# Fast Assembler

## 1.8 28-11-2023
## for SpartaDOS X and BW-DOS 1.4



holgerjanz@abbuc.social
Revision 28-11-2023

# Contents

# Introduction

Fast Assembler is a file to file assembler for SpartaDOS X and BW-DOS, requires at least version 1.4. For more information about BW-DOS 1.4 please refer to https://github.com/HolgerJanz/BW-DOS.The assembler can generate object files and executable files for SpartaDOS X and Atari DOS. The source text must be available in ATASCII files.

The assembler has the following features:
- Support of SpartaDOS X and Atari DOS object files
- Support of relocatable SpartaDOS X blocks
- Support of SpartaDOS X symbols
- The maximum file length of the source code and object data only depends on the file system, as it is a real file to file assembler

Fast Assembler is fully compatible with Quick Assembler, but is not compatible with MAC / 65 or Atari Assembler Editor. The Fast Assembler syntax is supported by MADS the 6502 cross assembler. MADS is available for macOS, Linux and Windows.

The Fast Assembler package contains the following files and directories:

`BIN>FA.COM`
This is the executable file in SpartaDOS X format. This is the only file needed to start the assembler. This file can be copied into a directory, which is contained in $PATH, in order to be able to start the assembler easily from all directories.

`MAN>FA.MAN`
This is the manual page for Fast Assembler (see SpartaDOS X instruction MAN). The most important information is summarized here for your reference. This file should be copied into a directory which is contained in $MAN.

`DOS>`
Contains the BW-DOS executable FA.COM and the manual page FA for BW-DOS.

`EXAMPLES>`
Contains well-commented examples.

`SDXLIB>`
Contains the SDX Lib (see chapter SDX Lib) used by FastAssembler for BW-DOS and some examples.

`SOURCES>`
This directory contains the source code for Fast Assembler. Fast Assembler can assemble itself and can thus be further developed under SpartaDOS X (FA_SDX.ASM) or BW-DOS (FA_BWD.ASM).

`TESTS>`
This directory contains a test environment. It contains the batch file TEST_SDX.BAT for SpartaDOS X and TEST_BWD.BAT for BW-DOS to start the tests. The EXAMPLES, SDXLIB, DOS, BIN, and SOURCES directories are also required for the tests.

# Usage

```
FA src[.ext] [dst[.ext]] [/L|S|E|Y|Q]
```

The source file `src[.ext]` is assembled and the result is saved as an object file `dst[.ext]`. The standard extensions are `.ASM` and `.OBJ`. The drive and path are taken from the location where the assembler was called.

The easiest way to call the assembler is as followed:

```
FA MY_SOURCES
```

The assembler will compile the source MY_SOURCE.ASM and create a binary file MY_SOURCE.OBJ. The progress output will look like this:

```
Pass 0 Lines 42 Pending 1
Pass 1 Lines 42
Completed
```

The progress output contains the line count per pass and the pending lines of pass to be resolved in the next pass. Usually 2 passes are need to compile a source (see chapter The Assembly).

If an error is detected by the assembler the error message looks like this:

```
Pass 0 Lines 42 Pending 1
Pass 1 Lines 42 Pending 1
Error at D1:MY_SOURCE.ASM:13:3E40
  Undeclared label
Failed
```

An error message consists of two lines. The first line contains the source file name, the line number within the source, and the current program counter in hexadecimals. The second line contains the text of the error message. In the example above the error “Undeclared label” occurred in source file D1:MY_SOURCE.ASM in line 13 and the current program counter of the assembly was $3E40. For more information about error messages see chapter “Error Messages” and “List of Error Messages”.

The additional command line options are described In the following paragraphs.

With the /L option, all program lines are output during assembly. The lines are output as the assembler reads them, i.e. all includes are resolved, the line numbers are assigned consecutively, and the current program counter in hexadecimals.

Example line:

```
142:28FE:          lda #$00
```

This is overall line number 142. The current program counter is $28FE followed by the source line.

With the /S option, a summary is output at the end of the assembly, e.g. total number of labels, blocks and memories. The memory is output in the following format:

```
Memory: code-stat_mem-stat_end dyn_mem-dyn_end
```

`code` is the address to which the assembler was loaded (`MEMLO` at the time the assembler was started). The start of the static buffers, e.g. for line processing and file access is specified with `stat_mem` and the end with `stat_end`. The dynamic buffers, e.g. for the symbol table, start at

dyn_mem and end at dyn_end. This can be used to check how much memory was used during assembly. If dyn_end reaches the value of MEMTOP the error “Out of Memory” occurs. See SpartaDOS X and BW-DOS 1.4 command MEM for current MEMLO.

The /E option converts assembly compile errors to DOS error codes beginning with 224 ($E0) if errors occur during assembly. This can be used in batch processing. In BW-DOS 1.4 batch processing will be aborted. In SpartaDOS X the STATUS variable is set and the error number can be checked with IF ERROR, see section Error Messages.

The /Y option dumps the symbol table after assembling with file name dst.SYM. See chapter Symbol Table for explanation of the file structure.

