How to Get the Most Out Of Your Embedded Hardware While Keeping Development Time to a Minimum

A Comparison of Two architectures and Two IDEs for Atmel AVR 8-bit Microcontrollers

NICLAS ARNDT

KTH ROYAL INSTITUTE OF TECHNOLOGY
INFORMATION AND COMMUNICATION TECHNOLOGY

How to Get the Most out of Your Embedded Hardware while Keeping Development Time to a Minimum

A Comparison of Two Architectures and Two IDEs for Atmel AVR 8-bit Microcontrollers

Niclas Arndt
Bachelor Thesis, Information Technology

Abstract

This thesis aims to answer a number of basic questions about microcontroller development:

- What's the potential for writing more efficient program code and is it worth the effort? How could it be done? Could the presumed trade-off between code space and development time be overcome?
- Which microcontroller hardware architecture should you choose?
- Which IDE (development ecosystem) should you choose?

This is an investigation of the above, using separate sets of incremental code changes (improvements) to a simple serial port communication test program. Two generations of Atmel 8-bit AVR microcontrollers (ATmega and ATxmega) and two conceptually different IDEs (BASCOM-AVR and Atmel Studio 6.1) are chosen for the comparison.

The benefits of producing smaller and/or faster code is the ability to use smaller (cheaper) devices and reduce power consumption. A number of techniques for manual program optimization are used and presented, showing that it's the developer skills and the IDE driver library concept and quality that mainly affect code quality and development time, rather than high code quality and low development time being mutually exclusive.

The investigation shows that the complexity costs incurred by using memory-wise bigger and more powerful devices with more features and peripheral module instances are surprisingly big. This is mostly seen in the IV table space (many and advanced peripherals), ISR prologue and epilogue (memory size above 64k words), and program code size (configuration and initialization of peripherals).

The 8-bit AVR limitation of having only three memory pointers is found to have consequences for the programming model, making it important to avoid keeping several concurrent memory pointers, so that the compiler doesn't have to move register data around. This means that the ATxmega probably can't reap the full benefit of its uniform peripheral module memory layout and the ensuing struct-based addressing model.

The test results show that a mixed addressing model should be used for 8-bit AVR ATxmega, in which "static" (absolute) addressing is better at one (serial port) instance, at three or more the "structs and pointers" addressing is preferable, and at two it's a draw. This fact is not dependent on the three pointer limitation, but is likely to be strengthened by it.

As a mixed addressing model is necessary for efficient programming, it is clear that the driver library must reflect this, either via alternative implementations or by specifying "interfaces" that the (custom) driver must implement if abstraction to higher-level application code is desired. A GUI-based tool for driver code generation based on developer input is therefore suggested.

The translation from peripheral instance number to base address so far used by BASCOM-AVR for ATxmega is expensive, which resulted in a suggestion for a HW-based look-up table that would generally reduce both code size and clock cycle count and possibly enable a common accessing model for ATmega, ATxmega, and ARM.

In the IDE evaluation, both alternatives were very appreciated. BASCOM-AVR is found to be a fine productivity-enhancement tool due to its large number of built-in commands for the most commonly used peripherals. Atmel Studio 6.1 suffers greatly in this area from its ill-favored ASF driver library. For developers familiar with the AVRs, the powerful avr-gcc optimizing compiler and integrated debugger still make it worthwhile adapting application note code and datasheet information, at a significant development time penalty compared to BASCOM-AVR.

Regarding ATmega vs. ATxmega, it was decided that both have its place, due to differences in feature sets and number of peripheral instances. ATxmega seems more competitively priced compared to ATmega, but incurs a complexity cost in terms of code size and clock cycles. When it's a draw, ATmega should be chosen.

Table of contents

1 Introduction

1.1 Outline

This is a long thesis that covers a wide area. The reader might want to choose the parts of most interest and here I briefly describe the contents and provide reading advice.

If you want to digest this work as quickly as possible, it is recommended that you browse chapter 1, read section 3.4.1 and then read chapters 7 and 8. Chapters 3, 5, and 6 can in this case be consulted for details about particular tests and their results.

- Chapter 1 is the introduction.
- Chapter 2 describes the method and the test setup. It briefly explains and illustrates how the tests were performed. An experienced microcontroller programmer could probably skip this part.
- Chapter 3 presents the AVR 8-bit microcontroller architecture, differences between ATmega and ATxmega, and programming-related properties relevant to this thesis. It is very detailed with regards to register design, I/O and peripheral device registers, internal memories, and Atmel's advice on efficient programming. The alternative peripheral module register layout that is very important to this paper is explained in 3.4.1. I recommend every reader to read this last piece, but if you are seriously interested in efficient AVR programming you must get a solid understanding of this entire chapter.
- Chapter 4 is an overview of the two IDEs, describing their most important features and qualities. If you are mostly interested in their consequences you can find this in chapter 7.
- Chapters 5 and 6 each contain one separate IDE-specific analysis and discussion of the findings. Written as log books that document my progress, they are very detailed and include personal remarks indicating my reactions to the results. These chapters provide the empirical groundwork that also explains or "proves" my findings. You can read it as a whole or read the parts that lead up to the results you found interesting in chapter 7.
- Chapter 7 compiles and discusses the results, which leads up to a number of conclusions. All aspects considered relevant are treated here. A must-read for this paper.
- Chapter 8 is a summary of the conclusions. This is where the different lines of investigation end in IDE choice, HW selection, results from the programming tests, and conclusions on efficient programming.
- Chapter 9 holds the table of references.
- Various additional information, sources, and incremental pieces of code have been put in appendix A.
- The source code and disassemblies reside in the external appendix B due to their size. Please contact the author for a copy.

1.2 General background

For many years, I have been doing microcontroller (a.k.a. embedded systems) prototyping as a hobby. I have now reached the level at which I consider turning my hobby into a business and one of many questions is which platform I should choose in terms of

- hardware (HW) architecture and
- Integrated Development Environment (IDE).

I also want to get a deeper understanding of (microcontroller) programming; a feeling for how much computer programs can be improved in terms of performance and compiled code size and how further studies in this area could be designed:

- Should I use a generic programming style or are there differences in IDE and HW architecture that motivate different approaches?
- How good are the predefined software (SW) libraries and IDE commands with respect to compiled code size, performance, and development time?
- Should I use high-level language only or combine it with inline assembly or custom assembly functions?

On a similar note, as the great yearly increase in computer HW performance that we had grown accustomed to seems to have been slowed down, I believe that there is reason to rekindle our interest in SW performance:

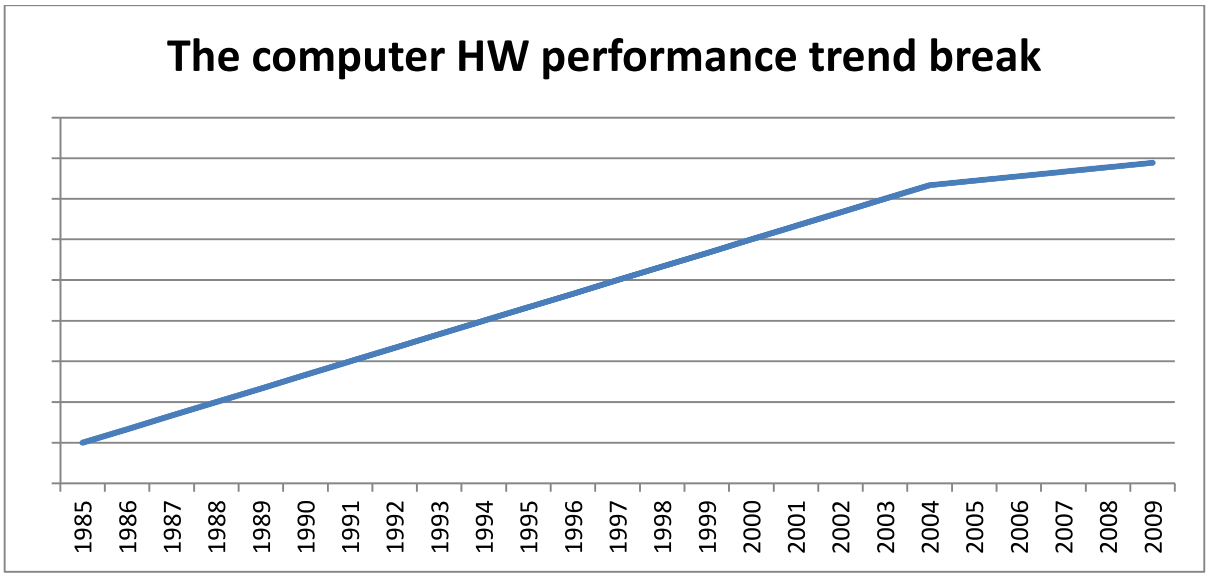


Figure 1: The general computer HW performance trend (relative performance vs. year)

This illustrative graph is representative of a number of real charts in “The Future of Computing Performance: Game Over or Next Level?” (1) [1]. In many of the most important HW metrics, the increase in performance has slowed down:

- Integer and floating-point performance
- Power dissipation and clock frequency

[1] Free download at http://www.nap.edu/catalog.php?record_id=12980

So, what ways are there to further increase performance?

"The claimed benefits of high-level languages are now widely accepted. In fact, as computers got faster, modern programming languages added more and more abstractions. For example, modern languages - such as Java, C#, Ruby, Python, F#, PHP, and Javascript - provide such features as automatic memory management, object orientation, static typing, dynamic typing, and referential transparency, all of which ease the programming task. They do that often at a performance cost, but companies chose these languages to improve the correctness and functionality of their software, which they valued more than performance mainly because the progress of Moore's law hid the costs of abstraction." (1) [1] *p107*

"Future growth in computing performance will have to come from software parallelism that can exploit hardware parallelism. Programs will need to be expressed by dividing work into multiple computations that execute on separate processors and that communicate infrequently or, better yet, not at all." (1) [1] *p105*

I too see parallelism as a very important area in software development, but I also see great potential in more efficient programming. 8-bit microcontrollers are simple and enable high-level development from which the machine code consequences can be analyzed directly. I'm hoping that such an analysis will give insights that are also applicable to PC- and server-class programming.

1.3 Commercial background

I am currently developing a series of (uninterruptible) power supply products. They have quite modest requirements in terms of performance and program memory size, but I still want to make an informed platform decision and lay a solid code foundation for what will be common functionality:

- I believe that writing good code once is cheaper in the long run.
- If the code size reduction is substantial, it will enable me to use smaller (and cheaper) devices.
- According to Johnny Burlin at IAR Systems (one of the world-leading compiler makers for embedded processors) (2), the best way to reduce power consumption is to speed-optimize the code so that the microcontroller gets the job done as quickly as possible and then goes into sleep mode. In this paper I won't go into power efficiency, but it is relevant for battery-powered devices.

1.4 Problem description

My previous designs are based on Atmel's AVR 8-bit microcontrollers, more specifically the ATmega architecture (3) [2], with the BASCOM-AVR IDE (4) [3]. Its syntax is close to Visual Basic 6, here called VB.

I have now started to use the more powerful ATxmega series (5) [4] and I am considering a switch to Atmel's IDE, Atmel Studio 6 (6) [5]. The main reason for this would be the optimizing compiler, integrated debugger, and being able to use the industry-standard C or C++ that are more easily portable to other HW. It also has support for Atmel's ARM-based products and a claimed easy transition from ATxmega to ARM due to the common Atmel Software Foundation (ASF) (7) [6] driver library.

[2] http://www.atmel.com/products/microcontrollers/avr/megaavr.aspx
[3] http://www.mcselec.com/
[4] http://www.atmel.com/products/microcontrollers/avr/avr_xmega.aspx
[5] http://www.atmel.se/microsite/atmel_studio6/
[6] http://www.atmel.com/tools/avrsoftwareframework.aspx?tab=overview

I decided that a simple feature comparison wouldn't answer all my questions. Instead, I will implement the same basic test program (a serial port communication routine) for each of the HW/IDE combinations below, with a number of incremental code modifications in order to find the optimum programming style in each situation.

I will try to see how much I can improve the generic high-level code (mostly in terms of compiled code size, but in some cases also clock cycle count and RAM usage) and then how much further I can reduce it by replacing parts with inline assembly. As a last step, I will see how much can be saved by swapping the protocol-unbound design for a protocol-bound implementation.

BASCOM AVR VB-only
BASCOM AVR VB + inline assembly
BASCOM AVR VB + inline assembly, protocol-bound implementation
Atmel Studio C-only
Atmel Studio C + inline assembly
Atmel Studio C + inline assembly, protocol-bound implementation

Table 1: Test overview, for ATmega and ATxmega respectively

Below are the main questions that will guide my work. As the investigation is open-ended and the further direction of the analysis is decided during its execution, the summary and conclusions will be shaped by the actual findings, not necessarily following this structure.

Software-related:

- How much can you improve your code? Is it worth the time and effort?
  - High-level language only
  - With inline assembly (or custom assembly functions)
- How do the two IDEs (BASCOM-AVR 2.0.7.6 and Atmel Studio 6.1) compare?
  - Ease of use
  - Productivity-enhancement tools (software libraries / built-in commands)
  - Efficiency / optimization of compiled code
  - Simulation and debugging possibilities
  - SW stability
  - Code reusability and portability to other device types or brands
  - Coverage
    - HW architectures (including easy transition between different HW)
    - SW longevity (have the version changes been smooth?)
  - User forum usefulness
- What are the differences between developing in BASCOM VB and AVR GCC C code (using their respective IDE)?

Hardware-related:

- Should you strive to migrate to the newer and more feature-rich ATxmega?
  - Features
  - Complexity
  - Maturity
  - New programming model and its effect on code size and clock cycle count

1.5 Purpose and goal

This thesis has the following purpose:

- Evaluate two microcontroller IDEs and two HW architectures for a decision about the platform for my future commercial products.
- Investigate the area of efficient microcontroller programming:
    - Learn how big is the potential for writing smaller or faster computer programs.
    - Could the presumed code space / development time trade-off be overcome?
    - Understand to what extent the programming style should be adapted to IDE and HW, how to balance the development time savings to the cost of the abstractions added by generic libraries / commands, and how much inline assembly should be used.
    - Get an initial picture of how further studies in this area could be designed.

In other words: To search for a way to get the most out of my embedded hardware while keeping development time to a minimum.

1.6 Delimitations

The thesis title is chosen for two reasons: it captures the essence of the thesis and it is believed to catch the reader's interest. It is however too big a topic to be properly addressed by a bachelor thesis. In this respect, this work aims to give a fundamental understanding of what drives (microcontroller) program size and runtime.

It seems that many companies replace their old architecture with 32-bit ARM. Maybe this is what I too will choose in the end. However, I decided that a comparison between BASCOM-AVR on 8-bit Atmel AVR ATmega and IDE *ABC* on brand *XYZ* 32-bit ARM wouldn't be meaningful, as so many things would be different that not many generalizations could be made. By choosing AVR ATxmega and Atmel Studio IDE as the alternatives, real comparisons are possible.

Development time and execution time might be difficult to actually measure. For this reason, development time might have to be a subjective "feeling" of the effort required and execution time might have to be measured by counting microcontroller clock cycles for the instructions in a disassembly of the compiled code. I chose to focus on compiled program size.

With the invention of smart phones, surf pads, and so on, it could be argued that many embedded systems are now so complex and versatile that developing them requires an operating system, high-level languages, generic drivers, and lots of abstraction. I don't oppose to this view on a part of the embedded market, but my designs (like many other microcontroller systems) are fairly simple one-task devices, so I will only consider such designs in this thesis.

1.7 Terminology

I use “IDE” (Integrated Development Environment) in the sense of development ecosystem - not only the GUI or front end.

Some of the terms I use in this thesis are partly my own:

- “Static” or “absolute” addressing: The address (typically to a peripheral IO register) is hardcoded. This should not be confused with the C language attribute “static”.
- “Dynamic” addressing: I came to use this expression when analyzing the BASCOM-AVR ATxmega code that translates a port number 0-7 to the corresponding peripheral module base address. This address is then (inside the built-in commands) used in a call to a “structs and pointers” driver routine.
- “Structs and pointers” (S&P): This refers to the new way of addressing peripherals that Atmel introduced with the ATxmega uniform register layout. For each type of peripheral, a struct with all the registers is defined. Its fields implicitly denote an offset or a displacement from the start of the struct (= the base address). The driver is written so that it only exists in one generic version. In the driver function call you include a pointer to the peripheral module’s (register group’s) base address. The base address is typically placed in the Z pointer and the sub-registers are accessed via LD (load) or LDD (load with displacement) instructions (and ST/STD for storing). The difference between “dynamic” and S&P is that the former takes a port number and the latter an address.

I use the terms “UART” and “USART” in the same sense. (USART = Universal Synchronous and Asynchronous serial Receiver and Transmitter, while UART is only asynchronous.)

After the tests I renamed them, so that there would be a clear structure. This means that in some places (e.g. code comments, paths names, and examples) the old name are still used, but I decided that it doesn’t cause any significant confusion.

There are two ways of numbering the “cells” in an array; row-major and column-major. The usual definition can be found here: (8) [7] When you think of an array like this:

0	1	2	3
A	B	C	D

the row-major representation in memory is

0	1	2	3	A	B	C	D

and the column-major is

0	A	1	B	2	C	3	D

This is a fixed part of the language you are developing in, but in a row-major language like C, you can achieve column-major behavior by swapping row and column in your declaration: When I use the term “column-major”, this is what I actually mean.

0	A
1	B
2	C
3	D

1.8 References

In this thesis I am using Zotero, Vancouver citation style. For the reader’s convenience, I generally both provide the reference and a footnote with a URL (web hyperlink) to the document so that it isn’t necessary to jump back and forth when reading.

[7] http://en.wikipedia.org/wiki/Row-major_order

1.9 Other considerations

1.9.1 IDE company participation and previous connections

Both MCS Electronics (4) [8] (the company behind the BASCOM-AVR IDE) and Atmel (9) [9] were invited to participate and/or comment on this thesis. I leave it to the reader to decide whether I am biased.

The owner of MCS, Mark Alberts, made two comments that can be found in appendix A.1. Prior to this thesis, I already had a friendly professional relationship with Mark Alberts, having shared library code with an application note for an SD memory card driver and moderating its user forum thread at MCS' web site.

Atmel sponsored "my" team in a robot project course last spring and very generously gave all members an ARM development board and debugger afterwards. However, we had severe difficulties with the initial delivery and nearly had to abandon Atmel. We were afterwards asked to provide feedback on the software and shared a strong opinion on the usefulness of their driver library.

At the start of my thesis work, Atmel declined my request for a contact for this thesis (appendix A.1). In February 2014 when my work was almost complete, I contacted Atmel again with an invitation to read and comment on my work, but I did not receive a reply.

1.9.2 Environmental aspects / sustainable development

On a large scale, even small improvements in clock cycle count should amount to a significant difference in total power consumption.

1.9.3 Gender, ethnic, or religious aspects

Not applicable. The areas of programming types and HW / IDE selection are orthogonal to questions of discrimination based on gender, ethnic belonging, and religious beliefs.

[8] http://www.mcselec.com/
[9] http://www.atmel.com/

2 Method

What's the best way to compare two IDEs or HW architectures? I fear that a feature table with summation of weighted scores wouldn't capture the real qualities that in my experience become clear only after a period of actual use.

Considering that I also want to compare programming styles, I decided that the center of this thesis should be the incremental code changes on each platform. By focusing on a specific test program and going as deep into this topic as possible, I believe that I will implicitly also get a reasonably good picture of the IDEs' ease of use, qualities, and (part of) the two HW architectures.

I decided that the best place to start is the main program loop, which in this application is quite strongly tied to the (serial port) communication with the PC. It controls the program flow, is relatively application-dependent, well delimited, and also involves a specific hardware module (i.e. driver development).

The BASCOM-AVR part is completed before the start of the Atmel Studio 6 part.

2.1 Method description

The method used in this thesis is a fairly controlled (dual) set of iterative and incremental experiments. The area (main loop with serial port communication routines) is fixed, but the direction for the incremental changes is determined during the actual testing. When- or wherever I find something of interest, I investigate its cause and consequences, directly influencing the direction of the rest of the testing. The two analysis chapters are separate logs of what I do and find.

This work could be seen as an initial scientific investigation, with a complete set of source code and incremental analyses so that others could repeat and question the actual findings. Perhaps the results could be used as a starting point when formulating a series of tests of all ATmega and ATxmega peripherals or a bigger programming model analysis, but that's for others to consider.

2.2 Test equipment and setup

The PC application sends a serial port sequence of binary bytes, starting with 254 followed by message type byte, the actual data byte(s), and terminated by 255. The AVR responds to this with a message of the same kind. The PC will always wait for the response before sending the next message. The AVR will only initiate a conversation to send an error message (which is not part of this work).

The test application implements two messages:

- The PC sends [254, 243, 255] and receives [254, 242, 1, 2, 3, 255]
- The PC sends [254, !=243, …, 255] and receives [254, 251, 255]

The AVR code should be written for an ATmega324A and an ATxmega128A1 with conditional compilation. The first high-level-only versions are based on a circular buffer that gets its data from the RX interrupt routine for USART. The main program loop calls a sub-routine that polls this buffer and extracts any received data and puts it into a separate array. When the entire message is received, the appropriate response is sent. At the end, a protocol-bound implementation is developed. It might have to be based on inline assembly.

2.2.1 Test beds

I used the following microcontroller types:

- ATmegaXX4 (164/A/P/PA, 324/A/P/PA, 644/A/P/PA, and 1284/P)
    - This family supports JTAG debugging.
- ATxmegaA1(U) (64A1/A1U and 128A1/A1U), where "U" indicates that it is a later (bug-fixed) revision that includes a USB module. This USB module is not covered by this thesis.
    - This family supports JTAG and PDI debugging.

Within each type, the main difference lies in the size of the various memories, where the number states the program memory size (324 == 32 kB and 128 == 128 kB program flash EEPROM).

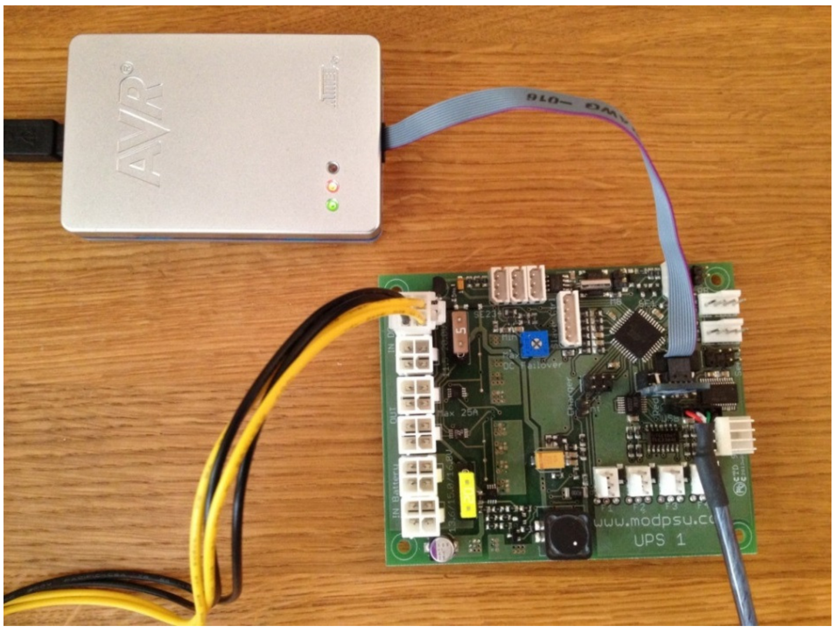


Figure 2: ATmega324A-based board, with two USARTs

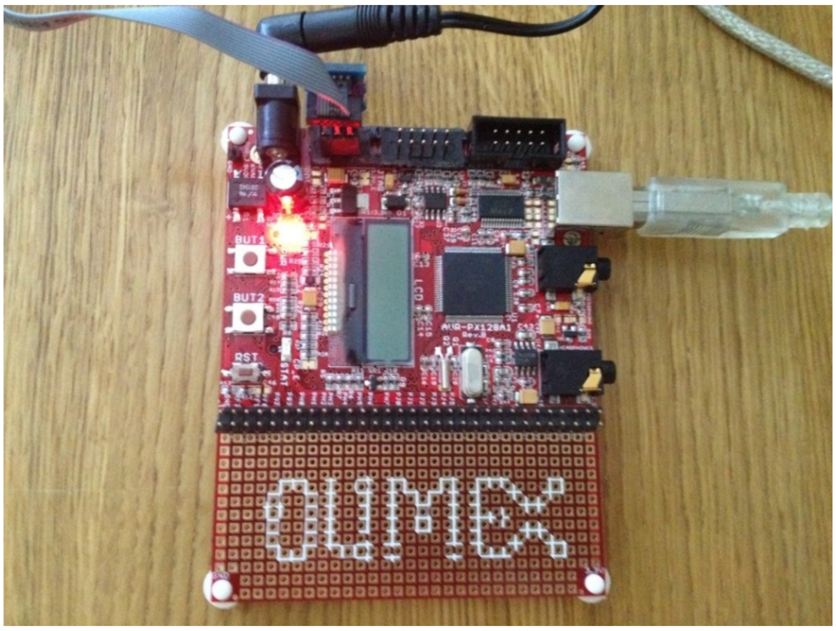
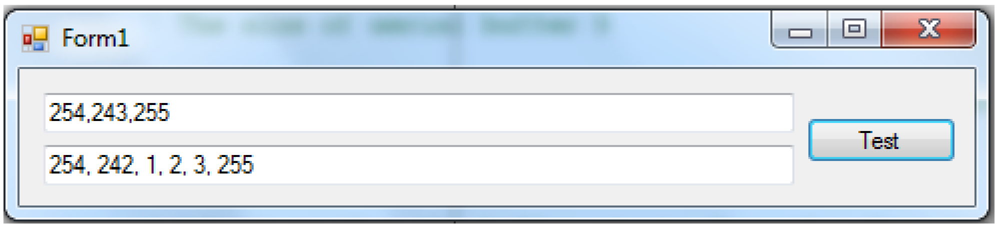


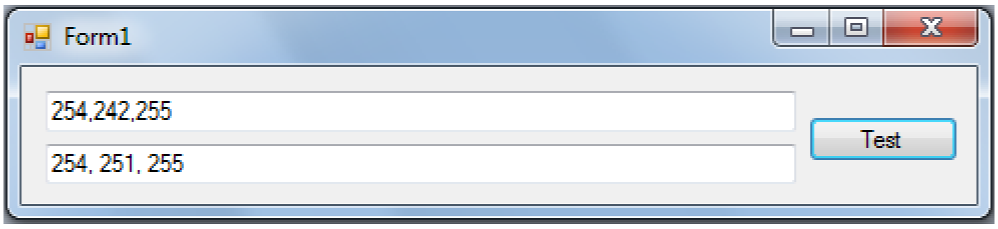
Figure 3: ATxmega128A1-based board, with eight USARTs, of which two are used in the tests

2.2.1.1 PC application

The PC application is developed in MS Visual Studio 2010 C#.Net, using the SerialPort class. Just getting this to work with static COM port number assignment is very simple. However, nowadays people mostly use USB<->serial bridges. They tend to acquire a new port number each time you plug them in to a different port. For this reason, I added the FTDI .dll and wrapper class for USB bridge identification and found a nice generic COM port listing class on the internet. With them the application automatically connects to the right COM port number.



```
254,243,255
254, 242, 1, 2, 3, 255
```



```
254,242,255
254, 251, 255
```

Figure 4: The two supported messages with response

(Enter the message in the upper textbox, click ”Test” and the response is shown in the lower one.)

2.2.1.2 How to disassemble

Please see appendix A.3 for information about how to disassemble.

3 The Atmel AVR 8-bit microcontrollers

3.1 Introduction

The AVR microcontroller is an 8-bit modified Harvard load/store RISC architecture with a 2-stage 1-wide pipeline, which means:

- RISC, Reduced Instruction Set Computing: By using instructions that each does a very small and specialized task, the clock speed can be increased. This boils down to higher over-all performance. The other (and older) philosophy is CISC, Complex ISC, which has instructions that often do very intricate (series of) operations requiring several clock cycles. RISC also often uses the "load/store architecture" that only operates on memory using specific instructions, rather than as part of the aforementioned complex instructions. (10) [10]
- Harvard architecture: It uses separate buses for program and data memory. (11) [11]
- Modified: It is possible to access the program memory area as read-only data memory (11) (and also update the program memory using a so called boot-loader program).
- 8-bit: It uses data registers 8 data bits wide (but the program memory uses 16 bit or sometimes 2*16 bit wide instructions).
- 2-stage pipeline: The first stage fetches the next instruction and the second stage executes the current instruction. (12) [12]
- 1-wide: It does one operation at a time. (12)
- Microcontroller: A processor with most of the peripherals and memories on the chip.
- AVR: Believed to stand for "Alf-Egil and Vegard's RISC" processor.

AVR originates in the 1992 graduation thesis written by the two Norwegian students Alf-Egil Bogen and Vegard Wollan. In 1997 the AT90S1200 was launched as a microcontroller product by Atmel Corporation. It was one of the first in the industry to use internal flash program memory. (13) [13]

The AT90S series evolved into two product lines with self-explanatory names, ATtiny and ATmega, which a few years later were accompanied by the ATxmega, a major revision or even redesign. They all use the same instruction set (although each model might not support every instruction). A 32-bit AVR was launched in 2006 (14) [14] and starting in 2008, Atmel is now licensing much of the 32-bit ARM-based microcontrollers and microprocessors. (15) [15] This thesis only treats the AVR 8-bit Atmega and ATxmega, henceforth referred to as AVR, ATmega, or ATxmega.

The "AVR and AVR32 - Quick Reference Guide" (16) [16] is slightly outdated (especially as it doesn't contain Atmel's ARM offering), but it still provides a good overview of the AVR products. I could also point to "Microprocessor (MPU) or Microcontroller (MCU)?" (17),[17] which is a marketing presentation that gives a good background to what was considered important in 2013.

[10] http://en.wikipedia.org/w/index.php?title=Reduced_instruction_set_computing&oldid=594087688
[11] http://en.wikipedia.org/w/index.php?title=Harvard_architecture&oldid=585324105
[12] http://www.atmel.se/Images/Atmel-8331-8-and-16-bit-AVR-Microcontroller-XMEGA-AU_Manual.pdf
[13] http://www.youtube.com/watch?v=HrydNwAxbcY
[14] http://en.wikipedia.org/w/index.php?title=AVR32&oldid=587706001
[15] http://en.wikipedia.org/w/index.php?title=AT91SAM&oldid=584613739
[16] http://www.atmel.se/Images/doc4064.pdf
[17] http://www.atmel.se/Images/MCU_vs_MPU_Article.pdf

3.2 Architecture details

3.2.1 Registers

AVR has 32 general-purpose eight-bit working registers. The last six can be used as three pairs of 16-bit registers, called X, Y, and Z, e.g. when addressing memory locations. All of these can do pre- or post-incementation, while Y and Z also support positive 6-bit displacement, which is practical when accessing arrays, SW stack, or sub-registers that control a peripheral. Z can be used to read or write flash program and special device settings. The register with the higher number is the most significant.

16 bits equates to a 64 k bytes data memory or a 64 k words program memory addressable space. (AVR program memory is made up of 16-bit instruction words, so 128 kB of program memory can be addressed with 16 bits.) When accessing a location above this, you must use an additional register for the >16 bits:

- RAMPX, RAMPY, or RAMPZ: for the X, Y, or Z register pairs >64k byte (kB) data memory.
- RAMPD: when the instruction includes a 16-bit constant to access >64kB data memory.
- EIND: to do jumps or calls to >64k word program memory.

The SP (Stack Pointer) is a special register pair that resets to the highest internal SRAM address and automatically updates when you execute PUSH or POP instructions. It is also the place where the return address for the CALL instructions is stored.

The R0+R1 register pair is also the destination for the MULxx multiplication instructions.

The SREG (Status REGister) contains bit-wise results from or input to arithmetic and logic operations and the global interrupt on/off setting.

Some instructions only operate on the top half of the registers (R16-R31), typically the “immediate” ones taking a constant, and yet some others only work with R16-R23.

The 16-bit ADIW and SBIW instructions add or subtract a constant to/from the register pairs R24+R25, X, Y, and Z. As you will typically want to reserve X, Y, and Z for stack operations and use as memory pointers, R24+R25 is left for other 16-bit purposes, for example a counter.

This sub-section is largely based on (12) [18] and (18) [19].

I present the conventions for register use and calling in appendix A.5.

[18] http://www.atmel.se/Images/Atmel-8331-8-and-16-bit-AVR-Microcontroller-XMEGA-AU_Manual.pdf
[19] http://en.wikipedia.org/w/index.php?title=Atmel_AVR_instruction_set&oldid=571841646

3.2.2 ATmega324 data memory

32 Registers	0x0000 - 0x001F
64 I/O Registers	0x0020 - 0x005F
160 Ext I/I Reg.	0x0060 - 0x00FF
Internal SRAM 1024/2048/4096/16384 x 8)	0x0100 - 0x04FF/0x08FF/0x10FF/040FF

Table 2: Data Memory Map for ATmega164A/324A/644A/1284 et al

(The table above is based on ATmega164A/PA/324A/PA/644A/PA/1284/P Complete (19) [20], p21)

The data memory is actually a collection of different types of memory that often have two different addressing modes:

- The 32 general-purpose working registers. Apart from their register number (by which they are directly accessible by most instructions), they are also mapped into the data memory space at 0x0000 – 0x001F, accessible via instructions LD/LDS/LDD and ST/STS/STD.
- The 64 lowest I/O registers. They can be accessed with the “short” instructions IN and OUT on I/O address space 0x00 – 0x3F. They are also mapped into the data memory space at 0x0020 – 0x005F, in which area they can be accessed by instructions LD/LDS/LDD and ST/STS/STD. This is the reason why these particular I/O registers are referred to with the double notation 0x00 (0x0020).

  The lower 32 of these 64 I/O registers can also be bit-accessed on I/O address space 0x00 – 0x1F using instructions SBI (Set Bit in I/O register) or CBI (Clear Bit in I/O register) and the “mini-branch instructions” SBIS (Skip if Bit in I/O Register is Set) or SBIC (Skip if Bit in I/O Register is Cleared).

  In the ATmega324’s family, these 32 addresses are most importantly home to the physical ports A – D, which makes it possible to do bit manipulations on all the ports. The device also has three GPIO (General-purpose I/O) registers that are particularly useful for status flags or global variables. GPIOR0 is in I/O address space at 0x1E, while GPIOR1 and GPIOR2 are outside of the bit-operable area.

- The 160 extended I/O registers only reside in the data memory space at addresses 0x0060 – 0x00FF, accessible by instructions LD/LDS/LDD and ST/STS/STD.
- The internal SRAM starts at data memory space address 0x0100 and ends at a device-specific address that is also the end of the data memory. It can only be used with LD/LDS/LDD and ST/STS/STD instructions.

In ATmega1284, 32/100 of the peripheral registers can be accessed via IN/OUT, plus the digital IO pin registers. For more information, please see the datasheet, pp 554-557 (19)

[20] http://www.atmel.se/Images/Atmel-8272-8-bit-AVR-microcontroller-ATmega164A_PA-324A_PA-644A_PA-1284_P_datasheet.pdf

3.2.3 ATxmegaAU data memory

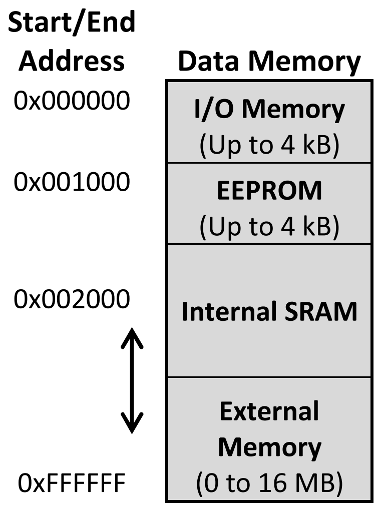


Table 3: ATxmegaAU data memory map

(The table above is based on Atmel AVR XMEGA AU Manual rev F (12) [21], p23)

Currently, there are five ATxmega series, A through E, with certain differences in functionality and intended area of use. The A series is divided into one or a few "sub"-series, e.g. A1, A3, and A4, each implementing a subset of the full A series functionality, peripheral modules, and ports (and thereby pin count). Finally, e.g. A1 exists in two memory sizes, 64kB and 128kB. The "U" states that it has built-in HW support for USB.

In the ATxmega, the 32 working registers are not mapped into the data memory space. Instead, it starts with (up to 4 kB of) I/O memory with only one address numbering. The first 64 locations can be accessed with the IN and OUT instructions and the first 32 of these can be bit-manipulated:

- At 0x0000 – 0x000F there are 16 GPIO registers that should typically be used for global variables and flags.
- At 0x0010 – 0x001F there are four sets of virtual ports. Each port can be mapped to one of the 11 physical ports A – R (whichever are available in the specific device). A port set consists of the sub-registers DIR (direction), OUT, IN, and INTFLAGS (interrupt settings), so they can be used for easy interaction bit- or byte-wise with the outside world. (Not for communicating with the built-in peripherals.)

After the 32 bit-operable registers, there are 32 more IN/OUT-operable registers for CPU, CLK, SLEEP, and OSCillator. In ATxmegaA1U, 4 out of the 61 peripheral register groups can be accessed via IN/OUT, excluding the digital IO pin registers. 4 out of the 11 IO ports can be mapped to virtual ports that are covered by IN/OUT.

Then follow the rest of the I/O registers that are accessible by instructions LD/LDS/LDD and ST/STS/STD.

[21] http://www.atmel.se/Images/Atmel-8331-8-and-16-bit-AVR-Microcontroller-XMEGA-AU_Manual.pdf

In ATxmega, the on-chip EEPROM can be accessed either in its own EEPROM address space or mapped into the data memory space starting at 0x1000 and ending no later than 0x1FFFF (depending on device-specific EEPROM size). In the data memory space, the EEPROM is only accessible by instructions LD/LDS/LDD and ST/STS/STD.

At 0x2000 the internal SRAM (of device-specific size) starts, immediately followed by (optional) external SRAM, both only accessible by instructions LD/LDS/LDD and ST/STS/STD.

3.3 HW design and programming considerations

Due to the AVR design based on the load/store architecture with 32 general-purpose working registers, a great fraction of the instructions require only one clock cycle. In internet user forums I remember seeing claims that the effective average CPI (Clock cycles Per Instruction) is about 1.5, but I haven't been able to find the source. However, the clock cycle counts in this thesis' analyses roughly confirm a CPI of this magnitude.

Another distinguishing feature of the AVR is its non-banked memory, which means that the entire data memory space is linear and continuous (even though the RAMPx and EIND registers can be seen as a way to achieve 64k banks). This makes memory pointer displacement easy and efficient.

These two things have programming, compilation, and performance consequences that I will soon delve into. I have found one Atmel document that looks to architectural choices and two that describe how they affect the optimum programming style:

- "The AVR Microcontroller and C Compiler Co-Design" (20) [22]
- "AVR035: Efficient C Coding for 8-bit AVR microcontrollers" (21) [23]
- "AVR4027: Tips and Tricks to Optimize Your C Code for 8-bit AVR Microcontrollers" (22) [24] [25]

Here I will summarize the first of these documents. The last two partly contain programming conventions that I am actually treating in a separate section, but I include them in appendix A.4 as the C code recommendations so heavily depend on the underlying hardware.

Please also see the "AVR Instruction Set" (23) [26] document.

3.3.1 The AVR Microcontroller and C Compiler Co-Design

"The AVR microcontroller was developed with the C language in mind in order to make it possible to construct a code efficient C compiler for AVR." This was done in cooperation with compiler company IAR Systems [27]:

- By not using paged memory, the memory pointers can reach 64 displacement locations instead of just 16.
- The orginal two 16-bit pointers were too few to support both SW stack and efficiently copying from one memory location to another, so a third one, X, was added.

[22] http://www.atmel.com/dyn/resources/prod_documents/COMPILER.pdf
[23] http://www.atmel.se/Images/doc1497.pdf
[24] www.atmel.se/Images/doc8453.pdf
[25] www.atmel.se/Images/AVR4027.zip
[26] http://www.atmel.com/Images/doc0856.pdf
[27] http://www.iar.com

- It was decided that the AVR would benefit from both indirect addressing (separately loading the address into e.g. XL and XH and then loading the content of this location into a working register) and direct addressing (one instruction loads the content of a specified memory location into a working register). Direct addressing results in fewer instruction words for 1-byte variables, while indirect addressing is more efficient when loading a 4-byte long integer.
- Atmel also decided to propagate both carry and zero flags in certain instructions so that 16- or 32-bit operations would be easier.
- Due to space constraints, there is no ADDI (16-bit constant addition without carry) but instead a SUBI (16-bit constant subtraction without carry) and an SBCI (16-bit constant subtraction with carry). Addition is accomplished as a subtraction by a negation of the actual value.
- They also made room for a non-destructive CPI (ComParison with Immediate) and non-destructive CPC (Compare with Carry). (20)

3.4 (Other) differences between ATmega and ATxmega

So far I have mostly discussed (some of) the common properties of the AVR family: CPU, working register, instruction set, and data memory space (well…). This is because I expect that they will have the greatest effect on the optimum programming style (for my test application). Please see the device and family datasheets for more information:

“ATmega164A/PA/324A/PA/644A/PA/1284/P Complete” (19) [28]

“Atmel AVR XMEGA AU Manual” (12) [29]

“ATxmega64A1U/128A1U Complete” (24) [30]

(And the Atmel documentation web site is a good place to find e.g. application notes. (25) [31])

There are also (great) differences between ATmega and ATxmega. In short: from a feature perspective, ATxmega is vastly superior to the ATmega with the following additions:

- DMA controller
- Event system
- AES and DES crypto engine
- High-speed DAC and ADC with higher resolution
- Lower power consumption
- 1.6V operation
- 32MHz maximum clock frequency (compared to 16 or 20MHz for ATmega)
- More advanced clock system and sleep modes
- More advanced physical ports
- Virtual port mapping of physical ports to the bit-operable I/O address area

[28] http://www.atmel.se/Images/Atmel-8272-8-bit-AVR-microcontroller-ATmega164A_PA-324A_PA-644A_PA-1284_P_datasheet.pdf
[29] http://www.atmel.se/Images/Atmel-8331-8-and-16-bit-AVR-Microcontroller-XMEGA-AU_Manual.pdf
[30] http://www.atmel.com/Images/Atmel-8385-8-and-16-bit-AVR-Microcontroller-ATxmega64A1U-ATxmega128A1U_datasheet.pdf
[31] http://atmel.no/webdoc/atmel.docs/atmel.docs.3.application.note.html

- More GPIO registers in the bit-operable I/O address area
- Multilevel interrupt controller
- EBI, External Bus Interface, for external SRAM or SDRAM
- Often “more of everything” compared to ATmega peripherals

The above and more information can be found in these documents:

“AVR XMEGA” (26) [32]

“Introducing a New Breed of Microcontrollers for 8/16-bit Applications” (27) [33]

“AVR1005: Getting started with XMEGA” (28) [34]

There’s also a new (alternative) addressing scheme with uniform placement of peripheral registers, so that one common driver can be used with module base pointer and sub-register displacement. This is such an important change, that it gets its own sub-section:

3.4.1 Alternative struct-based addressing mode

As the ATmega series grew with more families and the families were extended with additional devices, the I/O register layout(s) became more and more cluttered. This meant that static addressing was more or less necessary, which meant that sometimes the same code had to exist in as many copies as the used number of each peripheral type. It also required more work from Atmel to write and maintain the datasheets. Something had to be done.

Atmel’s solution to this was to create a limited number of series (named A – E) for their new ATxmega AVR. All devices within a series share a common set of properties and features and thus part of the datasheets could be maintained as one per series. The device-specific data remains in one datasheet per device type, which is why ATmega has one datasheet and ATxmega two.

Atmel also took the opportunity to bring order to the I/O register layout. Central to ATxmega is the “module”. I have failed to find an exact definition, but (29) [35] seems to call every separate function of the device a module. I pragmatic view is that whatever needs to be controlled resides in an adjoined set of registers that together constitute a module, exactly defined by a module type. Some functions exist in more than one instance and each one is internally exactly like the other modules of the same type. The instances are often(?) (always?) placed at an equal distance from the previous one. This means that you can access a particular I/O register by:

1. Finding the base address of the first instance
2. Adding (a multiple of) the inter-module offset to find the base address of the instance
3. Based on the module type definition (struct), find the memory pointer displacement

[32] http://www.atmel.com/Images/doc7925.pdf
[33] http://www.atmel.com/Images/doc7926.pdf
[34] http://www.atmel.com/Images/doc8169.pdf
[35] Available from: http://www.atmel.com/Images/doc8075.pd

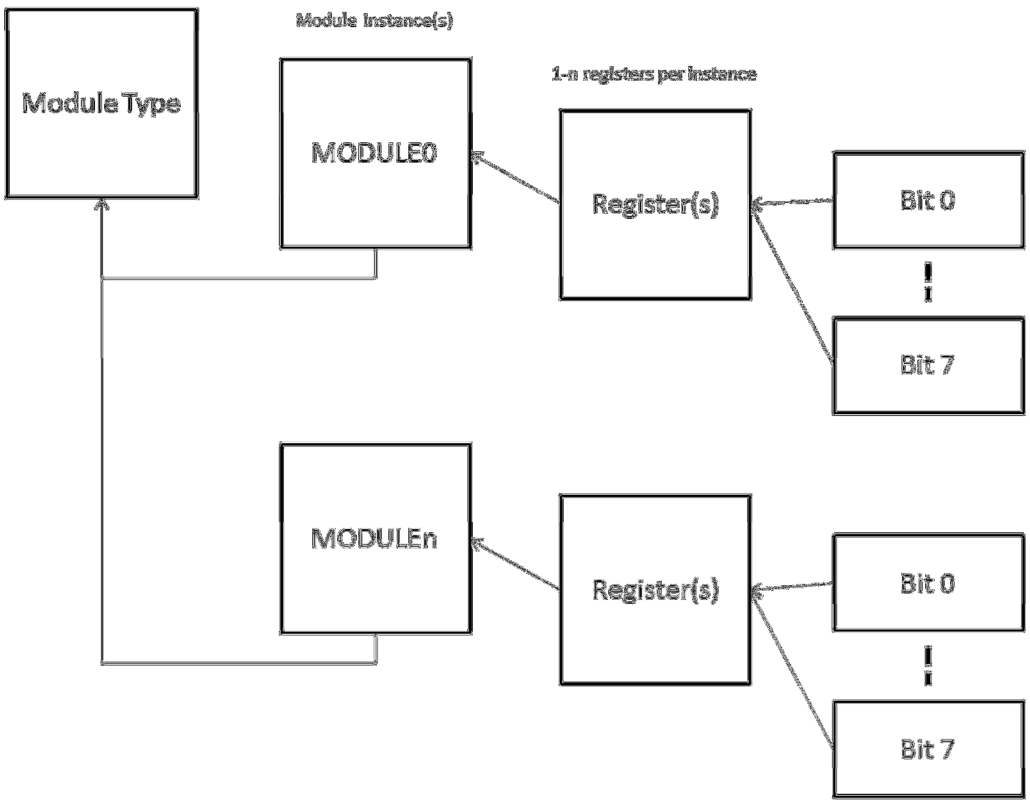


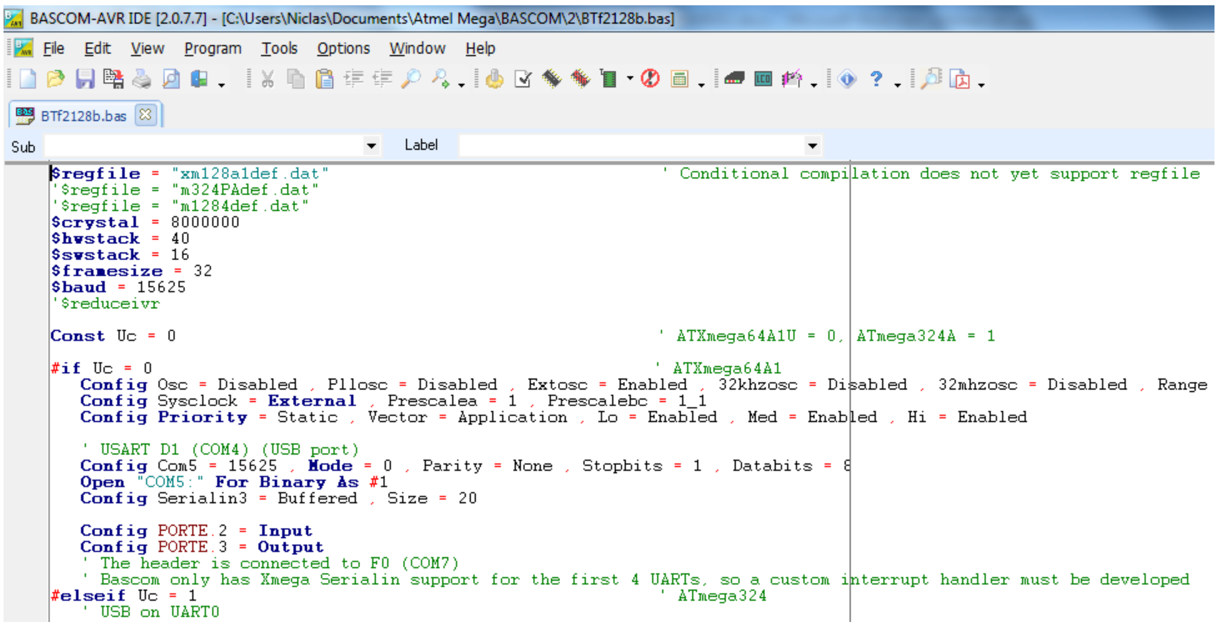
Figure 5: Module types, instances, registers, and bits

(The figure above is based on "AVR1000: Getting Started Writing C-code for XMEGA" (29) , p2)

4 Presentation of the IDEs

4.1 BASCOM-AVR

BASCOM-AVR is an IDE developed by a small Dutch company called MCS Electronics. It is designed for procedural programming in a Basic dialect similar to Visual Basic 6, henceforth referred to as VB. You can also use inline assembly intermixed with your high-level code or you can define your own assembly subroutines and functions. (A Basic subroutine is the same as a C void function.)


