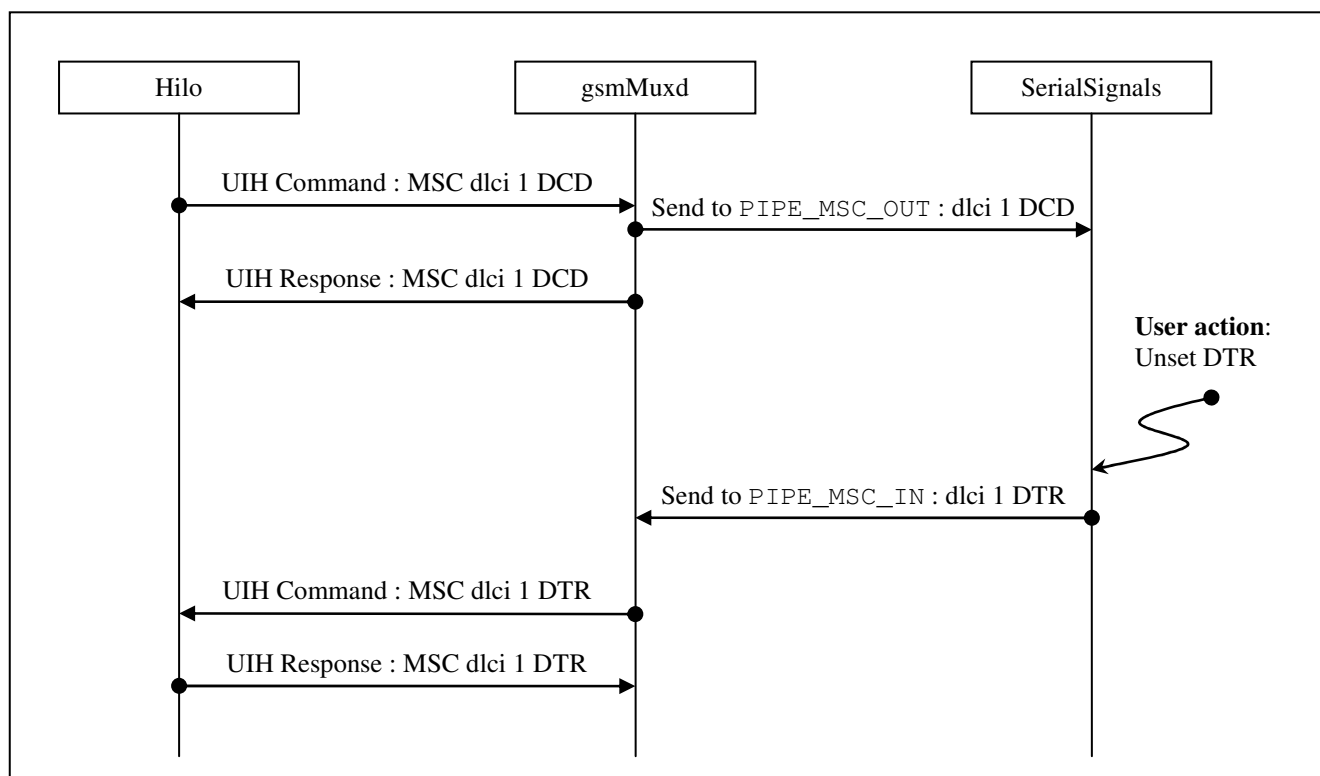


Figure 6: Messages exchanged between gsmMuxd and SerialSignals



Sagem Communications SAS
Energy & Telecom Business Unit
Headquarters : Le Ponant de Paris
27, rue Leblanc - 75015 Paris - FRANCE
Tel : +33 1 53 23 18 16 - Fax : +33 1 58 12 42 95
www.sagem-communications.com