The /Q option suppresses the progress message for the line processing. There is only one summary message per pass. This can be used if the console output is redirected to a file for later analysis and the file should not be polluted by the progress messages.

# Examples

**Atari DOS Programm - RAINBOW.ASM**

This example shows how Atari DOS programs can be created. It's a little demo with classic color scrolling.

Assemble with:

```
FA RAINBOW.ASM RAINBOW.COM
```

Execute with:

```
RAINBOW
```

The program is ended with START.

The program consists of two blocks. One block contains the actual program, the second the start address for automatic start. It is a classic Atari DOS program. Blocks are introduced with the pseudo instruction `blk` (see section Pseudo instructions). The second parameter contains the type. DOS stands for Atari DOS Block. The third parameter is the start address of the block. The instruction `blk DOS $3000` corresponds to the instruction `ORG $3000` or `*=$3000` in other assemblers.

```
* Example for FA
* Demo Atari Rainbow Color
* Atari DOS executable
* press START to end

RTCLK   equ $14
RUNAD   equ $02e0
COLPF2  equ $d018
CONSOL  equ $d01f
WSYNC   equ $d40a
VCOUNT  equ $d40b

* main program Atari DOS block
        blk DOS $3000
* reset console keys
start   lda #$08
        sta CONSOL
loop    clc
* calc next color
        lda VCOUNT
        adc RTCLK
* wait for sync
        sta WSYNC
* set color
        sta COLPF2
* check START
        lda CONSOL
        and #$01
        bne loop
        rts

* run address Atari DOS block
        blk DOS RUNAD
        dta a(start)

        end
```

**SpartaDOS X Program - HELLO.ASM**

This is an example of a SpartaDOS X program. It cannot be run under BW-DOS or Atari DOS. This example uses relocatable blocks and symbols.

Assemble with:

```
FA HELLO.ASM HELLO.COM
```

Execute with:

```
HELLO
```

After entering a name e.g. Holger, it says hello:

```
Hello Holger !
```

The SpartaDOS symbols PRINTF and GETS are used. These symbols are resolved when the program is started. They are called with JSR and await parameters immediately afterwards. If you make a mistake with the name of the symbol, you will get a runtime error.

Example, change this line:

```
PRINTF  smb 'PRINTF'
```

To this line:

```
PRINTF  smb 'PRINTFX'
```

The program can be assembled, but when started under SpartaDOS X the following error message is output:

```
Symbol PRINTFX
154 Symbol not defined
```

More information about the symbols can be found in the “SpartaDOS X Programming Guide”.

The program consists of two blocks, a relocatable block (reloc block) and a memory-reserving block (empty block). Both have in common that the loading address is not known at the time of assembly. Such blocks are always automatically loaded from MEMLO by SpartaDOS X and MEMLO is then incremented. The addresses in the program are then also adapted. For the empty block, SpartaDOS X only continues to set MEMLO, i.e. the memory is not initialized. After the end of the program, SpartaDOS X will reset MEMLO.

```
* Example for FA
* Hello from Atari
* SpartaDOS X executable with
* relocatable code

* used SpartaDOS X symbols
* see SDX programming guide
PRINTF  smb 'PRINTF'
GETS    smb 'GETS'

* name buffer size
nam_siz equ $40

* relocatable code block
        blk reloc main
* print question
        jsr PRINTF
        dta c'What''s your name?'
        dta b($9B,$00)
* get name, due to code
* relocatability you cannot use
* lda <name and ldx >name because at
* compile time we do not know where
* the program will be located!
        lda namev
        ldX namev+1
        ldy #nam_siz
        jsr GETS
* print greetings
        jsr PRINTF
        dta c'Hello %s !'
        dta b($9b,$00)
        dta v(name)
        rts
* vector is needed for get name
namev   dta v(name)

* relocatable data block
* program counter is incremented!
* by BLK EMPTY, symbols must be
* definded beforehand
name    equ *
        blk empty nam_siz main

* blocks for address and symbol update
        blk update addresses
        blk update symbols

        end
```

**Memory-resident program - TSR / TSR_CALL.ASM**

In SpartaDOS X it is very easy to write memory-resident programs. These are programs that are not removed from memory when finished.

This can be achieved with the INSTALL symbol. It is initialized with 0 when a program is started. If INTALL has the value $ FF when the program is ended, it is not removed from memory, i.e. MEMLO is not reset. This makes it very easy to write additional drivers.

You can also give your subroutines a name and thus define new SpartaDOS X symbols. This is done with the following block command:

```
blk update new <address> '<name>'
```

If the name begins with @, it is also recognized as a command in COMMAND.

This example consists of two programs:
1) TSR.ASM defines a new symbol @GREET, which is also known as a command (the name begins with @)
2) TSR_CALL.ASM uses this symbol. For this reason it only works if TSR has been started beforehand and the @GREET symbol is known. If not, the following error occurs:

```
@GREET symbol
154 Symbol not defined
```

When TSR has been carried out, GREET can be called directly from SpartaDOS X, or the program TSR_CALL can be called, which calls the symbol.

