i-easy pro

PPP-TCP/IP MODULE

PROGRAMMING & APPLICATION
GETTING STARTED GUIDE

V 1.3

embedded-inet
SOLUTIONS

ABOUT **3**

AVR-GCC AND AVR STUDIO **3**

About AVR-GCC 3
About AVR Studio 3
Installing AVR Studio 3
Installing AVR-GCC 4

THE IEASY PROJECT IN AVR STUDIO **5**

General 5
Getting started 5
Customise demo.c 5
ichip.h 6
uart.h 6
The makefile 7

CONFIGURING AVR STUDIO TO USE AVR-GCC **9**

Using Win9x 9
Using Win2000 or WinXP 9
Target Options for AVR Studio 9

BUILDING THE IEASY PROJECT **10**

FLASHING THE I-EASY PRO WITH LETATWORK PROGRAMMER AND DEBUG ADPATER (INCLUDED IN THE STARTERKIT) **10**

FLASHING THE I-EASY WITH OPTIONAL STK500 COMPATIBLE PROGRAMMER (ATAVRISP) **11**

I-EASY PROGRAMMING **13**

Introduction 13
How does an internet connection work? 13
Transmission Control Protocol 13
User Datagram Protocol 14
Service ports 15
Sockets 15
Client / Server applications 16
Functional overview 17
i-easy demo and i-easy lib v 0.4 17
Establishing a connection 18
Using i-easy socket for client connection 18

INFORMATION RECOURCES **20**

About

This manual should help you to get the first steps of working with the i-easy.

The first time steps described here are
Installing the AVR-GCC Compiler.
Installing and configuring AVR Studio (an IDE).
(You are of course free to use your other compilers and IDE.)
Opening, customizing and building the ieasy demo project, and flashing the module.

After that we will have a closer look at the principles of TCP/IP and give a first impression of the i-easy lib. A closer look to the ieasy lib will be taken in the `refman.pdf`.

AVR-GCC and AVR Studio

About AVR-GCC

AVR-GCC is a freeware Ansi C compiler (and assembler) for AVR that is available under the GNU public license. GCC is short for "GNU Compiler Collection".

For most recent releases you might check

`http://combio.de/avr/`	AVR-GCC Linux(/Windows) Archives
`http://www.avrfreaks.net/AVRGCC/`	AVR-GCC Windows Archives
`http://www.amelek.gda.pl/avr/libc/`	Lib-C

About AVR Studio

AVR Studio 3.x is an Integrated Development Environment (IDE) for writing and debugging AVR applications in Windows 9x/NT/2000/XP environments. AVR Studio is supplied by Atmel and supports the AVR JTAG ICE, the AVRISP, and the ICE10 including on-line help.

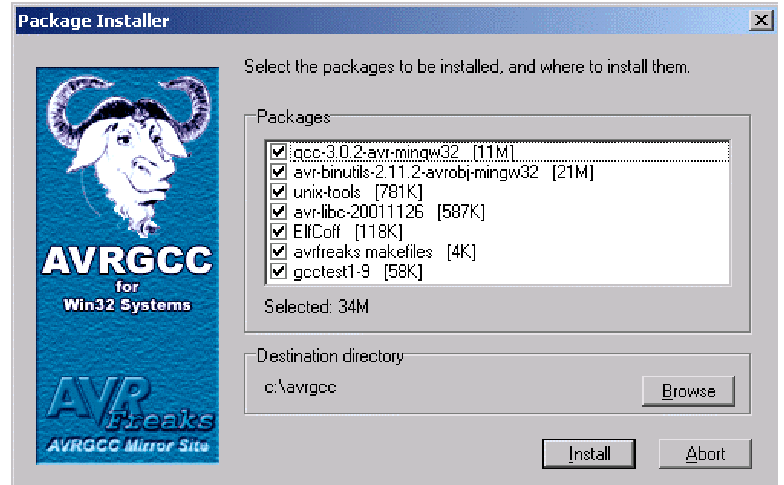
You will find a copy of AVR Studio on the optiCompo CD-ROM in folder `/IDE/Atmel`.
In future times you may get updates at
`http://www.atmel.com/atmel/products/prod203.htm`

Installing AVR Studio

Run the self-extracting archive `astudio3.exe` and unpack it to a folder of your choice, e.g. “`c:\astudio\`”.
Enter the directory “`c:\astudio\cdrom\`”, start `SETUP.EXE` and follow the InstallShield instructions.

Installing AVR-GCC

This is also a fairly straightforward process; just run the executable `avrgcc_freaks20011214a.exe` (or higher; the "freaks" version additionaly includes the usefull gcctest-files and ElfCoff).
For convenience, choose the default install location;
c:\avrgcc.



Make sure to leave all boxes checked. You may not think you need the "Unix Tools", but these include also the *make.exe* and *rm.exe* utilities; you *will* need them.

