

# Pico-C Chip Commander

## PicoC

Torsten Jaekel  
April 2024

### Contents

Overview.....	3
Hardware Overview.....	4
Usage Approach.....	4
Configuration.....	5
How to start?.....	5
Commands.....	8
C-Library Commands.....	8
void printf(char *, ...);.....	9
char *sprintf(char *, char *, ...);.....	9
void gets(char *, int);.....	9
int getchar();.....	9
void *malloc(int);.....	10
void free(void *);.....	10
void strcpy(char *,char *);.....	10
void strncpy(char *,char *,int);.....	10
int strcmp(char *,char *);.....	10
int strncmp(char *,char *,int);.....	10
void strcat(char *,char *);.....	10
char *index(char *,int);.....	10
char *rindex(char *,int);.....	10
int strlen(char *);.....	10
void memset(void *,int,int);.....	10
void memcpy(void *,void *,int);.....	10

int memcmp(void *,void *,int);.....	10
Command Extensions.....	10
void help();.....	10
unsigned long readmem(unsigned long);.....	10
void writemem(unsigned long,unsigned long);.....	10
unsigned long preadmem(unsigned long);.....	11
void pwords(unsigned long,unsigned long);.....	11
void pdump(unsigned short *,unsigned long);.....	11
void pbytes(unsigned long,unsigned long);.....	11
void runscript(char *);.....	12
int exitvalue();.....	12
SD Card related commands.....	12
int sdopen();.....	12
int sdclose();.....	13
int sddir();.....	13
int sdload(char *, unsigned long, unsigned long);.....	13
int sdread(char *,void *,unsigned long);.....	13
int sdprint(char *);.....	13
int sddelete(char *);.....	13
int sdcreate(char *);.....	14
int sdappend(char *);.....	14
int sdwrite(int, void *, int);.....	14
int sdreceive(char *);.....	14
int sdformat(unsigned long);.....	15
void sdscrip(char *);.....	15
void sdhprint(char *);.....	15
int sdmemwrite(int,unsigned long,int);.....	15
int sdwritefifo(int,int);.....	16
Chip Specific Commands.....	16
void Cclrintcnt();.....	17
unsigned long Cgetintcnt();.....	17
unsigned short Cgetchipid();.....	17

unsigned short Cgetintstat();.....	17
unsigned short Cgetspistat();.....	17
unsigned short Cpeek(unsigned short);.....	17
void Cpoke(unsigned short,unsigned short);.....	17
int Creadregs(unsigned short,unsigned short *,int);.....	18
int Cwriteregs(unsigned short,unsigned short *,int);.....	18
int Creadblk(unsigned short,unsigned short *);.....	19
int Cwriteblk(unsigned short,unsigned short *);.....	19
int Cspitrans(unsigned char *,unsigned char *, int);.....	20
void Cresetfifo();.....	20
int Creceivevals(char *, unsigned short *);.....	20
Other commands.....	21
void sdraminit();.....	21
unsigned long gettimestamp();.....	22
void delay(unsigned long);.....	22
Example Script.....	22
Explanations and Hints.....	26
Sending scripts via UART.....	26
How to use Host Side Scripts?.....	27
UART flow control.....	27
Define and user-functions.....	27
MFIFO record on SDRAM.....	28
HW Setup.....	28

## Overview

*PicoC* is an Embedded Firmware running on ARM processors, e.g. STM32F4 boards or PortentaH7.

It provides a UART terminal with **C-like command syntax**. The Pico-C is a project taken several years back as Open Source on Internet, improved and extended over the time meanwhile (e.g. bug fixes and extension for UVM functions).

PicoC works only on and with a **UART terminal** (not yet a full network remote access possible, e.g. from a Python script).

Users can write scripts in C-like language (not a full C89 or C99 language feature set and some additions, but very similar related to main features of the C-coding language) and let it execute as interpreted C-code (no compiler needed to run C-like Pico-C scripts, license free, no need for additional tools).

*PicoC* provides commands as C-functions.

These C-functions can communicate to a chip (here a sensor chip is used as example) via SPI, based on high-level functions (commands).

*PicoC* is used to bring up chips, connected at the MCU board via SPI, using C-like scripts, stored on SD Card or transferred via UART to the MCU board and executed there.

## Features

Almost all C-code statements are possible, including standard c-library functions, e.g. `strlen()`;

PicoC can also execute scripts (as text files) stored on SD Card (see `RunScript(char *);` )

## Limitations

Not supported in Pico-C are C constructs like:

- `typedef`
- *arrays of structures* (but a singled structure is possible)
- *unions* (and *array of unions*)
- complex pre-processor macro substitutions (e.g. strings via `##`)

## Hardware Overview

The platform, the MCU board, to run *PicoC* needs and provides:

- SPI interface in order to connect a chip, target board intended for the bring-up, to use this remote chip from MCU board via terminal sessions
- A UART interface, used as connection to a host PC via a Serial Terminal Program
- MCU board provides an SD Card module or slot where scripts or drained data can be stored  
Remark: only a **FAT File System** is supported (right now).

The MCU board is powered via a second USB interface, acting as debug interface (not used and needed in operation mode). But this USB connection is needed to power the MCU board.

## Usage Approach

Users will open a Serial Terminal Program, e.g. *TeraTerm* and will have access to the commands.

Commands are C-like function calls. Users can define new function, variables etc., by writing scripts in C-like language.

Commands or scripts are interpreted and executed if properly specified. There is no need for a C-Compiler or compiler turn around.

Scripts can be stored and executed from SD Card or transferred via UART to the MCU board. Even copying files from host via UART to SD card is possible, so that the script is permanently available in SD Card and faster to execute.

Already defined variables, functions and user defined functions or variables can be used in scripts or they can be invoked from command line (terminal session).

Drained data is stored on a mass storage device, e.g. inside on-board SDRAM or on SD Card. This data can be transferred after tests to the host PC, e.g. via UART or taking SD Card to host in order to copy files from it onto the host.

The MCU firmware provides *Pico-C* as a command interpreter which will accept new function definitions and calls, variable and code definitions in a C-like language format. The already defined functions create a set of Standard-C-Library functions, such as `memcpy()`; `strlen()`; and also commands for basic communication to a chip (SPI transfer, read/write registers on a remote chip, check if interrupts are seen, use the SD card etc.).

