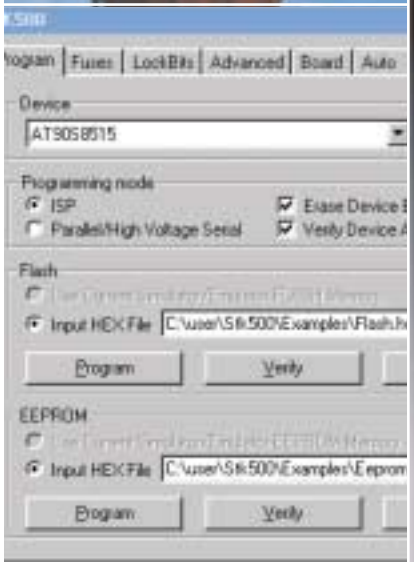
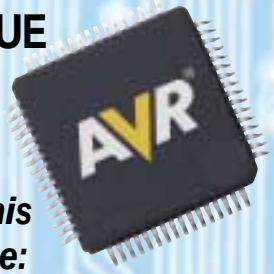
ATMEL APPLICATIONS JOURNAL

Number 1 • Summer, 2003

JOURNAL



Special AVR Microcontrollers ISSUE



In this issue:

ATMEL

Our AVR microcontroller is probably 12 times faster than the one you're using now.

(It's also smarter.)

Introducing the Atmel AVR.® An 8-bit MCU that can help you beat the pants off your competition. AVR is a RISC CPU running single cycle instructions. With its rich, CISC-like instruction set and 32 working registers, it has very high code density and searingly fast execution–up to 16 MIPS. That's 12 times faster than conventional 8-bit micros. We like to think of it as 16-bit performance at an 8-bit price.

With up to 128 Kbytes of programmable Flash and EEPROM, AVR is not only up to 12 times faster than the MCU you're using now. It's probably 12 times smarter, too.

And when you consider that it can help slash months off your development schedule and save thousands of dollars in project cost, it could make you look pretty smart, too.

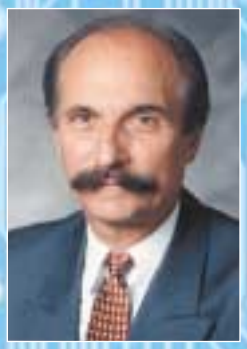
AVR comes in a wide range of package and performance options covering a huge number of consumer and industrial applications. And it's supported by some of the best development tools in the business.

So get your project started right. Check out AVR today at www.atmel.com/ad/fastavr. Then register to qualify for your free evaluation kit and bumper sticker. And get ready to take on the world.

Check out AVR today at www.atmel.com/ad/fastavr

AVR 8-bit RISC Microcontrollers		Memory Configurations (Bytes)			Debug and
Processor	Package	Flash	EEPROM	RAM	Development Tools
tinyAVR™	8-32 pin	1-2K	up to 128	up to 128	Available Now
low power AVR	8-44 pin	1-8K	up to 512	up to 1K	Available Now
megaAVR®	32-64 pin	8-128K	up to 4K	up to 4K	Available Now

ATMEL®



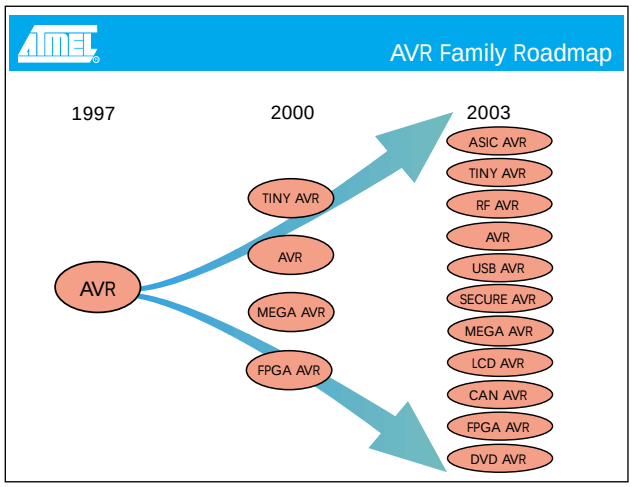
Jim Panfil,
Director of Marketing

Welcome to the AVR Edition of the Atmel Applications Journal

By Jim Panfil, Director of Marketing

The AVR Microcontroller introduced by Atmel in 1997 is the only new 8-bit Architecture launched in the last two decades. During the first six years of production, the AVR attracted twenty thousand new customers because of its unique in-system programmable Flash program memory and price performance.

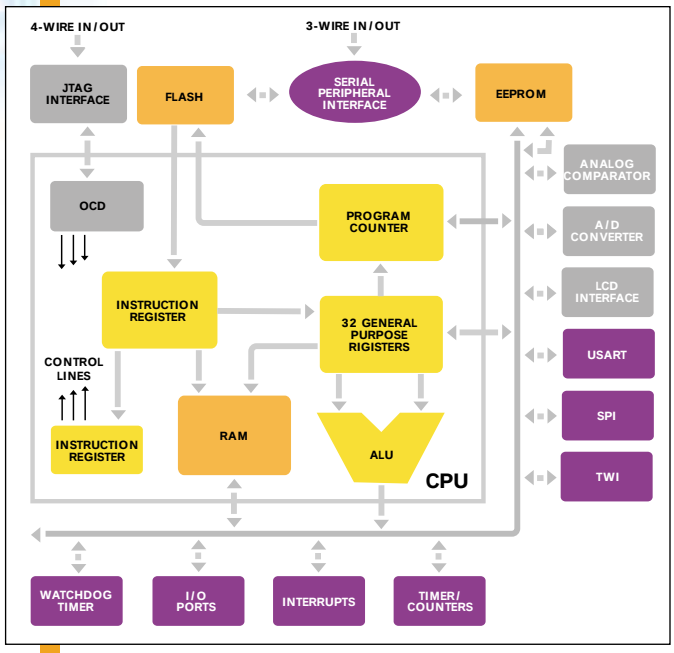
The success of the AVR Architecture is due in part to the large and growing number of partners and suppliers offering products and services that reduce development and programming time and cost. Developers using the AVR in their next application require all manner of support including design services; help in using new tools, compilers and the ability to communicate and share lessons learned with other engineers who encountered and solved similar problems. In addition to the application notes on the Atmel web site and AVR Freaks.com users forum, we now have our own publication-- the Atmel Applications Journal. Here is the charter issue, dedicated to the AVR Microcontroller.



AVR Roadmap

The Mega AVR Family has a unique Self-Programming Memory and Read while Write capability. This is a break through technology that enables new applications and provides the user a significant cost reduction by eliminating additional circuitry including a second CPU to implement remote programming capability. Today the AVR is more than a solution supporting the general-purpose marketplace. The product family now integrates unique peripheral functions that provide solutions for specialized markets. Examples are the portable appliances, wireless communication, security and PC segments.

Atmel recently introduced four new application specific extensions to the AVR Family. The Mega 169 is the first member of a family of devices with an integrated LCD controller. Typical power consumption is less than 20uA at 32KHz operation. It targets battery-powered applications like residential thermostats, portable medical instruments and cordless communication devices.



AVR Chip Diagram

The AT86RF401 SmartRF processor combines the AVR with a high performance RF transmitter operating at 250 – 460MHz.

It is targeted at cost sensitive wireless remote control markets such as auto keyless entry, garage door openers and home convenience controls. The AT97SC3201 Trusted Computing Platform Module brings affordable hardware security to the PC Platform. It consists of an AVR with sophisticated tamper detection circuitry designed to be mounted on the motherboard of the smallest platforms including pocket PC's and PDA's. AT90SC Secure AVR Family integrates a random word generator, crypto coprocessor and on chip security to enable GSM SIM card, Internet transaction, pay TV and banking designs. These smart card applications require high performance to perform encryption functions in real time. There are six members of the USB AVR product offering. The AT43USB351M is the only configurable low / high speed USB controller supporting five end points. With integrated 12 channel 10-bit A/D capability, it supports video game controllers, data acquisition devices, and sensors and mass storage applications.

Based on unprecedented market acceptance and customer demand, Atmel created two new AVR design teams. One is located in Nantes, France, the other in Helsinki, Finland. These teams have specialized DSP, analog and communication protocol design skills, which will expand the product portfolio and open new markets for the AVR. To ensure adequate production capacity, we commissioned a new 8-inch wafer Fab in Tyneside, UK. This facility is capable of processing twelve thousand eight-inch wafers per week with geometries as small as 0.13u technology. This new production facility will accelerate cost reduction efforts for both existing and new AVR components. Atmel has made a significant investment in design capability, process technology, plant and equipment to ensure that state of the art AVR solutions are readily available and are cost effective. ❑

ATMEL APPLICATIONS Number 1 • Summer, 2003

JOURNAL

TABLE OF CONTENTS

Departments

Atmel Notes...

First secureAVR Microcontroller with 32Mega-bit Flash

Atmel® Corporation announced that it is sampling a secureAVR™ RISC Microcontroller with 32Mega-bit Flash. This product is based on the AT90SC3232CS (secureAVR processor, 32Kbytes Flash, 32Kbytes EEPROM) with in addition 32Mega-bit of Flash for very efficient and secure data storage. The AT90SC3232CS-F32M is a unique and innovative solution that combines programmability and processing power with a very large Flash memory.

With the AT90SS3232CS-F32M, mobile communication operators have access to a powerful product for emerging applications with an important secure memory in a SIM card. For other applications, wherever any system requires a large amount of data to be protected, the AT90SC3232CS-F32M provides a highly secure, high memory capacity solution.

The AT90SC3232CS-F32M offers all of the AT90SC3232CS features, a flexible secure microcontroller using Flash program memory to satisfy a user's code modification requirements. For example, its state-of-the-art security features that make it resistant against the most aggressive hardware or software attacks and its powerful cryptography capabilities (Elliptic Curves hardware support, fast DES/TDES and fast RSA processor). It is designed to meet Common Criteria EAL4+ security certification. The AT90SC3232CS-F32M is available in a 3V version, as well as 5V, and can be delivered in module form or in package form.

Herve Roche, Smart Card IC Marketing Manager stated, "The content protection barrier is being surpassed. The AT90SC3232CS-F32M is the industry's first high-end secure microcontroller with this type of large Flash memory capacity. The other great performance is the availability of this outstanding product for the smart card market with deliveries in module form. For example the ITSO's (Interoperable Transport Smartcard Organization) secure access module has been developed with the AT903232CS-F32M in a standard SIM format". The AT90SC3232CS-F32M is available in engineering samples. The price for 1000 units is $20. ❑

IAR visualSTATE® for AVR tool entry updated

Automated design and code generation with IAR visualSTATE for AVR:

The first state machine graphical design tool generating highly compact C code for embedded systems
- Automatic generation of C/C++ code from state machine models
- Automatic generation of full documentation
- Intuitive, easy to use graphical editor
- Full verification and simulation tools including on-target debug capability using RealLink
- UML compliant state machine standard

IAR visualSTATE allows you to represent your specification/requirement in a graphical state machine model, debug, simulate, document and generate micro-tight C code for your embedded device.

Benefits include:

- Faster development through graphical design of any application
- Rapid implementation of change requests avoiding C code rewrite
- Accurate, structured documentation that is always in-sync with the final design
- Interactively simulate and model your system before committing to hardware
- Embedded applications that are much easier to maintain
- Generate micro-tight embedded C code from graphical design with a single mouse click

A Fully Integrated USB Secure µC Solution in a Single Package

Atmel Corporation is sampling a fully integrated USB Full-Speed secure microcontroller in a PQFP44 package. The AT90SC6464C-USB-I integrated package solution that requires no external clock is based on Atmel's secureAVR(TM) RISC microcontroller. It includes 128Kbytes of on-chip non-volatile memory, powerful cryptography capabilities, a very high level of physical and data security, and a dual interface USB V2.0 Full-Speed interface as well as the standard ISO 7816 smart card interface.

The AT90SC6464C-USB-I targets eTokens used in PC-based secure applications. This package solution can also be embedded in peripherals, set-top boxes, modems, PDAs (Personal Digital Assistants), copyright protection devices and other equipment. Wherever data needs to be protected, the AT90SC6464C-USB-I can provide highly secure and cost-effective solutions in applications such as transactional security, e-mail and network encryption, software and file protection, MP3 and digital camera data storage protection.
The AT90SC6464C-USB-I features a dual communication interface, including both USB V2.0 Full-Speed (12 Mbps) and ISO 7816, for direct connection to either of these popular communication parts. It also incorporates 64Kbytes of on-chip Flash memory and 64Kbytes of EEPROM. Flash program memory gives unrivalled flexibility for new applications with the ability to load or upgrade application code during the production run with no delay.

The AT90SC6464C-USB-I includes all the security features already built into the AT90SC secure microcontroller series. In particular, it provides a 16-bit RISC crypto processor for very efficient execution of the highest-level encryption algorithms, RSA, AES 128/128, SHA-256. In addition, a hardware T-DES (Triple Data Encryption Standard) coprocessor, a true RNG (Random Number Generator) and support for ECC (Elliptic Curve Cryptography) enhance the high cryptography performance of this device.

The AT90SC6464C-USB-I in PQFP44 package is available now in engineering samples. Production quantities are also available at a price of US $4.00 in quantities of 200.000 units. ❑

Atmel Notes... continued

Atmel Extends the 8-bit AVR Product Family

New Devices Serve Wireless,
PC Peripheral and Security Markets

Atmel announced today four new application specific extensions to the AVR Family. The AVR Microcontroller introduced by Atmel in 1997 is the industry's only new 8-bit Architecture launched in the last two decades. During the first six years of production, the AVR has become the design engineers' microcontoller of choice because of its unique in-system- programmable Flash program memory and price performance. Today the AVR represents more than a solution supporting the general-purpose marketplace as the product family now integrates unique peripheral functions that provide solutions for specialized markets. Examples of this integration now support wireless communication, security and PC peripheral segments.

The four new application specific extensions to the AVR Family include the AT86RF401 SmartRF processor, which combines the AVR with a high performance RF transmitter operating at 250 - 460MHz. It is targeted at cost sensitive wireless remote control markets such as auto keyless entry, garage door openers and home convenience controls.

The second application specific area is covered by the AT90SC Family Secure AVR, which integrates a random word generator, crypto coprocessor and on chip security to enable GSM SIM card, Internet transaction, pay TV and banking designs. These smart card applications require high performance to perform encryption functions in real time. There are six members of the USB AVR product offering. The AT43USB351M is the only configurable low / high speed USB controller supporting five end points. With integrated 12 channel 10-bit A/D capability, it supports video game controllers, data acquisition devices, and sensors and mass storage applications.

Finally, the Mega 169 is the first member of a family of devices with integrated LCD controller. Typical power consumption is less than 20uA at 32KHz operation. It targets battery-powered applications like residential thermostats, portable medical instruments and LCD cordless communication devices.

"Based on unprecedented market acceptance and customer demand Atmel created two new AVR design teams located in Nantes, France and Helsinki, Finland in addition to those in San Jose, CA, Rousset, France and Trondheim, Norway to work on further expanding this award winning product portfolio. These teams have specialized skills in analog, DSP and communication protocol design which will open new markets for the AVR." Said Jim Panfil, Director of Microcontroller Products. To ensure adequate production capacity, we commissioned a new 8-inch wafer Fab in Tyneside, UK. This facility is capable of processing ten thousand eight-inch wafers per week with geometries as small as 0.13u technology. We are making a significant investment in design capability, process technology, plant and equipment to accelerate the growth of the AVR portfolio." He added. ❑

Atmel Announces a New Secure Memory Solution for Embedded Applications

Boasting the industry's ONLY family of secure memory devices with data encryption

Atmel announced that its CryptoMemory® product line is now available in plastic packages. These integrated circuits are the industry's only low cost, high security memory chips with data encryption utilizing synchronous protocols for embedded applications. The CryptoMemory family of products, available from 1K bit to 256K bits, fills an industry need for affordable, secure devices for customers who require traditional plastic packages. CryptoMemory's secure nonvolatile EEPROM provides the customer with data security through an authentication protocol, data encryption and tamper protection circuits. A common 2-wire serial interface is used for fast data rate exchanges. This innovative technology, which previously was available in physical forms suitable for only the smart card market, opens up new opportunities for customers who need embedded security at an affordable price.

The CryptoMemory family can be used to secure an endless array of embedded applications, including authenticating individual users who need access to sensitive information and securing data on printed circuit boards, networking systems, PDAs and other electronic equipment. The ability to authenticate OEM subassemblies within a system, including removable storage devices, automotive piece parts, and replaceable components such as printer cartridges, is now an affordable option.

"We are pleased with the technology advancements we have made in our secure portfolio over the last few months. We are well positioned to provide leading edge solutions to the security and embedded markets," said Kerry Maletsky, Business Unit Director for Atmel Corporation. "This new technology secures Atmel's position as the only semiconductor manufacturer to provide an affordable, secure solution for embedded applications."

High volume pricing of these new products in package form range from $.30 to $.85 depending on memory density. ❑

JOURNAL

Publisher: Glenn ImObersteg
glenn@convergencepromotions.com

Managing Editor: Bob Henderson
bob@convergencepromotions.com

Designer: Dave Ramos
dbyd@garlic.com

This Special Issue of the Atmel Applications Journal is published by Convergence Promotions.

An Inexpensive Altitude Deviation Alert

By Larry Rachman, Innovation Design and Solutions, Inc.

Overview

A complex and expensive regulatory approval process for avionics equipment in the US and other countries has created a market for accessories and devices that, since not connected directly to the aircraft, are exempt from the approval process. Such devices are expected to be substantially less expensive than panel-mounted equipment, yet the pilot-user's expectation of low cost is still accompanied by one for high reliability, ease of operation, and the small size expected of a portable device. Our challenge was to provide an altitude deviation alarm that would alert the pilot to an inadvertent climb or descent, possibly into restricted airspace. Given recent global events, such deviations are subject to quick and often extreme

Background

Altitude is most commonly measured by its non-linear relationship to air pressure (Figure 1). At sea level, air pressure is approximately 30 in. Hg (100 kPa), decreasing to approximately 13 in. Hg (45 kPa) at an altitude of 18,000 feet (5500 m). Traditionally altimetry is performed by mechanical means – a sealed air bellows is subject to atmospheric pressure; its dimensional changes are geared to move indicators on a mechanical display, or to operate a Gray-coded encoder wheel providing the information in electronic form. Cost and size limitations would not permit such a solution for this project; instead a monolithic piezoresistive pressure sensor was used. The device is comprised of a sealed vessel containing a known air pressure, one wall of which is a silicon die incorporating a micromachined strain gauge and amplification stage. The package design permits air pressure to be introduced to the other side of the die, causing it to deflect due to the pressure differential. A buffered voltage proportional to the pressure differential is provided as output.

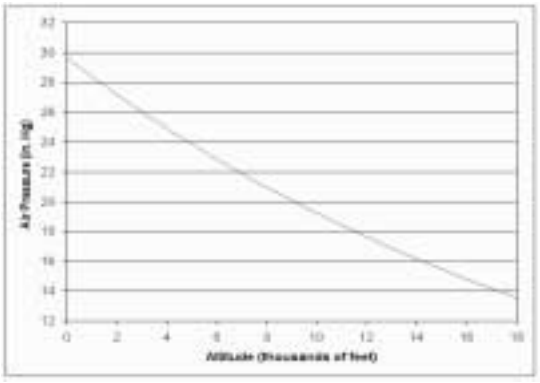


Figure 1

Our goal was to sound an alert upon a deviation as small as 100-200 feet. This required a resolution of 4 millivolts over a range of 2.3 volts, corresponding to an Analog-to-Digital Converter resolution of 10 bits, before allowing for granularity, hysteresis, or low-order-bit ADC noise. Ideally, a resolution of 13-14 bits would be needed to accomplish our goal.

Design Strategy

Our processor selection was the Atmel AVR at90s4433, chosen for size, cost, and low power consumption. The part includes an excellent low-power ADC, but with a resolution of only 10 bits. There are numerous serial ADC devices available in the market, but costs tend to rise above 10-12 bits. With cost, size, and power consumption all factors, we decided to re-examine our design requirements. In normal operation the voltage corresponding to the altitude being measured is static, though it may be static at any point over a range of 2.3 to 4.6 volts. If the voltage were scaled to the point where the required 4 millivolt resolution were measurable with the AVR ADC, it would be out of range for other altitudes.

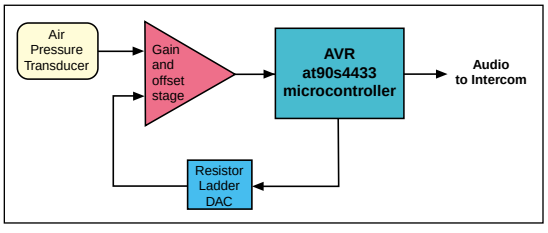


Figure 2

The problem was solved by implementing a 4-bit resistor ladder Digital to Analog Converter driven by the AVR controller. The output of this DAC was used to control the offset of the gain stage between the pressure sensor and the ADC. (Figure 2) This permitted a step selection to be made by the AVR firmware, insuring that the voltage being measured was within the ADC rails. Since only a delta measurement was being made, the DAC could be constructed with relatively low-precision resistors (1%). By choosing offset and gain stage values appropriately, the steps were overlapped, permitting any voltage over the input range to be offset to a point in the center 80% of the ADC range.

When a stable altitude is reached a button press initiates a search for the DAC output step that will cause the ADC input to fall near the center of its range. Based on the step selected, the firmware can compensate for the non-linearity of the relationship between altitude and air pressure. The deviation alert is generated both by LED indicator and an audio tone introduced into the aircraft intercom system. The tone is generated with one of the AVR programmable timers, with a second timer used for both system timing and control of the tone cadence. Different tones indicating climb, descent, return to proper altitude, and low battery (detected by another ADC channel) are generated while the processor spends approximately 99% of its time asleep, saving power.

The application firmware was developed using Atmel AVR Studio, the ICE200, and the Imagecraft AVR Compiler. The AVR's complier-friendliness allowed the application to be implemented in C with code size being only a minor factor. Source-level debugging simplified the development effort and later bug fixes. During debugging, software state and ADC input values were reported to the AVR serial port, simplifying the development process.

A 0.050" (1.27 mm) PCB card-edge pattern was included in the PCB design for the AVR in-system programming connections. The connector also included power and ground connections, permitting one-step production programming of the AVR in-situ, without bed-of-nails ICT fixturing, and with zero additional component cost.

Conclusion

By re-examining our design goals, we were able to utilize the ADC in the at90s4433, where at first this did not seem possible. With the exception of a buffer, the AVR was the only digital part in our design, keeping both cost and size to a minimum. (PCB dimension was 2x2" (50x50 mm) including connectors and the pressure transducer) With a remaining code space of 50%, as well as a pin-compatible growth path in the Mega8, we have an ample growth path for new features. ❑

FOOTNOTES:

1- Several drawings of a conventional barometric altimeter have been made available at http://www.4innovation.biz/altimeter

2- Motorola MPX5100 series, http://www.motorola.com/webapp/sps/site/prod_summary.jsp?code=MPX5100&nodeId=01126990368716

3- Imagecraft, http://www.imagecraft.com/software/ adevtools.html

Novice's Guide to AVR Development

An Introduction intended for people with no prior AVR knowledge.

By Arild Rødland, AVRFreaks

Starting with a new µC architecture can be quite fustrating. The most difficult task seems to be how to get the information and documentation to get the first AVR program up running. This tutorial assumes that you do not yet own any AVR devices or AVR development tools. It also assumes that you have no prior knowledge of the AVR architecture or instruction set. All you need to complete this tutorial is a computer running some flavour of the Windows operating system, and an internet connection to download documents and files.

Preparing your PC for AVR Development

Let's make an easy start, and download the files that we will need later on. First you should download the files to have them readily available when you need them. This could take some time depending on your internet connection.

Download these files to a temporary folder on your computer. (e.g. C:\Temp):

AVR STUDIO 4 (~15MB) 	This file contains the AVR Studio 4 Program. This program is a complete development suite, and contains an editor and a simulator that we will use to write our code, and then see how it will run on an AVR device.
Assembly Sample Code (~1kB) 	This file contains the Assembly Sample code you will need to complete this guide.
AT90S8515 Datasheet (~4MB) 	This is the Datasheet for the AT90S8515 AVR Microcontroller. This is a convenient "Getting Started" device. For now you don't have to worry about the different types of AVR micros. You'll see that they are very much alike, and if you learn how to use one (eg. 8515), you will be able to use any other AVR without any problems.
Instruction Set Manual (~2MB) 	This is the Instruction Set Manual. This document is very useful if you want detailed information about a specific instruction.