Also, it is important to make a few checks to ensure that the installation, including the compilation of all libraries are completed:

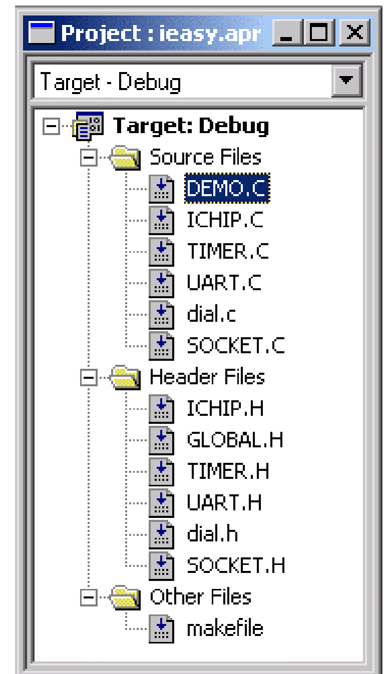
- During installation, a DOS-window should appear, dumping lines of compiler output to the screen for about a minute or so.
- When compilation is done, another DOS window appears, declaring that "your pop-up program is ready to run". Close this window by typing ctrl-c. **Note:** this does *not* happen on Windows2000.
  If this does NOT occur, there could be a problem. The object libraries may not have been compiled properly. In this case, you need to run the file `run.bat` manually. It can be found in the directory where you installed avr-gcc.
- Provided you chose the default installation target directory to be "`c:\avrgcc`", the directory "`c:\avrgcc\avr\lib`" should be crowded with .o files dated around the time you ran the installation. If not, execute `run.bat` manually.

The ieasy project in AVR Studio

General

We will now take the prepared demo project “`ieasy2.apr`” from the `ieasyII.zip` file, compile it and flash the i-easy.

Note: This introduction will give you a first hint on how to proceed. General recipies for using AVR-GCC with AVR Studio can be found e.g. in the `avrgcc_studio.pdf` available from `avrfreaks.net`, which is also included on the CD-ROM.



Getting started

Unzip the `ieasyII.zip` file on the CD-ROM (“`d:\ieasy\ieasy_avrgcc_XXX.zip`”) to your desired project directory, e.g. “`c:\avrgcc\my_projects\i-easyII_V0.4\i-easy0.4\`”.

On the *Project* menu of AVRstudio click *Open*. Select the filetype *Project Files (*.apr)*, choose your project directory like above and select *ieasy2.apr* .
[Note: AVR Studio will not find the files in case you choose a different directory, as the pathes are set absolutely. You can put all together on your own by right-clicking on the folder icons to choose “Add File…”.]

Most actions can be accessed by right-clicking on a name.
To open an editor window double-click the file name.

Customise demo.c

The demo application sends an e-mail and connects a time server using the i-easy.
To perform a first dial-up you have to specify a few variables in the `demo.c` and if necessary in the `ichip.h` and `uart.h` file.
Simply doubleclick the file name to get an editor window.

- ```
  void time_test(void)
  {
     …
     IP_PORT time_server_ip = {192,53,103,103,DAYTIME_PORT_H,DAYTIME_PORT_L};   (line 147)
     …
  }
  ```
  You do not need to change the time server IP, but it might be usefull to use a time server that resides in your local time zone; the given example is the IP of the PTB (an institute for standards) time server in Germany.
  ```

- ```
  u08 email_test(void)
  {
     …
     char __attribute__ ((progmem))  *emailcmd0="HELO login\r\n";               (line 177)
     char __attribute__ ((progmem))  *emailcmd1="MAIL FROM:<sender@senderhost.com>\r\n";
                                               // enter here the sender e-mail address
     char __attribute__ ((progmem)) *emailcmd2="RCPT TO:<receiver@receiverhost.com>\r\n";
                                               // enter here the receiver e-mail address
     char __attribute__ ((progmem))  *emailcmd3="DATA\r\n";
     char __attribute__ ((progmem))  *emailcmd4="subject: i-easy status\r\n";
                                               // e-mail subject
     char __attribute__ ((progmem))  *emailcmd5="Hi, email by i-easy\r\n";
                                               // first line of body
     char __attribute__ ((progmem))  *emailcmd6="\r\n.\r\n";
     …
  }
  ```
  In these lines specify sender and recipient e-mail addresses.
  If you like you can modify the mail content. (Keep the parentheses `" "`.) **Note:** The last line of your e-mail has to be `"\r\n.\r\n"`, otherwise you get into some problems.

- ```
  u08 email_test(void)
  {
     …
     IP_PORT smtp_server_ip = {1,2,3,4,SMTP_PORT_H,SMTP_PORT_L};               (line 189)
     …
  }
  ```
  Fill in your SMTP server IP and port (sendmail server); if necessary consult your provider to get the IP number.