## Configuration

- Configure UART on host as:  
Baud rate: 115200 (or on some board with VCP UART: any baudrate)  
8bit  
No Parity  
1 Stop bit  
No Flow Control

If UART communication is correct, the user will see a welcome message and a command line prompt as "> "

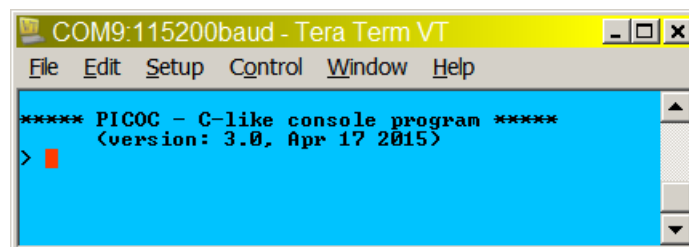


Figure 1: Welcome message and command line prompt

## How to start?

There is a `CHelp()` command:

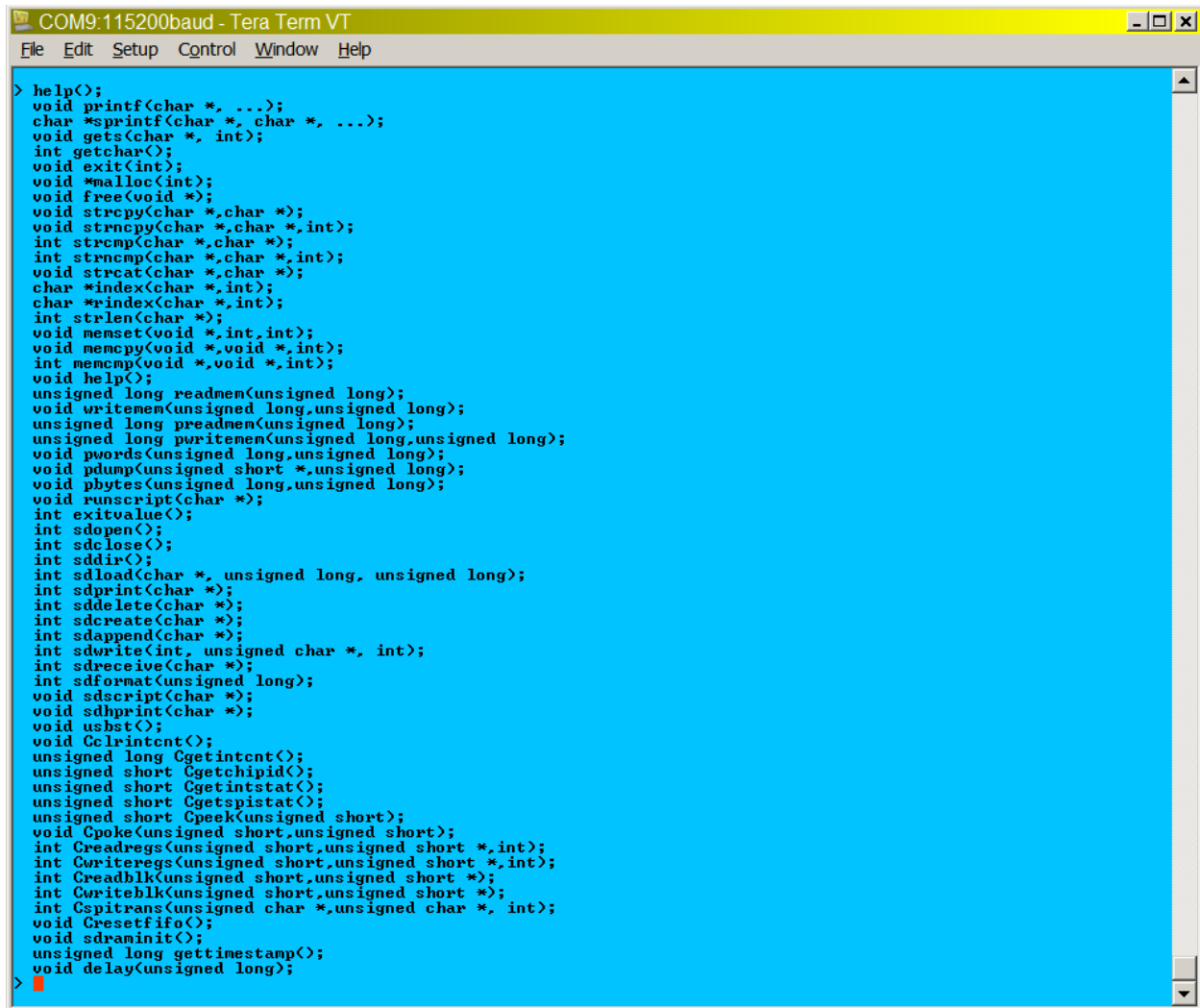
Please bear in mind:

- Commands are C-like functions calls
- Such a function call is native C as: *function name, open parameter body, parameters if needed, close function body* and: **semicolon** `;` **to complete the statement.**
- The command line can be spread over several lines: important is that the command (statement) is completed with semicolon `;` at the end. Until the semicolon was not seen, the command line statement is not yet completed, even several new lines entered in between.

The `CHelp()` command prints all the available, “hard-coded” functions.

Remark: if the user defines new functions – there are not listed via help, just the basic, hard-coded commands.

The `CHelp()` command lists all the commands as C-prototypes, with the return value, function name and parameter list. It does not provide detail help about the meaning of parameters or functions (please see below inside this document).



```
> help();
void printf(char *, ...);
char *sprintf(char *, char *, ...);
void gets(char *, int);
int getchar();
void exit(int);
void *malloc(int);
void free(void *);
void strcpy(char *,char *);
void strncpy(char *,char *,int);
int strcmp(char *,char *);
int strncmp(char *,char *,int);
void strcat(char *,char *);
char *index(char *,int);
char *rindex(char *,int);
int strlen(char *);
void memset(void *,int,int);
void memcpy(void *,void *,int);
int memcmp(void *,void *,int);
void help();
unsigned long readmem(unsigned long);
void writemem(unsigned long,unsigned long);
unsigned long preadmem(unsigned long);
unsigned long pwritemem(unsigned long,unsigned long);
void pwords(unsigned long,unsigned long);
void pdump(unsigned short *,unsigned long);
void pbytes(unsigned long,unsigned long);
void runscript(char *);
int exitvalue();
int sdopen();
int sdclose();
int sddir();
int sdload(char *, unsigned long, unsigned long);
int sdprint(char *);
int sddelete(char *);
int sdcreate(char *);
int sdappend(char *);
int sdwrite(int, unsigned char *, int);
int sdreceive(char *);
int sdformat(unsigned long);
void sdscrip(char *);
void sdhprint(char *);
void usbst();
void Cclrintent();
unsigned long Cgetintent();
unsigned short Cgetchipid();
unsigned short Cgetintstat();
unsigned short Cgetspistat();
unsigned short Cpeek(unsigned short);
void Cpoke(unsigned short,unsigned short);
int Creadregs(unsigned short,unsigned short *,int);
int Cwriteregs(unsigned short,unsigned short *,int);
int Creadblk(unsigned short,unsigned short *);
int Cwriteblk(unsigned short,unsigned short *);
int Cspitrans(unsigned char *,unsigned char *, int);
void Cresetfifo();
void sdraninit();
unsigned long gettimestamp();
void delay(unsigned long);
>
```

Figure 2: result of the `CHelp()` ; command

Remark:

The UART command line does not provide (yet) any editing functions, such as insert between characters, a command line history etc. The only function which is in place is to delete characters from the end via Backspace.

