# The SBI format

Basic introduction to the SBI executable
format and its interpreter library

Gi@cky98

# Index

# Introduction

## What’s SBI?

SBI (**Small Bytecode Interpreter**) is an **interpreter library** that allows you to interpret bytecode on **any platform** that has a port of the **GCC compiler**.

## Some applications

A simple SBI application is costituited by:

- The **interpreter program** compiled with the user functions you want
- A **stream** where to read the program bytecode
- The **SBI program** on the stream

For example, as it’s initial purpose, SBI can be used to **run programs from an SD card on an AVR**, avoiding the need of reprogramming it. It can be useful to avoid frequent rewriting of the flash, that after some time can damage it.

Another usage could be an application or a game that needs to **execute code from a file** (e.g. a game **script**).

## Pros and cons

There are some **pros and** some **cons of** using **SBI** (or, more generally, a bytecode interpreter).

The **pros** are the following

- **No** need to **recompile** the main program
- (For MCUs) **No** need to **reprogram** the MCU

And the **cons** are

- **Low speed**
- You **don’t have all the hardware control** like in C or Assembly (but you can obtain a basic control of the hardware using user functions (see it under Usage of the library)

# The SBI bytecode

## Introduction to the bytecode

The concept is to **assign to each** processor **instruction a byte value** (e.g. 0x48 or 0xB4) and to put the instruction codes one after the other in a file, so the interpreter can read it and execute the instructions as they are wrote. Besides, there's the **need to pass parameters to the instructions**, so a *"complete instruction"*, can take more than 1 byte. For example, the incr instruction (which increments the value of a variable), takes 2 bytes: 1 byte for the instruction code (0x20) and 1 byte for the first parameter: the variable number. **There are also long and complex instructions structures**, for example the structure of the cmpjump instruction. The complete instruction takes 8 bytes, respectively:

| Byte | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| Description | 0x42 | Var 1 type | Var 1 value | Var 2 type | Var 2 value | Jump v. type | Jump v. value | Jump mode |

## SBI executable organization

The **structure of a SBI program** is very simple and can be **represented by** the following **table**:

| **Block size (bytes)** | **Description** |
|---|---|
| 2 | Header |
| 1 | Label section identifier |
| *variable* | Labels data |
| 1 | Section separator |
| 1 | Interrupts section identifier |
| *variable* | Interrupts data |
| 1 | Section separator |
| *variable* | Program bytecode |
| 2 | Footer |

As you can see, the structure is very simple and keeps the programs size very small.

There are some special bytes of SBI, such as HEADER_0 and HEADER_1, SEPARATOR, LABELSECTION, etc. They are in the following table:

| Name (#define) | Description | Value (HEX) |
|---|---|---|
| HEADER_0 | Header lower byte | 0xAA |
| HEADER_1 | Header higher byte | 0x(version)B - 0x4B |
| LABELSECTION | Labels section identifier | 0xA3 |
| INTERRUPTSECTION | Interrupts section identifier | 0xB3 |
| SEPARATOR | Section separator byte | 0xB7 |
| FOOTER_0 | Footer lower byte | 0x3A |
| FOOTER_1 | Footer higher byte | 0xF0 |

## Instruction set

The following **table** contains informations **about the instruction set of SBI**, including description, parameters number and byte size of the instruction.

| Name (SASM) | ID | Params n | Params size | Size | Description |
|---|---|---|---|---|---|
| assign | 0x01 | 2 | 2 bytes | 3 | Assign value to variable |
| move | 0x02 | 2 | 2 bytes | 3 | Copy variable value into another |
| add | 0x10 | 3 | 5 bytes | 6 | Assign sum to variable |
| sub | 0x11 | 3 | 5 bytes | 6 | Assign difference to variable |
| mul | 0x12 | 3 | 5 bytes | 6 | Assign product to variable |
| div | 0x13 | 3 | 5 bytes | 6 | Assign quotient to variable |
| incr | 0x20 | 1 | 1 byte | 2 | Increment a variable (+1) |
| decr | 0x21 | 1 | 1 byte | 2 | Decrement a variable (-1) |
| inv | 0x22 | 1 | 1 byte | 2 | Invert the value of a variable |