When you have downloaded the files, it is time to install the software you need.

Step 2. Installing AVR Studio 4

AVR Studio is also available in a version 3. We will use AVR Studio 4 since this is the version that will eventually replace version 3.

Important note for people using Windows NT/2000/XP:
You must be logged in with administrator rights to be able to successfully install AVR Studio. The reason is that these Windows systems have restrictions regarding who can install new device drivers!

Installation:

1) Double click on the AVRSTUDIO.EXE file you downloaded. This file is a self extracting file, and will ask where you want to extract the files. The default path points to your "default" temp folder, and could be quite well "hidden" on your hard disk, so make sure to read and remember this path, or enter a new path to where you want the files placed (e.g. c:\temp)

2) Once all the files are extracted, open the temp folder, and double click on the SETUP.EXE file. Just follow the installation, and use the default install path. NB: You can use another path, but this tutorial assumes that you install it to the default path.

That's it. Now you have installed all the software you'll need to write code and run programs for all available AVR devices! Keep the Datasheet and Instruction set Manual in a place you remember.

Basic AVR Knowledge

The AVR Microcontroller family is a modern architecture, with all the bells and whistles associated with such. When you get the hang of the basic concepts the fun of exploring all these features begins. For now we will stick with the "Bare Bone" AVR basics.

The 3 different Flavors of AVR

The AVR microcontrollers are divided into three groups:

- tinyAVR
- AVR (Classic AVR)
- megaAVR

The difference between these devices lies in the available features. The ***tinyAVR*** µC are usually devices with lower pin-count or reduced feature set compared to the ***megaAVR's*** . All AVR devices have the same instruction set and memory organization, so migrating from one device to another AVR is easy.

Some AVR's contain SRAM, EEPROM, External SRAM interface, Analog to Digital Converters, Hardware Multiplier, UART, USART and the list goes on.

If you take a ***tinyAVR*** and a ***megaAVR*** and strip off all the peripheral modules mentioned above, you will be left with the AVR Core. This Core is the same for all AVR devices. (Think of Hamburgers: They all contain the same slab of meat, the difference is the additional styling in the form of tripled-cheese and pickles :)

Selecting the "correct" AVR

The morale is that the tinyAVR, AVR (Classic AVR) and megaAVR does not really reflect performance, but is more an indication of the "complexity" of the device: Lot's of features = megaAVR, reduced feature set = tinyAVR . The "AVR (Classic AVR)" is somewhere in between these, and the distinctions between these groups are becoming more and more vague.

So for your project you should select an AVR that only includes the features that you need if you are on a strict budget. If you run your own budget you should of course go for the biggest AVR possible, since eh... because!

Learning to write code on the AVR

Learning new stuff is fun, but can be a bit frustrating. Although it is fully possible to learn the AVR by only reading the datasheet this is a complicated and time-consuming approach. We will take the quick and easy approach, which is:

1. Find some pre-written, working code
2. Understand how this code works
3. Modify it to suite our needs

The device we will use is the AT90S8515 which is an AVR with a good blend of peripherals. Take a few minutes to browse through the Datasheet.

Learning to use the AVR Datasheets

It is easy to get scared when looking at the AVR Datasheets. E.g. the ATmega128(L) datasheet is almost 350 pages long, and reading it start to finish - and remembering the contents, is quite a task. Luckily you are not supposed to do that, either. The datasheets are complete technical documents that you should use as a reference when you are in doubt how a given peripheral or feature works.

OK! You have now installed the software, you have a vague knowledge of the different types of AVRs, and know that there is a lot of information in the datasheet that you don't yet know anything about! Good, now it's time to get developing! Click "Next" to advance to the next part of this tutorial.

When you open an AVR Datasheet you will discover that it can be divided into these groups:

1. First Page Containing Key information and Feature List
2. Architectural Overview
3. Peripheral Descriptions
4. Memory Programming
5. Characteristics
6. Register Summary
7. Instruction Set Summary
8. Packaging Information

This is quite convenient. When you are familiar with how to use the AT90S8515 Datasheet, migrating to another Datasheet should be a breeze. After completing this tutorial you should take some time and read through the Architectural Overview sections of the datasheets (At the beginning of the Datasheets). These sections contain a lot of useful information about AVR memories, Addressing modes and other useful information.

Another useful page to look at is the Instruction Set Summary. This is a nice reference when you start developing code on your own. If you want in-depth information about an instruction, simply look it up in the Instruction Set Manual you previously downloaded!

OK! You have now installed the software, you have a vague knowledge of the different types of AVRs, and know that there is a lot of information in the datasheet that you don't yet know anything about! Good, now it's time to get developing! Click "Next" to advance to the next part of this tutorial.

AVR Studio 4 GUI

Note: If you have not yet installed AVR Studio you should go to the Preparing your PC for AVR Development section of this tutorial before continuing.

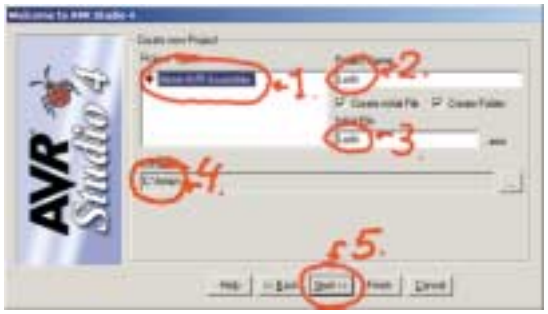
Step 1: Creating a New Project

Start AVR Studio 4 by launching AVR Studio 4 located at [START] | [Programs] | [Atmel AVR Tools]. AVR Studio will start up, and you will get this dialog box.

We want to create a new Project so press the "Create New Project Button"

Step 2: Configuring Project Settings

This step involves setting up what kind of project we want to create, and setting up filenames and location where they should be stored.

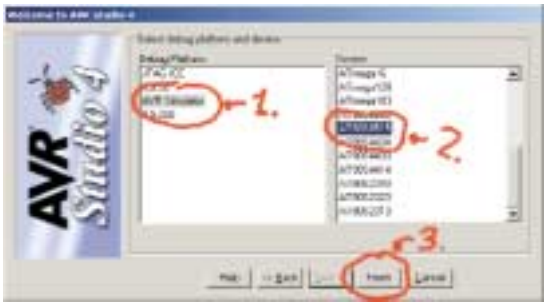


This is done in four steps:

1. Click on this to let the program know you want to create an Assembly program
2. This is the name of the project. It could be anything, but "Leds" is quite descriptive of what this program is going to do
3. Here you can specify if AVR Studio should automatically create a initial assembly file. We want to do this. The filename could be anything, but use "Leds" to be compatible with this tutorial!
4. Select the path where you want your files stored
5. Verify everything once more, and make sure both check-boxes are checked. When you are satisfied, press the "Next >>" button

Step 3: Selecting Debug Platform

The AVR Studio 4 Software can be used as a frontend software for a wide range of debugging tools.



1. AVR Studio 4 supports a wide range of emulation and debugging tools. Since we have not purchased any of these yet, we will use the built in simulator functionality.
2. ..and we want to develop for the AT90S8515 device
3. Verify all settings once more, then press "Finish" to create project and go to the assembly file

Step 4: Writing your very first line of code

AVR Studio will start and open an empty file named Leds.asm. We will take a closer look at the AVR Studio GUI in the next lesson. For now note that the Leds.asm is not listed in the "Assembler" folder in the left column. This is because the file is not saved yet. Write in this line: "; My Very First AVR Project" as shown in the figure below. The semicolon ; indicates that the rest of the line should be treated as a comment by the assembler.

To save the line press - S or select [Save] on the [File] menu. The Leds.asm will now show up in the Left Column as shown below.

OK, Now that we have AVR Studio up and running, it's time to take a closer look at the AVR Studio GUI..

AVR Studio 4 GUI

Let's take a closer look at the AVR Studio Graphical User Interface (GUI).

As you can see below, we have divided the GUI into 6 sections. AVR Studio 4 contains a help system for AVR Studio, so instead of reinventing the wheel here, I'll just explain the overall structure of AVR Studio 4 and point to where in the AVR Studio 4 On-line Help System you can find in depth information.

1. The first line here is the "Menus" Here you will find standard windows menus like save and load file, Cut & Paste, and other Studio specific menus like Emulation options and stuff.
2. The next lines are Toolbars, which are "shortcuts" to commonly used functions. These functions can be saving files, opening new views, setting breakpoints and such.
3. The Workspace contains Information about files in your Project, IO view, and Info about the selected AVR
4. This is the Editor window. Here you write your assembly code. It is also possible to integrate a C-Compiler with AVR Studio, but this is a topic for the more advanced user
5. Output Window. Status information is displayed here.
6. The System Tray displays information about which mode AVR Studio is running in. Since we are using AT90S8515 in simulator mode, this will be displayed here

More about the GUI

To complete this bare bone guide you don't need any more knowledge of the GUI right now, but it is a good idea to take a look at the AVR Studio HTML help system. You can start this by opening [HELP] [AVR Studio User Guide] from AVR Studio, or by clicking this link (and select: Open) if you installed AVR Studio to the default directory. When you have had your fill, we'll continue working on our first AVR Program.

Writing your First AVR Program

At this point you should have installed the software, and started up the a new project called "Leds" You should also have the AT90S8515 Datasheet, stored somewhere you can easily find it. If you can answer "Yes" to both these questions, you are ready to continue writing some AVR Code.

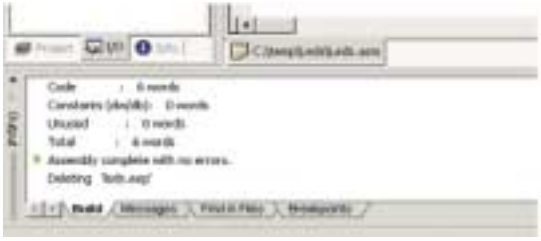
In the Editor view in AVR Studio, continue your program (which at this point only consists of the first line below) by adding the text top of next colum. (Cheaters can simply cut & paste the source below into AVR Studio...)

Sample Code (~1kB)

```
;My Very First AVR Project

.include "8515def.inc"  ;Includes the 8515
                        definitions file

.def Temp = R16         ;Gives "Defines" Register
                        R16 the name Temp

.org 0x0000             ;Places the following code
                        from address 0x0000
   rjmp  RESET          ;Take a Relative Jump to the
                        RESET Label

RESET:                  ;Reset Label
   ldi Temp, 0xFF       ;Store 255 in R16 (Since we
                        have defined R16 = Temp)
   out DDRB, Temp       ;Store this value in The
                        PORTB Data direction
                        Register

Loop:                   ;Loop Label
   out PORTB, Temp      ;Write all highs
                        (255 decimal) to PORTB
   dec Temp             ;Decrement R16 (Temp)
   rjmp Loop            ;Take a relative jump to the
                        Loop label
```

Note that the source code changes color when written in the editor window. This is known as syntax highlighting and is very useful make the code more readable. Once the Source code is entered, press CTRL + F7 or select [Build and Run] from the [Project] Menu.

In the output view (at the bottom left of the screen) you should get the following output indicating that the Project compiled correctly without any errors! From this output window, we can also see that our program consists of 6 words of code (12 bytes).



Congratulations!! You have now successfully written your first AVR program, and we will now take a closer look at what it does!

Note: If your program does not compile, check your assembly file for typing errors. If you have placed the include files (8515def.inc) in a different folder than the default, you may have to enter the complete path to the file in the .include "c:\complete path\8515def.inc" statement. When it compiles we will continue explaining and then debugging the code.

At this point you should have installed the software, and started up the a new project called "Leds" You should also have the AT90S8515 Datasheet, stored somewhere you can easily find it. If you can answer "Yes" to both these questions, you are ready to continue writing some AVR Code.

I guess you have figured out what our masterpiece is doing. We have made a counter counting down from 255 to 0, but what happens when we reach zero?

Understanding the Source Code

OK so the code compiled without errors. That's great, but let us take a moment to see what this program does, and maybe get a feeling how we should simulate the code to verify that it actually performs the way we intended. This is the complete source code:

Sample Code

```
;My Very First AVR Project

.include "8515def.inc" ;Includes the 8515 defini-
                       tions file

.def Temp = R16        ;Gives "Defines" Register R16
                       the name Temp

.org 0x0000            ;Places the following code
                       from address 0x0000
   rjmp  RESET         ;Take a Relative Jump to the
                       RESET Label

RESET:                 ;Reset Label
   ldi Temp, 0xFF      ;Store 255 in R16 (Since we
                       have defined R16 = Temp)
   out DDRB, Temp      ;Store this value in The
                       PORTB Data direction Register

Loop:                  ;Loop Label
   out PORTB, Temp     ;Write all highs
                       (255 decimal) to PORTB
   dec Temp            ;Decrement R16 (Temp)
   rjmp Loop           ;Take a relative jump to the
                       Loop label
```

Now let's take a line-by-line look at what's going on in this code.

;My Very First AVR Project

Lines beginning with " ; " (semicolon) are comments. Comments can be added to any line of code. If comments are written to span multiple lines, each of these lines much begin with a semicolon

.include "8515def.inc"

Different AVR devices have e.g. PORTB placed on different location in IO memory. These .inc files maps MNEMONICS codes to physical addresses. This allows you for example to use the label PORTB instead of remembering the physical location in IO memory (0x18 for AT90S8515)

.def Temp = R16

The .def (Define) allow you to create easy to remember labels (e.g. Temp) instead of using the default register Name (e.g. R16). This is especially useful in projects where you are working with a lot of variables stored in the general purpose Registers (The Datasheet gives a good explanation on the General Purpose Registers!)

.org 0x0000

This is a directive to the assembler that instructs it to place the following code at location 0x0000 in Flash memory. We want to do this so that the following RJMP instruction is placed in location 0 (first location of FLASH). The reason is that this location is the Reset Vector, the location from where the program execution starts after a reset, power-on or Watchdog reset event. There are a also other interrupt vectors here, but our application does not use interrupts, so we can use this space for regular code!

rjmp RESET

Since the previous command was the .org 0x0000, this Relative Jump (RJMP) instruction is placed at location 0 in Flash memory, and is the first instruction to be executed. If you look at the Instruction Set Summary in the Datasheet, you will see that the AT90S8515 do not have a JMP instruction. It only has the RJMP instruction! The reason is that we do not need the full JMP instruction. If you compare the JMP and the RJMP you will see that the JMP instruction has longer range, but requires an additional instruction word, making it slower and bigger. RJMP can reach the entire Flash array of the AT90S8515, so the JMP instruction is not needed, thus not implemented.

RESET:

This is a label. You can place these where you want in the code, and use the different branch instructions to jump to this location. This is quite neat, since the assembler itself will calculate the correct address where the label is.

ldi Temp, 0xFF

Ah.. finally a decent instruction to look at: Load Immediate (LDI). This instruction loads an Immediate value, and writes it to the Register given. Since we have defined the R16 register to be called "Temp", this instruction will write the hex value 0xff (255 decimal) to register R16.

out DDRB, Temp

Why aren't we just writing "ldi DDRB, Temp"? A good question, and one that require that we take a look in the Instruction Set Manual. Look up the "LDI" and "OUT" instructions. You will find that LDI has syntax : "LDI Rd, K" which means that it can only be used with General Purpose Registers R16 to R31. Looking at "OUT" instruction we see that the syntax is "OUT A, Rr" Which means that the content that is going to be written by the OUT instruction has to be fetched from one of the 32 (R0 to R31) General Purpose Registers.
Anyway, this instruction sets the Data Direction Register PORTB (DDRB) register to all high. By setting this register to 0xFF, all IO pins on PORTB are configured as outputs.

Loop

Another label...

out PORTB, Temp

We Now write the value 0xFF to PORTB, which would give us 5V (Vcc) on all PORTB IO pins if we where to measure it on a real device. Since the IO ports is perhaps the most used feature of the AVR it would be a good idea to open the Datasheet on the PORTB. Notice that PORTB has 3 registers PORTB, PINB and DDRB. In the PORTB register we write what we want written to the physical IO pin. In the PINB register we can read the logic level that is currently present on the Physical IO pin, and the DDRB register determines if the IO pin should be configured as input or output. (The reason for 3 registers are the "Read-Modify-Write" issue associated with the common 2 register approach, but this is a topic for the Advanced class.)

dec Temp

This Decrement (DEC) instruction decrements the Temp (R16) register. After this instruction is executed, the contents of Temp is 0xFE. This is an Arithmetic instruction, and the AVR has a wide range of Arithmetic instructions. For a complete listing of available instruction: Look in the Instruction Set Summary in the Datasheet!

rjmp Loop

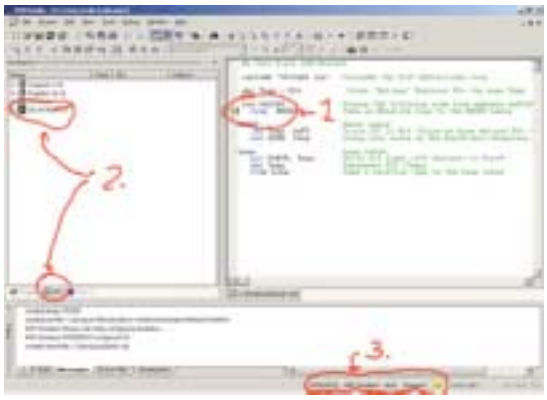
Here we make a jump back to the Loop lable. The program will thus continue to write the Temp variable to PORTB decrementing it by one for each loop.

I guess you have figured out what our masterpiece is doing. We have made a counter counting down from 255 to 0, but what happens when we reach zero?

Simulating with the Source Code

AVR Studio 4 operates in different "modes". Back when we where writing the code, we where in editor mode, now we are in debugging mode. Lets take a closer look at these:

1. Note that a Yellow arrow has appeared on the first RJMP instruction. This arrow points to the instruction that is about to be executed.
2. Note that the workspace has changed from Project to IO view. The IO view is our peek-hole into the AVR, and it will probably be your most used view. We will look closer at this one in a short while.
3. The bottom line contains status information. This Reads: AT90S8535 Simulator, Auto, Stopped. This is followed by a yellow icon. It is a good idea to check this information to verify that you have selected the correct device and emulation tool.

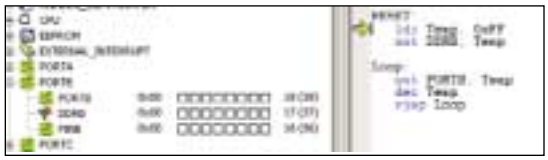


Setting up the IO View

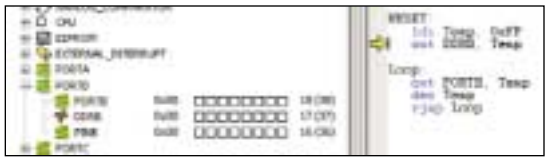
Since our program mainly operates on PORTB registers, we will expand the IO view so that we can take a closer look at the contents of these register. Expand the IO view (tree) as shown in the figure on left:

Stepping through the Code

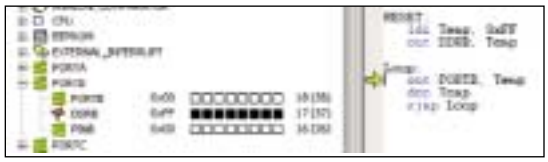
AVR Studio allows running the code at full speed until a given point, and then halt. We will however take if nice and slow, and manually press a button for every instruction that should be executed. This is called single-stepping the code.



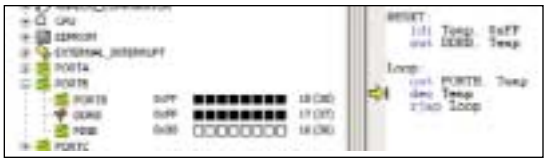
Press [F11] once. This is the key for single-stepping. Note that the yellow arrow is now pointing at the **LDI Temp, 0xFF** instruction. This is the instruction that is going to be executed next.



Press [F11] once more. The **LDI** instruction is executed, and the arrow points to the OUT instruction. The Temp Register has now the value 0xFF. (If you open the "Register 16-31" tree you will see that R16 contains 0xFF. We defined Temp to be R16, remember?)

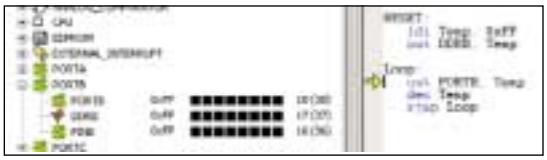


Press [F11]. DDRB is now 0xFF, As shown in the IO View above this is represented as black squares in the IO View. So, a white square represents logical low "0" and black squares are logical high "1". By setting DDRB high, all bits of PORTB is configured as outputs.

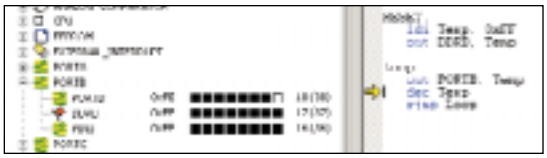


Press [F11]. 0xFF is now written to PORTB register, and the arrows points to the DEC instruction. Note that PORTB is equal to 0xFF. Note also that the PINB register is still 0x00!

Press [F11]. The Temp variable is decremented (0xFF - 1 = 0xFE). In addition the PINB register changes from 0x00 to 0xFF! Why? To find out why this happens you have to look at the PORT sections of the datasheet. The explanation is that the PORTB is first latched out onto the pin, then latched back to the PIN register giving you a 1 clock cycle delay. As you can see, the simulator behaves like the actual part! The next instruction is a relative jump back to the Loop label.



Press [F11]. The RJMP is now executed, and the arrow is back pointing at the OUT PORTB, Temp instruction.



Press [F11] to write the new Temp value to the PORTB register. Note that the content of PORTB is now updated to 0xFE! Continue pressing F11 until you have counted down the PORTB register to 0x00. What happens if you continue running the Program?

Conclusion and Recommended Reading

After running through this introduction you should have a basic idea of how to get a program up and running on the AVR µC.

As mentioned before, one of the most efficient methods of learning AVR programming is looking at working code examples, and understanding how these work. Here on AVRfreaks.net you will find a large collection of projects suitable to learn you more about the AVR.

In our tools section we have also linked up all Atmel AVR Application Notes. These are also very useful reading.

❑

After running through this introduction you should have a basic idea of how to get a program up and running on the AVR µC.

Basic Interrupts and I/O

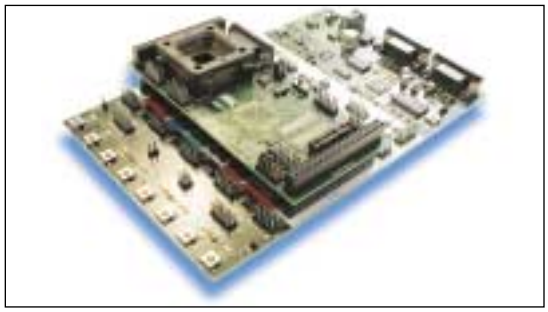
an introduction to interrupts and I/O with the AVR

By Eivind Sivertsen, AVRFreaks

This article is a small project for you people who are just getting into the AVR, and perhaps even microcontrollers in general.

Lets' get physical

The natural place to start is the **STK500**. It is a very nice development board for the AVR, reasonably priced (~USD79) and provides all the environment we need to test some pretty real applications on the AVR out in the wild.



We're gonna start out with some simple counting controlled by external interrupts and exposed on the nice LEDs of the STK500. Then we'll add a speaker (Oh yeah!), and before we know it we'll have a miniature amusement park on our desks; with lights AND noise and buttons to push! Perhaps fire as well, if something REALLY goes wrong.
This is what we'll use:

1. **AVRstudio** 3 or 4
2. **STK500** development kit, all set up with your computer and ready to go
3. An **AT90s8515** microcontroller (usually comes with the STK500)
4. Some small **speaker** that works, including wires soldered in place

The setup is based on a Windows configuration, but it is very possible use some other software as well, since we won't concentrate much on the use of AVRstudio besides assembling the project. If you are a Linux user, you could use:

* avr-gcc (i.e. avrasm) for the assembly
* uisp for programming

The program will be written in **assembly**, because:

* assembly is very "machine-near" and provides a very educative approach to what goes on inside the processor during our program
* high-level languages and different compilers all have different notations and routines for doing the same thing. Learning a compiler and the respective C-style (e.g.) is a story of itself.

