

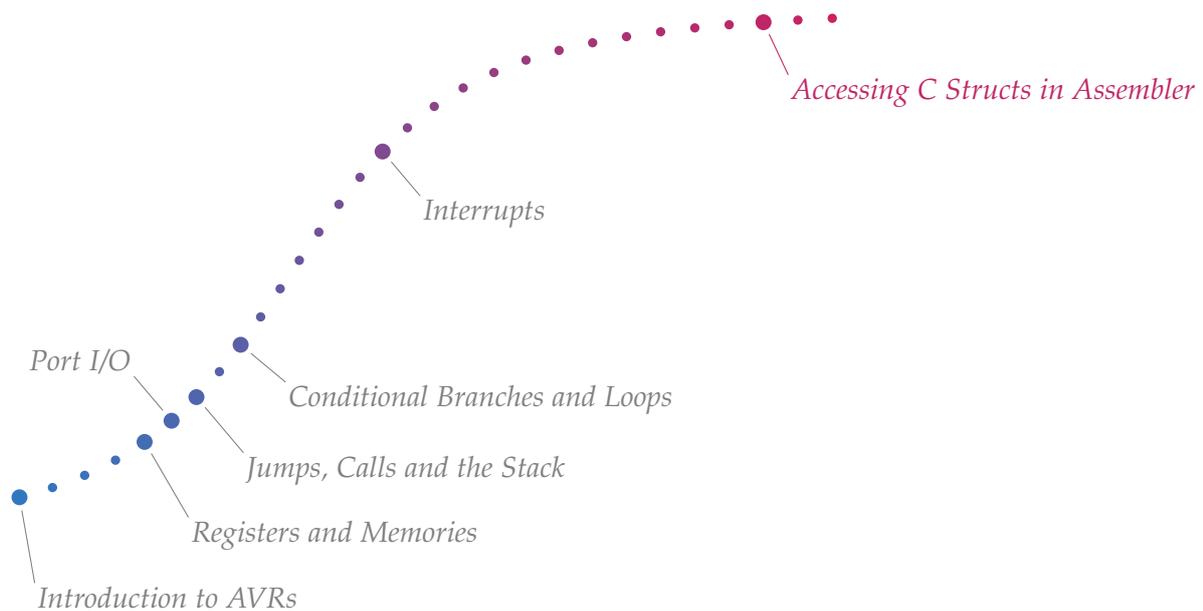
www.avrbeginners.net

C/Assembler Tutorial

Accessing C Structs in Assembler

Author: Christoph Redecker

Version: 1.1.2



This tutorial is licensed under a Creative Commons Attribution–NonCommercial–NoDerivs 3.0 Unported License:
<http://creativecommons.org/licenses/by-nc-nd/3.0/>.
Permissions beyond the scope of this license may be available at
<http://www.avrbeginners.net>.



ACCESSING C STRUCTS IN ASSEMBLER PROGRAMS is not an AVR issue, but a generic programming problem. However, knowing how to do it is especially useful when an ISR (or any other function) must be “hand-coded”. This tutorial outlines the steps necessary for making C structs and their members’ offsets visible to assembler code and how to use them.

The procedure of making C structs usable in assembler code is, as it is presented here, neither system- nor compiler-independent. Nonetheless, it is very easy to port between systems — I’m not sure about compilers. Furthermore, a text-processing tool such as gawk is needed. For this tutorial it is assumed that gcc is used.

Initial Situation

The first thing we need when assembler code must access a C struct is the C struct itself, including a type definition. These are in `myStruct.h`:

```
#include <stdint.h>

struct SMyStruct_impl
{
    uint8_t first;
    uint8_t second;
};
typedef struct SMyStruct_impl TMyStruct;
```

The defined type has two members, `first` and `second`, both are an unsigned 8-bit integer. `first` starts at offset 0 and `second` starts at offset 1. This file is included by our main file, `main.c`:

```
#include "myStruct.h"

volatile TMyStruct tMyStruct;
void asm_out(void); /* prototype */

int main(void)
{
    tMyStruct.first = 0xAA;
    tMyStruct.second = 0x55;
    /* here we call the asm function */
    asm_out();
    while(1);
    return 0;
}
```

Two things must be considered here: the compiler has no control over the code behind `asm_out()`. It only knows the prototype of

This document has embedded files. If your reader doesn’t support embedded files: don’t worry, their content is included inline in the text.

gawk is available at <http://www.gnu.org/s/gawk/>; a windows port is available at <http://gnuwin32.sourceforge.net/packages/gawk.htm>.