**Communication between programs - CALLER / CALLEE.ASM**

In SpartaDOS X you can directly start another program (CALLEE) from one program (CALLER) with the function U_LOAD. The FLAG symbol can be used to transfer a value to the called program. There are other ways of transferring data to the called program. To keep the example simple, these are not used here. The called program can set a return value with the STATUS symbol.

The CALLER program calls another program and sets FLAG beforehand. The value for FLAG and the program name are transferred via the command line, e.g .:

```
CALLER 2 CALLEE.COM
```

The CALLEE program reads FLAG, multiplies the value by 8 and sets STATUS with the result. The MUL_32 function is used for this. This function expects the parameters at certain addresses relative to COMTAB and thus also writes the result.

Both programs output the values of FLAG and STATUS that you have set or received.

**Program to compare files - MY_COMP.ASM**

This example shows how a new external command for SpartaDOS X can be developed. It is similar to the COMP command and compares two files. One difference is that the result is not only output with PRINTF, but also STATUS is set. If the files are not equal then STATUS is set to $ FF(255). Since this is a value above $80(128), it is interpreted by SpartaDOS X as an error. Since the error number $FF(255) has no error text, the following output occurs:

```
255 System ERROR
```

Setting an error number has the advantage that you can respond to it in a batch file with the IF ERROR 255 command. This can be used e.g. for automatic tests if a program to be tested creates a file and the expected result is already available in another file.

**Hardware Fine Scrolling - HSCROL.ASM**

This is an example for horizontal fine scrolling. A display list with one GRAPHICS 0 line is defined. This line is scrolled from right to left in an endless loop.

**ANTIC Demo with Display List und Vertical Blank Interrupts - ANTIC.ASM**

A display list with six different parts is defined. Only display list and vertical blank interrupts are used to change colors and lines are scrolled using fine scrolling.

**GTIA Demo with Graphic and Noise - GTIA.ASM**

The CPU is programmed in parallel to the screen display. The background color and two players are programmed directly via GTIA to produce an animation. Furthermore the sound connection of GTIA is used to produce noise. In 400/800 machines the GTIA is connected to the internal speaker.

**POKEY Demo with Music using Software Timer - POKEY.ASM**

This example shows a little frame work to produce music with a maximum of four voices using a software timer of the operation system.

**HI_SDX/HI_BWD Demo for SDX Lib - HI_MAIN.ASM**

This example shows how to write code that can be compiled as SpartaDOS X executable and Atari DOS executable for BW-DOS using the SDX Lib. It is a modification of the HELLO.ASM example.

**COMP_SDX/COMP_BWD Demo for SDX Lib - COMPMAIN.ASM**

This is a second example for writing code for SpartaDOS X and BW-DOS using SDX Lib. This is an optimized version of the MY_COMP.ASM example. Instead of reading files to compare byte by byte using FGETC this example uses FREAD and buffers for files to compare.

# Structure of Source Code

The source text is line-oriented. Each line can contain one statement. There may also be blank lines.

Comment lines begin with * or ; for compatibility with MADS cross assembler. Everything after */; up to the end of the line is ignored by the assembler. Example:

```
* This is a comment!
; This is also a comment!
```

Comments can also follow an instruction. There they do not have to be introduced with *. The assembler ignores all characters after an instruction up to the end of the line. An instruction line is structured as follows:

```
[label] opcode [operand] [comment]
```

Example with a label and a comment:

```
start   ldx #$08 set X register
```

Labels cannot have the same name like operations because if a line starts with a word that is like an operation then these word is interpreted as operation.
Example:

```
inc   ldx #$08 set X register
```

In these example the word inc is interpreted as operation inc and ldx as operand for the operation inc. The remaining line is handled as comment. Usually this will lead to an error like label ldx is not declared.

Example of source line without label or comment:

```
            ldy #$10
```

To use the low or high byte of a value, the operators < and > must be used. Example:

```
            lda <my_address
            ldy >my_address
```

A special feature compared to other 6502 assemblers are the operations that can be used both with the accumulator and with other types of addressing. In this case the @ sign must be used for the accumulator.

Example operation `ASL` (Arithmetic Shift Left) at address $3000:

```
            asl $3000
```

Example operation `ASL` at accumulator:

```
            asl @
```

# The Assembly

The assembler processes the source code line by line and in the final pass the object file or executable file is written accordingly. None of the files are kept in memory, i.e. the maximum size of the files that can be processed or created is only limited by the file system. The assembler only keeps the internal symbol table in memory.

The source text is run through at least twice. After each run, the number of identifiers that could not yet be resolved is listed. Sometimes more than two runs have to be carried out, e.g. according to the following definition:

```
a   equ b
b   equ c
c   equ 100
```

In the first pass c is determined, in the second by b and in the third by a. Care should be taken to avoid something like this, as this increases the time for the assembly unnecessarily (due to the number of runs). The following definition can be processed in two passes:

```
c   equ 100
b   equ c
a   equ b
```

The assembler supports a maximum of 8 passes.