The code for this project is something we found among leftovers from O'Guru *Sean Ellis;* which we brutally and without due respect ripped apart. Shame on us.

Basic interrupts

An interrupt is a flow control mechanism that is implemented on most controllers, among them the AVR. In an MCU application interacting with the outside world, many things are happening at the same time, i.e. not in a synchronized manner, that are to be handled by the microcontroller.

Examples: a switch pressed by the user, a data read on the UART (serial port), a sample taken by the ADC, or a timer calling to say that "time is up!". All these events neeeds to be handled by the MCU.

Instead of polling each instance round-Robin style to ask whether they are in need of a service, we can have them call out themselves when they need attention. This is called "interrupts", since the peripheral device (e.g. a switch pressed) interrupts the main program execution. The processor then takes time out of the normal program execution to examine the source of the interrupt and take the necessary action. Afterwards, normal program execution is resumed.

An interrupt service in other words is just like a subroutine; except that it is not anticipated by the processor to occur at a particular time, since there are no explicitly placed calls to it in the program.

What's in a name?

When reading this article you will from time to time get the feeling that you are confronting a term possibly denoting an actual physical entity or an entity in some sense relevant to the current activity; namely playing around or building serious applications with the AVR...: ***INT0, INT1, GIMSK, PORTB, PB7*** etc...

You are sure to come across such names in any assembly code, Atmel appnote, AVRfreaks Design Note or any posting in the AVRforum.

One might think these are just common names used by individuals accustomed to the jargon, but we will try to use them consciously - in the sense that these names actually denote actual memory locations in the AVR you will be programming.

The mapping of these name to actual memory locations is in the part's **def** file (*def cacros vn unknown term:

8515def.inc **Example snippet;** only a few lines are shown

(~6kB)

```
;***** I/O Register Definitions
.equ    SREG    =$3f
.equ    SPH     =$3e
.equ    SPL     =$3d
.equ    GIMSK   =$3b
 ..      ..      ..
 ..      ..      ..
```

When including this file in the assembly program file, all I/O registernames and I/O register bit names appearing in the data book will be known to the assembler and can be used in the program.

Note that some high-level language compilers may use proprietary terms other than these. But they will have files similar to this def file, defining the memory space of the AVRs. As previously stated; this is another story.

Another document that will prove very useful to anyone working with the AVR, is this document:Manual you previously downloaded!

8515 datasheet

(~2MB)

The datasheet.

The datasheet is the ultimate reference for **any** AVR microcontroller. It even includes an instruction set summary; look up every instruction you don't know when you come across it!

The vector table is reserved for storing interrupt vectors; i.e. locations to jump to when this or that interrupt is calling. This means that each interrupt has a reserved memory location, and when a particular interrupt comes in, the MCU looks in this location to find the address where code that handles this interrupt resides.

In this article, we will be using the 8515. Download this .pdf and keep it close for reference.

You may even want to print it, but think twice. It is **long**.

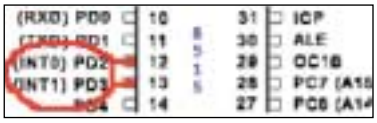
Now you know where to look when anything unknown pops up. Let's move on >.

Structure of an interrupt-driven program on the AVR

Take a deep breath. This is the heaviest part.

We are going to write an "interrupt-driven" program where the main loop simply does nothing but wait for interrupts to occur. What interrupts?

External interrupts INT0 and **INT1** on pins **PD2** and **PD3**



The interrupts are handled in turn, and a return to the main program is performed at the end of each interrupt service (that's what I call it; "service"). This is a rather wide topic with many pitfalls. But we need somewhere to start and will mainly discuss aspects concerning elements of our little example application. The main important thing that constitutes such elements in a program is:

1. Setting the interrupt vector jump locations: **.org**
2. Setting the correct interrupt mask to enable desired interrupts: **GIMSK**
3. Make necessary settings in control registers: **MCUCR**
4. Globally enable all interrupts: **SREG**

Setting the interrupt vector jump locations: .org

The lowest part of the AVR program memory, starting at address $0000, is sometimes referred to as the *"Program memory vector table"*, and the actual program should start beyond this space.

The vector table is reserved for storing interrupt vectors; i.e. locations to jump to when this or that interrupt is calling. This means that each interrupt has a reserved memory location, and when a particular interrupt comes in, the MCU looks in this location to find the address where code that handles this interrupt resides.

8515 Vector table — **Example**; only the few first vectors are shown

Program memory address	Vector	Comment
$0000	Reset	Start address of Reset handler is stored here
$0001	INT0	Start address of code to handle external INT0 is stored here
$0002	INT1	Start address of code to handle external INT1 is stored here
etc...	...	...

The number of interrupts available varies from processor to processor.

The .org directive

In assembly code, the .org directive is used to set vector jump locations. This assembler directive (or "command", if you like) tells the assembler to set the location counter to an absolute value. It is not part of the AVR instruction set, it is just a command that the assembler needs to make sure the program code is mapped correctly when making a binary for the AVR.

Example:

Sample Code

```
; Interrupt service vectors
; Handles reset and external interrupt vectors INT0
  and INT1

.org $0000
    rjmp Reset  ; Reset vector (when the MCU is reset)

.org INT0addr
    rjmp IntV0  ; INT0 vector (ext. interrupt from
                pin PD2)

.org INT1addr
    rjmp IntV1  ; INT1 vector (ext. interrupt from
                pin PD3)

; - Reset vector - (THIS LINE IS A COMMENT)
Reset:
    ldi         TEMP,low(RAMEND) ; Set initial stack
                ptr location at ram end
    out         SPL,TEMP
    ldi         TEMP, high(RAMEND)
    out         SPH, TEMP
    ...
    ...
```

Note that **labels** are used instead of absolute numbers to designate addresses in assembly code - The assembler stitches it all together in the end. All we need to do is tell the assembler where to jump when e.g. the **reset vector** is calling, by using the name of the code block meant for handling resets.

A **label** denotes a block of code, or *function* if you like; which is not terminated with a *"}"*, an *.endfunc* or anything like that. The only thing that ends a code block definition, is it being released by another block name, followed by a colon (":").

This also implies, unlike with functions in e.g. C, that all blocks are run by the processor consecutively, unless the flow is broken up by un/conditional jumps, returns, interrupts etc. In assembly, *the whole file* is the *main()* function, and the flow control is more like Basic...

Please also note the first lines of the **reset handler**. This is where the stack is set up. The **stack** is used to hold return addresses in the main program code when a sub- or interrupt routine is run; i.e. when a "digression" from the main program is made. For any interrupt service or subroutine to return to the main program properly; the stack must be placed outside their vector space. The **SP** is namely initialized with the value **$0000**, which is the same location as the reset vector. This goes for **any** program, especially such as this, where we are involving several interrupt vectors besides the reset vector.

For AVRs with more than 256 bytes SRAM (i.e. none of the Tinys, nor 2343 and 4433), the Stack Pointer register is two bytes wide and divided into **SPL** and **SPH** (low and high bytes).

Setting the interrupt mask: GIMSK

The GIMSK register is used to enable and disable individual external interrupts.

GIMSK — **General Interrupt Mask register**

Bit	7	6	5	4	3	2	1	0
	INT1	INT0	-	-	-	-	-	-
Read/write	R/W	R/W	R	R	R	R	R	R
Init. value	0	0	0	0	0	0	0	0

Note that only the INT0 and INT1 bits are writable. The other bits are reserved and always read as zero.

We are going to use the external interrupts INT0 and INT1 for the switches on the STK500. These interrupts are enabled by setting INT0 and INT1 in GIMSK; i.e. bits 6 and 7.

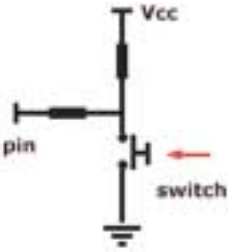
MCUCR					**MCU general control register**			
Bit	7	6	5	4	3	2	1	0
	ISRE	SRW	SE	SM	ISC11	ISC10	ISC01	ISC00
Init. value	0	0	0	0	0	0	0	0

The bits in MCUCR allow general processor control.
Consult the datasheet for an in-depth description of the registers and the individual bits.

We will be using bits 0,1,2 and 3 in this register to control the interrupt from INT0 and INT1. These bits control how to sense the external interrupts; either by level, falling edge on pin, or rising edge of pin:

ISCx1	ISCx0	Description
0	0	Low level on INTx pin generates interrupt
0	1	Reserved
1	0	Falling edge on INTx pin generates interrupt
1	1	Rising edge on INTx pin generates interrupt

We will use the **rising edge** of the switches on the STK500 to trig the interrupt; so the 8515 must be programmed to trig external interrupts on rising edges of each pin PD2 and PD3. Hence; all the ISCx bits must, for our program, be set to "1".



You can see on the diagram to the right how pushing the switch will close the lower branch and pull the pin low. Hence; releasing the switch causes a rising edge when the branch is re-opened and the pin is pulled high.

Globally enable all interrupts: SREG

In addition to setting up the interrupts individually, the SREG (Status Register) bit 7 must also be set to globally enable all (i.e. any) interrupts.

SREG						**S tatus register**		
Bit	7	6	5	4	3	2	1	0
	I	T	H	S	V	N	Z	C
Init. value	0	0	0	0	0	0	0	0

The bits in SREG indicate the current state of the processor.

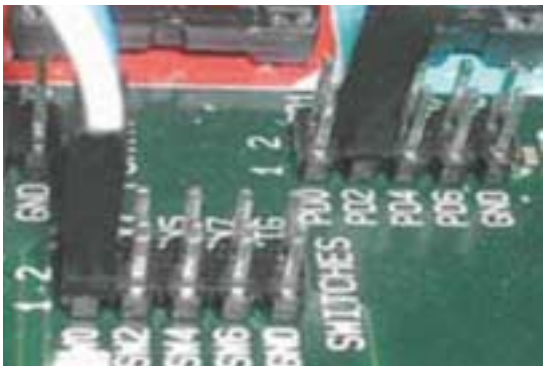
All these bits are cleared on reset and can be read or written by a program. **Bit7 (I)** is the one we are currently interested in; as setting this bit enables all interrupts. Vice versa, resetting it disables all interrupts.

In AVR code, we have an instruction of its own to set this flag; ***sei***:

```
    ; lots and lots of initialisation, and then...

sei ; this instruction enables all interrupts.
    ;...and off we go!
```

Real code part 1

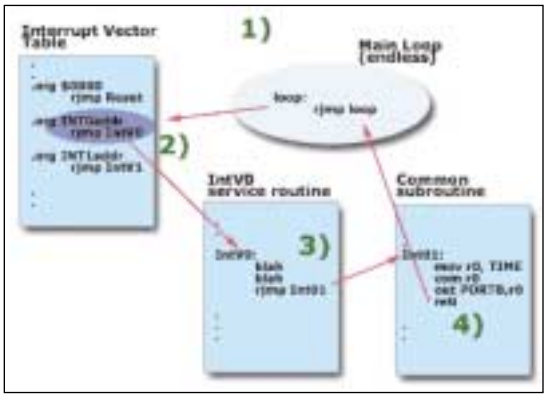
OK, let's start with the real code. Assuming you're already able to assemble your own code and even program the part in the STK500 - we'll just dig through the code.



Just remember to couple the switches with the appropriate inputs on the 8515; namely PORTD input pins **PD2** and **PD3**. Use any two switches on the STK500 you like; on the picture to the right I used switches **SW0** and **SW1**. Also connect the LEDs to PORTB with a 10-pin ISP connector cable.

When running this code on the STK500, at first all LEDs will be off. Press the switches a few times, and you will realize one of them counts something up, the other one down, and the results are reflected on the LEDs.

Let's have an overview of the program. Here's an example snapshot, after initialization:



1. A switch is pressed -> ext. INT0 generated
2. The vector for INT0 is found
3. Code at the according location is run, and jumps to a common subroutine
4. The common subroutine returns to the main loop by reti instruction

This is what our code will do. Nothing more. Besides initialization, the short routine for handling the other switch (generating INT1) and a few directives for the assembler, that's it all.

8515def.inc

(~6kB)

Just to make sure I'm still not kidding you; have a look in the 8515def.inc file and search for "INT0addr" and "INT1addr". Lo and behold; they are real addresses.
Reset is placed at **$0000**.

OK, here is the entire program code, with some excessive comments removed (these are still left in the available file). Look up any unknown instruction for full understanding while you read through it. You can click each code block label to jump to their respective comments next page.

This is what our code will do. Nothing more. Besides initialization, the short routine for handling the other switch (generating INT1) and a few directives for the assembler, that's it all.

OK, here is the entire program code, with some excessive comments removed (these are still left in the available file). Look up any unknown instruction for full understanding while you read through it. You can click each code block label to jump to their respective comments

INTs_1.asm **Source** for first part of program

(~3kB) 

```
;---------------------------------------------------------------------------
; Name:                                 int0.asm
; Title:     Simple AVR Interrupt Verification Program
;---------------------------------------------------------------------------

.include "8515def.inc"

; Interrupt service vectors

.org $0000
      rjmp Reset                        ; Reset vector
.org INT0addr
      rjmp IntV0                        ; INT0 vector (ext. interrupt from pin D2)
.org INT1addr
      rjmp IntV1                        ; INT1 vector (ext. interrupt from pin D3)

;---------------------------------------------------------------------------
;
; Register defines for main loop

.def   TIME=r16
.def   TEMP=r17
.def   BEEP=r18

;---------------------------------------------------------------------------
;
; Reset vector - just sets up interrupts and service routines and
; then loops forever.

Reset:
      ldi    TEMP,low(RAMEND)           ; Set stackptr to ram end
      out    SPL,TEMP
      ldi    TEMP, high(RAMEND)
      out    SPH, TEMP

      ser    TEMP                       ; Set TEMP to $FF to...
      out    DDRB,TEMP                  ; ...set data direction to "out"
      out    PORTB,TEMP                 ; ...all lights off!

      out    PORTD,TEMP                 ; ...all high for pullup on inputs
      ldi    TEMP,(1<<DDD6)             ; bit D6 only configured as output,
      out    DDRD,TEMP                  ; ...output for piezo buzzer on pin D6

      ; set up int0 and int1

      ldi    TEMP,(1<<INT0)+(1<<INT1) ; int masks 0 and 1 set
      out    GIMSK,TEMP
      ldi    TEMP,$0f  ; interrupt t0 and t1 on rising edge only
      out    MCUCR,TEMP
      ldi    TIME,$00  ; Start from 0

      sei                               ; enable interrupts and off we go!


loop:
      rjmp   loop                       ; Infinite loop - never terminates

;---------------------------------------------------------------------------
;
; Int0 vector - decrease count

IntV0:
      dec    TIME
      rjmp   Int01                      ; jump to common code to display new count


;---------------------------------------------------------------------------
;
; Int1 vector - increase count

IntV1:
      inc    TIME                       ; drop to common code to display new count

Int01:
      mov    r0,TIME                           ; display on LEDs
      com    r0
      out    PORTB,r0
      reti
```

OK, lets go through the code step by step, though at a pace. It may be easier if you have the source printed out next to you while reading the following comments:

The **first lines** includes the define file for the 8515; thus making all register and I/O names known to the assembler. What happens next is the Interrupt vector table is defined. At **$0000**, the **reset** vector is set up. This is where the 8515 wakes up in the morning - everything is supposed to start from here. Also, the **INT0** and **INT1** vectors are set up, and their handling routines named **IntV0** and **IntV1**, respectively. Look up their labels down the code, and you can see where they are declared.

Following this, registers r16, r17 and r18 have labels put onto them. This is a way to make **variables** in assembly - only we also get to decide where they are placed in memory. Where? In registers r16, r17 and r18... hence; they are all **one byte** wide.

The reset label

Py-haa! The reset label contains all initialization code; this block is run at start-up. The first 4 lines sets up the stack pointer, as mentioned earlier. Note how the *ldi*(load immediate) instruction is used to hold any value temporarily before writing to the actual location by out. *low*() and *high*() are macros returning the immediate values of their arguments, which are memory locations defined in the .def file.

The next six lines sets the *Data Direction Registers* of ports PORTB (used for LEDs) and PORTD (switches). Please check the datasheet under "I/O Ports" for functional descriptions of these registers.
Now, notice this line:

```
ldi     TEMP,(1<<DDD6)
```

This line of code simply (!) means:

"Load TEMP register with a byte value of 1 shifted DDD6 places leftwards".

Ok. Then what is DDD6? From the .def file, we find that this value is 6, and it is meant to point to the 6th bit of the PORTD Data Direction Register. The value loaded into **TEMP** and then into **DDRB**, becomes **01000000** in binary. Hence, the bit in this position in the DDRB register is set.

So what happens? That pin (pin PD6) is to be used for a special twist in the next stage of the program, so that particular pin is set as an **output**; the others will be **inputs**. For now, just notice the notation.

You can probably imagine what happens if you combine such notation in an addition? Well, this is what happens next, when the **GIMSK** register is loaded, and then the **MCUCR**. Please refer to the previous section or the datasheet for a description of these registers and why they are set this way.

Only thing remaining in the **reset** block now, is to call our friend the **sei** instruction for enabling the interrupts we have just set up.

The loop label

The **loop** label simply contains *nothing* but a call to itself. It's an equivalent of writing while(1); in C. After reset is run, the program pointer falls through to the loop block and it will run forever only interrupted by - *interrupts*.

The IntV0 label

This label comtains the handling code for the **INT0** interrupt. Whenever that interrupt calls, this code will be run. It will simply decrement the TIME register. Then it just jumps to a common block called...:

The Int01 label

This block consists of common code that displays the value of TIME (r16) on the LEDs connected to PORTB. Note that the value is inverted by 1's complement (*com* instruction) before written to PORTB, since a low value means no light and vice versa. This block then performs a return to wherever it was called from through the *reti* instruction - which was from the **loop** label.

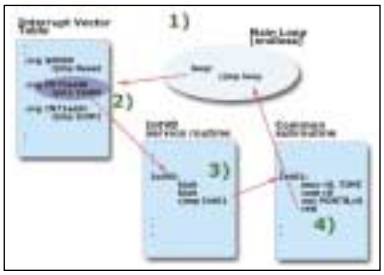
The IntV1 label

You've probably figured that this code runs every time the switch connected to pin PD3 is pressed (i.e. released, due to our MCUCR settings). It increases TIME. Then it *just falls through to the common routine Int01*, since it contains *no jump or return instruction*. We could just as well have put in an

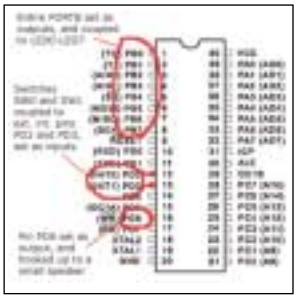
```
rjmp Int01
```

here as well. But we don't need it. Though it may be good common practice to be consequent with this :-)

Now, take a last look at this figure to recollect, before moving.



The timer overflow interrupt



After stating our success with the previous experiment, we are going to ameliorate this little design a little more. We are going to add another interrupt into the application; a *Timer Overflow* interrupt for the **timer/counter 0** of the 8515. Also, we're going to supply a small speaker to make some noise.

If you think the hardware requirements for this project are getting too demanding now, you don't **have** to hook up the speaker. It is for illustrative purposes only, and your code will work perfectly well without it.

Every which way; you will see how to set up the timer overflow interrupt and write handling code for it.

Timer overflow 0

The 8515 has **two** timer/counters; one 8 bits wide and one 16 bits wide. This means that they are capable of counting from any value you set, until they reach their limit which is determined by the number of bits available (256 or 65535, respectively). Then they will issue an interrupt, if you have set it up to do so.

Upon overflow; the Timer/Counter just keeps counting "around" the range... so if you have set the timer to start from some special value and want it to start from there again; you will have to reset it to that value. What we need to do in the code, is to add three little blocks of code more. These are (could you guess them?):

1. Another interrupt vector, for the TimerOverflow 0 interrupt: **OVF0addr**
2. Initialization code for timer/counter 0: **TIMSK, TCCR0,TCNT0**
3. The interrupt **handling subroutine**.

OVF0addr

This is the name set in the **8515def.inc** file for the location where this interrupt vector should reside (check it, I may be pulling your leg). We add these two lines of code to the vector block:

```
; -- new interrupt vector -

.org OVF0addr
        rjmp TimerV0        ; T/C0 overflow vector
```

Py-haa!
The reset label contains all initialization code; this block is run at start-up. The first 4 lines sets up the stack pointer, as mentioned earlier.

...and again; the only thing you really need to know for this little tutorial; is the position of one special little bit: this one is called "Timer/Counter0 Overflow Interrupt enable", abbreviated "TOIE0" and found in bit postition 1 of this register.

You are **very able** to read this now, and realize that it is just like the previous .org's in this program. Let's move on!

TIMSK, TCCR0,TCNT0
Together, these 3 registers are all we need consider to have timing interrupts in an application on the AVR.

TCCR0 controls the operation of Timer/counter 0. The count is incremented for every clock signal at the input of the timer. But the clock input can be selected, and **prescaled** by **N**. We'll just consider the 3 lowest bits of this register:

TCCR0	**Timer/Counter0 register**							
Bit	7	6	5	4	3	2	1	0
	-	-	-	-	-	CS02	CS01	CS00
Read/write	R	R	R	R	R	R/W	R/W	R/W
Init. value	0	0	0	0	0	0	0	0

Note that bits 7-3 are reserved, and always read as zero

This table shows the different settings of these 3 control bits:

CS02	CS01	CS00	Description
0	0	0	Stop the timer/counter
0	0	1	CK
0	1	0	CK/8
0	1	1	CK/64
1	0	0	CK/256
1	0	1	CK/1024
1	1	0	Ext. pin T0, falling edge
1	1	1	Ext. pin T0, rising edge

TIMSK; the Timer/Counter Interrupt Mask register is simply a "mask" register for enabling/disabling interrupts just like you have already seen with the **GIMSK** register:

TIMSK	**Timer/Counter Interrupt Mask register**							
Bit	7	6	5	4	3	2	1	0
	TOIE1	OCIE1A	OCIE1B	-	TICIE1	-	TOIE0	-
Read/write	R/W	R/W	R/W	R	R/W	R	R/W	R
Init. value	0	0	0	0	0	0	0	0

Note that bits 4,2 and 0 are reserved, and always read as zero

...and again; the only thing you really need to know for this little tutorial; is the position of one special little bit: this one is called **"Timer/Counter0 Overflow Interrupt enable"**, abbreviated **"TOIE0"** and found in bit postition 1 of this register. To enable our Timer interrupt; set this bit (to "1"). **TCNT0** is the actual "Timer/Counter" register. This is where the timing and counting is done, in accordance with the settings in **TCCR0**. This is simply a register for storing a counter value; there are no special bits in it. It is entirely readable/writable; so you can load it with any desired starting value for your counting if you like. **Note** that it does not reset itself automatically, even if an interrupt is issued.

This is already becoming old news to you now, since it's just more or less another instance of registers controlling similar functions that you have already heard about regarding the external interrupts... So let's go right to the code>.

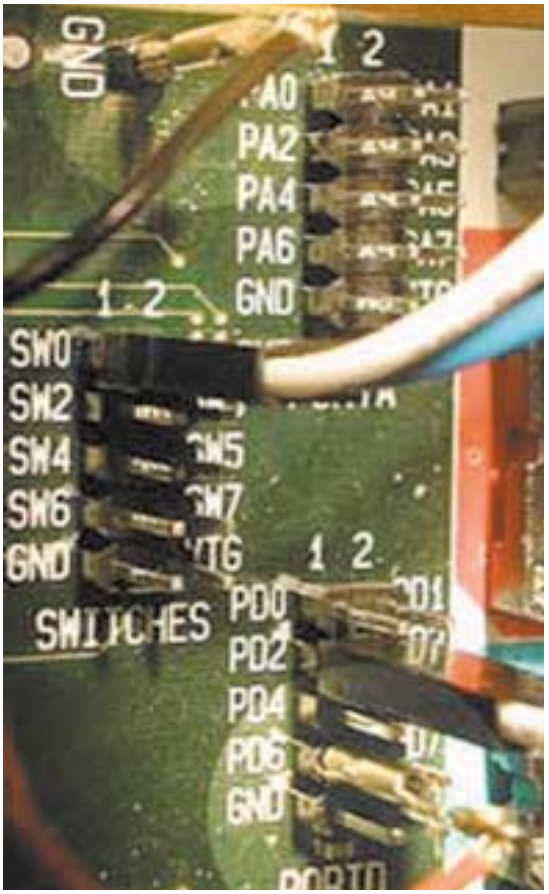
Real code part 2

For illustrating the Timer0 Overflow interrupt; we connect a small speaker to an output pin of the 8515. Each Timer0 overflow interrupt will toggle the pin. The result is that the speaker will buzz with a base frequency proportional to the frequency with which the pin is toggled. I.e. the base frequency will be:

CLK/2*Prescale*(256-TIME)

where TIME is the current value in the TIME register (r16).