`myStruct.h`’s header guard is not displayed here to save space. The embedded file does contain a header guard.

this function, but not what variables it might touch. This is why `TMyStruct` is declared `volatile` — it prevents the compiler from optimizing away the two assignments to the struct in `main()`. `asm_out()` will later use the values assigned to the struct members. This code is written as if there was no interface between C and Assembler; in fact, one could know write `asm_out()` in C.

Extracting the Structure Offsets

The structure offsets are known when `TMyStruct` is typedef'ed, so `myStruct.h` will be needed for this process. Only the compiler can turn the type definition into offsets, so compiler call is necessary in which the compiler is used to calculate the offsets and write them into a separate file. The following is the file `myStruct_offsets.c`:

```
#include <stddef.h>
#include "myStruct.h"

#define _ASMDEFINE(sym, val) asm volatile \
    ("\n-> " #sym " %0 \n" : : "i" (val))
#define ASMDEFINE(s, m) \
    _ASMDEFINE(offsetof_##s##_##m, offsetof(s, m));

void myStruct_defineOffsets() {
    ASMDEFINE(TMyStruct, first);
    ASMDEFINE(TMyStruct, second);
}
```

These macros, as well as the following procedures, are based on those presented at <http://docs.blackfin.uclinux.org/doku.php?id=toolchain:gas:structs>

Two macros are defined: `_ASMDEFINE(sym, val)` and `ASMDEFINE(s, m)`.

`ASMDEFINE` uses `s` (a structure name) and `m` (a structure member) to assemble a symbol string. Assuming that `s` is `TMyStruct` and `m` is `first`, the string is `offsetof_TMyStruct_first`.

Then `_ASMDEFINE` is used to insert a line of assembler into the compiler output, using `asm volatile`. This is a “fantasy” instruction, starting with `->`. Afterwards `val` is added as an integer value at the end of the line. `val` was supplied by the calling macro, `ASMDEFINE`, and is equal to the member’s offset in the struct.

At the end of `myStruct_offsets.c`, the two offsets are actually created, by calling `ASMDEFINE` in a function. This is a dummy function and will not show up in the final binary!

The assembler, which is `gas` in this case, cannot assemble the lines starting with `->` — it would emit an error, because `->` is not a valid assembler instruction. The compiler needs to be stopped *before* it calls the assembler. For `gcc`, this is done with the `-S` option. If you haven’t done this before, create the files

myStruct.h and myStruct_offsets.c, each with the content shown above, open a terminal and execute

```
avr-gcc -S myStruct_offsets.c -o -
```

The trailing - is important, as it tells the -o option to write the output to stdout. You should see the object file output, including two lines starting with ->. Here is a sample:

```
myStruct_defineOffsets:
    push r29
    push r28
    rcall .
    in r28, __SP_L__
    in r29, __SP_H__
/* prologue: function */
/* frame size = 2 */
/* #APP */
; 53 "myStruct_offsets.c" 1

-> offsetof_TMyStruct_first 0
```

As you can see, the function myStruct_defineOffsets is created, and at the end of the above piece of object code the offset of the first member of TMyStruct is shown — we're almost there. The -> lines now have to be extracted from the object code, which can be done using gawk:

```
avr-gcc -S myStruct_offsets.c -o - | gawk '($1 == "->")
{ print "#define " $2 " " $3 }' > myStruct_offsets.h
```

The compiler output is piped to gawk, which replaces -> with #define, and writes the resulting line to its own output. gawk's output is written to myStruct_offsets.h. That file now contains:

```
#define offsetof_TMyStruct_first 0
#define offsetof_TMyStruct_second 1
```

The offsets are now ready for use in an assembler program!

