OPTIMAY

OptiFlash and CmdFlash Utilities

User Manual

Version 3.0

OptiFlash and CmdFlash Utilities

User Manual

Document Summary

Document Type	User Manual
Document Title	*OptiFlash and CmdFlash Utilities*
Document Number	100-01-FMU
Version	3.0
Author	Martin Lohse
Date	10-Oct-2002

Change History

Rev.	**Date**	**Author**	**Change**
1.0	15-Oct-01	Lohse	Initial edition
2.0	08-Jan-02	Lohse	Added description of passive boatloader protocol; made some minor corrections
3.0	10-Oct-02	Lohse	Added new section on how to use the OpiFlash DLL in user applications; corrected various errors in the text.

Table of Contents

1 Introduction

Transferring code and data to or from a mobile phone requires a special program. Optimay provides two utilities for this purpose: the GUI-based "OptiFlash", intended for interactive use; and the command-line driven "CmdFlash", intended for use in batch files and Perl scripts. The utilities share a common functionality, and it may be assumed that–unless otherwise stated–anything said about OptiFlash applies equally to CmdFlash.

These tools are intended for use in a *development*, not a *production* environment. The requirements for these two different environments differ greatly. Refer to section **7**, "**Bootloader protocol**", for a description of the communication protocol implemented in the boot loader. This can be used to implement a flash loader suitable for production needs.

2 Overview

2.1 How it works

When the phone is switched on, the bootloader code in ROM checks whether OptiFlash is connected to the phone and is trying to "flash" it. If OptiFlash is *not* connected, the bootloader starts the protocol stack software.

Otherwise OptiFlash and the bootloader establish communication. To use the client-server terminology, the phone acts as a server, which runs in a loop and accepts commands from OptiFlash, the client.

Because space in ROM is scarce, and because ROM code/data is difficult to change, the bootloader is kept small and simple. It understands just sufficient commands to be able to load data into the phone's RAM. After establishing communication with the phone, the first thing OptiFlash does is to use the bootloader to download a so-called 'flash loader' to the phone's RAM. This is the program that provides all the necessary functions to erase and/or program the flash memory on the phone. Again, this code will act as a server, accepting commands from OptiFlash. Since there are many different flash chips from various manufacturers, all of which differ in their sector layout and capabilities, the flash loader does not contain much information about specific chips. It is merely capable of detecting the flash chip type and of programming the chip. All the details about the chip are handled on the PC side by OptiFlash, which is aware of the sector layout and other idiosyncrasies of the chips.

2.2 Bootloader types

There are two different types of bootloader communication protocol implemented in the ROM of Trident chipsets: active and passive. The names indicate whether the phone initiates the communication (active) or waits for OptiFlash to initiate the communication (passive).

The active bootloader initiates communication with the PC by sending a special byte-sequence to the serial port when the phone is switched on. If OptiFlash is connected and wants to "flash" the phone, it recognises this sequence.

The passive bootloader merely listens to the serial ports for a short period of time. If it detects a special character, it assumes that OptiFlash is connected and wants to “flash” the phone. The advantage of the passive approach is that other data applications are not confused by receiving unexpected characters from the serial port.

To determine which type of bootloader is in a particular phone, refer to section **5.2**, "**What type of bootloader?**". The passive bootloader is built into all new chipsets; the active bootloader is currently being phased out.

2.3 Basic operations

OptiFlash supports three modes of operation: flash, verify and read.

2.3.1 Flash mode

In this mode the contents of an S-record file (SRE for short) are loaded into the flash memory of the phone. It is used to apply software updates to the phone.

2.3.2 Verify mode

In this mode the contents of an SRE file are compared with the contents of the appropriate memory ranges in flash memory of the phone. This is a convenient and fast way to check whether the phone contains the correct software or data.

2.3.3 Read mode

This mode is used to read memory ranges from the flash memory on the phone into an SRE file on the PC. This is useful for backing up important data which is phone- or user-specific (e.g. calibration data, phonebook data).

2.4 A note on S-record files

OptiFlash expects that all files it handles will be in Motorola S-record format . All files written by OptiFlash are in this format.

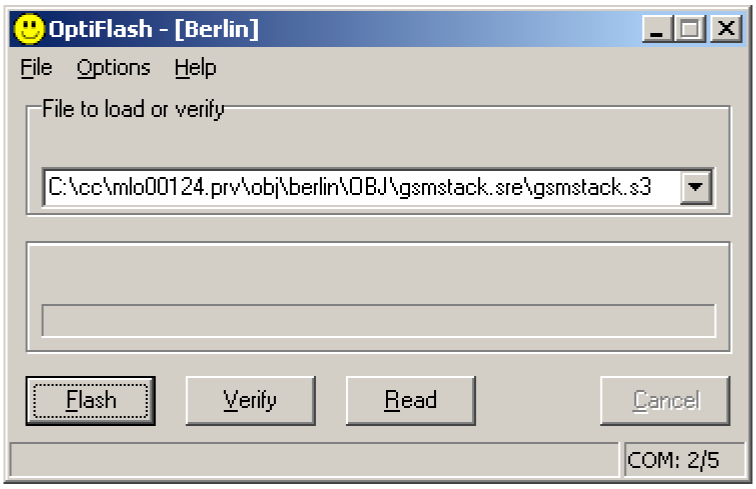
A SRE file is an ASCII file containing addresses and data in hexadecimal form, guarded by a checksum byte at the end of each row. Conventionally, the filename extension is ‘.sre’. The user does not need any deeper knowledge about SRE files, but it is useful to understand some basics about this format.

Since the SRE contains not only data but also the addresses where this data should be written to or read from, the data in the file constitutes multiple regions in consecutive memory. When loading a SRE file OptiFlash will check that all memory addresses are in ascending order and that the memory regions are not overlapping (which would lead to overwriting of content).

If both software and data need to be downloaded to the phone, the two SRE files can simply be concatenated in the correct order and the resulting single SRE file flashed into the phone.