The statement entered on the command line is executed immediately. It means:

- If a function call is done, e.g. as:  
`CHelp()` ;  
it will be executed after semicolon and Enter on keyboard pressed.

- A definition, if correct in syntax, is taken and done, e.g.  

```
unsigned long myVar;
```

 Afterwards such a defined variable can be used immediately.
- Statements which are correct, are executed immediately, e.g.  

```
myVar = 10;
```
- Definitions are taken and stored. But the definition must be complete. For instance, if a new function will be defined, such one has to finish with the closing function body (last '`}`'):
 

```
void MyFunction(int myParam)
{
    int myLocalVar;
    for (myLocalVar = 0; myLocalVal < 10; myLocalVar++)
        printf("%d\n", myParam * myLocalVar);
}
```

It is completed just with the last closing curly bracket '`}`'.

Otherwise, all other commands, statements are taken as part of the function definition.

Remark: the function is just defined, not yet called.

- User defined functions, can be seen as new commands and are executed via a function call:  

```
MyFunction();
```
- Pre-Processor statements are also possible. But the macro processing is very limited:  

```
#define A 10
printf("%d | %x\n", A, A);
```

 or:  

```
#undef B           //just to make sure in case already defined
#ifdef A
#define B 20
#endif
printf("%d\n", B); //would fail if A is not defined !
```
- ATT: make sure, that all definitions, macro usages are complete and closed, e.g. `#endif` nesting is correct. Otherwise the command line will not return to the top level: all is considered still as part of the not closed definition or macro block.
- Existing user defined functions (not hard-coded functions), variables and macros can be deleted. They are not available and existing after the related delete functions. So, they can be defined again, with new types, sized, value etc.:  

```
#undef A
delete MyFunction;
delete myVar;
```



Remark: if a definition of a function, variable or macro is already there, it will not be redefined. An error message will be displayed.

Solution: use `delete` or `#undef` before.

Best practice: `delete` or `#undef` all definitions at the end or at the beginning of a script.

Left definitions can block to run the same script or definition again.

It is harmless trying to delete something which is not defined or available anymore, e.g.

```
delete MyNotDefinedVar; //no error even it does not exist
```

Notice: if an error happens during the definition of a user-function – even deleting this function is accepted – the parser is out of sync (stuck in wrong state) and will not work properly afterwards. A new function definition will not work.

Make sure not to have syntax errors during function definitions.

## Commands

Please find here following the pre-defined, “hard-coded” commands. These commands cannot be deleted and redefined. Just user-defined commands can be deleted and afterwards defined again.

Remark:

Commands, functions can require a parameter list. The provide parameters must match in terms of type. A return value provided can be used in a script. But similar to C-code, a return value can be also left un-assigned to a variable.

## C-Library Commands

Remark:

The following functions (commands) are examples of standard C-library functions available. It depends on the *PicoC* generation/configuration which are really available. For instance to save space some might be not enabled or additional functions, e.g. from the Math Library, e.g. `sin(float)`, can be available too.

Remark:

All commands (with some exceptions marked clearly) use variables, memory addresses etc. inside the MCU. Such parameters are not related to a remote chip, e.g. register address. Parameters when needing an address are related to the inside of the MCU itself. Such an address is local, inside the MCU.

### `void printf(char *, ...);`

Similar to the `printf` function on C-libraries. The first parameter as `char *` is a string address for the format. The format specification is limited (no floating point, e.g. `%f` or `%e`, no complex formats, e.g. `%2.4x`, just `%4x`).

Variable numbers of parameters are possible as known from C-like printf.

Example:

```
printf("my format %d %06x %4d\n", mVar1, myVar2, 20);
```

or:

```
char * fmtStr = "%d %x %4x\n";  
printf(fmtStr, 10, 20, 30);
```

**char \*sprintf(char \*, char \*, ...);**

**void gets(char \*, int);**

**int getchar();**

***void exit(int);***

Parameter: the exit value which can be retrieved later

will just finish a script. It does not reboot or kills the command line. On a command line it does not have any effect, except to store the value as exit parameter which can be checked with a different command, `exitvalue()` ; .

```
void *malloc(int);  
void free(void *);  
void strcpy(char *,char *);  
void strncpy(char *,char *,int);  
int strcmp(char *,char *);  
int strncmp(char *,char *,int);  
void strcat(char *,char *);  
char *index(char *,int);  
char *rindex(char *,int);  
int strlen(char *);  
void memset(void *,int,int);  
void memcpy(void *,void *,int);  
int memcmp(void *,void *,int);
```

### Command Extensions

These commands are considered as higher level commands in order to access the MCU internal memory or features.

Remark: Which user commands are provided depends on the platform used. New user commands (C-code functions) can be added on the MCU FW source code.

#### **void CHelp();**

It prints the list of pre-defined (hard-coded) functions available. The user-defined functions (done during runtime, with scripts) are not displayed.

#### **unsigned long readmem(unsigned long);**

Parameter: a memory address as value (immediate), an address as value inside the MCU board

Return: the value read from this address

This function operates quiet, nothing displayed. If you want to see directly the value read, displayed, please use `preadmem (...) ;` .

#### **void writemem(unsigned long,unsigned long);**

Parameter 1 : the memory address as value (immediate), as address inside the MCU board

Parameter 2 : the value to write there

Example (write to SDRAM):

```
writemem(0xD0000000, 0x11223344);
```

### **unsigned long preadmem(unsigned long);**

Similar to `readmem(...)`, just print also the value read from the memory address

Example (read from SDRAM):

```
preadmem(0xD0000000);
```

### **unsigned long pwritemem(unsigned long, unsigned long);**

Similar to `writemem(...)`, just print also the value read from the memory address after writing (write and read back plus display)

Example:

```
pwritemem(0xD0000000, 0x11223344);
```

### **void pwords(unsigned long, unsigned long);**

print (display) 32bit words as memory dump, starting at address (parameter 1), for length in bytes (parameter 2)

Parameter 1: address of memory location (inside MCU) – make sure address is accessible

Parameter 2: length in bytes (rounded up to 32bit words),  $\geq 4$  as value for 32bit words

Example (print SDRAM content):

```
pwords(0xD0000000, 128);
```

### **void pdump(unsigned short \*, unsigned long);**

print (display) an array of unsigned short, for the sensor chip, e.g. a register value table, as internal (Pico-C) variable. `pdump()` is intended to print unsigned short arrays as the sensor chip words.

Parameter 1: address of an array variable

Parameter 2: the length of 16bit words to display from array

Example:

```
unsigned short myArray[10] = {1,2,3,4,5,6,7,8,9,0};  
pdump(x, 10);
```

Example: reading sensor chip registers (all) and display content of all read registers:

```

unsigned short regVals[257];    //first is dummy!
Creadregs(0x00, regVals, 256); //read all registers
pdump(regVals, 256);           //first is SPI_STATUS!