Also, the two switches affecting the value in TIME will make the buzz frequency waver up or down.



The picture shows how to connect the speaker. We have chosen pin **PD6** for no particular reason.

Huh? **Why is that the formula for the base frequency?**

?

The Timer/counter counts up from some value **every clock cycle** until it overflows. Then we reset it, to repeat the drill.

Let's say the timer can only count to **1** before overflow. Flipping the pin **every time**, will give us one cycle of a square-wave like waveform **every 2 flips**, right? (up a while, then down a while, repeat...). Hence, the base frequency would be:

CLK/2

Now; the Timer/Counter register is **8 bits wide**, and can count from any value it is set (TIME) to 255. The formula becomes:

CLK/2*(256-TIME)

Besides; we have a prescaler which, when set to **N**, makes the Timer count just every **Nth** cycle...

CLK/2*N*(256-TIME)

These are the three snippets of code to insert. Please consult the complete source code (INTs_2.asm, available below) to where the snippets are inserted:

INTs_2.asm

(~3kB)

Source snippets for second part of program

```
;------------------ CODE SNIPPET #1 - OVF0addr vector -----------------------------
; inserted below the existing vector defs
;--------------------------------------------------------------------------------
.org OVF0addr
       rjmp TimerV0                     ; T/C0 overflow vector

.
.

;------------------ CODE SNIPPET #2 - Initializing TIMSK,TCCR0,TCNT0 -------------
; inserted in the Reset: label, right before the 'sei' call
;--------------------------------------------------------------------------------
       ldi    TIME,$80                  ; Start from 128. NB!
       out    TCNT0,TIME                        ; set Timer/counter also.

       ldi    TEMP,(1<<TOIE0)           ; timer overflow interrupt enable 0
       out    TIMSK,TEMP

       ldi    TEMP,$02                  ; clock prescaler = clk/8
       out    TCCR0,TEMP
.
.
;---------- CODE SNIPPET #3 - handling the Timer overflow int. --
; new subroutine label, inserted at the end of the file
;-------------------------------------------------------------
TimerV0:
       out    TCNT0,TIME                ; reset time

       com    BEEP
       ori    BEEP,$BF                  ; bit 6 only
       out    PORTD,BEEP

       reti                             ; important!
```

;---------- CODE SNIPPET #1 - OVF0addr vector ----------
This part simply declares the Timer Overflow vector address.

;---------- CODE SNIPPET #2 - Initializing TIMSK,TCCR0,TCNT0 ----
First, we set the TIME register to a higher value (0x80 = 128 decimal) for starters, and load it into the Timer/Counter register. It's just a more fitting start value if you run the 8515 on a low CLK freq. Then the relevant interrupt enable bit is set in the **TIMSK** register, and the prescaling bits in the Timer Control register are set to **Prescale=8**.

This way, if the 8515 runs @ **1.23MHz**; the speaker will buzz with a base frequency equal to 1.23E6/2*8*127 = **605.3 Hz**

;---------- CODE SNIPPET #3 - handling the Timer overflow int. --
The important issues in handling this interrupt is:
- Resetting the Timer/Counter - it won't do that itself!
- Flipping the beep pin
- Returning to the program

Resetting Timer/Counter is obviously done by loading the value of TIME (r16) into TCNT0, and returning from the interrupt routine is done by issuing a **reti** instruction.

Flipping the beep pin (PD6) is a little curious, however: This is done by inverting every bit in BEEP (r18) with the **com** instruction, and then OR'ing it with this value 0xbf = 10111111 b (note the 6th position is '0').

Follow the sequence below:

BEEP	00000000
after com:	11111111
'OR' with:	10111111
Result:	11111111
after com:	00000000
'OR' with:	10111111
Result:	10111111
etc...	...

As you may see; whichever value is in BEEP, the 6th bit of it will flip every time... So, the pin will toggle up and down, and the speaker beeps this little song: *"...10101010101010..."*. Haha.

Now, these were the very basic basics of interrupts and I/O. Feel free to experiment with what you have learnt in this article; use other prescaler settings, try other flanks of external interrupt triggering, write programs that use switches to make flow control decisions, whatever...

Good luck!

❑

Now, these were the very basic basics of interrupts and I/O. Feel free to experiment with what you have learnt in this article; use other prescaler settings, try other flanks of external interrupt triggering, write programs that use switches to make flow control decisions, whatever...

Good luck!

Device Drivers and the Special Function Register Hell

By Evert Johansson, IAR Systems

The Atmel megaAVR devices are designed for flexible use with a lot of powerful peripherals which limit the need of external components. These devices are well designed, and the peripherals can be set-up in many different ways to support many different application needs. Because of the flexibility in the microcontroller, it is necessary to set up the pins in the way your specific board is designed, and also to set the operation of the peripherals according to your product needs.

Using the built-in power of the microcontroller

A modern microcontroller has a lot of peripherals, and it is a time-consuming part of each embedded project to write the code needed to use that built-in power and flexibility. It is a tedious work to read the hardware manual and understand how peripheral modules like I/O, timers, USART, etc are implemented, and how the software is to get access to the hardware. Each peripheral is controlled via a number of special function registers where each bit has a special meaning, and many of these bits need to be written and read using a specific protocol.

Atmel megaAVR

The Atmel megaAVR devices are designed for flexible use with a lot of powerful peripherals which limit the need of external components. These devices are well designed, and the peripherals can be set-up in many different ways to support many different application needs. Because of the flexibility in the microcontroller, it is necessary to set up the pins in the way your specific board is designed, and also to set the operation of the peripherals according to your product needs. For instance, the I/O input/output pins are multiplexed with peripheral pins, and need to be initialized according to the hardware implementation.

Application notes

One way to speed up the set up and coding is to use software application notes, which help to use the peripheral. The drawback with application notes is that you do not have the same requirements for your product as the engineer who wrote the application note. Therefore, you need to update the special function register settings manually, and you might also need to modify the application note source code to suit your needs.

Software analysis

If device driver software written for different products is analysed, you will see that most of these drivers are written in much the same way. The reason for this is that the microcontroller is designed in a particular way, and therefore the access to the special function register bits, such as control/status and data bits, needs to be done in a certain way. This actually means that a lot of engineers are writing the same kind of software for different products over and over again. Writing the same kind of software at different places will also need a lot of extra testing to verify that the code runs correctly in the hardware.

The special function register Hell

Microcontrollers include hundreds of special function registers placed at certain addresses in the address space, and it is common that a register is made up of many bitfields or bits. This means that the application needs to access or control thousands of bits, and the access needs to be performed in the way the microcontroller is designed for. Therefore, the productivity for modelling and writing device driver software is normally four times lower than ordinary software coding.

Some registers or bits are both read- and write-accessible, while others are only accessible via read, write, set, or clear. It is also common that registers need to be accessed via a specific protocol. Sometimes the register or bit needs to be read by the software before it can be updated with a write, set, or clear instruction. Some registers are also related to each other, so that one register bit needs to be set or cleared in a register before a bit in the corresponding register can be read or updated.

Get your product to the market quickly

IAR MakeApp for Atmel megaAVR is a tool that guides you through the special function register hell, and helps you with the writing of device drivers. This new low-cost product from IAR Systems includes property dialog boxes which make it easy to configure the megaAVR microcontroller to suit your needs. IAR MakeApp warns you if you try to make a setting that will occupy an already used resource, e.g. the same I/O pin. The product also presents a visual view of the microcontroller and how the pins are configured. Special function register values are calculated automatically according to your settings, and a complete set of device drivers can be generated. The product also includes a component browser and a project report generation function that helps you with the design and documentation.

Device drivers generated by IAR MakeApp

IAR MakeApp contains a powerful code generation technology, and generates a complete set of customized device driver functions according to your project settings. The code generation engine uses the component database information, and automatically calculates the special function register values according to the current property settings. ANSI C source code is generated for each peripheral, and the files are well commented and easy to follow. The drivers include initialization, run-time control, and interrupt handling functions. The functions are ready to be used by your application software and tested with IAR Embedded Workbench for AVR and the Atmel STK500 starter kit. Use IAR MakeApp from Idea to Target. "Click & Go" for driver variants during all phases of your embedded project.

Example: IAR MakeApp USART configuration, code generation, and usage

ATmega128 includes two USART channels for serial communication.

1. Open the USART property dialog box in IAR MakeApp.
2. Make the settings for the channel your hardware is designed for, and make the following minimum selection: Select operating mode, activate USART receive/transmit pins, set baud rate, and define your protocol (number of data bits, parity, and stop bits). Finally, choose if you want to use interrupts.
3. At any time you can view the special function register settings by clicking the Registers button in the property dialog box.
4. The output generation tab in the USART property dialog box includes the device driver functions for USART that will be generated according to your current settings. The device drivers (APIs) for channel 0 normally include the following functions: MA_InitCh0_USART(), MA_ResetCh0_USART(), MA_PutCharCh0_USART(), MA_PutStringCh0_USART(), MA_GetCharCh0_USART(), MA_GetStringCh0_USART(), MA_IntHandler_RX0_USART(), MA_IntHandler_TX0_USART(), MA_IntHandler_UDRE0_USART().

   If your application software will only use some of these functions, you can choose to have only these ones generated by the tool.
5. Click OK to save the settings.

continued on page 40

Variable Message Sign Development with AVR and ImageCraft

by Patrick Fletcher-Jones and Chris Willrich

To launch their new Variable Message Sign products, a leading UK supplier of street lighting and exterior decorative lighting equipment obtained consulting services from Dedicated Controls Ltd. This article is a case study on the design and implementation of this project. By selecting the right hardware and software development tools, this project was finished within 6 months.

Patrick Fletcher-Jones is the principal engineer of Dedicated Controls Ltd, which designs embedded software and electronics systems primarily for industrial control and traffic management systems. Dedicated Controls Ltd. specializes in embedded TCP/IP, radio telemetry, GSM and low power battery operated products. He can be contacted via patrick@dedicated-controls.com

Chris Willrich is the web designer and technical writer of ImageCraft Creations Inc., the producer of the ICCAVR compiler. She can be reached at chris@imagecraft.com

Product Requirements

The design was to have LED dot matrix characters that would be mounted into road traffic information signs. The LED characters would then plug into a controller board, which had the ability to communicate with the traffic control center. Various communication methods such as unlicensed radio bands, cellular phone network and private wire network had to be supported.

The remote signs had to support a level of intelligence such as automatically adjusting the LED character brightness for different viewing conditions including bright sunlight and at night. For product maintenance and support, remote error or fault detection was also needed for detecting communication problems, vandalism, fuse failures etc…

The remote signs needed to be easily configurable, as no two sites were the same. However, to reduce cost and maintenance, the main controller cards needed to be generic, without custom code programmed in for each remote sign.

The signs also had to support a number of different characters; some signs might only have 6 characters where another might have 40. The design of the hardware and software needed a modular approach.

Finally, to allow remote checking of system status and some amount of remote configuration using standard technology, we decided the system should communicate with the control center using TCP and HTML so that a standard Internet browser might be used for these tasks.

Selecting the Hardware and Software

General system architecture was defined where there would be a generic controller card, which supported a serial interface plugged into intelligent expansion cards that would drive the LED characters. Each intelligent expansion card would be uniquely addressable so that multiple expansion cards could sit on the same serial interface bus. It was decided that each intelligent splitter card would support up to 8 characters, so a sign of 8 characters or less would only consist of the generic controller and one intelligent splitter board. 16 characters would only need the addition of another splitter board.

Choosing the processor was fairly simple. Patrick had used the Atmel AVRs successfully on several other projects in the past, and the customer liked the In System Programming (ISP) of the internal flash. (With flash, program updates no longer require swapping out EPROMs or even worse, replacing OTP devices.) The latest flagship AVR device from Atmel is the Mega128 with two serial ports, 128K bytes flash, 4K bytes RAM and 4K bytes EEPROM; it was the perfect choice for the main controller.

The intelligent splitter boards and characters were more cost-sensitive, especially the character cost. This prohibited the use of a processor for each character, but the 8535 seemed the perfect choice for the intelligent splitter with the built-in ADC, IO count, and only needed the addition of a simple connector for the ISP interface.

One of the customer's requirements was the ability to maintain the source code themselves if necessary. There are a few different C compilers available for the Atmel AVR, and ImageCraft ICCAVR came out on top after careful evaluation - for easy of use, code generation quality and support. It was a professional package that did not need a degree in computer science to set up before any code could be compiled. One of the many great features about the ImageCraft tools is the Application Builder, which allowed quick setting up of all the AVR's peripherals. ICCAVR also includes a built in ISP tool which made the whole development process very easy. Another important feature about ImageCraft is its conformance to standard C. Other C compilers have too many unnecessary extensions to the C language, which can make coding seem quicker at first, but the code is then a lot less portable and the source code is then tied to that individual C compiler. Standard C has enough expressiveness for most of Embedded Systems needs, even on an 8 bit CPU such as the Atmel AVR. In places where extensions are needed (for example, writing a function as an interrupt handler), the syntax is clean and even follows the Standard C recommended method of using the #pragma facility. Also, the source code for the library functions within ICCAVR, such as the EEPROM read and write routines, are accessible to the programmer. Other C compilers may provide you with similar functions but they may not allow you to tweak the source code at a C level if required.

On to Development

Now that the processors and development tools had been chosen, the task of product development started. The Atmel development kits STK500 and STK501 provide a great development platform on which 90% of the code could be developed without having to have any custom PCBs made. Using them allowed software development to start with an already known good hardware platform.

To quickly demonstrate to the customer how the system would eventually be set up and configured, Patrick prototyped a terminal driver user configuration interface using the one of the serial ports on the Mega128. Rapid development of the main core software functions was made easy by using the Application Builder within ICCAVR to set up the timers, ADC and UARTs.

After the basic user interface was running, and with a bit of debug information thrown in to make life easier, development of the communications protocol using the second serial port could start. The primary communications medium is unlicensed radio operating on 458MHz at 500mW, so a fully synthesised radio transceiver was used giving over 64 channels to choose from. For initial prototyping and design, the communications protocol was done using the serial connection between the PC and the STK500 development board. Once that was done, the serial connection was replaced with the radio modems.

Gremlins in the Air!

Replacing the simple serial connections with the radio modems immediately introduced new gremlins. Radio preamble was needed; this is where you transmit a number of bytes first, for example;

continued from page 39

GPS-GSM Mobile Navigator

By Ma Chao & Lin Ming

What's the more laudable engineering feat, designing a navigation system capable of tracking ships in Shanghai Port or placing at the top of a competitive design contest? With the award-winning GPS-GSM Mobile Navigator, Ma and Lin accomplished both.

Ma Chao is a professor of Electronic Engineering at East China Normal University in Shanghai, China. He is a specialist in digital image compression and processing, embedded control systems, and computer network systems. You may reach Ma at ma-chao@online.sh.cn.

Lin Ming is a graduate student completing a Master's degree in Electronic Engineering at East China Normal University. He works primarily with embedded systems and microcontroller-based applications. You may reach him at lmcrr@online.sh.cn.

With today's stand-alone global position system (GPS) receivers, you are able to pinpoint your own position. But, what's more useful about stand-alone GPS receivers is that they can transmit your position information to other receivers. We decided to use both of these features to create a wireless vehicle tracking and control system for the Design Logic 2001 Contest, sponsored by Atmel and Circuit Cellar.

To design the Port Navigation System, we combined the GPS's ability to pinpoint location along with the ability of the Global System for Mobile Communications (GSM) to communicate with a control center in a wireless fashion. The system includes many GPS-GSM Mobile Navigators and a base station called the control center.

Let us briefly explain how it works. In order to monitor ships around a port, each ship is equipped with a GPS-GSM Mobile Navigator. The navigator on each ship receives GPS signals from satellites, computes the location information, and then sends it to the control center. With the ship location information, the control center displays all of the ships' positions on an electronic map in order to easily monitor and control their routes. Besides tracking control, the control center can also maintain wireless communication with the GPS units to provide other services such as alarms, status control, and system updates.

Hardware

GPS became available in 1978 with the successful launch of NAVSSTAR 1. Later, in May of 2000, the U.S. government ended selective availability (SA); as a result, the GPS accuracy is now within 10 to 30 m in the horizontal plane and slightly more in the vertical plane. For more information on GPS and its accuracy, read Jeff Stefan's article, "Navigating with GPS" (Circuit Cellar 123).

The GPS-GSM Mobile Navigator is the main part of the Port Navigation System. The design takes into consideration important factors regarding both position and data communication. Thus, the project integrates location determination (GPS) and cellular (GSM)—two distinct and powerful technologies—in a single handset (see Photo 1).

The navigator is based on a microcontroller-based system equipped with a GPS receiver and a GSM module operating in the 900-MHz band. We housed the parts in one small plastic unit, which was then mounted on the ships and connected to GPS and GSM antennas. The position, identity, heading, and speed are transmitted either automatically at user-defined time intervals or when a certain event occurs with an assigned message (e.g., accident, alert, or leaving/entering an admissible geographical area).

This information is received by the system in the dispatching or operations center, where it is presented as a Short Message Service (SMS) message on a PC monitor. SMS is a bidirectional service for sending short alphanumeric (up to 160 bytes) messages in a store√and√forward fashion. If the only data received is time and position, then the data can be displayed on a digitized map and also recorded in a database file; the recorded information can be replayed later for debriefing or evaluation of a mission.

The hardware block diagram is shown in Figure 1. The AT90S8515 microcontroller assures that all of the components work well together; it controls all incoming and outgoing messages as well as the I/O channels, serial interfaces (RS-232), peripheral devices (e.g., LCD and buttons), and all other parts. The GPS module receives the GPS signals and outputs the data to the AT90S8515 microcontroller via a TTL-level asynchronous serial (UART) interface. The microcontroller works with the GSM module by communicating with the GSM network. The interface between the GSM module and AT90S8515 is also TTL async serial. An RS-232 interface is used to exchange data with the PC.

Photo 1—On the front side of the main board, you can see an LCD, four programmable keys, a GSM module, an RS-232 connector, and some other components.

Because the AT90S8151 has only one UART, a three-channel multiplexer is used to switch among three working modes. The location information and other data is stored in the 2-Mb serial data flash memory of the AT45D021. The flash memory stores up to 2160 pieces of location information in 12 h, because the GPS-GSM Mobile Navigator saves GPS signals every 20 s. Four buttons, an LCD, and a buzzer enable you to display the system status and information and control the navigator.

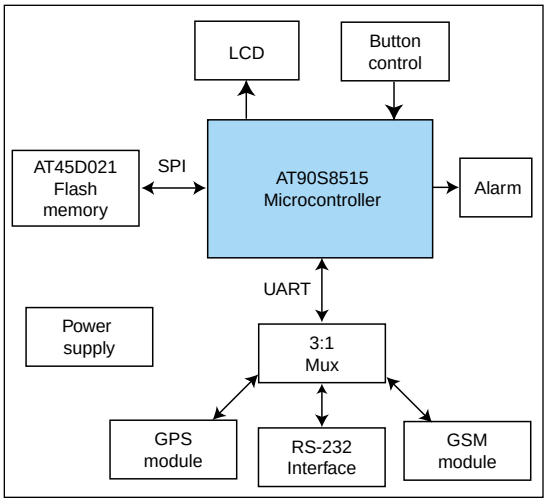


Figure 1—The AT90S8515 microcontroller is the basis for the GPS-GSM Mobile Navigator.

System Features

As we explained, the GPS module outputs the ship location information such as longitude, latitude, and Greenwich Time every 2 s. The location information is then stored every 20 s in flash memory, which has enough power to memorize the track of a ship even when the power is off.

Name	Example	Units	Description
Message ID	$GPRMC	–	RMC protocol header
UTC Position	161229.487	–	hhmmss.sss
Status	A	–	A = data valid; V = data not valid
Latitude	3723.2475	–	ddmm.mmmm
N/S Indicator	N	–	N = north; S = south
Longitude	12158.3416	–	dddmm.mmmm
E/W Indicator	W	–	E = east; W = west
Speed over ground	0.13	Knots	–
Course over ground	309.62	Degrees	True
Date	120598	–	ddmmyy
Magnetic variation	–	Degrees	E = east; W = west
Checksum	*10	–	–
<CR><LF>	–	–	End of message termination

Table 1—The NMEA RMC data values are based on the following example: $GPRMC,161229.487,A,3723.2475,N,12158.3146,W,0.13,309.62,120598,,*10.

Note that the GSM wireless communications function is based on a GSM network established in a valid region and with a valid service provider. Via the SMS provided by the GSM network, the location information and the status of the GPS-GSM Mobile Navigator are sent to the control center. Meanwhile, the mobile navigator receives the control information from the control center via the same SMS. Next, the GPS-GSM Mobile Navigator sends the information stored in flash memory to the PC via an RS-232 interface. (Note that you can set up the navigator using an RS-232 interface.)

There are two ways to use the mobile navigator's alarm function, which can be signified by either a buzzer or presented on the LCD. The first way is to receive the command from the control center; the second way is to manually send the alarm information to the control center with the push of a button.

The GPS-GSM Mobile Navigator is powered by either a rechargeable battery or DC input.

Getting GPS Data

After the GPS module computes the positioning and other useful information, it then transmits the data in some standard format—normally in NMEA-0183 format. When you're building this project, it's nice to be able to buy standalone GPS OEM modules. Just check the pages of Circuit Cellar for manufacturers. We used a Sandpiper GPS receiver from Axiom for this project. The Sandpiper is intended as a component for an OEM product that continuously tracks all satellites in view and provides accurate satellite positioning data. With differential GPS signal input, the accuracy ranges from 1 to 5 m; however, without differential input, the accuracy can be 25 m.

The Sandpiper has two full-duplex TTL-level asynchronous serial data interfaces (ports A and B). Both binary and NMEA initialization and configuration data messages are transmitted and received through port A. Port B is configured to receive RTCM DGPS correction data messages, which enable the GPS unit to provide more accurate positioning information. But, we didn't require the use of port B for this project.

About 45 s after the GPS module is cold booted it begins to output a set of data (according to the NMEA format) through port A once every second at 9600 bps, 8 data bits, 1 stop bit, and no parity. NMEA GPS messages include

Name	Byte	Definition	Description
Start byte	1	:	Start symbol of data package
Data package ID	1	0~9	Package ID is repeated from 0 to 9
System password	3	000~999	System password
Terminal ID	4	0000~9999	Terminal ID
Position data	19	E000000000~E180000000	E means east longitude, which is from 000° and 00.0000 min. to 180° and 00.0000 min.
		N00000000~N90000000	N means north latitude, which is from 00° and 00.0000 min. to 90° and 00.0000 min.
UTC	6	hhmmss	Greenwich Time (hour, minute, second)
Upload time rate	3	001~255(003)	Upload time interval = basic upload time ˇ upload time rate
Alarm information	4	xxxx	0 means OK; 1 means alarm Byte 1: aberrance alarm Byte 2: over-speed alarm Byte 3: dangerous area alarm Byte 4: manual alarm
Stop byte	1	#	Stop symbol of data package

Table 2—Take a look at the 42-byte data package format and the following example ready to be saved: :10019999E121263457N31186429074216003000#.

