AVR Assembler for Complex Projects

Author: Dr. Thomas Redelberger redetho@gmx.de

Version 0.3 2018-01-20

Contents

1. Legal Notices

Copyright

Disclaimer

This information is provided as-is. I do not take any responsibility and do not grant any warranty nor do I imply any fitness for any purpose or kind, nor shall I be liable for any issues when using it.

Translations

Both the English and German language versions are from me. German is my mother tongue. Improvement suggestions are always welcome. People are invited to do translations to other languages. But they shall properly cite and credit me and refer back to this original.

2. Preface

This document addresses all people who want to write assembly code that is well structured, re-usable, easy to maintain and efficient. It is based on my own experience with Microchip (previously Atmel) AVR 8-bit microcontrollers, which I program in assembler. This document is *not* an introduction to programming AVR devices. In particular the AVR 8-bit instruction set and how to achieve various computational results are *not* covered in this document. Please see information elsewhere on the web regards those.

I decided to use assembler language for my very first steps with AVR microcontrollers to learn how to use them starting from the basics. For my first bigger AVR ATMega project I decided to stay with assembler because the functionality I needed was very hardware related, was heavy in low level interfacing and used very few mathematical computations.

I am using the AVR Assembler from Microchip. Thus, what I report here is most relevant if you also use this assembler. Some tips might also be useful when using other assemblers.

With AVR Assembler I mean here AVR Assembler 2 (AVRASM2). The older AVRASM, which had been distributed with AVR-Studio earlier, does not have the functionality described here.

For many projects assembler will not be the optimal tool. GCC from the AVR-GNU tool chain is a powerful, proven and free C compiler. Many example code and libraries are substantial advantages. You might want to look also at the GNU Assembler (abbreviated GNU AS or GAS; in the AVR world it is called avr-as). Presumably avr-as has a steeper learning curve than AVRASM2, but it is very powerful and can be combined with GCC.

3. Key AVRASM2 Characteristics

No Linking

AVRASM2 assembles directly from assembly source code to a binary executable (well, to a HEX file that is a representation of the executable). There are no intermediate steps. There are no object or library files. This has the following pros and cons:

- When you assemble, you always use all the latest code. There is no need to check for the latest versions, which you would usually using a tool like "make"
- Dependencies are explicitly specified in the source code. If linking was used dependencies needed to be managed. Again, this is usually done using "make"
- Assembly time increases the more complex a project is, because everything needs assembling every time. On today's development hardware I did not find this to be a serious issue. Example: one of my projects consisting of 29 files and a total of 3 000 lines of code (LOC) assembles in less than one second
- With AVRASM2 there is only one assembly unit and there is only one name space. This requires extra care to avoid dangerous name clashes
- Code re-use is more difficult as for example in a GCC/avr-as environment. The programmer needs to organise this on his own. In particular re-using assembly code from other people is hardly possible because there are no defined and agreed standards for software interfaces
- Mixing assembly code and code from other programming languages is not possible. This is a serious drawback, because for bigger projects you might want mix programming languages and re-use code that was written in those.

Two Pass Operation

AVRASM2 is a "two pass assembler". This means AVRASM2 reads and processes the source file(s) two times as two "passes". With the first pass over the source code it builds some tables it then re-uses at the second pass. This allows for quite some freedom to arrange the order of code.

As a side note: Most C-compilers are single pass compilers. This forces some specific order of code, like include statements need to go to the top of the including file.

C-Style Pre-Processor

Atmel had added a C-style pre-processor to AVRASM2. Pre-processor directives start with a "#" character instead of "." (Dot) which signifies assembler directives. According to Microchip, the C-style pre-processor of AVRASM2 shall work like a standard C-compiler pre-processor.

Dynamically Generated Symbols

At first sight AVRASM2 does not allow for symbols that are generated dynamically. Using appropriate macro programming, this is still possible to achieve. This can be used to implement control structures for example similar to the C-language's "switch" statement.

4. General Principles

As with other programming languages, there are some general principles that you might want to follow:

Go for Well-Structured Source Code

No doubt assembly code can look very "ugly". Ugly code is more difficult to understand, debug and maintain. Thus, you might want to

- implement a clear hierarchy of the code that helps to understand the structure and purpose of the code
- modularise the code to enable re-use.

Help Code Portability

Code portability here means to make sure the code can run on sufficiently similar AVR devices with no or minimal changes.

Enable Performance

- Assembly code can yield superior speed to high level languages. However, this advantage should not be overestimated. Today's compilers get very close to optimum speed
- What you can do at development (assembly time) do a development time rather than at run time. Assembler has features and tools that allow to do that without negatively effecting the readability of code.

All this requires using advanced features of AVRASM2. On the downside this might have the consequence, that porting to another assembler will require quite some extra work, for example porting to avr-as.

5. How to Structure the Source Code

Using the File System to Structure Code

It is common sense to reduce the complexity of a problem by breaking it down in smaller, less complex parts. This strategy is called "divide and conquer". When programming, an obvious means to do this is to break the source code in smaller units. Like with most developments tools, AVRASM2 supports breaking up the source code in various files on the operating system's file system (Windows or Linux supported by Wine in the case of AVRASM2).

In AVRASM2 the `.include` directive allows to structure the source code in a tree like manner. A project must have a "main" or "root" source file which can include a number of other source code files, which in turn can include more source code, such that a tree is formed. If an include file contains another include, a relative path is not relative to the include file, but rather relative to the root source file that contains the first include.

In AVRASM2 the lexically first executable code generated will also be the first code that executes after power up or reset of the microcontroller.

AVRASM2 does not pose any limitations on include files. AVRASM2 leaves it completely to the programmer what to put in included files. The included files are just literally inserted at the point of inclusion. Thus, the programmer has full freedom how to structure his source files.

One way to find reasonable include files is to group all source code belonging to the same object in one include file. Finding appropriate objects to implement for your problem at hand in assembler is not different than for other programming languages. Hardware units can be considered objects too. For example, I group all code belonging to UART handling in one source include file, all code for TWI handling in another. The idea being of course, to re-use these "objects" in various projects that need UART or TWI support.