The assembler supports includes. The maximum depth of nesting of includes depends on how many files can be opened at the same time. At least two files are opened, for the source and the object file, for each further nesting of includes, a further source file is opened. The number of files that are open at the same time can be configured with the SPARTA driver in CONFIG.SYS. The standard value of SpartaDOS X is 5, i.e. a maximum depth of nesting of 3 is possible.

In the following it is described which expressions can be used and which pseudo-instructions (instructions which do not represent 6502 instructions, but rather are used to control the creation of the object file or the processing of expressions and values).

# Expressions

Text expressions are defined by quotation marks, 'or “. To display quotation marks in a text expression, two consecutive ones must be used, e.g .:

```
'Windows ''95'
```

Numbers in expressions:

- Binary numbers start with %, e.g.:

```
%101
```

- Hexadecimal numbers start with $, e.g.:

```
$FDAACDE
```

- Decimal numbers, e.g.:

```
23454
```

The maximum length of numbers is 4 Bytes.

Characters in ATASCII, e.g.:

`‘A'`, or `''''` for ‘ character

Character in internal display code (like in screen memory), e.g.:

`"I"` or `""""` for “ character

The length of text expressions is always 1 Byte and the value of the first character is taken, e.g.:

`'A'` and `'ABC'` return the same value of character A.

The following operators are supported:

`&` – Operator AND
`|` – Operator ORA
`^` – Operator EOR
`*` – Multiplication
`/` – Division
`+` – Addition
`–` – Subtraction

Sequences of operators are evaluated according to the above-mentioned priorities (AND highest, subtraction lowest). It is always calculated with 4 bytes. A maximum of 32 operators can be used per expression. The priority can be changed within an expression with the square brackets, e.g .:

```
[[4+5]*[256/21]]&$FF
```

There are two other special characters. * and ! return current program counter. * can only be used at the beginning of an expression, while ! can be used in general e.g .:

```
*+4
!-22
$2300+!
```

# Pseudo Instructions

All supported pseudo instructions are described below. At the end of the section, the pseudo instructions are described that are only supported for compatibility with Quick Assembler. These are not required if new programs are developed.

---

### blk

This statement declares a new program block. The total number of blocks in a program is limited to 256. The following block types are supported:

---

### blk n[one] a

A block without a header is declared. The program counter is set to address `a`.

---

### blk d[os] a

Declares an Atari DOS block with $FFFF header or without a header if the program counter is already at `a`.

---

### blk s[parta] a

Declares a SpartaDOS X block with fixed load address with header $FFFA, the program counter is set to `a`.

---

### blk r[eloc] m[ain]|e[xtended]

Declares a relocatable SpartaDOS X block with header $ FFFE in main (`m[ain]`) or extended memory (`e[xtended]`).

---

### blk e[mpty] a m[ain]|e[xtended]

Declares a relocatable SpartaDOS X block with header $FFFE to reserve bytes in main (`m[ain]`) or extended memory (`e[xtended]`). The program counter is immediately increased by a.

---

### blk u[pdate] s[ymbols]

Generates a SpartaDOS X block with header $FFFB, which updates the symbol addresses in Sparta or Reloc blocks.

## blk u[pdate] a[ddresses]

Generates a SpartaDOS X block with header $FFFD, for updating addresses in Reloc blocks.

## blk u[pdate] n[ew] a text

Generates a SpartaDOS X block with header $ FFFC, which declares a new symbol in a Reloc block with the address a. If the symbol name is preceded by an @ and the address comes from main memory, such a symbol can be called up via COMMAND.COM.

## end

End of the source or include file. The processing of the source file is aborted. If the source was included using statement `icl` then the source processing is continued after the statement `icl.`

## label equ a

Assigns a value to a name. The name can be a maximum of 240.

Example:

```
RUNAD equ $02e0
```

## [label] dta x(expr)

Generates data, where x can be defined as follows:

- `b(expr)` – `one` Byte
- `a(expr)` – `two` Bytes, Address
- `v(expr)` – `two` Bytes, Address, is updated in Reloc blocks, for all others blocks like `a(expr)`
- `e(expr)` – `three` Bytes
- `f(expr)` – `four` Bytes
- `g(expr)` – `four` Bytes in reverse order
- `l(expr)` – `lower` Byte
- `h(expr)` – `higher` Byte
- `c'ATASCII'` – Text in ATASCII
- `d'INTERN'` – Text in internal display code

In addition, numeric data can be specified as expressions separated by commas, e.g .:

```
myDat dta e(0,15000,$FFAACC)
      dta b($FF),a($C000)
```

## icl source[.ext]

Inserts the source text of a file. The drive and path are taken from the current directory by default where the assembler was started, e.g.:

```
      icl 'FAMAIN.ASM'
```

The maximum length of a source name is 32 ($20) characters including the end of line character.