3 Using OptiFlash

This is the main window of OptiFlash:



C:\cc\mlo00124.prv\obj\berlin\OBJ\gsmstack.sre\gsmstack.s3

COM: 2/5

The file name shown is that of the file that will be loaded/verified. This can be changed via Options/Settings (see below) or via "Drag & Drop". The list box contains a history of the most recently used files. The history is maintained on a per-profile basis (see section **3.3**, "**Profiles**"); i.e. every profile has its own history list that is independent of the other profile history lists.

There are four buttons, three for the basic operations OptiFlash supports, and a fourth one to cancel the operation currently in progress (greyed out in this picture).

Once you have made all the necessary settings for your intended operation, you merely have to click the appropriate button. OptiFlash now initializes the operation. This may take a few seconds, depending on the chosen operation and on the size of the involved SRE files. When OptiFlash is ready, it prompts you to switch on the phone by displaying the message “Power on mobile”. Do not power on the phone before that prompt is displayed, otherwise the communication setup will fail.

The currently used COM port(s) are shown in the lower right corner of the window.

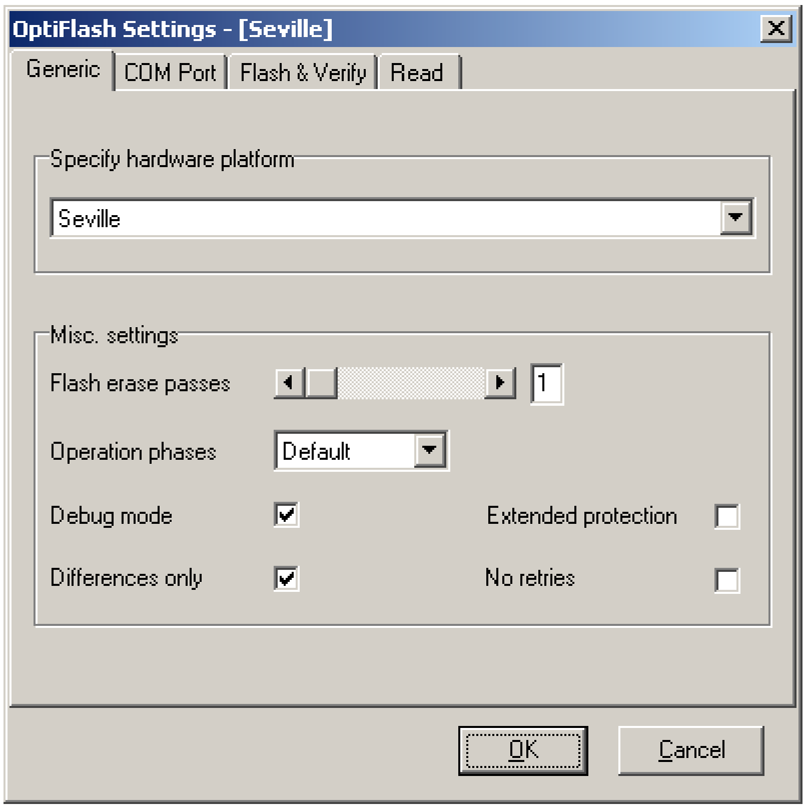
3.1 The settings window

The most important menu item is the settings window, which can be reached via Options/Settings from the main window.

The settings window consists of four different tab sheets, which are explained below.

3.1.1 Settings/General

On this tab sheet all the general settings can be selected.



Specify hardware platform: Seville

Misc. settings

Flash erase passes: 1

Operation phases: Default

Debug mode: ✓

Extended protection: ☐

Differences only: ✓

No retries: ☐

Specify hardware platform

Here you specify the type of hardware you want to flash/verify/read. This is the most important setting, as this describes various details of the hardware to OptiFlash. If you select an incorrect platform here, it is very likely that OptiFlash will not be able to communicate with the platform.

The parameters needed to flash the MS are stored in a *platform definition file* (***platform.def***) which is supplied by Optimay.

Flash erase passes

This setting specifies how many times a flash sector should be erased (1..8). It is supplied for debugging purposes–the number of passes should be set to 1 for normal operation. A higher number does no harm (apart from reducing the life of the flash chip), but consumes considerably more time.

Operation phases

This option is for special development purposes. It must be set to *Default* for normal operation. The other two settings determine which phases of the loading process will be executed. When *Stop after BL* is selected, OptiFlash will load the currently selected flash loader into the phone and will then exit. When *Start at FL* is selected, OptiFlash will assume that a flash loader is already loaded into the phone and will start sending commands to it.

Debug mode

Enabling debug mode causes additional information to be written to the log window. If you are experiencing problems while flashing a phone and want assistance from Optimay, it is absolutely essential to have this option enabled and to send the resulting log file to Optimay.

Differences only

This option allows for more efficient data transfer. When enabled, OptiFlash checks every sector that would be affected by the currently selected SRE file. If the sector has the same contents as those specified by the SRE file it is not changed, thus saving the time for required for programming the flash memory .

Extended protection

Enabling this option reduces the probability of a break in the serial-link communication causing corrupted data in a flash sector. When new data has to be added to a sector which already contains data, then the old data must be merged with the new data before the sector is erased and rewritten. If extended protection is disabled the sequence of events is:

1) Make a copy of the flash sector in RAM.
2) Erase the sector.
3) Download the new data and merge into the RAM copy.
4) Write the RAM copy back to the sector.

If extended protection is enabled, the sequence is changed to:

1) Make a copy of the flash sector in RAM.
2) Download the new data and merge into the RAM copy.
3) Erase the sector.
4) Write the RAM copy back to the sector.