Reprinted with permission of Circuit Cellar® - Issue 151 February 2003

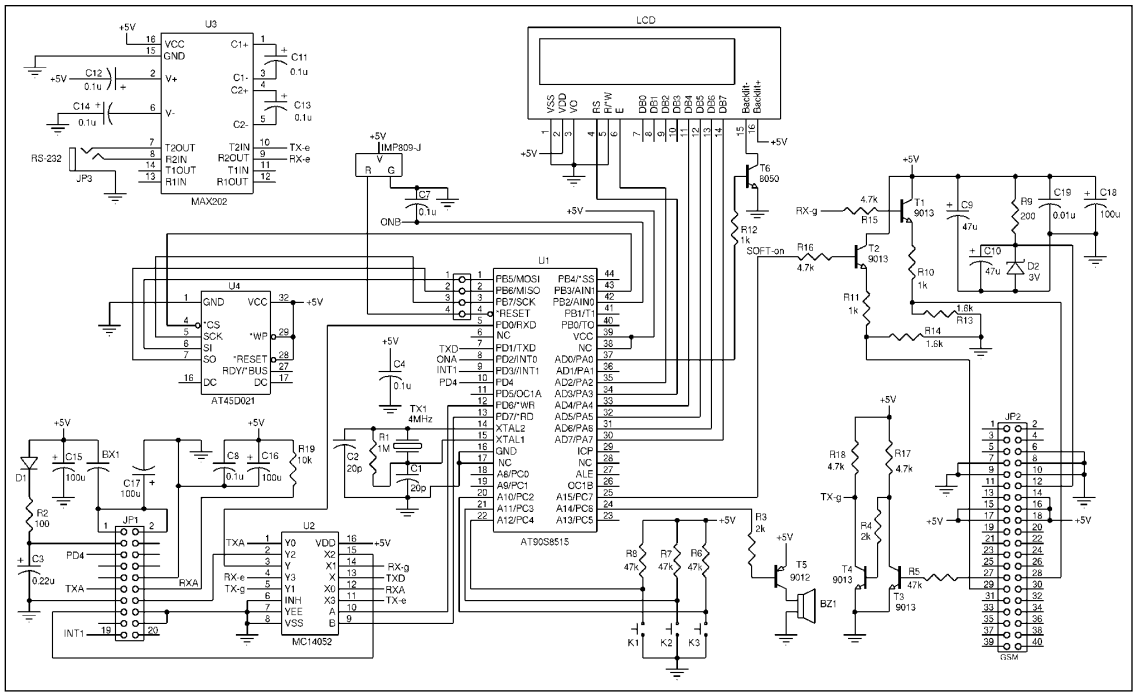


Figure 2—Jack port JP1 is the 20-pin GPS socket header. Jack port JP2 is the 40-pin GSM socket header. U2 is the dual four-channel multiplexer controlled by PA2 through PA3. All of the data traffic runs at 9600 bps.

six groups of data sets: GGA, GLL, GSA, GSV, RMC, and VTG. We use only the most useful RMC message—Recommended Minimum Specific GNSS Data—which contains all of the basic information required to build a navigation system. Table 1 lists the RMC data format.

We only need position and time data, so the UTC position, longitude with east/west indicator, and latitude with north/south indicator are picked out from the RMC message. All of this data will be formatted into a standard fixed-length packet with some other helpful information. Next, this data packet will be transmitted to the control center and stored in the AT45D021's flash memory.

The data packet is a 42-byte long ASCII string, which includes the package ID, system password, terminal ID, position data, UTC, and other operational information. Table 2 shows the definition of a reforming data packet and an example ready to be saved or transmitted.

GSM TRANSMITS DATA

A committee of telecom vendors and manufacturers in Europe—the European Telecommunications Standards Institute (ETSI)—designed GSM as a digital wireless communications system. Commercial service began in mid-1991, and by 1993 there were 36 GSM networks in 22 countries, with 25 additional countries looking to participate. Furthermore, the standard spread quickly beyond Europe to South Africa, Australia, and many Middle and Far Eastern countries. By the beginning of 1994, there were 1.3 million subscribers worldwide. Today, GSM is also the most widely used communications standard in China, and covers almost all of the country. So, we didn't need to set up a communications base station for our system; this, of course, significantly reduced the total cost of the project. The most basic teleservice supported by GSM is telephony. Group 3 fax, an analog method described in ITU-T recommendation T.30, is also supported by the use of an appropriate fax adapter. SMS is one of the unique features of GSM compared to older analog systems. For point-to-point SMS, a message can be sent to another subscriber to the service, and an acknowledgment of receipt is sent to the sender. SMS also can be used in Cell Broadcast mode to send messages such as traffic or news updates. Messages can be stored on the SIM card for later retrieval.

SMS is effective because it can transmit short messages within 3 to 5 s via the GSM network and doesn't occupy a telephony channel. Moreover, the cost savings makes it a worthwhile choice (i.e., in China, each message sent costs $ 0.01 and receiving messages is free). With SMS transmitting, gathering position data is easy and convenient.

Command	Definition
AT+CSCA	Set the SMS center address. Mobile-originated messages are transmitted through this service center.
AT+CMGS	Send short message to the SMS center
AT+CMGR	Read one message from the SIM card storage
AT+CMGD	Delete a message from the SIM card storage
AT+CMGF	Select format for incoming and outgoing messages: zero for PDU mode, one for Text mode
AT+CSMP	Set additional parameters for Text mode messages

Table 3—To send SMS messages, you can use these (mainly) AT commands. For more details, you may want to read the GSM 07.07 protocol on the ETSI web site.

PROJECT FILES
To download the pin assignments and source code, go to ftp.circuitcellar.com/pub/Circuit_Cellar/2003/151/.

REFERENCES
[1] European Telecommunications Standards Institute, "ETSI GTS GSM 07.05," V.5.5.0, 1998.

[2] ———, "ETSI GTS GSM 07.07," V.5.0.0, 1996.

RESOURCE
NMEA Specification
National Marine Electronics Association
(919) 638-2626
www.nmea.org

SOURCES
AT90S8515 and AT45D021
Atmel Corp.
(714) 282-8080
www.atmel.com

Sandpiper GPS receiver
Axiom Navigation, Inc.
(714) 444-0200
www.axiomnav.com

FALCOM A2D GSM module
Falcom Wireless Communications GmbH
(800) 268-8628
www.falcom.de

BASCOM-AVR
MCS Electronics
+31 75 6148799
www.mcselec.com

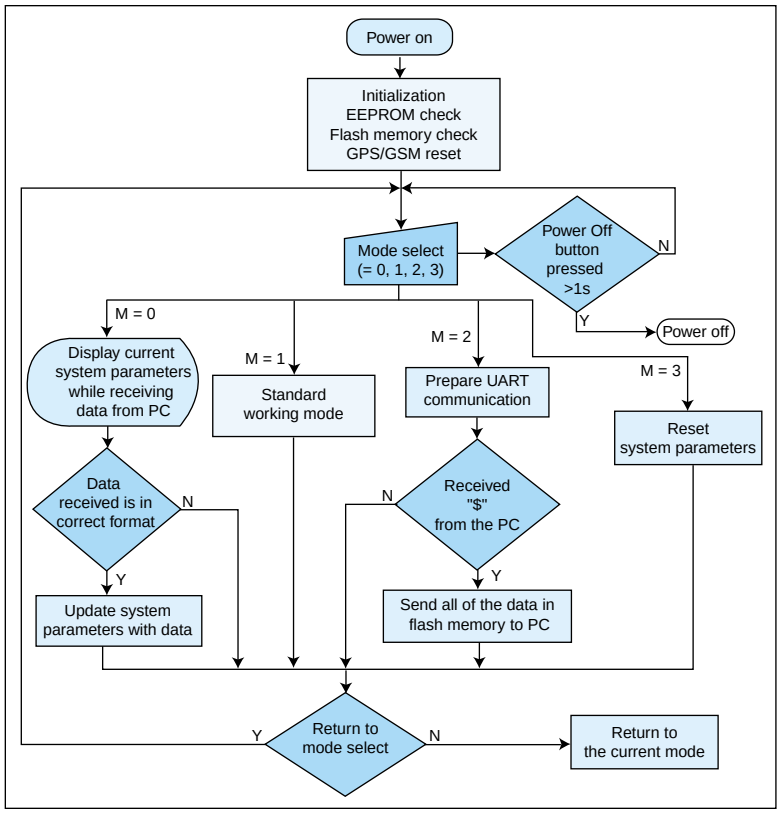


Figure 3—After initialization, you can select the function mode by pressing the Menu button and Enter button. The LCD will show the status and system parameters.

As with GPS modules, stand-alone GSM OEM modules are available. We used the FALCOM A2D from Wave-com for this project. The FALCOM A2D is a dual-band embedded GSM module (GSM900/DCS1800). It features the following services: telephony, SMS, data, and fax. The GSM module has one TTL-level serial data interface. We use AT commands to control and program the FALCOM A2D. The data and control commands are exchanged between the microcontroller and GSM module through the serial interface.

There are many groups of AT commands, including: Call Control, Data Card Control, Phone Control, Computer Data Card Control, Reporting Operation, Network Communication Parameter, Miscellaneous, and Short Message Service. We use some of the SMS commands to communicate with the control center. The main AT commands for using SMS are listed in Table 3. You can download the GSM 07.07 and GSM 07.05 protocols for more details about the AT commands that are used in GSM communications. [1, 2]

Let's review an example of how to make a GSM module send and read a sample SMS in Text mode. First, initialize the GSM module with AT commands AT+CSCA and AT+CMGF. Using the former sets the SMS center number to be used with outgoing SMS messages. Remember, the number will be saved on the SIM card just like in normal mobile phones. There are two different modes—Text mode and Protocol Data Unit (PDU) mode—for handling short messages. The system default is PDU mode; however, Text mode is easier to understand. So, use the AT+CMGF=1 command to set the module to the GSM 07.05 standard SMS Text mode.
The AT+CMGS command is used to send a short message. The format of this command is:

```
AT+CMGS=<da><CR>Message
  Texts<CTRL-Z>
```

Here, <da> is a subscriber's mobile phone number that you want to send the short message to. The GSM module can receive incoming short messages and save them on the SIM card automatically. You can use the AT+CMGR command to read an incoming short message from the SIM card storage, and then use the AT+CMGD command to delete it when you're finished.

If you want to read an SMS message, then send a AT+CMGR=x command to tell the GSM module which short message you want to read. Next, check the serial port to receive the message from the GSM module. Rs232_r is a subroutine used to receive data from the UART. Listing 1 demonstrates sending and reading a short message in a BASCOM-AVR program. In this code segment, chr(34) converts the ASCII value 34 to the right quote character ("). It also converts chr(13) to <CR> and chr(26) to <CTRL-Z>. As you can see, "My SMS Message" is the message you want to send.

Circuit Description

The difficult part of designing this project was learning both the NMEA GPS message and GSM AT command protocols. The easy part was designing the hardware circuit (see Figure 2). You may download a table of the pin assignments from the Circuit Cellar ftp site. As you can see from the schematic, there are three jack ports. JP1 (20 pins) is used for the GPS module, JP2 (40 pins) is for the GSM module, and JP3 is used for communication with the PC.

The AT90S8515 (U1) is the core of the circuit. This low-power CMOS 8-bit microcontroller is based on the AVR-enhanced RISC architecture. By executing powerful instructions in a single clock cycle, the AT90S8515 achieves throughputs approaching 1 MIPS per megahertz, allowing you to optimize power consumption versus processing speed. The AT90S8515 features 8 KB of in-system programmable flash memory, 512 bytes of EEPROM, 512 bytes of SRAM, and 32 general-purpose I/O lines. Flexible timer/counters with compare modes, internal and external interrupts, a programmable serial UART, an SPI serial port, and two software-selectable power-saving modes are also available. The high speed of the AT90S8515 makes it possible to complete multiple tasks between the GPS and GSM modules, although it has only one UART serial port. With the programmable flash memory, you have high reliability and can update your system. The EEPROM makes it possible to store system parameters such as the SMS center number, control center number, and predetermined time intervals.

Other components on the board are the four-channel multiplexer, a large capacity data memory, and the user interface. The latter consists of a 2 ˘ 16 LCD, a buzzer, and three push buttons.

Accessories

An AT45D021's serial-interface flash memory is used as a black box to store data packages. The 2,162,688 bits of memory are organized as 1024 pages of 264 bytes each. In addition to the main memory, the micro also contains two data SRAM buffers of 264 bytes each. The simple SPI serial interface facilitates the hardware layout, increases system reliability, and reduces the package size and active pin count. The AT90S8515 saves GPS data to flash memory via an SPI port at a user-defined specific interval. Or it reads data from the

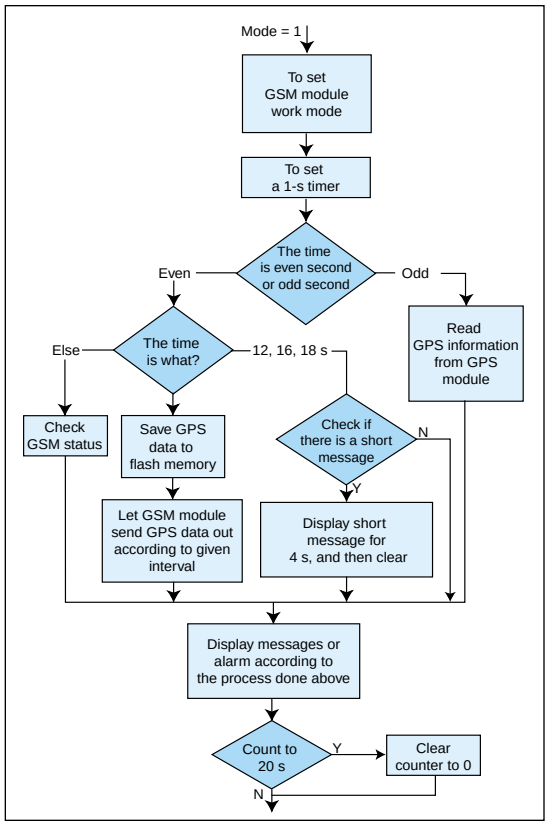


Figure 4—The main function is mode 1. The AT90S8515 microcontroller receives the ship location data every 2 s from the GPS module, and then saves the data in flash memory every 20 s. At a user-defined time interval, the AT90S8515 sends the location data to the control center, and then receives the control information from the control center via the GSM module.

flash memory to backup to PC. Up to 2160 pieces of information can be stored in flash memory. Because the AT90S8515 has only one UART port, another chip is used to expand the serial port for three kinds of different functions. The digitally controlled MC14052B analog switch is a dual four-channel multiplexer. With two I/O pins, the AVR controls it to switch among three channels, all of which are UART serial interfaces.

One MAX202 chip accomplishes the conversion between TTL/CMOS level and RS-232 level, which is necessary for the RS-232 interface between the navigator and PC. Using the RS-232 port, the system can backup the data in flash memory to the PC. Also, you can change some system parameters through the PC via the RS-232 port.

With two control pins and four data pins, the AVR gives the LCD specific information to display. Port pins PC2 through PC4 individually sense the three push-button switches. There is a Menu button to select the work mode, and an Enter button to confirm the selection. The third is an SOS button used to send an alarm message to the control center.

Software Description

We used the powerful BASCOM-AVR to develop the software. An IDE is provided with an internal assembler. You can also generate Atmel OBJ code. Additionally, the BASCOM-AVR has a built-in STK200/300 programmer and terminal emulator. Other notable features include: structured BASIC with labels; fast machine code instead of interpreted code; special commands for LCDs; I2C; one wire; PC keyboard and matrix keyboard; RC5 reception; and RS-232 communications. The BASCOM-AVR has an integrated terminal emulator with download option, an integrated simulator for testing, and an integrated ISP programmer.

You can easily write the firmware for this project using the BASCOM-AVR. And with the ISP benefit of AVR, on-line emulation is almost unnecessary, so you can program and test with ease. The flow charts in Figures 3 and 4 describe the AT90S8515 program that controls the devices. The software handles a number of key functions, such as initializing the system and starting the GPS and GSM modules. The software also selects the working mode. Additionally, it checks and sets the system parameters in mode 0, backs up the trace data stored in flash memory to the PC in mode 2, and resets the system parameters in mode 3.

Mode 1 is the standard working mode during which many tasks are completed. During mode 1, the GPS signals are read every 2 s from a satellite; the location information is saved in flash memory every 20 s; and the GSM module sends location data to the control center according to the given interval time. Meanwhile, the navigator receives the control information from the control center from the GSM module.

❑

Our system is now being used in Shanghai Port, China for navigation and monitoring of ships. Aside from tracking ships, the GPS-GSM Mobile Navigator can also find use in other applications, such as navigating taxis. The system works quite well, and we plan to adapt it for future projects.

Listing 1—We created a program to send an SMS message to a mobile phone (13916315573). The program directs the GPS-GSM Mobile Navigator to read and delete an incoming short message. The Print command is a BASCOM-AVR instruction that sends output to the serial port. The Rs232_r subroutine is used to read input from the serial port.

```
constant definition
Const Gsm_center = "+8613800210500"//SMS center number
Const Send_number = "13916315573"          //Phone number the SMS sends to
Const Sms_texts = "My SMS Message"         //Message texts to be sent

//Initialize the GSM module
Print "AT+CMGF=1"
//Set GSM module in Text mode
Print "AT+CSCA=" ; Chr(34) ; Gsm_center ; Chr(34)
                                           //Set SMS center number
//Send a message
Print "AT+CMGS="; Chr(34); Send_number; Chr(34); Chr(13); Sms_texts; Chr(26)

//Read and delete an incoming short message
Print "AT+CMGR=1"                          //Read first short message from SIM card storage
Gosub Rs232_r                              //Receive message
Print "AT+CMGD=1"                          //Delete message from SIM card storage
```

AT86RF401 Reference Design

By Jim Goings, Applications Manager, Atmel, North American RF&A

AT86RF401 Reference Design

It seems that many systems are requiring a radio frequency (RF) wireless link. We don't like standing on a chair to adjust the ceiling fan speed, we don't like climbing out of our car to open the garage door, and we certainly don't like walking outside on an early winter morning to see just how cold it is. Whether we're driven by cost, convenience, or performance, low cost RF wireless designs are here to stay. So, if you're not an expert in manipulating Maxwell's equations… is there an easy way to add RF to your design?

Fortunately, the answer is an emphatic YES. Atmel made your work much easier by recently introducing the AT86RF401, an RF wireless data micro-transmitter. By developing a chip that integrates the mysterious part of the RF transmitter design (normally reserved for an RF expert) and throwing in an AVR® microcontroller, your life just got a little bit simpler.

The heart of this chip is an AVR® microcontroller that's been given supervisory responsibility over a narrowband Phase-Locked-Loop (PLL) RF transmitter. What sets this device apart from many on the market today is that the solution is a true System On a Chip (SOC). It isn't a multi-chip package where each chip was designed by different teams having different priorities. Rather, it is a SINGLE chip resulting from the cooperative efforts of a cross-functional design team where the RF and control logic were designed to work together… from the beginning. With access to key RF control parameters such as output power attenuation, voltage controlled oscillator tuning, RF modulation and PLL lock/unlock criteria, the AVR core takes much of the headache out of getting your RF link's performance up to where you'd like it.

The AT86RF401 (see Figues A; page 40) is designed to operate down to 2.0V. C1, C7, and C8 provide an attenuation path to ground for unwanted high frequency transients. J2 provides an interface to the software development tools and allows you to flash the AVR®'s memory while it's still soldered onto the PCB. Switches, S1 – S4, along with the current limiting resistors, R1 – R4, trigger an event that awakens the device from a very low current sleep mode (typically less than 100 nA) and initiates the RF transmission.

The rest of the parts on the PCB support the RF transmitter. While X1 provides a clock source for the AVR®, it also is used as the reference frequency for the PLL. The PLL contains an internal divider fixed to 24 so the RF carrier will always be 24 times the frequency of X1 (24*13.125MHz = 315MHz). The VCO requires L2 to put its output in a controllable range enabling the PLL to closely track the reference frequency X1. All that's left to finish the design is to attach a tuned antenna to the chip and your hardware is ready. The complete Parts List is shown in table 1 below.

To minimize cost (while not the most efficient way to radiate RF), a tuned loop PCB trace antenna can be used. A reasonable impedance match between the output of the AT86RF401 and the PCB trace antenna AND assurance of an Federal Communications Commission (FCC) compliant design can be obtained using the component placement and geometry of the traces as shown in Figure 1a (top side PCB artwork including antenna) and Figure 1b (bottom side PCB artwork). Complete PCB design and fabrication documentation is available upon request. See contact information at the conclusion of this article.

In this design, peak resonance of the tuned loop antenna occurs with a non-standard capacitance value. So, three capacitors, C2-C4, are required to be connected in series to achieve this equivalent capacitance. This isn't necessarily a bad thing as a benefit to a series connection of three capacitors is a reduction in the overall variation of the equivalent capacitance.

ATMEL REMOTE KEYLESS ENTRY TRANSMITTER 315MHz version (REV B1 APRIL 15, 2003)

Item	Moose	Qty	Ref Designator	Description	Manufacturer	Part Number	Value	Tolerance	Rating	PCB Decal
1		2	C2 C4	0603 SIZE SMT CERAMIC CAPACITOR	Any		6p8F	+.25pF	50V NPO	603
2		1	C3	0603 SIZE SMT CERAMIC CAPACITOR	Any		33pF	5%	50V NPO	603
3		2	C1 C8	0603 SIZE SMT CERAMIC CAPACITOR	Any		100pF	5%	50V NPO	603
4		1	C7	0603 SIZE SMT CERAMIC CAPACITOR	Any		10nF	10%	50V X7R	603
5		1	J1	2032 COIN CELL HOLDER SMT	KEYSTONE	1061				KEYSTONE-1061
6		1	J2	3X2 PIN 0.1" RIGHT ANGLE HEADER	3M	929838-04-03				RTHEAD-2X3
7		1	L2	0603 SIZE CHIP INDUCTOR	COILCRAFT	0603CS-82NXJB	82nH	5%		603
8		4	R1 R2 R3 R4	0603 SURFACE MOUNT RESISTOR	Any		1k	5%	1/16 W	603
9		4	S1 S2 S3 S4	LIGHT TOUCH SWITCH	PANASONIC	EVQ-PPDA25				PANASONIC-EVQ-PP
10	X	1	U1	"SMARTRF" WIRELESS DATA	ATMEL	AT86RF401U				TSSOP20
11	X	1	X1	CSM-7 STYLE SMT CRYSTAL	CRYSTEK	16757	13.125MHz	+/-20ppm	CL 20pF	ECS-CSM-7
12		1	PCB 1	PRINTED CIRCUIT BOARD	JET	AT0308 rev B				

Table 1 - Parts list

Software development for this device can be done using AVRStudio. A recent upgrade, AVR Studio4, now includes drop down menus unique to the AT86RF401. When used with an AVR Starter Kit, STK500, a complete software development environment including editing, assembly, simulation, and serial flash programming can be realized.

But, if you're anxious to start playing with the hardware in the lab, try using the SPI Controller software (included with the AT86RF401U-EK1 Evaluation Kit). The SPI Controller gives you real time access to the key registers within the AT86RF401 that control the RF transmitter using a graphical user interface (GUI) as shown in Figure 2. By connecting the cable & dongle assembly (provided in AT86RF401U-EK1) between the parallel port of your PC and the programming header on the reference design, you'll be ready to go! Once you've connected your hardware and initialized the software, you can toggle the

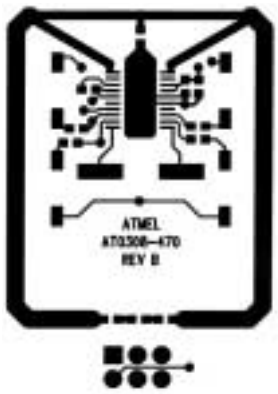


Figure 1a

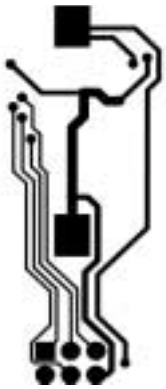


Figure 1b

appropriate bits in various registers to do things like change the output power of the RF signal (PWR_ATTEN[5:0) or activate the RF power amplifier (TX_CNTL[6:4,2]. Be sure to check out some of the canned routines located under the tool button labeled "PRESET FUNCTIONS" as shown in Figure 3. There are quite a few helpful programs that will allow you to evaluate many aspects of the RF transmitter without having to write any software.

Now that you've had a chance to try out the '401 in the lab using the SPI Controller tool, it's time to understand a sample software program that was developed to demonstrate the generation of a constant RF carrier whenever any of the switches S1 through S4 are pressed.