To check if your choice of objects is reasonable, you might want to check if the reduction of complexity is effective. One indication for this could be whether the different files which include each other are roughly of balanced size, if size is a good enough approximation of complexity. To use the analogy of a tree, branches should be of similar size such that the tree looks well formed.

Structuring the source code as a tree is a necessity in AVRASM2 because there is no other means, as there is no linking step. It is less obvious, but an environment which uses linking is also working in a tree-like manner, as code which is added in the linking step might require other code to be included as

well (usually from code libraries) which in turn can lead to additional code needing to be added and so on.

Declaration Type Code versus Definition Type Code

One key consideration when using any programming language is, which language constructs actually produce executable code (definitions) and which are "helper" constructs (declarations). With AVR microcontrollers, executable code means code that gets stored in program flash memory (`.cseg`) or *constants* that get stored in the program flash memory (`.cseg`) or data in the EEPROM flash memory (`.eseg`). Language constructs which relate to RAM (`.dseg`) are a special case. They do not generate code but rather reserve storage space. But I still call them definitions.

The key difference between using assembler and any high-level language is that the programmer has ultimate control what code gets generated. Hence the distinction between definitions and declarations is very important but not straight forward:

- Assembler "directives" declare things, for example symbolic names for constants. The directives `.db, .dw, .dd, .dw` are exceptions as they *do* generate code. The directives `.byte` and `.org` are special cases that do not generate code but rather reserve storage space.
- Assembler "instructions" stand for processor instructions that get assembled to executable machine code and define hence the program flow. An exception to this are instructions that appear within a macro declaration, that is between the `.macro` and `.endm` directives. These instructions are being stored temporarily by the assembler. Code will only be generated by a macro *invocation*. You could think of macro declarations like small internal include files, that the assembler stores in RAM of the development PC. At macro invocation, the assembler inserts the code fragment one-to-one, similar to an `.include`.
  Well, one-to-one is not entirely correct, because you can modify the generated code at macro invocation time using macro arguments.

To further structure include files, I am using a paradigm that is commonly also used with languages like C. I distinguish between include files which do not generate any code when inserted using `.include` , but just contain declarations. My convention is to give these include files the extension ".inc". By contrast include files with extension ".asm" contain definitions which generate code.

Lexical Order of Include Files

Lexical order is the order in which the assembler "sees" the source code, considering that all the includes are inserted at the respective places such that the source code can be seen as one long string of statements.

The two-pass action of the assembler leaves considerable freedom as to where put what in the main source file and in include files. This of course raises the question of a "reasonable" order of the source code.

I chose to not rely on the two-pass feature too much, rather I adopt techniques usually seen in C source code. There, C compilers pose stricter requirements, because most C compilers are one pass:

- Any declarations that are used by the including part of the source code shall be included *before* they are lexically referred to. In other words: include files that contain declarations are included at the top of the including source file. To flag that a file contains only declarations, I give it the file extension ".inc". This corresponds to ".h" files in C.
- Include files that contain definitions i.e. which generate code, I put in files with extension ".asm". I include those at the bottom of the including source file. These files are in effect replacing the missing linking functionality of AVRASM2.

Example how I thus modularise and re-use UART code:

```
…
.include "trUARTS.inc"
…
rjmp      RESET                ; Interrupt vectors follow
.org      URXC0addr            ; USART0 receive complete
rjmp      trUARTrxInterrupt0
.org      UDRE0addr            ; USART0 transmitter register empty
rjmp      trUARTtxInterrupt0
```

```
…
.set       TRUART_RX_BUF_SIZE0 = 256       ; default in trUARTS.inc is 16
MtrUARTinit 0, BAUD_MIDI
…
;; send a byte
rcall      trUARTtx0
…
…


;;; lowest level UART access; this also includes the interrupt handlers
          .include "trUARTS.asm"
```

The symbol `TRUART_RX_BUF_SIZE0` is declared in trUARTS.inc and set to default value of 16. The including code changes this to 256 to have a larger UART receive buffer. Based on this symbol trUARTS.asm will reserve RAM (`.dseg`) of 256 bytes to hold this receive buffer.

The macro `MtrUARTinit` is declared in the file trUARTS.inc. There is no function `trUARTinit` because with most projects, hardware initialisation is done lexically only once. Hence a function would be just more overhead and add no further value. However, `trUARTtx0` is a function defined in trUARTS.asm. There is a corresponding macro defined in trUARTS.inc. Below I will explain the reasons to do so. Interrupt handlers like `trUARTrxInterrupt0` are defined in trUARTS.asm .

Units of Functionality

Traditionally, the smallest functional unit in programming are functions, sub-routines or method invocations.

Assembler offers these as well, albeit with more work needed by the programmer. Assembler macros provide the programmer with another feature usually not available in high level languages (well, the C language macros of the C-pre-processor are not as powerful as their assembler cousins).

Assembler macros can be used to "inline" code. The programmer himself decides.

In high level languages, the compiler decides which functions to inline. The programmer may be able to influence this to a certain degree, for example using compiler options.

I will describe below how to implement the choice between function and inline code using AVRASM2 efficiently.

6. It is All About Symbols

Symbols are a core assembler concept. In AVRASM2 symbols always stand for numbers. If you want to give strings a symbolic name, you have to use the C-style pre-processor. Symbols can stand for any constant numbers and also for addresses in program memory (`.cseg`), in EEPROM (`.eseg`) or in RAM (`.dseg`).

Labels are special symbols, which always stand for memory addresses. Syntactically they are the first item in a source code line and are terminated with a colon. A label which stands for an address in program memory (`.cseg`) denotes either a jump target or the start address of a function or an interrupt handler. Labels which stand for addresses in EEPROM or RAM are used for storage operations. By using labels, there is never a need to use absolute memory addresses.

As said above labels are also symbols. AVRASM2 does do some book-keeping how labels and symbols have been declared, as we can see by inspecting the "map" file, that AVRASM2 optionally creates (default file extension: ".map"). But there are no restrictions whatsoever how labels and symbols are used and if you confuse them, there are no assembler warnings.

