AVR COFF (COMMON OBJECT FILE FORMAT) SPECIFICATION

Preliminary release

Revision History

Date	Version	Author	Remarks
01/05/98 1:51 PM	1.0	Jo Inge Lamo	First Draft

Introduction

AVR COFF is supported by AVR Studio v1.30 and above. The assembler and linker may create object files in AVR COFF. The format is implemented in AVR Studio to support third party vendors and their products. The format encourages modular programming and provides more powerful and flexible methods for managing code segments and target system memory.

Overview of the Document

This document describes the AVR COFF format and special cases which are supported by AVR Studio. The overall structure and the details of each section in the COFF is described.

Overall Structure

The table below show the overall structure of the object file.

File Header
Optional Information
Section 1 Header
…
Section n Header
Raw Data for Section 1
…
Raw Data for Section n
Relocation Information for Section 1
…
Relocation Information for Section n
Line Numbers for Section 1
…
Line Numbers for Section n
Symbol Table
String Table

The last four sections (relocation, line numbers, symbol table, and the string table) may be missing. Also, if there are no unresolved external references after linking, the relocation information is no longer needed and is absent. The string table is also absent if the source file does not contain any symbols with names longer than eight characters.

File Header

The file header has 20 bytes of information as shown in the table below.

Bytes	Description
0-1	Magic number (0xA12 for AVR COFF)
2-3	Number of sections
4-7	Time and date stamp indicating when the file was created, expressed as the number of elapsed seconds since 00:00:00 GMT, January 1, 1970
8-11	File pointer to the symbol table
12-15	Number of entries in the symbol table
16-17	Number of bytes in the optional header
18-19	Flags

The magic number specifies the target machine on which the object file is executable. For AVR COFF it is 0xA12. File pointer is the byte offset to the start of the symbol table from the beginning of the file. The flags describe the type of the object file, which are shown in the table below.

Flag	Meaning
0x0001	Relocation information stripped from the file
0x0002	The file has no unresolved external references
0x0004	Line numbers stripped from the file
0x0008	Local symbols stripped from the file

Section Headers

A section is the smallest portion of the object file that is relocated and treated as one separate and distinct entity. In the most common case, there are three sections named .text, .data, and .bss. .text section contains executable text. .data section contains initialised data. .bss section contains uninitialised data.

The information in the section header is described in the table below.

Bytes	Description
0-7	8-character null padded section name
8-11	Physical address of section
12-15	Virtual address of section
16-19	Section size in bytes
20-23	File pointer to raw data
24-27	File pointer to relocation entries
28-31	File pointer to line number entries
32-33	Number of relocation entries
34-35	Number of line number entries
36-39	Flags

The physical address is the address at which the section data should be loaded into memory. For linked executables, this is the absolute address within the program space. For unlinked objects, this address is relative to the object's address space (i.e. the first section is always at offset zero). The size of a section is padded to a multiple of 4 bytes. File pointers are byte offsets that can be used to locate the start of data, relocation, or line number entries for the section. The flags are described in the table below.

Flag	Meaning
0x0020	Section contains executable text (.text)
0x0040	Section contains initialised data (.data)
0x0080	Section contains uninitialised data (.bss)
0x0200	Comment section

Sections

The raw data for each section begins on a 4-byte boundary in the file.

Relocation Information

There is only one relocation entry for each relocatable reference in the text or data. The following table describes the relocation section contents.

Bytes	Description
0-3	Virtual address of reference

4-7	Symbol table index
8-9	Relocation type

The first 4 bytes of the entry are the virtual address of the text to which this entry applies. The next field is the index, counted from 0, of the symbol table entry that is being referenced. The type field indicated the type of relocation to be applied, as shown in the table below.

Flag	Meaning
0x0000	Reference is absolute; no relocation is necessary
0x0001	Direct
0x0002	PC-relative

Line Numbers

When the compiler is invoked with -g option, the compiler causes an entry in the object file for every source line where a breakpoint can be inserted. All line numbers in a section are grouped by function as shown in the table below.

Symbol index	0
physical address	line number
physical address	line number
•	•
•	•
•	•
symbol index	0
physical address	line number
physical address	line number

The first entry in a function grouping has line number 0 and has, in place of the physical address, an index into the symbol table for the entry containing the function name. Subsequent entries have actual line numbers and addresses of the text corresponding to the line numbers. The line number entries are relative to the beginning of the function and appear in increasing order of address.

Symbol table

The order of symbols in the symbol table is very important. Symbols appear in the sequence shown in the table below.

filename 1
function 1
local symbols for function 1
function 2
local symbols for function 2
…
statics for file 1
filename 2
function 1
local symbols for function 1
…
statics for file 2
…
defined global symbols
undefined global symbols

The symbol table consists of at least one fixed-length entry per symbol with some symbols followed by auxiliary entries of the same size. The entry for each symbol is a structure that holds the value, the type, and other information.