| | | | | | |
|---|---|---|---|---|---|
| tob | 0x23 | 1 | 1 byte | 2 | Converts a variable in 0 or 1 |
| cmp | 0x30 | 3 | 5 bytes | 6 | Compares two values (=) |
| high | 0x31 | 3 | 5 bytes | 6 | Compares two values (>) |
| low | 0x32 | 3 | 5 bytes | 6 | Compares two values (<) |
| jump | 0x41 | 2 | 3 bytes | 4 | Jump to an address |
| cmpjump | 0x42 | 4 | 7 bytes | 8 | Jump to an address if *cmp = 1* |
| ret | 0x43 | - | - | 1 | Return from a subroutine |
| debug | 0x50 | 1 | 2 bytes | 3 | Print a value in the debug stream |
| error | 0x52 | 1 | 2 bytes | 3 | Print an error and exits |
| sint | 0x60 | 1 | 2 bytes | 3 | Select user function to use |
| int | 0x61 | 1 - 8 | 16 bytes | 17 | Use selected user function |
| exit | 0xFF | - | - | 1 | Exit the program |

**Note:**
If you want **more informations about the instructions** or if you want to start programming in **SASM**, please **read the** SASM User Guide. Here you will get more informations about the parameters and the work of each instruction.

# The SBI library interpreter

## Introduction to the library

The **library** consists of **3 files**: a C file, its header and an header for the user functions.

- *sbi.c*
- *sbi.h*
- *funclib.h*

You need to **link together** the object file generated by sbi.c (**sbi.o**) **with the other** project **object files**.

To see a valid example, please visit the Examples section.

## Defines

In the *sbi.h* header file you can find a lot of **#define** statements. Most of them are used only by the interpreter core (sbi.c), but there are **some** of them that **the user should know**.

**#define VARIABLESNUM:**
**Changing the value of this #define**, you can **set the maximum number of variables** to reserve to the program. For example, if you want **more free RAM** and your program doesn't use a lot of variables, **you can reduce this number** to - for example - 16 (only 16 bytes of RAM will be used). Note that if you put 16 as maximum number, in the program you can use only variables from t0 to t15 (not t16, t20, etc.).

**#define USERFUNCTIONSN:**
**Changing the value of this #define**, you can **set the maximum number of user functions** to load. As the number of the variables, if you want **more free RAM** and your program doesn't use a lot of user functions, **you can reduce this number** to - for example - 8 (16 bytes of RAM will be used, 2 bytes for each pointer - user function).

**#define RETURNADDRESSESN:**
This value is used to set the **size of the return addresses array**. This array is used to **store the return addresses of the subroutines** when you use the *ret* instruction. If the programs that the interpreter will load doesn't contain a lot of concatenated subroutines, you can reduce this number to - for example - 6 (12 bytes of RAM will be used, 2 bytes for each address).

## Functions

The **library** gives to the user the following **functions**:

- `void _sbi_init(void)`
- `int _sbi_begin(void)`
- `int _sbi_run(void)`

- `byte _getval(byte type, byte val)`
- `byte _setval(byte type, byte num, byte val)`

- `void _interrupt(byte id)`

**Note:**

```
typedef unsigned char byte;
```

Follows the **description** of each function:

| `void _sbi_init(void)` |
|---|
| Initializes the library. Call this when the interpreter starts. |

| `int _sbi_begin(void)` |
|---|
| Begins the interpreter. Call this every time you need to run a program and be sure to assign pointers to _getfch, _setfpos and _getfpos.<br><br>Returns:<br>0: No errors<br>1: No function pointers for _getfch, _setfpos and _getfpos<br>2: Old version of executable format<br>3: Invalid program file |

| `int _sbi_run(void)` |
|---|
| Runs an instruction - only one. |

Returns:
0: No errors
1: Reached end (no exit found)
2: Program exited
3: Wrong instruction code
4: Can't understand byte
5: User error

```
byte _getval(byte type, byte val)
```