BASCOM-AVR IDE [2.0.7.7] - [C:\Users\Niclas\Documents\Atmel Mega\BASCOM\2\BTf2128b.bas]
File Edit View Program Tools Options Window Help
BTf2128b.bas
Sub
Label

```
$regfile = "xm128a1def.dat"                                   ' Conditional compilation does not yet support regfile
'$regfile = "m324PAdef.dat"
'$regfile = "m1284def.dat"
$crystal = 8000000
$hwstack = 40
$swstack = 16
$framesize = 32
$baud = 15625
'$reduceivr

Const Uc = 0                                                  ' ATXmega64A1U = 0, ATmega324A = 1

#if Uc = 0                                                    ' ATXmega64A1
   Config Osc = Disabled , Pllosc = Disabled , Extosc = Enabled , 32khzosc = Disabled , 32mhzosc = Disabled , Range
   Config Sysclock = External , Prescalea = 1 , Prescalebc = 1_1
   Config Priority = Static , Vector = Application , Lo = Enabled , Med = Enabled , Hi = Enabled

   ' USART D1 (COM4) (USB port)
   Config Com5 = 15625 , Mode = 0 , Parity = None , Stopbits = 1 , Databits = 8
   Open "COM5:" For Binary As #1
   Config Serialin3 = Buffered , Size = 20

   Config PORTE.2 = Input
   Config PORTE.3 = Output
   ' The header is connected to F0 (COM7)
   ' Bascom only has Xmega Serialin support for the first 4 UARTs, so a custom interrupt handler must be developed
#elseif Uc = 1                                                ' ATmega324
   ' USB on UART0
```

Figure 6: BASCOM-AVR developer view with a configuration code example

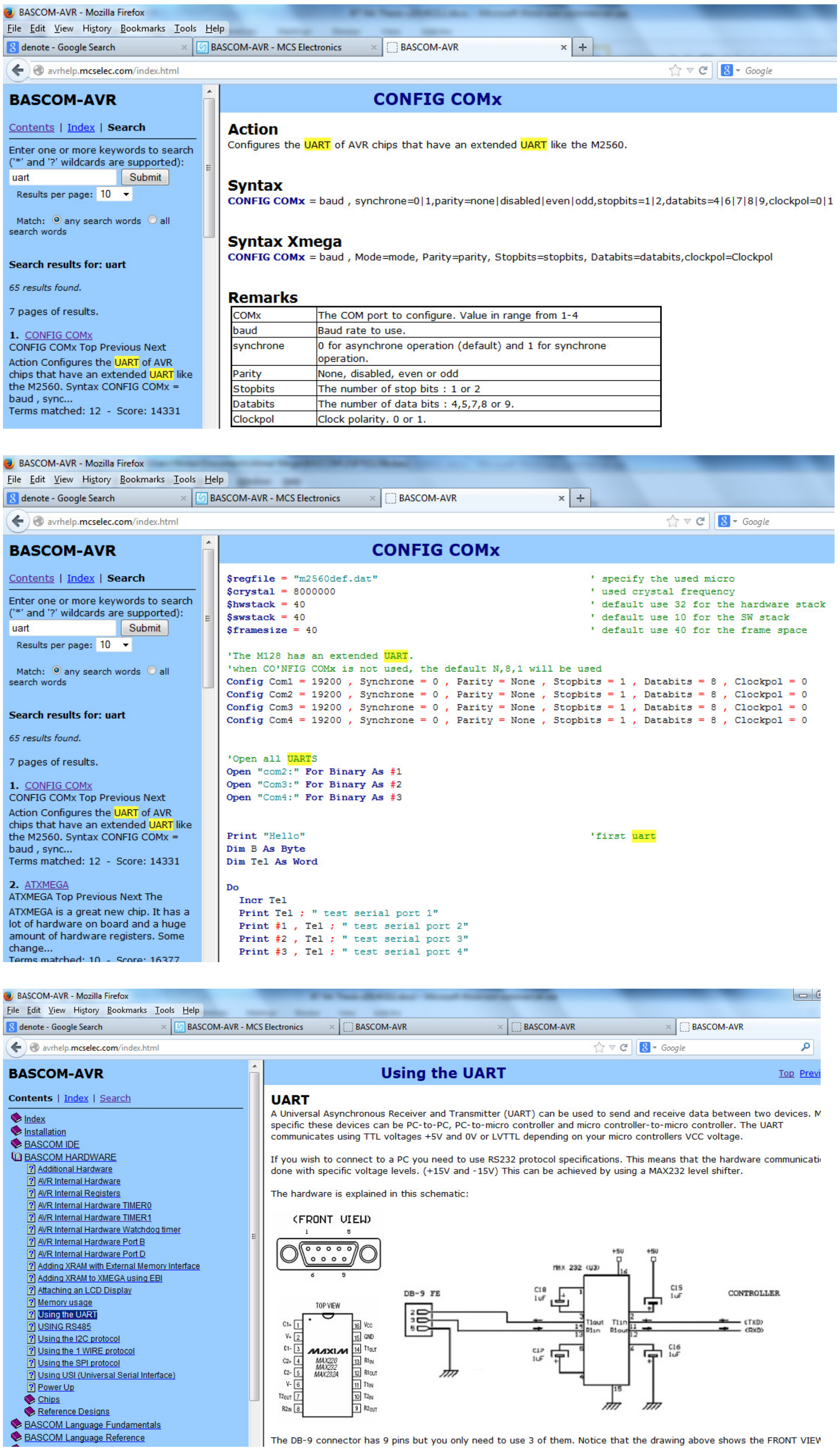
The concept of built-in commands is fundamental. They are hand-written assembly routines with the necessary auxiliary code for handling parameters and return values. There are commands both for configuration (like in the above screen dump) and subs/functions. The complete program is a stichwork of these hand-optimized commands and the non-optimized VB application code that "uses" and inter-connects them.

The company focused on functionality and ease of use, rather than ultimate performance (appendix A.1.1), which means that it doesn't have an optimizing compiler. There is support for most common microcontroller peripheral types out of the box. In the following screen dumps from the online help (30) [36] you get a glimpse of extended UART configuration command options, some code samples, and additional information. There's currently around 220 entries in the language reference, which gives a rough estimate of the number of built-in commands.

BASCOM-AVR has a simulator but no debugger. It outputs files that can easily be used for debugging with Visual Studio 6.

I end this very short presentation with a reference to the "Products" web page for BASCOM-AVR. Please look here for more details: (31) [37]

[36] http://avrhelp.mcselec.com/index.html
[37] http://www.mcselec.com/index.php?option=com_content&task=view&id=14&Itemid=41


BASCOM-AVR - Mozilla Firefox

avrhelp.mcselec.com/index.html

BASCOM-AVR

Contents | Index | Search

Enter one or more keywords to search ('*' and '?' wildcards are supported):

uart [Submit]

Results per page: 10

Match: [✓ any search words] all search words

Search results for: uart

65 results found.

7 pages of results.

1. CONFIG COMx
CONFIG COMx Top Previous Next
Action Configures the UART of AVR chips that have an extended UART like the M2560. Syntax CONFIG COMx = baud , sync...
Terms matched: 12 - Score: 14331

CONFIG COMx

Action
Configures the UART of AVR chips that have an extended UART like the M2560.

Syntax
CONFIG COMx = baud , synchrone=0|1,parity=none|disabled|even|odd,stopbits=1|2,databits=4|6|7|8|9,clockpol=0|1

Syntax Xmega
CONFIG COMx = baud , Mode=mode, Parity=parity, Stopbits=stopbits, Databits=databits,clockpol=Clockpol

Remarks

COMx	The COM port to configure. Value in range from 1-4
baud	Baud rate to use.
synchrone	0 for asynchrone operation (default) and 1 for synchrone operation.
Parity	None, disabled, even or odd
Stopbits	The number of stop bits : 1 or 2
Databits	The number of data bits : 4,5,7,8 or 9.
Clockpol	Clock polarity. 0 or 1.

BASCOM-AVR - Mozilla Firefox

avrhelp.mcselec.com/index.html

BASCOM-AVR

Contents | Index | Search

Enter one or more keywords to search ('*' and '?' wildcards are supported):

uart [Submit]

Results per page: 10

Match: [✓ any search words] all search words

Search results for: uart

65 results found.

7 pages of results.

1. CONFIG COMx
CONFIG COMx Top Previous Next
Action Configures the UART of AVR chips that have an extended UART like the M2560. Syntax CONFIG COMx = baud , sync...
Terms matched: 12 - Score: 14331

2. ATXMEGA
ATXMEGA Top Previous Next The ATXMEGA is a great new chip. It has a lot of hardware on board and a huge amount of hardware registers. Some change...
Terms matched: 10 - Score: 16377

CONFIG COMx

```
$regfile = "m2560def.dat"                                ' specify the used micro
$crystal = 8000000                                       ' used crystal frequency
$hwstack = 40                                            ' default use 32 for the hardware stack
$swstack = 40                                            ' default use 10 for the SW stack
$framesize = 40                                          ' default use 40 for the frame space

'The M128 has an extended UART.
'when CO'NFIG COMx is not used, the default N,8,1 will be used
Config Com1 = 19200 , Synchrone = 0 , Parity = None , Stopbits = 1 , Databits = 8 , Clockpol = 0
Config Com2 = 19200 , Synchrone = 0 , Parity = None , Stopbits = 1 , Databits = 8 , Clockpol = 0
Config Com3 = 19200 , Synchrone = 0 , Parity = None , Stopbits = 1 , Databits = 8 , Clockpol = 0
Config Com4 = 19200 , Synchrone = 0 , Parity = None , Stopbits = 1 , Databits = 8 , Clockpol = 0


'Open all UARTS
Open "com2:" For Binary As #1
Open "Com3:" For Binary As #2
Open "Com4:" For Binary As #3


Print "Hello"                                            'first uart
Dim B As Byte
Dim Tel As Word

Do
  Incr Tel
  Print Tel ; " test serial port 1"
  Print #1 , Tel ; " test serial port 2"
  Print #2 , Tel ; " test serial port 3"
  Print #3 , Tel ; " test serial port 4"
```

BASCOM-AVR - Mozilla Firefox

avrhelp.mcselec.com/index.html

BASCOM-AVR

Contents | Index | Search

Using the UART

Top Previ

UART
A Universal Asynchronous Receiver and Transmitter (UART) can be used to send and receive data between two devices. M specific these devices can be PC-to-PC, PC-to-micro controller and micro controller-to-micro controller. The UART communicates using TTL voltages +5V and 0V or LVTTL depending on your micro controllers VCC voltage.

If you wish to connect to a PC you need to use RS232 protocol specifications. This means that the hardware communicati done with specific voltage levels. (+15V and -15V) This can be achieved by using a MAX232 level shifter.

The hardware is explained in this schematic:

The DB-9 connector has 9 pins but you only need to use 3 of them. Notice that the drawing above shows the FRONT VIEW

Figure 7: BASCOM-AVR UART help file examples

4.1.1 User forum

The BASCOM-AVR user forum is located at the company's web site www.mcselec.com. (32) [38] It is active and a good place to get in touch with both employees and independent developers. Apart from posting in the forum, users can also share working code and publish application notes that typically present a complete design or a major piece of code.

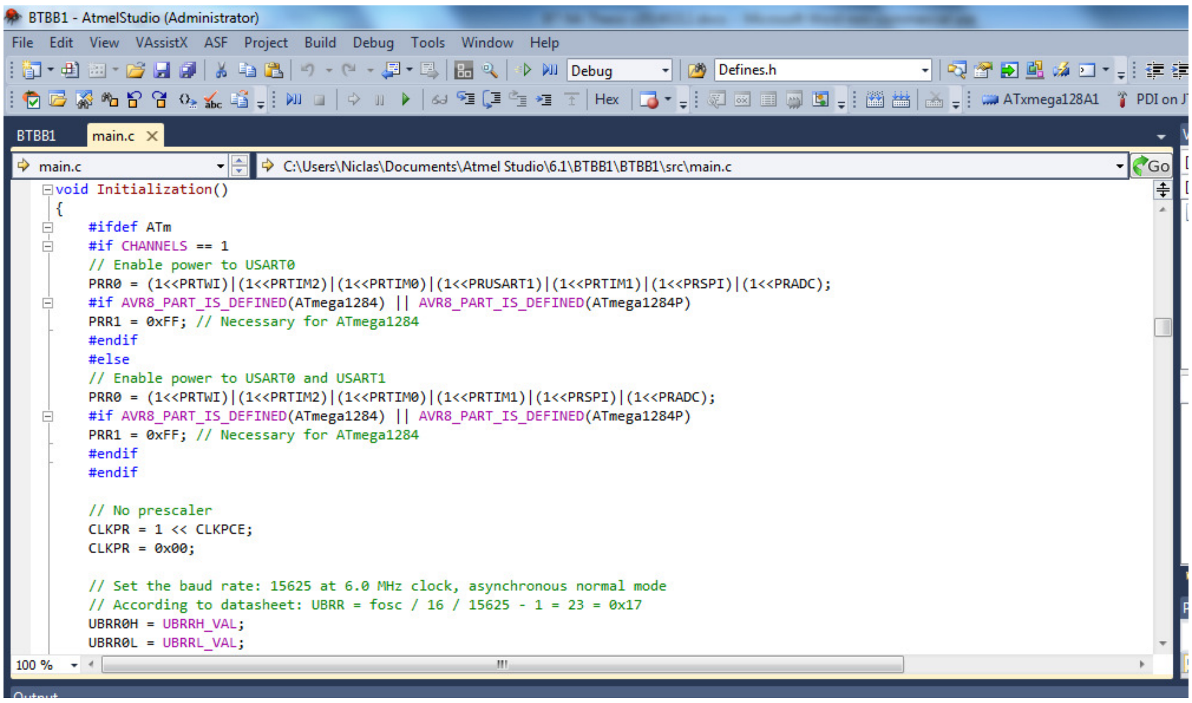
4.1.2 Price

There is a free version (usually lagging some releases) that supports almost all features up to 4 kB of compiled code. The full commercial version costs €89 at the company's web site.

4.2 Atmel Studio 6

Atmel Studio 6.x is the company's second release based on Microsoft Visual Studio. It has support not only for all 8-bit AVRs, but also for AVR32 and Atmel's ARM devices. At its heart is AVR-GCC (Gnu Compiler Collection), which has a powerful optimizing compiler. I won't go into AVR-GCC, but you can find detailed information about it here: (33) [39] (34) [40]

Two other useful documents are:
The GCC (GNU Compiler Collection) manual on optimization options (35) [41]
The AVR-Libc manual (36) [42]



```
void Initialization()
{
    #ifdef ATm
    #if CHANNELS == 1
    // Enable power to USART0
    PRR0 = (1<<PRTWI)|(1<<PRTIM2)|(1<<PRTIM0)|(1<<PRUSART1)|(1<<PRTIM1)|(1<<PRSPI)|(1<<PRADC);
    #if AVR8_PART_IS_DEFINED(ATmega1284) || AVR8_PART_IS_DEFINED(ATmega1284P)
    PRR1 = 0xFF; // Necessary for ATmega1284
    #endif
    #else
    // Enable power to USART0 and USART1
    PRR0 = (1<<PRTWI)|(1<<PRTIM2)|(1<<PRTIM0)|(1<<PRTIM1)|(1<<PRSPI)|(1<<PRADC);
    #if AVR8_PART_IS_DEFINED(ATmega1284) || AVR8_PART_IS_DEFINED(ATmega1284P)
    PRR1 = 0xFF; // Necessary for ATmega1284
    #endif
    #endif

    // No prescaler
    CLKPR = 1 << CLKPCE;
    CLKPR = 0x00;

    // Set the baud rate: 15625 at 6.0 MHz clock, asynchronous normal mode
    // According to datasheet: UBRR = fosc / 16 / 15625 - 1 = 23 = 0x17
    UBRR0H = UBRRH_VAL;
    UBRR0L = UBRRL_VAL;
```

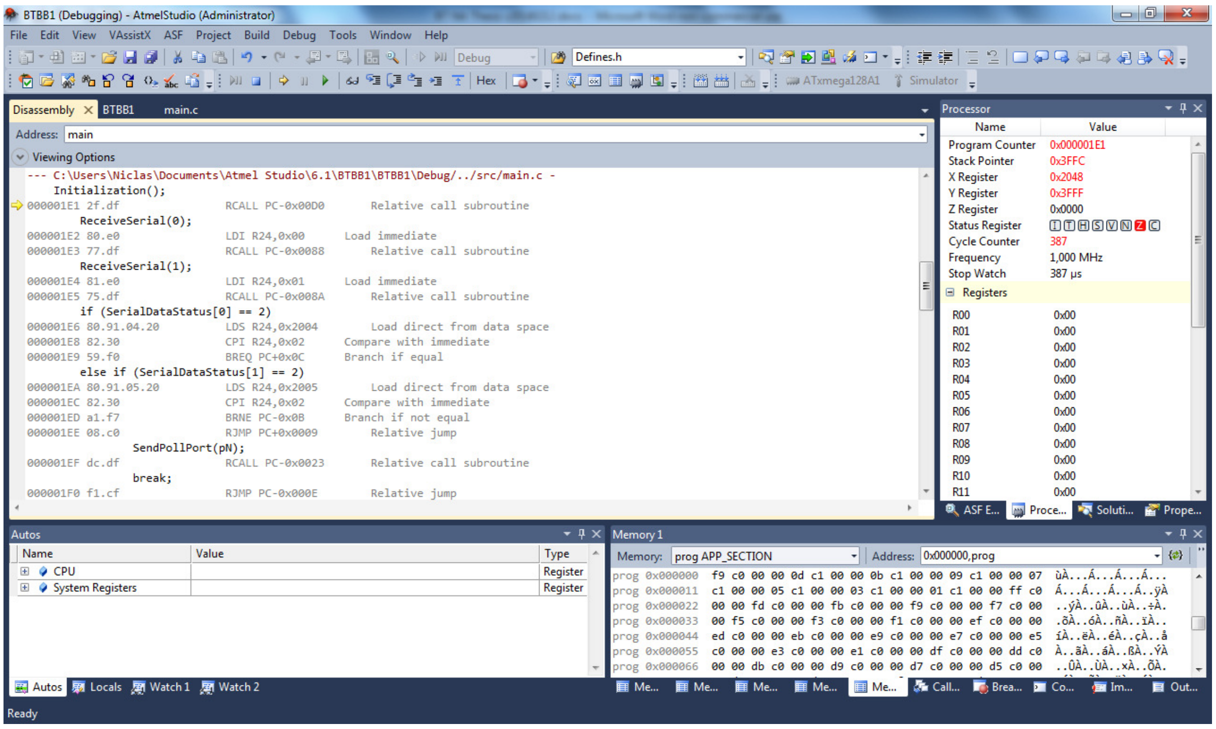
BTBB1 - AtmelStudio (Administrator)
File Edit View VAssistX ASF Project Build Debug Tools Window Help
Debug
Defines.h
Hex
ATxmega128A1
PDI on J
BTBB1
main.c
main.c
C:\Users\Niclas\Documents\Atmel Studio\6.1\BTBB1\BTBB1\src\main.c
Go
100 %

Figure 8: Atmel Studio 6.1 developer view

In Atmel Studio you can develop in Assembly, C, and C++. For detailed information, please see the Atmel Studio 6 web site: (6) [43]

[38] http://www.mcselec.com/index2.php?option=com_forum&Itemid=59
[39] http://gcc.gnu.org/wiki/avr-gcc
[40] http://www.avrfreaks.net/wiki/index.php/Documentation:AVR_GCC/AVR_GCC_Tool_Collection
[41] http://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html#Optimize-Options
[42] http://www.nongnu.org/avr-libc/user-manual/

You can simulate your program in either high-level or disassembly mode and you can also attach a debugger to your development board / custom PCB and verify real program behavior:


BTBB1 (Debugging) - AtmelStudio (Administrator)

Disassembly — Address: main

```
--- C:\Users\Niclas\Documents\Atmel Studio\6.1\BTBB1\BTBB1\Debug/../src/main.c -
    Initialization();
000001E1 2f.df                RCALL PC-0x00D0       Relative call subroutine
        ReceiveSerial(0);
000001E2 80.e0                LDI R24,0x00      Load immediate
000001E3 77.df                RCALL PC-0x0088       Relative call subroutine
        ReceiveSerial(1);
000001E4 81.e0                LDI R24,0x01      Load immediate
000001E5 75.df                RCALL PC-0x008A       Relative call subroutine
        if (SerialDataStatus[0] == 2)
000001E6 80.91.04.20          LDS R24,0x2004        Load direct from data space
000001E8 82.30                CPI R24,0x02      Compare with immediate
000001E9 59.f0                BREQ PC+0x0C      Branch if equal
        else if (SerialDataStatus[1] == 2)
000001EA 80.91.05.20          LDS R24,0x2005        Load direct from data space
000001EC 82.30                CPI R24,0x02      Compare with immediate
000001ED a1.f7                BRNE PC-0x0B      Branch if not equal
000001EE 08.c0                RJMP PC+0x0009        Relative jump
            SendPollPort(pN);
000001EF dc.df                RCALL PC-0x0023       Relative call subroutine
            break;
000001F0 f1.cf                RJMP PC-0x000E        Relative jump
```

Processor

Name	Value
Program Counter	0x000001E1
Stack Pointer	0x3FFC
X Register	0x2048
Y Register	0x3FFF
Z Register	0x0000
Status Register	I T H S V N Z C
Cycle Counter	387
Frequency	1,000 MHz
Stop Watch	387 µs
Registers	
R00	0x00
R01	0x00
R02	0x00
R03	0x00
R04	0x00
R05	0x00
R06	0x00
R07	0x00
R08	0x00
R09	0x00
R10	0x00
R11	0x00

Autos

Name	Value	Type
CPU		Register
System Registers		Register

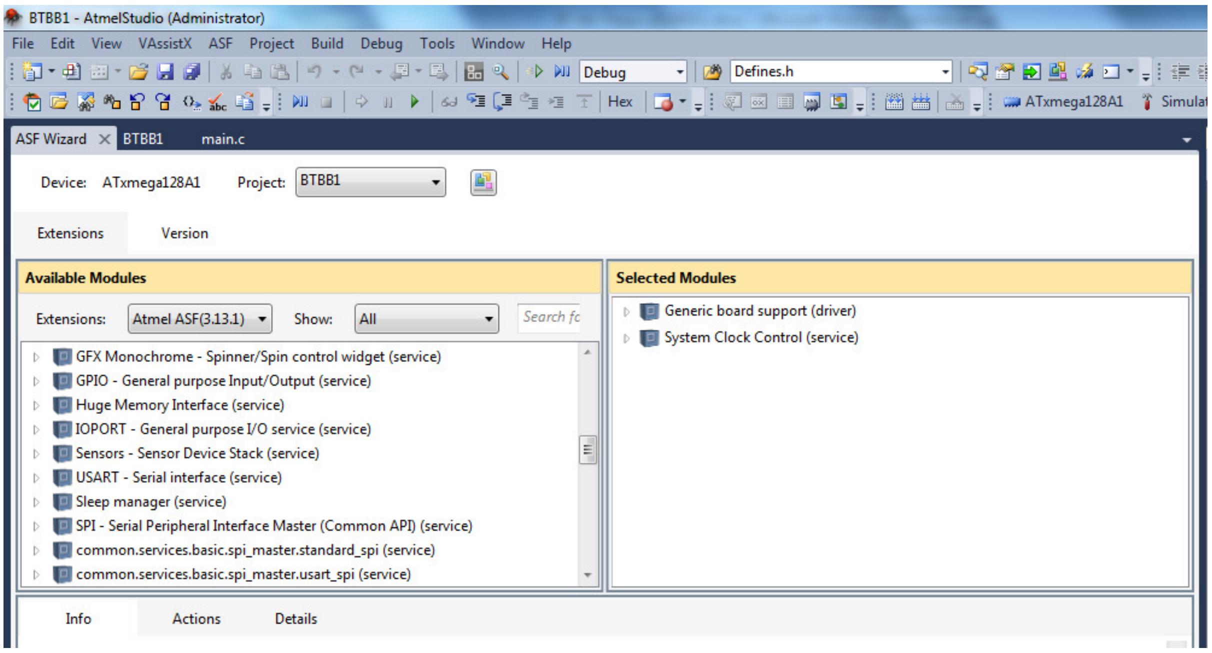
Memory 1 — Memory: prog APP_SECTION — Address: 0x000000,prog

```
prog 0x000000  f9 c0 00 00 0d c1 00 00 0b c1 00 00 09 c1 00 00 07  ùÀ...Á...Á...Á...
prog 0x000011  c1 00 00 05 c1 00 00 03 c1 00 00 01 c1 00 00 ff c0  Á...Á...Á...Á..ÿÀ
prog 0x000022  00 00 fd c0 00 00 fb c0 00 00 f9 c0 00 00 f7 c0 00  ..ýÀ..ûÀ..ùÀ..÷À.
prog 0x000033  00 f5 c0 00 00 f3 c0 00 00 f1 c0 00 00 ef c0 00 00  .õÀ..óÀ..ñÀ..ïÀ..
prog 0x000044  ed c0 00 00 eb c0 00 00 e9 c0 00 00 e7 c0 00 00 e5  íÀ..ëÀ..éÀ..çÀ..å
prog 0x000055  c0 00 00 e3 c0 00 00 e1 c0 00 00 df c0 00 00 dd c0  À..ãÀ..áÀ..ßÀ..ÝÀ
prog 0x000066  00 00 db c0 00 00 d9 c0 00 00 d7 c0 00 00 d5 c0 00  ..ÛÀ..ÙÀ..×À..ÕÀ.
```

Ready

Figure 9: Atmel Studio 6.1 debugging

ASF (Atmel Software Foundation, formerly AVR SF) is a repository for standardized drivers and example projects that demonstrate some Atmel Evaluation kit feature.


BTBB1 - AtmelStudio (Administrator)

ASF Wizard

Device: ATxmega128A1 Project: BTBB1

Available Modules — Extensions: Atmel ASF(3.13.1) Show: All

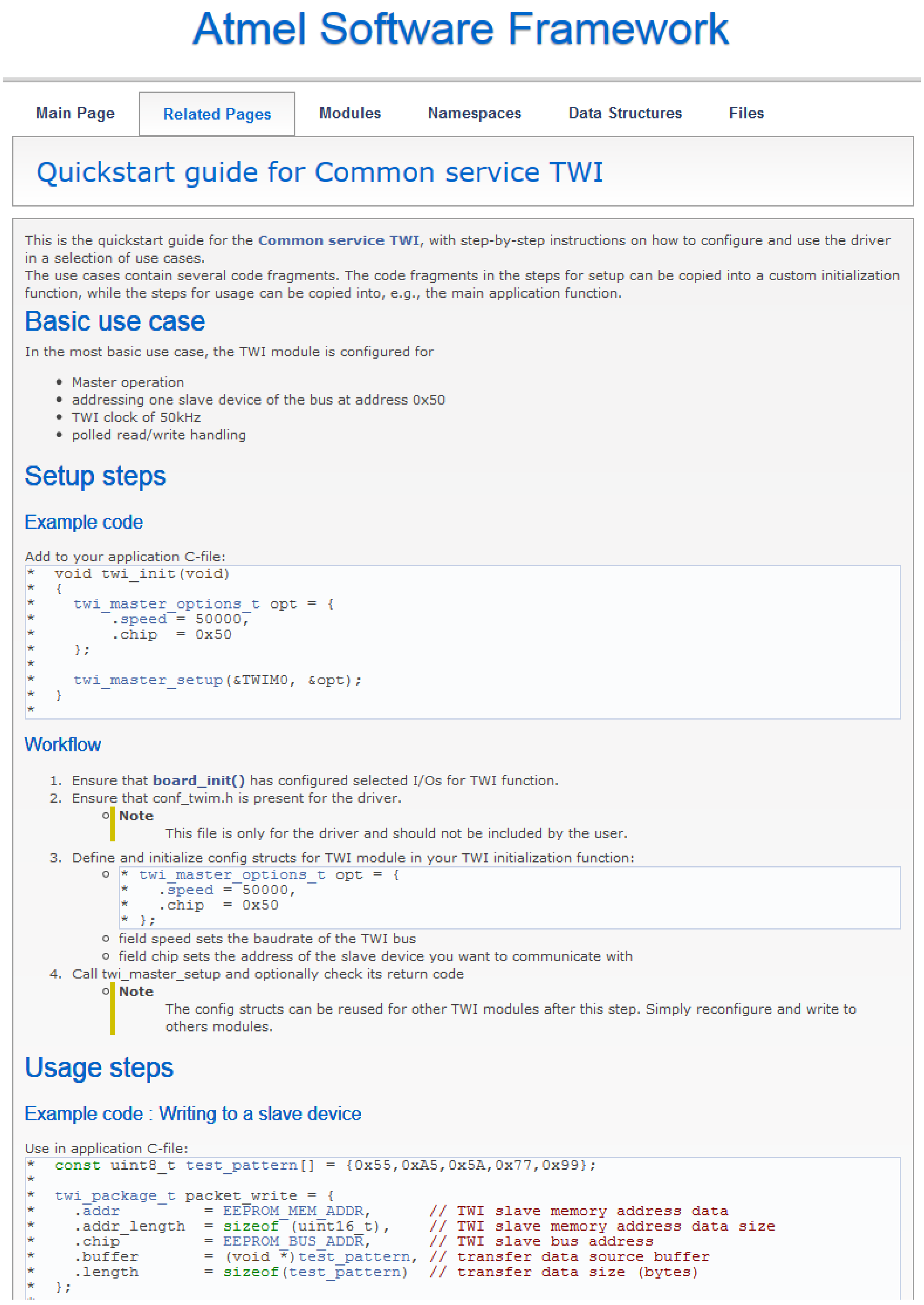
- GFX Monochrome - Spinner/Spin control widget (service)
- GPIO - General purpose Input/Output (service)
- Huge Memory Interface (service)
- IOPORT - General purpose I/O service (service)
- Sensors - Sensor Device Stack (service)
- USART - Serial interface (service)
- Sleep manager (service)
- SPI - Serial Peripheral Interface Master (Common API) (service)
- common.services.basic.spi_master.standard_spi (service)
- common.services.basic.spi_master.usart_spi (service)

Selected Modules

- Generic board support (driver)
- System Clock Control (service)

Figure 10: ASF Wizard in Atmel Studio 6.1

[43] http://www.atmel.se/microsite/atmel_studio6/


Atmel Software Framework

Main Page | Related Pages | Modules | Namespaces | Data Structures | Files

Quickstart guide for Common service TWI

This is the quickstart guide for the **Common service TWI**, with step-by-step instructions on how to configure and use the driver in a selection of use cases.
The use cases contain several code fragments. The code fragments in the steps for setup can be copied into a custom initialization function, while the steps for usage can be copied into, e.g., the main application function.

Basic use case

In the most basic use case, the TWI module is configured for

- Master operation
- addressing one slave device of the bus at address 0x50
- TWI clock of 50kHz
- polled read/write handling

Setup steps

Example code

Add to your application C-file:

```
*   void twi_init(void)
*   {
*     twi_master_options_t opt = {
*         .speed = 50000,
*         .chip  = 0x50
*     };
*
*     twi_master_setup(&TWIM0, &opt);
*   }
*
```

Workflow

1. Ensure that **board_init()** has configured selected I/Os for TWI function.
2. Ensure that conf_twim.h is present for the driver.
   - **Note**
     This file is only for the driver and should not be included by the user.
3. Define and initialize config structs for TWI module in your TWI initialization function:

   -
     ```
     * twi_master_options_t opt = {
     *    .speed = 50000,
     *    .chip  = 0x50
     * };
     ```
   - field speed sets the baudrate of the TWI bus
   - field chip sets the address of the slave device you want to communicate with
4. Call twi_master_setup and optionally check its return code
   - **Note**
     The config structs can be reused for other TWI modules after this step. Simply reconfigure and write to others modules.

Usage steps

Example code : Writing to a slave device

Use in application C-file:

```
*   const uint8_t test_pattern[] = {0x55,0xA5,0x5A,0x77,0x99};
*
*   twi_package_t packet_write = {
*     .addr         = EEPROM_MEM_ADDR,      // TWI slave memory address data
*     .addr_length  = sizeof (uint16_t),    // TWI slave memory address data size
*     .chip         = EEPROM_BUS_ADDR,      // TWI slave bus address
*     .buffer       = (void *)test_pattern, // transfer data source buffer
*     .length       = sizeof(test_pattern)  // transfer data size (bytes)
*   };
```

Figure 11: ASF online documentation example (quickstart guide)

For more information, please see the following two documents:

"AVR4029: Atmel Software Framework - Getting Started" (37) [44]

"AVR4030: AVR Software Framework - Reference Manual" (38) [45]

4.2.1 History: AVR Studio 4 & 5, WinAVR, and Eclipse

Please see appendix A.5.3 for information about Atmel Studio 6's history that might shed some light on its current state.

4.2.2 User forum

Atmel's main user forum for their AVR offering is www.avrfreaks.net. (39) [46] It is active and a mix of independent developers and a number of more or less official employees. Users can also create "projects" that typically contain a working application or a driver.

4.2.3 Price

AVR Studio 4 & 5 and Atmel Studio 6 are free for registered users.

4.2.4 (Inline) assembly documentation

I'm just including these documents here for future reference:

- "AVR Assembler User Guide" (40) [47]
- "Atmel AT1886: Mixing Assembly and C with AVRGCC" (41) [48] [49]
- "AVR000: Register and Bit-Name Definitions for the 8-bit AVR Microcontroller" (42) [50]
- "AVR001: Conditional Assembly and portability macros" (43) [51]

[44] http://www.atmel.com/Images/Atmel-8431-8-and32-bit-Microcontrollers-AVR4029-Atmel-Software-Framework-User-Guide_Application-Note.pdf
[45] http://www.atmel.com/Images/doc8432.pdf
[46] http://www.avrfreaks.net/
[47] www.atmel.com/images/doc1022.pdf
[48] http://www.atmel.se/Images/doc42055.pdf
[49] http://www.atmel.se/Images/AT1886.zip
[50] http://www.atmel.com/Images/doc0931.pdf
[51] http://www.atmel.com/Images/doc2550.pdf

5 BASCOM-AVR analysis

I started with a previously developed piece of BASCOM-AVR VB code used for serial communication between a PC monitoring application and an AVR microcontroller. I first did most of the development on the ATmega and then added the ATxmega with conditional compilation.

5.1 Serial communication analysis test log

5.1.1 VB high-level code implementations

Please note: The original disassemblies were made on versions with "Config Com1 = 15625..." and without "Config Portd.0" = "Input and Config Portd.1 = Output". The comments are on ATmega324A. In this section, all code sizes are in bytes.

Step	Atmega 324A	ATxmega 128A1	Action	Comment
BA1a	1006	1720	Local variable Uartsendbyte in Sendpollport sub and Senderror sub. Printbin command used for each USART sending	
BA1b	956	1670	Global variable Uartsendbyte. Printbin command used for each USART sending	Simply by using a global variable instead of a local one, we save 50 bytes of compiled code (5%). See disassembly BA2_324_dis_.dump_b.txt, ReceiveSerial sub, for the operations concerning creating three local variables on the frame and pointers to them on the software stack. (And, at the end of the sub, the frame and software stack pointers must be restored.)
BA1c	938	1670	Changed Config Com1 = 15625 , Synchrone = 0 , Parity = None , Stopbits = 1 , Databits = 8 , Clockpol = 0 to Config Com1 = Dummy , Synchrone = 0 , Parity = None , Stopbits = 1 , Databits = 8 , Clockpol = 0. Global variable Uartsendbyte.	Assumingly this change removes the duplicate mentioned further down in BA2.
BA2	896	1596	Created gosub Prbin for Printbin command. Global variable Uartsendbyte.	Moving the Printbin commands used for each byte to a gosub with a common Printbin command saves us another 42 bytes.

Figure 12: BASCOM-AVR iterations 1-2

Before we continue, let's take a look at the BA2 ATmega 324A disassembly:

- The actual program starts at 0x7C (after the interrupt vector). By default, an initialization phase is run:
- It sets the stack pointer to the end of RAM.
- Register Y (pair R28 & R29) is used as the software stack pointer.
- Pair R4 & R5 is used as the frame pointer.
- Register MCUSR (reset flags) is cleared except for WDRF (watchdog refresh).
- Watchdog is disabled.
- The entire internal SRAM is cleared (zeroed). This means that all global variables are automatically initialized to 0, so my current sub Initialization is unnecessary.

This initialization can be omitted by using $NOINIT at the beginning of the .bas file.

Then follow the setup of USART0, clearing of special register R6, and enabling of RX0 interrupt. For some reason the USART0 setup is done twice. According to the datasheet (p180) (19) [52], this shouldn't be necessary. (This turned out to be a programmer mistake, partly due to incomplete documentation. See comment on BA1c above.)

Apart from the clearing of the global variables in sub Initialization, I was surprised to see that the compiler clears R24 for each and every variable. The same can be seen at the beginning of the Receiveserial sub. A similar case is 0x17C & 0x17E vs. 0x182 & 0x184.

Another peculiarity is that the compiler doesn't check if the jump destination is another jump (e.g. in nested if statements). See the main program loop and the Receiveserial sub.

The routines at 0x30A to 0x30E and 0x31C to 0x324 are not used. They are probably part of frequently used code that's included in one standardized package for simplicity. I'll come back to them at the end of the BASCOM-AVR analysis and subtract their size from the final comparison.

It is worth noticing that turning optimization on produces no code size difference in the BA2 code. It's still 896 bytes. I didn't disassemble to see if there are code changes.

Let's try using array-based sending instead of byte-wise sending:

Step	**Atmega 324A**	**ATXmega 128A1**	**Action**	**Comment**
BA3	880	1580	Changed from global Uartsendbyte to global Serialoutdata(20) array and Serialoutcount. Subs Sendpollport and Senderror fill this array, update the counter, and finally make one call to Prbin. Now the Prbin gosub contains the following command: Printbin #1 , Serialoutdata(1) , Serialoutcount	At first I couldn't get this to work, neither using Serialoutcount nor a fixed value (6). I often got the correct response, but sometimes several bytes with the value 0. The correct syntax according to documentation is with "; Serialcount", but that would sometimes send additional bytes. As the saving with this version would be only 6 bytes (a total of 890) with a 20 + 1 byte increase in RAM, I didn't look closer into this until later: By changing to ", Serialoutcount" it seems to work properly and the size becomes 880.

Figure 13: BASCOM-AVR iteration 3

As mentioned in the comment, I didn't continue building on this branch as the RAM increase surpasses the program code saving.

[52] http://www.atmel.se/Images/Atmel-8272-8-bit-AVR-microcontroller-ATmega164A_PA-324A_PA-644A_PA-1284_P_datasheet.pdf

I next investigated different uses of global and local variables and (byref) parameters:

Step	Atmega 324A	ATXmega 128A1	Action	Comment
BA4a	878		BA2 is used as the basis for BA4. Comment out the initialization of global variables to 0.	
BA4b	862		In sub Receiveserial, omit local Serialwaiting and use "If Ischarwaiting(#1) = 1 Then".	(10 bytes saved by "If Ischarwaiting(#1) = 1 Then". 6 bytes saved by removing local byte Serialwaiting.)
BA4c	868		In sub Receiveserial, break out "Serialdata(receivecounter) = Serialbyte" and place it in new sub with byref parameter.	
BA4d	836		Convert sub Receiveserial's local Serialbyte to global Serialbyte and remove the byref parameter.	
BA4e	836		Convert sub Insertserialdata to a gosub.	No change as a parameterless sub is in fact a gosub.
BA4f	792	1492	Convert sub Receiveserial's local Continue to global Continue.	This is the final BA4 version.
BA4g	830		Added a local Test byte to sub Receiveserial. This variable isn't used.	We just saw that a local byte requires 6 bytes of program code, so the "fixed cost" of using the first local byte is 830 - 792 - 6 = 32 bytes. As we'll see from the disassembly of BA5, 22 of these saved bytes come from the two sections that make room on the frame for local variables, of which 10 refer to unused code.

Figure 14: BASCOM-AVR iteration 4

We now have a figure for the cost of using local variables, both in terms of an offset and a variable "fee" for each one. If you want to optimize your BASCOM-AVR development, you should only use parameters and locals when there is a good reason to do so. This is quite contrary to the general "rule" of using no global variables at all unless absolutely necessary. I'll return to this later in this section.

To proceed, we need to look at improving the structure of the program itself:

Step	Atmega 324A	ATXmega 128A1	Action	Comment
BA5a	754		Changed all subs to gosubs. Revised the main loop and gosub Receiveserial. No longer keep the program looping inside gosub Receiveserial after start token until end token.	
BA5b	748	1444	Removed global Continue. Removed global Receivedata. Renamed global Serialdataready to Serialdatastatus (value 1). Removed global Serialcommandfound. Its meaning incorporated in Serialdatastatus (value 2).	

Figure 15: BASCOM-AVR iteration 5

At this point, I assumed that BA5b 748 is the furthest I could improve this code without resorting to even more exotic programming. (It turned out I was very wrong.)
So far I had "only" replaced a while loop with a goto and used global variable aliasing to mimic local

variable use at global variable cost. As we saw earlier, the assembly implementation of nested if clauses end with jumps to the outer if clause's ending jump and so on. This doesn't lead to an increase in compiled code, but you lose a few clock cycles. This should be taken care of by the compiler, but it is possible to replace the if-else-end if and Select case-case-case else-end select with gotos to labels, but I won't do it in VB code for fear of cluttering up the code completely.

Let's sum up:

The interrupt vector takes 124 bytes compiled code. For simplicity's sake, let's say that the default initialization (except USART setup) takes another 58 bytes. In other words, the application-specific code starts at 0xB6 (182 dec). The initial (worst) design required 1006 bytes, netting at 824 bytes application code. BA5b 748 has a net application size of 566 bytes. This is a reduction of (824 - 566) / 824 = 31%.

Now I'll see how much more I can improve this on the assembler level.

5.1.2 Looking for inline assembly improvements to standard BASCOM-AVR funtionality

5.1.2.1 Receiveserial gosub

As mentioned before, nested if clauses result in jump to jump to destination rather than jump to destination. Three jumps could be modified so they go directly to destination, but this is hardly worth the conversion into inline assembler. The only real reason to do this would be if it would enable us to realize a potential saving in Insertserialdata gosub.

5.1.2.2 Insertserialdata gosub

Change to non-autoincrementing (AC 90). This enables removal of the next operation.

```
0000019C   AD 90     ld     r10, X+    R10 = global Receivecounter, X post-increment
```

Remove this:
```
000001A2   B1 E0                ldi       r27, 0x01    ; 1             ...
```

So long as the entire array SERIALDATA resides within the same RAM address LSB (Least Significant Byte), this RAM address MSB operation is unnecessary.

```
000001A6   BB 1D               adc       r27, r11                        ...
```

Total potential saving: two 1-word operations = 4 bytes. Is it possible to realize this by using inline assembler? Yes, if we can be sure that R24, X, R10, and R11 can be used freely without pushing and popping them on the stack.

The Bascom register convention [53] doesn't mention any of these, so we should be safe. (Please see appendix A.5.1):

Just looking at the compiled code, it seems like Bascom is generally only using / tying up the "other registers" inside Bascom commands.

[53] Look up ”Mixing ASM and BASIC” at http://avrhelp.mcselec.com/index.html

Two examples from the BA2 disassembly's Receiveserial:

```
 144:    81 e0        ldi     r24, 0x01   ; 1       Local Serialbyte on frame
 146:    0e 94 93 01  call    0x326       ;  0x326   ...
??14a:   81 e0        ldi     r24, 0x01   ; 1       Local Serialwaiting on frame
 14c:    0e 94 93 01  call    0x326       ;  0x326   ...
??150:   81 e0        ldi     r24, 0x01   ; 1       Local Continue on frame
 152:    0e 94 93 01  call    0x326       ;  0x326  ...

 17c:    aa 81        ldd     r26, Y+2    ; 0x02     X points to local Serialwaiting
 17e:    bb 81        ldd     r27, Y+3    ; 0x03    ...
 180:    8c 93        st      X, r24                Local Serialwaiting = R24 (return from ISCHARWAITING)
??182:   aa 81        ldd     r26, Y+2    ; 0x02    X points (again) to local Serialwaiting
??184:   bb 81        ldd     r27, Y+3    ; 0x03    ...
 186:    0c 91        ld      r16, X                R16 = local Serialwaiting
```

Similarly, R10 and R11 are only used in the Insertserialdata gosub, so it would seem safe, but how can we know that this is true?

Please see appendix A.5.1 for user forum postings on this topic. Apparently, BASCOM-AVR could be seen as a stitch-work of handwritten assembly code blocks (i.e. the commands) interconnected with compiled VB statements. As far as you stay away from the reserved registers, you don't have to take any other precautions when writing inline assembly. It's only in interrupt routines that you must remember to save SREG and any used registers to stack. The downside is that the interconnections are completely non-optimized (e.g. the repeated assignment of the same value to the same register and the jumps to jumps). It shall be interesting to compare the BASCOM-AVR compiled code to the one generated by Atmel Studio.

"Mixing ASM and BASIC" in the online help: (30) [54] contains instructions on how you write inline assembly and creates custom subroutines and functions. You can copy from the assembly versions of the built-in commands in the LIB installation folder.

5.1.2.3 Assembly improvements to the USART send routines

The ATmega324A datasheet code example uses sbis to check if the USART data register is ready to be written to.