```

### **void pbytes(unsigned long,unsigned long);**

print (display) a memory inside MCU as byte dump

Parameter 1: address (inside MCU)

Parameter 2: length in bytes

Example (display SDRAM as bytes):

```
pbytes (0xD0000000,128);
```

### **void runscript(char \*);**

Enter UART mode to receive a File. The parameter is a file name, any file name is possible, enclosed on double quotes "filename". The file name is just used for error report.

The command line will change into empty line. All characters received via UART are used as script file. Please, do not enter anything on command line. It will be taken as received script content.

You can use "Send File", e.g. in *TeraTerm*. Send an ASCII text file intended as a script. Such a script should be correct in terms if syntax.

As 'end of file' have a dollar sign \$ as the last character in file (or a CTRL-Z (0x1A)). Or press \$ or CTRL-Z after a file transmission was completed (e.g. after few seconds a file was transferred and sure that all was received, but a \$ or CTRL-Z was not part of the file).

The received script is executed immediately after \$ or CTRL-Z was seen.

Example:

```
runscript("filename");
```

### **int exitvalue();**

It returns just the exit value set on a previous `exit(val)`;

## **SD Card related commands**

### **int sdopen();**

It opens the SD Card with a FAT16 file system. The FAT16 File System has the limitation of **maximal 4 GB storage** able to use. And: FAT16 uses **file names as 8.3**: maximal 8 characters for file name (lower or capital) and maximal 3 characters as file extension. Long file names are not supported.

It reads the Master Boot Record (MBR) in order to have the File System parameters available.

This command is needed as first on all SD card related commands.

Remark: **SD card and SDRAM are exclusive**. If SD card is opened, the access to SDRAM is not possible. Please, make sure that a MFIFO Scan is not active and running (it would not store anymore data on SDRAM). The SD card should just be used if a Scan is inactive. Close SD card if not needed anymore (it will enable SDRAM again).

### **int sdclose();**

Close SD card and make SDRAM available again. All other SD card commands cannot be used anymore.

### **int sddir();**

If SD card was opened via `sdopen()`; the directory of the file system root is displayed.

Remark: Sub-Directories are not supported right now, even they might be displayed. Files in Sub-Directories cannot be used or created. Use all files on root directory entry only.

### **int sdload(char \*, unsigned long, unsigned long);**

Load a file (any, ASCII or binary, at it is) specified via file name to MCU internal address (parameter 2) with length in bytes (parameter 3).

Parameter 1: a file name (as string or with double quotes “), valid as 8.3. FAT16 file name

Parameter 2: memory address (immediate), inside MCU board (a free memory region)

Parameter 3: the length to load in bytes

Example:

```
sdload("config.bin", 0x20001000, 1024);
```

### **int sdread(char \*,void \*,unsigned long);**

Read a binary (!) file from SC card into an array variable.

Parameter 1: file name stored on SD card

Parameter 2: address of the array variable where to store

Parameter 3: length of bytes to read from file and to write to array. Make sure array variable is large enough to store

### **int sdprint(char \*);**

Print content of an ASCII file (!).

Parameter: valid file name for an ASCII file on SD card

### **int sddelete(char \*);**

Delete an (existing) file on SD card. The memory on SD card is freed and file name removed from directory entries.

Example:

```
sddelete("filename.ext");
```

### **int sdcreate(char \*);**

It creates a new, empty file. It is used before `sdwrite(...)`; is used. It creates and opens the file for following writes.

Remark: **You have to use this function before to use `sdwrite()`; .**

### **int sdappend(char \*);**

Open existing file, without to delete or remove content. It is prepared for following `sdwrite(...)`; in order to append file content. Following writes will write after existing file end.

Remark: **You have to use this function before to use `sdwrite()`; .**

### **int sdwrite(int, void \*, int);**

Write bytes (binary or ASCII, as it is) to the opened file. **You must use `sdcreate(...)`; or `sdappend(...)`; before.**

Parameter 1: fileID – not used currently, you can use 0 here for now

Parameter 2: the pointer to a character (byte) buffer, any type can be passed in here

Parameter 3: the length of bytes to write

Remark: **this command can take a lot of time to complete.** The file is written and all File System entries are updated to make it consistent after every call. It can have a large delay. There is no need to close a file, the file is available after the command (consistent File System operation without a dedicated close or flush function to call).

You can keep writing to it with a following `sdwrite(...)`; . It will append to the currently used file.

Remark: **currently you can have only ONE file opened** for `sdwrite(...)`; . A new `sdcreate(...)`; or `sdappend(...)`; is needed for writing to another file. It is NOT possible yet to have more than ONE file open for `sdwrite(...)`; .

Remark: the file system uses an approach to store in a consistent way. There are not buffers to flush, e.g. in order to take the SD card out. Every write will make sure to have all entries on file system properly written, no need to flush or close a file. The drawback: it makes the functions

and commands for SD card quite slow (longer delay due to writing back all file system information on every call).

Before you remove the SD card – please use `sdclose()` ; .

### `int sdreceive(char *);`

This command creates (deletes an existing) a new file on SD card. The content is received via UART. So, you do not need to take SD card out in order to copy a new file to it on host. You can leave SD card in MCU SD card slot and receive and write a file received via UART. It becomes available as file on SD card afterwards, e.g. for `sdscript(...)` ; ;

The command line enters the UART receive mode (empty command line). **Do not press any characters on keyboard (will be part of stored file).**

Send a file – ATT: open as **binary mode**, set in TeraTerm the “BIN” mode checkbox. So, the file is received as sent (when left in ASCII mode some characters, e.g. Line Feed (0x0A), are skipped and script is not error free).

If \$ or CTRL-Z is not part of the transmitted file, you can enter it on terminal program to finish the transfer (press on command line). See also command `runscript(...)` ; .

### `int sdformat(unsigned long);`

This command formats the SD card. All files and data are lost. The parameter is the size in Mbytes and used to figure out how to configure the file system parameters. Please, use 2048 ... 4096 as parameter, assuming to format as a **4 GB SD** card.

Remark: even you use an SD card with a capacity larger as 4 GB – the file system will support only **maximal 4 GB storage space**.

Parameter: the size in Mbytes for the File System storage, use 2048 ... 4096 and at least a 4 GB SD Card.

Example:

```
sdformat(4096);  
sddir();
```

Remark: **SD card and SDRAM are exclusive**. Make sure that a Scan with storing MFIFO on SDRAM is not running in background.

### `void sdscript(char *);`

Load and execute a script as file name from SC card.

Parameter: file name as 8.3 DOS file, as ASCII file

Example:



```
sdcsript("myscript.c");
```

**void sdhprint(char \*);**

Print (display) a file as hex, hex dump of the file content.