Using the AVRStudio4 and file CW Mode.asm as an example (see Figure 4), the essential elements of the software are:

Figure 2

Figure 3

- initialization of digital logic (e.g. AVR clock divide, stack pointer, I/O definition, etc.) and RF control registers (e.g. fine tuning the VCO, defining the PLL lock detector criteria, selecting output power, etc.)
- controlling the RF signal
- entering the sleep mode after RF transmission is complete

Upon power up, the program counter is reset to 0x0000 and execution begins at the "Reset" label. Initialization starts with establishing the AVR clock divider ratio and defining the stack pointer address. After these tasks are completed the "VCO" subroutine is called. This subroutine steps through an internal VCO tuning capacitor array to determine the optimal setting for the tuning capacitor array. This tuning process monitors both the PLL's ability to lock (TX_CNTL, Bit[2]) and the value of the VCO's control voltage window comparator (VCO-

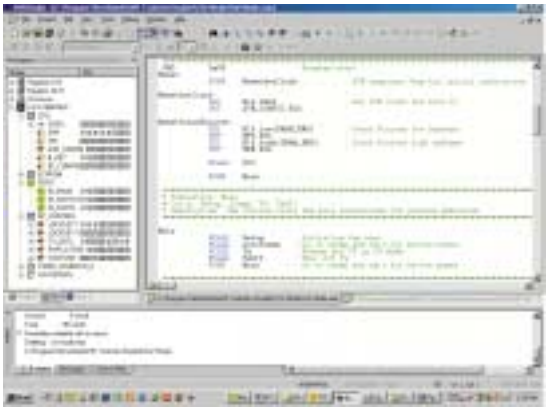


Figure 4

TUNE, Bits[7:6]). When both of these conditions are determined to be acceptable, the value of the tuning capacitor is retained in VCOTUNE, Bits[4:0].

It is important to note that optimal performance of the PLL lock detector has been determined empirically at the factory. Therefore, the constants programmed into registers LOCKDET1 and LOCKDET2 (0x07 and 0x85 respectively) do not require modification in most applications. The final steps of initialization involve the definition of the I/O registers corresponding to switches S1-S4. In this application, they are configured as inputs capable of generating a "button wake-up" (IO_ENAB, bits[5:0] and IO_DATOUT, bits[5:0]). This feature allows a switch depression to awaken the AT86RF401 from its low current sleep mode. Polling of the Button Detect Register (B_DET, bits[5:0]) provides an indication of which I/O was the source of wake up. Care must be taken to clear the bit(s) set in this register prior to entering the sleep mode.

After initialization is complete, generation of the RF carrier is straightforward. When, the appropriate bits in the Transmit Control Register, TX_CNTL, bits[5:4] are set, the RF carrier is routed to the antenna pins of the AT86RF401. This is controlled in the subroutine called "Tx". The RF continues as long as the Button Detect register indicates a switch was pressed (B_DET, bits[5:0]). Once the switch is released, the entire PLL controlling the RF carrier is powered off and the software resumes its sequence of control defined in the main loop of the program, "Main" and quickly enters the sleep mode.

This design was successfully tested for FCC compliance and yielded an output field strength of 85.8 dBuV/m. The FCC limit at 315MHz is 75.62dBuV/m but up to 20dB of relaxation on this limit is allowed if the RF is modulated. This raises the FCC limit to 95.62dBuV/m. This

means the design has a margin of 9.8dB. Results of FCC compliance testing for the fundamental and harmonics of interest are shown in Figures 5 and 6.

The formula to calculate the relaxation factor is:

$dB_{relaxation}$ = 20log(100mS/mS the RF is "on" time during 100mS)

Based on the margin of 9.8dB measured in the lab, we can calculate the maximum amount of RF is "on" during 100mS interval to determine the theoretical boundary of our modulation scheme. Using the equation above we can solve for RF "on" time as follows:

$$20dB \cdot 9.8dB = 20\log(100/t_{RF"on"})$$

$$t_{RF"on"} \leq 30.90mS$$

Based on this information, it would be possible to modulate the RF carrier using On-Off-Keying with a 50% duty cycle at a data rate of up to 10KHz (limited by the AT86RF401) for a duration of 61.8 mS and still meet the limits of the FCC requirements for intermittent operation as defined in FCC part 15.231, "Periodic operation above 70 MHz". Under these conditions, 618 bits of data could be sent at a data rate of 10Kb/S and the transmitter would still be FCC compliant!

As you can see, the AT86RF401 can make adding an RF link to your system easy and economical priced at only $1.36 in quantities of 100K. To get your design to market faster, try ordering an evaluation kit that contains the hardware and software described in this article. Your local Atmel distributor can provide this for $199. Use the order number AT86RF401U-EK1 for a 315MHz design or AT86RF401E-EK1 for a 433.92MHz. Both are available in stock today!

continued on page 40

For more information on this product or for additional design documentation, you may contact the author by phone: 719-540-6873 or email: jgoings@atmel.com. ❑

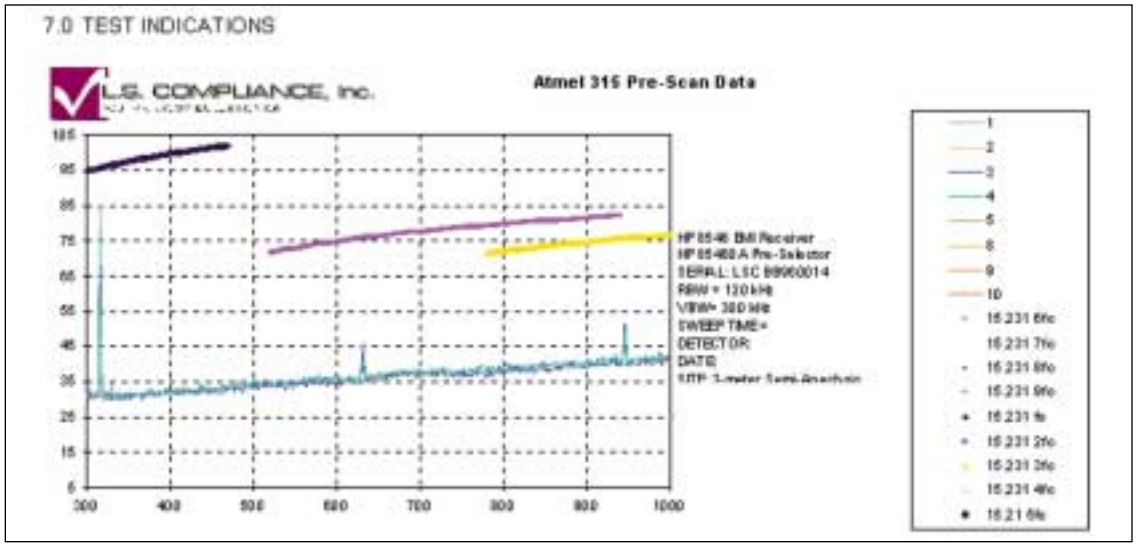


Figure 5

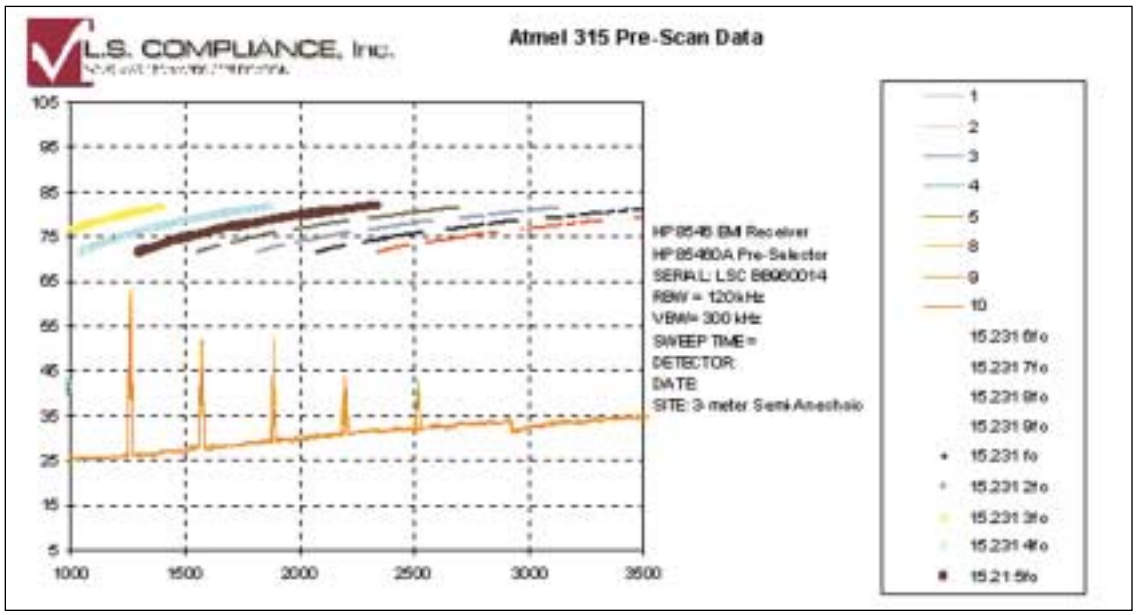
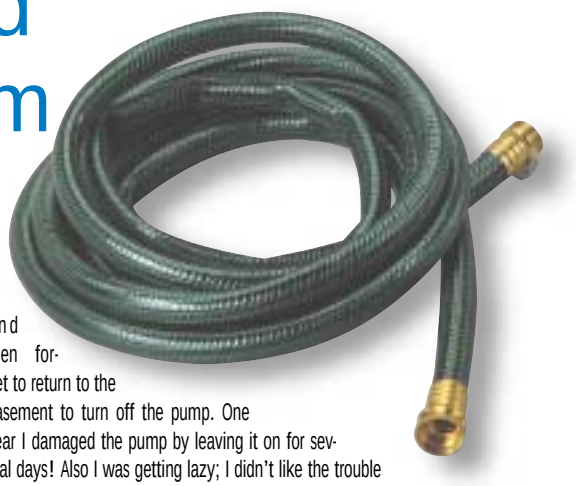


Figure 6

An RF-Controlled Irrigation System



With access to a steady water supply, Brian's garden should flourish in even the driest of times. Having caught wireless fever, he set out to use an AVR and some RF products to man the pump and close the valves. Now, watering only takes a press of the green thumb.

By Brian Millier

When I sat down to write this article last fall, the leaves on the trees had not yet turned their autumn colors, but the beauty of the flowers in our garden beds was certainly on the wane. It was a dry summer, particularly punishing for farmers, and our gardens weren't particularly splendid last year. Not that I didn't try to keep them well watered, it's just that it's hard to beat a steady dose of rainwater.

We're fortunate to have built a home on a large lake. Twelve years ago, we chose the lot based mainly on recreational concerns—swimming, canoeing, and such. I became seriously interested in gardening about five years back, and decided to install an irrigation system to make use of the unlimited supply of "free" water.

Our lot is about 25 feet above the lake's level. As any mechanical engineer will tell you, it's a lot easier to "push" water than it is to "pull" it, so I installed a 0.75-hp jet pump at the water's edge. I decided against using a pressure tank and switch, as the water would be needed only when the pump was switched on, and the maximum continuous flow rate was desirable.

Because most of the rough landscaping had been done when the house was built, I decided it would be too much effort and expense to bury irrigation lines throughout the 0.75 acre of lawn and gardens that I have. Instead, I ran 1.5≈ plastic pipe on the surface, along the side border of my property. Six valves/garden hose fittings are spaced along the 400 foot length.

For a number of years, I was content to run down to the electrical panel in the basement to switch on the pump when I wanted to do some watering. Besides being inconvenient, occasionally I'd shut off the water valves when finished and then forget to return to the basement to turn off the pump. One year I damaged the pump by leaving it on for several days! Also I was getting lazy; I didn't like the trouble of hooking up a hose, unraveling 100 feet of it into the desired position, attaching a sprinkler head, and then having to walk all of the way back to the other end to turn on the water valve.

I decided what I needed was a controller that allows me to program specific watering times and durations. Units like this are commercially available, of course, but I also wanted to be able to control the water using a small keyfob transmitter while I puttered around in the gardens.

In my last article, I described a wireless MP3 player, which used low-cost UHF transmitter/receiver modules from Abacom Technologies ("Listen Everywhere," Circuit Cellar 134). I was pleased with their performance and technical support from Abacom, so I decided to check out Abacom's products again.

I wanted the transmitter to fit in a keyfob, so I chose the AT-MT1-418 AM transmitter module, which is about the size of a penny. I also chose Abacom's keyfob transmitter case, which comes in various switch cutout configurations. I decided to use a sensitive receiver because I anticipated a low transmitted signal level given such a small transmitter. The QMR1 Quasi AM/FM superhet receiver module fit my needs. I particularly like this module because its 1-square-inch SIP mounts easily on a circuit board by pins on 0.1≈ centers. I like one-stop shopping, so of course I was pleased to be able to get Holtek encoder/decoder chips from Abacom, as well. I'll describe the chips in more detail later in the article.

Photo 1—Here's the actual controller/receiver sitting in my family room. Just visible in the background is a glimpse of the lake—the source of water for the gardens. Not visible is the AC adapter used for power or the power relay, which is located at the electrical panel in the basement.

Controller/Receiver

If you've read my recent articles, it should come as no surprise that I used an Atmel AVR controller chip, the AT90S8535-8PC (40-pin DIP package), for this project. This device contains four 8-bit ports, eight 10-bit ADC channels, 8 KB of flash memory, and 512 bytes each of data EEPROM and RAM. Like most AVR devices, this one is easily serially programmable in-circuit. You may want to refer to my article, "My fAVRorite Family of Micros" (Circuit Cellar 133) for an overview of this family, along with the details of a free ISP programmer for these chips.

I must admit up front that I probably could have done this project with the smaller AT90S2313 by multiplexing some of the I/O pins and writing the program in assembly language. I decided it was more productive for me to spend the extra dollars (Can $) on the '8535, whose larger flash memory would allow me to program in BASIC, using the BASCOM AVR compiler.

Author's Note: I want to thank John Barclay of Abacom Technologies for the support and samples that helped out significantly while I was putting this article together.

Brian Millier is an instrumentation engineer in the Chemistry Department of Dalhousie University, Halifax, Canada. He also runs Computer Interface Consultants. You may reach him at brian.millier@dal.ca.

Figure 1 is a schematic of the controller/receiver. Let's start by looking at the user interface. The user interface consists of a 4 ˘ 20 LCD and four push buttons. The display is operated in the common 4-bit mode; in this case, because it saved some wiring, not because of a shortage of I/O pins.

The four push-button switches are individually strobed by port pins PC0–3 and sensed by the INT1 input of the '8535. I hooked up the switches this way because I originally drove the LCD using the same four port C lines. I had been saving the ADC inputs of port A for future use, but later changed my mind and switched the LCD over to port A, leaving this switch circuit intact.

The four push-button switches operate this unit the same way that many small electronic devices work. There is a Menu button to scroll through several menus as well as a Select/Cursor button. The buttons are used to position the cursor within a time field for adjustment purposes or to select a particular value when finished changing it. Finally, there are plus sign and negative sign buttons used to increment or decrement the current parameter.

I chose to implement the real-time clock in the software. One reason I initially picked the '8535 over the slightly less expensive '8515 is because it includes a third timer, which may be driven by a 32,768-Hz watch crystal. I must say that my attempts to implement the RTC using this feature gave me some problems! Atmel's datasheet for the '8535 advises you to merely connect the 32,768-Hz watch crystal between the TOSC pins 1 and 2 with no capacitors to ground. [1]

When I did this, I could see a reasonable 32,768-Hz sine wave signal on either crystal pin with my oscilloscope using a 10˘ probe. I soon discovered, though, that my clock was losing about 1 min./h. After troubleshooting, I found that the crystal oscillator waveform contained serious glitches coinciding with LCD screen refreshes.

At that point, I was using the port pin adjacent to TOSC1 to drive the LCD ENABLE pin. Moving the LCD ENABLE pin over to port A eliminated the glitches, but the clock was still slow. This was odd because I could not see anything wrong with the crystal waveform with my oscilloscope, and the built-in frequency counter in the oscilloscope indicated that the frequency was "bang-on." So next, I contacted Mark at MCS Electronics to see if he had run into the problem. He mentioned capacitors, which made me think that capacitance to ground was probably needed (contrary to the datasheet). It turns out that my oscilloscope was providing the necessary capacitance, but only when it was hooked up. Adding 22-pF capacitors to ground cured the problem, at least with the particular crystal I was using. However, for this project, I decided to play it safe and implement the RTC using Timer0 of the '8535 clocked by the 4.194304-MHz crystal of the CPU, which works perfectly. A side effect of this was that I couldn't use BASCOM's intrinsic real-time clock function and instead had to write my own routine.

Reprinted with permission of Circuit Cellar® - Issue 138 January 2002

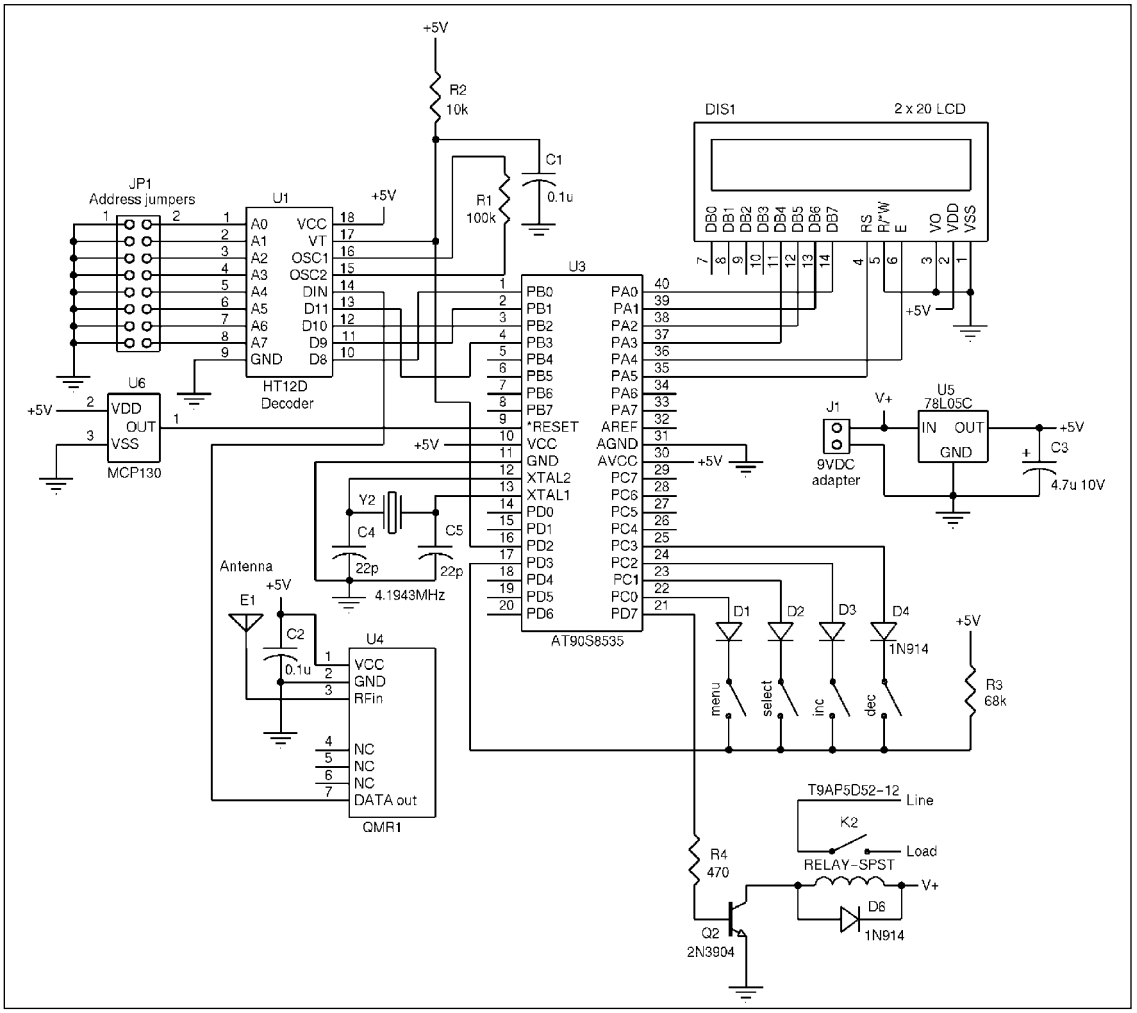


Figure 1—The Atmel 8535 AVR controller is at the center of the action of the irrigation controller. An Abacom QMR1 receiver takes care of the wireless reception functions. The LCD operates in 4-bit mode.

My pump draws about 10 A when running (much more when starting), so I chose a Potter & Brumfield T9AP5D52-12, which is inexpensive and rated for 20-A continuous current. A small 2N3904 transistor is all that is needed to handle the 200 mA that its coil requires. This sealed relay is small. I haven't used it long enough to know how well it will hold up, so the jury is still out on this component choice.

The controller/receiver is powered by a 9-VDC adapter followed by a 78L05 regulator. The actual output of the adapter is closer to 12-V, and is enough to operate the relay coil. Photo 1 shows the controller in place in my family room. The wireless part of the controller consists of an Abacom QMR1 receiver followed by a Holtek HT12D decoder chip. This receiver is one of the choices recommended for use with the AT-MT1 AM transmitter that I use. The datasheet that comes with the package (available soon on www.abacom-tech.com) calls the QMR1 a quasi-AM/FM receiver module. The datasheet doesn't spell out if it also works with FM transmitters, but it sounds like it would.

In any AM transmitter/receiver link, one thing for certain is that the receiver will spit out a stream of noisy data during much of the time when its companion transmitter is not transmitting. The QMR1 is sensitive (RF sensitivity specification is –110 dBm) and it has no squelch circuitry to suppress spurious output signals arising from any RF interference that it might receive. With cell phone towers cropping up all over the countryside, even my rural home is probably not "RF-quiet" anymore. I definitely see lots of noise output from the QRM1 receiver module.

My intention is to emphasize the need for some form of error detection/ data formatting in any AM RF link. What I haven't mentioned is that the circuitry in the receiver that recovers the data from the RF signal (called the data slicer) is choosy about the form of data modulation that it will accept.

For example, most data slicers work reliably only if there is a roughly even distribution of zeros and ones in the datastream, even within the short-term such as the time taken to send 1 byte of data. This means that you cannot, for example, just feed in the signal from a UART to an AM transmitter, and expect to hook up a UART to the receiver output.

Instead, Manchester encoding is generally used because it guarantees an equal number of zeros and ones in the datastream, regardless of the particular data being sent. Furthermore, it is good practice to send the same data several times and check that it matches when it comes out of the receiver. A final precaution could include some form of checksum or better still, a CRC byte in the data packet to further verify the integrity of the received data.

Another concern is the amount of time it takes the receiver to adjust itself to the strength of the incoming signal or wake up from an idle state if that feature is present in your receiver module. To allow for this, the transmitter must send out a short stream of known data, called a preamble, to allow the receiver to get ready for data reception, so to speak.

This is a lot tougher than your average RS-232 serial data link! There are many books that cover in depth the theory of reliable RF data communication; An Introduction to Low Power Radio by Peter Birnie and John Fairall is a good starting point for those of you starting out in this area. [2]

Encoder/decoder

To address these concerns, it made sense to use the inexpensive line of encoder/ decoder devices from Holtek (HT12D/E) rather than roll my own. These matching chips address the concerns, at least for applications that need only to transmit the status of a small number of switches.

There are a number of good reasons for choosing this device. The HT12E encoder chip consumes only about 0.1 µA in Standby mode, so it can be left permanently connected across the small transmitter battery. It comes in a small, 20-pin SOP and fits in a small transmitter case (the same could be said for the Atmel ATiny and smaller PIC processors). To reduce parts count and cost, it uses a single resistor to set its internal RC clock. RC clocks are not known for their frequency stability; the design of this encoder/ decoder pair allows the receiver to be able to lock onto the transmitter's data clock frequency even though it may vary considerably over time or temperature. Refer to Figure 2 for the schematic of the transmitter module.

Both the encoder and decoder sample eight lines (A0 through A7), which act as device address inputs. That is to say, a given encoder/decoder pair can be set to operate at one of 256 discrete addresses. This strategy, for example, prevents your neighbor's remote control from operating your garage door opener.