Special Symbols

The symbol table consists of some special symbols that are generated by the compiler, assembler and linker. These symbols are given in the following table.

Symbol	Meaning
.file	filename
.text	address of .text section
.data	address of .data section

.bss	address of .bss section
.bb	address of start of inner block
.eb	address of end of inner block
.bf	address of start of function
.ef	address of end of function
.target	pointer to the structure or union returned by a function
.*x*fake	dummy tag name for structure, union, or enumeration
.eos	end of members of structure, union, or enumeration
etext	next available address after the end of the output section .text
edata	next available address after the end of the output section .data
end	next available address after the end of the output section .bss

Six of these special symbols occur in pairs. The .bb and .eb symbols indicate the boundaries of inner blocks; a .bf and .ef pair brackets each function. An .*x*fake and .eos pair names and defines the limit of structures, unions, and enumerations that were not named. The .eos symbol also appears after named structures, unions, and enumerations.

When a structure, union, or enumeration has no tag name, the compiler invents a name to be used in the symbol table. The name chosen for the symbol table is .*x*fake, where *x* is an integer. If there are three unnamed structures, unions, or enumerations in the source, their tag names are .0fake, .1fake, and .2fake. Each of the special symbols has different information stored in the symbol table entry as well as the auxiliary entries.

The C language defines a block as a compound statement that begins and ends with braces "{" and "}". An inner block is a block that occurs within a function (which is also a block). For each inner block that has local symbols defined, a special symbol .bb is put in the symbol table immediately before the first local symbol of that block. Also a special symbol .eb is put in the symbol table immediately after the last local symbol of that block.

For each function, a special symbol .bf is put between the function name and the first local symbol of the function in the symbol table. Also a special symbol .ef is put immediately after the last local symbol of the function in the symbol table.

Symbol Table Entries

All symbols, regardless of storage class and type, have the same format for their entries in the symbol table. The symbol table entries each contain 18 bytes of

information. The meaning of each of the fields in the symbol table entry is described in the table below. It should be noted that indices for symbol table entries begin at 0 and count upward. Each auxiliary entry also counts as one symbol.

Bytes	Description
0-7	These 8 bytes have a symbol name or an index to a symbol in the string table.
8-11	Symbol value; storage class dependent
12-13	Section number of symbol
14-15	Basic and derived type specification
16	Storage class of symbol
17	Number of auxiliary entries

Symbol Name

The first 8 bytes in the symbol table entry are a union of a character array and two longs. If the symbol name is eight characters or less, the (null-padded) symbol name is stored there. If the symbol name is longer than eight characters, then the entire symbol name is stored in the string table. In this case, the 8 bytes contain two long integers, the first is zero, and the second is the offset (relative to the beginning of the string table) of the name in the string table. Since there can be no symbols with a null name, the zeroes on the first 4 bytes serve to distinguish a symbol table entry with an offset from one with a name in the first 8 bytes as shown in the table below.

Bytes	Description
0-7	8-character null-padded symbol name
0-3	Zero in this field indicates the name is in the string table
4-7	Offset of the name in the string table

Storage Class

The storage class field has one of the values described in the following table.

Mnemonic	Value	Storage Class
C_EFCN	-1	physical end of a function
C_NULL	0	-
C_AUTO	1	automatic variable
C_EXT	2	external symbol

C_STAT	3	static
C_REG	4	register variable
C_EXTDEF	5	external definition
C_LABEL	6	label
C_ULABEL	7	undefined label
C_MOS	8	member of structure
C_ARG	9	function argument
C_STRTAG	10	structure tag
C_MOU	11	member of union
C_UNTAG	12	union tag
C_TPDEF	13	type definition
C_USTATIC	14	uninitialised static
C_ENTAG	15	enumeration tag
C_MOE	16	member of enumeration
C_REGPARM	17	register parameter
C_FIELD	18	bit field
C_BLOCK	100	beginning and end of block
C_FCN	101	beginning and end of function
C_EOS	102	end of structure
C_FILE	103	file name

Some special symbols are restricted to certain storage classes only, and some storage classes are restricted to certain special symbols only.

Symbol value

The meaning of a symbol value depends on its storage class. This relationship is given in the table below.

Storage Class	Meaning of Value
C_AUTO	stack offset in bytes
C_EXT	relocatable address
C_STAT	relocatable address
C_REG	register number
C_LABEL	relocatable address
C_MOS	offset in bytes
C_ARG	stack offset in bytes
C_MOE	enumeration value
C_REGPARM	register number
C_FIELD	bit displacement

C_BLOCK	relocatable address
C_FCN	relocatable address
C_EOS	size

If a symbol has storage class C_FILE, the value of that symbol equals the symbol table entry index of the next .file symbol. That is, the .file entries form a one-way linked list in the symbol table. If there are no more .file entries in the symbol table, the value of the symbol is the index of the first global symbol.