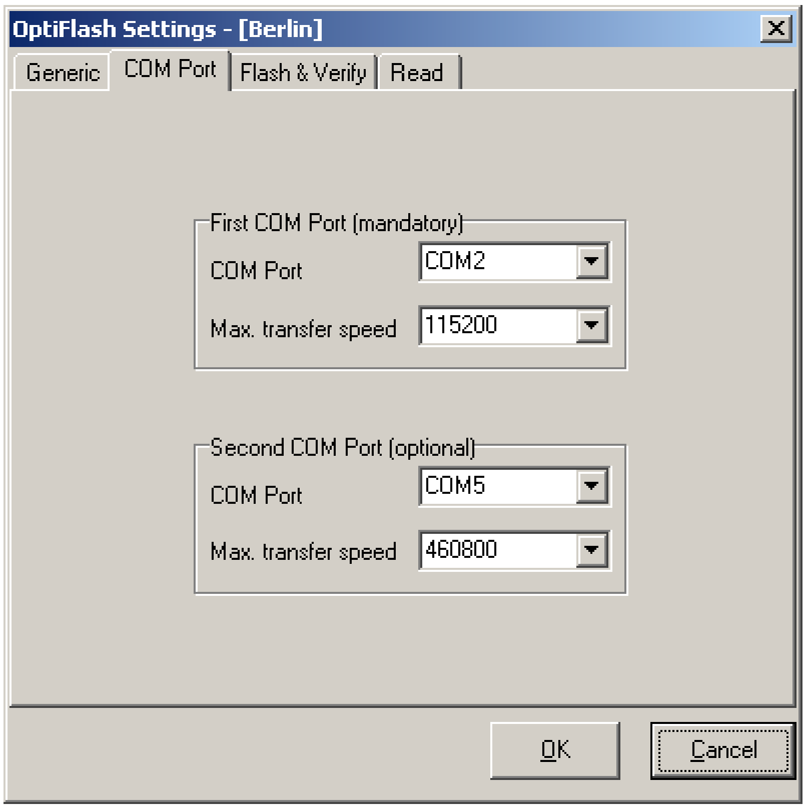
Since in this case the flash loader has all the required data before it starts to make changes to the flash, the danger of corruption is much less.

No retries

Normally, when a command times out before being executed, it is retried several times. When this option is enabled, no retries are attempted and the operation is terminated immediately with an error.

3.1.2 Settings/COM Port

This tab sheet is used to select the COM ports.


OptiFlash Settings - [Berlin]

Generic | COM Port | Flash & Verify | Read

First COM Port (mandatory)
COM Port: COM2
Max. transfer speed: 115200

Second COM Port (optional)
COM Port: COM5
Max. transfer speed: 460800

Up to twelve ports are supported. Additionally you can select the maximum transfer speed OptiFlash will use to communicate with the phone. However, OptiFlash will use a slower speed if either the PC's or the phone's serial hardware is incapable of handling the selected speed.

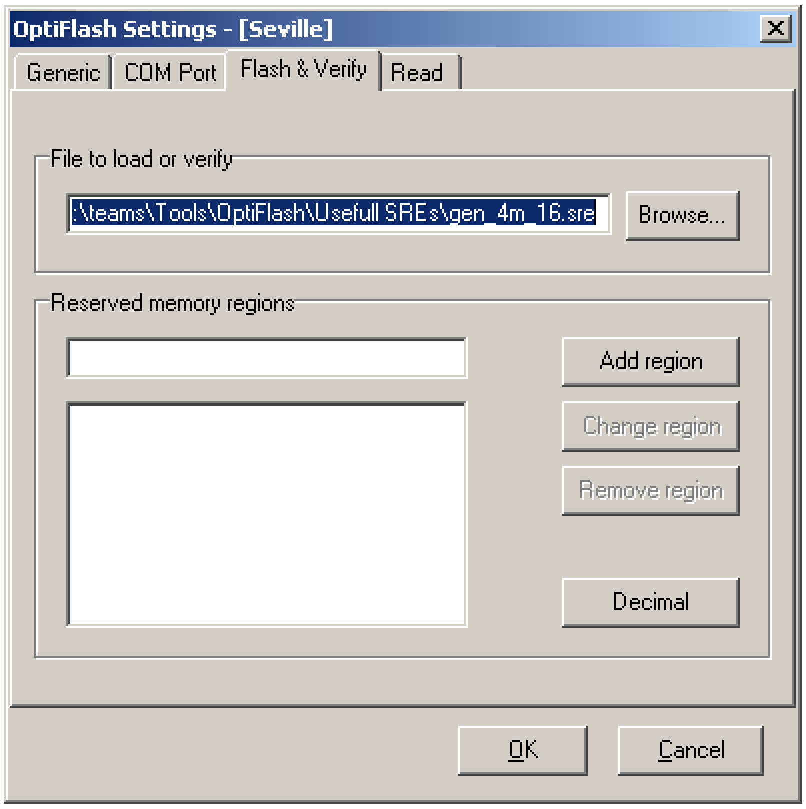
You can set up to two COM ports. The first COM port is mandatory and is used to transfer the flash loader to the phone. If a second COM port has been specified (optional), then OptiFlash will use this COM port as well as the first one to communicate with the flash loader, thus increasing the transfer speed. If the second COM port is not specified, all communication is done using the first COM port.

The reason for having two different COM ports results from idiosyncrasies of USB/serial adapters. They impose some latency in the serial communication that will exceed timeout values in active boot loaders. The trick is to transfer the flash loader into a phone with an active boot loader over the first COM port using a normal serial cable. Then to benefit from the higher transfer speeds of the USB/serial converter, the rest of the communication is done over the second COM port. This scenario is only necessary for phones with an active boot loader; phones with a passive boot loader should be able to cope with the communication latencies.

The speed increases you will experience when you use two COM ports in parallel is highly dependant on the PC used, the current workload on the system, the quality of the serial kernel drivers, etc.

3.1.3 Settings/Flash & Verify

This tab sheet is used to specify the SRE image filename and define protected regions of memory for flash and verify operations.