The maximum of nested includes depends on the maximum number of open files allowed by SpartaDOS X / BW-DOS. There are always two open files during compilation, the source and the object file. Every nesting level added one open file. In SpartaDOS X you can configure the number of maximum simultaneously open files with a parameter of the device SPARTA. In BW-DOS the number of maximum open files is fixed with 5 simultaneously open files.

## ins source[.ext]

Inserts a binary file and accordingly increases the program counter. The drive and path are taken from the current directory by default where the assembler was started, e.g.:

```
      ins 'PICDATA.DAT'
```

The same rules for length of source name and nested includes apply, see command `icl`.

## label smb text

Declaration of the use of a SpartaDOS X symbol. After using blk update symbols, the assembler generates a block that automatically updates the address of the symbols used in the program, e.g.:

```
pf    smb 'PRINTF'
      jsr pf
```

The assembler inserts the correct symbol address after jsr.

This declaration is not transitive, i.e. the following example leads to an assembly error:

```
cm    smb 'COMTAB'
wp    equ cm-1.    (Error !)
      sta wp
```

Instead, you have to do the following:

```
cm    smb 'COMTAB'
      sta cm-1     (ok !)
```

All symbol declarations must first be defined in the program, i.e. before all other declarations and before the actual program.

# Supported Quick Assembler Instruction

These instruction only exists for compatibility reason. They do not provide extra functionality and should not be used in general.

---

## `opt b`

Parameters for the assembly, only for the compatibility with Quick Assembler, whereby the bits of b have the following meaning:

Bits 0-1 output of source
- `00` – none
- `01` – only at error
- `10` – complete source

Bits 6-7 Objekttyp
- `11` – without header,
- `01,10` – with Atari DOS header

Example for Atari DOS with output of the entire source code:

```
opt %01000010
```

---

## `org a`

Defines a new block and creates a header as defined with **opt** and sets the program counter to **a**. This instruction is only necessary for compatibility with Quick Assembler, for blocks see pseudo instruction **blk**.

---

## `lst all|bad|not`

Option for the output of the source text during assembly:
- `not` – no output
- `all` – output of complete source
- `bad` – output of lines with error

# Blocks

The most important new feature in SpartaDOS X is the ability to write programs that can be easily moved. Since the 6502 processor has no relative addressing (with the exception of brief conditional jumps), the ICD programmers used special processes for loading program blocks. The entire process always loads a relocatable block from MEMLO. Then MEMLO is incremented by the length of the block and then a special block is loaded to update addresses. All addresses in the program block are zero-based. So it is enough to add the MEMLO value to get the correct address. Which addresses should be updated and which not? There is a special block for this which contains coded pointers to these addresses. After the Reloc block or blocks, a `blk update addresses` must be executed so that the program can be executed. It is also necessary after Sparta blocks, where instructions (or vectors) refer to Reloc or Empty blocks.

Another innovation is the introduction of symbols. Some SpartaDOS X service routines are defined by names. These names are always 8 letters long (as are file names). Instead of vector or jump arrays (as in the Atari operating system), symbols defined with the pseudo instruction smb are used. After reading a block or program blocks, SpartaDOS X loads the block to update the symbols and exchanges addresses in the program in a similar way to Reloc blocks. Symbols can be used in Reloc and Sparta blocks.

The programmer can define his own symbols to replace Sparta DOS X symbols, or entirely new symbols that can be used by other programs. This is done using the update new block. New symbols must be implemented in a Reloc block.

The number of relocatable blocks (Reloc and Empty blocks) is limited to 7 by SpartaDOS X.

Blocks can be combined to form chains, e.g .:

```
blk sparta $600
...

blk reloc main
...

blk empty $100 main
...

blk reloc extended
...

blk empty $200 extended
```

This means that instructions in these blocks can apply to all blocks in the chain. This chain is not destroyed by address or symbol updates, but only by new symbol definitions or another type of block (e.g. DOS).

Such a chain only makes sense if all blocks are loaded in the same memory type (main or extended) or if the program changes the memory accordingly.

Instructions and vectors in Reloc and Empty blocks should not refer to Sparta blocks. This can lead to an error if the program is loaded with the SpartaDOS X instruction LOAD and is only used later. While Reloc and Empty blocks are safe, it is not certain what is in memory where the last Sparta block was.

It is just as dangerous to reference in Reloc and Empty blocks in Sparta blocks (reason as in section above). However, when installing overlays (.sys files) with INSTALL, this is sometimes required and therefore allowed. A Sparta block can also be initialized (via INITAD $2E2).

Address collisions can occur between Sparta, Reloc and Empty blocks. Fast Assembler recognizes references to other blocks on the basis of the addresses, whereby addresses from $1000 are assumed for Reloc and empty blocks. When shuffling these blocks, make sure that Sparta blocks are below $1000 (e.g. $600) or above the last movable block. Usually $4000 is enough. This error is not recognized by the assembler.

# Symbol Table

The symbol table can be dumped using the option /Y. The resulting file contains one line per symbol table entry. The structure of the line depends of the kind of the symbol table entry. The different kinds of symbol table entries are described below. All numbers are hexadecimal.