Hence, I have adopted the convention to use all upper-case letters and underscores for symbols that represent constants. Symbols that are mixed case and start with lower case stand for addresses. For constants I follow an additional convention to show the purpose of the constant by appending a suffix:

XYZ_OFFS	Offset, e.g. byte offset of an element of a structure from the start of the structure
XYZ_BITNO	Bit number, from 0-7
XYZ_MASK	Bit mask, e.g. 0x0F
XYZ_LEN	Length, e.g. of an array or a structure

XYZ_SIZE	same as LEN
XYZ_NUM	Number, e.g. number of array elements
XYZ_END	Index of the last array element = highest array index i.e. XYZ_END = XYZ_LEN -1, because the index starts at 0

However, the suffixes do no mean anything regards the data type. AVRASM2 treats all symbols as 64-bit signed integers.

Scope of Names

A key characteristic of AVRASM2 is the direct assembly from source code to executable. Each project consists of only one "assembly unit", which is the root source file and the tree of all included files. The assembler sees all this as one long string of statements.

This has the consequence that all symbols are global. To avoid name clashes, you must use some systematics to keep symbol names unique. This has to be achieved using name conventions too. To avoid name clashes, I use the prefix technique. Each symbol defined is prefixed with a short character sequence which is related to the file name of the include file in which it is declared. With this you also avoid clashes with the vast number of symbols for hardware addresses and bit numbers, which are declared in the official include files from Microchip, which are specific for the microcontroller at hand.

For example, I got the files trUARTS.inc and trUARTS.asm containing code to deal with UARTs. There are for example the symbols `trUARTrxInterrupt0` - which is the entry point to the receiver interrupt handler - and `TRUART_RX_BUF_SIZE0` which sets the receiver FIFO buffer size.

There are many coding conventions and even more debates about which are "better". It is just important for you to decide for one of them and then stick to it.

Compare this with using avr-as which requires a separate linking step. Hence you can divide each project into different objects (units, linking objects), which are assembled separately and linked together finally. Each assembly unit has a separate symbol name space. Hence you can choose if a symbol shall only be known in this assembly unit or shall be known globally. The linker only knows of global symbols and he can use those to resolve references. The linker does not know of local symbols, unless they are deliberately given to the linker to be used later by a debugger.

Name Spaces

Assemblers are using few namespaces compared to high level languages. In AVRASM2 there are the following classes of symbols and names

1. Symbols declared using `.equ` or `.set`
2. Symbolic names for processor registers, declared using `.def`
3. Macro names, declared using `.macro`
4. CSEG Labels
5. DSEG Labels
6. ESEG Labels.
7. CSEG labels in a macro declaration
8. DSEG labels in a macro declaration
9. ESEG labels in a macro declaration

As it turns out, for cases 1. to 6. there is only *one*, global name space per project. For example, this has the consequence that a macro cannot have the same name as a label or symbol. For the cases 7. to 9. there is one name space per macro *invocation*. All assemblers offer a way to create labels that are local to a macro. Some assemblers require a special syntax for a label to be local, like e.g. avr-as. In AVRASM2 *all* labels defined in a macro are local. Because there is separate label namespace per macro invocation, labels do not clash

- when the same macro gets invoked multiple times

- when other macros use the same labels.

Below I will discuss how to use local symbols and by doing so also mitigate the negative consequences from the fact that there is only one assembly unit.

Note that you *can* define a global label in a macro by using the syntax

```
            .equ        myGlobalLabel = PC
```

This look useless at first sight because the assembler will issue an error code when the macro is invoked a second time. But if you construct "`myGlobalLabel`" dynamically using a macro parameter, then you can use that construct to create control structures, for example something similar to the switch statement in C.

Because there is only one global name space - except for macros - I use further conventions to avoid clashes. All my processor register names start with a capital "R", all my macro names start with a capital "M", except for macros, which deliberately look like processor instructions; please see the example "`ldiw`" below. The capitalisation is just meant to increase readability, because by default AVRASM2 does not distinguish case.

Labels for Control Structures

Contrary to high level languages, you always need a lot of labels for loops and jump targets. Because labels are local to macros, it is easy to come up with unique names there.

Therefor I always use the same consistent label names for loops. My loop label names do *not* explain what the loop does. This I do in comments. Rather the label names mark the beginnings and endings of loops.

Local labels can be short because they need only be unique within a macro. I use the "t" for the top of a loop, "b" for the bottom and "c" for any "continue" like branch target. I append Arabic numerals if there are multiple loops in the same macro.

Example:

```
            ...
            ;; algorithm to make sure I store neither the checksum nor the F7 byte
            ld          r23,        z+          ; get first (previous) byte
            rjmp        b1

t1:          st         x+, r23     ; store previous (first) byte
             inc        r25         ; byte counter
             breq       error       ; maximum 256 bytes
             mov        r23, r22    ; the next (second) becomes the previous (first)
b1:          ld         r22, z+     ; get second (next) byte
             cpi        r22, 0xF7   ; end?
             brne       t1

            ...
            rjmp        exit

error:
            ...

exit:
```

Quite often I also use an `exit` and `error` label in a macro. I put `exit` always at the macro bottom

Above example also shows the practice to first jump to the bottom of a loop, if checking a loop condition is the first thing that needs to be done. This is more efficient then checking the loop condition at the top of the loop, because otherwise the bottom of the loop would need to contain an unconditional jump which is overhead and is usually more often executed than the unconditional jump to the bottom which occurs only once.

7. Implementing Functions and Interrupt Handlers

How to Use Macros to Trade Speed for Size

Another important feature of macros is that they provide the feature of "inline" code. The macro code is inserted at the point where the macro is called (invoked). The invocation has the effect that the assembler inserts the code that has been put in the macro body at macro declaration time. This is happening at development (assembly time) and not at run time, as the term "invocation" might suggest. This has a speed advantage over function calls, because the overhead of call/return and related manipulation of the stack is not needed. The disadvantage is of course, that code size is growing with each macro invocation, whereas code size is only growing marginally with each function call.

In my AVR projects I decided to implement almost all functions as macros. This allows me to decide later if I want inline code or a function.