OptiFlash Settings - [Seville]

File to load or verify: :\teams\Tools\OptiFlash\Usefull SREs\gen_4m_16.sre

File to load or verify

This is the complete filename of the SRE image you want to use for flashing or for verifying.

Reserved memory regions

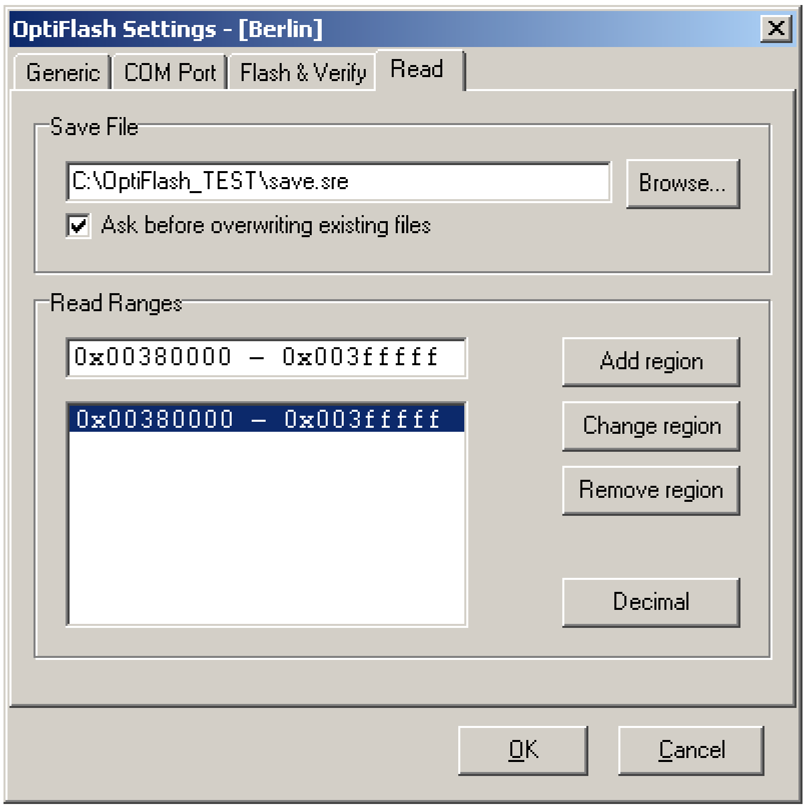
Here you can specify one or more *reserved memory regions*, i.e. memory ranges which OptiFlash must not change. OptiFlash checks whether the specified SRE file contains any memory ranges which overlap any of the reserved regions. If so, the operation is terminated with an appropriate error code.

Reserved memory ranges can be used to protect valuable data in the phone from accidental overwriting.

Memory ranges in OptiFlash can be specified in decimal or hexadecimal notation (prefixed with 0x). The lower and upper ends are separated with a dash '-'. The ends of the range are inclusive, e.g. the range 5-10 contains the 6 addresses 5, 6, 7, 8, 9 and10.

3.1.4 Settings/Read

This tab sheet is used to specify the SRE filename and define the required regions of memory for a read operation.


OptiFlash Settings - [Berlin]

Save File: C:\OptiFlash_TEST\save.sre

[✓] Ask before overwriting existing files

Read Ranges: 0x00380000 – 0x003fffff

0x00380000 – 0x003fffff

Save File
: This is the complete filename of the SRE file into which the data which has been read from the phone will be saved. You can select whether OptiFlash will ask for confirmation before overwriting existing files.

Read Ranges
: Here you specify the memory range(s) which are to be read out of the mobile. For further information about memory ranges as they are used by OptiFlash, refer to the description of **Reserved memory regions** in section **3.1.3**, "**Settings/Flash & Verify**".

3.2 The log window

A log window can be opened/closed via Options/Log. All output OptiFlash is generating goes into the log window. It is useful to find the exact cause of an error, especially when used in conjunction with the debug option enabled.

3.3 Profiles

The user can define multiple profiles. All the settings (except the global ones) are saved into a profile. This makes it easy to maintain collections of settings which are appropriate for various operations. For example, you can have profiles for:

- saving the calibration data of the phone
- flashing in the latest software version
- exchanging user data.

The file history on the main window is maintained separately for every profile, i.e. two different profiles have two different histories.

3.4 Errors

If OptiFlash does not complete an operation successfully, one the following error conditions will be shown:

ERR_ABORT
The user has aborted the operation.

ERR_BAD_CMDLINE
The given command line could not be successfully parsed because of syntax errors.

ERR_BAD_PARAMS
The parameters passed to the DLL are invaid.

ERR_BAD_PLATFORM
The specified hardware platform does not exist or the platform definition file (***platform.def***) is corrupted.

ERR_BAD_SREC_ADDR
The SRE file contains an invalid address. Either the addresses are not in ascending order, or the address lies outside the flash chip.

ERR_BAD_SREC_CHECKSUM
There was a checksum error for a line in an SRE file (i.e. the SRE file has been corrupted).

ERR_BAD_SREC_FILE
A file is not a valid SRE file.

ERR_CMD_FAILED
A command sent to the phone failed.

ERR_COM
The specified COM port could not be opened. It may be in use by another program.

ERR_CRC
All transmissions over the serial link are guarded by CRC checks. There was a CRC mismatch.

ERR_FILEIO
General problem with reading/writing files.

ERR_FLASHTYPE
Either the type of the flash chip in the phone is unknown to OptiFlash, or the chip is defective.

ERR_LEGACY_LOADER
The specified flash loader is too old to be used with this version of OptiFlash.

ERR_NOMEM
Insufficient memory to complete the operation.

ERR_PROTOCOL
There was a problem with the protocol on the serial link between the PC and the phone. Either OptiFlash received something unexpected, or the link is defective.

ERR_RANGES_OVERLAP
Two memory ranges overlap.

ERR_SETUP
An error occurred during the setup-phase of the flash loader.

ERR_TIMEOUT
The operation has exceeded its timeout period and has been aborted.