To use into user functions. Returns the value of a parameter specifying its type (value / variable id). For example if you have the array of the parameters (byte b[16]) and you want to get the current value of the third parameter, use

```
byte value = _getval(b[4], b[5]);
```

```
byte _setval(byte type, byte num, byte val)
```

To use into user functions. Sets the value of a variable specifying its type (value / variable id), its value (used if the type is variable id, else return error), and the value to assign. For example if you have the array of the parameters (byte b[16]) and you want to set the value of the second parameter to 18, use

```
_setval(b[2], b[3], 18);
```

Returns:
0: No errors
1: The specified parameter is not a variable but a value

```
void _interrupt(byte id)
```

Causes a program interrupt and sets the program counter to the selected interrupt routine address.

## Notes

**Before** you call `_sbi_begin`, you need to **set some pointers for** `_getfch`, `_setfpos` and `_getfpos` **functions.** This is an **example**:

```
byte getfch(void)
{
	// ...
}

void setfpos(int p)
{
	// ...
}

int getfpos(void)
{
	// ...
}

int main(void)
{
	// ...

	_sbi_init();

	_getfch=&getfch;
	_setfpos=&setfpos;
	_getfpos=&getfpos;

	int ret = _sbi_begin();

	// ...

	return 0;
}
```

For **more informations** see the Examples section.

# Usage of the library

## Basic informations

**First** of all you need to **initialize the library** using `_sbi_init`.
After you need to **assign function pointers** for the **stream access functions** (`_getfch`, `_setfpos` and `_getfpos`). When you are sure that your stream is ready, you have to call `_sbi_begin` to **initialize the program** and, eventually, get errors about the format/version of the executable.
If the program initialization doesn't returned any error, you can proceed **executing the instructions** by **calling `_sbi_run` for each instruction**. A good way to do this is to put `_sbi_run` in a **loop** that **exits only when** `_sbi_run() > 0` (**program exited or some errors**).

## User functions

To allow SBI programs to access your host functions (e.g. MessageBox on Windows or I/O on AVRs), you can use user functions. They are assigned in compilation-time from the file *funclib.h.*

A user function is declared as:

```
int myfunc(byte b[16]);
```

Note that the interpreter doesn't care about what a user function returns.

The structure of *funclib.h* is the following:

```
// Debug printing function
void debugn(byte n)
{
	// ...
}


// Error printing function
void errorn(byte n)
{
	// ...
}
```

```
// User function(s)
int myfunc(byte b[16])
{
	// ...
}

// User function(s) initialization routine
void _inituserfunc(void)
{
	_sbifuncs[0] = &myfunc;
	// ...
}
```

`debugn` and `errorn` functions are required to write to the debug and error stream (debug and error SBI instructions). After you have configured them, you can add your user functions (like `myfunc`). Remember to add your function initialization statement to the `_inituserfunc` routine, as the one in the code above.

# Examples

## Simple SBI bytecode

For example, the following SASM program

```
assign _t0 0

label 0
     debug _t0
     incr _t0
     low _t0 10 _t1
     cmpjump _t1 1 0 0

exit
```

Compiles into the following SBI bytecode:

| Address | Value | Description |
|---|---|---|
| 0x00 | 0xAA | **HEADER_0** |
| 0x01 | 0x4B | **HEADER_1** (also version identifier - in this case 4) |
| 0x02 | 0xA3 | **LABELSECTION** |
| 0x03 | 0x01 | Number of labels (1) |
| 0x04 | 0x0D | Label [1] address HIGH |
| 0x05 | 0x00 | Label [1] address LOW |
| 0x06 | 0xB7 | **SEPARATOR** |
| 0x07 | 0xB3 | **INTERRUPTSECTION** |
| 0x08 | 0x00 | Number of interrupt routines (0) |
| 0x09 | 0xB7 | **SEPARATOR** |
| 0x0A | 0x01 | ASSIGN |
| 0x0B | 0x00 | Variable number 0 |
| 0x0C | 0x00 | Value 0 |