**int sdmemwrite(int,unsigned long,int);**

Write the memory content (MCU internal, address as immediate) to SD card. Use `sdcreate (...)` ; or `sdappend (...)` ; before.

Parameter 1: fileID – not used currently – use any value, e.g. 0

Parameter 2: start address of the MUC internal memory, as value (immediate)

Parameter 3: size in bytes to write to file

Remark: you can call in a repeated way, it will append. But you will write to file created via `sdcreate (...)` ; or append to file with `sdappend (...)` ; .

**It is currently NOT possible to handle more as ONE file at a time.**

**int sdwritefifo(int,int);**

Write the MFIFO data, stored on SDRAM (**0xD0000000**, size 0x00800000 = 8 MB). It saves the MFIFO content from SDRAM to a file, created via `sdcreate (...)` ; or append to file via `sdappend (...)` ;

It can be called in a repeated way as long as the `sdcreate (...)` ; or `sdappend (...)` ; was done before, just once. Another call will open another file.

Parameter 1: fileID – not used currently, use any value, e.g. 0

Parameter 2: the size of bytes to write

Example (it will write from start of SDRAM 0xD0000000 used as MFIFO storage, on very call starting at 0xD0000000):

```
sdcreate("mfifo.bin");           //create and open file first!
sdwritefifo(0, 1024);           //write from 0xD0000000 to SD
```

or (entire SDRAM):

```
sdcreate("mfifo.bin");
sdwrite(0, 0x00800000);
```

ATT: due to fact that **SD card and SDRAM are exclusive** – the command will toggle between both. Make sure that a SCAN and write to SDRAM is NOT active in background (Scan is stopped). The command will return with SD opened so that `sddir()` ; can be used immediately afterwards.

ATT: this command will take a lot of time. The file is written in consistent mode. It does not need to be closed or flushed, it will be available after SD card is removed.

If you remove SD card in order to take to host: please use `sdclose()` ; first to make sure.

## Chip Specific Commands

Several commands, functions, are provided in order to talk to a remote chip (a sensor chip assumed here).

### **void Cclrintcnt();**

Reset the internal Interrupt Counter. On every interrupt (GPIO pin, INTB), the counter is incremented, independent of the INT\_STATUS0 register.

### **unsigned long Cgetintcnt();**

Return the value of the internal interrupt Counter.

### **unsigned short Cgetchipid();**

Return the CHIP\_ID. The CHIP\_ID is stored with the first interrupt processed, or any interrupt where the CHIP\_ID can be read successfully. The CHIP\_ID is stored on any interrupt, just once, when it can be read successfully.

Remark: the CHIP\_ID has to be recognized before any DATA\_RDY interrupt will be processed and the MFIFO data stored on the SDRAM (used as FIFO).

Remark: a reset of the MCU board needs one interrupt, as SYS\_RDY interrupt, in order to store the CHIP\_ID and enable the MFIFO draining process. Potentially, the sensor chip should do a reset cycle after the MCU board was reset.

### **unsigned short Cgetintstat();**

Return the INT\_STATUS0 register content when seen a new interrupt. It is the INT\_STATUS register content when an interrupt happens, before it will be cleared.

Remark: the GPIO INTB interrupt handler clears all bits after an interrupt is received and processed. Reading back the INT\_STATUS0 register will reflect the cleared register. In order to check what the INT\_STATUS0 was when an interrupt has happened – use this function to get the value before it was cleared.

### **unsigned short Cgetspistat();**

Return the SPI\_STATUS, the first 16bit word on a SPI transaction. It will be the SPI\_STATUS value when an MFIFO read transaction will be done during the interrupt handler in order to drain MFIFO on a DATA\_RDY interrupt flag.

### **unsigned short Cpeek(unsigned short);**

Read a single sensor chip register and return the value.

Parameter: sensor chip register address (regular sensor chip register), as value (immediate)

### **void Cpoke(unsigned short,unsigned short);**

Write a single sensor chip register with a 16bit value.

Parameter 1: sensor chip register address (regular sensor chip register), as value (immediate)

Parameter 2: 16bit value to write to the register

### **int Creadregs(unsigned short,unsigned short \*,int);**

Read consecutive registers (regular registers) and store in a variable (array of 16bit values).

Parameter 1: register start address (regular register), value (immediate)

Parameter 2: address of a 16bit element array (unsigned short) to store the register content there

Parameter 3: the number of registers to read

ATT: **define the array (size) to store the register values as “number of registers” plus 1.** The array must be one element larger as number of registers to read. The first word is used for the SPI\_STATUS. The first register value related to the start address is in element 1 of the array.

Remark: the **first entry in the array is the SPI\_STATUS word** (16bit), the first value related to the start address is stored in element 1 (index starting at 0).

Example:

```
unsigned short regVals[21];  
Creadregs(0xD0, regVals, 20);
```

### **int Cwriteregs(unsigned short,unsigned short \*,int);**

Write consecutive registers (regular registers) with the value stored in a variable (array of 16bit values).

Parameter 1: register start address (regular register), value (immediate)

Parameter 2: address of a 16bit element array (unsigned short) which holds the register content to write

Parameter 3: the number of registers to write

ATT: **define the array (size) to store the register values as “number of registers” plus 1.** The array must be one element larger as number of registers to read. The first word is used for the SPI CMD during the SPI transaction. The first register value related to the start address is in element 1 of the array. The first element with index 0 is a dummy value.

Remark: the **first entry in the array is used for the SPI CMD** (16bit), the first value related to the start address is stored in element 1 (index starting at 0).

Example:

```
unsigned short regVals[11] = {0,1,2,3,4,5,6,7,8,9,10};  
//the first element and value is dummy, index 1 is related  
//to the start address  
Cwriteregs(0xD0, regVals, 10);
```

### **int Creadblk(unsigned short,unsigned short \*);**

Read consecutive block registers (FIFO, RAMs, via BLKR) and store in a variable (array of 16bit values).

Parameter 1: register start address (FIFO, RAMs, 0..3), value (immediate)

Parameter 2: address of a 16bit element array (unsigned short) to store the register content there

ATT: the length must match with the BLK register length, e.g. MFIFO, address 0x01, as 196 bytes = 98 words plus 1 for SPI\_STATUS. Other RAMs have other length.

ATT: **define the array (size) to store the register values as “number of registers” plus 1.** The array must be one element larger as number of registers to read. The first word is used for the SPI\_STATUS. The first register value related to the start address is in element 1 of the array.

Remark: the **first entry in the array is the SPI\_STATUS word** (16bit), the first value related to the start address is stored in element 1 (index starting at 0).

Example:

```
unsigned short regVals[99];  
Creadblk(0x01, regVals);    //read MFIFO, 0x01, 98*2 =  
//196 bytes to read
```

### **int Cwriteblk(unsigned short,unsigned short \*);**

Write consecutive block registers (FIFO, RAMs, via BLKW) with the value stored in a variable (array of 16bit values).