Using the Structure Offsets

Now it's time to include this header file in an asm file called asm_out.S:

```
#include <avr/io.h>
#include "myStruct_offsets.h"
#define work 18 // our working register

.extern tMyStruct // "import" tMyStruct from main.c
```

```

.section .text

.global asm_out // "export" for the linker

asm_out:
    // directly accessing the first struct member, as in
    // PORTD = tMyStruct.first;
    lds work, tMyStruct + offsetof_TMyStruct_first
    out _SFR_IO_ADDR(PORTD), work

    // directly accessing the second member
    lds work, tMyStruct + offsetof_TMyStruct_second
    out _SFR_IO_ADDR(PORTD), work

    // indirect access: Z->tMyStruct as in
    // TMyStruct* Z = &tMyStruct;
    ldi ZL, lo8(tMyStruct)
    ldi ZH, hi8(tMyStruct)

    // indirectly accessing the first member as in
    // PORTD = Z->first;
    ldd work, Z+offsetof_TMyStruct_first
    out _SFR_IO_ADDR(PORTD), work

    // indirectly accessing the second member
    ldd work, Z+offsetof_TMyStruct_second
    out _SFR_IO_ADDR(PORTD), work

```

The assembler file includes the headers, imports the symbol `tMyStruct` and places `asm_out` in the text section.

`asm_out` writes the values of each structure member to `PORTD` in two different ways (you should be familiar with them). It is “exported” as a global symbol so that the linker can find it. Finally, it is possible to call the `asm` function from `main.c` (see above). One final note: the structure must be declared `volatile`, because the compiler could otherwise optimize the member assignments away – they are not used in any part of code *that the compiler is aware of*.

Putting it all Together

After everything is more or less explained, we can put the whole thing together:

First, create a C file for each structure you need the offsets of. The macros defined above can be put into a header file, call it `asmDefine.h` for example. The newly created C files need to be

More about sections at
http://www.nongnu.org/avr-libc/user-manual/mem_sections.html

compiled every time the structure definition changes, using the command line that first creates object code and then extracts the offsets. This can be done with an IDE that support pre-build steps, a batch file, or built into a makefile — your choice.

Second, include the generated header file in any assembler file that needs access to the offsets. Any structure that is used must be imported as an external symbol for the linker to find it.

Third, take into account that the compiler is not aware of the assembler code's functionality. Depending on optimisation settings, it might delete assignments in your main code if they are of no effect to the compiler.

Using this in Real Projects

In a Makefile

I have not yet managed to add the required functionality to a makefile. If anyone wants to try it, go ahead, I'll be happy to include the results here!

In the Code::Blocks IDE

C::B can compile files in different ways, based on their extension. The first step is to register a new file type in C::B's advanced options for AVR-GCC. Go to "Settings" → "Compiler and Debugger" and select AVR-GCC. Go to the "Other Settings" Tab and open the "Advanced Options" (at the bottom). You should see a window that lists a number of commands and their associated source extensions on the left, and a list of macros on the right. Add the source extension "coffsets" (without the leading period) to the "Compile single file to object file" command. Select the new extension and change the command line macro to

```
$compiler $options $includes -S -x c $file -o - |
gawk -f coffsets.awk > $file_dir/$file_name.h
```

Again, this is supposed to be just *one* line. The gawk commands have to be stored in `coffsets.awk`, because for some reason C::B cannot handle gawk commands in the command line. This `coffsets.awk` has to be stored in the same directory as your project if no path is specified for gawk's `-f` option.

The compiler usually automatically compiles C code if the file extension is `.c`. The compiler does not know the new extension `coffsets`, which is why the option `-x c` is added to tell the compiler that our input file contains C code. Now `myStruct_offsets.c` must be renamed to `myStruct_offsets.coffsets`,

Tested with Code::Blocks 10.05. C::B is available at <http://www.codeblocks.org>.

and C::B will automatically apply the new compilation command to this file.

One more thing is important: when C::B has compiled all files, it also links all files. It would try to link the object file `myStruct_offsets.o`, which doesn't exist. Remove the "Link" tick from the file's advanced build options: right click on `myStruct_offsets.coffsets` in the project tree, select "Properties", go to the "Build" tab, and uncheck "Link". Also lower its "Priority weight", so that the file is compiled earlier — before `asm_out.S` is assembled.

To summarise it:

- Add a new compilation rule for `.coffsets` files,
- rename the relevant files to end with `.coffsets`,
- change the files' options so that they are not linked,
- lower the files' priority weight (so that it is compiled earlier).

Revision History

1.1 Changes:

- Removed second appearance of `main.c` in the text,

Additions:

- Embedded source code files,
- Added Code::Blocks configuration section.

1.0 Initial release