ERR_TOO_MANY_RANGES
Too many memory regions have been defined.

ERR_TOO_MANY_RETRIES
An operation has repeatedly failed and the maximum retry count has been exceeded.

ERR_UNKNOWN_HW
OptiFlash could not identify the hardware platform.

ERR_UNSUPPORTED
The executable has attempted an operation which is not supported by the DLL. (Cf. ERR_WRONG_DLL below.)

ERR_VERIFY_DIFFERENCE
The verify operation found a difference between the given SRE file and the contents of the flash memory in the phone.

ERR_WRONG_DLL
Both OptiFlash and CmdFlash consist of an executable (the GUI and the command-line interpreter respectively), and a DLL. The version numbers of the executable and the DLL do not match.

ERR_WRONG_FLASHLOADER
The specified flash loader is not suitable for this type of phone.

ERR_WRONG_LOADERFILE
The SRE file containing the flash loader is invalid in some way.

ERR_WRONG_LOADFILE
The SRE file containing the image to be loaded is invalid in some way.

ERR_WRONG_READLENGTH
An invalid memory range has been given for a read operation.

ERR_WRONG_SAVEFILE
An invalid filename was specified for the SRE file into which data read from the phone is to be saved.

4 Using CmdFlash

The command line utility “CmdFlash.exe” is intended for use in batch files and Perl scripts. It supports the same set of operations and options that OptiFlash does.

A few notes about the command-line options:

- The order and the case of command line options is not important.
- If the value of an option (i.e. the right side of the ‘=’) contains blanks, the value has to be enclosed in double quotations marks.
- Unless otherwise stated below, if an option is specified more than once, then it is the last occurrence which is used.

Here is a short description of the available options; default values are in **bold** type. References such as ***Debug mode*** are to the more detailed descriptions given in section **3**, “**Using OptiFlash**”.

`/com=<1..12>[,<1..12>]` ***Settings/COM Port***
Specifies the number(s) of the COM port(s) to be used.

`/maxspeed=<#>[,<#>]` ***Settings/COM Port***
Specifies the maximum transfer speeds (in bytes/second) for the COM port(s). Default is 115200 bps.

`/debug=<`**`0`**`/1>` ***Debug mode***
Disables (`/debug=0`) or enables (`/debug=1`) verbose output

`/diff=<0/`**`1`**`>` ***Differences only***
Disables (`/diff=0`) or enables (`/diff=1`) the differential flashing mode.

`/erase=<1..8>` ***Flash erase passes***
Specifies the number of erase passes for a flash sector.

`/extprotect=<`**`0`**`/1>` ***Extended protection***
Enables (`/extprotect=1`) or disables (`/extprotect=0`) the extended protection option.

`/file=<filename>` ***File to load or verify***
For `/mode=flash`
Specifies the the full filename of the SRE file containing the image to be flashed into the phone.
For `/mode=verify`
Specifies the full filename of the SRE file with which the phone’s flash memory contents are to be compared.

`/mode=<`**`flash`**`/verify/read>`
Specifies the required operation.

`/noretry=<`**`0`**`/1>` ***No retries***
Enables (`/noretry=0`) or disables (`/noretries=1`) retries for a failed operation.

`/phases=<`**`all`**`/bl/fl>` ***Operation phases***
Specifies which phases of the flashing process are to be performed.

`bl`	download the flash loader (“Stop after BL” in OptiFlash)
`fl`	communicate with the (already downloaded) flash loader (“Start at FL” in OptiFlash)
`all`	both of the above (“Default” in OptiFlash)

`/platform=`<platform name> ***Specify hardware platform***
Specifies the type of hardware connected to OptiFlash.

`/read=`<#-#> ***Read Ranges***
Specifies a memory range to be read from the phone. Use multiple `/read` options to specify multiple read regions.

`/readfile=`<filename> ***Save File***
For /mode=read:
Specifies the full filename of the SRE file into which the contents of the flash memory are to be written.

`/reserved=`<#-#> ***Reserved memory regions***
Specifies a reserved memory range. Use multiple `/reserved` options to specify multiple reserved regions.

5 Tips and tricks

5.1 Using two ports in parallel

For performance reasons it is possible to use two COM ports simultaneously. OptiFlash will use the first COM port to talk to the phone's boot loader. For phones with an active bootloader it is important, that this COM port is connected to the phone using a normal serial cable. For passive bootloaders, this can be either a serial cabel or a USB/serial converter.

When the communication phase with the bootloader is complete, OptiFlash will switch to the fastest port for communication with the flash loader. When OptiFlash has to do bulk data transfers to the phone, it will use all defined ports simultaneously. It is not necessary that all the used ports be operated at the same baud rate. OptiFlash will distribute the transferred data across the individual ports according to the used transfer speeds.

Using two ports in parallel is dependent on many PC-related factors, such as:

- system load
- quality of kernel drivers for the used ports
- mixture of different drivers or multiple instances of a single driver

As a result, the benefit in flashing speed is highly PC-dependent; but it should never be slower than using a single port.

5.2 What type of bootloader?

To find out which type of bootloader is contained in the phone's ROM, you can do the following:

1. Connect a dumb terminal to the phone (115 KBaud, 8N1) and switch on the phone.
2. If the first two characters you see on the terminal are 'OK', the phone has an active bootloader.
3. If you don't see 'OK', connect to the other UART and retry from step 1. If you still don't see 'OK', continue with step 4.
4. Switch off the phone and, keeping the question mark key ('?') pressed, switch it on again.
5. If you now see some exclamation marks ('!'), the phone has a passive bootloader.
6. If you *don't* see the exclamation marks, something is wrong with either the phone, the OptiFlash/phone setup, or the cables.