For this I put the body of the function in a macro, omitting the `ret` instruction. If I need the function, I call the macro followed by a `ret` and preceded by a label that stands for the function entry point. Example:

```
            .macro     MtestFunction
            .ldi       r24, 123
t:
            ...
            cp         r24, r25
            breq       t

            ...
exit:
            .endm      ; MtestFunction


            ...
            rcall      testFunction          ; function call
            ...


testFunction:                                ; function definition
            MtestFunction                    ; with body delivered by macro invocation
            ret
```

This technique does not have any disadvantage and I get in addition the advantage of local labels in the function body as I described above.

Note that the construction of functions that way still leaves the possibility to call the macro elsewhere to get an in-lined version if the speed advantage is required and justifies the longer code.

```
            ...
            MtestFunction                    ; macro invocation = inlining
            ...
```

I put function definitions at the end of the source code or in a .asm include file, if the function(s) is(are) part of a module, that can be re-used. As a convention for the macro name, I take the function name and prefix it with the capital letter "M" to correlate function and macro. I start all macro names with "M" to be able to quickly identify them and to avoid clashes with valid processor instructions, unless the macro shall have the look and feel of a processor instruction; for that please see macro "`ldiw`" below.

If, however, I find that a function is lexically invoked (called) only once, I omit the function definition and just put a macro invocation instead of the function call.

As a consequence of this construct, I follow in my macros all calling conventions and register usage conventions applicable to functions.

Using Macros with Include Files

I combine macros with the systematics of structuring and modularising the code using include files. The macro that constitutes the function body will go into .inc files. The function definition will go in a related .asm file.

I apply the same systematics also to interrupt handlers, with the following differences:

- Interrupt handler definitions end with `rte` instead of `rts`
- Macros which make up the interrupt handler bodies are declared in .asm files - the same that contains the interrupt handler definition - rather than in .inc files, because it does not make sense to inline interrupt handlers.

Limiting the Scope of Functions and Variables

Functions Local to a Macro

Functions local to a macro can be constructed like this:

```
            .macro     Mtest
            ldi        R24, 123
            ...
            rcall      myPrivateFunc1
            ...
            rcall      myPrivateFunc2
            ...
            rjmp       exit


myPrivateFunc1:
            clr        r24
            ...
            ret

myPrivateFunc2:
            ldi        r24,789
            ...
            ret


exit:
            .endm      ; Mtest



            Mtest
```

Labels `myPrivateFunc1` and `myPrivateFunc2` are only defined in the separate name space of the macro `Mtest`. Hence different functions of the same name could exist elsewhere without a name clash.

Caveat: also in nested macros, that are invoked in such macros like `Mtest`, labels like `myPrivateFunc1` are not known.

Variables Local to a Macro

If you want to extend the notion of local functions to local variables (which you would call "static" in C), AVRASM2 puts a hurdle:

```
            .macro     Mtest
            ...

            .dseg
myPrivateVar:
            .byte      1

            .cseg
            sts        myPrivateVar, r24

            .endm      ; Mtest


            Mtest
```

AVRASM2 issues the error message: "error: .byte directive illegal in macro definition". Variables in RAM which are private and local to a macro do not seem to be possible. However, they are possible if your use the following construct:

```
            .macro     MtestNew
            ...

            .dseg
myPrivateVar:
            .org       myPrivateVar +1

            .cseg
            sts        myPrivateVar, r24

            .endm      ; MtestNew


           MtestNew
```

This is perfectly valid, because `.byte` does nothing else but to advance the internal `.org` pointer by the given number of bytes.

The construct does have a small disadvantage though: AVRASM2 is book-keeping how many bytes a project is using and prints this at the end as small table. The "Begin", "End" and "Code" columns are correct. The "Data" column is not in this case. The assembler seems to just add-up all `.byte` statements and does not refer to `.org`. In other words: the executable code is correct, but the data column in the table is wrong. Hence you need to calculate the used RAM yourself using Begin and End.

Also, local variables in EEPROM can be created with a bit of care, because

```
            .macro     Mtest
            ...
            .eseg
myPrivateVar: .db      123

            .endm      ; Mtest
```

yields the cryptic error message:

```
AVRASM: AVR macro assembler 2.2.6 (build 63 Apr 26 2016 14:42:08)
Copyright (C) 1995-2016 ATMEL Corporation
===> FATAL ERROR: Internal: malformed directive: myPrivateVar:      .db
Assembly failed, 1 errors, 0 warnings
```

When you put the label and `.db` of different lines, all is fine

```
            .macro     Mtest
            ...
            .eseg
myPrivateVar:
            .db        123

            .endm      ; Mtest
```

8. When to Use the C-style Pre-Processor

AVRASM2 features in addition a C-style pre-processor. However, this does not seem to be a separate processing pass before the two assembly steps. Rather the pre-processor directives seem to be executed together with the first assembly pass.

Some C-pre-processor directives seem to be redundant to their older dot-style cousins of the assembler. For example, Microchip documents that `#include` is functionally equivalent to `.include` .

Pre-processor `#define` seems to be redundant to assembler `.equ` or `.set` directives. The differences are:

- If you need to set symbols from the assembler command line, you can define a pre-processor symbol. Assembler symbols cannot be set from the command line.

It is confusing but the simple C-pre-processor `#defines` - which stand for strings or numbers and do not take arguments - are also called "macros" even though they much rather feel like symbols.

Concerning operators, functions and expressions:

- The assembler *can* calculate with assembler symbols and constants. This however is limited to arithmetic and logic operations and is done with 64-bit accuracy.
- AVRASM2 symbols *cannot* stand for strings and there are no string functions, except for `strlen()` . The argument for `strlen()` has to be either a string literal, for example `"abc"` - which does not look very useful - or an assembler macro parameter - which seems a bit more useful, or a C-pre-processor macro, which might be useful too.
- The C-pre-processor *cannot* calculate with symbols or constants even though if often looks like this (it is the assembler who does it or the C-compiler). The only exception is `#if` where the pre-processor *does* evaluate the expression. On the other hand, the pre-processor does have some string functions
- The pre-processor string functions are limited to concatenation and stringification to a string literal and pattern matching. But you can do amazing things with this limited set. Unfortunately, the AVRASM2 pre-processor does not always work as expected from a standard C pre-processor. The example code in Appendix 1 highlights this.