| | | |
|---|---|---|
| 0x0D | 0x50 | DEBUG |
| 0x0E | 0x04 | Parameter type: variable number (0x04) |
| 0x0F | 0x00 | Variable number (0) |
| 0x10 | 0x20 | INCR |
| 0x11 | 0x00 | Variable number (0) |
| 0x12 | 0x32 | LOW |
| 0x13 | 0x04 | Parameter type: variable number (0x04) |
| 0x14 | 0x00 | Variable number (0) |
| 0x15 | 0xF4 | Parameter type: value (0xF4) |
| 0x16 | 0x0A | Value (10) |
| 0x17 | 0x01 | Output variable number (1) |
| 0x18 | 0x42 | CMPJUMP |
| 0x19 | 0x04 | Parameter (to compare) type: variable number (0x04) |
| 0x1A | 0x01 | Variable number (1) |
| 0x1B | 0xF4 | Parameter (to compare) type: value (0xF4) |
| 0x1C | 0x01 | Value (1) |
| 0x1D | 0xF4 | Parameter (label number) type: value (0xF4) |
| 0x1E | 0x00 | Value (0) -> Label 0 |
| 0x1F | 0x00 | Jump type (normal/subroutine): 0x00 -> 0 -> Normal |
| 0x20 | 0xFF | EXIT |
| 0x21 | 0x3A | **FOOTER_0** |
| 0x22 | 0xF0 | **FOOTER_1** |

## Simple SBI interpreter (Windows)

To create a basic SBI interpreter, you need to create an empty C project and import the sbi.c, sbi.h and funclib.h source files from the latest SBI package (see the *sbi* folder).

Now you need to create the main interpreter file and edit the funclib.h file to adapt the `debugn` and `errorn` functions for your platform.

This is the main.c file of the interpreter:

```
#include <stdio.h>
#include "sbi.h"

FILE* f;
int pos;

byte getfch(void)
	return fgetc(f);

void setfpos(int p)
	fseek (f, p, SEEK_SET);

int getfpos(void)
	return (int)ftell(f);

int main(int argc, char** argv)
{
	f = fopen("program.sbi", "rb");
	if (!f) { printf("Can't open \"program.sbi\"!\n"); return 1; }

	_getfch=&getfch; _setfpos=&setfpos; _getfpos=&getfpos;
	_sbi_init(); pos = 0;

	int ret = _sbi_begin();
	if (ret>0) { printf("Initialization error\n"); return 1; }

	while (ret==0)
	{
		ret = _sbi_run();
	}

	fclose(f);

	if (ret<2) return 0; else return 1;
}
```

Then change funclib.h to the following:

```
#include <stdio.h>

#ifndef _FUNCLIB
	#define _FUNCLIB

	void debugn(byte n)
		printf("DEBUG\t\t0x%02X\t\t%i\n", n, n);

	void errorn(byte n)
		printf("ERROR\t\t0x%02X\t\t%i\n", n, n);

	void _inituserfunc(void)
	{
		// No user functions
		return;
	}

#endif
```

To compile your work type the following commands on the command prompt:

```
gcc -c main.c -o main.o
gcc -c sbi.c -o sbi.o
gcc main.o sbi.o -o main.exe
```

Now (if you haven't done it before) compile the SASMC compiler from the source files provided with the SBI package and copy its executable file sasmc.exe into your project's directory.

Then create a simple program, such as the following:

```
assign _t0 0

label 0
	debug _t0
	incr _t0
	low _t0 10 _t1
	cmpjump _t1 1 0 0

exit
```

And save it to your project's directory with the name of program.sasm.

Then compile it to program.sbi:

```
sasmc -i program.sasm -o program.sbi -cl
```

Now you can run the executable compiled before (main.exe) to see your SBI program running.