---

### SRC number name

Describes a source file. All input files are numbered starting from 00 in the order of their processing. The source file provided at the command file has the number 00.

---

### OBJ number name

Describes the destination (object) file. There is only one destination file that always has the number 00.

---

### EQU kind vale name

Describes an equate/label. The value is always 16bit. The following kinds are possible

- 01 - Zero page equate/label
- 21 - Absolute equate/label
- A1 - SparatDOS X symbol label

---

### BLK number relonumber kind start_address end_address

Describes a code block defined with pseudo command BLK. All blocks are numbered .

number - All blocks are numbered starting from 0.
relonumber - Relocatable blocks are numbered from 1.

The following kinds are possible

- FF - Atari DOS block
- FE - SpartaDOS X block
- 0x - Relocatable SpartaDOS X block
- 8x - Relocatable Empty SpartaDOS X block
- x0 - Relocatable block in main memory
- x2 - Relocatable block in extended memory

The start_address points to the first byte of the block. The and_address points to the first byte behind the en of the block

---

### SMB next name

Describes a used SpartaDOS X symbol defined with SMB. Next is a 16bit identifier for the double link list of references (see SMR).

## SMR prev next flag1 flag2 flag3 address

Describes a reference to aSpartaDOS X symbol defined with SMB. All references are double linked in the order of their processing. The first reference points with pre to the definition (see SMB). The last reference has next to be set to 0000.

# SDX Lib

The SDX Lib provides a subset of the SpartaDOS X function library for BW-DOS. The SDX Lib does not provide relocatable executable for BW-DOS. The SDX Lib consists of different parts defining groups of supported SpartaDOS X functions. These different parts of the SDX Lib are described later. Let's start with an example.

The following code uses the SpartaDOS X functions PRINTF and GETS:

```
* print question
        jsr PRINTF
        dta c'What''s your name?'
        dta b($9B,$00)
* ask for name
        lda namev
        ldx namev+1
        ldy #nam_siz
        jsr GETS
* print Hi
        jsr PRINTF
        dta c'Hi %s !'
        dta b($9b,$00)
        dta v(name)
        rts

namev   dta v(name)
```

This is a modified HELLO.ASM example. If you put this code in an include HI_MAIN.ICL then the frame program for SpartaDOS X would look like this:

```
* Example for FA SpartaDOS X
* Hi from Atari

PRINTF  smb 'PRINTF'
GETS    smb 'GETS'

nam_siz equ $40

        blk reloc main
        icl 'HI_MAIN.ICL'

name    equ *
        blk empty nam_siz main

        blk update addresses
        blk update symbols

        end
```

If you compile this with FastAssembler the result is a SpartaDOS X executable. Now you can use the code include HI_MAIN.ICL and a frame program using SDX Lib to compile an Atari DOS executable that runs on BW-DOS.

The frame program with SDX Lib for BW-DOS HI_BWD.ASM would look like this:

```
* Example for FA BW-DOS
* Hi from Atari

nam_siz equ $40

        blk dos $3000
        jsr CHKBW14
        icl 'HI_MAIN.ICL'
* SDX lib for BW-DOS
        icl '<SDXLIB>SDXCOMTA.ICL'
        icl '<SDXLIB>SDXSTD.ICL'
        icl '<SDXLIB>SDXFAIL.ICL'

name    equ *

        end
```

If you compile this with FastAssembler the result is an Atari DOS executable using SDX Lib (located in  a sub directory SDXLIB one sub director up) that runs under BW-DOS. The SDX Lib part SDXSTD provides the functions PRINTF and GETS. The part SDXFAIL is required by the part SDXSTD for error handling.

For both executable the same code is used just differently compiled.

Another more complex example is COMP_SDX.ASM, COMP_BWD.ASM, and COMPMAIN.ICL.

The next chapters describe the different parts of the SDX Lib. It is explained what functions with what limitations are supported. Please refer to the “SpartaDOS X Programming Guide” (https://sdx.atari8.info/index.php?show=en_docs) for a detailed description of the functions.

---

## SDXCOMTA

This part provides parts of the COMTAB of SpartaDOS X. It is required by SDXPARAM, SDXFILE, and SDXARITH. The following symbols are supported:

**COMTAB**
This is a symbol for an extra COMTAB for SDX Lib only. It is not a symbol for the COMTAB of BW-DOS. It is only used with SDX Lib and the offsets defined with CT_*.

**CT_COMFNAM**
Offset in the COMTAB of SDX Lib for the parameter buffer. The buffer is 32 ($20) bytes/characters long. It is used by SDXFAIL, and SDXPARAM.

**CT_TRAILS**
Offset in the COMTAB of SDX Lib containing the length of the current parameter in COMTAB+CT_COMFNAM. It is used by SDXPARAM.

**CT_VAR32_1|2|3**
Offset in the COMTAB of SDX Lib containing the three 32bit variables for MUL_32 and DIV_32 (see SDXARITH). These are the equivalents to COMTAB+$FF/$103/$107 in SpartaDOS X.