The limitations in the string functions and the related pattern matching has the consequence that more advanced uses of the pre-processor do not work, like X-Macros https://en.wikipedia.org/wiki/X_Macro or to emulate arithmetics.

Pre-processor macro functionality partly overlaps with assembler `.macro` functionality. For example

- `.macro` arguments can be numeric, symbols or literals or string literals (using " as delimiter)
- pre-processor macro arguments can also be numeric, symbols or literals or string literals (using " as delimiter)
- Assembler macro arguments can be concatenated with one another or with literals. The literals cannot be numbers however (the reason for this is unclear)
- Pre-processor macro arguments can be concatenated as well. This is what the concatenation operator `##` is meant for. The rules for this are arcane but are well documented. Alas AVRASM2 is deviating here as was mentioned earlier.

Quite often it is a matter of taste which of the two macro constructs to use. I usually try to get as far as possible using assembler directives and use the C-pre-processor only exceptionally, for example to avoid including the same code more than once. For example, in the file trUARTS.inc:

```
;;; file trUARTS.inc
;;;
#ifndef _TRUARTS_INC_
#define _TRUARTS_INC_
…

#endif  /* _TRUARTS_INC_ */

;;; ***** END OF FILE ****************************************************
```

This technique is also often used with C and especially with C library functionality. The same effect could be achieved using assembler directives. But then symbols like `_TR_UARTS_INC_` would appear in the assembler symbol table and in the map file which I want to avoid. Hence, I prefer the C-pre-processor variant here.

9. Lexical Details

I try to stay as compatible to avr-as as much as possible. Hence, I write all assembler directives in lower case, despite they appear in upper case in the official Microchip documentation. AVRASM2's default setting is to ignore case.

Because of avr-as compatibility I write `.endm` rather than `.endmacro` .

I do comments and code indentation as suggested by AS mode of EMACS.

10. Using Processor Resources in an Optimal Way

To Use or Not to Use the Stack

Push and pop instructions on AVR devices are relatively efficient, each taking two clock cycles only. This is important as you often need to save processor registers on the stack. This is mandatory in interrupt handlers and may occur in functions to preserve registers for the calling or called function (see register usage conventions below).

If you do not have enough registers available, the standard procedure is to allocate short lived variables (so called "auto" variables in C) on the stack. For this you need to use a so-called frame pointer, who is pointing to the allocated stack space. There is no alternative to this on AVR devices because the stack pointer is not available as a standard register-pair, let alone for register relative addressing. (Microchip calls this addressing "data indirect with displacement"). Two registers, more precisely register-pairs, offer this functionality: y and z. AVR GCC is using y as frame pointer, which is plausible, because z is needed for other things that are not supported by y.

Storing and saving register contents on the stack using the y register as a frame pointer also takes two clock cycles each with displacement. The displacement is limited to 0-63 bytes and is always positive.

However, allocating and de-allocating space on the stack adjusting the stack pointer is relatively costly, because the stack pointer is implemented as I/O registers and not as a 16 bit or 24-bit processor register. The problem is that I/O registers can only be manipulated 8-bit at a time. To safely manipulate the stack, which is usually 16 bits wide, requires interrupts being switched off during allocation and de-allocation and then on again. This needs time and adds to code length.

Because of this I try to get away with using processor registers as much as possible rather than using the stack for "auto variables".

If I need arrays, I check carefully if they really need to be dynamically created & destroyed (=auto variables in C speak) or whether I can afford to have them statically allocated in RAM i.e. in `.dseg`.

Speaking of interrupts and interrupt handlers: I always save registers on the stack. On AVR devices it is possible to save registers in other registers that you reserve for this purpose. This has some small speed advantage over push and pop instructions. This option is possible because on AVR devices normally interrupts cannot be interrupted, unless you enable this explicitly in an interrupt handler.

How to Work with Structures and Arrays

Except for the most trivial cases you have to allocate structures and arrays in RAM.

Example for a structure in C-Syntax:

```
struct point {
          int x;
          int y;
};
```

which you might use like:

```
          struct point pt;
          pt.x = 456
          pt.y = 789
```

If the integer data type uses two bytes, you could write the following equivalent assembly code:

```
          ;;define structure
          .dseg
ptxlo:    .byte     1
ptxhi:    .byte     1
ptylo:    .byte     1
ptyhi:    .byte     1

          .cseg
```

```
            ldi         r24,  low(456)
            ldi         r25, high(456)
            sts         ptxlo, r24
            sts         ptxh, r25
            ldi         r24,  low(789)
            ldi         r25, high(789)
            sts         ptylo, r24
            sts         ptylo, r25
```

Addressing using absolute addresses - like in the example using `sts` - is called by Microchip "Data Direct". The code above is correct, but the fact that we are dealing with a structure is only apparent in the variable names.

I find the following way to program clearer (the macros used are listed in Appendix 3):

```
            ;; declare structure offsets
            MtrSetOffset          0
            MtrNewOffsetSymbol    POINT_X_OFFS, 2
            MtrNewOffsetSymbol    POINT_Y_OFFS, 2
            ;; length of a structure instance
            .equ        POINT_LEN = TR_OFFSET

            ;; allocate RAM = define structure
            .dseg
pt:         .byte       POINT_LEN

            .cseg
            ldi         r24,  low(456)
            ldi         r25, high(456)
            sts         pt+POINT_X_OFFS+0, r24
            sts         pt+POINT_X_OFFS+1, r25
            ldi         r24,  low(789)
            ldi         r25, high(789)
            sts         pt+POINT_Y_OFFS+0, r24
            sts         pt+POINT_Y_OFFS+1, r25
```

This way to code is especially clearer if you got multiple instances of the same structure, either one by one or as an array. Using the naive coding above, you need one set of symbols per instance, whereas in the coding using offsets, you only get one additional symbol per instance.

Just to be clear: the generated machine code is identical! The issue here is clarity and how easy it is to manage and expand the code.