Parameter 1: register start address (FIFO, RAMs, 0..3), value (immediate)

Parameter 2: address of a 16bit element array (unsigned short) which holds the register content to write

ATT: the length must match with the BLK register length, e.g. MFIFO, address 0x01, as 196 bytes = 98 words plus 1 for SPI CMD. Other RAMs have other length.

ATT: **define the array (size) to store the register values as “number of registers” plus 1.** The array must be one element larger as number of registers to read. The first word is used for the SPI

CMD during the SPI transaction. The first register value related to the start address is in element 1 of the array. The first element with index 0 is a dummy value.

Remark: the **first entry in the array is used for the SPI CMD** (16bit), the first value related to the start address is stored in element 1 (index starting at 0).

Example:

```
unsigned short regVals[257];  
//fill in the value  
//...  
//the first element and value is dummy, index 1 is related  
//to the start address  
Cwriteblk(0x00, regVals); //write entire WVRAM, 0x00
```

**int Cspitrans(unsigned char \*, unsigned char \*, int);**

Low level SPI transaction, send any byte buffer via SPI.

Parameter 1: byte array for the SPI packet to send (pointer)

Parameter 2: byte array to store the received SPI packet during the transaction (pointer)

Parameter 3: the length in bytes for the SPI packet to transfer

Remark: the length is the real length in bytes. The first word in buffer addressed via Parameter 1 has to be set as the 16bit SPI CMD. On the receiving buffer (parameter 2), the first 16bit word will be the SPI\_STATUS.

**void Cresetfifo();**

Reset the start address of the MFIFO FIFO memory (SDRAM) to the beginning of the FIFO (start of SDRAM, 0xD0000000).

**int Creceivevals(char \*, unsigned short \*);**

Example: get the values to write into registers via sending a file via UART from terminal program:

```
unsigned short regVals[257]; //for all 256  
registers, first is dummy entry  
  
sdreceive("any_name", regVals);  
//will enter UART reception mode  
//now send a file with ASCII content which represents the  
values to store on regVals  
  
pdump(regVals, 257); //display all stored values  
//the first is dummy, place holder, starting with index 1  
//it is related to the register values from start address
```

```
Cwriteregs(0x00, regVals, 256); //write all registers
//length is 1 less as size of array, first is dummy
```

The UART reception mode waits for ASCII characters. Please do not enter any character on keyboard (will be send and stored). Or: if you use keyboard – you can – it does not have a remote echo, you had to type input blindly.

The file with ASCII characters you will send from terminal program, e.g. TeraTerm “File” -> “Send File” can have this content (example):

```
0,
0x1234, 0x5678,
1 2 3 4 5
6
7
8,
9,10    11
$
```

The **first value is a dummy** entry (place holder for SPI CMD or SPI STATUS word). The second value (index 1) is related to the values going to registers.

The values can be decimal integer (not floating point) or hexadecimal, starting with **0x**.

Values are separated by comma ‘,’ , space(s), tab(s) or new line.

The “end of file” and end of transmission is marked via the ‘\$’ sign. If it is not part of the file sent – you can also enter it (or CTRL-Z) on the UART terminal command line (finish the transfer via keyboard if not there).

Remark: You cannot use anything else in such a file to send. **Comments are NOT possible to use in such a file for this command.**

**The maximal size of such a file (in ASCII bytes) is 16 KB.** It should be large enough even to transfer WVRAM content. There is not any limitation for the size of one line (up to 64 KB as single line possible).

Remark: do not use too much spaces, e.g. for indents. It will reduce the maximal capacity to transfer values, entire length of file limited to 16 KB.

ATT: **make sure that the size of the variable is large enough to store all these values** (as 16bit unsigned short values taken). There is not a cross-check for a potential buffer overrun!

## Other commands

### **void sdraminit();**

Initialize the access to the SDRAM, address 0xD0000000.

Remark: the **SD Card and SDRAM are exclusive**. If SD Card was used it should enable automatically the SDRAM with `sdclose()`; This function can be used to enable SDRAM manually. SDRAM is initialized and enabled as default. `sdclose()`; should change back to SDRAM access.

ATT: The SDRAM must be enabled, `sdclose()`; be done, when a Scan will be enabled.  
**SD card should not be used when a Scan is active and running in background.**

### **unsigned long gettimestamp();**

Return the current time stamp, based on the 32.768KHz clock generator (based on 32.768KHz for the time stamp counter).

Remark: currently, just 16bit as LSB are returned, a full 32bit time stamp with valid MSB part will be implemented in future version.

### **void delay(unsigned long);**

Delay the execution of commands, scripts by the delay time specified in milliseconds.

Parameter: time to delay in milliseconds

## **Example Script**

An example script is provided as following. It is a script to configure sensor chip for a scan, here as DCS.

This script can be sent via UART `runscript(...)`; or stored as file on SD card and executed from there, `sdscrip(...)`;

Remark:

The script itself does not use so many comments: comments are possible but it would increase the size of the file.

Currently, the maximal file size if scripts from SD Card or sent via UART is **limited to 16 Kbytes**.

Therefore, better not use comments, indents via spaces or many empty lines. Optimize scripts for shortest file size.

The comments as `//comment` are just provided here, not needed to be in the script (just for documentation here). Try to avoid comments, they consume just memory space.

```
delete xregs;           //delete all variable so that we can run
delete rr;              //script again without error due to defined items
delete ic;
delete is;delete ss;

unsigned short xregs[257]; //define array for all registers plus 1

void rr() {              //define a function we can use later
    Creadregs(0,xregs,256); //read all registers: ATT: array length is + 1!
    pdump(xregs,256);      //dump/display all values save in array
}
```

```

rr(); //call the defined function

printf("\n*** PICO-C MCU Scan Test ***\n");

void ic() {printf("%d\n", Cgetintcnt());} //define helper function

void is() {printf("%x\n", Cgetintstat());} //define helper function

void ss() {Cpoke(0xbd, 0x8000);} //define helper function, Start Scan

Cpoke(0x2f, 0xffff); //write single register
Cpoke(0x32, 0x015E);
Cpoke(0x33, 0x1fff);
Cpoke(0x2F, 0xFFEF);
Cpoke(0x90, 0x1c11);
Cpoke(0xA6, 0x000a);
Cpoke(0x16, 0x8f34);
Cpoke(0x22, 0x4000);
Cpoke(0x24, 0x0008);
Cpoke(0x15, 0x0008);
Cpoke(0x27, 0x0002);
Cpoke(0x27, 0x0000);
Cpoke(0x22, 0x0000);
Cpoke(0x24, 0x0000);
Cpoke(0x00, 0x8480);
Cpoke(0x94, 0x3354);
Cpoke(0x81, 0x0008);
Cpoke(0x5F, 0x7fff);
Cpoke(0x17, 0x7fff);
Cpoke(0x70, 0x7fff);
Cpoke(0x34, 0x7fff);

printf("--- Configure WVF RAM ---\n");

unsigned short regs[257]; //define array for WVRAM, length plus 1 word !

regs[0] = 0; //first element is dummy, for SPI CMD or STATUS

unsigned short
x1[16]={0x0015,0x000C,0x0010,0x0014,0x0019,0x001E,0x0025,0x002C,0x0035,0x003F,0x004A,
0x0056,0x0064,0x0074,0x0086,0x0099};

//define parts of the large consecutive array

unsigned short
x2[16]={0x00AF,0x00C7,0x00E2,0x00FF,0x011F,0x0142,0x0169,0x0193,0x01C0,0x01F2,0x0228,
0x0262,0x02A0,0x02E4,0x032D,0x037B};

unsigned short
x3[16]={0x03CE,0x0428,0x0487,0x04ED,0x055A,0x05CD,0x0648,0x06CA,0x0753,0x07E5,0x087E,
0x0920,0x09CB,0x0A7F,0x0B3B,0x0C01};

unsigned short
x4[16]={0x0CD0,0x0DA9,0x0E8C,0x0F78,0x106F,0x1170,0x127C,0x1391,0x14B2,0x15DD,0x1712,
0x1853,0x199D,0x1AF3,0x1C53,0x1DBE};