**FILE_P**
Symbol for address of file name used by FOPEN (see SDXFILE). Set to COMTAB+CT_COMFNAM by U_GETPAR/ATR (see SDXPARAM).

---

## SDXFAIL

This part provides the error handling. It requires SDXCOMTA. It supports the following functions:

**U_FAIL**
Expects the error code in register A and exits the program with error message

**U_SFAIL**
Expects the address of a sub routine in register AX. This routine is called in case of an error before the program is aborted.

**U_XFAIL**
Removes a sub routine that was set with U_SFAIL.

---

## SDXSTD

This part provides functions for standard output and input for the console. It requires SDXFAIL for error handling. It supports the following functions:

**PRINTF**
This is the implementation of a classical PRINTF function. The version of SDXLIB only supports the following format patterns:

- %% – print single %-character
- %c – print character from specified address
- %[*|1..8]s – print string from specified address, optional with length
- %[*|1..8]p – ditto, just address of string pointer is given instead
- %[*|2|4|6|8]x – print value from specified address as 16-bit hex number, optional with length
- %[*|1..8]b – print value from specified address as 8-bit dec number, optional with length
- %[*|1..8]d – print value from specified address as 16-bit dec number, optional with length
- %[*|1..8]e – print value from specified address as 24-bit dec number, optional with length

PRINTF does not change any register.

**TOUPPER**
Expects a character in register A and converts it to upper case if it is between a-z. Does not change register XY.

**PUTC**
Writes a single character in register A to the console. Does not change register AXY.

**PUTS**
Writes a text record terminated by EOL, address in register AX, to the console. Does not change register AXY.

**GETC**
Gets a byte, returned in register A, from the console.

**GETS**
Reads a text record from the console. As for PUTS, the address in registers AX.

## SDXPARAM

This part provides functions to parse parameter from the command line. It requires SDXFAIL and SDXCOMTA. The following functions are supported:

**U_GETPAR**
Copies the next parameter to COMTAB+CT_COMFNAM. If there are no more parameters it returns with Z=1 otherwise Z=0. Register X contains the length of the parameter and COMTAB+CT_TRAILS is set to this length.

**U_GETART**
SDX Lib does not support file attributes so this function is just an alias for U_GETPAR.

**U_GETNUM**
Checks wether the next parameter is a number. Returns Z=0 and the 16bit number in register AX. If the next parameter is not a number it does nothing and returns with Z=1.

**U_SLASH**
Checks for options e.g., /XYZ. Options must be passed as an array of two bytes containing a flag and the character of the option. The address of this array is passed in register AX and the length of the array in register Y. If options are found the flags are set accordingly with Z=0. If no options are found it returns Z=1. If an option occurs that is not contained in the option array the error 156 (Bad parameter) is set and U_FAIL is called.

## SDXFILE

This part provides file access. It requires SDXFAIL and SDXCOMTA. The following functions are supported:

**FHANDLE, FMODE, FATR1-2, FAUX1-5, SYSCALL**
Parameters for file operations. File attributes and extended memory are not supported so values in FATR* and SYSCALL are not set and not used.

**FOPEN**
Opens file with name pointed by FILE_P and returns handle in FHANDLE. FATR* and SYSCALL are ignored because SDX Lib does not support file attributes and extended memory. If an error occurs then U_FAIL is called.

This implementation returns IOCB numbers e.g. $10 etc. It uses the channels 1-3 and 6-7 (exactly in this order). 4 and 5 are not used because they are used by BW-DOS for batch processing and hard copy. Channels 6-7 are also used by BASIC for graphic, disk etc. So if your Programm should be able to run within BASIC or should be called directly form BASIC via USR() then you must not use more than 3 open files at a time.

**FCLOSE**
Closes a file identified by FHANDLE.

**FCLOSEAL**
Closes all open files.

**FREAD**
Reads binary block from a file identified by FHANDLE. Buffer address is passed in FAUX1/2 the length in FAUX4/5. If EOF is reached it returns with N=1. FAUX4/5 are set to the actually read length. If an error occurs then U_FAIL is called.

**FWRITE**
Writes binary block to a file identified by FHANDLE. Buffer address is passed in FAUX1-2 the length in FAUX4-5. If an error occurs then U_FAIL is called.

**FGETC**
Reads a single byte into register A from a file identified by FHANDLE. If EOF is reached it returns with N=1. If an error occurs then U_FAIL is called.

**FPUTC**
Writes a single byte in register A to file identified by FHANDLE. If an error occurs then U_FAIL is called.

**FTELL**
Returns the current position in file identified by FHANDLE and writes it as 24bit number to FAUX1-3.

**FSEEK**
Sets the current position in file identified by FHANDLE to a 24bit number from FAUX1-3.

**FILELENG**
Returns the length of a file identified by FHANDLE and writes it as 24bit number to FAUX1-3.

---

## SDXARITH

This part provides a 32bit multiplication and division. It requires SDXCOMTA. The following functions are supported:

**MUL_32**
Executes a 32bit multiplication from COMTAB+CT_VAR32_1 and COMTAB+CT_VAR32_2. The result is stored in COMTAB+CT_VAR32_3. If an overflow occurs then C=1.