Relocatable symbols have a value equal to the virtual address of that symbol. When the section is relocated by the link editor, the value of these symbols changes.

Section Number

The meaning of the section number field is summarised in the table below.

Number	Meaning
-2	Special symbolic debugging symbol
-1	Absolute symbol
0	Undefined external symbol
1 - 077777	Section number where symbol is defined

The special symbol -2 marks symbolic debugging symbols including structure names, union names, enumeration tag names, typedefs, and the name of the file. Absolute symbols include automatic and register variables, function arguments and .eos symbols.

Type specification

The type field in the symbol table entry contains information about the basic and derived type for the symbol. This information is generated by the C compilation system only if the -g option is used. Each symbol has exactly one basic or fundamental type but can have more than one derived type. The format of the 16-bit type entry is:

d6	d5	d4	d3	d2	d1	typ

Bits 0 through 3, called typ, indicate one of the fundamental types given in the table below.

typ	Meaning
0	type not assigned

1	function argument
2	character
3	short integer
4	integer
5	long integer
6	floating point
7	double word
8	structure
9	union
10	enumeration
11	member of enumeration
12	unsigned character
13	unsigned short
14	unsigned integer
15	unsigned long

Bits 4 through 15 are arranged as six 2-bit fields marked d1 through d6. These d fields represent levels of the derived types given in the table below.

Value	Meaning
0	no derived type
1	pointer
2	function
3	array

Auxiliary Table Entries

An auxiliary table entry of a symbol contains the same number of bytes as the symbol table entry. However, unlike symbol table entries, the format of an auxiliary table entry of a symbol depends on its type and storage class.

File Names

Each of the auxiliary table entries for a file name contains a 14-character file name in bytes 0 through 13. The remaining bytes are zeroes.

Sections

The auxiliary table entry for sections have the format as shown below.

Bytes	Description
0-3	section length

4-5	number of relocation entries
6-7	number of line numbers
8-17	unused (padded with zeroes)

Tag Names

The auxiliary table entries for tag names have the format shown below.

Bytes	Description
0-5	unused (padded with zeroes)
6-7	size of the structure/union/enumeration
8-11	unused (padded with zeroes)
12-15	index of next entry beyond this structure/union/enumeration
16-17	unused (padded with zeroes)

End of Structures

The auxiliary table entries for the end of structures have the format shown below.

Bytes	Description
0-3	tag index
4-5	unused (padded with zeroes)
6-7	size of the structure/union/enumeration
8-17	unused (padded with zeroes)

Functions

The auxiliary table entries for functions have the format shown below.

Bytes	Description
0-3	tag index
4-7	size of function in bytes
8-11	file pointer to line number

12-15	index of next entry beyond this point
16-17	unused (padded with zeroes)

Arrays

The auxiliary table entries for arrays have the format shown below.

Bytes	Description
0-3	tag index
4-5	line number of declaration
6-7	size of array
8-9	first dimension
10-11	second dimension
12-13	third dimension
14-15	fourth dimension
16-17	unused (padded with zeroes)

End of Blocks and Functions

The auxiliary table entries for the end of blocks and functions have the format shown in the table below.

Bytes	Description
0-3	unused (padded with zeroes)
4-5	C-source line number
6-17	unused (padded with zeroes)

Beginning of Blocks and Functions

The auxiliary table entries for the beginning of blocks and functions have the format shown in the table below.

Bytes	Description
0-3	unused (padded with zeroes)
4-5	C-source line number
6-11	unused (padded with zeroes)

12-15	index of next entry past this block
16-17	unused (padded with zeroes)

Names related to Structures, Unions, and Enumerations

The auxiliary table entries for structure, union, and enumeration symbols have the following format shown in the table below.

Bytes	Description
0-3	tag index
4-5	unused (padded with zeroes)
6-7	size of the structure/union/enumeration
8-17	unused (padded with zeroes)

Aggregates defined by typedef may or may not have auxiliary table entries.

String Table

Symbol table names longer than 8 characters are stored contiguously in the string table with each symbol name delimited by a null byte. The first 4 bytes of the string table are the size of the string table in bytes; offsets into the string table, therefore, are greater than or equal to 4.

Remarks

1. Optional headers will be added before final release.
2. Magic number is 0x**A12**.
3. When symbol value is refering to storage class C_REGPARM or C_REG, the *symbol value* represent the following:
   Each of the four bytes represent one register number. Byte 0xff is indicating no register.

   Example; register pair R1:R0 => Symbol value = {0xff,0xff,0x01,0x00}