Addressing can be done with a dip switch, jumpers, or by cutting traces on a PCB. Modern encoder/decoder chipsets used in remote car starters use, by necessity, a much more complex addressing scheme because there's a much greater chance of false triggering by other, unintended transmitters in the vicinity. Obviously, this leads to worse repercussions.

The data packet sent by the HT12E consists of the 8-bit address followed by a 4-bit data field corresponding to the state of up to four switches connected to

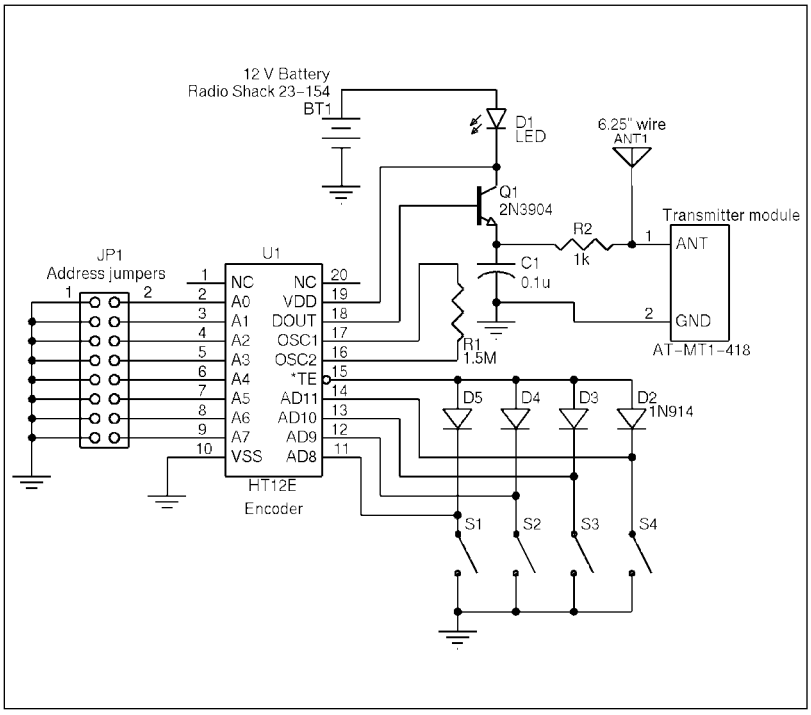


Figure 2—There isn't too much to the schematic diagram of the keyfob transmitter. However, getting it to fit into the small keyfob was another matter!

SOFTWARE

To download the code, go to ftp.circuitcellar.com/pub/Circuit_Cellar/2001/138/.

REFERENCES

[1] Atmel Corp., "8-bit AVR Microcontroller with 8K Bytes In-System Programmable Flash—AT90S8535

AT90LS8535," rev. 1041GS, September 2001.

[2] P. Birnie and J. Fairall, An Introduction to Low Power Radio, Character Press Ltd., UK, 1999.

[3] Holtek Semiconductor Inc., "212 Series of Decoders," July 12, 1999.

[4] ———, "HT12A/HT12E 212 Series of Encoders," April 11, 2000.

SOURCES

AT-MT1-418 AM Transmitter module
Abacom Technologies
(416) 236-3858
Fax: (416) 236-8866
www.abacom-tech.com

AT90S8535-8PC Microcontroller
Atmel Corp.
(714) 282-8080
Fax: (714) 282-0500
www.atmel.com

HT12D/E Decoder chip
Holtek Semiconductor Inc.
(510) 252-9880
Fax: (510) 252-9885
www.holtek.com

BASCOM-AVR Compiler/programmer
MCS Electronics
31 75 6148799
Fax: 31 75 6144189
www.mcselec.com

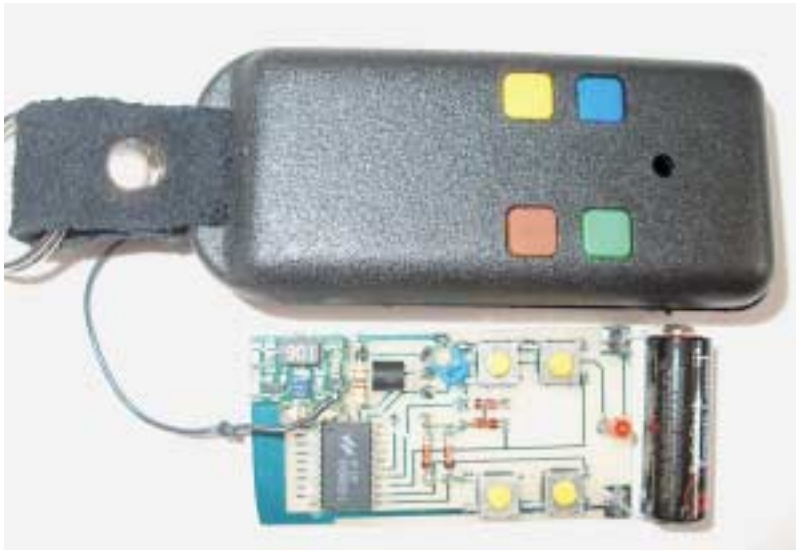


Photo 2—The PCB that I fabricated for the transmitter sits below the keyfob case. You can see a bit of the thin black wire, which forms the antenna, connected to the tiny transmitter module.

inputs D8–D11. The datasheets for the HT12D/E devices don't mention a preamble being sent before the data, nor do they mention a checksum nor CRC bytes for data checking. [3, 4]

In place of this, the data packet is transmitted three times for each switch closure and then checked for equality by the receiver. Holding the switch down for any more than an instant, will result in the repetition of the datastream. Presumably this is how the lack of a preamble is handled—the receiver likely misses out on the first occurrence of the data packet, but catches subsequent ones.

The Abacom AT-MT1 transmitter has a maximum data transmission rate of 2400 bps. There-fore, I set the encoder's oscillator of the HT12E to 2 kHz by using a 1.5-MW resistor across OSC1 and OSC2. [4]

The AT-MT1 transmitter is a two-wire device. It is not modulated per se; instead it is powered up and down in step with the datastream. The SAW oscillator used in this module is able to turn on and off quickly—fast enough to handle the maximum data rate. The output of an encoder chip is supposed to directly power the AT-MT1, according to its datasheet. Although the data output pin of the HT12E is capable of sourcing up to 1.6 mA, the AT-MT1 requires up to 9 mA at 12 V to operate. So, in this case, I had to add a 2N3904 emitter follower to provide the necessary current boost.

I intended to use a Linx Splatch antenna, which is a small PCB containing a 418-MHz antenna and ground plane. Unfortunately, this small antenna radiated much less signal than a quarter-wave whip antenna and would not provide the range I wanted. However, it wasn't too great a loss because I was having trouble fitting everything into the keyfob anyway. I ended up using a 6.25≈ piece of flexible wire as an antenna, which just hangs out of the keyfob case and doesn't mind being stuffed into my pocket.

Photo 2 is a close-up of the transmitter PCB, which has to fit in the case and line up with the switch cutouts. I included the PCB layout in PDF format along with the firmware files, because the design of the transmitter PCB is tedious.

Choosing a battery for the transmitter wasn't difficult. There seems to be only two choices in small batteries: 3.6-V coin cells and the 12-V alkaline batteries used in many remote car starters. The HT12E encoder would have worked fine at 3.6 V, but the output power of the transmitter module would have been low. Thus, I chose the 12-V batteries.

The Firmware

One of the reasons for choosing the AT90S8535 instead of one of its little brothers, like the '2313, was to allow me the luxury of programming the firmware in BASIC. From past experience, I thought there was not enough space in the 2-KB flash memory of the '2313 for an application such as this using compiled BASIC.

I wrote the firmware using the MCS Electronics BASCOM-AVR compiler. It took up more than half, 4800 bytes, of the 8192 bytes of flash program memory, confirming my fears that it would not have fit into the memory of the smaller '2313 device. Incidentally, the demo version of the BASCOM-AVR is available free from MCS Electronics, and is fully functional apart from the fact that its program size limit is 2 KB.

As I mentioned earlier, problems I had using Timer2 (designed for RTC purposes) of the '8535 prevented me from using the built-in RTC routines in the BASCOM-AVR. This had an upside: The RTC routines needed by this application do not require week, month, or year, so they use less memory space even though they were coded in BASIC (Note: The BASCOM intrinsic RTC function is done in assembly language).

Most of the firmware takes care of the user interface. An LCD with four push buttons is easy to build, but takes up considerable program space to implement a friendly user interface. There is a routine that allows you to set the clock to the current time. Another routine enables you to enter up to six programs. Each program consists of a time, action (pump on/off), and a Daily or Once-Only mode. And, a final menu item allows you to turn the pump on and off immediately from the controller.

The six user-defined programs are stored in EEPROM, so that they survive a power failure. However, because the CPU (and therefore the RTC) will stop if the power goes off, this is a moot point, unless I add a battery backup for the controller's CPU.

When a command comes in from the wireless transmitter, the valid transmission (VT) line on the decoder will go high, and its four data output lines will reflect the state of the four buttons on the keyfob transmitter. The VT signal is fed into the INT0 interrupt input of the '8535 (through RC filtering to prevent false triggering). An interrupt service routine checks the state of the decoder's four outputs and turns the pump on or off accordingly. Although I fitted four buttons into the transmitter and allowed for all four in the controller, the firmware currently responds to only two switches—pump on and pump off. I will likely think of some other device to hook up to this in the future.

Time's up

There's no doubt that it's much less expensive to buy a remote control module off the shelf than it is to build your own, if you can find one that suits your needs. However, if your requirement is unique or you can combine a few functions into one unit, then the satisfaction of designing your own unit makes it all worthwhile. I find building these wireless gadgets addictive. In the back of my mind, I'm already thinking of my next project: a controller for air exchanger in my home using indoor/outdoor temperature and humidity sensors and a power line modem.

❑

Designer's Corner:

AVR Project Boards Make Embedded System Design Modular and Easier

I like working with microcontrollers from sunrise to sunset and then just a little bit more at night as a hobbyist building robotic applications. The time I enjoy the most while working with microcontroller is my spare hours at night when I develop robots and gadgets. This implies that I have to use my budget to buy all the necessary components, at the same time I pay my house and other bills. That is why I can not afford to buy a 10K emulator to make my embedded system design experience easier. While most companies think of hobbyists just as a group of people playing and not big expenders (reason for which our needs are not necessarily supported), I like to think that most us could easily be the future of many microcontroller based applications.

I knew I was not alone when a company decided to target designers with limited budget to use their microcontroller. Atmel's AVR 8 bit RISC architecture is one of the greatest and easiest I have explored, but its real kick to me was that the tools were inexpensive and extremely powerful. With an ICE200 for around $200 (at the time, it sells for $100 now!) and the STK200 at hand, my home projects started to take place and my wallet to breath with ease!

I never found a complaint with regards to the ICE200. The STK200 on the other hand was a different story. The tool was great and economical when evaluating a particular microcontroller for a small project. Unfortunately it lacked a vital part for my style of development. I needed many boards where each could hold a microcontroller based application with little breadbording or wire wrapping as possible. Also, I wanted to interconnect these boards without having to use tedious harnesses.

What I needed was a development board with prototyping space and some means to connect more than one together. Browsing through the web didn't help. That was when I designed the AT90SMINIPB. This little board has ton of prototyping space. It will accommodate IC's, passive components as well as all the other items a designer need to develop an application revolving around any 20 pin or 8 pin device from the AT90S Clasical and ATtiny architectures (Refer to Figure 1).

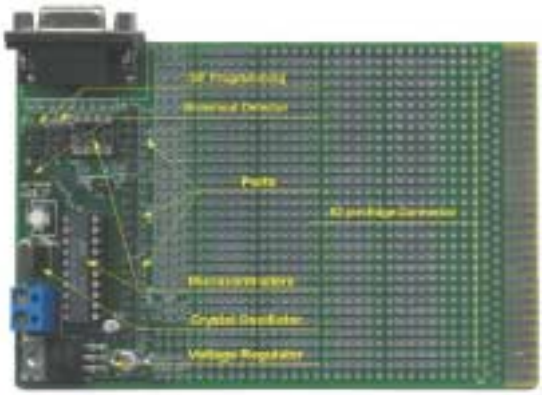


Figure 1

The board worked awesomely! Thanks to the easy access to all ports I was able to develop applications to control steppers, DC motors, high power loads, sound recording chips, etc very, very fast!

In order to interconnect more than one board together so that they could share signals such as power and control lines, the board bottom side has an edge connector with extra pads to give a door for the microcontroller to the outer world.

For this concept to work we need the PBMB (Project Board Mother Board). This board has three edge card slots where the project boards can be plugged in. Each connector contains 62 signals which are totally shareable between the three cards. Thanks to a fourth set of pads, these signals can also be interconnected to the available prototyping area. To make it more universal, the PBMB already comes with its own RS-232 port. Extremely handy when wanting to use the UART on most of the AVR microcontrollers.

Figure 2

Of course most designers will agree that not all projects can be achieved with an AT90S2313 or an ATtiny. It came to the fact that I needed more power; something along the line of an AVR Mega. To meet this requirement, the AT90S15PB and AT90S35PB were designed. I was now able to create massive projects with up to 32 I/O lines which included resources such as ADC, Timers, PWM, Input Captures, SPI, etc.

Again life was good. But I had learned my lesson and remembered the concept of flexibility. What if I were to need more space? More holes to put extra components that the microcontroller needs to fully work as intended in the desired application. The ProtoXP (from Prototype Expander) gives new added flexibility as even more holes with the same architectural pattern can now be plugged into one of the PBMB slots.

What you get:

Each Project Board contains either one large microcontroller (AT90S15PB and AT90S35PB) or two small ones (AT90SMINIPB). To make the microcontroller work, the board includes all necessary circuitry such as voltage regulator, crystal based external oscillator and reset voltage manager (brownout detector).

We're interested in your experiences in working with the AVR. Please send your tips, shortcuts, and insights to: bob@convergencepromotions.com, and we'll try and print your submissions in future issues.

Figure 3

To allow the board to be programmed, the MINIPB includes an ISP connector per chip, compatible with the ATAVRISP cable. AT90S15PB and AT90S35PB boards include the same ISP connector plus the JTAG connector that allows in circuit debugging, as well as programming, with the ATJTAG-ICE cable. The boards also include a good set of pins and pads that connect to the microcontroller ports. This is the place were the microcontroller is connected to the external peripherals localized on the huge prototyping area. The prototyping area is not a bunch of independent holes as in other prototyping boards. There are spaces were the holes are connected to other holes, but there are as well patches of independent holes and power planes holes.

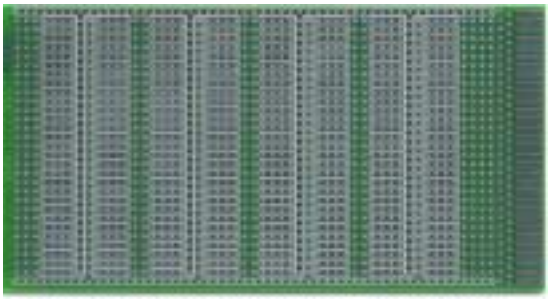


Figure 4

Finally, but equally important, each board contains a female DB9 meant for RS-232 communications. The board does not include the RS-232 driver, but there is enough space to interface such device if needed.

These projects boards can be used as a stand alone unit, but in the case more than one are to be interconnected the PBMB offers such capability. The PBMB does contain the fully functional RS-232 standard driver and is ready to work. Just patch the RS-232 Rx and Tx to the microcontroller through the edge connector bus and the application has PC compliant serial communications. The PBMB also offers voltage regulation to generate 12V and 5V.

The last board is the ProtoXP. Its middle name is expandability and it is nothing more than an extended prototyping area to add more and more components to the embedded system application. It has the same edge connector so that it can be connected on the PBMB along with other Project Boards.

Conclusion:

The ideas behind Avayan Electronics' Project Boards are modularity, flexibility and general purpose design. Users will find that a project based on a mother board is desirable as it allows for the different modules to be worked upon. Obviously this implies expandability as well. Because the boards are not set in stone and simply include all the necessary circuitry for the microcontroller to work, as well as a good amount of prototyping area, any application can be designed. Some designers may argue that the boards are too simple and that some important components are missing like LED's, drivers, etc. Because not everybody needs the same features, the boards were designed as general purpose as possible. The huge prototyping area should be enough to accommodate such needed features. For more information visit www.avayanelectronics.com or contact Avayan@avayanelectronics.com. ❑

Third Party Hardware and Software Tools Support

ADAPTERS

Adapters.Com
Programming and emulator adapters
Tel: +1 408 855-8527 Fax: +1 408 855-8528

Aprilog
Adapters for programming, emulation, logic analyzers and breadboarding.
Tel: +1 702 914-2361 Fax: +1 702 914-2362
Email: sales@aprilog.com

Emulation Technology, Inc.
Programming and emulator adapters. Online store
Tel: +1 408 982-0660 Fax: +1 408 982-0664

Logical Systems
Programming and emulator adapters
Tel: +1 315 478-0722 Fax: +1 315 479-6753

Winslow Adaptics
Programming and emulator adapters adapters@winslow.co.uk
Tel: +44 1874 625555 Fax: +44 1874 625500

APPLICATION BUILDERS

Gennady Gromov
Development tools
Tel: +7 0872 458 225 algrom@tula.net

IAR Systems
IAR MakeApp for AVR. Device driver
America: Tel: (415) 765-5500
UK: Tel: +44 207 924 3334
Germany: Tel: +49 89 90069080
Sweden: Tel: +46 18 167800

Kanda Systems
STK200 value added pack info@kanda.com
Tel: +44 1970 621030 Fax: +44 1970 621040

Mentjies, Dirk
Assembler template builder dirk@kivtronics.co.za

Unis
Processor expert, multi language builder

ASSEMBLERS

Gennady Gromov
Graphic Visual Macroassembler (editor, compiler, simulator, programmer)
Tel: +7 0872 458 225, algrom@tula.net

GNU
Freeware compiler from the GNU Project

IAR Systems
IAR Embedded Workbench
America: Tel: +1 415 765-5500
UK: Tel: +44 207 924 3334
Germany: Tel: +49 89 90069080
Sweden: Tel: +46 18 167800

Mortensen, Tom
Assembler

Virtual Micro Design
AT90S/ATmega Assembler and Simulator
Tel: +33 559 013 080 Fax: +33 559 013 081

COMPILERS

AVR-GCC, GNU Project
Freeware C Compiler

CodeVisionAVR C Compiler
C Compiler
Tel: (+40) 723469754 Fax: (+401) 722181658
office@hpinfotech.ro

Digimok
BASIC Compiler and Java Virtual Processor
Tel: +33 3 21 86 54 88 Fax: +33 3 21 81 03 43

Dunfield Development Systems
Micro-C Developers Kit
Tel: +1 613 256-5820

E-Lab Computers
Pascal Compiler
Tel: +49 7268 9124-0 Fax: +49 7268 9124-24

FastAVR
Basic Compiler
microdesign@siol.com

FORTH, Inc.
Forth Compiler
forthsales@forth.com

IAR Systems
IAR Embedded Workbench, C and C++ Compiler America: Tel: +1 415 765-5500
UK: Tel: +44 207 924 3334
Germany: Tel: +49 89 90069080
Sweden: Tel: +46 18 167800

ImageCraft Inc.
C Compiler for tiny, classic and mega AVR
Tel: +1 650 493-9326 Fax: +1 650 493-9329

Kreymborg, Ron
C Compiler

Kuehnel, Dr. Ing. Claus
C-, Pascal- and C BASIC Compiler
Fax: +41 1 7850275 info@ckuehnel.ch

MCS Electronics
BASCOM-AVR BASIC compiler
Tel: +31 75 6148799 Fax: +31 75 6144189
info@mscelec.com

RAM Technology Systems
Multi-Tasking Forth Optimising Compiler
tel: +44 1202 686308 alan@ram-tech.co.uk

Rhombus
Basic compiler including simulator, ISP, Terminal Emulator
Tel: +1 864 233-8330 Fax: +1 864 233-8331
info@rhombusinc.com

SPJ Systems
C Compiler
spj@spjsystems.com

DEBUGGERS

IAR Systems Ltd.
AT90S/ATmega Debugger
Tel: +46 1816 7800 Fax: +46 1816 7838

Virtual Micro Design
AT90S/ATmega Debugger
Tel: +33 559 438 458 Fax: +33 559 438 401

DEVELOPMENT BOARDS

Akida LLC
Design and develop boards
Minesh@Akida.com

Avayan Electronics
Tel: +1 585 217-9578 avayan@avayanelectronics.com

Baritek Inc.
AT90S8535 & ATmega103 Development Boards
Tel: +1 781 749-2550

Bluoss Elektronik
AT90S2313 Development Board
Fax: +49 911 474 2588 info@bluoss.de

Dontronics, Inc
AVR Simmstick

Embedded Systems, Inc
Self-contained microprocessor module
Tel:+1 763 757 3696
Fax:+1 763 767-2817

Equinox Ltd
AVR Evaluation Board(s)
Tel: +44 1204 529000
Fax: +44 1204 535555

Flash Designs Ltd.
Development Boards
Tel: +353 (0) 876 687 763
Fax: +353 (0) 8756 687 763 sales@flash.co.uk

Futurlec
Development board for the AT90S2313 sales@futurlec.com

Lawicel
Evaluation Board and CAN
Tel: +46 0451 59877 Fax: +46 0451 59878
info@lawicel.com

Opticompo
ATmega103/128 Development Board
support@opticompo.com

Progressive Resources LLC
MegaAVR Based, Single Board Computer
Tel: +1 317 471-1577 Fax: +1 317 471-1580
sales@prllc.com

Shuan-Long Electronics
Tel +86 10 82623551

UINFO
AVR Ethernet Controller Development Kit
Tel/Fax: +420 67 721 0608 uinfo@bigfoot.com

Third Party Hardware and Software Tools Support

REAL TIME O/S

CMX Systems, Inc.
All AVR platforms, CMX-RTX: Real Time Multi-Tasking O.S., CMX-Tiny+: Tiny Real Time Multi-Tasking O.S., CMX-MicroNet: Small TCP/IP stack
Tel: +1 904 880-1840 Fax: +1 904 880-1632
cmx@cmx.com

egnite Software GmbH
Nut/OS and Nut/Net, (RTOS and TCP/IP Stack)
Tel +49 (0)2323-925375 Fax +49 (0)2323-925374
harald.kipp@egnite.de

Micrium, Inc.
AT90S/ATmega RTOS www.micrium.com

Nilsen Elektronikk
AT90S/ATmega RTOS
Tel: +47 6758 3162 Fax: +47 6758 9761

OSE Systems
Tel: +1 214 346-9339 support@enea.com

Progressive Resources LLC
PR_RTX, a task switcher for CodeVision
Tel: +1 317 471-1577 Fax: +1 317 471-1580
sales@prllc.com

SEGGER Microcontroller Systeme GmbH embOS
Tel: +49-2103-2878-16 Fax: +49-2103-2878-28
ivo@segger.com ❑

Atmel AVR, MEGAAVR, LCD AVR, TINYAVR, USB AVR, SECURE AVR, DVD AVR, RF AVR and FPGA AVR Devices

AVR

AT90VC8544
8-Kbyte In-System programmable Flash Program Memory, 256 byte SRAM, 512 Byte EEPROM, 8-channel 10-bit A/D. Up to 4 MIPS throughput at 4 MHz. 3.6 and 5 volt operation.

AT90S1200
1-Kbyte In-System programmable Flash Program Memory, 64-Byte EEPROM, 32-Byte Register File. Up to 12 MIPS throughput at 12 MHz.

AT90S2313
2-Kbyte In-System programmable Flash Program Memory, 128 Byte SRAM and EEPROM. Up to 10 MIPS throughput at 10 MHz.