5.3 Troubles flashing the phone?

If you have troubles flashing at all on a slow PC (laptops in particular are sometimes tricky), try the following:

- Use a different serial port. Laptops in particular sometime have UARTs that don't supply enough power to operate the level shifter properly. You could try a CardBus serial port.
- If you are using an USB/serial converter, try to flash the phone using a normal serial cable and see if that works.
- Check if the Windows FIFO settings have been changed. The recommended setting is 'default'.
- Check that the voltage to power the phone is about 4.2V.
- Disable the 'debug' option
- Close the log window
- Use the command line tool "CmdFlash" instead of "OptiFlash"

5.4 Generating command line options

Whenever you click one of the flash/verify/read buttons on OptiFlash's main window, the appropriate command line options representing your current settings will be written to the ***OptiFlash.ini*** file under section [Global] as value "LastCmdLine". It can then simply be copy/pasted into a perl script, etc.

5.5 Reporting problems

Should you experience any problems with OptiFlash, please do the following:

1. enable debug mode
2. open the log window
3. repeat the failing operation
4. save the log to a file
5. send the log file to Optimay.

Without this information, it is nearly impossible to determine the reason for any problems.

6 Using OptiFlash in your own application

All the functionality of OptiFlash is contained in a DLL named ***flashdll.dll*** (called OptiFlash DLL in the following). The two programs OptiFlash and CmdFlash are just different frontends to make the DLL's functionality available to the user. You can use this DLL in your own applications in order to flash phones. All necessary declarations and definitions can be found in the file ***OptiFlashDLL.h***.

6.1 The functions

The OptiFlash DLL API consists of six functions. They all return **FLASH_SUCCESS** in case of success and an appropriate error code otherwise.

6.1.1 Main functions

The three main functions are **FlashInit**, **FlashLoad** and **FlashFinish**.

unsigned int FlashInit (FlashIO *pIO, FlashParameters *pParams);

This function has to be called first in order to be able to use the OptiFlash DLL.

pIO	points to a structure of type **FlashIO**. A handle will be placed in the structure which is required by several other DLL functions.
pParams	is either **NULL** or points to a structure of type **FlashParameters**. If supplied, the structure will be initialised with default values.

---

unsigned int FlashLoad (FlashIO *pIO, FlashParameters const *pParams);

This function performs a specified flash operation.

pIO	points to the initialised **FlashIO** structure.
pParams	points to a **FlashParameters** structure containing a description of the required operation.

---

unsigned int FlashFinish (FlashIO *pIO);

This function must be called after all flash operations have been performed. It releases the internally allocated resources.

pIO	points to the initialised **FlashIO** structure.

6.1.2 Helper functions

The are three helper functions that are convenient, but not required, for using the OptiFlash DLL: **FlashParseOptionString**, **FlashMakeOptionString** and **FlashGetErrorString**.

**unsigned int FlashParseOptionString (FlashIO *pIO, char *pszCmdLine,
unsigned int *puOptions,
FlashParameters *pParams);**

This function parses a specified command-line options string and places the results in a FlashParameters structure.

pIO	points to the initialised **FlashIO** structure.
pszCmdLine	points to a nul-terminated character string containing the command line options.
puOptions	points to a variable into which the number of options will be placed. If the command line contains errors, **puOptions* will be set to 0 and the return value to **FLASH_ERR_BAD_CMDLINE**.
pParams	points to the **FlashParameters** structure.

**unsigned int FlashMakeOptionString (FlashParameters *pParams,
char *pszCmdLine,
unsigned int uMaxLen);**

This is the inverse function to **FlashParseOptionString**; it constructs a command-line parameter string from the values in a **FlashParameters** structure.

pParams	points to the **FlashParameters** structure.
pszCmdLine	points to a character array into which the command line options will be placed as a nul-terminated string.
uMaxLen	specifies the size of the character array. If the constructed command line exceeds this size, the function will fail with an appropriate error code.

---

unsigned int FlashGetErrorString (unsigned int uError, char *pszBuffer);

This function converts an error code into a descriptive error text.

uError	specifies the error code.
pszCmdLine	points to a character array into which the error text will be placed as a nul-terminated string. The supplied buffer must be large enough to accommodate the string (50 characters or more).

6.2 The structures

There a two defined structures, *FlashIO* and *FlashParameters*.

FlashIO

contains a handle used internally by the OptiFlash DLL, and (optionally) pointers to callback routines. The callbacks will be invoked to supply the application with display information (status messages, progress of operations etc).

Before passing this structure to OptiFlash, the *ulSize* member must be set to `sizeof FlashIO`.

FlashParameters

contains various parameters for the flashing process itself. This structure can be either filled directly by the calling application, or can be filled by using a command line string and passing it to the function **FlashParseOptionString**.

7 Bootloader protocol

It may be found desirable to develop an application which is able to communicate with a phone *without* using OptiFlash (for production issues for example). The following is a description of the communication protocol that is implemented in the phone's ROM. The protocol can be used to load code and data into the phone's RAM and execute it. There are two different variations of bootloader protocols, 'active' and 'passive'. The 'passive' protocol is the newer protocol and is the only one that will be used in the future.

7.1 Passive bootloader

The bootloader will do a minimal EMI (External Memory Interface) setup appropriate for the hardware platform it is running on. External RAM will be mapped to address 0 (mapping range 256KB). No further assumption apart from the RAM mapping should be made about the EMI setup.

All that the bootloader is capable of, is copying data into RAM or reading data out of RAM. Every operation beyond this has to be done by a user-written program that is to be loaded into RAM by the bootloader.