```



```

unsigned short
x5[16]={0x1F33,0x20B2,0x223C,0x23CF,0x256C,0x2713,0x28C3,0x2A7C,0x2C3E,0x2E08,0x2FDB,
0x31B5,0x3396,0x357E,0x376C,0x3961};

unsigned short
x6[16]={0x3B5A,0x3D59,0x3F5B,0x4162,0x436B,0x4577,0x4785,0x4993,0x4BA3,0x4DB2,0x4FC0,
0x51CD,0x53D7,0x55DE,0x57E1,0x59E0};

unsigned short
x7[16]={0x5BD9,0x5DCD,0x5FB9,0x619E,0x637A,0x654D,0x6717,0x68D5,0x6A89,0x6C30,0x6DCB,
0x6F58,0x70D7,0x7247,0x73A8,0x74F8};

unsigned short
x8[16]={0x7639,0x7768,0x7885,0x7990,0x7A89,0x7B6E,0x7C40,0x7CFE,0x7DA7,0x7E3D,0x7EBD,
0x7F28,0x7F7E,0x7FBF,0x7FEA,0x7FFF};

unsigned short
x9[16]={0x7FFF,0x7FEA,0x7FBF,0x7F7E,0x7F28,0x7EBD,0x7E3D,0x7DA7,0x7CFE,0x7C40,0x7B6E,
0x7A89,0x7990,0x7885,0x7768,0x7639};

unsigned short
x10[16]={0x74F8,0x73A8,0x7247,0x70D7,0x6F58,0x6DCB,0x6C30,0x6A89,0x68D5,0x6717,0x654D,
0x637A,0x619E,0x5FB9,0x5DCD,0x5BD9};

unsigned short
x11[16]={0x59E0,0x57E1,0x55DE,0x53D7,0x51CD,0x4FC0,0x4DB2,0x4BA3,0x4993,0x4785,0x4577,
0x436B,0x4162,0x3F5B,0x3D59,0x3B5A};

unsigned short
x12[16]={0x3961,0x376C,0x357E,0x3396,0x31B5,0x2FDB,0x2E08,0x2C3E,0x2A7C,0x28C3,0x2713,
0x256C,0x23CF,0x223C,0x20B2,0x1F33};

unsigned short
x13[16]={0x1DBE,0x1C53,0x1AF3,0x199D,0x1853,0x1712,0x15DD,0x14B2,0x1391,0x127C,0x1170,
0x106F,0x0F78,0x0E8C,0x0DA9,0x0CD0};

unsigned short
x14[16]={0x0C01,0x0B3B,0x0A7F,0x09CB,0x0920,0x087E,0x07E5,0x0753,0x06CA,0x0648,0x05CD,
0x055A,0x04ED,0x0487,0x0428,0x03CE};

unsigned short
x15[16]={0x037B,0x032D,0x02E4,0x02A0,0x0262,0x0228,0x01F2,0x01C0,0x0193,0x0169,0x0142,
0x011F,0x00FF,0x00E2,0x00C7,0x00AF};

unsigned short
x16[16]={0x0099,0x0086,0x0074,0x0064,0x0056,0x004A,0x003F,0x0035,0x002C,0x0025,0x001E,
0x0019,0x0014,0x0010,0x000C,0x0015};

```

```

//build a large array for write block later, combine all parts to single, large array
//necessary to do because of maximal length of command line limited to 255 characters

```

```

memcpy(&regs[1],    x1,32);
memcpy(&regs[17],   x2,32);
memcpy(&regs[33],   x3,32);
memcpy(&regs[49],   x4,32);
memcpy(&regs[65],   x5,32);
memcpy(&regs[81],   x6,32);

```

```

memcpy(&regs[97],    x7,32);
memcpy(&regs[113],   x8,32);
memcpy(&regs[129],   x9,32);
memcpy(&regs[145],   x10,32);
memcpy(&regs[161],   x11,32);
memcpy(&regs[177],   x12,32);
memcpy(&regs[193],   x13,32);
memcpy(&regs[208],   x14,32);
memcpy(&regs[224],   x15,32);
memcpy(&regs[240],   x16,32);

Cwriteblk(0,regs);           //write entire array as WBLK command to WVAM
//ATT: Cwriteblk has implicit length, depending on address for the block, 0..3
//make sure to define array large enough, with length + 1 and leave 1st element
//unused, as dummy

printf("--- Configure sensor chips ---\n");

delete x1;
delete x2;
delete x3;
delete x4;
delete x5;
delete x6;
delete x7;
delete x8;
delete x9;
delete x10;
delete x11;
delete x12;
delete x13;           //delete variables if not needed - save space
delete x14;           //and to run same script again
delete x15;
delete x16;

unsigned short
x1[16]={0x0012,0x1404,0x1421,0x2021,0x2131,0x004c,0x0000,0x0001,0x0080,0x1000,0x52b2,0
x0084,0x2de0,0x0020,0x0e55,0xc312};

unsigned short
x2[16]={0xe29f,0x4013,0x64b9,0xd8fc,0xa259,0x0000,0x0329,0x0000,0x1200,0x0005,0x0001,0
x0492,0x02f4,0x0012,0x0000,0x0000};

//combine a larger array to write later to regular registers

memcpy(&regs[1], x1,32);      //combine parts to larger array - first is dummy
memcpy(&regs[17],x2,32);

Cwriteregs(0xA8,regs,32);     //write 32 consecutive registers

delete x1;
delete x2;
delete regs;                 //delete if not needed anymore - save space

//prepare new set of values for consecutive registers - first entry is dummy

```

```

unsigned short
regs[9]={0x0000,0x0000,0x0000,0x0000,0x0000,0x0000,0x0000,0x0000,0x2100};

Cwriteregs(0xC8,regs,8);

delete regs;

unsigned short regs[8]={0x0000,0x0000,0x5572,0x5041,0x0352,0x4607,0x3736,0x3e3b};

Cwriteregs(0xD0,regs,7);

printf("*** Done ***\n");

delete regs;

printf("*** Read Back all registers ***\n");

rr(); //call defined function - here register dump

printf("~~~ Enable Scan via 'Cpoke(0xBD,0x8000);' or 'ss();' ~~~\n");

```

## Explanations and Hints

The command line is limited to maximal 255 characters. In order to initialize large arrays, it might be needed to use parts and to combine, via `memcpy()` ; into a final, larger array.