If you need access by pointer - like array access - the coding using offsets is the only way to do it. And that highlights the advantage of that notation because static and dynamic structures are dealt with using similar syntax:

```
            ldiw        z, pt                ; load pointer register with address
            ldi         r24,  low(456)
            ldi         r25, high(456)
            std         z+POINT_X_OFFS+0, r24
            std         z+POINT_X_OFFS+1, r25
            ldi         r24,  low(789)
            ldi         r25, high(789)
            std         z+POINT_Y_OFFS+0, r24
            std         z+POINT_Y_OFFS+1, r25
```

Access via a pointer register with offset - like in the example above using `std` - is called "Data Indirect with Displacement" by Microchip.

If `pt` was an array, for example with `PT_ARR_NUM` elements

```
            .dseg
ptArr:      .byte       PT_ARR_NUM*POINT_LEN
```

then, after an

```
            adiw        z, POINT_LEN
```

the next array element could be accessed.

By the way: AVR processors access RAM only slightly slower than registers. Absolute addressing using `lds` or `sts` just needs two clock cycles. The indirect (pointer) access using `ld` or `st` or indirect with offset using `ldd` or `std` or pre-decrement or post-increment are also only using two clock cycles. Exception: `st x, Rr` and `st x+, Rr` only need one clock cycle instead of two.

This means that indirect access using pointer registers mostly does not bring a speed advantage over absolute access. Hence, it is perfectly fine to use absolute addressing for static objects, like shown above. However absolute addressing needs one more program memory word because of the address word.

Structures as Parts of Static Objects

When you need to create several instances of a structure, you need to reserve RAM for each instance and create a symbol that points to the start address of the structure. Instead of writing this down again and again using `.dseg`, you could use the following macro to save writing:

```
            .macro     MtrFIFONew
            ...

            .dseg                           ; switch to data section
dummy:                                      ; dummy local symbol in dseg
            .equ       DSEGtrFIFO@0 = dummy ; proper global symbol
            .org       DSEGtrFIFO@0 + TRFIFO_BUF_OFFS + @1       ; allocate space


            .cseg                           ; switch back to code
            ...                             ; initialise instance

            .endm      ; MtrFIFONew
```

The complete macro is listed in Appendix 4. As I wrote above, the label "dummy" creates a symbol that is only known locally in the macro. As the name dummy implies, I do not use it as such. Rather the following `.equ` creates a global symbol with the same value, i.e. the same address. The FIFO methods will later access the object using this symbol. The macro parameter @0 is providing for a unique name for this object instance. Hence you need to provide a unique name literal in the macro invocation. The instance is further identified using that literal:

```
            MtrFIFONew UARTrx, 256
```

In this example the second macro argument specifies the FIFO buffer length in bytes.

The FIFO access methods are also implemented as macros, for example:

```
            MtrFIFOgetNxtByte    UARTrx
```

I intend to publish the full FIFO code under GPL.

Special Case EEPROM

EEPROM access on AVR devices is only possible using special I/O-registers. Access takes several processor cycles and further more you might need to wait, if a write access is still pending.

Hence it might make sense to copy data from EEPROM to RAM.

What Registers to Use for What?

The standard AVR 8-bit microcontrollers, like an ATMega, have 32 registers. The assembler programmer has all freedom to use these as he wants.

However, I decided to not re-invent the wheel, but to adopt the register usage conventions of the AVR GNU-C compiler (AVR GCC). Please see the table in Appendix 2.

As a rule of thumb, I use registers r18 - r27 and r30, r31 for short lived purposes in functions (auto variables in C-speak) as some of them are any way used for argument passing. Hence, I use r1 - r17 for longer living variables (these would be "static" variables in C-speak). To help keeping track which registers are used for what, I use symbol definitions for registers a lot, using the assembler directives

`.def` and `.undef`. I use this especially for r1 - r17 to make sure I do not use a register twice accidentally.

I use register names also for short lived variables to improve code readability. Then `.def` and `.undef` come in pairs like so:

```
.macro     MtrUARTrxInterrupt
.def       scratch    = r25
.def       data       = r24
push       scratch
LOAD       scratch, SREG
push       scratch              ; save status reg
push       data
…
LOAD       data, UDR@0          ; get data as fast as possible to make
                                ; room in the hardware buffer
…
…
pop        data
pop        scratch              ; pop status from stack
STORE      SREG, scratch        ; restore status register
pop        scratch
.undef     data
.undef     scratch
.endm      ; MtrUARTrxInterrupt
```

Registers r26 to r31 - also called x, y, z - support special addressing modes. There are differences between x, y, and z as to which addressing modes are supported:

Data Indirect with Displacement (e.g. `ldd`, pointer access with 0-63 byte offset):	y, z
Data Indirect with Pre-Decrement or Post-Increment (e.g. `ld`):	x, y, z
Program Memory Constant Addressing (`lpm`, `elpm`, `spm`):	z
Program Memory with Post-Increment (`lpm z+`, `elpm z+`):	z
Indirect Program Addressing (`ijmp`, `icall`):	z

In addition to x, y, z, register pair r25:r24 also supports 16-bit operations like e.g. `adiw` .

11. Program Flow and Concurrency

Concurrency Due to Hardware Devices

Modern microcontrollers offer a multitude of built-in hardware units that can run independently from and in parallel to the main processor. For example: ADC, timer, UART, TWI, SPI and there can even be a multiple of such units within a microcontroller. As they can work in parallel with the processor, we need to cater for this concurrency. Such hardware devices usually offer a "polling" mode or an "interrupt" mode to communicate with the main processor. Usually the interrupt mode is more efficient because no processor time is wasted in waiting. However, interrupt operation is more complex to design and implement.

In other environments an operating system takes care of low level hardware functions and interrupts and it intermediates with the user application. On small microcontrollers there might not be enough space for an operating system to fit next to the application. Recent models however offer ample program space such that an operating system environment may be viable. However, for AVR microcontrollers such as ATMegas, a well-known and de-facto standard operating system environment does not seem to exist.

How Multiple Tasks Run Quasi Concurrently

Even simple microcontroller applications need to do multiple things that appear to the user to happen in parallel. For example, polling a keyboard or other input devices, updating a display, etc. Again, managing such might be handled by an operating system.

If you do not have an operating system, the simplest strategy is to just let the task run one after the other in a collaborative way. That is, if a task has nothing to do, it let others run.