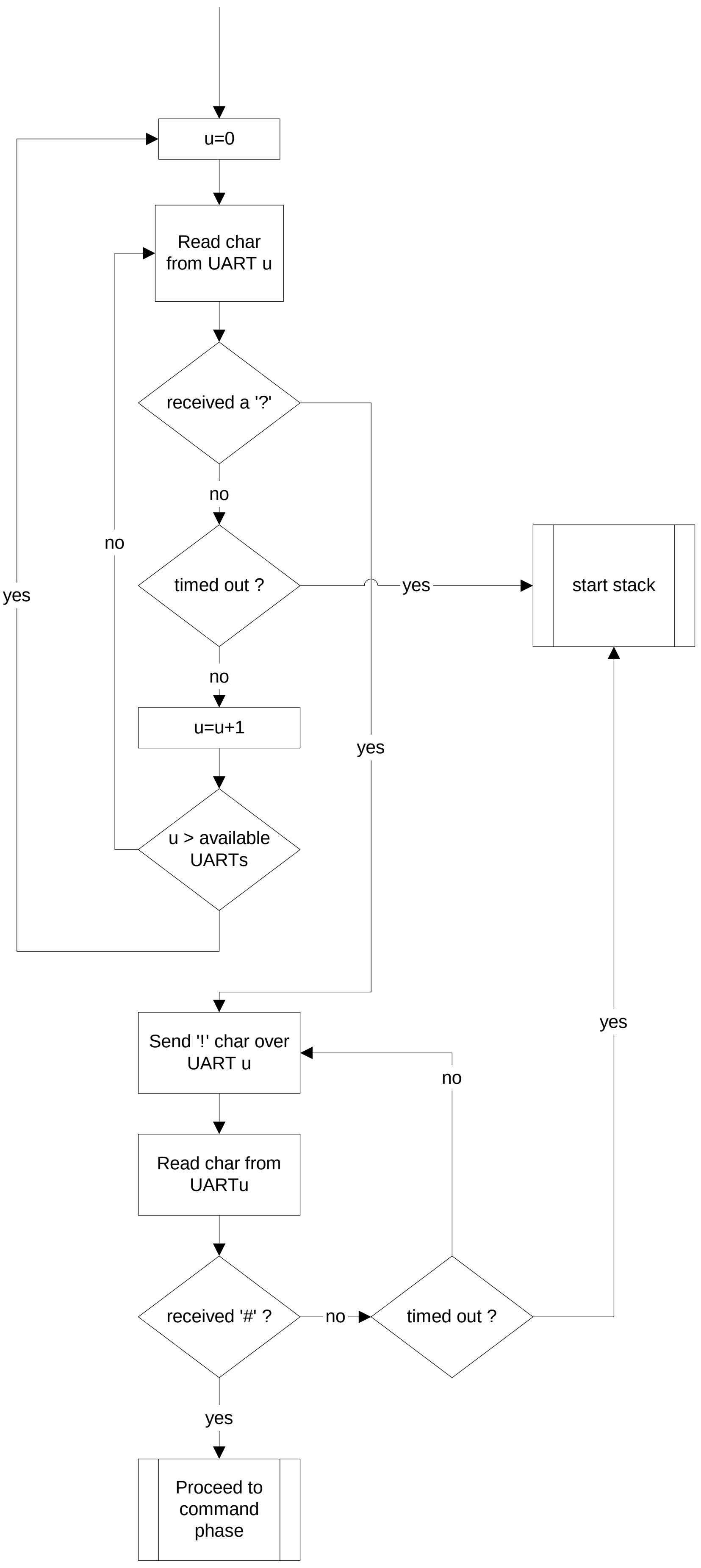
7.1.1 Initial setup phase

When the phone is powered on, the bootloader enters the initial setup phase. In this phase the bootloader checks whether a PC is attached to phone and if so, whether an application wants to communicate with the bootloader. If the bootloader does not detect such an application, it transfers control to the code in flash memory. Otherwise it enters the bootloader protocol.

After power-up, the bootloader in ROM checks cyclically all available UARTs to see whether characters are being send to the phone. If the phone does not receive a '?' character in ~200ms on any UART, the bootloader will transfer control to the code in flash.

If the bootloader receives a '?' on one of the UARTs, it repeatedly sends '!' characters over this UART until either a timeout occurs or a '#' character is received on that UART.

A flow chart of the initial bootloader protocol phase is shown below.


u=0
Read char from UART u
received a '?'
no
timed out ?
yes
start stack
no
u=u+1
yes
u > available UARTs
no
yes
Send '!' char over UART u
Read char from UARTu
received '#' ?
no
timed out ?
no
yes
yes
Proceed to command phase

7.1.2 Command phase

When the initial setup phase is complete, the bootloader enters the command phase. This is basically a loop that waits for the PC-side application to send commands.

All commands consist of multiple 4-byte codes (called 4CC = four character code). A list of the defined 4CCs can be found in the bootloader source code (file ***blprot.h***).

Most of the commands (send by the PC application) and the responses (from the bootloader) have the same structure. There are two exceptions: the `PC_SYNC` command, which is used for synchronisation, and the `PC_DONE` command, which ends bootloader execution.

7.1.3 Synchronisation

If the PC application loses synchronisation, it sends the 4CC for synchronisation (PC_SYNC) and stops sending characters until it receives a synchronisation acknowledgement (MOBILE_SYNC_ACK) from the mobile. When the bootloader encounters PC_SYNC, it enters a loop that reads characters from the UART and discards them. When no more characters are received, the bootloader sends MOBILE_SYNC_ACK. Now the application and the bootloader are synchronised again.