```
USART_Transmit:
sbis UCSRnA,UDREn
rjmp USART_Transmit
```

However, as sbis can only operate on the lowest 0x1F (32) registers, this is actually a typo. In other words, the BASCOM code is optimal:

```
USART_Transmit:
lds         r0, 0xC0    ; UCSR0A
sbrs        r0, 5
rjmp        .-8         ;  USART_Transmit:
```

If we want to keep the current USART send functionality, there are no possible improvements to the BASCOM commands. If we are prepared to alter the functionality, we could write the entire USART send code as custom inline assembly. This will be done in versions BA7 and BA8, but first another high-level language improvement:

[54] http://avrhelp.mcselec.com/index.html

5.1.3 Sendpollport and Senderror gosubs, Prbin command

I thought that the Printbin command doesn't support an absolute parameter value, as this isn't mentioned in the documentation. (Only variable-based parameters are covered.) However, as I thought that Sendpollport, Senderror, and Prbin would be great candidates for custom assembly, I on a whim decided to try using Printbin with an absolute value. Judging by the disassembly, it looks like this works, which brings us to BA6:

Step	Atmega 324A	ATXmega 128A1	Action	Comment

BA6a	730	1444	Remove Prbin gosub.
			Change Sendpollport and Senderror like this:
			
			Sendpollport:
			  Printbin #1 , 254
			  Printbin #1 , 242
			  Uartsendbyte = 1
			  Printbin #1 , Uartsendbyte
			  Uartsendbyte = 2
			  Printbin #1 , Uartsendbyte
			  Uartsendbyte = 3
			  Printbin #1 , Uartsendbyte
			  Printbin #1 , 255
			Return
			
			Senderror:
			  Printbin #1 , 254
			  Printbin #1 , 251
			  Printbin #1 , 255
			Return
				(Net use 730 - 182 = 548). Saving: (824 - 548) / 824 = 33.5%.

Step	Atmega 324A	ATXmega 128A1	Action	Comment

BA6b	724	1444	Bring Prbin back in:
			
			Sendpollport:
			  Printbin #1 , 254
			  Printbin #1 , 242
			  Uartsendbyte = 1
			  Gosub Prbin
			  Uartsendbyte = 2
			  Gosub Prbin
			  Uartsendbyte = 3
			  Gosub Prbin
			  Printbin #1 , 255
			Return
			
			Senderror:
			  Printbin #1 , 254
			  Printbin #1 , 251
			  Printbin #1 , 255
			Return
			
			Prbin:
			Printbin #1 , Uartsendbyte
			Return
				(Net use 724 - 182 = 542). Saving: (824 - 542) / 824 = 34.2%.

Figure 16: BASCOM-AVR iteration 6

Note that the ATxmega code remains 1444 while the ATmega code shrinks from 748 to 724. It seems that the implementations differ.

5.1.4 Custom USART inline assembly send functionality

Step	Atmega 324A	ATXmega 128A1	Action	Comment
BA7	688	1364	Send data one byte at a time, either from a global byte variable or from r24. In the odd event that array data should be sent, it should use additional inline assembly like so: LOADADR Serialdata0, X ' Load start address of Serialdata0 array into register pair X ld r24, X+ ' Load the value of this address into r24 and post-increment X rcall Senduart0b ' Send the byte in r24	

Figure 17: BASCOM-AVR iteration 7

For some reason, we save 36 bytes on ATmega324A but 80 bytes on ATmega128A1. Could this be because the use of hardcoded registers in the custom assembly code avoids using lots of address calculations necessary for the new ATxmega addressing scheme?

5.1.5 Custom USART inline assembly receive functionality

Serial communication is driven from the PC, in the form of request-response. For this reason, there should never be more than one message in the serial buffer at any one time. This means that the serial buffer doesn't have to be circular and that there is no need for copying out the message to a separate array.

BASCOM-AVR's circular buffer error handling in the interrupt routine only sets r6 bit 2 on error, after which it silently discards the overflowing byte and leaves the interrupt routine. This doesn't seem to be documented, so it's only after disassembly and additional r6.2 handling in the main loop that "buffer full" error could be handled.

Step	Atmega 324A	ATXmega 128A1	Action	Comment
BA8a	484	1080	Use status flag Serialbuffer0status to indicate "message being processed". In case a new message comes in while this is set, the interrupt routine calls Senderror and then resets.	
BA8b	464		No error handling. (Just to compare the sizes.)	100% stable, but error handling is nice. ;-)

Figure 18: BASCOM-AVR iteration 8

5.2 BASCOM-AVR summary #1

Step	ATmega 324A	ATxmega 128A1 v2.0.7.6	ATxmega 128A1 v2.0.7.7	Action	Comment
BA1a	1006	1720	1676	Worst VB-only implementation	
BA6b	724	1448	1410	Best VB-only implementation with the use of global variable aliasing to mimic local variable, goto-based loop, and undocumented Ischarwaiting syntax.	(Net use 724 - 182 = 542). Saving: (824 - 542) / 824 = 34.2%.
BA7	688	1364	1330	BA6b with custom inline assembly send routine.	(Net use 688 - 182 = 506). Saving: (824 - 506) / 824 = 38.6%
BA8a	484	1080	1030	BA7 with custom protocol-bound inline assembly receive routine, including error handling.	(Net use 484 - 182 = 302). Saving: (824 - 302) / 824 = 63.3% Or: (542 - 302) / 542 = 44,3% compared to best VB-only version BA6b.

Figure 19: BASCOM-AVR iteration summary #1

For ATmega324A, I was able to reduce the actual program code (excluding interrupt vector and default initialization) by 34% just by improving the VB code. As mentioned, I am using a few tricks that might be frowned upon, but even without these there's significant room for improvement without resorting to inline assembly.

On top of this, another 44% reduction in code size was possible by replacing BASCOM commands and "ordinary" VB serial handling routines by custom inline assembly send and procotol-bound receive routines. Let's look at the pros and cons:

Pros:

- A whopping reduction in size (a total of 63%). I didn't count the decrease in clock cycles, but since the size reduction doesn't come from removing loop unrolling or other techniques that favor speed over size, it is most likely that it also has a significant impact on processing time.
- Full error handling.

Cons:

- It (especially the receive handling) is now application-specific and protocol-bound. The tokens for message start (254) and end (255) are central to the receive interrupt routine.
- It took several hours (somewhere between 8 and 16) to implement.

Much of the assembly development time was general platform and architecture learning, that only has to be done once.

Please note that the serial handling is probably a special case. Judging by the disassembly of the BASCOM-AVR commands, they are well written with respect to the fact that they are general-purpose. This big reduction was only possible by making this functionality very strongly tied to this protocol. It is not very likely that I could repeat this in (many) other parts of the functionality.

It is also worth noticing that we can use almost the same code for ATmega324A and ATxmega128A1. Apart from the need to change register names, the only real difference is in clock setup and interrupt enabling. That said, I am surprised to see that the ATxmega uses so much more program code than the ATmega:

Architecture	**Interrupt vector**	**Total code excl IV**			
		BA1a	**BA6b**	**BA7**	**BA8a**
ATmega324A	124	882	600	564	360
ATxmega128A1 v2.0.7.6	~512	1208	936	852	568
Difference, ATx bigger by	~388 B	326 B	336 B	288 B	208 B
Difference, ATx bigger by		37.0%	56.0%	51.0%	57.8%
ATxmega128A1 v2.0.7.7	~512	1164	898	818	518
Difference, ATx bigger by	~388 B	282 B	298 B	254 B	158 B
Difference, ATx bigger by		32.0%	50.0%	45.0%	43.9%

Figure 20: BASCOM-AVR code size differences

(The v2.0.7.7 data will be explained later.)

I hadn't expected that the ATxmega would require so much more code to do exactly the same thing. It's not that I had reason to believe otherwise, I just think that not very many people have looked upon it like this. Atmel Sweden's tech support Marcus Woxulv said (very generally) that "*developers demand bigger program flash*"(44). This could be one reason. Even with all exotic code maneuvers, the smallest ATxmega is still bigger than the biggest ATmega implementation.

At this point, I had only tested and disassembled the ATmega324A code. Part of the ATxmega128A1 code consists of default initialization, which should rather be seen as part of the "offset" than the dynamic code. The question was if the actual ATxmega program code is also bigger than the ATmega counterpart and if so: Why?

I disassembled the ATxmega version BA6b and looked at the reasons for this difference. I knew beforehand that ATxmega's larger register space requires a greater fraction of LDS/STS operations. I also knew that ATxmega's new device addressing scheme with an individual offset into identical register structures makes it possible to use the same code block (together with the offset) to service more than one hardware device. I assumed that the address calculations and operations are code-intensive, but I had to disassemble and see.

5.3 Why is the ATxmega code so much bigger?

5.3.1 Initialization

The ATmega BA6b disassembly shows that the initialization takes 50 words.

The BASCOM ATxmega initialization takes 78 plus a call to shared code for USART register start calculation and register writing at 19 words = 97 words. Part of this is a few extra words for the more complex ATxmega system clock, but much of it refers to dynamic addressing:

Please see appendix A.6.1 for the ATxmega BASCOM-AVR v2.0.7.6 compiled code for USART setup. Total code size for one port: 32 words.

- Initialization 13 words (one set per serial port)
- USART setup 11 words (one set if serial ports are used)
- USART address calculation 8 words (one set if serial ports are used, shared with USART writing routine as shown in the next section)

This is surprising. It's a lot of operations just to write to five registers. In appendix A.6.2 you can see how it scales. That code must be added if you want to set up a second serial port identical to the first one. It adds 8 words, so the total cost is 13 + 8 + 11 + 8 = 40 words. If you want to set the second port differently, the additional cost is up to 13 (instead of 8) words.

Let's write our own USART setup with exactly the same functionality (appendix A.6.3). The total code size is 13 words, one set per serial port.

How does it scale (appendix A.6.4)? The total cost for setting up two serial ports: 7 * 2 + 9 = 25 words.

We end up with the following list:

- Dynamic addressing one port: 32 words (of which 24 are setup-specific)
- Dynamic addressing two identical ports: 32 + 8 = 40 words
- Dynamic addressing two different ports: 32 + 13 = 45 words
- Static addressing one port: 13 words
- Static addressing two identical ports: 7 * 2 + 9 = 25 words
- Static addressing two different ports: 13 * 2 = 26 words

This is even more surprising. Dynamic addressing starts off worse and scales the same or slightly worse than static addressing. Remember that this is just the code size. The performance loss is significant, as will be seen in a while.

Let's continue with the USART sending on one port. Appendix A.6.5 contains the ATmega original disassembly; Total words: 26.

Appendix A.6.6 holds the ATxmega original disassembly; Total words: 46.

Both the ATmega and the ATxmega codes only operate on two physical registers (UCSR0A & UDR0 / USARTE1_STATUS & USARTE1_DATA). The main reason why the ATxmega code is bigger is because it uses dynamic addressing. When using only one serial port, this causes roughly a doubling of the code size.

Here I must mention that I chose to analyze USART sending for its simplicity. At this point, I didn't consider the fact that when sending (multiple bytes of) serial port data, most of the processing time will be spent busy-waiting for the previous byte to leave the output buffer, which means that the clock cycle count reduction will result in a very tiny runtime improvement. However, reading data from the instance's circular buffer (using commands "Ischarwaiting(#1)" and "Inputbin #1 , Serialbyte") will not involve busy-waiting, so it is most likely that the (assumed) decrease in that code size would show a similar significant runtime decrease, although I didn't have time to analyze that. Generally speaking, so long as the application doesn't have to busy-wait for a HW peripheral, the code size reduction should be accompanied by a runtime improvement by migrating from dynamic to static addressing. In that sense, the figures should still be a good illustration of the behavior in situations where the clock cycle count reduction actually leads to a performance increase.

Clarification 1: Clock cycle count and busy-waiting (above)

Let's scale to two serial ports sending one constant and one variable through two different ports. The one-port code is the actual BASCOM-compiled one, but in order to make it more fair, I have made small adjustments. Please see appendix A.6.7 for the modified ATmega code for two serial ports. Its total size is 47 words.

5.3.2 Modified ATxmega for two serial ports

BASCOM-AVR is slightly less efficient than the code below, as can be seen in the BA6b disassembly (appendix B). I moved the ST -Y,R23 operation that puts R23 on the stack from address 1B3 to label __USART_b1, so that it will only be included once. This was done in order to make it fairer for the ATxmega, i.e. so that no part of the difference between static and dynamic addressing could be explained by inefficient implementation. (This is only relevant for static addressing and hence only occurs for ATxmega in the version of BASCOM-AVR I was analyzing.)

Please see appendix A.6.8 for the code that requires 55 words. Each new serial port adds 10 words. All of these 10 are "variable", in the sense that they "cost" this much for each "use".

5.3.3 Modified ATmega for three serial ports

This code can be found in appendix A.6.9, amounting to 27 + 9 + 3 * 10 = 66 words. Each new serial port adds 9 + 10 = 19 words. Of these, 10 are "variable" with actual "use" in the VB code.

So, with my sample code, at three serial ports the two addressing modes are just about the same code size. (ATmega at 66 and ATxmega at 65.) Based on the above code modifications, the following table and graph emerge:

Code size, words (1 word = 2 bytes):

# serial ports	0	1	2	3	4	5	6	7	8
Dynamic addressing	28	46	55	65	75	85	95	105	115
Static addressing	12	26	47	66	85	104	123	142	161
Actual/Modified	M	A	M	M	M	M	M	M	M

Table 4: Code size extrapolation (Actual = data from real disassembly, Modified = data from original disassembly re-written for more ports and (where specified) minimal code size. See 5.3.2.)

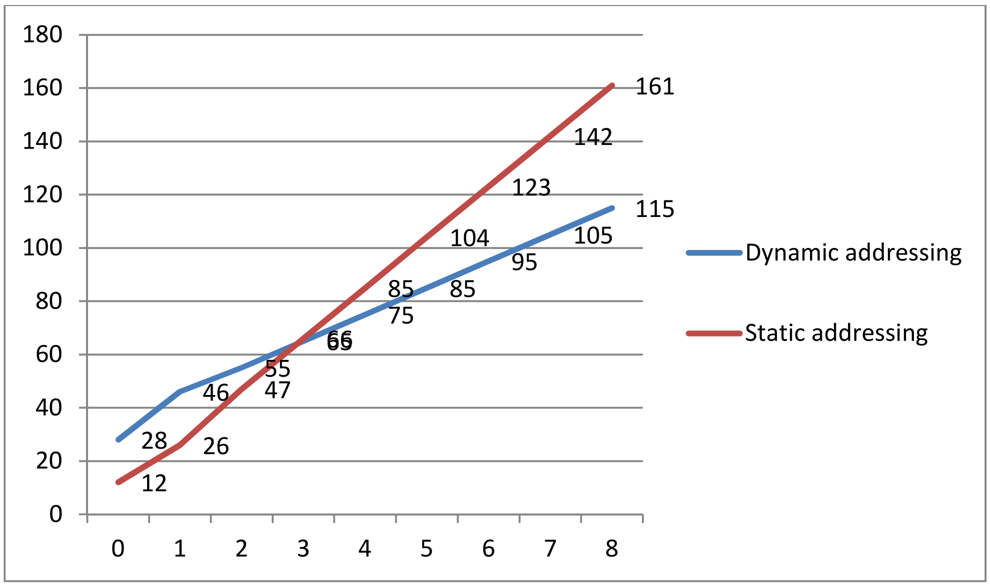


Figure 21: Serial port scaling

Just looking at these, it seems like the dynamic addressing “wins” after 3 serial ports. By using dynamic addressing, we reduce the code size at eight serial ports by (161 - 115) / 161 = 28.6%.

However, it’s not just the program code size that we’re interested in. We also want to see the performance consequences. Let’s do a small cycle count on a part of both of the modified two-port samples (but please note Clarification 1):

Printbin #1, 254 - dynamic addressing, best case:
1+1+3+2+1+1+2+1+1+1+2+1+1+4+2+1+2+2+1+2+2+4=38 cycles.

Printbin #1, 254 - static addressing, 1st port used, best case:
1+1+3+1+1+2+2+2+4=17 cycles. (Dynamic addressing requires 38 / 17 = 124% more clock cycles.)

Printbin #1, 254 - static addressing, 8th port used:
1+1+3+1+2+1+2+1+2+1+2+1+2+1+2+1+2+2+2+2+4=36 cycles.

With static addressing and 8 ports, the average clock cycle count would be (17 + 36) / 2 = 26.5. (Dynamic addressing requires 38 / 26.5 = 36% more clock cycles.)

With static addressing and 3 ports, the average clock cycle count would be (17 + 23) / 2 = 20. (Dynamic addressing requires 38 / 20 = 90% more clock cycles.)

In this example, dynamic addressing leads to a remarkable increase in the number of clock cycles for exactly the same functionality. This is largely “variable” with actual “use” (meaning the number of times the VB commands are used in the program). It turns out that some of this is due to the BASCOM-AVR implementation. Please see 5.3.4 for an improved version.

The above is for sending either a single byte or a byte-sized constant. To enable byte array sending, an additional gosub with a few statements would be necessary, equal for ATmega and ATxmega, so it doesn’t change the comparison in absolute numbers.

I should point out that several of the instructions used for the sample code have a better implementation in the ATxmega than in the ATmega. (Appendix A.4.3.)

5.3.4 Improved ATxmega dynamic addressing for two serial ports

The BASCOM-AVR implementation in 5.3.2 is not optimal: The Printbin command (at label --USART_a and onwards) first loads (and autoincrements) the X pointer and then calls the send routine at label --USART_b, that calls label --USART_c for module instance address calculation before sending. This is repeated for each array byte, which seems quite unnecessary as the entire array is sent through the same instance.

So, I decided to rewrite the 5.3.5 disassembly (using “MS Word assembly”) in order to see if I could reduce its size and / or clock cycle count (appendix A.6.10): Total words: 46. This is 9 words less than the BASCOM-AVR implementation. Each new port adds 10 words. Please note the extra requirement placed on the compiler by the __Prbin_gosub.

We have achieved 2-port dynamic addressing at the same program size as static addressing (or slightly worse if you play safe with the compiler requirement mentioned above). Furthermore, the serial port base address is only calculated once per transfer.

Printbin #1, 254 improved addressing:
1+1+3 +1+1 +1+1 +1+1+2+1+1+ 2+ 3+2+1+1+1 +2+2+4 = 33.

This is 5 clock cycles less than the BASCOM-AVR implementation but 33 - 26.5 = 6.5 more than the 8-port average for static addressing. (Again, please note Clarification 1.)

5.3.5 Investigating hardware-based port address lookup

I thought about adding hardware support for address lookup. In the above solution, the last six of the appendix A.6.10 instructions in *italic* (or perhaps optionally even the two first) could be replaced by one new instruction as described in the summary chapter.

Assuming that it could (like LDD) be constructed so that it requires only 1 instruction word, the 2-port USART sending routine above could be reduced to 41 words. Compared to the static addressing version that takes 47 bytes, it’s quite ok. Given that each peripheral module has its “*italic*” address calculation code that could hereby be removed, we would be looking at a total code size reduction of (5 or 7) * (the number of different peripherals we’re using) words.

However, assuming that the new instruction “block” should use a maximum of 2 clock cycles (LD to r24 and the new read instruction), the above example would at most amount to 1+1+3 +1+1 +1+1 +2+ 2+ 3+2+1+1+1 +2+2+4 = 29 clock cycles. This is only a (33-29)/33 = 12% reduction compared to code-based calculation, but at least we’re getting close to the 8-port static addressing average 26.5.

Perhaps other peripherals require more than two register operations per “run”. In that case, the fraction of “overhead” generated by address calculation would be slightly less. Nevertheless, it’s clear that dynamic address calculation significantly increases the number of clock cycles compared to static addressing, even with hypothetic tailor-made hardware support. When sending one byte constant using only one serial port, dynamic hardware-based address lookup requires 29/17 = 70% more clock cycles than static addressing. The 5.3.7-improved dynamic code-based address calculation requires 33/17 = 94% more.

While the improvement might be too small to justify a hardware address lookup table, it has one other advantage: It would enable a uniform programming style for both ATmega and ATxmega (and AVR32 and ARM?).

5.3.6 Improvements from 2.0.7.6 to 2.0.7.7

I posted a question in the BASCOM-AVR forum about the possibility to use the interrupt vector program code area for regular code. (45) [55] As a result of this, version 2.0.7.7 includes an unsupported setting ($reduceivr) that places the regular program code just after the last used interrupt's address rather than after the end of the entire interrupt vector. This saves 724 - 684 = 40 bytes on the current ATmega324A code for BA6b and 1410 - 1278 = 132 bytes on ATxmega128A1. While this is a nice feature, I haven't included this saving in my program size measures simply because that would hide part of the ATxmega inefficiency in the actual program code.

When I was finalizing the work with BASCOM-AVR, I e-mailed Mark Alberts, the owner of MCS Electronics that produces BASCOM-AVR, pointing out that the ATxmega compiled code was much larger than the ATmega compiled code of exactly the same BASCOM-AVR program. The ATxmega USART initialization in version 2.0.7.7 is now using exactly the same code as I sent him (appendix A.6.11).

In addition to this code size reduction, he discovered some other possible improvements in other parts of the code. This is the reason why version 2.0.7.7 compiles to smaller size than 2.0.7.6.

5.3.7 How big is the initialization code?

Please see appendix A.6.12 for the ATmega324A initialization (50 words) and A.6.13 for ATxmega128A1 (75 words).

In other words, for the BA6b code, I had expected the ATxmega size to be close to 512 - 124 + 724 + (75 – 50) * 2 = 1162 bytes. The 2.0.7.6 version compiles to 1448 and 2.0.7.7 compiles to 1410 bytes. As mentioned before, 2.0.7.6 is using address calculation for USART initialization and sending, while 2.0.7.7 is using static addressing for initialization and address calculation for sending.

5.3.8 Other compiled code that's unused by the test application

Apart from the device-specific initialization and the actually utilized program code, BASCOM-AVR also adds pieces of code that I suppose is generic, "frequently used" code and therefore always includes it. This section was compiled with v2.0.7.7.

Common to both architectures (and identically implemented):

- Delay
- Set error bit in R6
- Clear error bit in R6

The size of this is 9 words (18 bytes).

After looking at the ATxmega disassembly, I realized that I had missed the fact that devices with more than 64 kB program memory require additional MSB bit(s) for addressing. This doesn't affect

[55] http://www.mcselec.com/index2.php?option=com_forum&Itemid=59&page=viewtopic&t=11718

the ATmega 324, but it does affect the ATxmega128A1. For this reason, I also compiled the BA6a version for ATmega1284.

For these two, we have a common piece of code (identically implemented):

- RAMPZ register addressing

Its size is 15 words (30 bytes).

Then there's a chunk of 45 words (90 bytes) of code that operates on a BASCOM-AVR internal _XMEGAREG 32 bytes RAM area that I don't know what it's for:

- Clear the entire area
- Double a 5-byte number at the start of this area

I have googled and searched the BASCOM forum but found no clue.

- Both ATmega and ATxmega: 9 words (appendix A.6.14.1).
- Only ATmega1284 and ATxmega128A1: 15 words (appendix A.6.14.2).
- Only ATxmega128A1: 45 words (appendix A.6.14.3).

5.3.8.1 Summing up the unused code section

In order to make the comparison fair, I must compensate for the RAMPZ difference in the summary table. After some consideration, I decided that the ATxmega-specific 32-byte data area handling should be included as it's an actual difference compared to the BASCOM-AVR implementation of the ATmega architecture, but it's not established that it's really required by the ATxmega architecture itself. This is getting messy…

For some reason, the ATmega1284 compilation requires 770 = 132 + 638 bytes in total (compared to 724 = 124 + 630) for ATmega324. Part of the difference is a slightly bigger interrupt vector table and 8 more bytes of compiled program code inside used routines that I haven't analyzed further. (After the Atmel Studio 6 analysis, I think that these 8 bytes are ISR RAMPZ stack operations.) The comparisons between ATmega and ATxmega are still done on ATmega324 (to avoid having to recalculate all the data). As you can see in the table below, it doesn't make much of a difference:

Architecture	**IV table**	**Total code excl IV, bytes**					
		BA1a	**BA6b**	**BA6b+RAMPZ**	**BA6b+RAMPZ-_XMEGAREG**	**BA7**	**BA8a**
ATmega324A	124	882	600	630	630	564	360
ATmega1284	132			638	638		
ATxmega128A1 v2.0.7.6	~512	1208	936			852	568
Difference, ATx bigger by	~388 B	326 B	336 B			288 B	208 B
Difference, ATx bigger by		37.0%	56.0%			51.0%	57.8%
ATxmega128A1 v2.0.7.7	~512	1164	898	898	808	818	518
Difference, ATx bigger by	~388 B	282 B	298 B	238 B	178 B	254 B	158 B
Difference, ATx bigger by		32.0%	50.0%	42.5%	28.3%	45.0%	43.9%

Figure 22: BASCOM-AVR total code size comparison

Even when compensating for RAMPZ and excluding the 32-byte BASCOM-AVR ATxmega data area, the ATxmega still compiles to about 28% bigger for exactly the same functionality. I suspected that the difference is caused mostly by the following:

- Bigger initialization code due to ATxmega having a more complex architecture
- Dynamic address calculation being more “expensive” than static addressing
- The ATxmega’s bigger register address space leads to a higher fraction of LDS/STS rather than IN/OUT operations.

The question is: how much of the difference does each of the above cause? As we saw in the previous subsection, the ATmega initialization is 50 words and the ATxmega is 75 words. A quick look at the datasheets tells us that the lowest ATmega324 USART register is placed at 0xCE (206) and the ATxmegaA1 at 0x8A0 (2208). So, no IN/OUT instructions are used, which means that the entire remainder of 178 - (75-50)*2 = 128 bytes are due to the difference between static and dynamic addressing.

In other words, after compensating for initialization, RAMPZ, the zero effect of IN/OUT vs LDS/STS, and unused generic code, an additional 128/630 = 20% of code is the consequence of the different programming styles.

(Adjustment: After the Atmel Studio ISR analyses, we know that the static ATxmega128A1 ISR is 10 words bigger than the ATmega3241’s, due to the use of RAMPD and RAMPZ. The S&P ISR is 14 words bigger as it also requires RAMPX for ATxmega128A1. This doesn’t have a significant effect - the 20% above becomes 17%.)

5.4 Generalization

To what extent is this result generally valid? Or, is this just an unfortunate coincident in the otherwise successful use of dynamic addressing?

I can hardly see how you can get around the performance issue but dynamic addressing should scale size-wise comparably better when you are working with a (larger) series of sub-register operations than our two (status and data) registers.

I think that it is safe to say that whole-hearted conformance to a specific ideal or concept runs the risk of losing focus of what is really important, in this case both code size and performance. Here I would use a mix of static and dynamic addressing, probably leaving it up to the programmer to decide which scheme the compiler should use. However, it is expensive to develop and support multiple ways to do the same thing. Unless the users demand efficient code, it is just an additional cost for the IDE developer.

5.4.1 What's the problem with bigger code and lower performance?

Why do I spend so much time and effort on the addressing topic? Firstly it's because I set out to evaluate the two architectures, so a 2:1 performance difference is too big to overlook. On the other hand, it's probably still not noticeable in any of my designs. I guess this fact is true for many (most?) other designs as well.

However, I strongly believe in making informed decisions. This is the difference between dabbling and being a professional. By properly understanding the basics, you become a better programmer.

Disregarding work ethics, If for example you are using the current version (2.0.7.7) of BASCOM-AVR on ATmega644 running at 20MHz with 64kB program code and you want to change it to an ATxmega, you will need to buy the next bigger program flash size and clock it significantly faster if you want it to act the same.

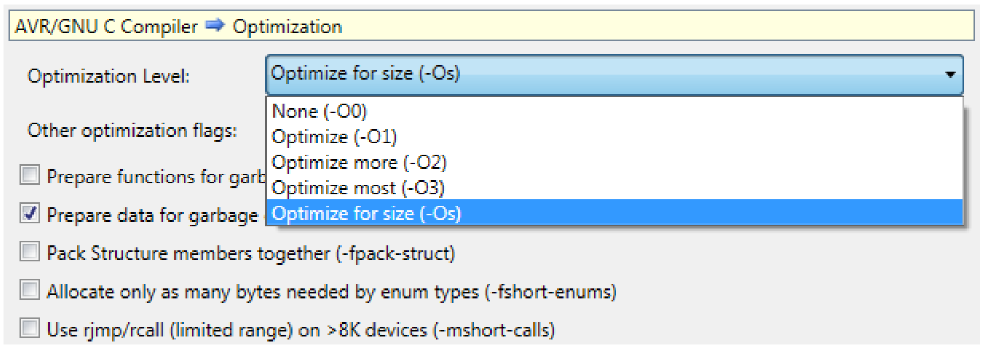
The ATxmega is a more powerful and complex design. This means that initialization needs more code (with this sample code roughly 50% more) and that a greater fraction of the instructions must use the "bigger" LDS and STS instructions instead of IN and OUT. On the other hand, some instructions require fewer clock cycles in ATxmega than in ATmega, so the ATxmega could actually run general application code roughly as fast at the same clock frequency. The additional RAMPD/X/Y/Z and EIND registers on ATxmega128A1 incur an additional cost, but probably (almost) only for ISRs.

The big BASCOM-AVR difference comes from the choice between static addressing and dynamic address calculation.

6 Atmel Studio 6.1 2562 using ASF3.13.1 code analysis

6.1 Optimization primer

There are five pre-defined optimization levels:



AVR/GNU C Compiler ➡ Optimization

Optimization Level: Optimize for size (-Os)

- None (-O0)
- Optimize (-O1)
- Optimize more (-O2)
- Optimize most (-O3)
- Optimize for size (-Os)

Other optimization flags:

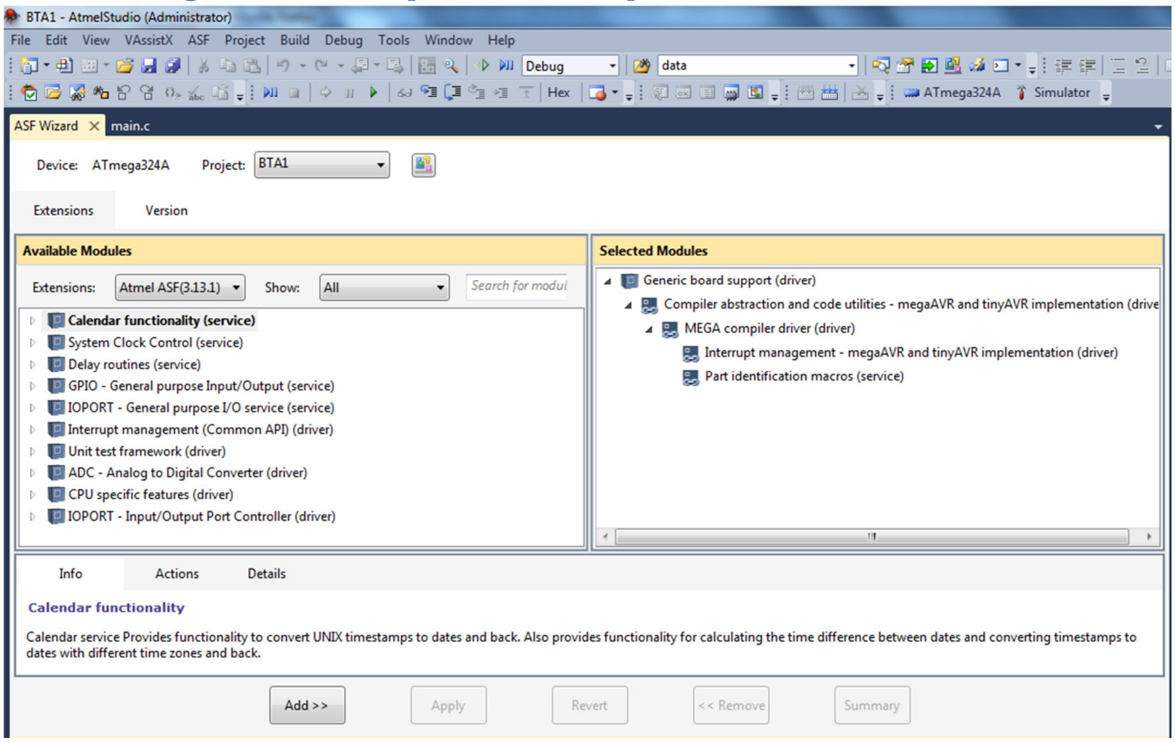
- [ ] Prepare functions for garb
- [x] Prepare data for garbage
- [ ] Pack Structure members together (-fpack-struct)
- [ ] Allocate only as many bytes needed by enum types (-fshort-enums)
- [ ] Use rjmp/rcall (limited range) on >8K devices (-mshort-calls)

Figure 23: AVR-GCC optimization levels

The GCC online docs has a manual page on optimization. (35) [56] There are many options (flags), but basically -O1 does optimizations that don't "take a great deal of compilation time", -O2 does "nearly all supported optimizations that do not involve a space-speed tradeoff". -O3 is roughly -O2 with speed optimization (that might increase code size) and -Os is the parts of -O2 that "do not typically increase code size". It "also performs further optimizations designed to reduce code size".

6.2 ATmega324A analysis

6.2.1 Getting the base serial port routines in place



Device: ATmega324A Project: BTA1

Available Modules

Extensions: Atmel ASF(3.13.1) Show: All Search for modul

- Calendar functionality (service)
- System Clock Control (service)
- Delay routines (service)
- GPIO - General purpose Input/Output (service)
- IOPORT - General purpose I/O service (service)
- Interrupt management (Common API) (driver)
- Unit test framework (driver)
- ADC - Analog to Digital Converter (driver)
- CPU specific features (driver)
- IOPORT - Input/Output Port Controller (driver)

Selected Modules

- Generic board support (driver)
  - Compiler abstraction and code utilities - megaAVR and tinyAVR implementation (drive
    - MEGA compiler driver (driver)
      - Interrupt management - megaAVR and tinyAVR implementation (driver)
      - Part identification macros (service)

Calendar functionality

Calendar service Provides functionality to convert UNIX timestamps to dates and back. Also provides functionality for calculating the time difference between dates and converting timestamps to dates with different time zones and back.

Figure 24: ASF wizard for ATmega324A (showing the available modules)

[56] http://gcc.gnu.org/onlinedocs/gcc-4.8.2/gcc/Optimize-Options.html#Optimize-Options

I begun by creating a new project, C/C++, User-Boards, User Board template - megaAVR ATmega324A. The User Board template includes the Interrupt management driver and CPU specific features. AS1 started as an empty project that compiled to 156 bytes program code (text) (-O1). It contains a 124 byte interrupt vector, proceeds with clearing R1 and SREG, sets SP via Y, calls board_init(), clears R24 and R25 (W), disables global int, and does eternal loop.

Then I added the "System Clock Control (service)" using ASF Wizard and made the following additions:

In board_init(): call sysclk_init();

In conf_clock.h: change to
```
#define SYSCLK_SOURCE       SYSCLK_SRC_XOC16MHZ
#define CONFIG_SYSCLK_PSDIV       SYSCLK_PSDIV_1
```

AS1 now compiled to (bytes text):
362 text, ? bss (-O0)
192 text, ? bss (-O1)
190 text, ? bss (-O2)
190 text, ? bss (-O3)
190 text, ? bss (-Os)

It also shuts down the peripherals (PRR0 and PRR1, Power Reduction Registers to 0xFF): "Since all non-essential peripheral clocks are initially disabled, it is the responsibility of the peripheral driver to re-enable any clocks that are needed for normal operation." (46) [57]

Then it reads SREG, writes 0x80 and then 0x00 to CLKPR (CLocK PRescaler register) to set prescaler to 1, and restores SREG.

This is about as far as I can go when developing my ATmega test program using Atmel Studio 6.1 and ASF 3.13.1. It has interrupt handling code but no USART driver. (It does have IOPORT, ADC, delay, and calendar functionality.) I therefore looked at application note AVR306 (47) [58] [59] that has USART code samples for AT90S8515 and ATmega128, both polled and interrupt-based with circular buffer. ATmega128's USART is almost identical to ATmega324. For ATmega324, register UCSRnC.7 must also be set (or cleared).

The application note is fairly valid, even though it was released in 2002 for what I suspect is the IAR Systems compiler. I had to update the SEI instruction call and the interrupt handler syntax, which was quite simple.

The basic serial port routines without base-2 requirement and without overflow protection: 315 text, 18 bss (-O1). Please note that this isn’t functionally equivalent to the BASCOM-AVR built-in interrupt handler that discards overflowing USART characters with an R6.2 error flag.

The basic serial port routines with base-2 requirement and without overflow protection: 298 text, 18 bss (-O1). This is the version closest to the application note code.

[57] http://asf.atmel.com/docs/3.13.1/mega/html/group__sysclk__group.html
[58] http://www.atmel.se/Images/doc1451.pdf
[59] http://www.atmel.com/images/avr306.zip

Before proceeding, I did some rearranging and cleaning up of the application note code. The appendix AS1a code and disassembly don't re-initialize the global variables to zero:
290 text, 17 bss (-O1)

6.2.2 Early impressions of Atmel Studio 6.1 and ASF 3.13.1

At this point, I would like to summarize my early impressions of Atmel Studio 6.1 and ASF3.13.1: For ATmega it seems that the core drivers (and ADC) are available in ASF, while the majority of the peripheral drivers aren't included. This is in line with my expectations. I guess Atmel had to prioritize and decided to make a minimal ATmega ASF implementation (although this assumption is placed in a different light in the ATxmega analysis later on). The available ATmega ASF parts are a good start and there's a very big user base that publishes code samples and asks or replies to user forum questions. During my research for and writing of the chapter on AVR HW, I found a very satisfying number of Atmel datasheets, application notes, and marketing material that helped me write the serial communication routines in a few hours (and later the ASF documentation).

However, I also came across a bit too many dead ASF documentation links to not mention it here. Please see appendix A.7.1 for a number of screen dumps documenting various ASF issues. Atmel Studio 6.1 was stable when developing and using ASF but on rare occasions it hang when debugging. To be fair, much documentation exists and ranges from ok to very good (appendix A.7.2).

There's also another thing that (at least as a beginner) annoys me about the Atmel driver library: it's very difficult to get an overview of which header and code files your project really consists of and is using, as they form a very big tree of includes within includes within includes... I used this IDE for about a week in a robot project course half a year ago and then ended up utilizing the ASF library as a copy&paste sample code repository, into very few custom header and code files. I realize that I have started doing this with the ATmega test program as well. I fear that this might be a common way of doing things in the GCC (Gnu Compiler Collection) "world", but I will keep this in mind when writing the ATxmega test program.

Except for these issues, my overall impression of developing and debugging in Atmel Studio 6.1 is good. It takes a long time to install and start up the program but it's intuitive and very agreeable. I'm not partial to ASF though, as we will soon discover.

6.2.3 The first version of the test program

With the USART routines in place, I proceeded by translating the rest of the BASCOM-AVR BA6b test program. (This was the best VB-only version.) Please note that there are two functional differences: The BASCOM-AVR circular buffer used in the serial port receiver interrupt handler doesn't have to be dividable by 2 and it does overflow signaling by setting the R6.2 error bit.

When building the test program I noticed that -O2 and -O3 do performance optimization by inlining the transmit function:

```c
void USART0_Transmit( unsigned char data )
{
  while ( !(UCSR0A & (1<<UDRE0)) );          /* Wait for empty transmit buffer */
  UDR0 = data;                               /* Start transmission */
}
```

This compiles to a great many almost identical copies of this:

```
             while ( !(UCSR0A & (1<<UDRE0)) );          /* Wait for empty transmit buffer */
000000BB 80.91.c0.00          LDS R24,0x00C0           Load direct from data space
000000BD 85.ff                SBRS R24,5               Skip if bit in register set
000000BE fc.cf                RJMP PC-0x0003           Relative jump
             UDR0 = data;                               /* Start transmission */
000000BF 82.e0                LDI R24,0x02             Load immediate
000000C0 80.93.c6.00          STS 0x00C6,R24           Store direct to data space
```

The thing is that there is little point in speed optimizing the wait for sequential asynchronous serial port transmission. (Please see Clarification 1.) My application uses a 6 MHz system clock and a 15 625 baud rate in asynchronous mode. This equates to 6000 000 / 15625 = 384 system clock cycles per serial port bit or a minimum of 384 * 9 = 3456 system ticks per received byte with my settings. An RJUMP (with RET) to a dedicated assembly routine only costs an additional 2 + 5 = 7 clock cycles. There will still be plenty of turns in the while loop waiting for the previous byte transmission to be completed, so the performance gain when sending multiple bytes is actually never bigger than these seven clock cycles. (Two cycles to RJUMP to the dedicated routine the first time, then it's waiting until it can send the next byte so the RET and next RJMP don't matter, and finally it takes five cycles to do the last RET.)

However, the program code has grown substantially:

-O0 890, 37
-O1 514, 37
-O2 556, 37
-O3 544, 37
-Os 490, 37

Of course the compiler doesn't know this and it's a clear sign that you can't rely on the compiler to produce optimal code for you.

After changing the declaration of the transmit function to "__attribute__ ((noinline)) void USART0_Transmit(unsigned char data)" I get the following compilation results:

-O0 890, 37
-O1 514, 37
-O2 476, 37
-O3 470, 37
-Os 490, 37

-O2 has shrunk with 556 – 476 = 80 bytes (14%). I include the new -O3 compilation in the appendix.

Turning my attention to the two smallest compiler results -O1 and -Os (and the previous -O3), I notice some differences: -O1 contains one more instance of "SerialData[abc] = def;" and "while (DataInReceiveBuffer())" than the C code has. -Os has the same number of these two statements as the C code. -O3 has the same number of the first but one more of the second. I suppose that there are good reasons for this, but the question is how to properly predict the outcome.

It seems to me that professional use of IDEs based on the AVR GCC toolchain requires quite a bit of knowledge about GCC optimization. Perhaps even then you need to look at disassemblies (or compiler-generated assembly files) of your compiled code and give the compiler explicit instructions as you move along. How big is the step between this and simply converting (select parts of) your disassembly to hand-optimized inline assembly (which is what you (in some situations) must do in BASCOM-AVR if you want efficient code)?

(In certain cases you must give the compiler explicit instructions, as for example writing to a register and then reading it back will result in the compiler optimizing the read away. It doesn't "know" that the register value could have changed.)

6.2.4 Making the USART receiver interrupt handler protocol-bound

As there's no counterpart to the BASCOM-AVR USART send commands, I'm already using a custom one in my C program. For this reason, I go straight to the program version with protocol-bound USART receiver interrupt handler corresponding to BASCOM-AVR BA8a at the total size of 484 bytes. It can be found in the appendix, called AS1c. I also include the disassembly of -Os.

-O0: 742 text, 18 bss
-O1: 418 text, 18 bss
-O2: 414 text, 18 bss
-O3: 412 text, 18 bss
-Os: 412 text, 18 bss

This version isn't restricted to a 2^n buffer size, so they are (almost) functionally equivalent. (A 16-byte buffer is used for comparison to the restricted versions, while the BASCOM-AVR has a 20-byte buffer. This only affects the SRAM use.)

Let's do a rough backward comparison:

6.2.4.1 BASCOM-AVR

50 bytes of initialization:

- Set SP to RAMEND
- Set Y to SW stack start
- Set Z to frame start
- Store frame start in R4+R5
- Watchdog reset
- Clear any reset flag except watchdog
- Watchdog disable
- Clear entire SRAM
- USART settings
- Clear R6 (error flags)
- Enable global interrupts

18 bytes of unused code (delay and R6.2 error bit handling).

A total of 68 bytes.

6.2.4.2 Atmel Studio AVR GCC

50 bytes of initialization:

- Clear R1
- Clear SREG
- Set Y to RAMEND
- Clear SRAM global area
- Disable peripheral clocks
- Temp save SREG
- Disable global interrupts
- Set system clock prescaler = 1
- Restore SREG
- USART settings
- Enable global interrupts

Apart from the fact that the initializations are slightly different, we see that Atmel Studio produces smaller code:

- BASCOM-AVR: 484 - 124 - 50 - 18 = 292 bytes.
- Atmel Studio AVR-GCC -Os: 412 - 124 - 50 = 238 bytes, about 18.5% smaller.

Please bear in mind that this comparison is a bit rough as they don't do exactly the same things. Both contestants could be further improved, e.g. by reducing the interrupt jump table.

6.2.5 Trying I/O registers for the two global variables

How much can be gained by using I/O registers accessible by IN/OUT?

-O0 790 text, 16 bss
-O1 404 text, 16 bss
-O2 400 text, 16 bss
-O3 398 text, 16 bss
-Os 398 text, 16 bss

Additionally changing ReceiveCounter from GPIOR2 to R3 results in a 2-byte -Os reduction.

On the ATmega324A the following IN/OUT-accessible I/O registers could possibly be used for global variables:

0x28 (0x48) OCR0B Timer/Counter0 Output Compare Register B
0x27 (0x47) OCR0A Timer/Counter0 Output Compare Register A
0x26 (0x46) TCNT0 Timer/Counter0 (8 Bit)
0x2B (0x4B) GPIOR2 General Purpose I/O Register 2
0x2A (0x4A) GPIOR1 General Purpose I/O Register 1
0x21 (0x41) EEARL EEPROM Address Register Low Byte
0x20 (0x40) EEDR EEPROM Data Register
0x1E (0x3E) GPIOR0 General Purpose I/O Register 0 (bit-operable)

As we can see, the possibility to reduce program code size and execution time by using I/O registers for global variables is quite limited on this microcontroller. For this reason, I continue the analysis by using SRAM for global variables. AS1d is not included in appendix B, but its version-specific code can be seen in AS1e.

6.2.6 Custom initialization

I noticed two unwanted compilation results caused by using the library function “sysclk_init()”:

- It did a few RCALLs as the underlying code comes from several places (unnecessary extra size).
- It wrote to the non-existent PRR1 register when unnecessarily shutting down all peripherals so I had to re-enable USART0 with an additional statement.

Also, I like to have a clear picture of exactly what is done, which is very difficult when library functions make nested calls. For these reasons, I simply copied and altered the library code and declared the “Initialization” function “inline”. Now the custom initialization sequence disables interrupts, enables power only to USART0, sets prescaler to 1, does the USART initialization, clears SREG, and enables interrupts. (This is in addition to the default initialization code inserted by the compiler.)

Oddly enough, -O0 wouldn’t compile and gave me the error message shown in appendix A.7.3. I had to remove the “inline” directive for the “Initialization” function for it to work. The other compilation levels have the “inline” directive.

When testing this program version (on -Os), I noticed that my SW reset (by `goto *0x0000;`) wasn’t working properly. Sometimes the PC received a truncated error message. It turned out that removing the unnecessary library initialization code and avoiding the RCALLs sped up the entire initialization process so much that often it didn’t have time to finish sending all of the error message. When bugsearching I also noticed, on very few occasions, that this did in fact also happen when using the library sysclk_init() function. I could use the watchdog timer to wait for a certain period, but I would still have to enter a(n eternal) loop while waiting for it to trigger, so I chose to just copy the generic delay code “`do { barrier(); } while (--counter);`”. 20 passes seem to be enough, while 17 is too few. This makes the code sensitive to errors in case of future changes, but as an actual production program based on this code would have more global variables that need clearing by the default initialization (which takes longer), it should be less of a problem.

At this point -Os without the time-out loop makes 386 and with it 394 bytes. This is in comparison to version C at 412 bytes (without time-out loop).

Compiled size:
-O0 608 text, 18 bss (but only after I removed “inline”)
-O1 400 text, 18 bss
-O2 396 text, 18 bss
-O3 394 text, 18 bss
-Os 394 text, 18 bss

Looking at the initialization code, I notice that the USART register writes are done via STS instructions.

```
              UBRR0H = 0x00;
00000075 10.92.c5.00          STS 0x00C5,R1               Store direct to data space

              UBRR0L = 0x17;
00000077 87.e1                LDI R24,0x17                Load immediate
00000078 80.93.c4.00          STS 0x00C4,R24              Store direct to data space

              UCSR0B = (1<<RXCIE0)|(1<<RXEN0)|(1<<TXEN0);
0000007A 88.e9                LDI R24,0x98                Load immediate
0000007B 80.93.c1.00          STS 0x00C1,R24              Store direct to data space

              UCSR0C = (1<<UCSZ01)|(1<<UCSZ00);
0000007D 86.e0                LDI R24,0x06                Load immediate
0000007E 80.93.c2.00          STS 0x00C2,R24              Store direct to data space
```

Using ST with displacement with the Z pointer could be more efficient. Should this be done via inline assembly or using a struct? In the ATmega324 both USARTs have the same relative register placement, so first I try the struct, partly based on "AVR1000: Getting Started Writing C-code for XMEGA" (29) [60]:

```
/* Type definition for the USART struct */
typedef struct USART_struct {
   uint8_t UCSRA;
   uint8_t UCSRB;
   uint8_t UCSRC;
   uint8_t Reserved;
   uint8_t UBRRL;
   uint8_t UBRRH;
   uint8_t UDR;
}USART_t;

USART_t *USART_inst = (USART_t *)&(UCSR0A);
(USART_inst)->UCSRB = (1<<RXCIE0)|(1<<RXEN0)|(1<<TXEN0);
(USART_inst)->UCSRC = (1<<UCSZ01)|(1<<UCSZ00);
(USART_inst)->UBRRL = 0x17;
(USART_inst)->UBRRH = 0x00;
```

No difference using -Os. It produces exactly the same machine code. I then try to be even more specific:

```
uint8_t *reg = (uint8_t *)&(UCSR0A);
*(reg + 1) = (1<<RXCIE0)|(1<<RXEN0)|(1<<TXEN0);  // UCSR0B
*(reg + 2) = (1<<UCSZ01)|(1<<UCSZ00);            // UCSR0C
*(reg + 4) = 0x17;                               // UBRR0L
*(reg + 5) = 0x00;                               // UBRR0H
```

No difference using -Os. It produces exactly the same machine code. Am I doing something wrong or is the potential improvement too small for the compiler to use the Z pointer? I try inline assembly as a last resort:

[60] http://www.atmel.com/Images/doc8075.pdf

```
ASM(
   "ldi  r30, 0xC1           \n\t"
   "clr  r31                 \n\t"
   "ldi  r24, 0x98           \n\t"
   "st   Z+, r24             \n\t"
   "ldi  r24, 0x06           \n\t"
   "st   Z, r24              \n\t"
   "ldi  r24, 0x17           \n\t"
   "std  Z+2, r24            \n\t"
   "clr  r24                 \n\t"
   "std  Z+3, r24            \n\t"
   );
```

This results in an -Os reduction by a mere 2 bytes (= 1 instruction). I could shorten it slightly more by omitting the "clr r24" and instead doing "std Z+3, r1", but this is hardly worth the effort. Apparently I am overdoing this. A simple calculation tells me that the Z pointer and static addressing generate the same code size when writing to three registers. I don't think I can manage to reduce the ATmega code size any more than this. After some consideration, I revert to static addressing as I like this notation better than defining a struct and using C pointers. Version AS1e is in the appendix.

I mentioned this in an AVRfreaks thread and it seems that when testing on his own, user clawson received the following result with both -Os and -O3: (48) [61]

Struct-based: Does STD (store at Z+displacement) and uses 18 bytes, 13 cycles.

Absolute: Does STS (store at harcoded address) and uses 22 bytes, 11 cycles.

It seems that he too wasn't able to make it use ST+ (store at Z with post-increment). I suppose the compiler doesn't have a heuristic for identifying this.

6.3 ATxmega128A1 analysis

6.3.1 Getting the base serial port routines in place

AS2 started as an empty project that compiled to 544 bytes program code (text) (-O1). It contains a 500 byte interrupt vector, proceeds with clearing R1 and SREG, sets SP via Y, clears EIND and RAMPD/X/Y/Z, and the (at this point nonexisting) global variables, calls board_init(), clears R24 and R25 (W), disables global int, and does eternal loop.

Then I added the "System Clock Control (service)" (49) [62] using ASF Wizard (the interrupt handling is already part of the custom board template) and made the following additions:

In board_init(): call sysclk_init();

In conf_clock.h: change to
#define CONFIG_SYSCLK_SOURCE   SYSCLK_SRC_XOSC
#define CONFIG_XOSC_RANGE   XOSC_RANGE_2TO9

[61] http://www.avrfreaks.net/index.php?name=PNphpBB2&file=viewtopic&t=140145&postdays=0&postorder=asc&sid=e3717428757c8e0fcac437a5ae45306b
[62] http://asf.atmel.com/docs/3.13.1/xmegaa/html/group__sysclk__group.html

The actual program at this point consists of:

```
/* Initialize clock systems and turn off all peripherals */
sysclk_init();

/* Turn on power to USARTE0 */
PR_PRPE = (1 << PR_TWI_bp)|(1 << PR_USART1_bp)|(1 << PR_SPI_bp)|(1 << PR_HIRES_bp)|(1
<< PR_TC1_bp)|(1 << PR_TC0_bp);

/* Enable all interrupt levels */
irq_initialize_vectors();
```

AS2 now compiled to:
1002 text , 0 bss (-O0)
638 text, 0 bss (-O1)
636 text, 0 bss (-O2)
636 text, 0 bss (-O3)
636 text, 0 bss (-Os)

It turns off power to all peripherals, waits for XOSC to become ready, enables XOSC source, saves SREG, disables interrupts, disables the previous clock source, and restores SREG. (Finally it turns on power to USARTE0 and enables all interrupt levels (low, medium, and high).)

6.3.2 Trying out ASF - will it eliminate or reduce the need to read the datasheets?

It's now time to include the ASF USART library driver. This is my first real attempt at fully using the ASF documentation and code. I noticed that once you have added the documentation modules to your project, the links to the documentation can't be reached from the ASF Wizard. Before I realized that they can now be found in the top-right window nambed "ASF Explorer", I removed the modules from the project, clicked on the links, and then re-added the modules to use the documentation. Being somewhat annoyed by this, I decided to reference the XMEGA A Documentation (12) [63] so that I will find it easily the next time. Then I read "AVR4029: Atmel Software Framework - Getting Started" (37) [64] and realized my mistake…

The two available modules are:
USART - Serial interface (service) (50) [65] (51) [66]
USART - Universal Synchronous/Asynchronous Receiver/Transmitter (driver) (52) [67]

According to the ASF Wizard, the first one is a generic abstraction layer (wrapper) that's using the second one (which is included in the first module):

Service function "usart_serial_init(USART_SERIAL, &usart_options)" calls "sysclk_enable_module(SYSCLK_PORT_E,PR_USART0_bm)" and then "usart_init_rs232(usart, &usart_rs232_options)".

[63] http://asf.atmel.com/docs/3.13.1/xmegaa/html/index.html
[64] http://www.atmel.com/Images/Atmel-8431-8-and32-bit-Microcontrollers-AVR4029-Atmel-Software-Framework-User-Guide_Application-Note.pdf
[65] http://asf.atmel.com/docs/3.13.1/xmegaa/html/serial_quickstart.html
[66] http://asf.atmel.com/docs/3.13.1/xmegaa/html/serial_use_case_1.html
[67] http://asf.atmel.com/docs/3.13.1/xmegaa/html/xmega_usart_quickstart.html

Driver function “usart_init_rs232(USART_SERIAL, &USART_SERIAL_OPTIONS)” calls “sysclk_enable_peripheral_clock(usart)”, does the USART settings, and enables receiver and transmitter.

Please note that “sysclk_enable_peripheral_clock(usart)” calls “sysclk_enable_module(SYSCLK_PORT_E, SYSCLK_USART0)”. In other words, the wrapper turns on the USART clock twice. On the other hand, the ASF quick start document for the driver instructs you to call the sysclk_enable_module function before calling the usart_init_rs232 function, to the same effect. Maybe the optimizer discovers this and removes the superfluous one, but it demonstrates a problem with abstraction; it can hide the real world a bit too well.

I suppose that the reason for the first layer of abstraction (the driver) is to make it possible to write microcontroller programs without knowing the hardware details. Similarly, the second layer of abstraction (the service) is probably created to give the programmer one common API regardless of which type of microcontroller (AVR 8-bit ATxmega, AVR32, or ARM) you are working with. The question is whether these two abstractions really make it unnecessary to study the datasheets (as I would like) or if it just adds one (two?) entire additional terminology for the programmer to master. I will return to this topic.

While writing this, I notice that the ASF documentation pages referenced above have hung my Firefox browser.

Also, when googling (now using IE) for the correct syntax (not included in the ASF documentation) for what I figured out should be “sysclk_enable_peripheral_clock (&USARTE0)”, my third hit was an AVRFreaks user forum thread from December 2011 where two different people summarize their ASF experience:

"I think I have saved no time at all using this Atmel framework. It has been a complete pain. Totally frustrated with it." (User name TrevorWhite)

"The ASF could be documented better... But if you spend some time with it looking over the source it becomes clearer. As far as the TCs go, lately I find myself writing my own code rather than using the TC driver in the ASF." (User name GTKNarwhal) Both quotes found at (53) [68].

Appendix A.7.4 shows that the ASF project counter shows the sum of all releases’ counts (most of which are in all essence the same code). It would behoove Atmel to correct this miscalculation.

Anyway, with the help from the ASF quick start guide I have presumably successfully configured and initialized my serial port. I have added the following statements to the Initialization function:

```
/* Initialize USARTE0 */
static usart_serial_options_t usart_options = {
		.baudrate = USART_SERIAL_BAUDRATE,
		.charlength = USART_SERIAL_CHAR_LENGTH,
		.paritytype = USART_SERIAL_PARITY,
		.stopbits = USART_SERIAL_STOP_BIT
};
usart_serial_init(USART_SERIAL, &usart_options);
```

[68] http://www.avrfreaks.net/index.php?name=PNphpBB2&file=printview&t=115038&start=0

AS2 now compiles to:
3176 text , 10 data, 0 bss (-O0)
2186 text, 10 data, 0 bss (-O1)
2258 text, 10 data, 0 bss (-O2)
2434 text, 10 data, 0 bss (-O3)
1874 text, 10 data, 0 bss (-Os)

The service contains four more documented functions, to send and receive one or several characters. As my test program does non-interrupt-based sending, I add this to the main loop in order to send one character:

```
uint8_t received_byte = 0;
usart_serial_putchar(USART_SERIAL, received_byte);
```

This brings us to:
3862 text , 10 data, 0 bss (-O0)
2200 text, 10 data, 0 bss (-O1)
2272 text, 10 data, 0 bss (-O2)
2448 text, 10 data, 0 bss (-O3)
1898 1908 text, 10 data, 0 bss (-Os)

These numbers are so ridiculously big that I think I must have made a mistake. (No ISR nor application code yet.) I therefore include the code in the appendix (AS2a) and create a fresh project. Then I add the three ASF modules mentioned above, update conf_clock.h, and paste the AS2a code into the new main file. Nope. Still the same size. Something is definitely wrong here.

I come to think of the fact that I haven't told the compiler about the crystal properties (and the PLL settings of conf_clock.h mentions BOARD_XOSC_HZ), so I look for it in the System Clock Control ASF documentation (54) [69] but it only mentions that it needs to be defined in conf_board.h. After googling, I found some demo board code (55) [70] that I change and add:

```
#define BOARD_XOSC_HZ   8000000UL
#define BOARD_XOSC_TYPE   XOSC_TYPE_XTAL
#define BOARD_XOSC_STARTUP_US   2000
```

(The demo board documentation actually says "#define BOARD_XOSC_HZ   8000000", which I think will result in a data type error, but I am not sure.)

It now compiles to:
3906 text , 10 data, 0 bss (-O0)
2218 text, 10 data, 0 bss (-O1)
2296 text, 10 data, 0 bss (-O2)
2472 text, 10 data, 0 bss (-O3)
1922 text, 10 data, 0 bss (-Os)

[69] http://asf.atmel.com/docs/3.13.1/xmegaa/html/group__clk__group.html
[70] http://194.19.124.62/docs/latest/xmega.drivers.des.unit_tests.xmega_a1_xplained/html/group__atxmega128a1__xpld__config.html

This is surprising. I have only included initialization of the serial port and a dummy transmit statement. No receiver interrupt handler (can't find one in the ASF library) and no real program code.

I think that maybe this immense size is due to the generic, cross-platform service so I change calls from

```
usart_serial_init(USART_SERIAL, &usart_options);
usart_serial_putchar(USART_SERIAL, received_byte);
```

to the direct driver calls:

```
sysclk_enable_module(SYSCLK_PORT_E, PR_USART0_bm);
usart_putchar(USART_SERIAL, received_byte);
usart_init_rs232(USART_SERIAL, &USART_SERIAL_OPTIONS);
```

This brings a slight reduction in size:
3600 text , 10 data, 0 bss (-O0)
2136 text, 10 data, 0 bss (-O1)
2214 text, 10 data, 0 bss (-O2)
2390 text, 10 data, 0 bss (-O3)
1840 text, 10 data, 0 bss (-Os)

Finally, I remove the USART service from the project to see if it gets compiled in even though I wasn't calling it, but the code size is the same. The good news is that the cross-platform code-size overhead is only 1922-1840 = 82 bytes for whatever gets compiled in additionally with the service, so making a custom driver ARM-compatible shouldn't be so "expensive". The bad news is that it's the ATxmega-specific driver that's the culprit. This means that there's very little point in using the ASF USART code for anything but a copy&paste source of sample code bits into a custom driver.

Just to verify that I haven't made a mistake, I create a New Example Project based on "USART Example - STK600 - ATxmega128A1". It is a simple polling program that returns the incoming serial data, based on the "driver" code. -Os compilation yields 2102 text, 30 data. Apparently my AS2a implementation isn't an anomaly in the ASF world.

I continue to look for an example project with interrupt-based receiver. This is a slow and irritating process. I give up after looking at a handful of projects and google for "atmel asf xmega usart interrupt". I find application note AVR1522 (56) [71] [72], which is a demo for the XMEGA-A1 Xplained board. It has both polled and interrupt-based receiver, so hopefully it can be used. After asking for ASF support for serial port interrupt-based receiver at www.avrfreaks.net, I now know that there is only an ASF general-purpose FIFO queue:

"No. There is a FIFO service in the ASF that you can use, but you will have to write the ISR(s) to make this happen. ASF examples, at least as far as the USARTs are concerned, are very simple." (User name GTKNarwhal) (57) [73]

The PMIC (Programmable Multi-level Interrupt Controller) Quick Start guide (58) [74] has an ISR skeleton code sample, so it's possible to copy it and add the necessary content, but calling generic ASF FIFO buffer code from within an interrupt service routine seems like a very bad idea.

[71] http://www.atmel.com/Images/doc8408.pdf
[72] http://www.atmel.com/Images/AVR1522.zip
[73] http://www.avrfreaks.net/index.php?name=PNphpBB2&file=viewtopic&t=139232

It's time to answer the question in this section's header; does ASF eliminate or reduce the need to read the datasheets? Unfortunately, the answer is no, at least for the USART and clock management. I fear that there are clock and other settings that I have missed. At least I am very uncertain if the ones I have defined are the correct ones, with the correct value, and all the necessary ones. The Quick Start guide for System clock Management has a note that says *"For user boards, BOARD_XOSC_HZ should be defined in the board conf_board.h configuration file as the frequency of the crystal attached to XOSC."* (59) [75] and that's it. This and two other defines are mentioned in the "XMEGA-A1 Xplained Board Configuration" (55) [76] that I found by googling as mentioned above.

While trying to look at the ASF USART modules' Quick Start and API Documentation it hangs my Firefox browser again. And… now it crashed.

There's another problem with the USART documentation: It fails to mention that the USART transmitter pin must be set to output (and the receiver to input, which is the startup default). The closest I come is the text *"port_driver_group for peripheral io port control."* listed under dependencies in the driver's API Documentation (60) [77]. It's not included in the Quick Start code samples for either of the two modules, so that code actually won't work. It could be that the person(s) who wrote the USART Quick Start guides had all pin settings in a board_conf.h file and therefore forgot to include them in the guide, but that would mostly serve as an illustration to the problem of having too many includes.

The ATxmega AU manual's USART section is clear, though. It tells me what I need to know.

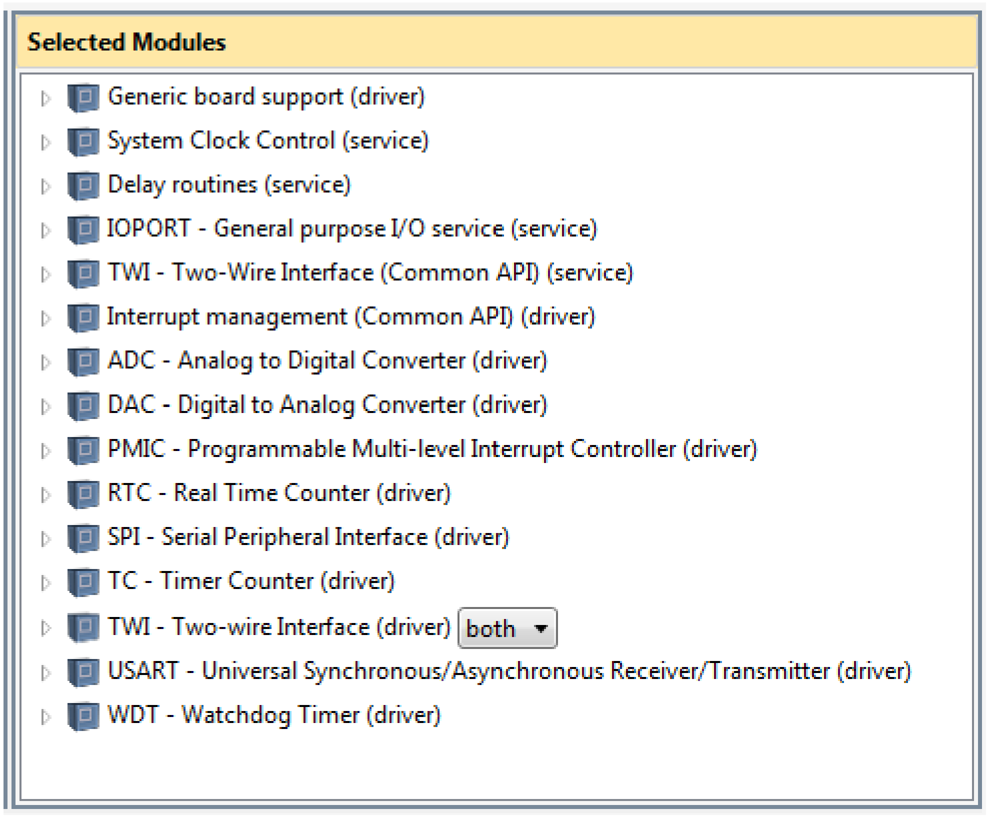
6.3.3 Checking how big the minimum ASF library would be for my real application

Before I proceed with my 100% custom driver based on Atmel application note AVR1522, I decide that I want to see how big the ASF library code would be for my actual application. It is using the following peripherals:

- Timer/counter for HW PWM
- USART for serial ports
- Analog to Digital conversion for voltage and current measuring
- Digital to Analog conversion for linear DC control
- External interrupts for events
- TWI (I^2C) for communication with peripherals
- SPI for communication with peripherals
- RTC (Real-Time Clock)
- General-purpose I/O for sensing and controlling the board
- Watchdog timer

[74] http://asf.atmel.com/docs/3.13.1/xmegaa/html/xmega_pmic_quickstart.html
[75] http://asf.atmel.com/docs/3.13.1/xmegaa/html/sysclk_quickstart.html
[76] http://194.19.124.62/docs/latest/xmega.drivers.des.unit_tests.xmega_a1_xplained/html/group__atxmega128a1__xpld__config.html
[77] http://asf.atmel.com/docs/3.13.1/xmegaa/html/group__usart__group.html

Based on AS2a, I add the relevant ASF modules, which means that the project now incorporates the following:


Selected Modules

- Generic board support (driver)
- System Clock Control (service)
- Delay routines (service)
- IOPORT - General purpose I/O service (service)
- TWI - Two-Wire Interface (Common API) (service)
- Interrupt management (Common API) (driver)
- ADC - Analog to Digital Converter (driver)
- DAC - Digital to Analog Converter (driver)
- PMIC - Programmable Multi-level Interrupt Controller (driver)
- RTC - Real Time Counter (driver)
- SPI - Serial Peripheral Interface (driver)
- TC - Timer Counter (driver)
- TWI - Two-wire Interface (driver) both
- USART - Universal Synchronous/Asynchronous Receiver/Transmitter (driver)
- WDT - Watchdog Timer (driver)

Figure 25: Selected ASF ATxmega128A1 modules

It leads to this:

14522 text , 10 data, 104 bss (-O0)
8762 text, 10 data, 104 bss (-O1)
8830 text, 10 data, 104 bss (-O2)
9954 text, 10 data, 104 bss (-O3)
8334 text, 10 data, 104 bss (-Os)

I try to reduce the code size slightly by setting the TWI module to “master”, but it complains about a missing file, so I have to put it back to “both”. I also deliberately choose the driver version instead of the service whenever possible in order to make this test as “good” as possible for ASF. Similarly, as I haven’t included code that initializes the new modules this is the smallest it could ever become.

All this code does (well, it’s not tested and I know that the transmitter pin needs setting to output, so I would need to add some things) is to initialize the system clock and one USART and send one byte through a serial port. A sneak peak at my subsequent test results tells me that a minimum hand-written C implementation takes -Os 822 bytes, of which the interrupt vector table is 500 bytes and the default initialization and clock management account for 122 bytes. I just got hit with about 8kB of ASF library code, most of which would not be used by the application. With hand-written C code, each individual module should require some 200 bytes, very roughly speaking.

A short note here: I posted the question "Does anybody know if I should do something else so that the compiler removes unused code from the ASF library?" in the user forum at www.avrfreaks.net (61) [78] and received two replies:

"Look at the .map file to find out where the bloat is but be warned that ASF is not designed for efficiency but ease of use and to present a generic interface across architectures. This results in sub-optimal code." (user name clawson, later refered to as Cliff)

"This is quite a common question. Basically, ASF is as Cliff says, meant to be a generic interface and have the same abstraction across multiple architectures. It will therefore to assertions, check data validity, transform parameters etc on a level that is not necessary at a single device level.

You are probably also seeing the module interconnectivity of asf, e.g. that modules are depending on functionality of other modules (e.g. usart would need the sysclk and sysclk need powermgmt....), which also lead to inclusion of all the generic code for these modules.

The positive thing (guess positive can be discussed), is that a full-fledged application does not increase the code size as much, as some of the code paths are already present in an already included module. So yes, using only USART may seem very big, but adding CHIPID and GPIO to this will be a small increase. (guess this is more true for SAM than for xMega, but I am most familiar with ASF on SAM and the idea should be similar)." (user name meolsen, Atmel employee according to the user profile)

First of all: these replies seem to come from people who generally know what they are talking about (as is most often my experience from these user forums).

Second: judging by my simple USART service vs. driver code size test, only 82 of the 1922 bytes come from the "generic interface". This is equal to (slightly more than) 82 / (1922 – 622) = 6,3%. (Later during the ATxmega analysis, we'll see that the interrupt vector and (slimmed) clock configuration take 622 bytes. In other words, there's reason to question the assumption that much of the ASF excess baggage comes from cross-platform transformation code. At least for the USART modules this doesn't seem to be the case. Most of it is caused by the fact that the ASF modules aren't written in such a way that unused code gets eliminated by the compiler. I have only looked at the compiled code sizes (i.e. not analyzed the .map file as clawson suggested) and I have only looked closer at the USART module. However, by the increase in total code size just by including ASF modules documented above, I think it is reasonable to believe that this ailment is common to all or much of the ASF code.

Third: I do agree that some of the library code would be used by the real application, but in my experience not even remotely close to this size. E.g. my application will use two serial ports for sending binary data, each having an ISR (Interrupt Service Routine) that either places the incoming data in a generic circular buffer or in a protocol-bound vector. The application will not change clock frequencies and the serial port configuration will not change. No matter how much my application grows, it will never use more of this ASF library code. The same is true for the other HW modules and peripherals. They are all used for one specific purpose, generally hardwired to another HW component.

Fourth: in my opinion, the beauty of microcontroller development is the fact that you are coding directly against the actual hardware. What I want is a tool that makes this as easy as possible, so that I wouldn't have to consult the datasheets and application notes so much. If I could make a wish, it

[78] http://www.avrfreaks.net/index.php?name=PNphpBB2&file=viewtopic&p=1126226#1126226

would be for a GUI tool that generates HW-specific driver code for you, based on input from the developer. In fact, BASCOM-AVR is doing this with built-in commands and their parameters that (behind the scene) lay out hand-written assembly based on these parameters.

With this I end my investigation of ASF. Perhaps my serial port-based test program (decided before I had looked at what ASF offers) was an unfortunate choice for ASF. For example, the ASF ADC module quick start guide is informative and (by a quick look) seems user-friendly and efficient to use. Also, the “System Clock Control (service)” did only result in a little dead code, even though it isn’t sufficiently documented. Still, as this test shows, generally speaking ASF comes at a code size cost that I am not willing to pay. Maybe this is part of the reason why Atmel doesn’t have much ASF support for ATmega. Many of these microcontrollers are very usable in 8kB program size (or even less), but there wouldn’t be much place for the user’s own code after ASF moved in.

Due to the excess ASF code, I can only use it as a copy&paste source, probably after looking for application notes describing the module on which I am developing. This will have the good side-effect of letting me create my entirely own inclusion tree, giving me a complete overview of what’s actually being done. But, <u>there is no productivity enhancement in sight</u>. It should also be mentioned that ASF can be used as a copy&paste source in two ways, both of which are better done in a separate project; you can use the ASF Wizard to include modules into your temp project so that you can drill down and copy the code and headers to your real project. Alternatively, you can look for an ASF Example Project that might (or might not) contain valuable sample code. In this case you end up with a number of unwanted projects that fill up your work folder:

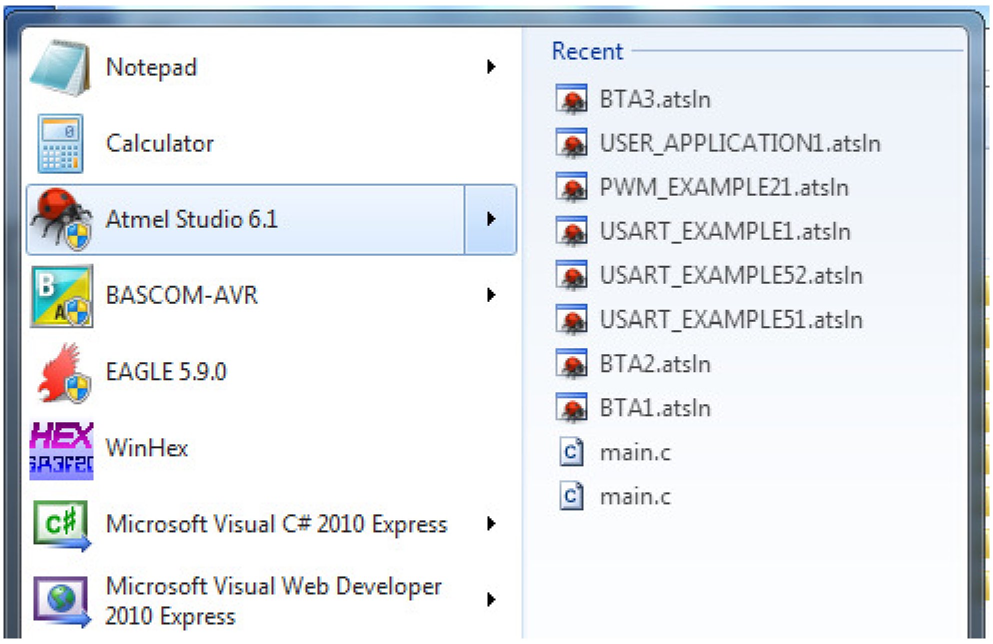


Figure 26: Sample project residue

6.3.4 Using application note AVR1522 USART driver

I start a new project named AS3 and copy the test program application code from the ATmega AS1b (which is the version using the generic circular buffer based on application note “AVR306 Using the AVR UART in C” (47) [79] [80]).

For the ATxmega USART, I look in two application notes: “AVR1522 XMEGA-A1 Xplained Training - XMEGA USART” (56) [81] [82], which is largely a copy of “AVR1307 Using the XMEGA USART” (62) [83] [84]. The main advantage of the first one is that it is using AVR GCC syntax while the second has IAR Systems syntax, but the zip file belonging to AVR1307 has a valuable tool: an Excel sheet baud rate calculator. The formulas can be found on p282 in the AU manual (63) [85]. The test board has an 8 MHz crystal (and as before we use 15 625 baud and asynchronous normal mode). There are several USART settings that result in the above and I’m using BSCALE = 0, BSEL = 31, and CLKX2 = 0.

Oddly enough, the AVR1522 polling example isn’t using the usart_driver library functions for configuration, while its interrupt example does. The original code in AVR1307 is consistently using the library code it is there to present. Anyway, after replacing much of the polling configuration code with the AVR1522 interrupt configuration code, a clear design idea can be seen. This sample code also takes care of the pin settings, that the ASF module library failed to do. Unfortunately, both application note sample program are omitting the system clock settings, so at first nothing works.

I include the ASF module System Clock Control (service) and modify conf_clock.h so that it contains the following definitions:

```
#define CONFIG_SYSCLK_SOURCE     SYSCLK_SRC_XOSC
#define CONFIG_SYSCLK_PSADIV     SYSCLK_PSADIV_1
#define CONFIG_SYSCLK_PSBCDIV    SYSCLK_PSBCDIV_1_1
#define BOARD_XOSC_HZ            8000000UL
#define BOARD_XOSC_TYPE          XOSC_TYPE_XTAL
#define BOARD_XOSC_STARTUP_US    2000                 // What should this value be?
#define CONFIG_XOSC_RANGE        XOSC_RANGE_2TO9
```

In my initialization function I start with

sysclk_init();
sysclk_enable_module(SYSCLK_PORT_E, PR_USART0_bm);.

Now it’s working and compiles to:
2114 text , 0 data, 47 bss (-O0)
1156 text, 0 data, 47 bss (-O1)
1144 text, 0 data, 47 bss (-O2)
1130 text, 0 data, 47 bss (-O3)
1136 text, 0 data, 47 bss (-Os)

[79] http://www.atmel.se/Images/doc1451.pdf
[80] http://www.atmel.com/images/avr306.zip
[81] http://www.atmel.com/Images/doc8408.pdf
[82] http://www.atmel.com/Images/AVR1522.zip
[83] http://www.atmel.com/Images/AVR1522.zip
[84] http://www.atmel.com/Images/AVR1307.zip
[85] http://www.atmel.com/Images/Atmel-8331-8-and-16-bit-AVR-Microcontroller-XMEGA-AU_Manual.pdf

Compare the AS3a version to the worst BASCOM-AVR high-level-only implementation (BA1a) at 1720 and best (BA6a) at 1444 bytes text.

I wonder why Atmel didn't include the application note code in ASF, as AVR1307 was released in February 2008.

Anyway, the next step is to see what happens if I copy all used code from the USART library files into the main file. AS3b is included in the appendix.

1944 text , 0 data, 41 bss (-O0)
1118 text, 0 data, 41 bss (-O1)
1060 text, 0 data, 41 bss (-O2)
1054 text, 0 data, 41 bss (-O3)
1064 text, 0 data, 41 bss (-Os)

Much of the reduction from 1136 to 1064 bytes is caused by exclusion of some struct fields relating to transmitter interrupt buffer and fewer function calls.

6.3.5 Adjusting the ATmega static addressing code for ATxmega

I was quite happy with the ATmega code with circular receiver buffer based on AVR306. What if I insert the ATxmega-specific code into it and clean up the ATxmega initialization?

The result can be seen in AS3c in the appendix. This contains three types of changes:

- The S&Ps are replaced by static addressing.
- The USART initialization has been cleaned up. -Os becomes 1014 text, 37 bss.
- The USARTE0 power-on call was replaced with a register write without preserving the interrupt flags.

We are now at:
1648 text , 0 data, 37 bss (-O0)
1010 text, 0 data, 37 bss (-O1)
976 text, 0 data, 37 bss (-O2)
970 text, 0 data, 37 bss (-O3)
990 text, 0 data, 37 bss (-Os)

I then dissected sysclk_init() and (by reading the ATxmega AU manual) extracted the things really needed at initialization. This reduced -Os to about 964-968 text. (I didn't record this state.) Then I included soft reset by jumping to the reset vector address. (Unlike the ATmega, the ATxmega can be software-reset by writing to a register. For cross-platform compatibility, I stay with the ATmega approach at least for now.) I also disable interrupts before sending the error message, no longer set the receiver pin to input as it's the startup default, and clear SREG at the end of initialization. These changes landed me at -Os 974 text. I noticed that the loop that disables all peripheral clocks do repetitive STS calls instead of using ST Z+. I would like to understand how to instruct the compiler to do it from C, but for now I settle with inline assembly. This seems to be the smallest implementation of one serial port on ATxmega.

Note that -Os (optimize for size) produces bigger code than -O2 and -O3:

1332 text , 0 data, 37 bss (-O0)
980 text, 0 data, 37 bss (-O1)
950 text, 0 data, 37 bss (-O2)
944 text, 0 data, 37 bss (-O3)
964 text, 0 data, 37 bss (-Os)

AS3d is included in appendix B, both C and disassembly. This will be my baseline when I go in a few different directions:

6.3.5.1 Only clock config with transmit and ISR-based circular buffer receiver

I strip all the application code so only the following remains (including clock and USART configuration):

- ISR-based circular buffer receiver
- Function to check if there is data in the circular buffer
- Function to read a byte from the circular buffer
- Function for transmission
- Simple check for data and transmission of read byte

This is just to get a definite figure of how big the generic one-port USART driver really needs to be:

1094 text , 0 data, 18 bss (-O0)
816 text, 0 data, 18 bss (-O1)
806 text, 0 data, 18 bss (-O2)
806 text, 0 data, 18 bss (-O3)
822 text, 0 data, 18 bss (-Os)

6.3.5.2 Only clock config

AS3e is included in the appendix, with the USART code commented out. It only does clock management:

708 text , 0 data, 0 bss (-O0)
618 text, 0 data, 0 bss (-O1)
622 text, 0 data, 0 bss (-O2)
622 text, 0 data, 0 bss (-O3)
622 text, 0 data, 0 bss (-Os)

For the sake of simplicity, let’s say that the USART functionality requires 200 bytes, and that the interrupt vector table and system clock configuration “offset” is 622 bytes.

6.3.6 Adding one more serial port

6.3.6.1 Simple two-dimensional array and if statement for port selection

I noticed that I had made a slight mistake with the variable SerialByte. By changing it from global to local -Os becomes 960 text, 36 bss.

The first dual-port version compiles to -Os 1268 text, 0 data, 72 bss. The second serial port adds 304 text, 36 bss, while the first cost 338 text, 36 bss. I notice that the function call to ReceiveSerial starts with PUSHing eleven registers. When inlining USART_Receive and DataInReceiveBuffer, -Os becomes 1190 text, 0 data, 72 bss.

By using a two-dimensional array for SerialData, much of the high-level code complexity can be avoided. However, indexing requires a few calculations. When I looked at the disassembly I realized that I had made a mistake; I had put the column before the row, which means that the two-dimensional array was in fact column-major addressed. I switched order so that I got row-major addressing, which surprisingly resulted in -Os 1204 text, 0 data, 72 bss:

Column-major:	Row-major:
1852 text , 0 data, 72 bss (-O0)	1884 text , 0 data, 72 bss (-O0)
1192 text, 0 data, 72 bss (-O1)	1216 text, 0 data, 72 bss (-O1)
1192 text, 0 data, 72 bss (-O2)	1210 text, 0 data, 72 bss (-O2)
1218 text, 0 data, 72 bss (-O3)	1220 text, 0 data, 72 bss (-O3)
1190 text, 0 data, 72 bss (-Os)	1204 text, 0 data, 72 bss (-Os)

Version AS3f is included in appendix B, with disassemblies before and after inlining, with row-major and column-major array addressing. I would have liked to analyze this, but I must press on.

The question is: is this better or worse than placing the global variables in structs and sending pointers for parameters?

6.3.7 Structs and pointers

I retrace my steps to the AVR1522 application note and its S&P-based design.

AS3g compiles to:
1980 text , 0 data, 80 bss (-O0) (without inlining of USART_RXComplete)
1248 text, 0 data, 80 bss (-O1)
1246 text, 0 data, 80 bss (-O2)
1316 text, 0 data, 80 bss (-O3)
1218 text, 0 data, 80 bss (-Os)

Without inlining of USART_RXComplete -Os compiles to 1250 text. This is because the register PUSHing and POPing when entering and exiting the ISR is more expensive than the code itself.

I tried to place the contents of USART_RXComplete in each ISR, but this didn't affect the total size (compared to inlined USART_RXComplete). While this might seem obvious, I wanted to make extra sure that it isn't possible to reduce it further. I also tried to move the vector and fields of struct USART_Buffer_t into struct USART_data_t. This didn't make any difference either (so a multilevel struct hierarchy doesn't seem to incur a penalty).

The BASCOM-AVR scaling tests only covered the bare sending routines (not receiver ISR and application code), but the results are similar. The general-purpose ISR circular receiver buffer for two serial ports requires slightly less program memory for static addressing with two-dimensional array than the "structs and pointer"-based version. For some reason this column-major array implementation is yet a bit more efficient from a code-size point of view.

BASCOM-AVR has commands and configuration options that automatically generate and use the circular buffer. This hides the ISR buffer from the developer, which from a high-level perspective makes it less obvious that you spend some program code, SRAM, and clock cycles on moving the received bytes from the circular buffer to the actual work area. I don't know how common it is to develop a protocol-bound serial port ISR routine and vector, but my protocol benefits from it.

It's now time to see how the S&P approach scales. Version AS3h is a fully developed test program for 1-3 ATxmega ports and 1-2 ATmega ports. It's included in the appendix, with -Os disassemblies of both types.

6.4 Scaling ATmega 324A, ATmega1284, and ATxmega128A1

I have omitted optimization level -O0 as it isn't a usable alternative.

6.4.1 Structs and pointers (AS3h)

6.4.1.1 ATmega324A total size

Atmega324A, interrupt vector table 126 bytes

Ports	1				2		
Opt	Text	Data	BSS	Delta	Text	Data	BSS
-O1	574	0	40	134	708	0	80
-O2	642	0	40	66	708	0	80
-O3	786	0	40	160	946	0	80
-Os	550	0	40	130	680	0	80

Atmega324A, interrupt vector table excluded

-O1	448	582
-O2	516	582
-O3	660	820
-Os	424	554

Note the surprising -O2 and-O3 deltas when going from 1-2 ports.

6.4.1.2 ATmega1284 total size

Atmega1284, interrupt vector table 140 bytes

Ports	1				2		
Opt	Text	Data	BSS	Delta	Text	Data	BSS
-O1	604	0	40	142	746	0	80
-O2	678	0	40	68	746	0	80
-O3	822	0	40	162	984	0	80
-Os	580	0	40	138	718	0	80

Atmega1284, interrupt vector table excluded

-O1	464	606
-O2	538	606
-O3	682	844
-Os	440	578

6.4.1.3 ATxmega128A1 total size

Atxmega 128A1, interrupt vector table 500 bytes

Ports	1				2				3		
Opt	Text	Data	BSS	Delta	Text	Data	BSS	Delta	Text	Data	BSS
-O1	1060	0	40	188	1248	0	80	174	1422	0	120
-O2	1078	0	40	168	1246	0	80	168	1414	0	120
-O3	1054	0	40	262	1316	0	80	266	1582	0	120
-Os	1050	0	40	168	1218	0	80	168	1386	0	120

Atxmega 128A1, interrupt vector table excluded

-O1	560	748	922
-O2	578	746	914
-O3	554	816	1082
-Os	550	718	886

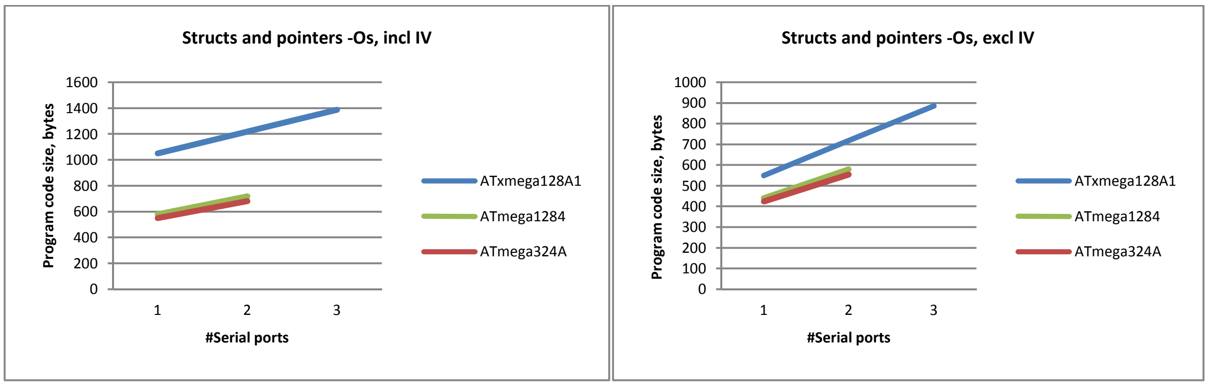


Figure 27: Structs and pointer scaling

This is surprising. I had expected an offset caused by the differently sized IVs (Interrupt Vector tables), but even without it there is a significant difference between ATmega and ATxmega:

- 2 serial ports: ATxmega128A1 is 718-554=164 bytes bigger than ATmega324A
- 1 serial port: ATxmega128A1 is 550-424=126 bytes bigger than ATmega324A
- Fictive 0 serial ports: ATxmega128A1 is 126-(164-126)=88 bytes bigger than ATmega324A

As we see in the tables on the previous page, the various optimization levels don't scale uniformly, but for -Os let's roughly say that an additional port (including both application code and driver) requires 164-126=38 more bytes for ATxmega128A1 than for ATmega324A. As the high-level application code is identical, we could at least assume that it's the driver itself that is more "expensive". It we subtract 38 from 126 we get a fictive 0-port difference of 88 bytes. Just looking at the high-level code, it's not possible to see where the 38 bytes come from. I can only find this:

- ATxmega needs to set TX pin as output, while ATmega does this automatically when enabling the USART.
- The ATxmega USART config writes to one more register than the ATmega counterpart (which is partly specific to my settings).

Could it be that the compiler is including support for the EIND and RAMP/D/X/Y/Z registers? I added support for the 128kB ATmega:

- 2 serial ports: ATxmega128A1 is 718-578=140 bytes bigger than ATmega1284
- 1 serial port: ATxmega128A1 is 550-440=110 bytes bigger than ATmega1284
- Fictive 0 serial ports: ATxmega128A1 is 110-(140-110)=80 bytes bigger than ATmega1284

Slightly different results, but

The -Os deltas are:

- ATmega324A: 130 bytes
- ATmega1284: 138 bytes
- ATxmega128A1: 168 bytes

I will have to look at the disassemblies to see what is going on.

An 88- or 80-byte initial offset is more credible, as the ATxmega clock, peripheral module power, and interrupt level configuration is much more extensive than ATmega's.

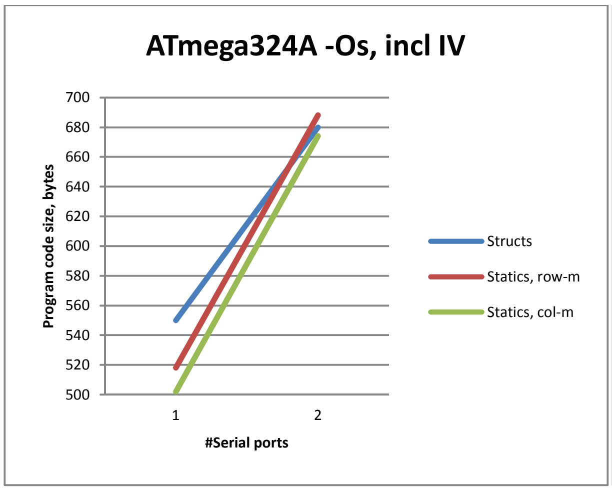
6.4.2 Statics

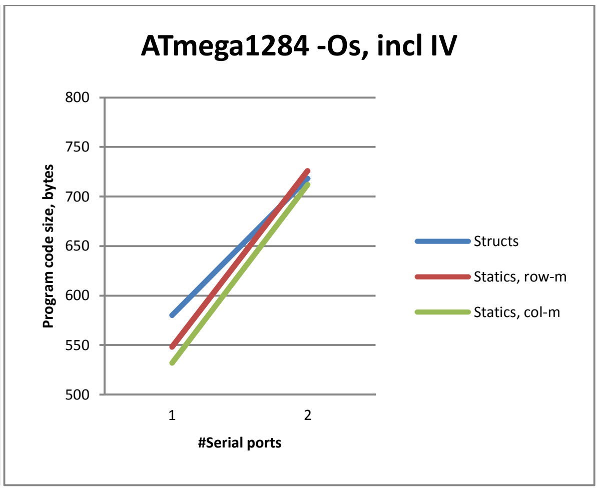
The test results are listed in appendix A.8. Generally speaking, the "statics, row-major" and "statics, column-major" graphs are very similar to the S&P graphs above.

6.4.3 Code size comparative graphs

From an ATmega code size perspective, static addressing is clearly better when using only one serial port. At two ports it's a tie. Column-major array addressing is consistently better than row-major (which is the C "default"). With three or more ports, S&P become increasingly better. A small note: row-major addressing might be better in the long run, as successive reads from one port's "working area" can use post-increment, which is more efficient than adding to the pointer or using displacement. This is why I choose to concentrate on row-major.

The ATxmega128A1 graph indicates something quite unexpected: the lines don't seem to be crossing. At least up to and including three ports, static addressing is better. Please bear in mind that after initialization, the only real high-level differences are the register addresses to exactly the same kind and number of registers. I had expected the ATxmega128A1 to scale very similarly to the ATmega1284. There is a fluctuation in the deltas, which indicates that the compiler, while adhering to its rules, might produce results that seem a bit random. This is pure speculation in an effort to make sense of the test results. We'll soon see.





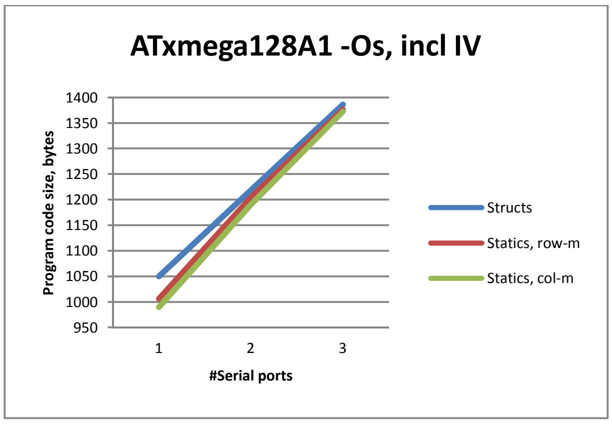


Figure 28: Scaling #ports

6.4.4 Code size and clock cycle count ISR, Interrupt Service Routine

6.4.4.1 Statics row-major

Please see appendix A.8.3.1 for the test result data. The clock cycle count on the static row-major ISR routine shows two reasons for the difference in code size (and clock cycles):

- Devices with >64kB program memory must use the RAMPX/Y/Z registers for the >16bit part of the address. This doesn't apply for ATmega324A but for the other two.
- Devices with support for EBI (External Bus Interface) can address more than 64kB of data memory. So long as the application only uses internal SRAM, it is far from reaching this limit. Of the microcontrollers in this test, this is only relevant for the ATxmega128A1. Its internal SRAM starts at 0x2000 (8kB) and ends at 16kB (0x4000).

I did a quick search for a way to tell the compiler that only internal data memory is used, but I only found one web page that expressly deals with it: (64) [86]

"Unfortunately the only way to do this is to use the "naked" function attribute on your ISRs, but then you'll have to take care of doing the ISR prologue and epilogue yourself.

[86] http://avr.2057.n7.nabble.com/How-can-I-turn-off-gt-64K-ram-support-for-ATxmega128a1-target-td10341.html

If you compile using a different architecture (that doesn't have > 64K RAM), then you'll have to deal with including the correct IO header file, and possible other issues."(User name Weddington, Eric)

"It seems the gcc does what I want if I use the atxmega128a3 target with the atxmega128a1 IO header file. The XMEGA manual doesn't say if the RAMP registers are cleared by reset, so to be sure I clear the RAMP* registers in init2, before first RAM access." (User name Rothe, Michael, who asked the original question)*

I suppose that the compiler keeps track of the RAMPD value and thereby knows that it never changes from zero, so that the actual application code isn't bothered by this. It probably generates a set of default ISR entry and exit operations. If this is the case, only the ISRs will suffer from the 5-word RAMPD waste.

I first thought that there's a bug in the ATmega1284 ISR entry. It PUSHes RAMPZ but doesn't clear it afterwards like the ATxmega128A1 code. After looking at the ATmega1284 datasheet, it seems that the RAMPZ register is only relevant for the ELPM/SPM program memory instructions.

Another thing is also clearly seen: A number of instructions have different implementations in ATmega and ATxmega. Mostly this seems to be to ATxmega's advantage and in the PUSH-frequent ISRs this means that although it is six instruction words larger, it takes only one more clock cycle. If it were possible to instruct the compiler to disable EBI support (so that RAMPD would only be cleared in the default initialization), the ATxmega128A1 ISR could actually be five clock cycles (9%) faster than the ATmega1284. (It could of course be done with inline assembly.) As it currently is, the ATxmega128A1 incurs a "complexity cost" of 18% over the program memory-wise equally-sized ATmega1284 microcontroller.

6.4.4.2 Structs and pointers

Please see appendix A.8.3.2 for the test result data. The S&P approach additionally uses the X pointer (including RAMPX), which adds five instructions and six clock cycles to ATxmega128A1. This is only part of the difference, however. Let's look at the disassemblies:

6.4.4.3 Structs and pointers function code for inlining into ISR

```
inline void USART_RXComplete(UsartData_t * usart_data)
{
   uint8_t tempRxHead = \
         (usart_data->RxHead + 1) & USART_RX_BUFFER_MASK; // Advance buffer head
   usart_data->RxHead = tempRxHead;              // Store new index
   usart_data->RxBuffer[tempRxHead] = \
         usart_data->Usart->DATA;                // Store received data in buffer
}
```

This becomes the following ATmega324A assembly (with the corresponding figures for ATxmega128A1):

	W	C	Description
LDI R26,0x4E	1	1	X points to RxHead
LDI R27,0x01	1	1	X points to RxHead
LD R30,X	1	1/2	Low(Z) = RxHead
SUBI R30,0xFF	1	1	Low(Z) = RxHead + 1
ANDI R30,0x0F	1	1	Low(Z) = (RxHead + 1) & bitmask
ST X,R30	1	1	RxHead = Low(Z)
LDI R31,0x00	1	1	High(Z) = 0 (Z now contains buffer index)
LDS R26,0x013C	2	2/3	X points to USART reg
LDS R27,0x013D	2	2/3	X points to USART reg
ADIW R26,0x06	1/0	2/0	X points to USART reg -> DATA/UDR
LD R24,X	1	1/2	R24 = received character
SUBI R30,0xC4	1	1	Add buffer base address to Z pointer
SBCI R31,0xFE	1	1	Add buffer base address to Z pointer
STD Z+2,R24	1	2	Store received character in buffer + 2
ATmega total	16	18	
ATxmega total	15	20	

Please note that the ADIW instruction isn't necessary for ATxmega, as its struct starts with the DATA register. In ATmega324/1284, the data register UDR is the sixth in its struct. Also, I fail to see why the compiler didn't include the displacement in the preceding SUBI&SBCI, as the ATxmega would have saved one clock cycle by it.

6.4.4.4 Statics row-major ISR code

```
uint8_t data = UDR1;                              // Read the received data
uint8_t tmphead = \
          ( USART_RxHead1 + 1 ) & USART_RX_BUFFER_MASK; // Calculate buffer index
USART_RxHead1 = tmphead;                          // Store new index
USART_RxBuf1[tmphead] = data;                     // Store received data in buffer
```

This becomes the following ATmega324A assembly (with the corresponding figures for ATxmega128A1):

	W	C	
LDS R24,0x00CE	2	2/3	R24 = received character
LDS R30,0x0101	2	2/3	Low(Z) = RxHead
SUBI R30,0xFF	1	1	Low(Z) = RxHead + 1
ANDI R30,0x0F	1	1	Low(Z) = (RxHead + 1) & bitmask
STS 0x0101,R30	2	2	RxHead = Low(Z)
LDI R31,0x00	1	1	High(Z) = 0 (Z now contains buffer index)
SUBI R30,0xC8	1	1	Add buffer base address to Z pointer
SBCI R31,0xFE	1	1	Add buffer base address to Z pointer
STD Z+0,R24	1	2/1	Store received character in buffer
ATmega total	12	13	
ATxmega total	12	14	

It is clear that the additional operations required by pointing into the struct come at a cost:

ISR actual code	ATmega324A		ATmega1284		ATxmega128A1	
	Instr W	Instr C	Instr W	Instr C	Instr W	Instr C
Statics, row-major	12	13	12	13	12	14
Structs and pointers	16	18	16	18	15	20
Worse %	33.3%	38.5%	33.3%	38.5%	25.0%	42.9%

The above table only contains the actual application code. On average, it's 29% more program code and 41% more clock cycles.

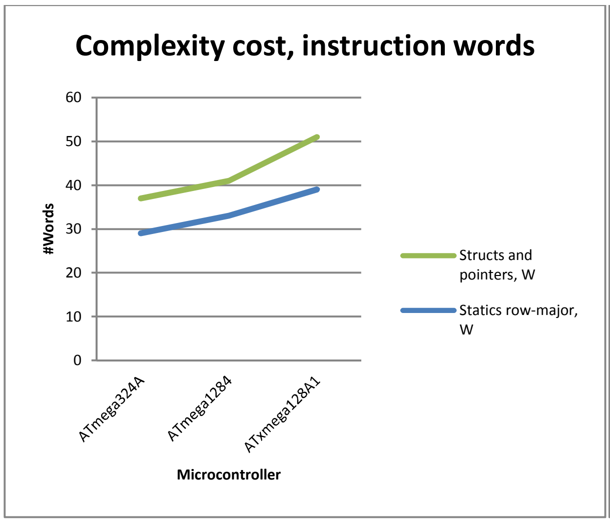
Just to get the complete picture, here are the figures for the entire ISRs:

Complete ISR	ATmega324A		ATmega1284		ATxmega128A1	
	Instr W	Instr C	Instr W	Instr C	Instr W	Instr C
Statics, row-major	29	46	33	53	39	54
Structs and pointers	37	59	41	66	51	72
Worse %	27.6%	28.3%	24.2%	24.5%	30.8%	33.3%

Including the ISR entry and exit code yields a slightly better result. It should be said that the ISR:s contain all code in each one, as the other option (additional function call) resulted in more stack operations plus (R)CALL&RET than was saved by code reuse. In non-ISR code, function calls might work better than here.

A rearrangement of the figures for complete ISR also clearly shows the complexity cost, as we change from smaller and simpler to larger and more complex:

	ATmega324A	ATmega1284	ATxmega128A1
Structs and pointers, W	37	41	51
Statics row-major, W	29	33	39
Structs and pointers, C	59	66	72
Statics row-major, C	46	53	54



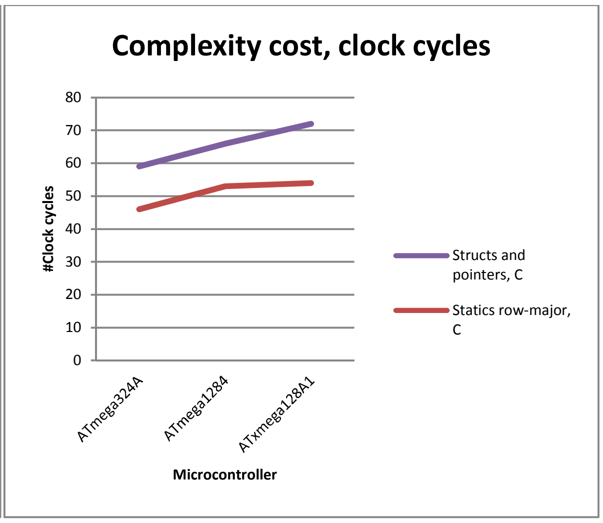


Figure 29: Complexity cost

6.4.5 Code size and clock cycle count, transmitting

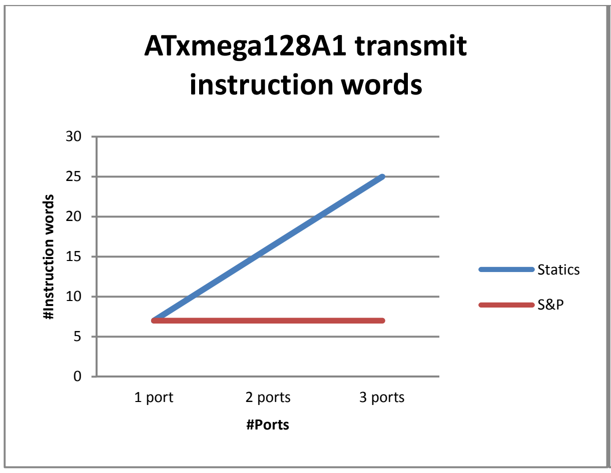
6.4.5.1 Statics row-major

Please see appendix A.8.4.1 for the test result data. The reason why the ATmegas have fewer clock cycle counts is more efficient LDS instructions and a quicker RET due to 16-bit PC. The code in *italic* is hand-written for a quick comparison.

6.4.5.2 Structs and pointers

Please see appendix A.8.4.2 for the test result data. This is the situation in which S&P is at its best, when recieving base address pointer and data it's as small and fast as one-port static implementation, for any number of ports.

This is also illustrated by the following two graphs:



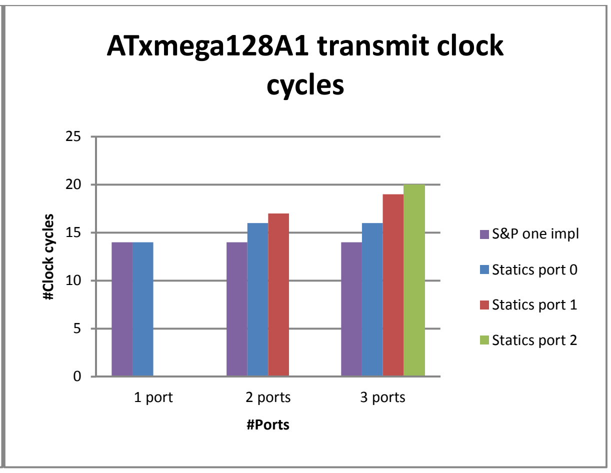


Figure 30: Transmit scaling

This very good result is maintained so long as your application can supply a base address pointer. As we saw in the BASCOM-AVR ATxmega implementation, it can also be useful to be able to pass a simple integer port number. This is what the current statics version wants for input, but the S&P approach requires a conversion. This could be done in several ways:

- As a switch statement that assigns the corresponding base address pointer (to the USART directly or to a struct variable with such a pointer).
- As a calculation (which is the BASCOM-AVR approach).
- As a lookup in a vector that holds the base address for each index.
- (As a hardware lookup table reached from a new assembly instruction as discussed in the summary.)

The first three add a one-time bit of code and a number of clock cycles, both adding an offset to the S&P numbers.

6.4.6 Why is the transmitting code so much neater than the ISR code?

Let's modify the USART_Transmit function so that instead of taking the data byte and a pointer to the USART it takes a pointer to the SerialPort struct that now also contains a field with the data byte:

Original version:

```
__attribute__ ((noinline)) void USART_Transmit( uint8_t data, USART_ATmega324_t *
usart ) {
   while ( !(usart->UCSRA & (1<<UDRE0)) );      // Wait for empty transmit buffer
   usart->UDR = data;                           // Start transmission
}
```

Structs and pointers
ATmega324A

Instr	Instr W	Instr C	Comment
MOVW R30,R22	1	1	
LDD R25,Z+0	1	1	
SBRS R25,5	1	2	1st try
RJMP PC-0x0003	1	2	
STD Z+6,R24	2	2	
RET	1	4	16-bit PC
	7	12	

Modified version:

```
__attribute__ ((noinline)) void USART_Transmit( UsartData_t * SerialPort ) {
   while ( !(SerialPort->Usart->UCSRA & (1<<UDRE0)) ); // Wait for empty tx buffer
   SerialPort->Usart->UDR = SerialPort->Data;          // Start transmission
}
```

Structs and pointers
ATmega324A

Instr	Instr W	Instr C	Comment	Description
MOVW R26,R24	1	1		X points to the struct SerialPort that starts with *Usart
LD R30,X+	1	1		Z points to the USART status register
LD R31,X	1	1		Z points to the USART status register
LDD R18,Z+0	1	1		R18 = content of the status register
SBRS R18,5	1	2	1st try	Check if data register is ready to receive
RJMP PC-0x0002	1	2		If not, repeat
MOVW R26,R24	1	1		X points to struct SerialPort
ADIW R26,0x14	1	2		Adjust so X points to SerialPort->Data
LD R24,X	1	1		R24 = SerialPort->Data
STD Z+6,R24	2	2		Write R24 to USART data register
RET	1	4	16-bit PC	
	12	18		

These very similar high-level functions become slightly different in assembly: 71.4% more code and 50% more clock cycles. The reason is that two pointers (X and Z) are now necessary.

The conclusion I draw from this is that with an object-inspired data model (S&P with encapsulated data) you run the risk of hidden complexity that makes the application bigger and slower. In my particular application I could probably avoid conversion from serial port #0-7 to a HW module base register pointer (as shown in the S&P code), but we have seen that if such a software-based conversion must be used, S&P will only become an option at higher module instance counts.

Please note that the differentiator isn't really the number of instances of a specific HW module but actually in how many different ways your application uses it. If your external peripheral driver operates on all instances in one go, S&P will never have a chance to scale into competitiveness. The serial ports, on the other hand, are used independently so S&P is an option.

6.4.7 How to choose between legacy static addressing and structs and pointers?

This is as far as I can go in my analysis of the "legacy" use of static addressing vs. the alternative use of S&P. There are several considerations I can immediately think of:

- Is the object-oriented data model (S&P) better when applications grow in size?
- Is S&P better at explaining the data model, due to its data grouping? Or is this rather a question of coming up with informative variable names and writing good documentation?
- Is static addressing better at showing what the program really does?
- At a certain point, it becomes inefficient to pass more parameters to a function (not to mention receiving the return value(s)). It depends on how many registers you can use without putting the parameters on the SW stack, so it also depends on how "big" your parameters are.

I thought about the possibility of combining the two models, so I changed the S&P ISR to be as similar to the statics version as possible:

```
ISR(USARTE0_RXC_vect)
{
   uint8_t data = SerialPort0.Usart->DATA;           // Read the received data
   uint8_t tmphead = \
      ( SerialPort0.RxHead + 1 ) & USART_RX_BUFFER_MASK; // Calculate buffer index
   SerialPort0.RxHead = tmphead;                     // Store new index
   SerialPort0.RxBuffer[tmphead] = data;             // Store received data in buffer
}
```

Unfortunately it -Os compiles to the same size as the original S&P version, so it's apparent that this could only be achieved by hardcoding the memory addresses for the instance fields. This would make the application very difficult to maintain.

Another way to use a hybrid data model might be to do HW module register operations with structs but otherwise use static addressing. However, I don't know if this would be possible and better and I don't have time to investigate it.

6.5 Protocol-bound ISR scaling (AS3j & AS3k)

As previously discussed, I chose row-major array addressing as it is more efficiently read (and written) with post-incrementing pointer, even though column-major consistently resulted in slightly smaller code.

The test results (appendix A.8.5) show that the optimization levels scale differently. I have mostly used -Os for comparisons, so I'll continue doing so. Please note that with -Os S&P is slightly better than statics, but with -O3 the situation is reversed.

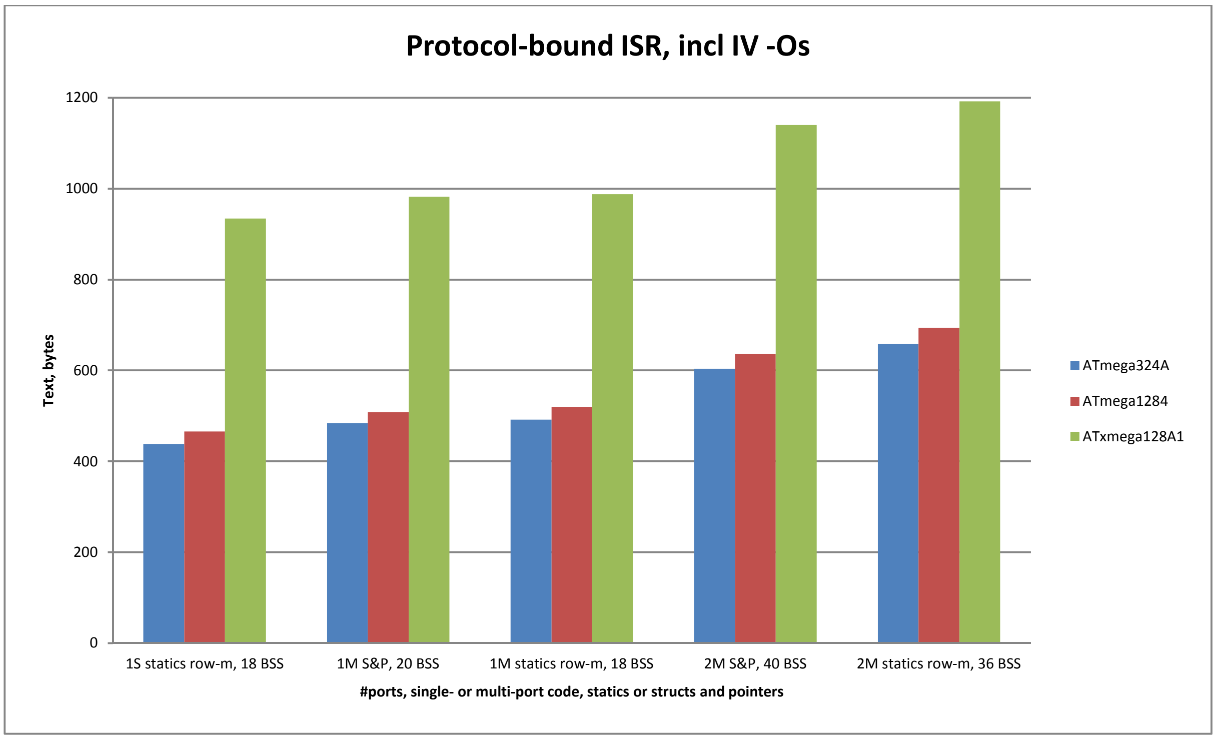


Figure 31: ISR scaling, incl IV

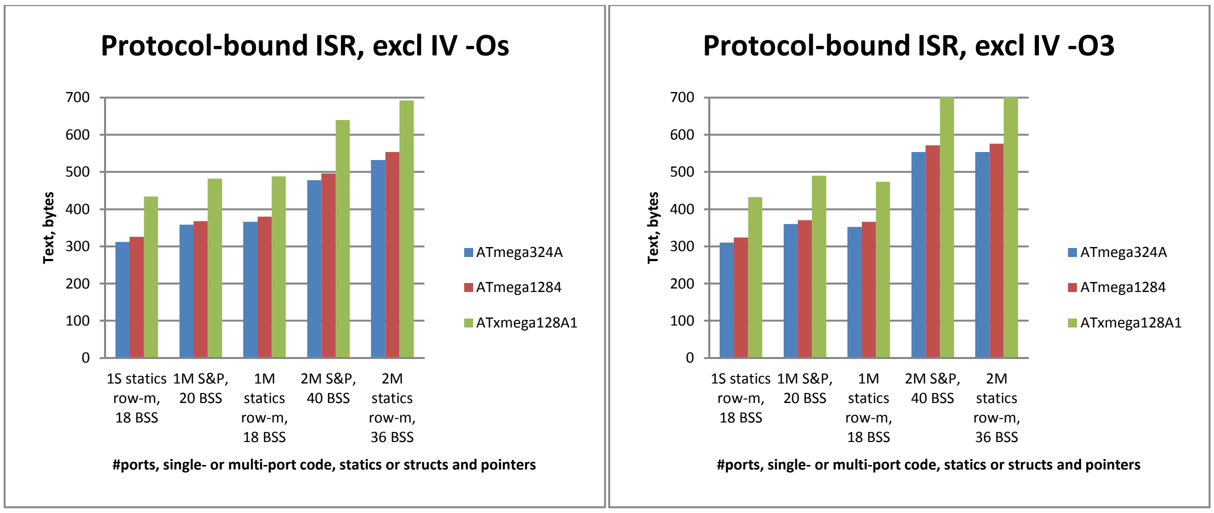


Figure 32: ISR scaling, excl IV

This test consists of one generic version that supports one to several ports and one version that only works with one port. This is to get a feeling for the cost of using a generic design. A very rough number is the average of (1M statics – 1S statics) / 1S statics for tests w/o IV but with initialization: 14.6%.

One thing stands out when looking at the protocol-bound ISR: The prologue (26 instructions) and epilogue (22 instructions) are big. What's more: the ISR code in appendix A.8.6 is included once per instance. With inline assembly, it should be possible to save some 90 bytes for ATxmega128A1. (Less for the other two, but still a significant reduction.) The "-mcall-prologues switch" can be used to generate one common set of this code. (65) [87]

Alternatively, it would be nice to be able to place it in the unused parts of the IV. An AVR Freaks user forum thread describes in detail two ways to do it: either provide a custom .vectors section or use the -nostartfiles flag and write all IV and default initialization yourself. (66) [88] Similar (and other) information can be found here: (67) [89] (68) [90] This post explains why it isn't (?) a standard feature; it's safer to let all the unused interrupts jump to an eternal loop. (69) [91]

[87] http://www.avrfreaks.net/index.php?name=PNphpBB2&file=viewtopic&t=97382&start=all&postdays=0&postorder=asc

[88] http://www.avrfreaks.net/index.php?name=PNphpBB2&file=viewtopic&p=1131093#1131093

[89] blog.schicks.net/wp-content/uploads/2009/09/bootloader_faq.pdf

[90] http://gcc.gnu.org/onlinedocs/gcc/Link-Options.html

[91] http://sourceware.org/ml/binutils/2013-02/msg00180.html

6.6 ATmega324A structs and pointers two-port USART ISR placed in IV

I was curious of how difficult and time-consuming it would be to place the USART ISR inside the ISR, so I had to test it. I made some minor changes to the interrupt handler C code and -Os compiled it to get the disassembly. (Appendix A.9.1.)

My real application uses two more interrupts, INT2 and PCINT2.18, so I had to reserve vectors 0x06 and 0x0C for them. There are three main differences compared to the default ISR prologue and epilogue generated by C code:

- It has one common epilogue and most of the prologue is also shared.
- As it doesn't call a function, the ISR only needs to PUSH&POP the used registers.
- It only cares about the RAMPD/X/Y/Z and EIND registers that are relevant. This makes no difference for the ATmega324A, some for ATmega1284, and some more for the ATxmega128A1 as shown earlier.

Reusing part of the disassembly code, I end up with the below myvects.S file. It took me some four to eight hours to complete this, most of which is research time. If I were to do it again, it might take me two hours. It uses global C variables, a global #define value, and hooks up to INT2 and PCINT2 ISR handlers written in C code. (Appendix A.9.2.)

It compiles to:
498 text, 0 data, 40 bss (-O1)
474 text, 0 data, 40 bss (-O2)
430 text, 0 data, 40 bss (-O3)
464 text, 0 data, 40 bss (-Os)

The same implementation in C code compiles to:
704 text, 0 data, 40 bss (-O1)
680 text, 0 data, 40 bss (-O2)
654 text, 0 data, 40 bss (-O3)
668 text, 0 data, 40 bss (-Os)

This is roughly 200 bytes more. The default IV ends with instruction word 0x3D (124 used bytes), which means that my ISR fits into the default IV plus five words. The RJUMPs past the interrupt vectors take four instructions, so let's say that the (two first of the) "main differences" listed above result in a 75 byte code size reduction. (You should be able to achieve something like this with the naked attribute and assembly placed outside of the IV.) It would have been more on the ATxmega128A1 due to the RAMPD/X/Y/Z and EIND registers. (I would disable interrupts when inside the bootloader and with only internal RAM there's not much need for >64k support in the ISR.)

Is it worth it? Well, once you have learned how to do it, it is pretty quick. I would recommend keeping the C code and doing future changes to the high-level code, with disassembly and manual modifications to the assembly code. It is of course more time-consuming to maintain than high-level code, but not very much for an experienced developer. Nevertheless, placing code in the IV space is something you normally only do when you've run out of program memory or when you need to squeeze in as much as possible in a bootloader. (Later I moved the ISR out of the IV and instead placed a number of short custom math (and other) routines there. They are easy to move if you need to start using an additional interrupt vector. The maintenance cost is practically zero.)

7 Compilation and discussion of the test results

7.1 Static vs. dynamic addressing (BASCOM-AVR) or S&P (AVR-GCC C)

The BASCOM-AVR implementation of the ATmega USART send commands is static, while the ATxmega design does base address calculation based on port number. We saw that the two approaches are almost identical in size at three ports, although the address calculation makes it much slower. At eight ports, dynamic addressing takes 36% more clock cycles compared to the static addressing average. At three ports it's 90% more and at one port it's 124% more clock cycles.

At one port, dynamic-addressed sending uses (46 - 26) / 26 = 76.9% more program code than static. At eight ports it saves (161 - 115) / 161 = 28.6%.

Much of the extra cost comes from the address calculation. In the AVR-GCC comparisons the pure S&P-based code is equal in size to static-addressed at two ports. At only one port they are equally fast, but S&P scales better (=flat), so from a performance perspective it is always a good choice. Please note that this test result requires that the base address is always known, i.e. an object-oriented way of always referring to (a struct variable with a pointer to) the USART base address. It also requires that only one variable is using pointers at each moment. With multiple simultaneous pointers the calculation and operations come at an additional cost.

If your program uses some kind of device numbering, e.g. port #0-7, and needs to translate this to either a (pointer to a) peripheral module base address or a (pointer to a) struct variable, you have a few alternatives that all add program code (mostly once) and clock cycles (for each call):

- As a switch statement or an "if elseif else" block that assigns the corresponding (base) address pointer. (The compiler might generate a lookup table.)
- As a calculation (which is the BASCOM-AVR approach).
- As a lookup in a vector that holds the base address for each index.

7.1.1 Suggestion for HW-based translation

To bring this down, I would like to suggest HW-based translation; a dedicated 256 lines * 16-bit wide SRAM or register lookup table that you call with a device type code and an instance number using a new set of assembly (machine) instructions:

rlut 0x12	Places the content of LUT address 0x12 in Z. It uses hardcoded device type and instance number.
rluu 0x3	Places the content of LUT address 0x3[r24 content] in Z. It uses hardcoded device type. It requires that the instance number is already in r24.
rluv	Places the content of LUT address 0x[r25 content][r24 content] in Z. It requires that the device type and instance number are already in r25 and r24.
wlut 0x07	Writes the value in Z to LUT address 0x07. It uses hardcoded device type and instance number.
wluu 0x0	Writes the value in Z to LUT address 0x0[r24 content]. It uses hardcoded device type. It requires that the instance number is already in r24.
wluv	Writes the value in Z to LUT address 0x[r25 content][r24 content]. It requires that the device type and instance number are already in r25 and r24.

Figure 33: Proposed instructions for look-up table operations

I realize that it's highly unlikely that Atmel would actually implement this, but I still think there is evidence that HW-based translation from instance number to base address is worth investigating further. Even though I have only analyzed the serial ports, I think that the results are generally valid. What I want to achieve is a uniform programming style, based only on the S&P approach, but without the penalties we have seen (especially in the BASCOM-AVR ATxmega implementation). It would make less of a difference in a pure struct variable design (where the base address is a pointer field), but the need to translate should be common enough.

Example: A call to a device driver:
Place the instance number in r24 unless it's already there (as a regular first byte parameter)
rcall your_function

Inside the function:
If necessary, place r31:r30 on the stack. (It would be done anyway, here or before the rcall.)
rluu (hardcoded device type)
Proceed with the driver code.

This would actually result in more than a reduction in the translation itself. Today, the AVR-GCC function call contains a pointer variable to the device instance base address, e.g. in r25:r24. Before the call, these two registers must be be written to (possibly after being saved on the stack). rluu only needs one register. Inside the function, movw r31:r30, r25:r24 is done. rluu eliminates the need for this. This is a minimum save of one instruction for each function that operates on a base address pointer and at least one instruction for each time this type of function is called, probably more with the stack operations.

The struct variable that today has a 2-byte pointer field for the base address would only need one byte for the instance number, a save of 1 byte of regular SRAM for each instance. This information might already be in a struct field, in which case 2 bytes would be saved.

This concept is not limited to drivers, but all types of struct variables or "objects". rluv or rluu could be used for iteration on objects of the same type, without an overlying vector or linked list that holds their base address. In this sense, the LUT would implicitly take the place of the data structure itself. It would certainly be a lot more efficient (and quicker) to increment one or two bytes for the LUT addressing than do vector indexing or linked list positioning with LDS from SRAM into r25 and r24.

It has another kind of potential advantage: it only requires that the peripheral register layout is internally uniform (for the registers that are used by the driver), not that the register groups are placed at an equal distance from each other. Compared to switch/if_elseif_else or vector-based translation, it wouldn't make a difference, but it would compared to calculation-based translation. I haven't analyzed this, but I think that it is worth looking at. It might make it possible to use generic S&P drivers also for ATmega. At least this is the case for the ATmegaXX4 USARTs.

There is of course a cost in terms of the dedicated LUT SRAM/registers and logic, and you have to make at least one initial write to it, which takes a minimum of three instructions. Alternatively, this functionality could be silicon-based, with only read operations. It would limit the functionality to peripheral IO registers, but at no initialization cost. As a third option, it could be an SRAM/register area that initializes to the silicon-based values but can be over-written.

Of course it requires some further analysis, but it would be interesting to test. Maybe I'll try it on a soft-core logic device.

Please see the XMEGA Custom Logic (XCL) for quite a different type of special functionality. (70) [92]

7.1.2 More simultaneous pointers with displacement

I have only touched on this subject, but we have seen that S&P performs best when it only needs as many simultaneous pointers as it can access without moving register content around. As Y is often used as a SW stack pointer, only Z remains with displacement functionality. X can be used, but requires addition or subtraction (ADIW, SUBI, SBCI) for each new address (unless post-increment or pre-decrement can be used).

It's clear that the struct variable-based programming model often wants more than 2-3 pointers. I suppose that there are HW or instruction set size limitations, but it would be very interesting to see how much could be gained by additional full-featured pointers. This need is also stated clearly in "AVR-GCC-Codeoptimierung" (71) [93] and implicitly in "The AVR Microcontroller and C Compiler Co-Design" (20).

In comparison, ARM has full pointer support on 13 general-purpose registers. (72) [94] This should enable quite different possibilities for the struct- or object-based programming model. The fact that AVR 8-bit microcontrollers only have three memory pointers is probably its greatest weakness. If ever Atmel releases an ATxmega2, it should definitely incorporate more pointers.

7.2 Hardware-related complexity costs

Experienced designers and developers are generally aware that there are hardware complexity cost factors, although I believe that their actual impact in terms of code size and clock cycles is less well-known. As we've seen from the disassemblies they can make a significant contribution, especially in certain cases.

7.2.1 Program memory sizes

If your application uses "many" interrupts, a lot of ATxmega instruction words will do RAMP stack operations (unless you tell the AVR GCC compiler to generate one common set of interrupt prologue and epilogue that does a lot of stack operations or use "ISR(vector_name, ISR_NAKED)" to prevent it from generating prologue and epilogue). BASCOM-AVR has a corresponding "nosave" attribute. The ISR disassemblies showed us that this effect can be quite big (37.8% more code with S&P and 34.5% with statics):

Structs and pointers ISR		Statics ISR	
ATxmega128A1	51W, 72C	ATxmega128A1	39W, 54C
ATmega1284	41W, 68C	ATmega1284	33W, 53C
ATmega324A	37W, 61C	ATmega324A	29W, 46C

[92] http://www.atmel.se/Images/Atmel-42083-XMEGA-E-Using-the-XCL-Module_Application-Note_AT01084.pdf
[93] http://www.mikrocontroller.net/articles/AVR-GCC-Codeoptimierung
[94] http://www.eng.auburn.edu/~nelson/courses/elec5260_6260/ARM_AssyLang.pdf

However, I don’t think that the typical application (except for the ISRs) suffers that much from this as I assume that the compiler only changes these registers when it has to.

A short summary:

- RAMPD together with constant K: necessary for addressing data memory above 64kB. This is (only?) relevant when using ATxmega EBI (External Bus Interface) with external RAM.
- RAMPX/Y/Z together with X/Y/Z pointer: used for indirect addressing of the data memory above 64kB. RAMPZ:Z is also used when reading and writing program memory above 64k words (128kB), typically by bootloader code.
- EIND together with Z pointer: used for certain jumps and calls on devices with more than 64k words (128kB) program memory. 128kB devices with bootloader space outside of the regular memory called “application section” and “application table section” need this, e.g. ATxmega128A1.

ATxmega64A1 has all of these except EIND while ATxmega128A1 has all.

ATmega1284 has RAMPZ for ELPM and SPM program memory operations, but the other ATmegaXX4 don’t have any.

7.2.2 Feature-fulness and architecture

7.2.2.1 IV, Interrupt vector

More peripheral types and instances means a bigger IV:

- ATmega324A: 124B
- ATmega1284: 140B
- ATxmega128A1: 500B

I have shown various ways of using all or the end of the IV space for custom code, but this is something you typically don’t do unless you have exceeded the program memory size. The simple two-port S&P ISR code without error handling for ATmega324A saved roughly 200 bytes, of which about 75 came from avoiding duplication of ISR prologue and epilogue.

As a result of a forum question by me, BASCOM-AVR 2.0.7.7 and later has an unsupported switch, $reduceivr, that truncates the unused end of the IV. This means that your application code will start inside the IV area, giving you some extra space.

7.2.2.2 Configuration code

More configuration options means more configuration code:

- ATmegaxx4: system clock setup is mainly done via flash fuse bits. With an external crystal, iIt’s only the prescaler that needs to be set with two two-word instructions. Power reduction is one two-word instruction for all ATmegaXX4 devices except ATmega1284 that (might) write to two registers.
- ATxmega128A1: A system clock setup sequence that takes about 26 instruction words for external crystal and six power reduction registers that require at least eleven one-word instructions to write to. ATxmega also requires that you set the USART transmitter pin to

output, while ATmega does this automatically. (This shouldn't make an actual difference in a real application, as you typically initialize each port in one go.)

7.2.3 Instruction set and CPU register file

I don't have time to go into these areas, but I should at least mention the fact that that differences in memory layout (e.g. banked or non-banked), instruction set, and the number and type of CPU registers have an impact. E.g. RISC programs have been found to be 30% bigger than CISC. (73)

It's also worth noticing that *"Our study suggests that at performance levels in the range of [ARM Cortex] A8 and higher, RISC/CISC is irrelevant for performance, power, and energy."* (74)

7.2.4 Is HW complexity costs a reason to use soft-core logic devices instead?

Is there a complexity-related boundary for general-purpose microcontrollers? I won't try to answer this question here, but rather just put it up for discussion.

7.3 Software-related costs

7.3.1 Optimizing or non-optimizing compiler

7.3.1.1 AVR GCC

The AVR GCC (Gnu Compiler Collection) toolchain is the heart of Atmel Studio 6. It has a powerful optimizing compiler with four usable optimization levels. As we've seen, you can't be sure to get the smallest code with the -Os level (size-optimization) nor the fastest one with -O3 (speed-optimization). If you are seriously interested in optimizing your code, you are adviced to disassemble and look at what it actually does, typically doing iterative modifications to your high-level code.

Nevertheless, I am very happy with its performance. Sometimes you can get a big improvement by using hand-written assembly (e.g. ISR), but on other occasions the two seem to be on a par with each other. Especially in very complex situations, I believe that the optimizing compiler wins so long as you write code that is easy to compile well. This is easier said than done in a complex situation.

7.3.1.2 BASCOM-AVR

BASCOM-AVR is a non-optimizing compiler that generates a stitch-work of pre-defined (hand-optimized) assembly code (i.e. built-in commands) and interconnecting pieces of compiled VB. It places all local variables and call parameters on a SW frame stack. This can save RAM compared to using global or static local variables and in some environments enables reentrancy and recursion. (General information about the use of stacks can be found here: (75) [95])

An optimizing compiler keeps as many variables as possible in the working registers, which can eliminate or reduce the need to work with the frame stack, with the advantage of smaller and faster code.

BASCOM-AVR's advantage is that it is completely predictable. You can look at the assembly library code and probably also modify it (or at least reuse it). It's also a trade-off against IDE and command development time. They put great effort into making it easy for the developer to implement the most common functionality, quickly.

[95] http://www.ece.cmu.edu/~koopman/stack_computers/sec1_4.html

7.3.1.3 Comparison

BASCOM-AVR's ATxmega sole design principle of doing address calculation makes it impossible for a comparison of compiler efficiency. (After writing this, BASCOM's owner Mark Alberts has read and commented on my thesis. He will include a developer option for ATxmega, which should put it back in the game.)

Luckily, BASCOM-AVR's ATmega implementation is static, so it's quite ok for such a benchmark:

ATmega324A	**All code**			**Excl. 124 B IV**			**Excl. 124 B IV and unused B-A code**			**Excl. 124 B IV, unused B-A code, and init**		
Compiler	**BA6b**	**BA7**	**BA8a**	**BA6b**	**BA7**	**BA8a**	**BA6b**	**BA7**	**BA8a**	**BA6b**	**BA7**	**BA8a**
BASCOM-AVR	724	688	484	600	564	360	582	546	342	532	496	292
AVR GCC -O3	470	470	412	346	346	288	346	346	288	296	296	238
AVR GCC -Os	490	490	412	366	366	288	366	366	288	316	316	238
B-A bigger by	51%	43%	17%	69%	58%	25%	63%	53%	19%	74%	62%	23%

The BASCOM-AVR version was developed first and then translated into C. BA6b and BA7 has the same C counterpart as the latter's send routine is already "custom-like" in BA6b.

- BA6b: VB high-level only version
- BA7: Custom inline assembly send routine
- BA8a: Custom inline assembly send routine and protocol-bound ISR

Please note that:

- There are slight differences in the ISR functionality, although both use a circular RX buffer.
- The BASCOM-AVR BA6b version has a more multi-purpose send functionality.

In other words, you must take the above figures with a grain of salt. Nevertheless, I draw two conclusions from this:

- BASCOM-AVR's design with command stiching generates some 70% bigger code in this test, when the IV is excluded.
- It is possible (and wise?) to reduce the difference by using inline assembly in select places.

7.3.2 Generic software library or built-in commands

The test results above (mostly BA6b and its C counterpart) come from using BASCOM-AVR built-in commands and Atmel application note code. For BASCOM-AVR it means a certain overhead when entering and exiting the commands, but it is still much more efficient than writing VB-only implementations of the commands. In other words: BASCOM-AVR benefits greatly from its commands.

When it comes to Atmel library code there are two choices:

- Application notes that present the design, its background and considerations, and give you a simple demo program with the driver code. I found these to be informative, to the point, and easy to use together with the datasheets.
- ASF, Atmel Software Foundation, is a repository for drivers and demo projects. I found this to be buggy, tedious, fragmented and bloated beyond comprehension. In the end, I have come to use the ASF Wizard to include ASF modules just to get convenient access to the driver header files. This also enables me to look at the ASF code and in some cases copy from it. Generally, I prefer reading application notes and datasheets.

  This means that I opt out of the claimed easy transition to ARM - it's not worth the ASF penalty. If and when that day comes, I will write custom drivers for ARM.

What I would like from Atmel in this area is a driver code generator with a simple GUI.

7.3.2.1 (Lack of) exclusion of unused code

BASCOM-AVR suffers a little from unused code, but the cases I have come across are mostly basic supporting functionality that you are likely to need at some later point in your actual application. Due to its use of built-in commands, it avoids the problem of massive inclusion of unused driver code. This means that compared to ASF, BASCOM-AVR in many cases actually produces a lot less code.

ASF, on the other hand, is like getting fleas: You don't just get one. You get all their aunts and cousins too. A simple ATxmega clock setup, USART driver/service configuration, and dummy sending of one byte (without the ISR) lands you at about 1900 bytes. We saw that a minimum hand-written C implementation takes -Os 822 bytes, of which the interrupt vector table is 500 bytes and the default initialization and clock management account for 122 bytes. Just including all the driver modules my real application needs would leave me with -Os 8334 or -O3 9954 bytes. This is just silly. I wonder if this is part of the reason why Atmel has so limited ASF support for ATmega; It simply wouldn't work - the program memory would already be filled with dead code from AZF, the Atmel Zombie Foundation.

The good news is that going from USART HW-oriented driver to cross-platform service only sets you back 82 bytes on top of the driver's 1840 bytes. It doesn't seem to be very expensive to add an abstraction layer to your custom driver code, although this is really only an assumption.

7.3.2.2 General-purpose driver that does something similar to what you need

The BASCOM-AVR send routines support three different types of transfer:

- Sending a constant
- Sending a 1-byte variable
- Sending an n-byte string variable

The Atmel application note code is a simple “send one character” routine – just what I wanted. To send a string you must add custom application code, which makes the BASCOM-AVR approach less expensive in comparison.

7.3.3 High-level code vs. assembly

Due to the lack of optimization in BASCOM-AVR, I would say that it benefits more from inline assembly and custom assembly subroutines and functions. While the built-in commands greatly reduce development time, sometimes a custom implementation is prefereable. One example is the protocol-bound ISR handler. A 44% reduction in code size was achieved by replacing the generic command-based ISR and VB application code with protocol-bound inline assembly.

With AVR GCC’s optimizing compiler you can manage with high-level code in most situations. Inline assembly or assembly functions are necessary when you want to make sure that operations are carried out in a certain order and proximity. We also saw that assembly makes it possible to use all of the IV, with a minimal increase in maintenance “cost” (which you’d normally only do for a bootloader).

I found that using Assembly doesn't take much time, if you start with high-level code and modify the disassembly. Much of the time spent is on initial learning, a one-time cost.

7.4 Programmer skills

The incremental improvements to the BASCOM-AVR ATmega324A test code showed that I could reduce the VB-only code (excl. IV) from 882 to 600 bytes (32%). I found a bug (in the routine for copying data from the circular buffer to the work vector) and understood some consequences of various programming styles and ways to encase commands. The most striking was how expensive it is to use locals and call parameters in a stack-only compiler philosophy. If you want to minimize your BASCOM-AVR code, you should use global variables with gosub-specific alisases. If you do, an absolute requirement is that you make 100% sure that your alias-globals really only have local scopes. You must also be certain that your code doesn’t have to be reentrant, e.g. from an ISR, which is the only real multi-process situation you can have with AVRs.

It is obvious that really knowing what you are doing can have a great impact on code size. Being willing to question and circumvent conventions helps.

7.5 ATmega or ATxmega or both?

7.5.1 Hardware aspects

So, how do you choose between ATmega and ATxmega? For my application needs, ATxmegaA1U has four main advantages: HW encryption engine (for my bootloader), USB interface, DAC, and a lower price for more or similar functionality. It has a higher maximum clock frequency, DMA, event system, bootloader in addition to the 64 or 128kB program memory, can use external RAM, and possibly has lower power consumption (and more), although I currently don't need this.

On 2014-02-05, www.farnell.se lists the following prices for similarly sized or featured AVRs:

- ATMEGA324A-AU: SEK 41.92
- ATMEGA1284-AU: SEK 58.46
- ATMEGA128A-AU: SEK 92.21
- ATMEGA1280-16AU: SEK 145.32
- ATMEGA1281-16AU: SEK 123.39
- ATXMEGA64A1-AU: SEK 75.17
- ATXMEGA128A1U-AU: SEK 51.12

We have seen that the additional and improved functionality has a cost in terms of:

- Bigger IV, interrupt vector table.
- More configuration code and possibly more transaction-related code (although this is an assumption).
- More memory-related additional operations needed (RAMPD/X/Y/Z and EIND). Especially for ISRs with individual prologue and epilogue, this makes a code size difference.
- A greater fraction of registers outside of the IO memory area. This couldn't be seen in my test programs as the ATmegaXX4 USARTs are outside of the IN/OUT instruction area.
  - In ATmega1284, 32 of the 100 peripheral registers can be accessed via IN/OUT, plus the digital IO pin registers.
  - In ATxmegaA1U, 4 of the 61 peripheral register groups can be accessed via IN/OUT, excluding the digital IO pin registers. 4 of the 11 IO ports can be mapped to virtual ports that are covered by IN/OUT.

ATxmega has a more efficient implementation of some instructions, which means that they take fewer clock cycles. The opposite case also exists.

All in all, ATxmegaA1U is slower at the same clock frequency and requires more code to do "the same thing", but its boot section is in addition to the specified program memory. It's cheaper than its ATmega counterparts and can be clocked at a 60% higher frequency. A1(U) has more internal peripheral types and instances, which means that you might need less external peripheral circuitry. It has 16 General-Purpose IO registers that can be used for global variables with single-instruction bit operations, whereas ATmegaXX4 has only three, of which one is bit-operable.

With the ATxmega product offering ranging from 8kB to 384kB and 32 to 100 pins, I think it's clear that Atmel is slowly phasing out the ATmega line. It must be very expensive to maintain so many different devices.

The complexity costs can be seen in the following graphs:

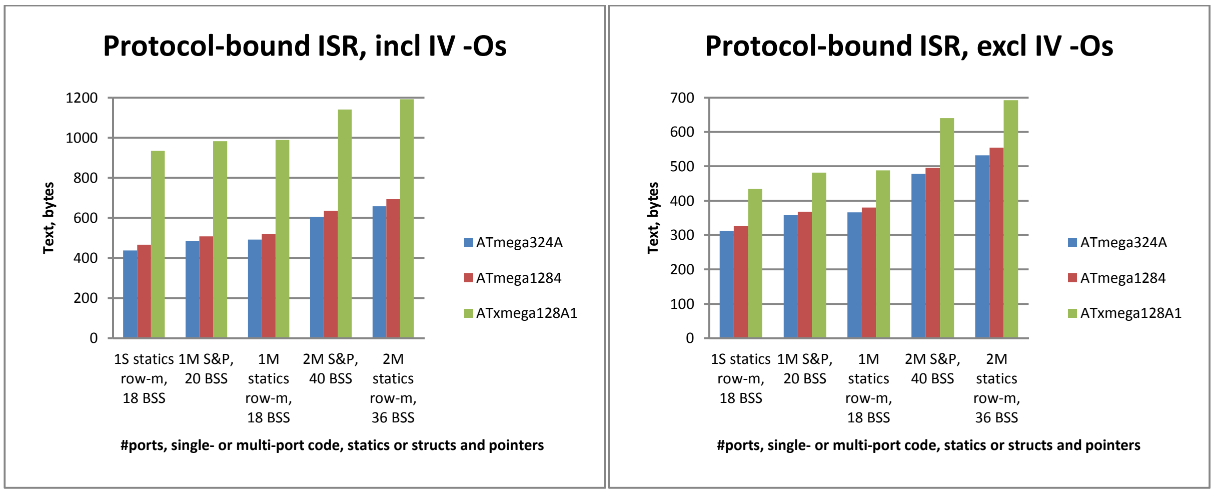


Figure 34: ISR scaling demonstrating complexity costs

7.5.2 IDE aspects on architecture choice

7.5.2.1 *Atmel Studio 6*

One of my main concerns about Atmel Studio 6 was ASF's weak support for ATmega. I was also wondering if using library drivers with the new ATxmega struct-based addressing would introduce a programming style that isn't compatible to ATmega, in effect forcing me to use two different programming models for the same functionality. This didn't sound very desirable.

However, with ASF being what it is, I am left with custom code based on application notes, datasheets, occasional ASF copy&paste, and user forum posts and projects. It takes more time, but it means that I could develop my own programming model. Based on my investigation results, it would be a mixed model.

My different test program versions have shown me that the S&P approach is size-wise on a par with static addressing at two ports if you use a pure S&P design with only one pointer. If you need more pointers or have to translate a port number to a base, the balance point shifts up. I suspect that break-even is case-specific but the general principle should hold. For single-instance drivers it generally seems better to use static addressing.

7.5.2.2 *BASCOM-AVR*

With BASCOM-AVR you really have one common IDE for ATmega and ATxmega. The commands are generally the same or similar and it is easy to develop for more than one device in the same 'project'. Its users don't have to worry about which AVR architecture to choose.

The downside is the current ATxmega address calculation design that is very big and slow. I would like the possibility to choose addressing mode in the configuration command (which will be possible in a later version due to the results in this thesis).

7.5.3 HW (and SW) maturity

One very important factor when deciding on a microcontroller (architecture) is whether the teething problems are over and it has gained a large user base. At least certain ATxmega devices have had serious problems. (76) [96] I have 10 pieces of ATxmega128A1 that I can't use because single-ended ADC seems to have been broken in all HW revisions until the silent replacement by the A1U series. I don't know what the status is right now, but my general impression is that the ATmega series is safer. This said, I have reported to Atmel that the ATmegaXX4 datasheet was very unclear about the fact that only ATmega1284 has timer3. This cost me a PCB redesign...

This is an important topic. HW that doesn't work as specified or is announced a long time before it's generally available, SW that aggravates the user base by simply not working properly, and faulty documentation is a risky business. At some point, customers might decide to look for another manufacturer that goes the extra mile to ensure that its customers don't get nasty surprises. This is even more important on the ARM market where it's so much easier to change suppliers.

However, launching two IDEs, a software library, ATxmega, and ARM in such a short period is a gargantuan task, so maybe they have managed quite well under the circumstances. I can only hope that their offering is stabilizing now.

7.6 IDE comparison

Both ASF and the BASCOM-AVR documentation could be improved, especially the first. The Atmel application notes I have used have been relevant and informative.

7.6.1 Atmel Studio 6 with ASF

Atmel Studio 6 is the company's second IDE based on MS Visual Studio. The predecessor, AVR Studio 5, had a very short life and does not seem to have had many admirers. (77) [97] (78) [98] I only used their proprietary AVR Studio 4 for programming the .hex files from BASCOM-AVR, so my first encounter with the actual IDE was last spring, on AS 6 for the Cortex M4 ARM. At that point we suffered from frequent crashes when debugging and had strong opinions on ASF.

Half a year later I am doing this AVR thesis tests on a later version and I am very happy with the studio itself, even though it takes a long time to start up. Once it's up and running it's responsive and intuitive. The integration with ASF works well, except for the weird fact that it has hung my Firefox browser on several occasions. This shouldn't be a problem as I would almost only use ASF for convenient inclusion of the proper header files. I have a very good impression of the application notes that I have come across. I also like both the ATmega single datasheet and the ATxmega dual datasheets, even though I have found and heard of several errors. (76) [99]

I haven't investigated what I would need to do to enable multi-device development in an Atmel Studio 6 project, but I assume that I would have to take care of all file inclusions. I guess that this would break the ASF integration, but all this is speculation.

[96] http://www.avrfreaks.net/index.php?name=PNphpBB2&file=printview&t=103269
[97] http://www.kanda.com/blog/microcontrollers/avr-microcontrollers/avrstudio-explored/
[98] http://www.avrfreaks.net/index.php?name=PNphpBB2&file=printview&t=103949
[99] http://www.avrfreaks.net/index.php?name=PNphpBB2&file=printview&t=103269

So long as you use custom drivers, Atmel Studio 6 with AVR GCC produces very efficient code. The toolchain has very many features that I have only just started to learn and master. All in all, it's a powerful developer tool that I like very much, so long as I can avoid ASF.

I would like for Atmel to develop a simple GUI tool for driver code generation. After all, this is what BASCOM-AVR does when it converts its commands to device-specific assembly blocks.

7.6.2 BASCOM-AVR

The BASCOM-AVR users haven't had to suffer major IDE changes with all the unavoidable initial problems. While I would say that Atmel Studio 6 is an excellent development environment with an auxiliary code base (application notes and ASF) that you can copy from, BASCOM-AVR is rather a productivity-enhancement tool (plenty of standardized built-in commands) with a development environment. If you want to minimize development time, BASCOM-AVR is a very good choice.

The fact that the company behind BASCOM-AVR is fairly small means that you can get in direct contact with the IDE developers and decision makers. I am the initiator of two pieces of BASCOM-AVR functionality and two implementation changes:

- The dword (unsigned 32-bit int) data type was necessary for the SD card and FAT16/32 library I wrote a few years ago, so they agreed to implement it. (79) [100] (80) [101]
- The unsupported $reduceivr switch that makes your application code start directly after the last used interrupt vector, was implemented after a forum post by me early in the thesis work. (45) [102]
- The change in v2.0.7.7 to static configuration of the USARTs is the direct result of my e-mail correspondence with BASCOM-AVR's owner.
- In the feedback on the nearly completed thesis from BASCOM-AVR's owner, Mark Alberts says that he will make it possible for the developer to choose static or dynamic addressing for ATxmega.

Being able to access and influence the development team is invaluable. I can't sufficiently stress the importance of this.

Multi-device development is easy with BASCOM-AVR. You just need to change a pre-compiler definition and uncomment the inclusion of your device and that's it.

I would like a BASCOM-AVR optimizer, pointer support, and debugging. (It has a built-in simulator and produces output files that can be debugged in Atmel Studio 6.)

7.6.3 User forums

Both IDE alternatives have very active, helpful, and competent user forums. They are a valuable source of code samples and information, lots of information.

[100] http://www.mcselec.com/index.php?option=com_content&task=view&id=291&Itemid=57
[101] http://www.mcselec.com/index2.php?option=com_forum&Itemid=59&page=viewforum&f=18
[102] http://www.mcselec.com/index2.php?option=com_forum&Itemid=59&page=viewtopic&t=11718

7.7 Suggestions for future work

- Do a paper-based analysis of the suggested HW-based peripheral instance base address look-up table. If it still seems like a good idea, implement it on a softcore microcontroller and test run it on a few real-world programs.
- Analyze the difference between using separate shared global arrays and vars compared to global struct variables or even extend it to a comparison between procedural and object-oriented programming
- We have clearly seen the complexity costs that more advanced devices with more instances incur. Analyze the trade-offs of this kind, on paper and on a real softcore microcontroller.
- Analyze the trade-off between writing code that is portable from e.g. an 8-bit microcontroller to ARM, using both pre-existing driver library (not ASF…) and handwritten code based on application notes and datasheets. How much does portability "cost" with clever driver design and what are the trade-offs?

8 Summary

8.1 IDE choice

The BASCOM-AVR IDE is found to be a fine productivity-enhancement tool for quick development (due to its well-documented built-in commands supporting most common peripherals) but its lack of an optimizing compiler makes its compiled test code size about 70% bigger than Atmel Studio 6's. For ATxmega, it currently goes all-in for the dynamic addressing model with calculation of base address from instance number, which makes it produce very inefficient code for this architecture for the most common use cases. After reading my thesis, the company will make it possible for the developer to choose ATmega-type static or dynamic addressing for ATxmega, effectively putting it back in the race. I also want to emphasize the value of being able to get in contact with "the right", people at the company, which I know is easy at BASCOM-AVR's support forum or by e-mail.

The Atmel Studio 6.1 IDE itself also wins my approval, being an enjoyable programming environment, having a graphic debugger and a powerful optimizing compiler (avr-gcc). Very disappointingly, the driver library Atmel Software Foundation (ASF) inserts huge amounts of unused ("dead") code, which together with annoying bugs, incomplete documentation, and weak support for ATmega makes it unusable for the author's purposes. It doesn't make it less necessary to read datasheets and application notes, so there is no productivity enhancement. With no real upgrade/downgrade path between ATmega and ATxmega and no desire to use it for ATxmega anyway, there is also no portability to (Atmel) ARM gained, which after all must have been Atmel's intent in the first place.

In the end I choose Atmel Studio 6 for my future 8-bit class applications, using datasheets and (the very good) application notes for custom drivers for a platform I by now am quite familiar with. For future ARM development I will have to look for either a competitively priced multi-brand IDE with a good driver library or one from another manufacturer who makes a good one themselves.

8.2 HW selection

ATxmega is a more powerful architecture, competitively priced compared to similarly sized ATmega devices, and by now hopefully free from its teething problems. The additional and more powerful peripherals together with its support for external RAM come at a considerable complexity cost in terms of code size and total clock cycle count. I decide to consider both types for my designs (based on the application's need for features and pin count), but choose ATmega when it's a tie, rather than switching completely to ATxmega.

8.3 The programming test results and the conclusions I draw from them

I was surprised to see how big the complexity costs associated with bigger and more powerful devices are. This is mostly seen in the IV table space (many and advanced peripherals), ISR prologue and epilogue (memory size above 64k words), and program code size (configuration and initialization of peripherals).

The programming investigations show that the legacy static (absolute) peripheral addressing model is better at one “instance of use” of the peripheral type than the S&P model introduced by the new uniform memory layout in ATxmega. At two ports it’s a draw and above that S&P is preferable.

For efficient programming, you should therefore use a mixed addressing model and the driver library concept should support it. To enable abstraction for higher-layer application code, the driver library should specify an “interface” that the (custom) driver must implement. A default implementation (or alternative ones) is welcome, but most importantly the driver library should contain a GUI-based tool in which the developer enters device brand, type, model, parameters, and optimization type. The output should be C and assembly code with an identification string that could be used to retrieve the same GUI settings for another brand or type. In fact, BASCOM-AVR’s configuration and use of its built-in commands do a similar thing behind the scene.

This paper also suggests a new type of HW-based look-up table from peripheral module type and instance to its base address. It should be a 256-line 16-bit wide RAM memory area initializing to the peripheral map but be possible to overwrite with pointers to generic application-specific data structures (global “objects”). It comes with new machine instructions for reading and writing. It would save both program code and clock cycles, effectively making S&P as good as static addressing for all numbers of instances. It would also enable a uniform programming style, making S&P possible also for ATmega and be a differentiator among competing ARM manufacturers.

8.4 On efficient programming

There is clearly a big potential in developing skills and methods for more efficient programming. Exactly how big wasn’t really possible to ascertain, due to BASCOM-AVR not using an optimizing compiler and ASF being so poor.

- The BASCOM-AVR test program could be reduced by 34% using only high-level language code and another 44% by replacing the generic receiver with handwritten protocol-bound assembly.
- The BASCOM-AVR test program was about 70% bigger than the compiled code from the avr-gcc toolchain.
- For the incomplete ASF serial port test program there was a 6:1 ((1840-622)/(822-622)) difference in compiled code size when comparing ASF to the slightly improved application note code.

All 8-bit AVRs suffer from the fact that they only have three memory pointers, which at a certain point forces the compiler to do expensive register data movements. This has a direct effect on the programming model I recommend for AVR: avoid occupying several concurrent memory pointers, if possible by keeping variables in registers or else (when there is no available memory pointer) by using standalone global variables. It thereby also affects the informed choice of using a data model with separate global arrays and variables or one with struct-based “objects”.

I (later) saw that when the code grows beyond the very simplest of test programs, the quickly increasing complexity (also witnessed in the simple thesis tests) makes it very difficult to foresee the consequences, e.g. when choosing between using separate varables and two-dimensional arrays for global data or grouping it in object-like struct variables. This is a fundamental design choice, but I believe that people today generally make it based on concept preference rather than performance facts. After seeing the complexity I find it easier to “forgive” such a decision, but I still think that it is highly relevant to analyze this on an academic level. With the slower HW performance increase we now have and the diminishing return on parallelism, this should be an area of big commercial interest, perhaps not by the companies selling server or desktop HW or operating systems but by the immense user base.

How could you make the study and gradual build-up of systematic knowledge of such a complex area into an academic discipline? Well, I am rather surprised that we (as it seems) have not yet made a serious attempt (at least in recent years), given the importance of computers in today’s society. I think that it requires a large number of very small test implementations of great many isolated areas, e.g. between procedural programming’s data structures and the object-oriented counterparts, scaling up in data size or comparing different processor architectures, or different high-level languages on their respective compilers. Perhaps a linguistic approach with grammars could be used to structure the test cases and organize the results in this multitude of valid and invalid combinations. There needs to be a system (approach / set of rules / calculus) for calculating the cost of a certain “construct”, e.g. a method or a set of data structures. One end goal should probably be to integrate it into the IDE, but most importantly it should be common knowledge at the design stage. However, this is all very early thinking.

Generally, you shouldn’t need to adapt your core programming style to the IDE when using an IDE equipped with an optimizing compiler, but as BASCOM-AVR doesn’t have one, you should in this case avoid local variables and parameters in subroutines and functions. Whether you should use the alternative struct-based addressing model for ATxmega depends on the use case. For few instances the legacy static (absolute) addressing is smaller and at least as fast.

How much inline assembly should you use? In retrospect, I find this question badly put. The fact that we are brought up writing practically only high-level code has made us fear assembly for no good reason. As I learned how to use it, I found it enjoyable and the natural choice in those situations when you want full control (e.g. configuration or timing-related situations) or do data manipulations that are easier to formulate on the instruction level (e.g. math routines). It is also necessary to understand assembly when analyzing how the high-level code gets compiled to machine code (represented by assembly). A practical approach is to write high-level code, compile and then modify this code for your purposes, and use it as inline assembly or assembly functions. Quick and efficient.

I think it is safe to say that I have got as much as possible out of the AVR HW using avr-gcc. I have continuously monitored and adapted to the disassembled code, replaced part of the C code with hand-written assembly, filled up the empty IV table space with custom assembly code, and shown the effect of the two different peripheral module addressing models.

Have I found a way to keep development time to a minimum, while also producing highly efficient code? Well, not out of the box with these two IDEs. With BASCOM-AVR you get development speed. Atmel Studio 6 and avr-gcc are tools for writing highly optimized code, using the techniques I have demonstrated, with application note sample code. There is a learning curve for this, but with some training you should be able to do it quite quickly, gradually reusing more of your own code base. It's an investment, like any other. A good driver library would of course mean a better starting point.

I can find no support for the notion that highly optimized code must mean low development speed and vice versa, nor the prejudice that writing efficient code is wasteful ("HW is cheap – developers expensive"). Instead, this thesis shows that the dedicated programmer can improve his or her 'baseline'. By gaining a better understanding of the HW and IDE ecosystem, it should indeed be possible to write better code about as fast as the uneducated developer writes worse code.

Is writing efficient code worth the effort? There are many different types of answers to that:

- For many simple one-task microcontroller applications, performance and code size (including energy efficiency) are not the primary concern.
- At the other end of the embedded SW spectrum – e.g. smartphones – all of the above are of great importance.
- In between, there are many applications that are battery-powered and hence benefit from energy-efficient fast code. (This thesis has mostly focused on code size, but in all relevance, the techniques are the same for clock cycle count reduction.)
- In large volumes, the cost saving of using smaller (=cheaper) devices is substantial.
- In PC- and server-class computing, performance and energy-efficiency can be very important. Although the differences in HW and IDEs are too great for directly applying the thesis results, I believe that they give insights that to a great extent are valid for other types of computers.

I have shown that once you master a few reasonably simple techniques, it really isn't that much of an effort. In the end it's up to you: Do you want to be a master?

9 References

Most of my references are datasheets and software documentation produced by Atmel Corporation and MCS Electronics or Wikipedia, without a reference to any specific individual. Hence, only some of the sources have a named author.

1. Fuller SH. Future of Computing Performance : Game Over or Next Level? National Academies Press; 2011.

2. Burlin J. Telephone interview IAR Systems AB. 2014.

3. megaAVR Microcontrollers [Internet]. [cited 2014 Feb 13]. Available from: http://www.atmel.com/products/microcontrollers/avr/megaavr.aspx

4. Home - MCS Electronics [Internet]. [cited 2014 Feb 13]. Available from: http://www.mcselec.com/

5. AVR XMEGA Microcontrollers [Internet]. [cited 2014 Feb 13]. Available from: http://www.atmel.com/products/microcontrollers/avr/avr_xmega.aspx

6. Atmel® Studio 6 - Supporting Two Architectures: AVR and ARM, with One Integrated Studio - Overview [Internet]. [cited 2014 Feb 12]. Available from: http://www.atmel.se/microsite/atmel_studio6/

7. Atmel Software Framework [Internet]. [cited 2014 Feb 13]. Available from: http://www.atmel.com/tools/avrsoftwareframework.aspx?tab=overview

8. Row-major order - Wikipedia, the free encyclopedia [Internet]. [cited 2014 Feb 14]. Available from: http://en.wikipedia.org/wiki/Row-major_order

9. Atmel Corporation. Atmel Corporation - Microcontrollers, 32-bit, and touch solutions [Internet]. [cited 2014 Jun 3]. Available from: http://www.atmel.com/

10. Reduced instruction set computing [Internet]. Wikipedia, the free encyclopedia. 2014 [cited 2014 Feb 13]. Available from: http://en.wikipedia.org/w/index.php?title=Reduced_instruction_set_computing&oldid=594087688

11. Harvard architecture [Internet]. Wikipedia, the free encyclopedia. 2013 [cited 2013 Dec 28]. Available from: http://en.wikipedia.org/w/index.php?title=Harvard_architecture&oldid=585324105

12. Atmel AVR XMEGA AU Manual [Internet]. 2013 [cited 2013 Dec 28]. Available from: http://www.atmel.se/Images/Atmel-8331-8-and-16-bit-AVR-Microcontroller-XMEGA-AU_Manual.pdf

13. The Story of AVR - YouTube [Internet]. 2008 [cited 2013 Dec 28]. Available from: http://www.youtube.com/watch?v=HrydNwAxbcY

14. AVR32 [Internet]. Wikipedia, the free encyclopedia. 2013 [cited 2013 Dec 28]. Available from: http://en.wikipedia.org/w/index.php?title=AVR32&oldid=587706001

15. AT91SAM [Internet]. Wikipedia, the free encyclopedia. 2013 [cited 2013 Dec 28]. Available from: http://en.wikipedia.org/w/index.php?title=AT91SAM&oldid=584613739

16. AVR and AVR32 - Quick Reference Guide [Internet]. 2009 [cited 2013 Dec 28]. Available from: http://www.atmel.se/Images/doc4064.pdf

17. Gaillard F, Eieland A. Microprocessor or Microcontroller [Internet]. 2013 [cited 2013 Dec 28]. Available from: http://www.atmel.se/Images/MCU_vs_MPU_Article.pdf

18. Atmel AVR instruction set [Internet]. Wikipedia, the free encyclopedia. 2013 [cited 2013 Dec 28]. Available from: http://en.wikipedia.org/w/index.php?title=Atmel_AVR_instruction_set&oldid=571841646

19. ATmega164A/PA/324A/PA/644A/PA/1284/P Complete [Internet]. 2012 [cited 2013 Dec 28]. Available from: http://www.atmel.se/Images/Atmel-8272-8-bit-AVR-microcontroller-ATmega164A_PA-324A_PA-644A_PA-1284_P_datasheet.pdf

20. Myklebust G. The AVR Microcontroller and C Compiler Co-Design [Internet]. 1996 [cited 2013 Dec 28]. Available from: http://www.atmel.com/dyn/resources/prod_documents/COMPILER.pdf

21. AVR035: Efficient C Coding for 8-bit AVR microcontrollers [Internet]. 2004 [cited 2013 Dec 28]. Available from: http://www.atmel.se/Images/doc1497.pdf

22. Atmel AVR4027: Tips and Tricks to Optimize Your C Code for 8-bit AVR Microcontrollers [Internet]. 2011 [cited 2013 Dec 29]. Available from: http://www.atmel.se/Images/doc8453.pdf

23. AVR Instruction Set [Internet]. 2012 [cited 2013 Dec 29]. Available from: http://www.atmel.se/Images/doc0856.pdf

24. ATxmega64A1U/128A1U Complete [Internet]. 2012 [cited 2013 Dec 30]. Available from: http://www.atmel.com/Images/Atmel-8385-8-and-16-bit-AVR-Microcontroller-ATxmega64A1U-ATxmega128A1U_datasheet.pdf

25. Application Notes - - Atmel Documentation Overview [Internet]. ???? [cited 2013 Dec 28]. Available from: http://atmel.no/webdoc/atmel.docs/atmel.docs.3.application.note.html

26. AVR XMEGA [Internet]. 2008 [cited 2013 Dec 30]. Available from: http://www.atmel.com/Images/doc7925.pdf

27. Introducing a New Breed of Microcontrollers for 8/16-bit Applications [Internet]. 2008 [cited 2013 Dec 30]. Available from: http://www.atmel.com/Images/doc7926.pdf

28. AVR1005: Getting started with XMEGA [Internet]. 2009 [cited 2013 Dec 30]. Available from: http://www.atmel.com/Images/doc8169.pdf

29. AVR1000: Getting Started Writing C-code for XMEGA [Internet]. 2008 [cited 2013 Dec 30]. Available from: http://www.atmel.com/Images/doc8075.pdf

30. BASCOM-AVR online help [Internet]. [cited 2014 Feb 12]. Available from: http://avrhelp.mcselec.com/index.html

31. BASCOM-AVR - MCS Electronics [Internet]. [cited 2014 Feb 12]. Available from: http://www.mcselec.com/index.php?option=com_content&task=view&id=14&Itemid=41

32. Forum - MCS Electronics [Internet]. [cited 2014 Feb 12]. Available from: http://www.mcselec.com/index2.php?option=com_forum&Itemid=59

33. avr-gcc - GCC Wiki [Internet]. [cited 2014 Feb 12]. Available from: http://gcc.gnu.org/wiki/avr-gcc

34. Documentation:AVR GCC/AVR GCC Tool Collection - AVRFreaks Wiki [Internet]. [cited 2014 Feb 12]. Available from: http://www.avrfreaks.net/wiki/index.php/Documentation:AVR_GCC/AVR_GCC_Tool_Collection

35. Optimize Options - Using the GNU Compiler Collection (GCC) [Internet]. 2014 [cited 2014 Jan 23]. Available from: http://gcc.gnu.org/onlinedocs/gcc-4.8.2/gcc/Optimize-Options.html#Optimize-Options

36. AVR Libc [Internet]. [cited 2014 Feb 12]. Available from: http://www.nongnu.org/avr-libc/user-manual/

37. Atmel AVR4029: Atmel Software Framework - Getting Started [Internet]. 2013 [cited 2013 Dec 30]. Available from: http://www.atmel.com/Images/Atmel-8431-8-and32-bit-Microcontrollers-AVR4029-Atmel-Software-Framework-User-Guide_Application-Note.pdf

38. Atmel AVR4030: AVR Software Framework - Reference Manual [Internet]. 2012 [cited 2013 Dec 30]. Available from: http://www.atmel.com/Images/doc8432.pdf

39. AVR Freaks [Internet]. [cited 2014 Feb 12]. Available from: http://www.avrfreaks.net/

40. AVR Assembler User Guide [Internet]. Pre-W2k [cited 2013 Dec 29]. Available from: http://www.atmel.com/images/doc1022.pdf

41. Atmel AT1886: Mixing Assembly and C with AVRGCC [Internet]. 2012 [cited 2013 Dec 28]. Available from: http://www.atmel.se/Images/doc42055.pdf

42. AVR000: Register and Bit-Name Definitions for the 8-bit AVR Microcontroller [Internet]. 2009 [cited 2013 Dec 30]. Available from: http://www.atmel.com/Images/doc0931.pdf

43. AVR001: Conditional Assembly and portability macros [Internet]. 2008 [cited 2013 Dec 30]. Available from: http://www.atmel.com/Images/doc2550.pdf

44. Woxulv M. Telephone conversation Atmel Sweden. 2013.

45. Inline assembler: Possible to specify code placement? [Internet]. [cited 2014 Feb 7]. Available from: http://www.mcselec.com/index2.php?option=com_forum&Itemid=59&page=viewtopic&t=11718&highlight=reduceivr

46. ASF ATmega System Clock Management Documentation [Internet]. [cited 2014 Jan 23]. Available from: http://asf.atmel.com/docs/3.13.1/mega/html/group__sysclk__group.html

47. AVR306: Using the AVR UART in C on tinyAVR and megaAVR devices [Internet]. 2002 [cited 2013 Dec 28]. Available from: http://www.atmel.se/Images/doc1451.pdf

48. View topic - HW lookup table for address pointers - good or bad idea? :: AVR Freaks [Internet]. [cited 2014 Feb 12]. Available from:

http://www.avrfreaks.net/index.php?name=PNphpBB2&file=viewtopic&t=140145&postdays=0&postorder=asc&sid=e3717428757c8e0fcac437a5ae45306b

49. ASF ATxmega System Clock Management Documentation [Internet]. [cited 2014 Jan 23]. Available from: http://asf.atmel.com/docs/3.13.1/xmegaa/html/group__sysclk__group.html

50. ASF Source Code Documentation - Quick start guide for Serial Interface service [Internet]. [cited 2014 Feb 13]. Available from: http://asf.atmel.com/docs/3.13.1/xmegaa/html/serial_quickstart.html

51. ASF Source Code Documentation - Advanced use case - Send a packet of serial data [Internet]. [cited 2014 Feb 13]. Available from: http://asf.atmel.com/docs/3.13.1/xmegaa/html/serial_use_case_1.html

52. ASF Source Code Documentation - Quick start guide for USART module [Internet]. [cited 2014 Feb 13]. Available from: http://asf.atmel.com/docs/3.13.1/xmegaa/html/xmega_usart_quickstart.html

53. White T, GTKNarwhal. AVR Freaks :: View topic - tc.h in the xmega atmel framework [Internet]. XMEGA forum - tc.h in the xmega atmel framework. 2011 [cited 2014 Jan 6]. Available from: http://www.avrfreaks.net/index.php?name=PNphpBB2&file=printview&t=115038&start=0

54. ASF Source Code Documentation - Clock Management [Internet]. [cited 2014 Feb 13]. Available from: http://asf.atmel.com/docs/3.13.1/xmegaa/html/group__clk__group.html

55. ASF Source Code Documentation - XMEGA-A1 Xplained Board Configuration [Internet]. [cited 2014 Feb 13]. Available from: http://194.19.124.62/docs/latest/xmega.drivers.des.unit_tests.xmega_a1_xplained/html/group__atxmega128a1__xpld__config.html

56. AVR1522: XMEGA-A1 Xplained Training - XMEGA USART [Internet]. 2011 [cited 2014 Jan 6]. Available from: http://www.atmel.com/Images/doc8408.pdf

57. View topic - AS 6.1 ASF ATxmega support for multiple UART and ISR :: AVR Freaks [Internet]. [cited 2014 Feb 13]. Available from: http://www.avrfreaks.net/index.php?name=PNphpBB2&file=viewtopic&t=139232

58. ASF Source Code Documentation - Quick start guide for PMIC driver [Internet]. [cited 2014 Feb 13]. Available from: http://asf.atmel.com/docs/3.13.1/xmegaa/html/xmega_pmic_quickstart.html

59. ASF Source Code Documentation - Quick Start Guide for the System Clock Management service (XMEGA) [Internet]. [cited 2014 Feb 13]. Available from: http://asf.atmel.com/docs/3.13.1/xmegaa/html/sysclk_quickstart.html

60. ASF Source Code Documentation - USART module (USART) [Internet]. [cited 2014 Feb 13]. Available from: http://asf.atmel.com/docs/3.13.1/xmegaa/html/group__usart__group.html

61. View topic - AS 6.1 ASF ATxmega USART library compiled code size :: AVR Freaks [Internet]. [cited 2014 Feb 13]. Available from: http://www.avrfreaks.net/index.php?name=PNphpBB2&file=viewtopic&p=1126226#1126226

62. AVR1307: Using the XMEGA USART [Internet]. 2008 [cited 2013 Dec 30]. Available from: http://www.atmel.com/Images/doc8049.pdf

63. Atmel-8331-8-and-16-bit-AVR-Microcontroller-XMEGA-AU_Manual.pdf [Internet]. 2013 [cited 2014 Jan 6]. Available from: http://www.atmel.com/Images/Atmel-8331-8-and-16-bit-AVR-Microcontroller-XMEGA-AU_Manual.pdf

64. AVR - gcc - How can I turn off >64K ram support for ATxmega128a1 target? [Internet]. 2009 [cited 2014 Jan 22]. Available from: http://avr.2057.n7.nabble.com/How-can-I-turn-off-gt-64K-ram-support-for-ATxmega128a1-target-td10341.html

65. View topic - [TUT][C]Optimization and the importance of volatile in GCC :: AVR Freaks [Internet]. [cited 2014 Jan 27]. Available from: http://www.avrfreaks.net/index.php?name=PNphpBB2&file=viewtopic&t=97382&start=all&postdays=0&postorder=asc

66. View topic - AVR-GCC. How to remove Interrupt table? :: AVR Freaks [Internet]. [cited 2014 Jan 27]. Available from: http://www.avrfreaks.net/index.php?name=PNphpBB2&file=viewtopic&p=1131093#1131093

67. Schick B. AVR Bootloader FAQ [Internet]. 2009. Available from: blog.schicks.net/wp-content/uploads/2009/09/bootloader_faq.pdf

68. Link Options - Using the GNU Compiler Collection (GCC) [Internet]. [cited 2014 Jan 27]. Available from: http://gcc.gnu.org/onlinedocs/gcc/Link-Options.html

69. Georg-Johann Lay - Re: [avr-gcc-list] [Patch, avr] Shrink interrupt vector table down to la [Internet]. [cited 2014 Jan 27]. Available from: http://sourceware.org/ml/binutils/2013-02/msg00180.html

70. Atmel-42083-XMEGA-E-Using-the-XCL-Module_Application-Note_AT01084.pdf [Internet]. [cited 2014 Feb 5]. Available from: http://www.atmel.se/Images/Atmel-42083-XMEGA-E-Using-the-XCL-Module_Application-Note_AT01084.pdf

71. AVR-GCC-Codeoptimierung - Mikrocontroller.net [Internet]. [cited 2014 Feb 14]. Available from: http://www.mikrocontroller.net/articles/AVR-GCC-Codeoptimierung

72. Knaggs P, Welsh S. ARM_AssyLang.pdf [Internet]. 2004 [cited 2014 Feb 14]. Available from: http://www.eng.auburn.edu/~nelson/courses/elec5260_6260/ARM_AssyLang.pdf

73. Jamil T. RISC versus CISC. IEEE Potentials. 1995 Aug;14(3):13–6.

74. Blem E, Menon J, Sankaralingam K. Power struggles: Revisiting the RISC vs. CISC debate on contemporary ARM and x86 architectures. 2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA2013). 2013. p. 1–12.

75. Koopman P. Stack Computers: 1.4 WHY ARE STACKS USED IN COMPUTERS? [Internet]. [cited 2014 Feb 5]. Available from: http://www.ece.cmu.edu/~koopman/stack_computers/sec1_4.html

76. AVR Freaks :: View topic - Why AVRFreaks members do not like XMEGA [Internet]. [cited 2014 Feb 7]. Available from: http://www.avrfreaks.net/index.php?name=PNphpBB2&file=printview&t=103269

77. Which AVRStudio Version is best? AVRStudio 6 versus AVRStudio 4 [Internet]. 2013 [cited 2014 Feb 7]. Available from: http://www.kanda.com/blog/microcontrollers/avr-microcontrollers/avrstudio-explored/

78. AVR Freaks :: View topic - AVR Studio 5 Released - Get Your BETA Here! [Internet]. [cited 2014 Feb 7]. Available from: http://www.avrfreaks.net/index.php?name=PNphpBB2&file=printview&t=103949

79. Arndt N. AN #186 - KokkeKat FAT-free SD card library - MCS Electronics [Internet]. [cited 2014 Feb 7]. Available from: http://www.mcselec.com/index.php?option=com_content&task=view&id=291&Itemid=57

80. Arndt N. KokkeKat FAT-free SD card lib forum [Internet]. [cited 2014 Feb 7]. Available from: http://www.mcselec.com/index2.php?option=com_forum&Itemid=59&page=viewforum&f=18

81. Barrett SF, Pack DJ. Microcontrollers fundamentals for engineers and scientists. [San Rafael, Calif.]: Morgan & Claypool Publishers; 2006.

82. Barrett SF, Pack DJ. Atmel AVR microcontroller primer: Programming and interfacing, second edition. Atmel AVR Microcontroller Primer Program Interfacing Second Ed. 2012;39:1–246.

83. Barrett SF. Embedded Systems Design with the Atmel AVR Microcontroller. Part I. San Rafael, CA, USA: Morgan & Claypool Publishers; 2010. xiii+164 p.

84. Barrett SF. Embedded Systems Design with the Atmel AVR Microcontroller. Part II. San Rafael, CA, USA: Morgan & Claypool Publishers; 2010. xii+296 p.

85. Salewski F, Kowalewski S. Hardware Platform Design Decisions in Embedded Systems: A Systematic Teaching Approach. SIGBED Rev. 2007 Jan;4(1):27–35.

86. ATAM: Method for Architecture Evaluation | SEI Digital Library [Internet]. [cited 2014 Feb 9]. Available from: http://resources.sei.cmu.edu/library/asset-view.cfm?assetid=5177

87. Slade M, Jones MH, Scott JB. Choosing the right microcontroller: A comparison of 8-bit Atmel, Microchip and Freescale MCUs [Internet]. Faculty of Engineering, The University of Waikato; 2011 Nov. Available from: http://researchcommons.waikato.ac.nz/handle/10289/5938

88. Wong W. IDEs of change. Electron Des. 2006;54(9):52–60.

89. Atmel AVR Studio 5 Provides Fully Integrated Development Platform for Embedded Microcontroller Designs. - Free Online Library [Internet]. [cited 2014 Feb 14]. Available from: http://www.thefreelibrary.com/Atmel+AVR+Studio+5+Provides+Fully+Integrated+Development+Platform+for...-a0250183971

90. Silvestre J, Cardoso D, Correia A. AVR Studio 5 | ImaginationOverflow [Internet]. [cited 2014 Feb 14]. Available from: http://imaginationoverflowsw.wordpress.com/tag/avr-studio-5/

91. Engineering Softwares: AVR Studio 5 - Atmel Studio 6 [Internet]. [cited 2014 Feb 14]. Available from: http://engineering-softwares.blogspot.se/2013/04/avr-studio-5-atmel-studio-6.html

92. Nath N. Atmel Studio 6 – Install Guide, Walk-Through, Review | Nicky goes Nuts and Bolts [Internet]. [cited 2014 Feb 14]. Available from: http://nishantnath.com/2012/05/05/atmel-studio-6-install-guide-walk-through-review/

93. Tomar A. Atmel: Atmel Studio 6 IDE Overview | element14 [Internet]. [cited 2014 Feb 14]. Available from: http://www.element14.com/community/docs/DOC-46581

94. New Atmel Studio 6 Release with Support for ARM Microcontrollers [Internet]. [cited 2014 Feb 14]. Available from: https://www.futurlec.com/News/Atmel/Studio6.shtml

95. Stroustrup B. Abstraction and the C++ machine model / Embedded Software and Systems. Springer Berlin / Heidelberg; 2005.

96. Wybolt N. Experiences with C++ and Object-oriented Software Development. SIGSOFT Softw Eng Notes. 1990 Apr;15(2):31–9.

97. Ada-Europe International Conference on Reliable Software Technologies J, Chatzigeorgiou A, Blieberger J, Strohmeier A. Evaluating performance and power of object-oriented vs. procedural programming in embedded processors / Reliable Software Technologies - Ada-Europe 2002. 2002.

98. Titzer BL. Virgil: Objects on the head of a pin. ACM SIGPLAN Not. 2006;41(10):191–207.

99. Program optimization - Wikipedia, the free encyclopedia [Internet]. [cited 2014 Feb 15]. Available from: http://en.wikipedia.org/wiki/Program_optimization

100. C code optimisation | Member Robot Tutorials [Internet]. [cited 2014 Feb 15]. Available from: http://www.societyofrobots.com/member_tutorials/node/202

101. EventHelix.com Inc. Optimizing C and C++ Code [Internet]. Optimizing C and C++ Code. ???? [cited 2013 Oct 5]. Available from: http://www.eventhelix.com/realtimemantra/basics/optimizingcandcppcode.htm#.UlAfOVPRVdh

102. Shlomi F. Optimizing Code for Speed [Internet]. 2009 [cited 2013 Oct 5]. Available from: http://www.shlomifish.org/philosophy/computers/optimizing-code-for-speed/

103. Edwards LARW. Embedded System Design on a Shoestring. Newnes; 2003. 1 p.

104. View topic - Memory barrier: what it does and what it does not do :: AVR Freaks [Internet]. [cited 2014 Jan 27]. Available from: http://www.avrfreaks.net/index.php?name=PNphpBB2&file=viewtopic&t=94571&start=all&postdays=0&postorder=asc

105. Taylor C. Mixing C and Assembly [Internet]. [cited 2014 Feb 17]. Available from: http://msoe.us/taylor/tutorial/ce2810/candasm

106. Doncaster R. Nerd Ralph: Trimming the fat from avr-gcc code [Internet]. 2013 [cited 2014 Jan 26]. Available from: http://nerdralph.blogspot.ca/2013/12/trimming-fat-from-avr-gcc-code.html

107. Hyde R. Write Great Code Volume 1: Understanding the machine. San Francisco: No Starch Press; 2004.

108. Hyde R. Write Great Code Volume 2: Thinking low-level, writing high-level. San Francisco No Starch Press; 2006.

109. Hyde R. The Art of Assembly Language, Second Edition. No Starch Press; 2010.

110. Hyde R. The Fallacy of Premature Optimization. Ubiquity [Internet]. 2009 Feb [cited 2014 Feb 9];2009(February). Available from: http://doi.acm.org/10.1145/1513450.1513451

111. Optimizing Code Performance and Size for Stellaris® Microcontrollers [Internet]. [cited 2014 Feb 15]. Available from: http://www.ti.com/general/docs/lit/getliterature.tsp?literatureNumber=spma014&fileType=pdf

112. Yiu, Frame J, Andrew. (ARM) 32-Bit Microcontroller Code Size Analysis Draft 1.2.4. [Internet]. [cited 2013 Oct 5]. Available from: www.arm.com/files/pdf/ARM_Microcontroller_Code_Size_(full).pdf

113. Isensee P. C++ Optimization Strategies and Techniques [Internet]. ???? [cited 2013 Oct 5]. Available from: http://www.tantalon.com/pete/cppopt/main.htm

114. Lee ME. Optimization of Computer Programs in C [Internet]. 1999 [cited 2013 Oct 5]. Available from: http://leto.net/docs/C-optimization.php

115. Hsieh P. Programming Optimization [Internet]. 2007 [cited 2013 Oct 9]. Available from: http://www.azillionmonkeys.com/qed/optimize.html

116. University of Iowa. Tips for Optimizing C/C++ Code [Internet]. 2007 [cited 2013 Oct 5]. Available from: https://www.cs.uiowa.edu/~cwyman/classes/spring07-22C251/handouts/optimize.pdf

117. Ghosh K. Writing Efficient C and C Code Optimization [Internet]. 2004 [cited 2013 Oct 5]. Available from: http://www.codeproject.com/Articles/6154/Writing-Efficient-C-and-C-Code-Optimization

118. Shalom H. Writing Efficient C Code for Embedded Systems [Internet]. 2010 [cited 2013 Oct 5]. Available from: http://www.rt-embedded.com/blog/archives/writing-efficient-c-code-for-embedded-systems/

119. Chan W. Writing optimized C code for microcontroller applications. Proceedings of Embedded Systems Conference, 1-4 March 1999. Miller Freeman; 1999. p. 45–57.

120. Ganssle J. The Firmware Handbook: Embedded Technology. Newnes; 2004. 385 p.

121. Ayache N, Amadio RM, Regis-Gianas Y. Certifying and Reasoning on Cost Annotations in C Programs. Formal Methods for Industrial Critical Systems 17th International Workshop, FMICS 2012, 27-28 Aug 2012. Springer-Verlag; 2012. p. 32–46.

122. Johnson NE. Code size optimization for embedded processors [Internet]. University of Cambridge, Computer Laboratory; 2004 Nov p. 159. Report No.: 607. Available from: www.cl.cam.ac.uk/techreports/UCAM-CL-TR-607.pdf

123. Leupers R. Compiler design issues for embedded processors. IEEE Des Test Comput. 2002;19(4):51–8.

124. Naik M, Palsberg J. Compiling with code-size constraints. Joint Conference on Languages, Compilers and Tools for Embedded Systems and Software and Compilers for Embedded Systems, June 19, 2002 - June 21, 2002. Association for Computing Machinery; 2002. p. 120–9.

125. Alba C, Carro L, Lima A, Suzim A. Embedded systems design with frontend compilers. Proceedings of the 1996 International Conference on Computer Design, ICCD'96, October 7, 1996 - October 9, 1996. IEEE; 1996. p. 200–5.

126. De Bus B, De Sutter B, Van Put L, Chanet D, De Bosschere K. Link-time optimization of ARM binaries. ACM SIGPLAN Not. 2004;39(7):211–20.

127. Zhao M, Childers B, Soffa ML. Predicting the impact of optimizations for embedded systems. ACM SIGPLAN Not. 2003;38(7):1–11.

128. Yang X. Eliminating the call stack to save RAM. ACM SIGPLAN Not. 2009;44(7):60–9.

129. Lin FX, Wang Z, Likamwa R, Zhong L. Reflex: Using low-power processors in smartphones without knowing them. 17th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2012, March 3, 2012 - March 7, 2012. Association for Computing Machinery; 2012. p. 13–24.

130. Brandon J. AVR op codes [Internet]. [cited 2014 Feb 14]. Available from: http://www.zbasic.net/download/AVR_opcodes.txt

131. AVR034: Mixing C and Assembly Code with IAR Embedded Workbench for 8-bit AVR microcontrollers [Internet]. 2003 [cited 2013 Dec 29]. Available from: http://www.atmel.com/Images/doc1234.pdf

132. Forum - MCS Electronics - Register conventions [Internet]. [cited 2014 Feb 13]. Available from: http://www.mcselec.com/index2.php?option=com_forum&Itemid=59&page=viewtopic&p=60805#60805

133. AVR Freaks :: View topic - How to combine C program with external ASM [Internet]. [cited 2014 Jan 30]. Available from: http://www.avrfreaks.net/index.php?name=PNphpBB2&file=printview&t=112779&start=0

134. WinAVR : AVR-GCC for Windows [Internet]. [cited 2014 Feb 12]. Available from: http://winavr.sourceforge.net/

135. Which AVR Studio and C compiler for AVR 8-bit microcontroller and JTAGICE? [Internet]. [cited 2014 Feb 14]. Available from: http://www.motherboardpoint.com/which-avr-studio-and-c-compiler-avr-8-bit-microcontroller-and-jtagice-t257051.html

136. Optimizing C/C++ Compilers and Debuggers from IAR Systems - IAR [Internet]. [cited 2014 Feb 15]. Available from: http://www.iar.com/Products/IAR-Embedded-Workbench/

137. CodeVisionAVR. The Lowest Price on the Web. High Performance ANSI C Compiler for Atmel AVR microcontrollers [Internet]. [cited 2014 Feb 15]. Available from: http://www.codevision.be/codevisionavr

10 Appendix A

A.1 Response from the IDE companies

A.1.1 History and plans for the future

Mark Alberts, the owner of MCS Electronics, tells the history of BASCOM-AVR and his plans for the future:

"As you can find in the help, I wrote BASCOM-LT for Windows 3.1 as a tool for personal use. This was for the 8051 family. The 8051 has just a few registers but you can use bank switching. Further it has very limited internal memory. All this memory needs to be used for the stack and internal variables.

Normally one would create a stack machine in order to support expressions but as the memory was limited and I had no need for complex expressions I decided to allow only simple assignments. The real power of BASCOM was that I added support for hardware. Controlling the serial port or LCD was usually done by assembler but now a simple CONFIG would allow you to use hardware without figuring out protocols.

Since writing the software was a lot of work and hardware was not as cheap as today, i decided to make a commercial version. I added a help file and simulator and many hobbyists liked the software. In the first year I got many wishes and the tool grew enormeously. With Windows 95, I rewrote the tool to support arrays and floating point. And many different 8051 processors were supported.

When users asked for an AVR version I had to study the AVR architecture which was very different. But I recognized the great linear memory and ISP programming with 5V, so this was a step forward compared to the 89c2051. Again I rewrote the software where I used the old BASCOM-8051 as a basis. That was probably not the smartest thing to do because the AVR was well suited as stack machine. And the additional work would have paid off. But I was doing all this in my spare time, and I was giving support to BASCOM-8051 users with new features, helping with their hardware problems, etc. Of course a lot of improvements were made compared to the 8051. Doubles were added, trig was added and so on.

With more users also came more support and ideas. I had to chose between my job and supporting BASCOM users so I quit my job to work full time on BASCOM, support, custom projects. All user ideas that were hard to implement I worked out in a new version where I rewrote almost everything. A number of these features you can find in BASCOM-AVR like the code explorer, and draw indents, proper indent, unused code marking.

The Xmega was a great new chip but had a lot of impact on support. As you can see from the help there are a lot of CONFIG commands just for xmega. In my normal code I used pointers to registers but that was not possible for xmega. So I had to recode that in a way all other code would still work. And all new hardware took a lot of time to implement. But it is great that these chips have so much hardware inside. With the linear fixed address I got carried away and I did not have a look at the impact. So it was good you had a look at that. All other compiler things I knew already.

As you probably have found out, is that the asm libs are efficient but depeding on the statement one uses, things can be done more efficient. Once in a while when I look at some code I find inefficient code which I then change, but some improvements require big changes. That is why I left that for another product. At some stage it is best to rewrite completely and not to try to change code. This because any mistake I make will cause problems for users. I can do only limited tests with hardware.

My goal was never to make the best compiler of the world but the easiest software tool for processors. With many professional users, I shifted focus to a better tool (like more build in error checking). And the focus for the next IDE is more about safe and reliable commands. So no poking, pointers, recursion etc."

(MCS Electronics' owner Mark Alberts, 2014-02-11)

A.1.2 BASCOM-AVR feedback on v0.9

"Hello Niclas

thanks for the update. I have read it with interest.

Here is my comment/opinion.

- bascom can handle recursion. and you can use isr's multiple times and/or at the same time. [This is in response to a previous error in my thesis.]

- it is great that you researched the dynamic xmega handling in bascom. I never realized the consequence. i will add a static option so the user can control how it works.

- i knew re-using the value of registers could optimize things. i used that for a new assembler but it is better if the compiler deals with it.

- i think you focused too much on pieces of code and how you could optimize it. It is also not a real good comparison to compare different products. You could better have checked the difference between normal mega, xmega and ARM using studio. Especially because studio supports them all.

- instead of focusing on pieces of code, a real world app would have been a better test IMO. If you write 1000 or 10000 lines of code, and you need to alter or change it later it will be harder if you write a lot of custom asm.

also, since some chips do not have some instructions, the code will not work, work different or requires modifications when porting to a new chip or platform. So what I miss in the investigation is what happens if you code some functions in asm and port it from mega to xmega to ARM compared to using plain high level code. for example 1 line, or 10, or 100.

In any case you made it clear that using a high level language including cpp, has a penalty. But that was clear already.

You put alot of effort in it and i hope your professor likes it. In any case it helped me.

best regards,

Mark"

(MCS Electronics' owner Mark Alberts, 2014-02-19)

A.1.3 Atmel's response

"I have received an answer from one of my colleagues at the Trondheim support about this. Unfortunately they have already filled the quota of students they can help this year. I will try to find an alternative way or some other contact with whom you can talk, but at this point the forecast is a bit dark."

(Marcus Woxulv, Atmel Sweden tech support, 2013-06-28, translated from Swedish)

A.2 Additional sources

There are two types of sources for this thesis. One is datasheets, application notes, documentation, user forum posts, and so on, that contain certain specific information. There's an abundance of these. They are frequently referenced in the next chapter that presents the AVR microcontroller and throughout the thesis.

However, a scientific publication is supposed to take off from previous writers' work. In this area I haven't been very successful. Either I have been searching for the wrong terms or there has been very little interest in these areas. I have spent several days browsing IEEE Explore, ACM Digital Library, Inspec, Compendex, Referex, and the internet.

A.2.1 AVR ATmega general functionality

There are several books and much material on ATmega functionality. Steven F. Barrett has published a few (partly overlapping) ones:

- "Microcontrollers Fundamentals for Engineers and Scientists" (81)
- "Atmel AVR Microcontroller Primer: Programming and Interfacing, 2nd edition" (82)
- "Embedded Systems Design with the Atmel AVR Microcontroller Part I" (83)
- "Embedded Systems Design with the Atmel AVR Microcontroller Part II" (84)

A.2.2 Hardware platform evaluation

- "Hardware Platform Design Decisions in Embedded Systems - A Systematic Teaching Approach" (85) lists a number of important HW attributes when deciding on a (teaching) platform covering both microcontrollers and soft-core programmable logic devices.
- "ATAM (Architecture Tradeoff Analysis Method): Method for Architecture Evaluation" (86) is a very thorough methodology that is mostly outside of the scope of this thesis.
- "Choosing the right microcontroller: A comparison of 8-bit Atmel, Microchip and Freescale MCUs" (87)

A.2.3 IDE evaluation

- "IDEs of change" (88) is an overview of development platforms in 2006.

Except for the above, I found very little in this area. Nothing else in the scientific databases and only a few mostly uninteresting hits on Google and Bing:

- A press release for AVR Studio 5: (89) [103]
- A short blog post on one person's experience from AVR Studio 5 and AVR32: (90) [104]
- Product presentations for AVR Studio 5 and Atmel Studio 6: (91) [105]
- An installation walk-through and a tiny "review" of Atmel Studio 6: (92) [106] It is mostly a change list from the previous version and a conclusion that WinAVR with AVR Studio 4 is preferred.

[103] http://www.thefreelibrary.com/Atmel+AVR+Studio+5+Provides+Fully+Integrated+Development+Platform+for...-a0250183971

[104] http://imaginationoverflowsw.wordpress.com/tag/avr-studio-5/

[105] http://engineering-softwares.blogspot.se/2013/04/avr-studio-5-atmel-studio-6.html

[106] http://nishantnath.com/2012/05/05/atmel-studio-6-install-guide-walk-through-review/

- A commercial product presentation of Atmel Studio 6 published by an electronic component vendor: (93) [107]
- A press release for Atmel Studio 6 published by another component vendor: (94) [108]

Actually, the information I found that most resembles an evaluation are user forum posts with outbursts from annoyed users. Some are referenced in the analysis.

A.2.4 Analyses of programming models

- “Abstraction and the C++ Machine Model” (95)
- “Experiences with C++ and Object-Oriented Software” (96)
- “Evaluating Performance and Power of Object-Oriented Vs. Procedural Programming in Embedded Processors” (97) presents a test that showed a significant penalty in code size and RAM and a modest increase in instructions and clock cycles when using OOP compared to procedural programming.
- “Virgil: Objects on the Head of a Pin” (98) presents *“a lightweight objectoriented language designed with careful consideration for resource-limited domains.”*

A.2.5 Efficient programming and (inline) assembly

This is a big area with lots of material. User forums often contain both answers to specific questions and more or less well-structured how-tos. I frequently reference this type of sources in my thesis.

- Wikipedia has a good overview on program optimization: (99) [109]
- The German web site www.mikrocontroller.net has an in-depth page on AVR-GCC code optimization: (71) [110] It also points out the consequences of only having 2-3 memory pointers.
- “C code optimization” at Society of Robots (100) [111] claims (in “08 - H files versus C files”) that *“If you define a method in a .h file then it is normally only compiled once and any references to it end up calling it. So the difference between .h and .c is small. Even if you define a method in a .h file that is never called then it still gets compiled. Equally: if you compile a .c file into a library and the rest of the code only accesses one of the methods in that file then the entire compiled .c file will be added to your program.”*
- “Optimizing C and C++ Code” (101)
- “Optimizing Code for Speed” (102) [112]
- The book “Embedded System Design on a Shoestring” has a very large section on the GNU toolchain (for ARM). (103)
- Optimization and volatile (65) [113]
- Memory barrier (104) [114]

[107] http://www.element14.com/community/docs/DOC-46581
[108] https://www.futurlec.com/News/Atmel/Studio6.shtml
[109] http://en.wikipedia.org/wiki/Program_optimization
[110] http://www.mikrocontroller.net/articles/AVR-GCC-Codeoptimierung
[111] http://www.societyofrobots.com/member_tutorials/node/202
[112] http://www.shlomifish.org/philosophy/computers/optimizing-code-for-speed/
[113] http://www.avrfreaks.net/index.php?name=PNphpBB2&file=viewtopic&t=97382&start=all&postdays=0&postorder=asc
[114] http://www.avrfreaks.net/index.php?name=PNphpBB2&file=viewtopic&t=94571&start=all&postdays=0&postorder=asc

- “Mixing C and Assembly Languages” (105) [115]
- An approach similar to mine on incremental code reduction and elimination: (106) [116]

I also came across three books by Randall Hyde on writing assembly or assembly-friendly high-level language:

- Write Great Code Volume 1: Understanding the machine (107)
- Write Great Code Volume 2: Thinking low-level, writing high-level (108)
- The Art of Assembly Language, Second Edition (109)

He also makes a strong case against today’s unwillingness to optimize code: (110)

A.2.5.1 Optimization on competing or generic architectures

- “Optimizing Code Performance and Size for Stellaris® Microcontrollers” (111) [117]
- “32-Bit Microcontroller Code Size Analysis” (ARM) (112) [118]
- “C++ Optimization Strategies and Techniques” (PC) (113) [119]
- “Optimization of Computer Programs in C” (UNIX) (114)
- “Programming Optimization” (PC) (115) [120] has lot of information and a number of links to other sites.
- “Tips for Optimizing C/C++ Code” (116)
- “Writing Efficient C and C Code Optimization” (Windows) (117) [121]
- “Writing Efficient C Code for Embedded Systems” (ARM) (118) [122]
- “Writing optimized C code for microcontroller applications” (119) [123]
- Chapters 18 and 19 of “The Firmware Handbook” (120) deal with optimization.

A.2.5.2 Compiler-related

- “Certifying and Reasoning on Cost Annotations in C Programs” (121) treats labelling of code costs done by the compiler.
- “Code size optimization for embedded processors” (122)
- “Compiler Design Issues for Embedded Processors” (123)
- “Compiling with Code-Size Constraints” (124)
- “Embedded Systems Design with Frontend Compilers” (125)
- “Link-Time Optimization of ARM Binaries” (126)
- “Predicting the Impact of Optimizations for Embedded Systems” (127)

A.2.5.3 Other approaches

- “Eliminating the Call Stack to Save RAM” (128)
- “Reflex: Using Low-Power Processors in Smartphones without Knowing Them” (129)

[115] http://msoe.us/taylor/tutorial/ce2810/candasm
[116] http://nerdralph.blogspot.ca/2013/12/trimming-fat-from-avr-gcc-code.html
[117] http://www.ti.com/lit/an/spma014/spma014.pdf
[118] www.arm.com/files/pdf/ARM_Microcontroller_Code_Size_(full).pdf
[119] http://www.tantalon.com/pete/cppopt/main.htm
[120] http://www.azillionmonkeys.com/qed/optimize.html
[121] http://www.codeproject.com/Articles/6154/Writing-Efficient-C-and-C-Code-Optimization
[122] http://www.rt-embedded.com/blog/archives/writing-efficient-c-code-for-embedded-systems/
[123] http://www.docshut.com/imwzpt/writing-optimized-c-code-for-microcontroller-applications.html

A.3 How to disassemble

If you are using Atmel Studio 6, you select the simulator or the debugger in the Tool tab for your project. Then you click the "Start Debugging and Break" icon, which opens up the Disassembly tab. It combines your high-level code and the compiled assembler instructions in the same view.

With BASCOM-AVR, I found it easiest to compile so that it generates a .obj file. Next, start Atmel Studio 6 and click File/Open/Open Object File For Debugging. This will set up a project for this object file and you use it like above.

As I started with BASCOM-AVR, I only figured this out at the middle of that analysis. I had first used two other methods:

2. Use AVR GCC's avr-objdump.
    - Requires a reboot into Linux (or using a virtual machine?).
    - avr-objdump -s -m avr5 -D /root/Downloads/BTf2128.hex > /root/BTf2128.dump
3. Open the .hex or .bin files in an advanced hex editor and if necessary convert .hex to the humanly readable .bin format.
    - Convert the machine code manually by converting it (minding that it's little-endian) to bit code, looking up the assembly instruction (130) [124] and its details (23) [125]. Very time-consuming…
    - I used a commercial version of WinHex.

[124] http://www.zbasic.net/download/AVR_opcodes.txt
[125] http://www.atmel.se/Images/doc0856.pdf

A.4 Atmel application notes on efficient programming

A.4.1 Efficient C Coding for 8-bit AVR microcontrollers

This document is based on IAR Systems' compiler [30] and published in 2004, so some of this information isn't valid for the Atmel Studio avr-gcc compiler. (This is the case for at least parameter passing and function return.)

It shows how the four (SP, X, Y, and Z) 16-bit pointers can be used with indirect addressing and displacement or pre- or post-decrement. It also has a large section about EEPROM handling.

There are a number of examples of syntactically correct C statements together with their compiled Assembly code. The "volatile" option (force read or write, i.e. don't optimize) is mentioned, we are advised to use as "small" variables as possible, and the ways to declare variables is mentioned:

- Global: Common to the program (i.e. not defined inside a function). Must be loaded from SRAM into working registers, so their use imposes a performance penalty.
- Local: Declared and only exists inside a function. As far as possible, working registers are used directly for locals, so no penalty.
- Static local: Function-internal variables that keep their value between the function calls. Typically stored in SRAM, so performance penalty.

The global variable penalty is demonstrated; a simple assignment requires 10 code bytes and 5 clock cycles for a global, but only 2 code bytes and 1 clock cycle for a local. A static local is loaded from SRAM at the function entry but only stored back to SRAM at the function exit, which potentially decreases the penalty hit.

Global variables should as far as possible be declared as part of a structure, which enables compilation to indirect access. An example is given, showing that with only one variable the code size is the same, but each additional global inside a structure saves four code bytes.

Furthermore, it is possible (after allocation in the compiler options setup) to utilize unused I/O registers for global flags. [As mentioned above, I/O 0x00 – 0x1F is bit-accessible and I/O 0x00 – 0x03F can use the shorter IN and OUT instructions, my comment.] An example is given, showing:

- Global bit-flag in SRAM: 10 code bytes
- Global bit-flag in working register: 4 code bytes
- Global bit-flag in I/O 0x20 – 0x3F: 6 code bytes
- Global bit-flag in I/O 0x00 – 0x1F: 2 code bytes

Action	**Data in SRAM**	**Data in I/O Above 0x1F**	**Data in Register File**	**Data in I/O Below 0x1F**
Set/clear single bit	10	6	4	2
Test single bit	6	4	2	2
Set/clear multiple bits	10	6	4	6
Compare with immediate value	6	4	4	4

Table 5: Code Size (Bytes) for some Common Operations

(The table above is based on "AVR035: Efficient C Coding for 8-bit AVR microcontrollers" (21), p15)

”The examples shows using free I/O locations are very efficient for flag variables that operates on single bits, while using dedicated registers are efficient for frequently accessed variables. Note that locking registers for global variables limits the compilers ability to optimize the code. For complex programs it may increase the code size when dedicating registers for global variables.”

The document proceeds with a comparison of bit-mask vs. bit-field for use with flags. Following the above, using a working register (e.g. a local variable) is most efficient but can only be used with bit-mask. “Below I/O” is roughly equally efficient and works with both bit-mask and bit-field, as does the less efficient “above I/O” and SRAM storage.

Global variables are initialized to zero unless a value is specified. For code density reasons, this is recommended compared to using a separate init routine.

Here I omit the parts about parameter passing and function return as it differs from avr-gcc. The differences can be seen in (131) and (41). I’ll return to the latter when I present the Atmel Studio IDE conventions.

The document ends with an IAR Systems-centered summary that’s better quoted in full:

“Eighteen Hints to Reduce Code Size

1. Compile with full size optimization.
2. Use local variables whenever possible.
3. Use the smallest applicable data type. Use unsigned if applicable.
4. If a non-local variable is only referenced within one function, it should be declared static.
5. Collect non-local data in structures whenever natural. This increases the possibility of indirect addressing without pointer reload.
6. Use pointers with offset or declare structures to access memory mapped I/O.
7. Use for(;;) { } for eternal loops.
8. Use do { } while(expression) if applicable.
9. Use descending loop counters and pre-decrement if applicable.
10. Access I/O memory directly (i.e., do not use pointers).
11. Declare main as C_task if not called from anywhere in the program.
12. Use macros instead of functions for tasks that generates less than 2-3 lines assembly code.
13. Reduce the size of the Interrupt Vector segment (INTVEC) to what is actually needed by the application. Alternatively, concatenate all the CODE segments into one declaration and it will be done automatically.
14. Code reuse is intra-modular. Collect several functions in one module (i.e., in one file) to increase code reuse factor.
15. In some cases, full speed optimization results in lower code size than full size optimization. Compile on a module by module basis to investigate what gives the best result.
16. Optimize C_startup to not initialize unused segments (i.e., IDATA0 or IDATA1 if all variables are tiny or small).
17. If possible, avoid calling functions from inside the interrupt routine [as this causes all registers to be placed on the stack].
18. Use the smallest possible memory model.

Five Hints to Reduce RAM Requirements

1. All constants and literals should be placed in Flash by using the Flash keyword.
2. Avoid using global variables if the variables are local in nature. This also saves code space. Local variables are allocated from the stack dynamically and are removed when the function goes out of scope.
3. If using large functions with variables with a limited lifetime within the function, the use of subscopes can be beneficial.
4. Get good estimates of the sizes of the software Stack and return Stack (Linker File).
5. Do not waste space for the IDATA0 and UDATA0 segments unless you are using tiny variables (Linker File).

Checklist for Debugging Programs

1. Ensure that the CSTACK segment is sufficiently large.
2. Ensure that the RSTACK segment is sufficiently large.
3. Ensure that the external memory interface is enabled if it should be enabled and disabled if it should be disabled.
4. If a regular function and an interrupt routine are communicating through a global variable, make sure this variable is declared volatile to ensure that it is reread from RAM each time it is checked."

A.4.2 Tips and Tricks to Optimize Your C Code for 8-bit AVR Microcontrollers

This document from 2011 is targeted at avr-gcc and it contains some general information about compiler (optimization) settings and the tool-chain. It is similar to the IAR Systems document:

- Use as small variables as possible, both for size and execution performance.
- Use locals instead of globals.
- Write loops to count down to zero, with pre-decrement as this sets the SREG Z flag, which means that a separate comparison isn't needed. This reduces both size and clock cycles.
- Use loop jamming (combining different loops).
- Programs often run out of SRAM before running out of flash. Therefore, store constants in flash with help from the PROGMEM AVR-Libc macro. Reading from flash is slower than SRAM, so if necessary, use a temporary (typically local) variable for it where it is needed.
- Global static variables can only be accessed in the file where they are defined. This is a way to prevent unplanned use. The same applies for static functions.
- Static functions that are only called from one place get optimized automatically as inline code (unless optimization level O0 is used).
- Static local variables preserve their value between function calls, but are only in scope inside their function.
- Use macros instead of functions that generate less than 2-3 lines of assembly code.
- Unroll short loops to avoid testing of loop index and branching. This can increase codesize but reduce the number of clock cycles.
- Always put the most probable outcome first in "if-else" statements.
- "Switch-case" statements are often compiled to a lookup table with indexed jumps, so there is less of a need to organize the outcomes. This might result in quicker but bigger code.

The document ends with an optimization example on a test program compiled with the -s option enabled. I include it here for plausibility comparison to my code samples:

Test Items	**Before optimization**	**After optimization**	**Test result**
Code size	1444 bytes	630 bytes	-56.5%
Data size	25 bytes	0 bytes	-100.0%
Execution speed (1 loop incl 5 ADC samples and 1 USART transmission)	3.88 ms	2.6 ms	-33.0%

Table 6: Example application speed and size optimization result

A.4.3 Some clock cycle counts for ATmega and ATxmega

- add 1
- adiw 2
- brcc 1 if false, 2 if true
- breq 1 if false, 2 if true
- brne 1 if false, 2 if true
- call ATmega 16bit PC 4, 22bit PC 5, ATxmega 16bit PC 3, 22bit PC 4
- cp 1
- cpi 1
- dec 1
- inc 1
- ld X 1
- ld X+ ATmega 2, ATxmega 1
- ldd with displacement from SRAM ATxmega 3
- ldi 1
- lds 2
- lsr 1
- pop 2
- push ATmega 2, ATxmega 1
- rcall ATmega 16bit PC 3, 22bit PC 4, ATxmega 16bit PC 2, 22bit PC 3
- ret 16bit PC 4, 22bit PC 5
- rjmp 2
- sbiw 2
- sbrs 1 if false, 2 if true and skipped instruction is 1 word, 3 if true and skipped instruction is 2 words
- sts 2
- st X ATmega 2, ATxmega 1
- st -Y 2

A.5 IDE-specific additional information

A.5.1 BASCOM-AVR register conventions

When writing inline assembly for BASCOM-AVR, what must you consider in order to avoid overwriting register data? The following information can be found in ”Mixing ASM and BASIC” in the online help: (30) [126]

- Y is used as the soft stack pointer.
- R4 and R5 are used to point to the stack frame or the temp data storage
- R6 is used to store some bit variables:
    - R6 bit 0 = flag for integer/word conversion
    - R6 bit 1 = temp bit space used for swapping bits
    - R6 bit 2 = error bit (ERR variable)
    - R6 bit 3 = show/noshow flag when using INPUT statement
- R8 and R9 are used as a data pointer for the READ statement.
- All other registers are used depending on the used statements.

One of the good things about the non-optimizing compiler is that the built-in commands are separate entities. You only have to make sure that you don’t disturb these registers.

I posted the following question in the Bascom user forum: (132) [127]

PostPosted: Wed Jul 10, 2013 7:36 pm   Post subject: Inline assembler register conventions   Reply with quote

Hi,

In the "Mixing ASM and BASIC" help file, I have found this info:
Y is used as the SW stack pointer.
R4 and R5 are used to point to the stack frame or the temp data storage
R6 is used to store some bit variables:
R6 bit 0 = flag for integer/word conversion
R6 bit 1 = temp bit space used for swapping bits
R6 bit 2 = error bit (ERR variable)
R6 bit 3 = show/noshow flag when using INPUT statement
R8 and R9 are used as a data pointer for the READ statement.
All other registers are used depending on the used statements.

If I want to write an inline assembly using for example R24, X, R10, and R11, can I safely assume that these registers will not have to be pushed and popped on HW stack? At least my disassembly seems to suggest that this is the case in 2.0.7.6, but what is certain?

Is there a more complete listing of Bascom's conventions?

Can I generally assume that the "other registers" are typically used inside Bascom commands?

Grateful for any input. Thanks in advance.Niclas

Posted: Wed Jul 10, 2013 9:13 pm   Post subject: Reply with quote

[126] http://avrhelp.mcselec.com/index.html

[127] http://www.mcselec.com/index2.php?option=com_forum&Itemid=59&page=viewtopic&p=60805#60805

ASM-Code between Bascom statements can use any registers freely without need of saving, beside those mentioned, of course. It's different in an ISR, there you don't know which instruction was interrupted, so every register used by the ISR needs to be saved.

MWS"

A.5.2 AVR-GCC register layout, frame layout, and calling convention

The below is taken directly from the AVR-GCC Wiki: (33) [128]

"Values that occupy more than one 8-bit register start in an even register.

Fixed Registers

Fixed Registers are registers that won't be allocated by GCC's register allocator. Registers R0 and R1 are fixed and used implicitly while printing out assembler instructions:

- R0: is used as scratch register that need not to be restored after its usage. It must be saved and restored in interrupt service routine's (ISR) prologue and epilogue. In inline assembler you can use `__tmp_reg__` for the scratch register.
- R1: always contains zero. During an insn the content might be destroyed, e.g. by a MUL instruction that uses R0/R1 as implicit output register. If an insn destroys R1, the insn must restore R1 to zero afterwards. This register must be saved in ISR prologues and must then be set to zero because R1 might contain values other than zero. The ISR epilogue restores the value. In inline assembler you can use `__zero_reg__` for the zero register.
- T: the T flag in the status register (SREG) is used in the same way like the temporary scratch register R0.

User-defined global registers by means of global `register asm` and / or `-ffixed-`*n* won't be saved or restored in function pro- and epilogue.

[128] http://gcc.gnu.org/wiki/avr-gcc

Call-Used Registers

The *call-used* or *call-clobbered* general purpose registers (GPRs) are registers that might be destroyed (clobbered) by a function call.

- R18–R27, R30, R31: These GPRs are call clobbered. An ordinary function may use them without restoring the contents. Interrupt service routines (ISRs) must save and restore each register they use.
- R0, T-Flag: The temporary register and the T-flag in SREG are also call-clobbered, but this knowledge is not exposed explicitly to the compiler (R0 is a fixed register).

Call-Saved Registers

- R2–R17, R28, R29: The remaining GPRs are call-saved, i.e. a function that uses such a registers must restore its original content. This applies even if the register is used to pass a function argument.
- R1: The zero-register is implicity call-saved (implicit because R1 is a fixed register).

Frame Layout

During compilation the compiler may come up with an arbitrary number of *pseudo registers* which will be allocated to *hard registers* during register allocation.

- Pseudos that don't get a hard register will be put into a stack slot and loaded / stored as needed.
- In order to access stack locations, avr-gcc will set up a 16-bit frame pointer in R29:R28 (Y) because the stack pointer (SP) cannot be used to access stack slots.
- The stack grows downwards. Smaller addresses are at the bottom of the drawing at the right.
- Stack pointer and frame pointer are not aligned, i.e. 1-byte aligned.
- After the function prologue, the frame pointer will point one byte below the stack frame, i.e. Y+1 points to the bottom of the stack frame.
- Any of "incoming arguments", "saved registers" or "stack slots" in the drawing at the right may be empty.
- Even "return address" may be empty which happens for functions that are tail-called.

incoming arguments
return address (2-3 bytes) saved registers
stack slots, Y+1 points at the bottom

Table 7: Frame layout after Function Prologue (reproduction of image in the Wiki)

Calling Convention

- An argument is passed either completely in registers or completely in memory.
- To find the register where a function argument is passed, initialize the register number R_n with R26 and follow this procedure:
    1. If the argument size is an odd number of bytes, round up the size to the next even number.
    2. Subtract the rounded size from the register number R_n.
    3. If the new R_n is at least R8 and the size of the object is non-zero, then the low-byte of the argument is passed in R_n. Subsequent bytes of the argument are passed in the subsequent registers, i.e. in increasing register numbers.
    4. If the new register number R_n is smaller than R8 or the size of the argument is zero, the argument will be passed in memory.
    5. If the current argument is passed in memory, stop the procedure: All subsequent arguments will also be passed in memory.
    6. If there are arguments left, goto 1. and proceed with the next argument.
- Return values with a size of 1 byte up to and including a size of 8 bytes will be returned in registers. Return values whose size is outside that range will be returned in memory.
- If a return value cannot be returned in registers, the caller will allocate stack space and pass the address as implicit first pointer argument to the callee. The callee will put the return value into the space provided by the caller.
- If the return value of a function is returned in registers, the same registers are used as if the value was the first parameter of a non-varargs function. For example, an 8-bit value is returned in R24 and an 32-bit value is returned R22...R25.
- Arguments of varargs functions are passed on the stack. This applies even to the named arguments.

For example, suppose a function with the following prototype:

```
int func (char a, long b);
```

then

- a will be passed in R24.
- b will be passed in R20, R21, R22 and R23 with the LSB in R20 and the MSB in R23.
- The result is returned in R24 (LSB) and R25 (MSB).“

A.5.3 Atmel Studio 6 history: AVR Studio 4 & 5, WinAVR, and Eclipse

Versions prior to 4 seem to be antiquated, but for some reasons some users still prefer AVR Studio 4 to Atmel Studio 6. (77) [129] This page claims that:

"In conclusion, AVRStudio 5 is rubbish and should be avoided, AVRStudio 6 is great if you have a very new PC with lots of resources and AVRStudio 4 is still a very good program and perfectly suited to developing AVR projects in C or assembler, especially AVRStudio 4.18, SP3. It would be easier for many users if Atmel could be bothered to fix v4.19 to eliminate the tool chain bug."

It also says:

"What about tool support in different versions? Most tools, including Kanda AVRISP programmers, AVRISP mkII programmer, AVRDragon and JTAGICE mkII programmer and emulators will work in all versions of AVRStudio. But the lowest cost emulator JTAGICE is not supported in version 5 or 6, so you need AVRStudio 4 to take advantage of this low cost unit."

The general impression I've got is that AVR Studio 5 had lots of teething problems and possibly that it was released as a beta too early. E.g. it seems that C++ and one of the most popular Atmel development boards (STK500) wasn't supported at that point. (78) [130] Also the final release seems to have caused strong emotional outbursts: (133) [131] I was only using AVR Studio 4 to burn my BASCOM-AVR .hex files so I don't have any first-hand experience.

Parallel to this, I think that many people used WinAVR, a Windows application for the AVR-GCC toolchain. (134) [132] According to this forum thread, it is possible to use Eclipse too, on both Windows and Linux: (135) [133] It also claims that *"Atmel hired the head WinAVR developer to work on toolchains for them."*

(In addition to these free IDEs, there are also at least two commercial development platforms that should be mentioned: IAR Embedded Workbench (136) [134] and CodeVisionAVR (137) [135].)

[129] http://www.kanda.com/blog/microcontrollers/avr-microcontrollers/avrstudio-explored/
[130] http://www.avrfreaks.net/index.php?name=PNphpBB2&file=printview&t=103949
[131] http://www.avrfreaks.net/index.php?name=PNphpBB2&file=printview&t=112779&start=0
[132] http://winavr.sourceforge.net/
[133] http://www.motherboardpoint.com/which-avr-studio-and-c-compiler-avr-8-bit-microcontroller-and-jtagice-t257051.html
[134] http://www.iar.com/Products/IAR-Embedded-Workbench/
[135] http://www.codevision.be/codevisionavr

A.6 BASCOM incremental code pieces

A.6.1 ATxmega USART setup

Here’s the actual ATxmega BASCOM-AVR v2.0.7.6 compiled code for USART setup:

```
 Config Com5 = 15625 , Mode = 0 , Parity = None , Stopbits = 1 , Databits = 8
00000134    78.e0         LDI R23,0x08       Load immediate                       PORTE_DIRSET = 0b0000 1000 (bit 3 output)
00000135    70.93.81.06   STS 0x0681,R23     Store direct to data space                      ...

00000137    8f.e9         LDI R24,0x9F       Load immediate                       Place 0x9F on stack
00000138    8a.93          ST -Y,R24         Store indirect and predecrement       ...

00000139    80.e8         LDI R24,0x80       Load immediate                       Place 0x80 on stack
0000013A    8a.93         ST -Y,R24          Store indirect and predecrement

0000013B    83.e0         LDI R24,0x03       Load immediate                       Place 0x03 on stack
0000013C    8a.93         ST -Y,R24          Store indirect and predecrement

0000013D    84.e0         LDI R24,0x04       Load immediate                       Place 0x04 on stack (USART number)
0000013E    8a.93         ST -Y,R24          Store indirect and predecrement
0000013F    0e.94.1e.02   CALL 0x0000021E    Call subroutine      Calculate USART register area start and write them from stack

--USART_c: (Point X to USART data register)
00000216    79.91         LD R23,Y+          Load indirect and postincrement      R23 = USART number from SW stack
00000217    a0.ea         LDI R26,0xA0       Load immediate                       R26 = 0xA0

00000218    76.95         LSR R23            Logical shift right                  Logical shift R23 right bit 0 to C (in this case 4 -> 2)
00000219    08.f4         BRCC PC+0x02       Branch if carry cleared    0x21B if USART number was even X = 0x08 A0 (USARTC0_DATA)
0000021A    a0.eb         LDI R26,0xB0       Load immediate             If USART number was odd, X = 0x08 B0 (USARTC1_DATA)
0000021B    b8.e0         LDI R27,0x08       Load immediate  ...
0000021C    b7.0f         ADD R27,R23        Add without carry          Adjust XH by USART offset to 0x0A A0 (USARTE0_DATA)
0000021D    08.95         RET                Subroutine return

0000021E    f7.df         RCALL PC-0x0008    Relative call subroutine             Point X to USART data register
0000021F    14.96         ADIW R26,0x04      Add immediate to word                Point X to USART ctrlb register

00000220    88.e1         LDI R24,0x18       Load immediate                       USARTxy_CTRLB = 0x18 (Enable RX and TX)
00000221    8d.93         ST X+,R24          Store indirect and postincrement     ...

00000222    89.91          LD R24,Y+         Load indirect and postincrement  USARTxy_CTRLC = 0x03 (Asynch, no par, 1 stop bit, 8-bit)
00000223    8d.93         ST X+,R24          Store indirect and postincrement     ...

00000224    89.91         LD R24,Y+          Load indirect and postincrement  USARTxy_BAUDCTRLA = 0x80 (BSEL = 0xF 80)
00000225    8d.93         ST X+,R24          Store indirect and postincrement

00000226    89.91         LD R24,Y+          Load indirect and postincrement  USARTxy_BAUDCTRLB = 0x9F (Baud rate scale factor 9)
00000227    8d.93         ST X+,R24          Store indirect and postincrement
00000228    08.95         RET                Subroutine return
```

A.6.2 ATxmega set up a second USART identical to the first one

```
Config Com6 = 15625 , Mode = 0 , Parity = None , Stopbits = 1 , Databits = 8
            1 word        LDI R23,0x80       Load immediate                       PORTE_DIRSET = 0b1000 0000 (bit 7 output)
00000135    70.93.81.06   STS 0x0681,R23     Store direct to data space                      ...

             1 word       SBIW YH:YL,3       Subtract 3 from the Y-pointer(r29:r28)

0000013D    1 word        LDI R24,0x05       Load immediate                       Place 0x05 on stack (USART number)
0000013E    8a.93         ST -Y,R24          Store indirect and predecrement
0000013F    0e.94.1e.02   0x0000021E         Call subroutine        Calculate USART register area start and write them from stack
```

A.6.3 ATxmega custom USART setup

Custom USART setup with exactly the same functionality as 6.1:

```
  Config Com5 = 15625 , Mode = 0 , Parity = None , Stopbits = 1 , Databits = 8
78.e0          LDI R23,0x08                  Load immediate                    PORTE_DIRSET = 0b0000 1000 (bit 3 output)
70.93.81.06  STS 0x0681,R23                  Store direct to data space                  ...

             LDI R26,0xA4                    Load immediate                    Point X to USARTE0_CTRLB
             LDI R27,0x0A                    Load immediate                    ...

88.e1        LDI R24,0x18                    Load immediate                    USARTE0_CTRLB = 0x18 (Enable RX and TX)
8d.93        ST X+,R24                       Store indirect and postincrement  ...

             LDI R24,0x03                    Load immediate  USARTE0_CTRLC = 0x03 (Asynch, no par, 1 stop bit, 8-bit)
8d.93        ST X+,R24                       Store indirect and postincrement  ...

             LDI R24,80                      Load immediate  USARTE0_BAUDCTRLA = 0x80 (BSEL = 0xF 80)
8d.93        ST X+,R24                       Store indirect and postincrement  ...

             LDI R24,9F                      Load immediate  USARTE0_BAUDCTRLB = 0x9F (Baud rate scale factor 1001 == -7)
             ST X,R24                        Store indirect                    ...
```

A.6.4 ATxmega USART setup scaling example

```
  Config Com6 = 15625 , Mode = 0 , Parity = None , Stopbits = 1 , Databits = 8
78.e0         LDI R23,0x08                   Load immediate                    PORTE_DIRSET = 0b0000 1000 (bit 3 output)
70.93.81.06  STS 0x0681,R23                  Store direct to data space                  ...
             LDI R26,0xA4                    Load immediate                    Point X to USARTE0_CTRLB
             LDI R27,0x0A                    Load immediate                    ...
2 words      CALL __USART_write_settings

             LDI R23,0x80                    Load immediate                    PORTF_DIRSET = 0b1000 0000 (bit 7 output)
70.93.81.06  0x0681,R23      Store direct to data space                        ...
             LDI R26,0xB4                    Load immediate                    Point X to USARTE1_CTRLB
             LDI R27,0x0A                    Load immediate                    ...
2 words      CALL __USART_write_settings

__USART_write_settings:
88.e1        LDI R24,0x18                    Load immediate  USARTE1_CTRLB = 0x18 (Enable RX and TX)
8d.93        ST X+,R24                       Store indirect and postincrement  ...

             LDI R24,0x03                    Load immediate  USARTE1_CTRLC = 0x03 (Asynch, no par, 1 stop bit, 8-bit)
8d.93        ST X+,R24                       Store indirect and postincrement  ...

             LDI R24,80                      Load immediate  USARTE1_BAUDCTRLA = 0x80 (BSEL = 0xF 80)
8d.93        ST X+,R24                       Store indirect and postincrement  ...

             LDI R24,9F                      Load immediate  USARTE1_BAUDCTRLB = 0x9F (Baud rate scale factor 9)
             ST X,R24                        Store indirect                              ...
08.95        RET                             Subroutine return
```

A.6.5 USART sending one port, ATmega original disassembly

```
Printbin #1 , 254
 19e:    8e ef          ldi      r24, 0xFE     ; 254              R24 = 254
 1a0:    0e 94 2c 01    call     0x258         ;  0x258           Call Send USART byte

Uartsendbyte = 1, Gosub Prbin
 1aa:    81 e0          ldi      r24, 0x01     ; 1                Uartsendbyte = 1
 1ac:    80 93 2d 01    sts      0x012D, r24                      ...
 1b0:    0e 94 f2 00    call     0x1e4         ;  0x1e4           Call Prbin gosub

Prbin gosub:
 1e4:    31 e0          ldi      r19, 0x01     ; 1                R19 = 1 (number of bytes to send)
 1e6:    ad e2          ldi      r26, 0x2D     ; 45               X points to global Uartsendbyte
 1e8:    b1 e0          ldi      r27, 0x01     ; 1                ...
 1ea:    0e 94 27 01    call     0x24e         ;  0x24e           Call Printbin command
 1ee:    08 95          ret

Printbin command:
 24e:       8d 91           ld          r24, X+                              R24 = value of X (Global Uartsendbyte)
 250:       03 d0           rcall       .+6             ;  0x258             rcall Send USART byte
 252:       3a 95           dec         r19                                  Repeat until there are no more bytes to send
 254:       e1 f7           brne        .-8             ;  0x24e             ...
 256:       08 95           ret

Send USART byte
 258:       00 90 c0 00     lds         r0, 0x00C0                           R0 = UCSR0A
 25c:       05 fe           sbrs        r0, 5                                Skip next op if bit 5 is set (UDREn: USART ready to receive
new data)
 25e:       fc cf           rjmp        .-8             ;  0x258             Repeat until register ready
 260:       80 93 c6 00     sts         0x00C6, r24                          UDR0 = R24
 264:       08 95           ret
```

(Disassembled with AVR-objdump.)

A.6.6 USART sending one port, ATxmega original disassembly

```
  Printbin #1 , 254
1B1:        8e.ef           LDI R24,0xFE            Load immediate                  R24 = byte to send
1B2:        74.e0           LDI R23,0x04            Load immediate                  Place USART number on SW stack Y
1B3:        7a.93           ST -Y,R23               Store indirect and predecrement ...
1B4:        0e.94.5d.02     CALL 0x25D  Call subroutine          --USART_b2

  Uartsendbyte = 1,    Gosub Prbin
1BB:        81.e0           LDI R24,0x01            Load immediate                  Uartsendbyte = 1
1BC:        80.93.56.20     STS 0x2056,R24          Store direct to data space      ...
1BE:        0e.94.e0.01      CALL 0x1E0  Call subroutine         --USART_b (Gosub Prbin)

Prbin gosub:  Printbin #1 , Uartsendbyte
1E0:        31.e0           LDI R19,0x01            Load immediate                  R19 = number of bytes to send
1E1:        a6.e5           LDI R26,0x56            Load immediate                  X = Uartsendbyte
1E2:        b0.e2           LDI R27,0x20            Load immediate                  ...
1E3:        74.e0           LDI R23,0x04            Load immediate                  R23 = USART number 4 (place on SW
stack Y)
1E4:        7a.93           ST -Y,R23               Store indirect and predecrement ...
1E5:        0e.94.2f.02     CALL 0x22F              Call subroutine                 --USART_a: (Printbin command)
1E7:        08.95           RET                     Subroutine return

--USART_c: (Point X to USART data register)
216:        79.91           LD R23,Y+               Load indirect and postincrement R23 = USART number from SW stack
217:        a0.ea           LDI R26,0xA0            Load immediate                  R26 = 0xA0
218:        76.95           LSR R23                 Logical shift right             Logical shift R23 right bit 0 to C (in this
case 4 -> 2)
219:        08.f4           BRCC PC+0x02            Branch if carry cleared   0x21B if USART number was even X = 0x08 A0
(USARTC0_DATA)
21A:        a0.eb           LDI R26,0xB0            Load immediate            If USART number was odd, X = 0x08 B0 (USARTC1_DATA)
21B:        b8.e0           LDI R27,0x08            Load immediate                  ...
21C:        b7.0f           ADD R27,R23             Add without carry         Adjust XH by USART offset to 0x0A B0 (USARTE1_DATA)
21D:        08.95           RET                     Subroutine return

Printbin command
--USART_a:
22F:        8d.91           LD R24,X+               Load indirect and postincrement R24 = Uartsendbyte from X, post-
increment
230:        2a.d0           RCALL PC+0x002B         Relative call subroutine        --USART_b
231:        3a.95           DEC R19                 Decrement                                    More bytes to
send?
232:        e1.f7           BRNE PC-0x03            Branch if not equal             Yes, --USART_a
233:        21.96           ADIW R28,0x01           Add immediate to word           No, Y++ (SW stack pointer)
234:        08.95           RET                     Subroutine return

--USART_b:
25B:        78.81           LDD R23,Y+0             Load indirect with displacement R23 = USART number from SW stack Y
25C:        7a.93           ST -Y,R23               Store indirect and predecrement Place copy of USART number on SW
stack Y

--USART_b2:
25D:        bf.93           PUSH R27                Push register on stack    Place X on stack (would be next byte if array)
25E:        af.93           PUSH R26                Push register on stack          ...
25F:        b6.df           RCALL PC-0x0049         Relative call subroutine  --USART_c (Point X to USART data register)
```

```
260:      11.96        ADIW R26,0x01         Add immediate to word             Point X to USART status register
261:      7c.91        LD R23,X              Load indirect        Check status register bit 5 until data reg ready
262:      75.ff        SBRS R23,5            Skip if bit in register set                 ...
263:      fd.cf        RJMP PC-0x0002        Relative jump                               ...
264:      11.97        SBIW R26,0x01         Subtract immediate from word      Point X back to USART data register
265:      8c.93        ST X,R24              Store indirect       Write Uartsendbyte to USART data register
266:      af.91        POP R26               Pop register from stack   Retrieve the X pointer to the next array byte to send
267:      bf.91        POP R27               Pop register from stack                     ...
268:      08.95        RET                   Subroutine return
```

(Disassembled with Atmel Studio 6.1.)

A.6.7 Modified ATmega for two serial ports

```
Printbin #1 , 254
19e:      8e ef        ldi       r24, 0xFE      ; 254          R24 = 254
                       ldi       r23, 0                        Send through channel 0
1a0:      0e 94 2c 01  call      0x258          ;  0x258       Call Send USART byte

Uartsendbyte = 1, Gosub Prbin
1aa:      81 e0        ldi       r24, 0x01      ; 1            Uartsendbyte = 1
1ac:      80 93 2d 01  sts       0x012D, r24                   ...
                       ldi       r23, 0                        Send through channel 0
1b0:      0e 94 f2 00  call      0x1e4          ;  0x1e4       Call Prbin gosub

Printbin #2 , 254
19e:      8e ef        ldi       r24, 0xFE      ; 254          R24 = 254
                       ldi       r23, 1                        Send through channel 1
1a0:      0e 94 2c 01  call      0x258          ;  0x258       Call Send USART byte

Uartsendbyte = 1, Gosub Prbin
1aa:      81 e0        ldi       r24, 0x01      ; 1            Uartsendbyte = 1
1ac:      80 93 2d 01  sts       0x012D, r24                   ...
                       ldi       r23, 1                        Send through channel 1
1b0:      0e 94 f2 00  call      0x1e4          ;  0x1e4       Call Prbin gosub

Prbin gosub:
1e4:      31 e0        ldi       r19, 0x01      ; 1            R19 = 1 (number of bytes to send)
1e6:      ad e2        ldi       r26, 0x2D      ; 45           X points to global Uartsendbyte
1e8:      b1 e0        ldi       r27, 0x01      ; 1            ...
1ea:      0e 94 27 01  call      0x24e          ;  0x24e       Call Printbin command
1ee:      08 95        ret

Printbin command:
 24e:     8d 91        ld        r24, X+                       R24 = value of X (Global Uartsendbyte)
 250:     03 d0        rcall     .+6            ;  0x258       rcall Send USART byte
 252:     3a 95        dec       r19                           Repeat until there are no more bytes to send
 254:     e1 f7        brne      .-8            ;  0x24e       ...
 256:     08 95        ret

Send USART byte
258:                   cpi       r23, 0
                       brne      USART1_send
__USART0_send:
          00 90 c0 00  lds       r0, 0x00C0                    R0 = UCSR0A
          05 fe        sbrs      r0, 5          Skip next op if bit 5 is set (UDREn: USART ready to receive new data)
          fc cf        rjmp      __USART0_send                 Repeat until register ready
          80 93 c6 00  sts       0x00C6, r24                   UDR0 = R24
          08 95        ret
__USART1_send:
          00 90 c0 00  lds       r0, UCSR1A                    R0 = UCSR1A
          05 fe        sbrs      r0, 5          Skip next op if bit 5 is set (UDREn: USART ready to receive new data)
          fc cf        rjmp      __USART1_send                 Repeat until register ready
          80 93 c6 00  sts       UDR1, r24                     UDR1 = R24
                       ret
```

A.6.8 Modified ATxmega for two serial ports

```
  Printbin #1 , 254
1B1:    8e.ef         LDI R24,0xFE          Load immediate                   R24 = byte to send (254)
1B2:    74.e0         LDI R23,0x04          Load immediate                   Place USART number on SW stack Y
                      CALL --USART_b1       Call subroutine                  --USART_b1

  Uartsendbyte = 1,  Gosub Prbin
1BB:    81.e0         LDI R24,0x01          Load immediate                   Uartsendbyte = 1
1BC:    80.93.56.20   STS 0x2056,R24        Store direct to data space       ...
1E3:    74.e0         LDI R23,0x04          Load immediate                   R23 = USART number 4 (place on SW
stack Y)
1BE:    0e.94.e0.01   CALL 0x1E0            Call subroutine                  --USART_b (Gosub Prbin)

Printbin #2, 254
1B1:    8e.ef         LDI R24,0xFE          Load immediate                   R24 = byte to send (254)
1B2:    74.e0         LDI R23,0x05          Load immediate                   Place USART number 5 on SW stack Y
                      CALL --USART_b1       Call subroutine                  --USART_b1

  Uartsendbyte = 1,  Gosub Prbin
1BB:    81.e0         LDI R24,0x01          Load immediate                   Uartsendbyte = 1
1BC:    80.93.56.20   STS 0x2056,R24        Store direct to data space       ...
                      LDI R23,0x05          Load immediate                   R23 = USART number 5 (place on SW
stack Y)
1BE:    0e.94.e0.01   CALL 0x1E0            Call subroutine                  --USART_b (Gosub Prbin)

Prbin gosub:
  Printbin #1 , Uartsendbyte
1E0:    31.e0         LDI R19,0x01          Load immediate                   R19 = number of bytes to send
1E1:    a6.e5         LDI R26,0x56          Load immediate                   X = Uartsendbyte
1E2:    b0.e2         LDI R27,0x20          Load immediate                   ...
1E4:    7a.93         ST -Y,R23             Store indirect and predecrement  ...
1E5:    0e.94.2f.02   CALL 0x22F            Call subroutine                  --USART_a: (Printbin command)
1E7:    08.95         RET                   Subroutine return

--USART_c: (Point X to USART data register)
216:    79.91         LD R23,Y+             Load indirect and postincrement  R23 = USART number from SW stack
217:    a0.ea         LDI R26,0xA0          Load immediate                   R26 = 0xA0
218:    76.95         LSR R23               Logical shift right              Logical shift R23 right bit 0 to C (in this
case 4 -> 2)
219:    08.f4         BRCC PC+0x02          Branch if carry cleared 0x21B if USART number was even X = 0x08 A0
                                                                             (USARTC0_DATA)
21A:    a0.eb         LDI R26,0xB0          Load immediate                   If USART number was odd, X = 0x08 B0
                                                                             (USARTC1_DATA)
21B:    b8.e0         LDI R27,0x08          Load immediate                   ...
21C:    b7.0f         ADD R27,R23           Add without carry                Adjust XH by USART offset to 0x0A B0
                                                                             (USARTE1_DATA)
21D:    08.95         RET                   Subroutine return

Printbin command
--USART_a:
22F:    8d.91         LD R24,X+             Load indirect and postincrement    R24 = Uartsendbyte from X, post-increment
230:    2a.d0         RCALL PC+0x002B       Relative call subroutine         --USART_b
231:    3a.95         DEC R19               Decrement                        More bytes to send?
232:    e1.f7         BRNE PC-0x03          Branch if not equal              Yes, --USART_a
233:    21.96         ADIW R28,0x01         Add immediate to word            No, Y++ (SW stack pointer)
234:    08.95         RET                   Subroutine return

--USART_b:
25B:    78.81         LDD R23,Y+0           Load indirect with displacement  R23 = USART number from SW stack Y

--USART_b1:
25C:    7a.93         ST -Y,R23             Store indirect and predecrement    Place copy of USART number on SW stack Y

--USART_b2:
25D:    bf.93         PUSH R27              Push register on stack         Place X on stack (would be next byte if array)
25E:    af.93         PUSH R26              Push register on stack           ...
25F:    b6.df         RCALL PC-0x0049       Relative call subroutine       __USART_c (Point X to USART data register)
260:    11.96         ADIW R26,0x01         Add immediate to word            Point X to USART status register
261:    7c.91         LD R23,X              Load indirect                  Check status register bit 5 until data reg ready
262:    75.ff         SBRS R23,5            Skip if bit in register set      ...
263:    fd.cf         RJMP PC-0x0002        Relative jump                               ...
```

```
264:      11.97        SBIW R26,0x01          Subtract immediate from word          Point X back to USART data register
265:      8c.93        ST X,R24               Store indirect                 Write Uartsendbyte to USART data register
266:      af.91        POP R26                Pop register from stack       Retrieve the X pointer to the next array byte to send
267:      bf.91        POP R27                Pop register from stack                   ...
268:      08.95        RET                    Subroutine return
```

A.6.9 Modified ATmega for three serial ports

```
Printbin #1 , 254
 19e:     8e ef        ldi       r24, 0xFE    ; 254               R24 = 254
                       ldi       r23, 0                           Send through channel 0
 1a0:     0e 94 2c 01  call      0x258        ;  0x258            Call Send USART byte

Uartsendbyte = 1
Gosub Prbin
 1aa:     81 e0        ldi       r24, 0x01    ; 1                 Uartsendbyte = 1
 1ac:     80 93 2d 01  sts       0x012D, r24                      ...
ldi       r23, 0                              Send through channel 0
 1b0:     0e 94 f2 00  call      0x1e4        ;  0x1e4            Call Prbin gosub

+++ two more like the above, each 10 words

Prbin gosub:
 1e4:     31 e0        ldi       r19, 0x01    ; 1                 R19 = 1 (number of bytes to send)
 1e6:     ad e2        ldi       r26, 0x2D    ; 45                X points to global Uartsendbyte
 1e8:     b1 e0        ldi       r27, 0x01    ; 1                 ...
 1ea:     0e 94 27 01  call      0x24e        ;  0x24e            Call Printbin command
 1ee:     08 95        ret

Printbin command:
 24e:     8d 91        ld        r24, X+                          R24 = value of X (Global Uartsendbyte)
 250:     03 d0        rcall     .+6          ;  0x258            rcall Send USART byte
 252:     3a 95        dec       r19                              Repeat until there are no more bytes to send
 254:     e1 f7        brne      .-8          ;  0x24e            ...
 256:     08 95        ret

Send USART byte
258:                   cpi       r23, 0
                       brne      USART1_send
__USART0_send:
          00 90 c0 00  lds       r0, 0x00C0                       R0 = UCSR0A
          05 fe        sbrs      r0, 5             Skip next op if bit 5 is set (UDREn: USART ready to receive new data)
          fc cf        rjmp      .-8                              Repeat until register ready
          80 93 c6 00  sts       0x00C6, r24                      UDR0 = R24
          08 95        ret

__USART1_send:
                       dec       r23
                       brne      USART2_send

          00 90 c0 00  lds       r0, UCSR1A                       R0 = UCSR1A
          05 fe        sbrs      r0, 5             Skip next op if bit 5 is set (UDREn: USART ready to receive new data)
          fc cf        rjmp      .-8                              Repeat until register ready
          80 93 c6 00  sts       UDR1, r24                        UDR1 = R24
                       ret

__USART2_send:
          00 90 c0 00  lds       r0, UCSR2A                       R0 = UCSR2A
          05 fe        sbrs      r0, 5             Skip next op if bit 5 is set (UDREn: USART ready to receive new data)
          fc cf        rjmp      .-8                              Repeat until register ready
          80 93 c6 00  sts       UDR2, r24                        UDR2 = R24
                       ret
```

A.6.10 Improved ATxmega dynamic addressing for two serial ports

```
Printbin #1 , 254
        1        ldi      r24, 0xFE    ; 254          R24 = 254
        1        ldi      r23, 4                      Send through channel 4
        2        call     __USART_send_const          Send constant

Uartsendbyte = 1, Gosub Prbin
        1        ldi      r24, 0x01    ; 1            Uartsendbyte = 1
        2        sts      0x012D, r24                 ...
        1        ldi      r23, 4                      Send through channel 4
        2        call     __Prbin_gosub               Call Prbin gosub

Printbin #2 , 254
        1        ldi      r24, 0xFE    ; 254          R24 = 254
        1        ldi      r23, 5                      Send through channel 5
        2        call     __USART_send_const          Send constant

Uartsendbyte = 1, Gosub Prbin
        1        ldi      r24, 0x01    ; 1            Uartsendbyte = 1
        2        sts      0x012D, r24                 ...
        1        ldi      r23, 5                      Send through channel 5
        2        call     __Prbin_gosub               Call Prbin gosub

__Prbin_gosub:                                        (Send 1 byte from global Uartsendbyte)
        1        ldi      r19, 0x01    ; 1            R19 = 1 (number of bytes to send)
        1        ldi      r26, 0x2D    ; 45           X points to global Uartsendbyte
        1        ldi      r27, 0x01    ; 1            ...
                 ; rjmp   __USART_send_varray         Not needed here (but will be for the next Prbin_gosub)
                 ; ret                                Not needed at all if compiler can handle un-ended gosub
                                                      (or inline assembly is used for the __Prbin_gosub)

__USART_send_varray:                                  Launcher for sending variable or array
        1        LDI      r18, 0x00                   Clear skip variable (variable or array sending)
        2        rjmp     __USART_point_Z             Send variable (or array)

                                                      Launcher for sending constant
__USART_send_const: 1  LDI  r18, 0xFF  ; 254          Set skip variable (constant sending)
        1        LDI      r19, 0x01    ; 1            R19 = 1 (number of bytes to send)

__USART_point_Z: (Point Z to USART data register), port number in R23 (R23 changed by routine)
        1        PUSH R30                             Free Z from frame use
        1        PUSH R31                             ...
        1        LDI R30,0xA0          Load immediate          ZL
        1        LSR R23               Logical shift right     Logical shift R23 right bit 0 to C (here 4->2)
        1        BRCC PC+0x02          Branch if carry cleared If USART # was even: Z = 0x08 A0 (USARTC0_DATA)
        1        LDI R30,0xB0          Load immediate          If USART # was odd: Z = 0x08 B0 (USARTC1_DATA)
        1        LDI R31,0x08          Load immediate          ...
        1        ADD R31,R23           Add without carry       Adjust ZH by USART offset up to 0x0B (USARTFx_DATA)

        1        SBRS R18, 7           Skip if R18 bit 7 set   When sending constant, don't change R24

__USART_loop: (Send byte in R24 to USART with base address in Z)
        1        ld       r24, X+      R24 <- X, post-increment  Load byte from X into R24
        1        ldd R4,Z+1            Load indirect w displm  Wait until port available (check status register bit 5)
        1        SBRS R4,5             Skip if bit in register set  ...
        1        RJMP PC-0x0002        Relative jump           … until data reg ready
        1        ST Z,R24              Store indirect          Write R24 to USART data register
        1        dec      r19                                  R19- -
        1        BRNE     __USART_loop                         … until all bytes have been sent
        1        POP R31                                       Restore Z for frame use
        1        POP R30                                       ...
        1        ret
```

A.6.11 ATxmega USART initialization improvements from 2.0.7.6 to 2.0.7.7

```
Config Priority = Static , Vector = Application , Lo = Enabled , Med = Enabled (as before)
258   77 E0            LDI R23,0x07          Load immediate           PMIC_CTRL = 0x07 (Enable high, mid, & low interrupts)
25A   70 93 A2 00      STS 0x00A2,R23        Store direct to data space           ...

Config Com5 = 15625 , Mode = 0 , Parity = None , Stopbits = 1 , Databits = 8 (changed to exactly the code I sent to BASCOM)
25E   78 E0            LDI R23,0x08          Load immediate           PORTE_DIRSET = 0b0000 1000 (bit 3 output)
260   70 93 81 06      STS 0x0681,R23        Store direct to data space...
264   88 E1            LDI R24,0x18          Load immediate           USARTE0_CTRLB = 0x18 (Enable RX and TX)
266   A4 EA            LDI R26,0xA4          Load immediate           Point X to USARTE0_CTRLB
268   BA E0            LDI R27,0x0A          Load immediate           ...
26A   8D 93            ST X+,R24             Store indirect and postincrement
26C   83 E0            LDI R24,0x03          Load immediate           USARTE0_CTRLC = 0x03 (Asynch, no par, 1 stop bit, 8-bit)
26E   8D 93            ST X+,R24             Store indirect and postincrement
270   80 E8            LDI R24,80            Load immediate           USARTE0_BAUDCTRLA = 0x80 (BSEL = 0xF 80)
272   8D 93            ST X+,R24             Store indirect and postincrement
274   8F E9            LDI R24,0x9F          Load immediate           USARTE0_BAUDCTRLB = 0x9F (Baud rate scale factor 9)
276   8C 93            ST X,R24              Store indirect

Config Serialin3 = Buffered , Size = 20 (as before)
278   80 91 B3 09      LDS R24,0x09B3   Load direct from data space R24 = USARTD1_CTRLA (USART D1 RXC interrupt level MED)
27C   8F 7C            ANDI R24,0xCF         Logical AND with immediate           R24 = R24 & 0b1100 1111
27E   80 62            ORI R24,0x20          Logical OR with immediate            R24 = R24 | 0b0010 0000
280   80 93 B3 09      STS 0x09B3,R24        Store direct to data space           USARTD1_CTRLA = R24
```

A.6.12 ATmega324A initialization code

```
 7c:    8f ef          ldi     r24, 0xFF     ; 255     K0xFF -> SPL
 7e:    8d bf          out     0x3d, r24     ; 61      ...
 80:    c8 ed          ldi     r28, 0xD8     ; 216     K0xD8 -> R28 (Y LSB) Software stack start
 82:    e0 ec          ldi     r30, 0xC0     ; 192     K0xC0 -> R30 (Z LSB) Frame start
 84:    4e 2e          mov     r4, r30                 R4, R5 holds the current frame position
 86:    88 e0          ldi     r24, 0x08     ; 8       K0x08 -> R24
 88:    8e bf          out     0x3e, r24     ; 62      SPH stack pointer now points to RAMEND (0x8FF) 0x100 to 0x8FF = 2kB
 8a:    d8 e0          ldi     r29, 0x08     ; 8       K0x08 -> R29 (Y MSB) Software stack start
 8c:    f8 e0          ldi     r31, 0x08     ; 8       K0x08 -> R31 (Z MSB) Frame start
 8e:    5f 2e          mov     r5, r31                 R4, R5 holds the current frame position
 90:    a8 95          wdr                             Watchdog reset
 92:    84 b7          in      r24, 0x34     ; 52      Read MCUSR
 94:    08 2e          mov     r0, r24                 Keep MCUSR in R0 ???
 96:    87 7f          andi    r24, 0xF7     ; 247     Mask MCUSR 0b1111 0111 (clear any reset flag except WDRF)
 98:    84 bf          out     0x34, r24     ; 52      ...
 9a:    88 e1          ldi     r24, 0x18     ; 24      Disable watchdog
 9c:    99 27          eor     r25, r25                ...
 9e:    80 93 60 00    sts     0x0060, r24             ...
 a2:    90 93 60 00    sts     0x0060, r25             ...

Clear entire SRAM (X address 0x0100 to 0x08FF (Z from 0x07FE to 0x0000))
 a6:    ee ef          ldi     r30, 0xFE     ; 254     Z = 0x07FE
 a8:    f7 e0          ldi     r31, 0x07     ; 7       ...
 aa:    a0 e0          ldi     r26, 0x00     ; 0       X = 0x0001
 ac:    b1 e0          ldi     r27, 0x01     ; 1       ...
 ae:    88 27          eor     r24, r24                Clear R24
 b0:    8d 93          st      X+, r24                 Clear X address and post-increment
 b2:    31 97          sbiw    r30, 0x01     ; 1       Z--
 b4:    e9 f7          brne    .-6           ;  0xb0   Repeat until Z == 0

 b6:    87 e1          ldi     r24, 0x17     ; 23      0x17 -> UBRR0L
 b8:    80 93 c4 00    sts     0x00C4, r24             ...
 bc:    80 e0          ldi     r24, 0x00     ; 0       K0 -> UBRR0H: fosc / ((UBRR0 + 1) * 16) = 6 000 000 / 24 / 16 = 15 625 Hz
 be:    80 93 c5 00    sts     0x00C5, r24             ... (UCSR0A.U2X0 is initialized to 0 => prescaler 16 above)
 c2:    88 e1          ldi     r24, 0x18     ; 24      K0b0001 1000 -> UCSR0B (USART control and status register 0 B)
 c4:    80 93 c1 00    sts     0x00C1, r24             ... Enable RXEN0 and TXEN0

 c8:    66 24          eor     r6, r6                  Clear R6 (reserved for e.g. error flag)

 ca:    86 e0          ldi     r24, 0x06     ; 6       K0b0000 0110 -> UCSR0C (Asynch, no parity, 1 stop bit, 8 bit char, pol 0)
 cc:    80 93 c2 00    sts     0x00C2, r24             ...
 d0:    70 91 c1 00    lds     r23, 0x00C1             UCSR0B -> R23
 d4:    70 68          ori     r23, 0x80     ; 128     R23 = R23 | 0b1000 0000 = 0b0001 1000 | 0b1000 0000 = 0b1001 1000
 d6:    70 93 c1 00    sts     0x00C1, r23             This enables RX0 interrupt
```

```
da:     50 98       cbi       0x0a, 0     ; 10     PortD.0 = input
dc:     51 9a       sbi       0x0a, 1     ; 10     PortD.1 = output
de:     78 94       sei                            Set global interrupt flag in SREG (enable interrupts)
```

A.6.13 ATxmega128A1 initialization code

```
100     8f.ef         SER R24            Set Register                 SPL = 0xFF
101     8d.bf         OUT 0x3D,R24       Out to I/O location          ...
102     c8.ed         LDI R28,0xD8       Load immediate               XL  = 0xD8
103     e0.ec         LDI R30,0xC0       Load immediate               YL  = 0xC0
104     4e.2e         MOV R4,R30         Copy register                R4  = 0xC0
105     8f.e3         LDI R24,0x3F       Load immediate               SPH = 0x3F (SP = 0x3F FF)
106     8e.bf         OUT 0x3E,R24       Out to I/O location          ...
107     df.e3         LDI R29,0x3F       Load immediate               YH  = 0x3F (Y = SW stack pointer = 0x3F C0)
108     ff.e3         LDI R31,0x3F       Load immediate               ZH  = 0x3F
109     5f.2e         MOV R5,R31         Copy register                R5  = 0x3F (Frame pointer = 0x3F C0)
10A     00.90.78.00   LDS R0,0x0078      Load direct from data space  R0  = RST_STATUS
10C     8f.e3         LDI R24,0x3F       Load immediate               RST_STATUS = 0x3F (clear all reset flags)
10D     80.93.78.00   STS 0x0078,R24     Store direct to data space   ...
10F     78.ed         LDI R23,0xD8       Load immediate               CCP (Configuration Change Protection) = 0xD8) IOREG
110     74.bf         OUT 0x34,R23       Out to I/O location          ...
111     80.93.80.00     STS 0x0080,R24   Store direct to data space   WDT_CTRL (Watchdog timer control) = 0x3F
                                                            (Enable WDT but w/ reserved setting that doesn't time out)

;  Clear SRAM from 0x20 00 to 0x3FF FF
113     ee.ef         LDI R30,0xFE       Load immediate               Z = 0x1F FE
114     ff.e1         LDI R31,0x1F       Load immediate               ...
115     a0.e0         LDI R26,0x00       Load immediate               X = 0x20 00
116     b0.e2         LDI R27,0x20       Load immediate               ...
117     88.27         CLR R24            Clear Register               R24 = 0
118     8d.93         ST X+,R24          Store indirect and postincrement ...
119     31.97         SBIW R30,0x01      Subtract immediate from word ...
11A     e9.f7         BRNE PC-0x02       Branch if not equal          ...
11B     8f.e1         LDI R24,0x1F       Load immediate               ???

11C     66.24         CLR R6             Clear Register               Clear BASCOM status register R6

;  Config Osc = Disabled , Pllosc = Disabled , Extosc = Enabled , 32khzosc = Disa
11D     73.e4         LDI R23,0x43       Load immediate               OSC_XOSCCTRL = 0x43 (2MHz - 9MHz, XTAL_256CLK)
11E     70.93.52.00     STS 0x0052,R23   Store direct to data space   ...
120     78.e0         LDI R23,0x08       Load immediate               OSC_CTRL = 0x08 (Enable external clock)
121     70.93.50.00     STS 0x0050,R23   Store direct to data space   ...

;  Config Sysclock = External , Prescalea = 1 , Prescalebc = 1_1
123     80.91.51.00     LDS R24,0x0051   Load direct from data space  Wait until bit 3 in OSC_STATUS is set (clock stable)
125     83.ff         SBRS R24,3         Skip if bit in register set  ...
126     fc.cf         RJMP PC-0x0003     Relative jump                ...

127     78.ed         LDI R23,0xD8       Load immediate               CCP (Configuration Change Protection) = 0xD8) IOREG
128     74.bf         OUT 0x34,R23       Out to I/O location          ...

129     73.e0         LDI R23,0x03       Load immediate               CLK_CTRL = 0x03 (Use external clock)
12A     70.93.40.00     STS 0x0040,R23   Store direct to data space   ...

12C     78.ed         LDI R23,0xD8       Load immediate               CCP (Configuration Change Protection) = 0xD8) IOREG
12D     74.bf         OUT 0x34,R23       Out to I/O location          ...

12E     70.e0         LDI R23,0x00       Load immediate               CLK_PSCTRL = 0 (Divide by 1, presc B and C no division)
12F     70.93.41.00     STS 0x0041,R23   Store direct to data space   ...

;  Config Priority = Static , Vector = Application , Lo = Enabled , Med = Enabled
131     77.e0         LDI R23,0x07       Load immediate               PMIC_CTRL = 0x07 (Enable high, mid, & low interrupts)
132     70.93.a2.00     STS 0x00A2,R23   Store direct to data space   ...