Example Program Flow

The structure I use follows the following principles:

- Keep interrupt service routines as short as possible
- Interrupts communicate with the main program by "producer-consumer" paradigm. Usually a FIFO is a good vehicle for this
- The main program is divided in "tasks"
- The tasks are executed one-by-one in a "round robin" type fashion
- Each task first checks whether this task shall do something or not. If not, it gives back control immediately to the main program which passes control to the next task round
- These checks shall be as short and as efficient as possible because they are overhead which is reducing the available total computing power.

I usually have the following program layout:

- Reset and Interrupt vectors
- Interrupt handler(s)
  - M...ISR1
  - M...ISR2
  - ...
  - M...ISRN

Reset:

- Initialisation
- Main program

mainloop:

  - M...Task1
  - M...Task2
  - ...
  - M...TaskM
  - rjmp mainloop

Both interrupt handlers and tasks are written as macros. Interrupt handlers are then expanded following a label signifying the entry point and a `rte` after the macro expansion, like this:

```
trUARTrxInterrupt0:
          MtrUARTrxInterrupt 0
          rte

trUARTrxInterrupt1:
          MtrUARTrxInterrupt 1
          rte
```

Above example shows two receive interrupt handlers for an ATMega that has two UARTs. The code for the two UARTs is hence duplicated by using the same macro with a macro argument that specifies the physical UART instance

Tasks are just in-lined rather than using functions, because they appear lexically only once and hence functions would just add overhead. Example:

```
mainloop:
          MhandleTWIrxTask
```

```
        MhandleMIDIpRxTask

        MhandleTWItxTask

        MhandleKeyPadRxTask
        rjmp    mainloop
```

The task macros have to check for work at the top and leave immediately (jump to the bottom of the macro) if there is no work to do.

This scheme is simple and is sufficient as long as the round robin approach is fair enough for the application at hand.

12. Tips and Tricks

How to Make Your Code Shorter and More Readable

Example macro:

```
        .macro  ldiw
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; Macro to simplify loading a RAM address into x, y, or z
;;; marco arg1=@0:   either x, y, z register to load with...
;;; marco arg2=@1:   ...address
;;;
        ldi     @0L,  LOW(@1)
        ldi     @0H, HIGH(@1)
        .endm   ; ldiw
```

Example invocation:

```
        ldiw    z, bufferAddress
```

This is a typical macro that looks like a valid processor instruction. That is deliberate, because `ldiw` *might* be part of the AVR processor instruction set. If Microchip decided one day to add such an instruction to the instruction set, you would just remove the macro declaration.

Portability

It is always a good idea to use macros other people have written and tested. One example is the file "macros.inc" from Atmel. The macros in macros.inc are described in Atmel Application Note AVR001: Conditional Assembly and portability macros - doc2550.pdf.

The portability macros abstract I/O-access from how it is implemented in various AVR devices. Thus, it helps writing code that runs on various AVR devices without change.

The only thing I added to macros.inc was

```
...
#ifndef _MACROS_INC_
#define _MACROS_INC_
...
#endif  /* _MACROS_INC_ */
```

to avoid multiple inclusion.

13. An Example Development Environment under Windows

I develop under Windows 10 Pro 64-Bit and use AVRASM2 form the command line or from within EMACS. For this I copied the relevant sub directory and "installed" it in a suitable subdirectory of "Program Files" (or Program Files (x86) for 64-bit) plus a registry entry.

On another PC, I use AVR Studio 4.19 under Windows 10 Pro 32 bit with AVRISP MKII to program AVR devices.

On both machines I use AVRASM2 version 2.2.6 (build 63 Apr 26 2016 14:42:08). I edit the source code using GNU EMACS 25.1 with CUA-keys enabled and AS-mode switched on. I have customised AS-mode a bit to fit my personal preferences.

I "installed" EMACS on OneDrive. Thus, I always have the same environment available on both PCs.

The EMACS serial-terminal-mode allows to show debug output which I output via an AVR UART interface.

14. References

AVRASM2 Manual:
http://ww1.microchip.com/downloads/en/DeviceDoc/40001917A.pdf

AVR Instruction Set Manual:
http://ww1.microchip.com/downloads/en/DeviceDoc/Atmel-0856-AVR-Instruction-Set-Manual.pdf

15. Appendix 1 - Shortcomings of the C-Pre-Processor

The following example code shows that the C-style pre-processor, that is built into AVRASM2, does not offer the same functionality like the C-pre-processor of avr-gcc (GNU CC), with respect to string functions, pattern matching and symbol substitution:

```
/*********************************************************************/
/* This code has been run through AVR GCC {gcc version 4.8.1
(AVR_8_bit_GNU_Toolchain_3.4.5_1522)}
and AVRASM2 {AVR macro assembler 2.1.17 (build 435 Apr 10 2008 09:27:55)}
with the command lines
 > avr-gcc -E testpp.c
The -E options runs pre-processing only and outputs to stdout.
And
 > avrasm2 -m map.txt testpp.c
The map file map.txt shows the names of the two generated assembler symbols and
their value in hex
*/


/* Example 1
 Works as expected: GCC preprocessor and AVRASM2 preprocessing give the same result
*/

#define CRSYMBOL(sym, val) .equ sym = val
#define VALUE 123


          CRSYMBOL(symname, VALUE)

/* this expands to
   .equ symname = 123
as expected */


/* Example 2
 Works as expected in GCC preprocessor. However AVRASM2 does not expand VARPOSTFIX,
hence is /not/ compliant to C pre-processor standards.
*/

#define CAT_(a,b) a ## b
#define CAT(a,b) CAT_(a,b)
```

```
#define VARPOSTFIX 3


          .equ CAT(symname, VARPOSTFIX) = VALUE

/* The GCC preprocessor expands this to
          .equ symname3 = 123
as expected.
However AVRASM2 expands this to
          .equ symnameVARPOSTFIX = 123
which is /not/ what we expect */


/* the following nop is just there to keep AVRASM2 from complaining about an "empty
source file" */
          nop

/************************************************************************/
```