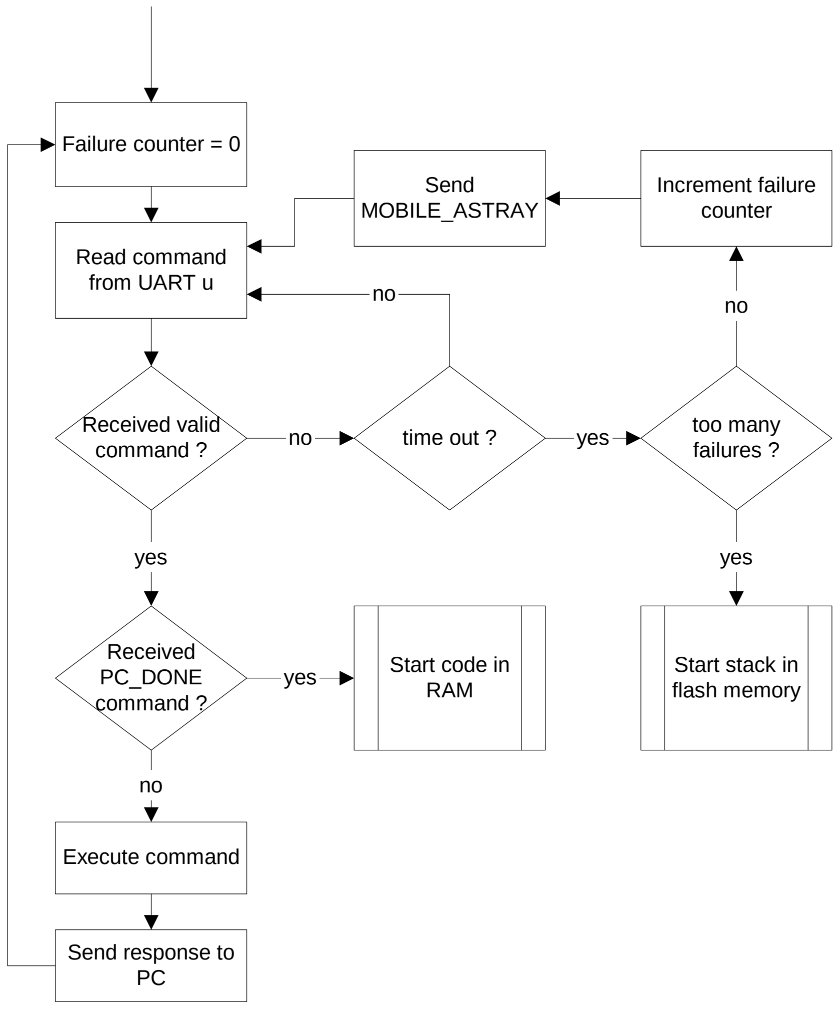
7.1.4 Command loop

During the command phase, the bootloader is basically executing a loop in which it is waiting for commands to be sent by the PC application. These commands are then executed by the bootloader and an appropriate response is send to the application. If the communication loses synchronisation, it is the responsibility of the application to re-synchronise. Whenever the bootloader times out while waiting for a command, it sends a MOBILE_ASTRAY code to the PC (this is useful for debugging, to check whether the phone is still alive).

The command loop is terminated by one of two events:

- the wait for a command times out
  The bootloader aborts the communication and starts the stack code in flash memory.
- the PC_DONE command is received
  The bootloader transfers control to the code at the beginning of RAM.

A flow chart of the command phase protocol phase is shown below.


Failure counter = 0
Send MOBILE_ASTRAY
Increment failure counter
Read command from UART u
no
no
Received valid command ?
no
time out ?
yes
too many failures ?
yes
yes
Received PC_DONE command ?
yes
Start code in RAM
Start stack in flash memory
no
Execute command
Send response to PC

7.1.5 Command format

All commands sent by the PC application (except `PC_SYNC` and `PC_DONE`) have the same basic message layout. Every code in the message is 4 bytes in size, though the transmitted data can have any size.

	Contents of field	Description
1	4CC command token	The 4CC token of the requested command.
2	Size of associated data	The size in bytes of the data associated with this command.
3	Associated data	The data associated with this command. Must be exactly the number of bytes as specified in step 2.
4	CRC over associated data	A 32-Bit CRC of the data in field 3.
5	4CC PC_EOM	End-Of-Message token (PC side).

Note: steps 4-5 are only present if the size of the associated data in step 2 is not 0.

7.1.6 Response format

All responses sent by the bootloader (except MOBILE_SYNC_ACK and MOBILE_ASTRAY) have the same basic message layout. Every code in the message is 4 bytes in size, though the transmitted data can have any size.

Field	Contents of field	Description
1	4CC MOBILE_RESP	Start-Of-Response token
2	4CC command token	The command token, for which this response is the answer.
3	MOBILE_ACK | MOBILE_NACK	Depending on the success or failure of the requested command, the appropriate token is send.
4	Size of associated data	The size in bytes of the data associated with this response.
5	Associated data	The data associated with this response. Must be exactly the size specified in field 4.
6	CRC over associated data	A 32-Bit CRC of the data in field 5
7	4CC MOBILE_EOM	End-Of-Message token (mobile side)

Note 1: fields 4-6 are only present if the command completed successfully, i.e. field contains MOBILE_ACK.

Note 2: fields 5-6 are only present if the size of the associated data (field 4) is not 0.

7.1.7 Bootloader commands

Below is a list of commands understood by the bootloader during the command phase. The input and output descriptions refer to the data associated with the commands.

Note: where bit numbers quoted, bit 0 refers to the low-order bit. In general, bit **n** refers to the bit with value **2^n**.

`PC_SYNC`

See description in section **7.1.3**, "**Synchronisation**"

`PC_INFO`

Input: none

Ouput:

Field	Type	Contents
1	UINT32	Bits 0- 7: bootloader version (minor) Bits 8-15: bootloader version (major) Bits 16-23: bootloader type Bits 24-31: unused
2	UINT32	Content of the Trident ID register
3	UINT32	Index of UART used for communication (starting at 0)
4	UINT32	Max. size in bytes the bootloader is able to receive, i.e. no command send to the bootloader may exceed this size !

PC_DATA

Input:

Field	Type	Contents
1	UINT32	Address in RAM (starting at 0) where the transmitted data should be placed
2	UINT32	Size of transmitted data in bytes
3	-	Data to be transmitted

Output: none

PC_SUCK

Input:

Field	Type	Contents
1	UINT32	Address in RAM (starting at 0) of the data to be sent to the PC
2	UINT32	Number of bytes to send

Output:

Field	Type	Contents
1	-	Requested data

PC_DONE

This command will cause the bootloader to jump to address 0 in RAM, i.e. the bootloader execution ends and control is passed to the code just loaded into RAM.