;  Config Com5 = 15625 , Mode = 0 , Parity = None , Stopbits = 1 , Databits = 8
;134    78.e0         LDI R23,0x08       Load immediate               PORTE_DIRSET = 0b0000 1000 (bit 3 output)
;135    70.93.81.06   STS 0x0681,R23     Store direct to data space   ...
        1 word        LDI R26,0xA4       Load immediate               Point X to USARTE0_CTRLB
        1 word        LDI R27,0x0A       Load immediate               ...
        88.e1         R24,0x18           Load immediate               USARTE0_CTRLB = 0x18 (Enable RX and TX)
        8d.93         ST X+,R24          Store indirect and postincrement...
```

```
          1 word      LDI R24,0x03          Load immediate                 USARTE0_CTRLC = 0x03 (Asynch, no par, 1 stop bit, 8-bit)
          8d.93       ST X+,R24             Store indirect and postincrement...
          1 word      LDI R24,80            Load immediate                 USARTE0_BAUDCTRLA = 0x80 (BSEL = 0xF 80)
          8d.93       ST X+,R24             Store indirect and postincrement...
          1 word      LDI R24,9F            Load immediate                 USARTE0_BAUDCTRLB = 0x9F (Baud rate scale factor 9)
          1 word      ST X,R24              Store indirect                 ...


;  Config Serialin4 = Buffered , Size = 20
141       80.91.b3.09   LDS R24,USARTE0_CTRLA  Load direct from data space  R24 = USARTE0_CTRLA     (USART E0 RXC interrupt level
MED)
143       8f.7c         ANDI R24,0xCF          Logical AND with immediate   R24 = R24 & 0b1100 1111
144       80.62         ORI R24,0x20           Logical OR with immediate    R24 = R24 | 0b0010 0000
145       80.93.b3.09   STS USARTE0_CTRLA,R24  Store direct to data space   USARTE0_CTRLA = R24

;  Config Porte.2 = Input
147       74.e0         LDI R23,0x04           Load immediate  PORTE_DIRCLR = 0b0000 0100 (bit 2 input) (unnecessary)
148       70.93.82.06   STS 0x0682,R23         Store direct to data space

;  Config Porte.3 = Output
14A       78.e0         LDI R23,0x08           Load immediate  PORTE_DIRSET = 0b0000 1000 (bit 3 output) (unnecessary)
14B       70.93.81.06   STS 0x0681,R23         Store direct to data space

Enable Interrupts
14D       78.94         SEI                    Global Interrupt Enable
```

A.6.14 Other compiled code that's unused by the test application

A.6.14.1 Both ATmega and ATxmega

Generic delay routine (place 16-bit delay value in Z and rcall x20D, which will count down to zero and return)

```
20D       31.97         SBIW R30,0x01              Subtract immediate from word
20E       f1.f7         BRNE PC-0x01               Branch if not equal
20F       08.95         RET                        Subroutine return
```

Set error bit 2 in R6

```
210       68.94         SET                        Set T in SREG
211       62.f8         BLD R6,2                   Bit load from T to register
212       08.95         RET                        Subroutine return
```

Clear error bit 2 in R6

```
213       e8.94         CLT                        Clear T in SREG
214       62.f8         BLD R6,2                   Bit load from T to register
215       08.95         RET                        Subroutine return
```

A.6.14.2 Only ATmega1284 and ATxmega128A1

RAMPZ

```
288       0F 93         Push R16                   Set RAMPZ to 1
28A       01 E0         Ldi R16, 0x01

28C       0B BF         Out RAMPZ, R16
28E       0F 91         Pop R16
290       88 94         Clc (clear carry flag)
292       08 95         Ret

294       0F 93         Push R16                   Set RAMPZ to 0
296       00 27         eor          r16, r16      Clear R16
298       F9 CF         rjmp -7+1                  0x28C

29A       0F 93         Push R16                   Set RAMPZ to 2
29C       02 E0         Ldi R16, 0x02
29E       F6 CF         rjmp -10+1                 0x28C

2A0       0F 93         Push R16                   Set RAMPZ to 3
2A2       03 E0         Ldi R16, 0x03
2A4       F3 CF         -13+1                       0x28C
```

A.6.14.3 Only ATxmega128A1

Clear _XMEGAREG 32 bytes BASCOM area

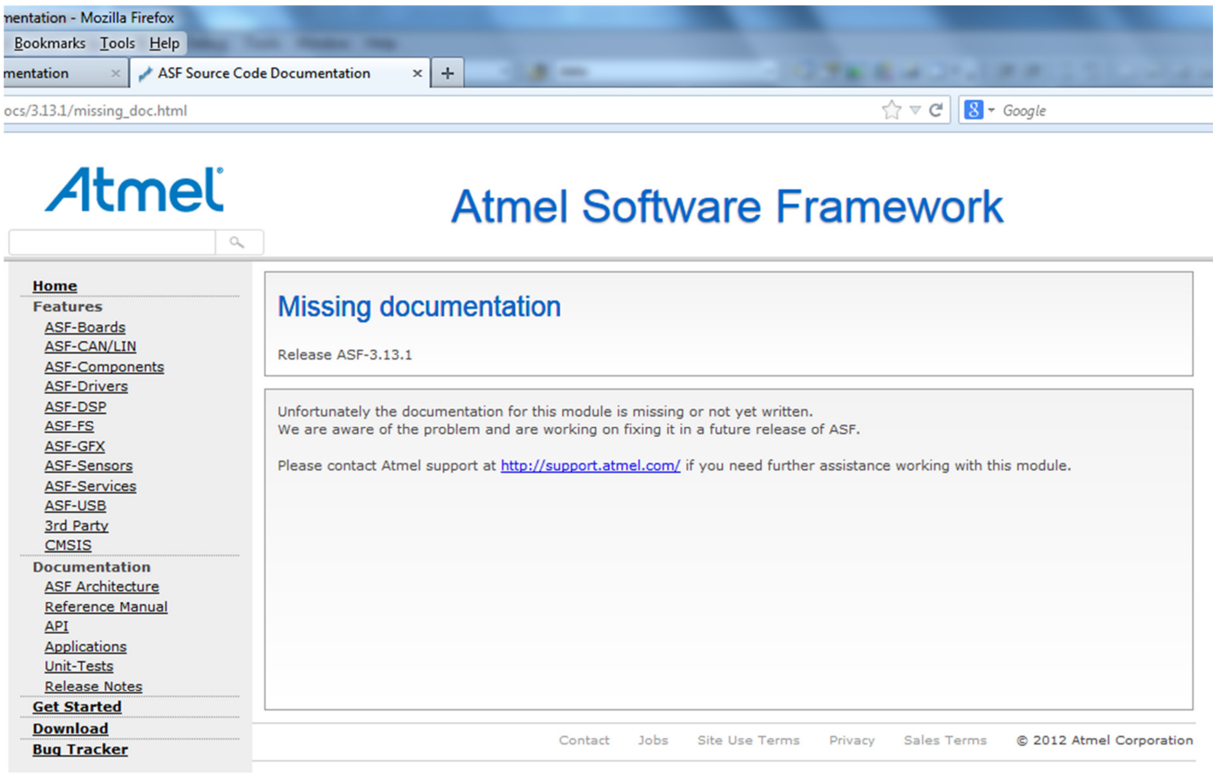
```
269     8f.93          PUSH R24             Push register on stack
26A     9f.93          PUSH R25             Push register on stack
26B     af.93          PUSH R26             Push register on stack
26C     bf.93          PUSH R27             Push register on stack

26D     88.27          CLR R24              Clear Register

26E     90.e2           LDI R25,0x20        Load immediate

26F     a1.e0          LDI R26,0x01         Load immediate          _XMEGAREG
270     b0.e2          LDI R27,0x20         Load immediate

271     8d.93          ST X+,R24            Store indirect and postincrement     Clear 0x2001 _XMEGAREG ___BTMPAX
272     9a.95          DEC R25              Decrement

273     e9.f7          BRNE PC-0x02         Branch if not equal     Do 32 times until R25 == 0

274     bf.91          POP R27              Pop register from stack
275     af.91          POP R26              Pop register from stack
276     9f.91          POP R25              Pop register from stack
277     8f.91          POP R24              Pop register from stack
278     08.95          RET                  Subroutine return
```

Shift each of the 5 first bytes of _XMEGAREG left by one through carry. Multiply 5-byte number by 2.

```
279     8f.93          PUSH R24             Push register on stack

27A     80.91.01.20    LDS R24,0x2001       Load direct from data space
27C     88.1f          ROL R24              Rotate Left Through Carry
27D     80.93.01.20    STS 0x2001,R24       Store direct to data space

27F     80.91.02.20    LDS R24,0x2002       Load direct from data space
281     88.1f          ROL R24              Rotate Left Through Carry
282     80.93.02.20    STS 0x2002,R24       Store direct to data space

284     80.91.03.20    LDS R24,0x2003       Load direct from data space
286     88.1f          ROL R24              Rotate Left Through Carry
287     80.93.03.20    STS 0x2003,R24       Store direct to data space

289     80.91.04.20    LDS R24,0x2004       Load direct from data space
28B     88.1f          ROL R24              Rotate Left Through Carry
28C     80.93.04.20    STS 0x2004,R24       Store direct to data space

28E     80.91.05.20    LDS R24,0x2005       Load direct from data space
290     88.1f          ROL R24              Rotate Left Through Carry
291     80.93.05.20    STS 0x2005,R24       Store direct to data space
293     8f.91          POP R24              Pop register from stack
294     08.95          RET                  Subroutine return
```

A.7 Atmel studio 6.1 and ASF screen dumps

A.7.1 ASF error messages


Atmel Software Framework

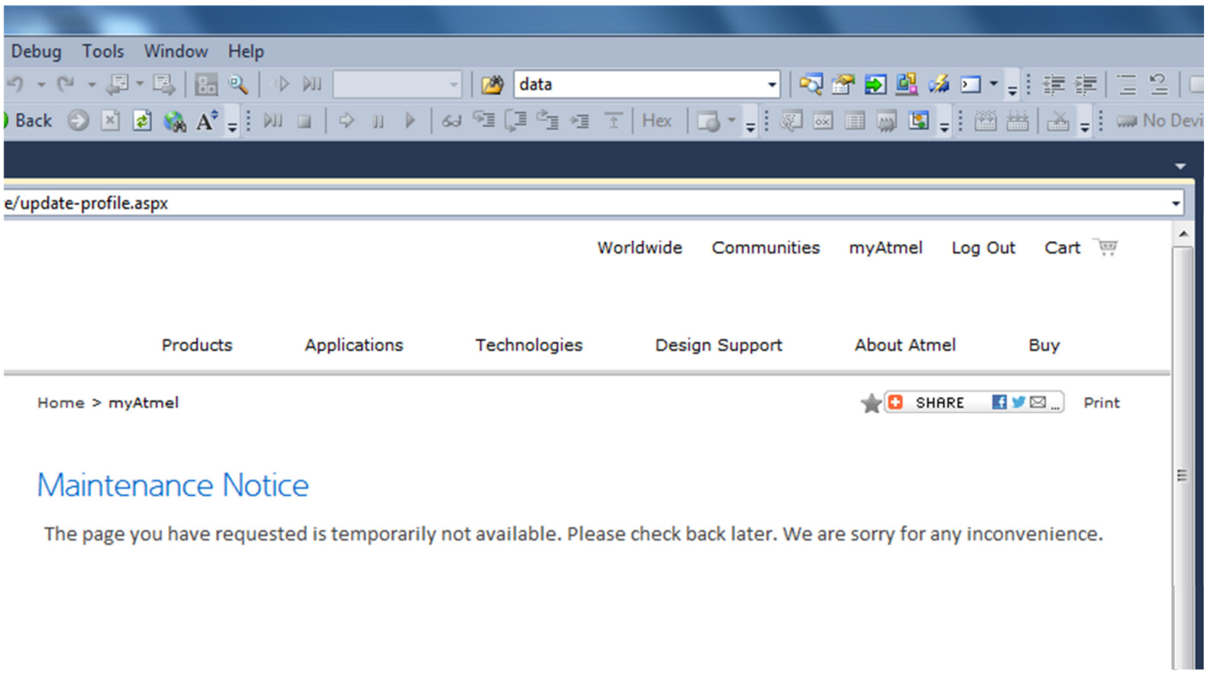
Missing documentation

Release ASF-3.13.1

Unfortunately the documentation for this module is missing or not yet written.
We are aware of the problem and are working on fixing it in a future release of ASF.

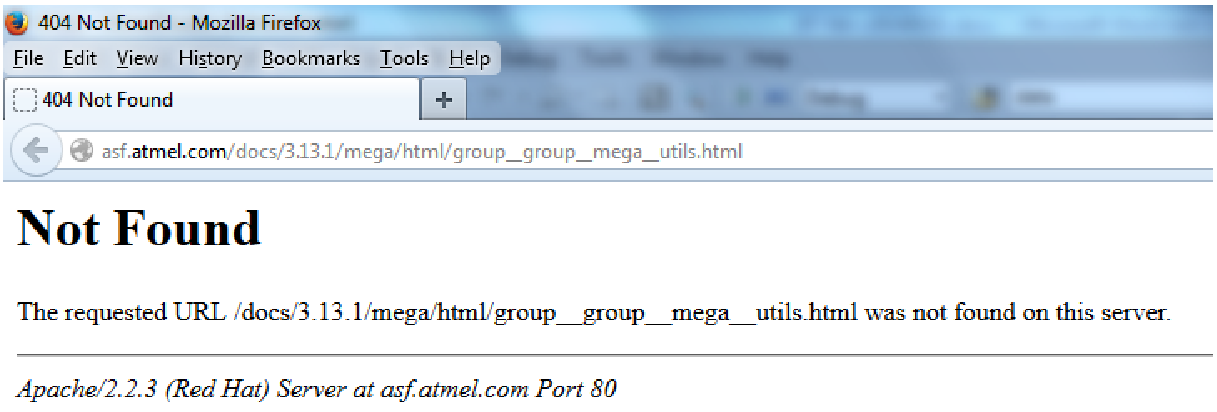
Please contact Atmel support at http://support.atmel.com/ if you need further assistance working with this module.

Figure 35: Missing ASF ATmega system clock control quickstart guide


Maintenance Notice

The page you have requested is temporarily not available. Please check back later. We are sorry for any inconvenience.

Figure 36: Maintenance Notice


404 Not Found - Mozilla Firefox

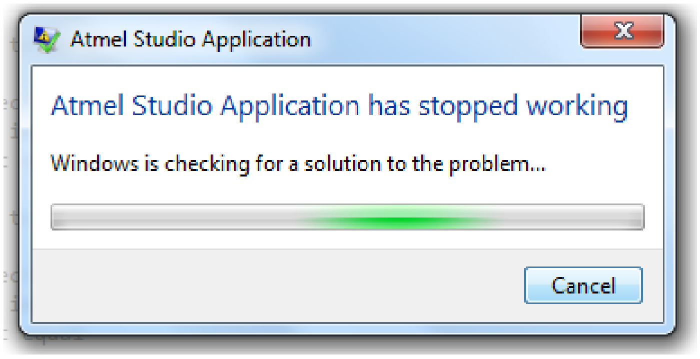
File Edit View History Bookmarks Tools Help
404 Not Found
+
asf.atmel.com/docs/3.13.1/mega/html/group__group__mega__utils.html

Not Found

The requested URL /docs/3.13.1/mega/html/group__group__mega__utils.html was not found on this server.

Apache/2.2.3 (Red Hat) Server at asf.atmel.com Port 80

Figure 37: Missing ASF MEGA compiler driver missing documentation

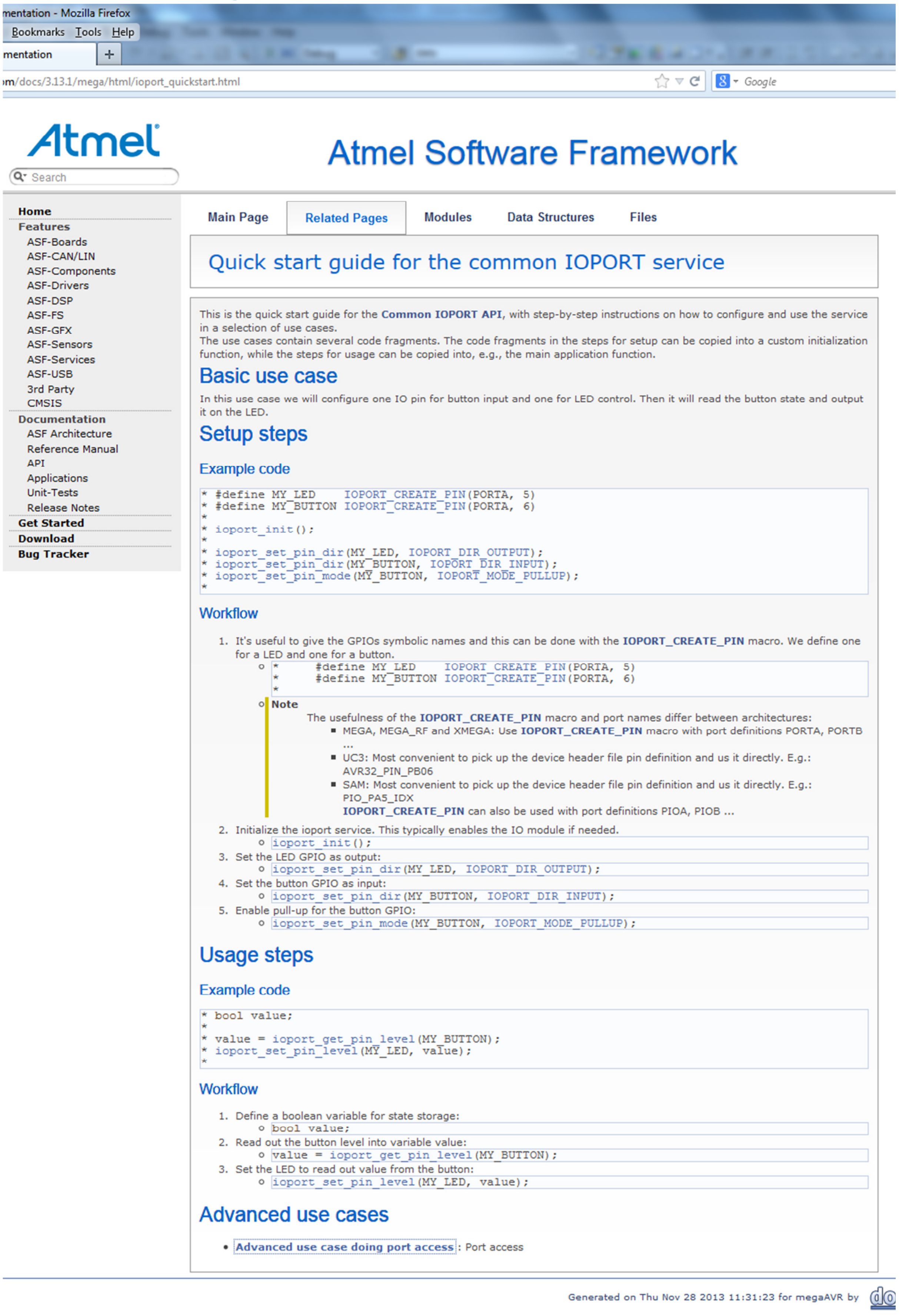

Atmel Studio Application

Atmel Studio Application has stopped working

Windows is checking for a solution to the problem...

Figure 38: Hanging when debugging, in this case when using the simulator

A.7.2 ASF quick start guide example



Quick start guide for the common IOPORT service

This is the quick start guide for the **Common IOPORT API**, with step-by-step instructions on how to configure and use the service in a selection of use cases.
The use cases contain several code fragments. The code fragments in the steps for setup can be copied into a custom initialization function, while the steps for usage can be copied into, e.g., the main application function.

Basic use case

In this use case we will configure one IO pin for button input and one for LED control. Then it will read the button state and output it on the LED.

Setup steps

Example code

```
* #define MY_LED    IOPORT_CREATE_PIN(PORTA, 5)
* #define MY_BUTTON IOPORT_CREATE_PIN(PORTA, 6)
*
* ioport_init();
*
* ioport_set_pin_dir(MY_LED, IOPORT_DIR_OUTPUT);
* ioport_set_pin_dir(MY_BUTTON, IOPORT_DIR_INPUT);
* ioport_set_pin_mode(MY_BUTTON, IOPORT_MODE_PULLUP);
*
```

Workflow

1. It's useful to give the GPIOs symbolic names and this can be done with the **IOPORT_CREATE_PIN** macro. We define one for a LED and one for a button.

   -
     ```
     *      #define MY_LED    IOPORT_CREATE_PIN(PORTA, 5)
     *      #define MY_BUTTON IOPORT_CREATE_PIN(PORTA, 6)
     *
     ```
   - **Note**
     The usefulness of the **IOPORT_CREATE_PIN** macro and port names differ between architectures:
     - MEGA, MEGA_RF and XMEGA: Use **IOPORT_CREATE_PIN** macro with port definitions PORTA, PORTB ...
     - UC3: Most convenient to pick up the device header file pin definition and us it directly. E.g.: AVR32_PIN_PB06
     - SAM: Most convenient to pick up the device header file pin definition and us it directly. E.g.: PIO_PA5_IDX
       IOPORT_CREATE_PIN can also be used with port definitions PIOA, PIOB ...
2. Initialize the ioport service. This typically enables the IO module if needed.
   - `ioport_init();`
3. Set the LED GPIO as output:
   - `ioport_set_pin_dir(MY_LED, IOPORT_DIR_OUTPUT);`
4. Set the button GPIO as input:
   - `ioport_set_pin_dir(MY_BUTTON, IOPORT_DIR_INPUT);`
5. Enable pull-up for the button GPIO:
   - `ioport_set_pin_mode(MY_BUTTON, IOPORT_MODE_PULLUP);`

Usage steps

Example code

```
* bool value;
*
* value = ioport_get_pin_level(MY_BUTTON);
* ioport_set_pin_level(MY_LED, value);
*
```

Workflow

1. Define a boolean variable for state storage:
   - `bool value;`
2. Read out the button level into variable value:
   - `value = ioport_get_pin_level(MY_BUTTON);`
3. Set the LED to read out value from the button:
   - `ioport_set_pin_level(MY_LED, value);`

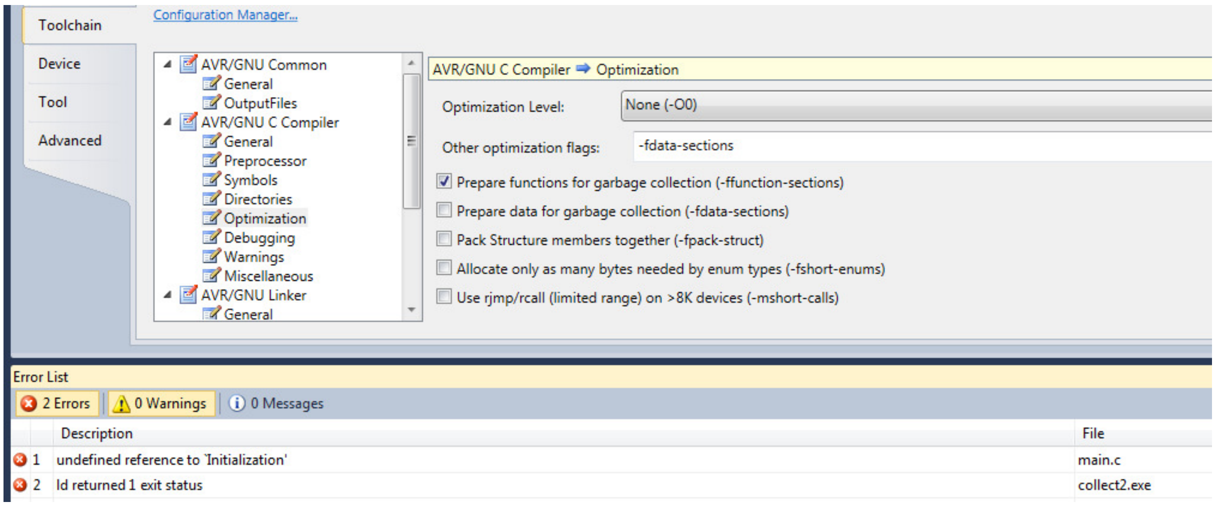
Advanced use cases

- **Advanced use case doing port access**: Port access

Generated on Thu Nov 28 2013 11:31:23 for megaAVR by

Figure 39: Quick start guide example

A.7.3 -O0 optimization error message



AVR/GNU C Compiler ➡ Optimization

Optimization Level: None (-O0)

Other optimization flags: -fdata-sections

- [x] Prepare functions for garbage collection (-ffunction-sections)
- [ ] Prepare data for garbage collection (-fdata-sections)
- [ ] Pack Structure members together (-fpack-struct)
- [ ] Allocate only as many bytes needed by enum types (-fshort-enums)
- [ ] Use rjmp/rcall (limited range) on >8K devices (-mshort-calls)

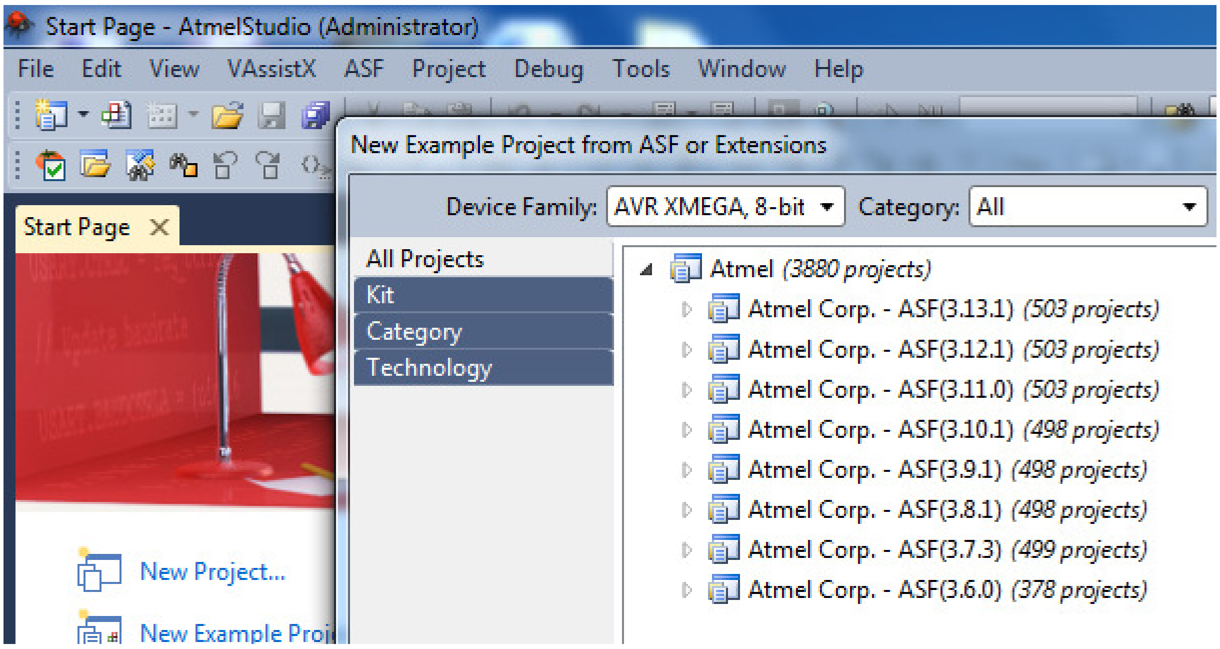
Error List

2 Errors | 0 Warnings | 0 Messages

	Description	File
1	undefined reference to `Initialization'	main.c
2	ld returned 1 exit status	collect2.exe

Figure 40: -O0 optimization error message

A.7.4 Misleading ASF project counter


Start Page - AtmelStudio (Administrator)
File Edit View VAssistX ASF Project Debug Tools Window Help
Start Page
New Project...
New Example Proj
New Example Project from ASF or Extensions
Device Family: AVR XMEGA, 8-bit
Category: All
All Projects
Kit
Category
Technology

Atmel (3880 projects)
- Atmel Corp. - ASF(3.13.1) (503 projects)
- Atmel Corp. - ASF(3.12.1) (503 projects)
- Atmel Corp. - ASF(3.11.0) (503 projects)
- Atmel Corp. - ASF(3.10.1) (498 projects)
- Atmel Corp. - ASF(3.9.1) (498 projects)
- Atmel Corp. - ASF(3.8.1) (498 projects)
- Atmel Corp. - ASF(3.7.3) (499 projects)
- Atmel Corp. - ASF(3.6.0) (378 projects)

(Atmel needs to correct the project counter so that it only counts a project once.)

A.8 Atmel Studio #ports scaling

A.8.1 Statics, row-major (AS3i)

A.8.1.1 ATmega324A total size

Atmega324A, interrupt vector table 126 bytes

Ports	1				2		
Opt	Text	Data	BSS	Delta	Text	Data	BSS
-O1	532	0	36	180	712	0	72
-O2	566	0	36	128	694	0	72
-O3	716	0	36	260	976	0	72
-Os	518	0	36	170	688	0	72

Atmega324A, interrupt vector table excluded

-O1	406	586
-O2	440	568
-O3	590	850
-Os	392	562

A.8.1.2 ATmega1284 total size

Atmega1284, interrupt vector table 140 bytes

Ports	1				2		
Opt	Text	Data	BSS	Delta	Text	Data	BSS
-O1	562	0	36	188	750	0	72
-O2	602	0	36	130	732	0	72
-O3	752	0	36	262	1014	0	72
-Os	548	0	36	178	726	0	72

Atmega1284, interrupt vector table excluded

-O1	422	610
-O2	462	592
-O3	612	874
-Os	408	586

A.8.1.3 ATxmega128A1 total size

Atxmega 128A1, interrupt vector table 500 bytes

Ports	1				2				3		
Opt	Text	Data	BSS	Delta	Text	Data	BSS	Delta	Text	Data	BSS
-O1	1012	0	36	204	1216	0	72	190	1406	0	108
-O2	1014	0	36	196	1210	0	72	178	1388	0	108
-O3	996	0	36	224	1220	0	72	240	1460	0	108
-Os	1006	0	36	198	1204	0	72	172	1376	0	108

Atxmega 128A1, interrupt vector table excluded

-O1	512	716	906
-O2	514	710	888
-O3	496	720	960
-Os	506	704	876

A.8.2 Statics, column-major

A.8.2.1 ATmega324A total size

Atmega324A, interrupt vector table 126 bytes

Ports	1				2		
Opt	Text	Data	BSS	Delta	Text	Data	BSS
-O1	510	0	36	178	688	0	72
-O2	548	0	36	128	676	0	72
-O3	698	0	36	270	968	0	72
-Os	502	0	36	172	674	0	72

Atmega324A, interrupt vector table excluded

-O1	384	562
-O2	422	550
-O3	572	842
-Os	376	548

A.8.2.2 ATmega1284 total size

Atmega1284, interrupt vector table 140 bytes

Ports	1				2		
Opt	Text	Data	BSS	Delta	Text	Data	BSS
-O1	540	0	36	186	726	0	72
-O2	584	0	36	130	714	0	72
-O3	534	0	36	472	1006	0	72
-Os	532	0	36	180	712	0	72

Atmega1284, interrupt vector table excluded

-O1	400	586
-O2	444	574
-O3	394	866
-Os	392	572

A.8.2.3 ATxmega128A1 total size

Atxmega 128A1, interrupt vector table 500 bytes

Ports	1				2				3		
Opt	Text	Data	BSS	Delta	Text	Data	BSS	Delta	Text	Data	BSS
-O1	990	0	36	202	1192	0	72	202	1394	0	108
-O2	996	0	36	196	1192	0	72	188	1380	0	108
-O3	996	0	36	222	1218	0	72	262	1480	0	108
-Os	990	0	36	200	1190	0	72	182	1372	0	108

Atxmega 128A1, interrupt vector table excluded

-O1	490	692	894
-O2	496	692	880
-O3	496	718	980
-Os	490	690	872

A.8.3 Code size and clock cycle count ISR, Interrupt Service Routine

A.8.3.1 Statics row-major

BTBB1i = BTAA1i Statics, row-major ATxmega128A1				BTBB1i = BTAA1i Statics, row-major ATmega1284				BTBB1i = BTAA1i Statics, row-major ATmega324A			
Instr	Instr W	Instr C	Comment	Instr	Instr W	Instr C	Comment	Instr	Instr W	Instr C	Comment
RJMP PC+0x0182	1	2		RJMP PC+0x00E9	1	2		RJMP PC+0x00DA	1	2	
PUSH R1	1	1		PUSH R1	1	2		PUSH R1	1	2	
PUSH R0	1	1		PUSH R0	1	2		PUSH R0	1	2	
IN R0,0x3F	1	1	SREG	IN R0,0x3F	1	1	SREG	IN R0,0x3F	1	1	SREG
PUSH R0	1	1		PUSH R0	1	2		PUSH R0	1	2	
CLR R1	1	1		CLR R1	1	1		CLR R1	1	1	
IN R0,0x38	1	1	RAMPD								
PUSH R0	1	1									
OUT 0x38,R1	1	1	RAMPD								
IN R0,0x3B	1	1	RAMPZ	IN R0,0x3B	1	1	RAMPZ				
PUSH R0	1	1		PUSH R0	1	2					
OUT 0x3B,R1	1	1	RAMPZ	*RAMPZ only relevant for ELPM/SPM*							
PUSH R24	1	1		PUSH R24	1	2		PUSH R24	1	2	
PUSH R30	1	1		PUSH R30	1	2		PUSH R30	1	2	
PUSH R31	1	1		PUSH R31	1	2		PUSH R31	1	2	
LDS R24,0x09A0	2	3	1 extra cycle inserted for int SRAM	LDS R24,0x00CE	2	2		LDS R24,0x00CE	2	2	
LDS R30,0x2001	2	3		LDS R30,0x0101	2	2		LDS R30,0x0101	2	2	
SUBI R30,0xFF	1	1		SUBI R30,0xFF	1	1		SUBI R30,0xFF	1	1	
ANDI R30,0x0F	1	1		ANDI R30,0x0F	1	1		ANDI R30,0x0F	1	1	
STS 0x2001,R30	2	2		STS 0x0101,R30	2	2		STS 0x0101,R30	2	2	
LDI R31,0x00	1	1		LDI R31,0x00	1	1		LDI R31,0x00	1	1	
SUBI R30,0xC8	1	1		SUBI R30,0xC8	1	1		SUBI R30,0xC8	1	1	
SBCI R31,0xDF	1	1		SBCI R31,0xFE	1	1		SBCI R31,0xFE	1	1	
STD Z+0,R24	1	1		STD Z+0,R24	1	2		STD Z+0,R24	1	2	
POP R31	1	2		POP R31	1	2		POP R31	1	2	
POP R30	1	2		POP R30	1	2		POP R30	1	2	
POP R24	1	2		POP R24	1	2		POP R24	1	2	
POP R0	1	2		POP R0	1	2					
OUT 0x3B,R0	1	1	RAMPZ	OUT 0x3B,R0	1	1	RAMPZ				
POP R0	1	2		POP R0	1	2		POP R0	1	2	
OUT 0x38,R0	1	1	RAMPD								
POP R0	1	2									
OUT 0x3F,R0	1	1	SREG	OUT 0x3F,R0	1	1	SREG	OUT 0x3F,R0	1	1	SREG
POP R0	1	2		POP R0	1	2		POP R0	1	2	
POP R1	1	2		POP R1	1	2		POP R1	1	2	
RETI	1	5	22-bit PC	RETI	1	5	22-bit PC	RETI	1	4	16-bit PC
Statics, row-major	**39**	**54**			**33**	**53**			**29**	**46**	

Blue colour indicates that the implementation is more efficient, red that it is less so. “Instr W” means Instruction size in words (=2 bytes) and “Instr C” means number of clock cycles.

A.8.3.2 Structs and pointers

BTB1h = AS1h Structs and pointers ATxmega128A1				BTB1h = AS1h Structs and pointers ATmega1284				BTB1h = AS1h Structs and pointers ATmega324A			
Instr	Instr W	Instr C	Comment	Instr	Instr W	Instr C	Comment	Instr	Instr W	Instr C	Comment
RJMP PC+0x017D	1	2		RJMP PC+0x00DB	1	2		RJMP PC+0x00CC	1	2	
PUSH R1	1	1		PUSH R1	1	2		PUSH R1	1	2	
PUSH R0	1	1		PUSH R0	1	2		PUSH R0	1	2	
IN R0,0x3F	1	1	SREG	IN R0,0x3F	1	1	SREG	IN R0,0x3F	1	1	SREG
PUSH R0	1	1		PUSH R0	1	2		PUSH R0	1	2	
CLR R1	1	1		CLR R1	1	1		CLR R1	1	1	
IN R0,0x38	1	1	RAMPD								
PUSH R0	1	1									
OUT 0x38,R1	1	1	RAMPD								
IN R0,0x39	1	1	RAMPX								
PUSH R0	1	1									
OUT 0x39,R1	1	1	RAMPX								
IN R0,0x3B	1	1	RAMPZ	IN R0,0x3B	1	1	RAMPZ				
PUSH R0	1	1		PUSH R0	1	2					
OUT 0x3B,R1	1	1	RAMPZ	*RAMPZ only relevant for ELPM/SPM*							
PUSH R24	1	1		PUSH R24	1	2		PUSH R24	1	2	
PUSH R26	1	1		PUSH R26	1	2		PUSH R26	1	2	
PUSH R27	1	1		PUSH R27	1	2		PUSH R27	1	2	
PUSH R30	1	1		PUSH R30	1	2		PUSH R30	1	2	
PUSH R31	1	1		PUSH R31	1	2		PUSH R31	1	2	
LDI R26,0x4E	1	1		LDI R26,0x4E	1	1		LDI R26,0x4E	1	1	
LDI R27,0x20	1	1		LDI R27,0x01	1	1		LDI R27,0x01	1	1	
LD R30,X	1	2	Below	LD R30,X	1	2		LD R30,X	1	2	
SUBI R30,0xFF	1	1		SUBI R30,0xFF	1	1		SUBI R30,0xFF	1	1	
ANDI R30,0x0F	1	1		ANDI R30,0x0F	1	1		ANDI R30,0x0F	1	1	
ST X,R30	1	1		ST X,R30	1	1		ST X,R30	1	1	
LDI R31,0x00	1	1		LDI R31,0x00	1	1		LDI R31,0x00	1	1	
LDS R26,0x203C	2	3	1 extra cycle inserted for int SRAM	LDS R26,0x013C	2	2		LDS R26,0x013C	2	2	
LDS R27,0x203D	2	3		LDS R27,0x013D	2	2		LDS R27,0x013D	2	2	
				ADIW R26,0x06	1	2		ADIW R26,0x06	1	2	
LD R24,X	1	2		LD R24,X	1	2		LD R24,X	1	2	
SUBI R30,0xC4	1	1		SUBI R30,0xC4	1	1		SUBI R30,0xC4	1	1	
SBCI R31,0xDF	1	1		SBCI R31,0xFE	1	1		SBCI R31,0xFE	1	1	
STD Z+2,R24	1	2		STD Z+2,R24	1	2		STD Z+2,R24	1	2	
POP R31	1	2		POP R31	1	2		POP R31	1	2	
POP R30	1	2		POP R30	1	2		POP R30	1	2	
POP R27	1	2		POP R27	1	2		POP R27	1	2	
POP R26	1	2		POP R26	1	2		POP R26	1	2	
POP R24	1	2		POP R24	1	2		POP R24	1	2	
POP R0	1	2		POP R0	1	2		POP R0	1	2	
OUT 0x3B,R0	1	1	RAMPZ	OUT 0x3B,R0	1	1	RAMPZ				
POP R0	1	2		POP R0	1	2					
OUT 0x39,R0	1	1	RAMPX								
POP R0	1	2									
OUT 0x38,R0	1	1	RAMPD								
POP R0	1	2									
OUT 0x3F,R0	1	1	SREG	OUT 0x3F,R0	1	1	SREG	OUT 0x3F,R0	1	1	SREG
POP R0	1	2		POP R0	1	2		POP R0	1	2	
POP R1	1	2		POP R1	1	2		POP R1	1	2	
RETI	1	5	22-bitPC	RETI	1	5	22-bit PC	RETI	1	4	16-bit PC
Structs & ptrs											
	51	**72**			**41**	**68**			**37**	**61**	

A.8.4 Code size and clock cycle count, transmitting

A.8.4.1 Statics row-major

BTBB1i = BTAA1i

Statics

ATxmega128A1	1 port	NN	2 ports	Port 0	Port 1	3 ports	Port 0	Port 1	Port 2	
Instr	Instr W	Instr C	Instr W	Instr C	Instr C	Instr W	Instr C	Instr C	Instr C	Comment
CPSE R22,R1			1	2	1	1	2	1	1	
RJMP PC+0x0008			1		2	1		2	2	
LDS R25,0x0AA1	2	3	2	3		2	3			Int SRAM
SBRS R25,5	1	2	1	2		1	2			1st try
RJMP PC-0x0003	1	2	1	2		1	2			
STS 0x0AA0,R24	2	2	2	2		2	2			
RET	1	5	1	5		1	5			22-bit PC
DEC R22						*1*		*1*	*1*	
BRNE _+8						*1*		*1*	*2*	
LDS R25,0x0AA1						*2*		*3*		*Int SRAM*
SBRS R25,5						*1*		*2*		*1st try*
RJMP PC-0x0003						*1*		*2*		
STS 0x0AA0,R24						*2*		*2*		
RET						*1*		*5*		*22-bit PC*
LDS R25,0x09A1			2		3	2			3	Int SRAM
SBRS R25,5			1		2	1			2	1st try
RJMP PC-0x0003			1		2	1			2	
STS 0x09A0,R24			2		2	2			2	
RET			1		5	1			5	22-bit PC
Statics ATxmega128A1	7	14	16	16	17	25	16	19	20	
Statics ATmega1284	7	13	16	15	16					
Statics ATmega324A	7	12	16	14	15					

A.8.4.2 Structs and pointers

BTB1h = AS1h

Structs and pointers

ATxmega128A1

Instr	Instr W	Instr C	Comment
MOVW R30,R22	1	1	
LDD R25,Z+1	1	3	Int SRAM
SBRS R25,5	1	2	1st try
RJMP PC-0x0003	1	2	
STD Z+0,R24	2	1	
RET	1	5	22-bit PC
S&P ATxmega128A1	7	14	
S&P ATmega1284	7	13	Quicker LDD but slower STD Z+
S&P ATmega324A	7	12	Above and quicker RET due to 16-bit PC

A.8.5 Protocol-bound ISR scaling (AS3j & AS3k)

Text, bytes	1S statics row-m, 18 BSS	1M S&P, 20 BSS	1M statics row-m, 18 BSS	2M S&P, 40 BSS	2M statics row-m, 36 BSS
ATmega324A					
-O1	444	512	502	636	700
-O2	438	490	488	606	666
-O3	436	486	478	680	680
-Os	438	484	492	604	658
ATmega1284					
-O1	472	536	530	668	736
-O2	466	514	516	638	702
-O3	464	510	506	712	716
-Os	466	508	520	636	694
ATxmega128A1					
-O1	932	996	990	1170	1216
-O2	934	990	984	1140	1200
-O3	932	990	974	1218	1214
-Os	934	982	988	1140	1192

A.8.6 Common ISR assembly code

```
000001D0 1f.92                PUSH R1                 Push register on stack
000001D1 0f.92                PUSH R0                 Push register on stack
000001D2 0f.b6                IN R0,0x3F              In from I/O location
000001D3 0f.92                PUSH R0                 Push register on stack
000001D4 11.24                CLR R1                  Clear Register
000001D5 08.b6                IN R0,0x38              In from I/O location
000001D6 0f.92                PUSH R0                 Push register on stack
000001D7 18.be                OUT 0x38,R1             Out to I/O location
000001D8 09.b6                IN R0,0x39              In from I/O location
000001D9 0f.92                PUSH R0                 Push register on stack
000001DA 19.be                OUT 0x39,R1             Out to I/O location
000001DB 0b.b6                IN R0,0x3B              In from I/O location
000001DC 0f.92                PUSH R0                 Push register on stack
000001DD 1b.be                OUT 0x3B,R1             Out to I/O location
000001DE 2f.93                PUSH R18                Push register on stack
000001DF 3f.93                PUSH R19                Push register on stack
000001E0 4f.93                PUSH R20                Push register on stack
000001E1 5f.93                PUSH R21                Push register on stack
000001E2 6f.93                PUSH R22                Push register on stack
000001E3 7f.93                PUSH R23                Push register on stack
000001E4 8f.93                PUSH R24                Push register on stack
000001E5 9f.93                PUSH R25                Push register on stack
000001E6 af.93                PUSH R26                Push register on stack
000001E7 bf.93                PUSH R27                Push register on stack
000001E8 ef.93                PUSH R30                Push register on stack
000001E9 ff.93                PUSH R31                Push register on stack
             USART_RXComplete(&SerialData1);
000001EA 84.e1                LDI R24,0x14            Load immediate
000001EB 90.e2                LDI R25,0x20            Load immediate
000001EC c2.df                RCALL PC-0x003D         Relative call subroutine
             }
000001ED ff.91                POP R31                 Pop register from stack
000001EE ef.91                POP R30                 Pop register from stack
000001EF bf.91                POP R27                 Pop register from stack
000001F0 af.91                POP R26                 Pop register from stack
000001F1 9f.91                POP R25                 Pop register from stack
000001F2 8f.91                POP R24                 Pop register from stack
000001F3 7f.91                POP R23                 Pop register from stack
000001F4 6f.91                POP R22                 Pop register from stack
000001F5 5f.91                POP R21                 Pop register from stack
000001F6 4f.91                POP R20                 Pop register from stack
000001F7 3f.91                POP R19                 Pop register from stack
000001F8 2f.91                POP R18                 Pop register from stack
```

```
000001F9 0f.90                POP R0                Pop register from stack
000001FA 0b.be                OUT 0x3B,R0           Out to I/O location
000001FB 0f.90                POP R0                Pop register from stack
000001FC 09.be                OUT 0x39,R0           Out to I/O location
000001FD 0f.90                POP R0                Pop register from stack
000001FE 08.be                OUT 0x38,R0           Out to I/O location
000001FF 0f.90                POP R0                Pop register from stack
00000200 0f.be                OUT 0x3F,R0           Out to I/O location
00000201 0f.90                POP R0                Pop register from stack
00000202 1f.90                POP R1                Pop register from stack
00000203 18.95                RETI                  Interrupt return
```

A.9 ATmega324A structs and pointers two-port USART ISR placed in IV

A.9.1 C code

```c
void USART_RX_complete(working_data_t * working_data)
{
   uint8_t data = working_data->usart->UDR;
   if (data == 254)
   {
      working_data->serial_data_status = 1;
      working_data->receive_counter = 0;
      working_data->serial_data[0] = 254;
   }
   else
   {
      working_data->receive_counter++;
      if (working_data->receive_counter < USART_RX_BUFFER_SIZE)
      {
         working_data->serial_data[working_data->receive_counter] = data;
         if (data == 255)
         {
            working_data->serial_data_status = 2;
         }
      }
      else
      {
         working_data->serial_data_status = 0;
         working_data->receive_counter = 0;
      }
   }
}
```

A.9.2 Assembly

```
#include <avr/io.h>
#include "defines.h"
.section .myvectors, "ax", @progbits
.global __vector_default
__vector_default:
// Reset vector
   rjmp __init                 // 0x00
rx_common_entry:
   push r1                     // 0x01
   push r0                     // 0x02
   in   r0, _SFR_IO_ADDR(SREG)   // 0x03 SREG
   push r0                     // 0x04
   rjmp rx_skip_int2           // 0x05
// INT2 vector
   rjmp __vector_3             // 0x06 INT2
rx_skip_int2:
   clr  r1                     // 0x07
   push r25                    // 0x08
   push r26                    // 0x09
   push r27                    // 0x0A
   rjmp rx_skip_pcint2         // 0x0B
// PCINT2 vector
   rjmp __vector_6             // 0x0C PCINT2
rx_skip_pcint2:
   cpi  r24, 0xfe              // 0x0D if (data == 254)
   brne rx_not_start_token     // 0x0E else
   ldi  r25, 0x01              // 0x0F working_data->serial_data_status=1;
   std  Z+18, r25              // 0x10
   std  Z+19, r1               // 0x11 working_data->receive_counter = 0;
   std  Z+2, r24               // 0x12 working_data->serial_data[0] = 254;
   rjmp rx_epilogue            // 0x13 jump to isr epilogue
rx_not_start_token:
   ldd  r25, Z+19              // 0x14 working_data->receive_counter++;
   subi r25, 0xff              // 0x15
   std  Z+19, r25              // 0x16
   ldd  r25, Z+19      // 0x17 if (working_data->receive_counter < USART_RX_BUFFER_SIZE)
   cpi  r25, USART_RX_BUFFER_SIZE // 0x18
   brcc rx_overflow            // 0x19
   ldd  r25, Z+19//0x1A working_data->serial_data[working_data->receive_counter]= data;
   movw r26, r30               // 0x1B
   add  r26, r25               // 0x1C
   adc  r27, r1                // 0x1D
   adiw r26, 0x02              // 0x1E
   st   X, r24                 // 0x1F
rx_cont4:
   cpi  r24, 0xff              // 0x20 if (data == 255)
   brne rx_not_end_token       // 0x21 else
   ldi  r24, 0x02              // 0x22 working_data->serial_data_status=2;
rx_not_end_token:
   std  Z+18, r24              // 0x23
rx_epilogue:
   pop  r27                    // 0x24
   pop  r26                    // 0x25
   pop  r25                    // 0x26
   rjmp rx_skip_usart0         // 0x27
// USART0 RX vector
   push r24                    // 0x28
   push r30                    // 0x29
   push r31                    // 0x2A
   ldi  r30, lo8(serial_data_0)   // 0x2B
   ldi  r31, hi8(serial_data_0)   // 0x2C
   lds  r24, _SFR_MEM_ADDR(UDR0)  // 0x2D UDR0
   // 2-word instruction          // 0x2E
   rjmp rx_common_entry           // 0x2F
rx_skip_usart0:
   pop  r0                        // 0x30
   out  _SFR_IO_ADDR(SREG), r0    // 0x31 SREG
   pop  r0                        // 0x32
   pop  r1                        // 0x33
   pop  r31                       // 0x34
   pop  r30                       // 0x35
   pop  r24                       // 0x36
   reti                           // 0x37
// USART1 RX vector
   push r24                       // 0x38
   push r30                       // 0x39
   push r31                       // 0x3A
   ldi  r30, lo8(serial_data_1)   // 0x3B
   ldi  r31, hi8(serial_data_1)   // 0x3C
   lds  r24, _SFR_MEM_ADDR(UDR1)  // 0x3D UDR1
   // 2-word instruction          // 0x3E
   rjmp rx_common_entry           // 0x3F
rx_overflow:
   std Z+18, r1                   // 0x40 working_data->serial_data_status=0;
   std Z+19, r1                   // 0x41 working_data->receive_counter = 0;
   rjmp rx_epilogue               // 0x42
```

TRITA-ICT-EX-2014:63