AT90S2323
2-Kbyte In-System programmable Flash Program Memory, 128 Byte SRAM and EEPROM. Up to 10 MIPS throughput of 10 MHz. 5V operation.
3V version: AT90LS2323

AT90S2343
2-Kbyte In-System programmable Flash Program Memory, 128 Byte SRAM and EEPROM. Up to 10 MIPS throughput of 10 MHz. 5V operation.
3V version: AT90LS2343

MEGAAVR

ATmega8
8-Kbyte self-programming Flash Program Memory, 1-Kbyte SRAM, 512 Byte EEPROM, 6 or 8 channel 10-bit A/D. Up to 16 MIPS throughput at 16 MHz. 5V operation.
3V version: ATmega8L

ATmega8515
8-Kbyte self-programming Flash Program Memory, 512 Byte SRAM and EEPROM. Up to 16 MIPS throughput at 16 MHz. 5V operation.
3V version: ATmega8515L

ATmega8535
8-Kbyte self-programming Flash Program Memory, 512 Byte SRAM and EEPROM, 8 channel 10-bit A/D. Up to 16 MIPS throughput at 16 MHz. 5V operation.
3V version: ATmega8535L

ATmega162
16-Kbyte self-programming flash Program Memory, 1-Kbyte SRAM, 512 Byte EEPROM, JTAG interface for on-chip-debug. Up to 16 MIPS throughput at 16 MHz.
1.8V version: ATmega162V

ATmega16
16-Kbyte self-programming Flash Program Memory, 1-Kbyte SRAM, 512 Byte EEPROM, 8 channel 10-bit A/D, JTAG interface for on-chip-debug. Up to 16 MIPS throughput at 16 MHz. 5V operation.
3V version: ATmega16L

ATmega32
32-Kbyte self-programming Flash Program Memory, 2-Kbyte SRAM, 1-Kbyte EEPROM, 8 channel 10-bit A/D, JTAG interface for on-chip-debug. Up to 16 MIPS throughput at 16 MHz. 5V operation.
3V version: ATmega32L

ATmega64
64-Kbyte self-programming Flash Program Memory, 4-Kbyte SRAM, 2-Kbyte EEPROM, 8 channel 10-bit A/D, JTAG interface for on-chip-debug. Up to 16 MIPS throughput at 16 MHz. 5V operation.
3V version: ATmega64L

ATmega128
128-Kbyte self-programming Flash Program Memory, 4-Kbyte SRAM, 4-Kbyte EEPROM, 8 channel 10-bit A/D, JTAG interface for on-chip-debug. Up to 16 MIPS throughput at 16 MHz. 5V operation.
3V version: ATmega128L

LCD AVR

ATmega169
16-Kbyte self-programming Flash Program Memory, 1-Kbyte SRAM, 512 Byte EEPROM, 8 channel 10-bit A/D, JTAG interface for on-chip-debug. 4 x 25 Segment LCD driver. Up to 16 MIPS throughput at 16 MHz. 5V operation.
3V version: ATmega169L
1.8V version: ATmega169V

TINYAVR

ATtiny11
1-Kbyte In-System programmable Flash Program Memory, 32 byte SRAM. Up to 6 MIPS throughput at 6 MHz.

ATtiny12
1-Kbyte In-System programmable Flash Program Memory, 32 Byte SRAM, 64 Byte EEPROM. Up to 12 MIPS throughput at 12 MHz.

ATtiny15L
1-Kbyte In-System programmable Flash Program Memory, 64 Byte EEPROM, 32 Byte Register File, 4 channel 10-bit A/D. Up to 1.6 MIPS throughput at 1.6MHz. 3V operation.

ATtiny26
2-Kbyte In-System programmable Flash Program Memory, 128 Byte SRAM and EEPROM, 11 channel 10-bit A/D. Universal Serial Interface. High Frequency PWM. Up to 16 MIPS throughput at 16 MHz. 5V operation.
3V version: ATtiny26L

Atmel AVR, MEGAAVR, LCD AVR, TINYAVR, USB AVR, SECURE AVR, DVD AVR, RF AVR and FPGA AVR Devices

ATtiny28L
2-Kbyte In-System programmable flash Program Memory, 128 Byte SRAM, 32 Byte Register File, Keyboard interrupt. Up to 4 MIPS throughput at 4 MHz. 3V operation.
1.8V version: ATtiny28V

USB AVR

AT43USB320A
512 Byte SRAM, Full Speed USB, 3 Function Endpoints, 4 Hub Ports. Up to 12 MIPS throughput at 12 MHz. 5V operation.

AT43USB325E/M
16-Kbyte EEPROM or Mask ROM, 512 Byte SRAM, Full Speed USB, 4 Function Endpoints, 4 Hub Ports, 5 LED Driver. Up to 12 MIPS throughput at 12 MHz. 5V operation.

AT43USB325
16-Kbyte Mask ROM, 512 Byte SRAM, Full Speed USB, 3 Function Endpoints, 2 Hub Ports, 4 LED Driver. Up to 12 MIPS throughput at 12 MHz. 5V operation.

AT43USB351M
24-Kbyte Mask ROM, 1-Kbyte SRAM, Low-Full Speed USB, 5 Function Endpoints. Up to 24 MIPS throughput at 24 MHz. 5V operation.

AT43USB353M
24-Kbyte Mask ROM, 1-Kbyte SRAM, Full Speed USB, 4 Function Endpoints, 2 Hub Ports. Up to 24 MIPS throughput at 24 MHz. 5V operation.

AT43USB355E/M
24-Kbyte EEPROM or Mask ROM, 1-Kbyte SRAM, Full Speed USB, 4 Function Endpoints, 2 Hub Ports. Up to 12 MIPS throughput at 12 MHz. 5V operation.

AT76C711
Full Speed USB to Fast Serial Asynchronous Bridge.

Secure AVR

AT90SC19236R
192-Kbyte Mask ROM, 36-Kbyte EEPROM, 4-Kbyte RAM. 3-5V operation.

AT90SC19264RC
192-Kbyte Mask ROM, 64-Kbyte EEPROM, 6-Kbyte RAM, Crypto Engine. 3-5V operation.

AT90SC25672R
256-Kbyte Mask ROM, 72-Kbyte EEPROM, 6-Kbyte RAM. 3-5V operation.

AT90SC320856
32-Kbyte Mask ROM, 8-Kbyte Flash, 56-Kbyte EEPROM, 1.5-Kbyte RAM. 3-5V operation.

AT90SC3232CS
32-Kbyte Flash, 32-Kbyte EEPROM, 3-Kbyte RAM, Crypto Engine. 3-5V operation.

AT90SC4816R/RS
48-Kbyte Mask ROM, 16-Kbyte EEPROM, 1.5-Kbyte RAM. 3-5V operation.

AT90SC6404R
64-Kbyte Mask ROM, 4-Kbyte EEPROM, 2-Kbyte RAM. 3-5V operation.

AT90SC6432R
64-Kbyte Mask ROM, 32-Kbyte EEPROM, 2-Kbyte RAM. 3-5V operation.

AT90SC6464C
64-Kbyte Flash, 64-Kbyte EEPROM, 3-Kbyte RAM, Crypto Engine. 3-5V operation.
USB version: AT90SC6464C-USB

Atmel AVR, MEGAAVR, LCD AVR, TINYAVR, USB AVR, SECURE AVR, DVD AVR, RF AVR and FPGA AVR Devices

AT90SC9608RC
96-Kbyte Mask ROM, 8-Kbyte EEPROM, 3-Kbyte RAM, Crypto Engine. 3-5V operation.

AT90SC9616RC
96-Kbyte Mask ROM, 16-Kbyte EEPROM, 3-Kbyte RAM, Crypto Engine. 3-5V operation.

AT90SC9636R
96-Kbyte Mask ROM, 36-Kbyte EEPROM, 3-Kbyte RAM. 3-5V operation.

AT97SC3201
Trusted Computing Platform Compliant Security Processor, On-Chip Secure Key Storage, 33 MHz LPC Interface. 3.3V operation

DVD AVR

AT78C1501
DVD/CD Interface Controller, ATAPI Compatible, Ultra DMA Support at 66 MB/sec.

AT78C1502
DVD Servo Controller, On-Chip Debugger Monitor. Up to 120 MIPS throughput at 40 MHz. 3V and 5V operation.

RF AVR

AT86RF401
11-19 MHz, 2-Kbyte In-System programmable Flash Program Memory, 128 Byte SRAM and EEPROM. 2V operation.

FPGA AVR

AT94K05AL
4-16 Kbyte In-System programmable Flash Program Memory, 4-16 Kbyte SRAM, JTAG interface for on-chip-debug, 5K FPGA Gates. 3V operation.

AT94K10AL
20-32 Kbyte In-System programmable Flash Program Memory, 4-16 Kbyte SRAM, JTAG interface for on-chip-debug, 10K FPGA Gates. 3V operation.

AT94K40AL
20-32 Kbyte In-System programmable Flash Program Memory, 4-16 Kbyte SRAM, JTAG interface for on-chip-debug, 40K FPGA Gates. 3V operation.

AT94S05AL
4-16 Kbyte In-System programmable Flash Program Memory, 4-16 Kbyte SRAM, 256 Byte EEPROM, JTAG interface for on-chip-debug, 5K FPGA Gates. 3V operation.

AT94S10AL
20-32 Kbyte In-System programmable Flash Program Memory, 4-16 Kbyte SRAM, 512 Byte EEPROM, JTAG interface for on-chip-debug, 10K FPGA Gates. 3V operation.

AT94S40AL
20-32 Kbyte In-System programmable Flash Program Memory, 4-16 Kbyte SRAM, 1-Kbyte EEPROM, JTAG interface for on-chip-debug, 40K FPGA Gates. 3V operation.

AVR Development Tools

AVR Studio 4.0

AVR Studio 4 is the Integrated Development Environment (IDE) for writing and debugging AVR applications in Windows 9x/NT/2000 environments. AVR Studio 4 includes an assembler and a simulator. The Studio supports the following tools: ICE50, ICE40, JTAGICE, ICE200, STK500/501/502 and AVRISP.

STK500

The STK500 is a starter kit and development system for Atmel's tinyAVR, megaAVR and LCD AVR devices. It gives designers a quick start to develop code on the AVR combined with features for using the starter kit to develop prototypes and test new designs. The STK500 interfaces with AVR Studio for code writing and debugging.

JTAGICE

The JTAGICE is an In-Circuit Emulator for Atmel's megaAVR and LCD AVR devices with 16K or more program memory. The JTAGICE communicates to the On-Chip debug module on the AVR which provides the most accurate emulation possible. The Emulator assists developers in identifying software bugs significantly reducing the development time. The JTAGICE interfaces with AVR Studio for code development and debugging.

JTAGICE

AVR Development Tools

ICE50

The AVR ICE50 is a top-of the-line development tool for in-circuit emulation of most megaAVR and LCD AVR, and tinyAVR devices. The ICE50 and the AVR Studio 4 user interface give the user complete control of the internal resources of the microcontroller, helping to reduce development time by making debugging easier.

86RF401E / U-EK1

The RF AVR evaluation kit was developed to familiarize the user with the features of the AT86RF401 MicroTransmitter and to provide all the tools needed to develop an application based on this device. Sample code is provided in the evaluation kit to speed development of this device, and allows the user to evaluate RF parameters without having to write software.

43DK355

The 43DK355 development kit has everything you need to develop your full-featured USB application using AT43USB355. It comes complete as a working hub with a programmable embedded USB function and up to four USB downstream ports. USB source code for an embedded function and a USB library for the HUB are provided, thus relieving the user from the tedious task of developing such code on their own.

ATV1-90SC

The ATV1 provides a flexible emulation platform to support current and future Secure AVR devices. For maximum flexibility, the system has been designed to be upgraded easily. Switching to support other devices within a given core family is done simply by running a software utility that guides the user through the device update process - there is no need to change PCBs or jumper settings.

STK94

The STK94 is a low-cost development kit for the designer who wishes to begin working with the FPGA AVR devices. A comprehensive tutorial takes designers through the complete FPGA AVR development process. The starter kit, which runs on PC Windows-based platforms, has everything needed to get started, including all software, boards, parts, cable, and documentation. ❑

Next Issue:

Third-Party AVR Emulators, Programmers, Consultants, and More!

Variable Message Sign Development with AVR and ImageCraft *continued from page 19*

10 bytes of 0x0AA (b10101010). This allows the receiving radio to synchronize with the incoming signal and stopped the first few bytes in the radio packet sometimes being lost. Another problem was noise on the output of the radio. Random radio output, for example, from the local taxi firm, could under some circumstances look like the start of a radio packet. This caused the radio receiving software function to get confused; it would see the start of a packet, shove the received data into a buffer, and then expect the remainder of the packet to arrive, which of course it never did. When a valid radio packet arrived the buffer was already half full of junk so the software went haywire. Although it was not so easy to track down this problem, it was fairly easy to fix. Since we know how long our maximum packet length is, and from the baud rate we could calculate how long the maximum packet should take to arrive from start to finish. By adding a simple timer that was started when the radio preamble and sync bytes were detected, we could delete the erroneous data from the buffer if the data did not arrive within the allotted time period.

Other functionality was added to the user diagnostic interface that allowed testing of various radio functions, which made development easier and has proven to be very helpful in fieldwork.

Sometimes Extensions Are Useful

So now we had our user terminal driven configuration interface, which was starting to have quite large, but well structured menu pages. Initially the standard C function, printf, was used to output the menu text. To preserve the semantics of the C language, ICCAVR by default allocates quoted literal strings (e.g. "HIGHWAY CLOSED") in the RAM space. With the large amount of strings in the menu system, the 4K bytes of RAM in the Mega128 were rapidly being used up. Fortunately, the Mega128 has a lot of flash (128K bytes). ICCAVR allows literal strings to be placed in the flash as an option. This means that some of the standard C functions will no longer operate on literal strings. This is a side effect of the Atmel AVR using a Harvard architecture: code and data are in separate address spaces (in flash and RAM respectively). While this allows the CPU core to run faster, it makes some common C usage slightly less convenient. Again ICCAVR came to the rescue by providing variants of the standard C functions that are compatible with flash based literal strings. For example, cprintf is used in place of the printf function. Again, we see that it is important to have the right extensions to C without polluting the source code unduly. Our radio communications were now operating quite happily and reliably with our user interface, and so far we were still using the STK500 and only 18% of the Mega128 memory.

Down the Home Stretch

Next was the expansion board that would interface between the characters and main controller. This used the 8535, and again the ICCAVR Applications Builder was used to generate all the code required for setting up the peripherals.

A simple SPI bus interface was used for communication between the controller and expansion boards with one slight change: the drive from the expansion board to the controller needed to be tri-state, as there would be more than one expansion board on the SPI bus. This was simply achieved by the use of an open collector transistor and some simple buffer circuits at the controller end. Each board needed to be addressable so a simple thumb-operated rotary HEX switch was added which gave 16 address ranges. It was decided that 0 would never be a valid address, so a handy little test function was added where if the expansion board was powered up with address 0 selected it entered a test mode and just cycled through A-Z and 0-9 without the need for a controller. This made basic testing of the expansion boards and characters very simple. It was decided that the character bit patterns required for the LEDs would be stored as bitmaps within the expansion boards' 8535 flash memory. This would allow for custom characters or different languages to be stored, or even for creating a bigger image from a number of characters working together in panels. Unfortunately this meant the relatively small 8 K bytes of flash in the 8535 was being pushed to the limit. Again ICCAVR came to the rescue, with yet another feature that had not been needed so far, the ICCAVR code compressor. This allowed around a 7-9% reduction in code space, which prevented the need for a bigger and more expensive AVR device.

The initial development for the expansion board was all done on the STK500, and indeed, if there had been two STK500s available, a lot more system development could have been done. At this point it was decided to make some prototype PCBs, starting with the controller card. The first board was built; given some power (on a current limited PSU!!), and plugged into the ISP lead. We pressed the ISP button on ICCAVR, and then just happily watched the programming progress bar work its way across the screen, to the customers' amusement. The PCBs work perfectly the first time! Perhaps the gremlins had taken a vacation elsewhere.

Even 8-Bit Embedded Systems Can Speak TCP/IP and HTML

The final and possibly the most complex part of the project was implementing a TCP/IP interface serving up embedded HTML pages giving the status and control of remote devices. The embedded web server is a separate box residing in the control room attached to the customer's local network. Given its own IP and MAC address, it receives data over a TCP interface converts the data into the radio packet and sends the relevant packet data over the radio network to the receiving node. The HTML interface allows anyone with correct passwords to view the status of remote outstations, and reconfigure certain parameters.

The TCP interface was implemented using a Mega128 with an external 32K RAM chip and a RTL8019AS Ethernet driver. The TCP stack is a trimmed down implementation, but supports ARP, ICMP, UDP and TCP communications. It also has a dynamic HTML interface on the application layer. The dynamic aspect of the interface is that each web page is generated on the fly when requested, rather than being a static page retrieved from the Flash of some external EEPROM. This allows for up-to-date system information to be displayed, such as temperature, radio / GSM signal strength, number of radio / GSM packets processed, TCP status for received, sent and lost / corrupt packets, etc.

Even though a limited TCP stack was implemented, it was still possible to achieve, the TCP sliding window, detection of lost packets, and automatic retries, with adaptive timeouts to cater for different roundtrip delays across different faster or slower networks. The HTML pages as generated by the AVR support buttons, tick boxes and radio buttons, and text fields where user information can be entered. It also supports GIF, animated GIF and JPEG images, and colour bar graphs. The complete TCP interface with a large quantity of HTML web pages including graphics still only takes less than 45% of the Mega128 memory.

Conclusion

All in all, from the start of nothing more than a rough drawing to equipment being signed off and in use on the UK highways, the whole project was completed in 6 months. This would not have been possible without quality development tools at reasonable prices like the STK500 / 501 and ImageCraft's ICCAVR C compiler with ImageCraft's fantastic after-sales support. ❑

Device Drivers and the Special Function Register Hell *continued from page 18*

6. If you want to view the files generated for USART, you can find the USART module in the project explorer. Here you can open the <ma_usart.c> or <ma_usart.h> files, and see the device driver source code.
7. To generate source code and header files to disk, click the code generation button in the toolbar.
8. Add the generated MakeApp files to your application project, and, with out having to read or understand the special function register implementation for the USART, you are now ready to use the USART in your product. Let your application call the MA_InitCh0_USART() function. Once the USART is initialized, the MA_PutStringCh0_USART() function will be ready to output your "Hello world" message.

Product development and product life cycle

IAR MakeApp can be used at all stages in a product, from the first idea, evaluation, and design, to the final test and maintenance of the product. It also becomes easy to tune system parameters throughout the entire life cycle of your key products. So not only do you get to market sooner, but, once there, you can easily and cost-effectively refine and improve your product offering. It is never too late to start using IAR MakeApp.

IAR MakeApp for Atmel megaAVR

The current product supports: ATmega128(L), ATmega64(L), ATmega32(L), ATmega16(L), and ATmega8(L).

The property dialog boxes support the configuration of:

- CPU - Bus control and memory
- INTC - Interrupt controller
- WDT - Watchdog timer
- I/O - I/O ports
- TMR - Timers, counters
- SPI - Serial peripheral interface
- USART - Universal synchronous/asynchronous serial communication
- TWI - Two-wire serial interface
- ACOMP - Analog comparator
- ADC - Analog to digital converter

General IAR MakeApp features:

- CAD-like drawing editor - As devices are configured, their pin usage is displayed graphically.
- Project explorer – Gives a tree view of the current project.
- Component library – Contains the necessary information about the components that can be configured.
- Property dialogues – Organized as a set of tabs with headings and property lists.
- Code generation engine – Contains a powerful code generation technology. It automatically calculates the special function register values, and modifies/optimizes the generated source code according to the property settings.
- Component browser - Gives extensive information on chip-internals, such as SFR bits, port pins, interrupts, memory maps, etc.
- Data book browser – Lists all known hardware manuals in PDF format in the databook catalog.
- Report generation and report viewer – The project report contains detailed information on chip resources (such as SFRs, pins, and interrupts), as well as the project settings (such as configuration, generated device driver functions, and function dependencies).
- Low product price!

AT86RF401 Reference Design *continued from page 27*

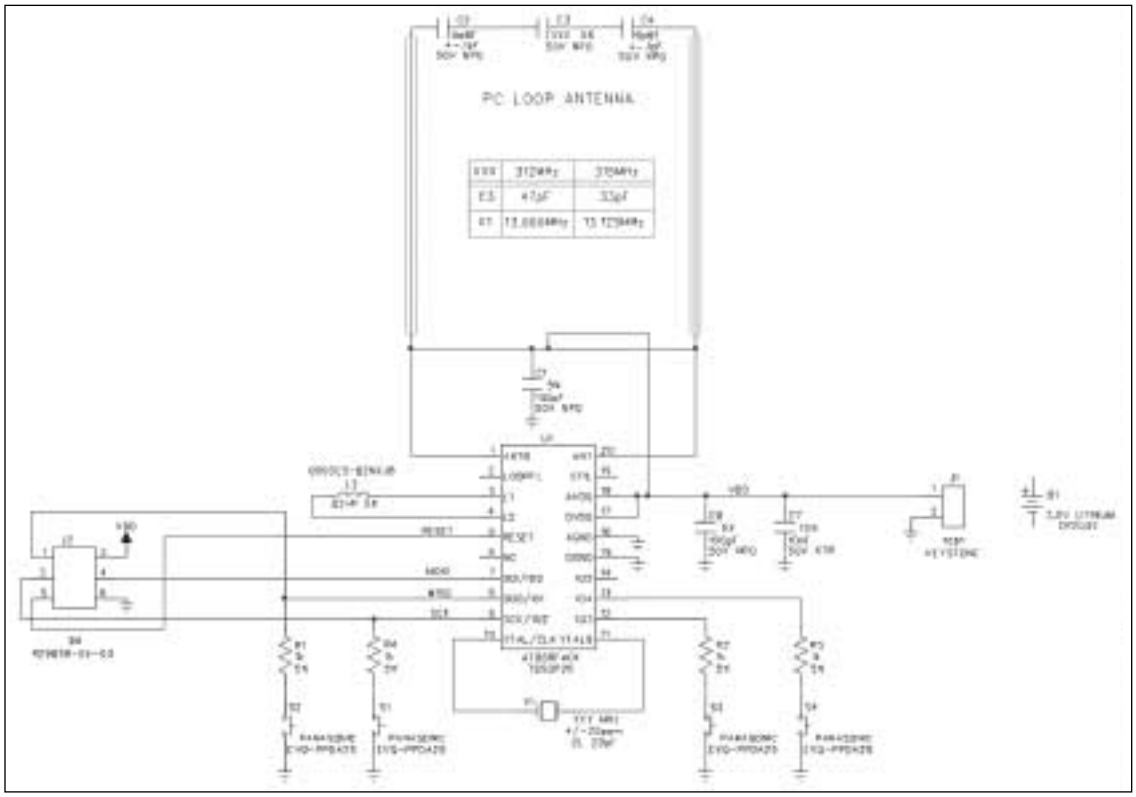


Figure A

A *triple latte*

or a

full year of
Circuit Cellar *magazine...*

Both will keep you up all night,
but only one will leave you with design solutions in the morning.
And now you can have both of them for around the same price.

For more than 15 years, Circuit Cellar has presented some of the most groundbreaking product applications in the embedded systems industry. It's easy when your goal is to give the readers what they pay for: innovative design solutions from the field, including schematics and code. Circuit Cellar articles feature in-depth analysis and honest opinions. If you haven't taken advantage of this information pipeline, you won't want to miss this special offer.

To help jump-start your AVR® applications, Atmel® has obtained reduced-cost Circuit Cellar Electronic Edition subscriptions. For only $5, you'll receive 12 monthly PDF files that mirror the Circuit Cellar print magazine. With the Electronic Edition, you can search by keywords and access additional information by clicking on helpful hyperlinks. You can even download the issue one article at a time.

So skip the coffee and order something that will help you meet your design deadlines without the caffeine. Sign up for your $5 subscription today!

www.circuitcellar.com/atmel03

Faster, smoother development for Atmel AVR

IAR SYSTEMS 1983 20 YEARS 2003 IN CODE WE TRUST

Smart enough to choose the Atmel AVR? Be smart enough to speed your application to market with IAR's highly competitive, easy-to-use development tools, fully supporting the AVR microprocessor.

IAR Embedded Workbench® for Atmel AVR
The state-of-the-art integrated development environment with a highly optimised IAR C/EC++ compiler and versatile IAR C-SPY source and assembly level debugger

IAR visualSTATE® for Atmel AVR
The only state machine design tool for embedded systems generating micro-tight C code

IAR MakeApp® for Atmel megaAVR
The device driver wizard generating initialisation code and full driver functions

IAR SYSTEMS
From Idea to Target

Visit us at **www.iar.com**

Don't worry about your upgrade path! IAR development tools also support the Atmel ARM 32-bit processor

LIT#3362A-CORP-VOL.1-20M