**DIV_32**
Executes a 32bit division from COMTAB+CT_VAR32_1 by COMTAB+CT_VAR32_2. The result is stored in COMTAB+CT_VAR32_3. If division by 0 occurs then C=1 and COMTAB+CT_VAR32_3 is set to 0.

# Error Messages

Error messages consist of two line. One line with source code information and one line with a short text.

Example 1:

```
FA TEST1

Error at D1:TEST1.ASM:11:3E40
  Too big number
Pass 0 Lines 37
Failed
```

This is an error message in source TEST1.ASM at line 11 with the message “Too big number”.

If an error occurs in a DOS operation then first the source information with the message “DOS error” is shown and followed by the DOS message.

Example 2:

```
FA TEST

Error at D1:TEST2.ASM:6:3D40
  DOS error
Failed

File TEST2.ICL

Error 170
```

This is the error message file TEST2.ICL could not be found in source TEST2.ASM at line 6 e.g., in line 6 is an include instruction for file TEST2.ICL but the file does not exists.

If option /E is set then a compile error will be converted to a DOS error. The compilation is aborted at the end of the current pass. If this command is part of an executed batch file then in BW-DOS the batch processing is also aborted, in SpartaDOS X the variable STATUS is set and can be checked with IF ERROR.

Example 3:

```
FA TEST1 /E

Error at D1:TEST1.ASM:11:3E40
  Too big number
Pass 0 Lines 37
Failed

Error 228
```

This is an example if the first example is started with option /E.

# List of Error Messages

This list contains all error messages of compilation with the error number if the message is converted to an DOS error (see option /E).

`Undeclared label 224($E0)`
Undefined label or symbol or recursive labels.

`Label declared twice 225($E1)`
Label or symbol defined twice.

`Unexpected eol 226($E2)`
Unexpected line break means there should be at least one character or parameter in the line.

`Too many passes 227($E3)`
Too many passes when the number of passes exceeds 8.

`Too big number 228($E4)`
Number too large, the number exceeded the allowed 4 byte range.

`String error 229($E5)`
Text expression error, no closing quotation mark or empty phrase.

`Illegal symbol 230($E6)`
Illegal character in line.

`Branch to far 231($E7`
The relative jump is beyond the range of +127 and -128 bytes.

`Improper type 232($E8)`
Addressing type not allowed for the given command.

`Label missing 233($E9)`
Missing label, EQU and SMB pseudo commands must always be preceded by a label.

`Expression expected 234($EA)`
Expected expression, after a numerical an expression must follow.

`Too many blocks 235($EB)`
Too many blocks, the total number of blocks cannot exceed 256 and the number of RELOC and EMPTY blocks cannot exceed 7.

`Undefined or too long 236($EC)`
Undefined or too long label, when defining labels a label block must be defined before using them, and the result of the address expression must not be larger than 2 bytes.

`Improper block type 237($ED)`
Wrong type of block, e.g. UPDATE NEW block cannot follow DOS block.

`Overflow or invalid symbol 238($EE)`
The value of the expression is too large or it contains an invalid symbol.

`Parenthesis not balanced 239($EF)`
Missing parentheses.

`Too many operations 240($F0)`

Too many operations in expression, the number of operations is limited to 32.

`Unexpected symbol 241($F1)`
Unexpected symbol, symbols cannot be used in blocks other than SPARTA or RELOC and in block definitions.

`Internal error 242($F2)`
Internal error, please drop me an email with instructions how to reproduce, holgerjanz@abbuc.social

`Program counter overflow 243($F3)`
Overflow of the program counter during compiling of statement

`DOS error 244($F4)`
Error at DOS operation, see the following DOS error message

`Truncated line 245($F5)`
The source line has more than 255 characters or there is no end of line at the end of the last line of a source file. The compilation is immediately aborted.

# Changes and Fixes

This list contains all fixes and changes from version 1.7 to version 1.8.

- fixed parameter parsing issue that prevents Fast Assembler working with SpartaDOS X 4.47 to 4.49d
- fixed multiplication and division issue, because of wrong overflow handling these operations always throw error "Too big number”
- new pseudo command INS to insert binary files
- fix END issue, processing of source was not aborted but the source in buffer was further processed
- also the ; character can be used to indicate a comment line (like with character *)
- allow spaces in empty lines before the error unexpected end of line was raised if an empty line contains spaces
- improved position information for error messages
- add line number to source output
- add option /Y for dump of symbol table
- add option /S for summary, e.g. count of labels and blocks, and memory usage
- add option /E to convert error message to DOS error if compile fails, for better use in batch processing
- Add option /Q for quite mode without line progress output, for better use if console output is redirected e.g. file
- remove option /B
- rename option /A to /L
- condense info output
- improve memory management
- improve performance of pass 1 and up
- reduce number of max passes to 8
- update copyright message
- add tutorial with extended examples
- add XEDIT by FJC (thanx for the permission), now it is a complete development package