16. Appendix 2 - Register Usage Conventions

The following table shows which registers are meant to be used for what. I took this over from the AVR GCC documentation:

```
;;;;;;;;;;;;;;;;;;;; Register Usage ;;;;;;;;;;;;;;;
;;;
;;; I use any of registers r0 - r17 only via .def !
;;;
;;;Reg.    Function   Saving               Return Value
;;;-------------------------------------------------------------------------
;;;R0      Temp Register        Scratch
;;;R1      Always 0   Callee must clear
;;;R2                 Callee must save
;;;R3                 Callee must save
;;;R4                 Callee must save
;;;R5                 Callee must save
;;;R6                 Callee must save
;;;R7                 Callee must save
;;;R8      Next Arg   Callee must save
;;;R9      Next Arg   Callee must save
;;;R10     Next Arg   Callee must save
;;;R11     Next Arg   Callee must save
;;;R12     Next Arg   Callee must save
;;;R13     Next Arg   Callee must save
;;;R14     Next Arg   Callee must save
;;;R15     Next Arg   Callee must save

;;;R16     Next Arg   Callee must save
;;;R17     Next Arg   Callee must save
;;;R18     Next Arg   Scratch
;;;R19     Next Arg   Scratch
;;;R20     Next Arg   Scratch
;;;R21     Next Arg   Scratch
;;;R22     Left most arg:       Scratch              Long byte0
;;;         Long byte0
;;;R23     Left most arg:       Scratch              Long byte1
;;;         Long byte1
;;;R24     Left most arg:       Scratch              Char
;;;         Char                              Int low byte
;;;         Int low byte                               Long byte2
;;;         Long byte2
;;;R25     Left most arg:       Scratch              Char: 0 or sign ext.
;;;         Int high byte                              Int high byte
;;;         Long byte3                        Long byte3
;;;R26     X:         Scratch
;;;R27     X:         Scratch
;;;R28     Y: Frame ptr.        Callee must save
;;;         Low Byte
```

```
;;;R29     Y: Frame ptr.        Callee must save
;;;        High Byte
;;;R30     Z:         Scratch
;;;R31     Z:         Scratch
```

17. Appendix 3 - Macros to Deal with Structures

AVRASM2 does not offer any directives to calculate structure member offsets. The following two macros help with that:

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;; Macros to generate symbols for structure member offsets
;;;
;;; Usage:
;;;        MtrSetOffset is used to set the offset "counter",
;;;        specifically to reset it
;;;        MtrNewOffsetSymbol generates a new global symbol to be used as
;;;        a structure member offset
;;;

           .macro     MtrSetOffset
           .set       TR_OFFSET  = @0
           .endm      ; MtrSetOffset


           .macro     MtrNewOffsetSymbol
           .equ       @0         = TR_OFFSET
           .set       TR_OFFSET  = TR_OFFSET + @1
           .endm      ; MtrNewOffsetSymbol
```

18. Appendix 4 - Marco to Create a New FIFO Instance

Extract from file trFIFO.inc. The macro creates a new FIFO instance in RAM and initialises it:

```
;;; Create data structure offsets. These are the same for all FIFO instances
           MtrSetOffset         0
           ;; structure offsets of the indices
           MtrNewOffsetSymbol   TRFIFO_HEAD_OFFS, 1
           MtrNewOffsetSymbol   TRFIFO_HDBT_OFFS, 1
           MtrNewOffsetSymbol   TRFIFO_TAIL_OFFS, 1
           MtrNewOffsetSymbol   TRFIFO_TLBT_OFFS, 1
           ;; also BUF_OFFS is the same for all FIFOs
           MtrNewOffsetSymbol   TRFIFO_BUF_OFFS, 0


           .macro     MtrFIFONew
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; Macro to create a new FIFO instance
;;; - the /instance/ FIFO symbols, i.e. static data specific to the instance
;;; - reserve space in dseg for instance
;;; - initialise instance data structure.
;;; The FIFO buffer proper in dseg is also instantiated but is /not/ cleared
;;;
;;; Macro arg1:      Name of the FIFO instance
;;; Macro arg2:      FIFO size in bytes (must be 2, 4, 8, 16, 32, 64, 128 or 256)
;;; Macro arg3:      Packet size in bytes (must be a power of 2 and < FIFO size)

           ;; check the macro arguments
           .if        @1 > 256 || @1 < 2
            .error "FIFO size must be <= 256 and >= 2"
           .endif
           .if        exp2(log2(@1)) != @1
            .error "FIFO size must be a power of 2"
           .endif
           .if        @2 >= @1 || @2 < 1
            .error "Packet size must be smaller than FIFO size and >= 1"
```

```
            .endif
            .if       exp2(log2(@2)) != @2
             .error "Packet size must be a power of 2"
            .endif

            ;; static characteristics specific to this FIFO instance,
            ;; later accessed as immediate values
            .equ      TRFIFO_SIZE_@0       = @1
            .equ      TRFIFO_MASK_@0       = TRFIFO_SIZE_@0-1
            .equ      TRFIFO_PACKET_LEN_@0 = @2


            .dseg               ; switch to data section
dummy:                          ; dummy local symbol in dseg to capture the dot
            .equ      DSEGtrFIFO@0         = dummy    ; proper global symbol =
                                                      ; address of the instance
            .org      DSEGtrFIFO@0 + TRFIFO_BUF_OFFS + @1       ; allocate space for
                                                                ; instance in dseg


            .cseg                                     ; switch back to code
            ;; set generic variables to zero = FIFO is empty.
            ;; The FIFO buffer is /not/ cleared
            ;; TRFIFO_BUF_OFFS is the same as the length of the generic part
            MtrClrMem  DSEGtrFIFO@0, TRFIFO_BUF_OFFS

            .endm     ; MtrFIFONew
```

19. Appendix 5 - An Example Project: Software for a Synthesizer HUI

I wrote this document after I built and programmed my own do-it-yourself human user interface (HUI) hardware to control my do-it-yourself analogue music synthesizer. The HUI has the following features:

- Uses rotary encoders and push button switches for input and LEDs as output
- Supports MIDI
- Uses a bunch of ATMega8 and ATMega644P

Please see http://web222.webclient5.de/prj/MusicEl/GTHL1/index.htm

* * *