`Creadregs(...)` ; `Cwriteregs(...)` ; `Cwriteblk(...)` ; and `Creadblk()` ; require an array as parameter with **length + 1** (for the SPI CMD or STATUS stored on first index, index 0).

Shorter arrays can be initialized on definition (if it can be done on command line within less of 255 characters for the line). But the number of initialization value must exactly match with the size of the array. An array cannot be initialized partially.

Best practice is to `delete variable;` or to `delete function;` if not needed anymore: it saves space (releases memory for other definitions) and it makes sure to run the same script again:

If a script would define a variable or function without to delete – a second run of the same script will result in an error: the variable or function is already defined and cannot be overwritten, cannot be redefined. A delete is helpful on entry of script to free all for new definitions.

Remark: it is possible to delete also user-defined functions, not used variables.

There is not any error reported if a variable or function should be deleted which does not exist.

Remark: macros can be deleted as well, but using `#undef macro` .

Suggestions:

Try to make a script as short as possible. Avoid spaces, empty lines and comments. All will require more memory space.

Use tabs as indent instead of sequence of spaces.

Currently, a script send via UART of read from SD card is limited to **16KB of maximal file size**. Internally a 64KB large memory is used with dynamic allocation to hold all parsed and defined statements (from scripts and command line).

## Sending scripts via UART

Via the command `runscript("filename")`; it is possible to receive a script via UART, e.g. from TeraTerm, sent as a file. This command will enter a mode where all characters received, **up to 16 Kbyte**, are stored internally on temporary buffer. If the reception is completed – it will be executed as a script.

The “end of transmission” is based on receiving a dollar sign **\$** (not part of any C-code) or **CTRL-Z (0x1A)**.

If the file sent does not end with a **\$** or **CTRL-Z** – you can enter it via keyboard, after the file transmission has completed. Or send it as last character in a host script, when all was sent.

Remark: you can use “File” -> “Send File” in TeraTerm in order to transfer a file as script to the MCU board. It should work without to set checkbox “Binary”, but it does not hurt to set it to “Bin” mode. (if not in binary mode – some characters can be skipped, e.g. Linefeed 0x0A might be dropped from the transmission).

ATT: for `sdreceive("filename.ext")`; **set always binary mode** when sending a file.

## How to use Host Side Scripts?

The UART uses “remote echo”: every character entered (sent) on a terminal program, e.g. TeraTerm is replied by the remote MCU board.

Therefore, using a UART in a script will receive also all characters sent. It might be important to clear UART receiver buffers or to parse accordingly the received characters and received string if the result of a command should be evaluated. Skip the remote echo before the result of a command will be found.

The “end of line” is Carriage Return (CR, 0x0D). A Line Feed (LF, 0x0A) should work as well as end of line.

If a line received is longer than 255 – 1 characters – it will be terminated automatically. This can result in errors when parsing commands and lines (statements cut or split into two separate lines).

Be aware of fact that the MCU board will send also prompt characters back to the host. The UART communication behaves in the same way like a terminal program used. Any command send will result in responses plus a new line with command prompt.

This command prompt might be needed to skip when parsing responses.

## UART flow control

The command prompt sent by the MCU board can be used as indication that a command was completely processed, before new data or commands will be sent.

A flow control can be implemented on higher levels, on commands. A script should only continue with a new command if a response for the previous one was received or the command prompt was sent after command was processed.

## Define and user-functions

The pre-defined commands can be used as they are from a script.

If new commands or slightly different behavior is needed, user-defined functions can be defined. The script can send UART strings as commands which will define new user-functions. These user functions can be used afterwards in a script. This can reduce the effort in the main part of the script and reduce complexity.

It is also possible to use SD card in combination with a script, using UART communication. A script can read or write data to SD card.

Just: **currently only one file, one SD card command, at a time is supported.**

It is not possible to load a script from SD card which will load again a script from SD card (nested, included). A script from SD card has to have finished before a new one can be used.

## MFIFO record on SDRAM

The DATA\_RDY interrupts are processed and the MFIFO is completely drained (all potential entries, independent of the number of active channels, 196 bytes, with SPI\_STATUS word).

The MFIFO frames are stored on the SDRAM, starting at address 0xD0000000.

The SDRAM is used as buffer, not a ring buffer. If the SDRAM is full, it will not overwrite the beginning of the SDRAM again. But the process to process DATA\_DRY interrupts remains active. Instead, the MFIFO is drained to an internal buffer. This buffer cannot be accessed or transferred to host.

It makes sure just to drain MFIFO. Otherwise the interrupts will stop if MFIFO is not drained and released again.

The user can reset the SDRAM MFIFO pointer via command `Cresetfifo()`; . This will set the pointer for the MFIFO interrupt again to the beginning of the SDRAM. The drained data should be recorded again starting from address 0xD0000000 there.

The format of the recorded entries on the SDRAM is 32bit aligned and the following (204 byte frames):

`32BIT_TS    SMP_INDEX    MFIFO_0 ... MFIFO_95    6B_STUFFING`

Please be aware of: the words are 16bit words, also the 32BIT\_TS. The 16bit words are Little Endian aligned (LSB part of 32BIT\_TS first, than MSB of time stamp). Also SMP\_INDEX (16bit) with first MFIFO entry is Little Endian Aligned (looks like MFIFO\_0 first, then SMP\_INDEX when dumping memory as 32bit words).

Remark: the 32BIT\_TS time stamp uses currently just 16 LSB bits. The MSB is set to 0xFFFF and can be used as a pattern in order to realize the beginning of a MFIFO frame.

The time stamp is based on the 32.768 KHz clock generator inside MCU, provided as PWM clock output with 50% duty cycle, 3.3V logic.

## HW Setup

**ATT: HW specific, on STM32F4xx board**

SPI1 - see SPI1 on main board silk screen, SPI4 is not used

UART1 - the UART on main board with the Profilic USB-to-UART bridge on main board  
(the user USB VCP UART is prepared, terminal can be changed to use it via FW recompilation,  
not used right now)  
HW flow control enabled but should be used on terminal without flow control.

PB5 - 32.768 KHz (PWM) clock out, 40% duty cycle, 3.3V logic level

PD5 - INTB, GPIO Interrupt as input from sensor chip, 3.3V logic level