- ```
  int main(void)
  {
     …
     status=connectModem("ATX0","ATDT1234567890");                              (line 273)
     …
     PPP_open("username", "secret");                                             (line 275)
     …
  }
  ```
  To dial up to the ISP replace the exemplary "modem init string" and "modem dialstring" with parameters, which suite your modem and provider.
  Replace `"username", "secret"` with your e-mail login and password.

ichip.h
Check for the correct modem baud rate (max. 115200 baud) in `ichip.h`.

```
#define BaudRate                19200                                            (line 75)
```

uart.h
If you like to see the debug messages on AVR TXD pin (PD1) check for the correct uart baud rate (max. 115200) in `uart.h`. It should match your terminal settings.

```
#define UART_BAUD_RATE          115200                                           (line 24)
```
```

The makefile

Let's have a look at the makefile. (Doubleclick the file name to get an editor window.)

```
# Tools and directories
CC            = avr-gcc
# AS          = avr-gcc -x assembler-with-cpp
RM            = rm -f
RN            = mv
BIN           = avr-objcopy
INCDIR = .
LIBDIR = $(AVR)/avr/lib
SHELL   = $(AVR)/bin/sh.exe
FORMAT  = srec
SILENT =

###########################################################################################
###########################################################################################

# CPU type
MCU = atmega323

# Target
TRG    = demo

# C-source files
SRC    = ichip.c timer.c uart.c dial.c socket.c $(TRG).c

# Assembler source files
ASRC    =

# Libraries
LIB = $(AVR)/avr/lib/libc.a

# Compiler flags
CPFLAGS       =  -g -O3 -Wall -Wstrict-prototypes -Wa,-ahlms=$(<:.c=.lst)

# Assembler flags
ASFLAGS = -Wa,-gstabs

# Linker flags
LDFLAGS = -Wl,-Map=$(TRG).map,--cref


###########################################################################################
###########################################################################################

#define all project specific object files
       OBJ    = $(ASRC:.s=.o) $(SRC:.c=.o)
       CPFLAGS += -mmcu=$(MCU)
       ASFLAGS += -mmcu=$(MCU)
       LDFLAGS += -mmcu=$(MCU)


#this defines the aims of the make process
all:   $(TRG).obj $(TRG).elf $(TRG).rom


#compile: instructions to create assembler and/or object files from C source
%o : %c
       $(SILENT)$(CC) -c $(CPFLAGS) -I$(INCDIR) $< -o $@

%s : %c
       $(SILENT)$(CC) -S $(CPFLAGS) -I$(INCDIR) $< -o $@

#assemble: instructions to create object file from assembler files
%o : %s
       $(SILENT)$(AS) -c $(ASFLAGS) -I$(INCDIR) $< -o $@
```

```
#link: instructions to create elf output file from object files
%elf: $(OBJ)
	$(SILENT)$(CC) $(OBJ) $(LIB) $(LDFLAGS) -o $@

#create avrobj file from elf output file
%obj: %elf
	$(SILENT)$(BIN) -O avrobj $< $@

#create bin (ihex, srec) file from elf output file
%rom: %elf
	$(SILENT)$(BIN) -O $(FORMAT) $< $@
	$(SILENT)avr-size $(TRG).elf

#make instruction to delete created files
clean:
	$(RM) $(OBJ)
	$(RM) $(TRG).map
	$(RM) $(TRG).elf
	$(RM) $(TRG).obj
	$(RM) $(TRG).eep
	$(RM) $(TRG).rom
	$(RM) *.bak
	$(RM) *.lst
	$(RM) *.?_sym

###########################################################################################
###########################################################################################

########### you should not need to change the following line #############
include $(AVR)/avrfreaks/avr_make

###### dependecies, add any dependencies you need here ###################
$(TRG).o : $(TRG).c timer.h global.h uart.h ichip.h dial.h socket.h
ichip.o : ichip.c ichip.h global.h dial.h
timer.o : timer.c global.h
uart.o  : uart.c global.h timer.h
dial.o  : dial.c global.h timer.h ichip.h
socket.o : socket.c global.h timer.h ichip.h
```

Note the line saying "`include $(AVR)/avrfreaks/avr_make`" in the makefile. This line takes care of including more dependencies and commands for the make process, from the file "*avr_make*". This file is included in the avr-gcc distribution from AVRfreaks.

Configuring AVR Studio to use AVR-GCC

For this we feed the right makefile to *make* using a .bat file and tell AVRstudio the path of the GNU tools by setting environment parameters.

Using Win9x

Copy the `gcc_cmp.bat` file (included in "`c:\AVRGCC\avrfreaks`" subdir) to "`c:\windows`".

This file will set some important environment parameters for the *make* utility, then *make* will be run.

Using Win2000 or WinXP

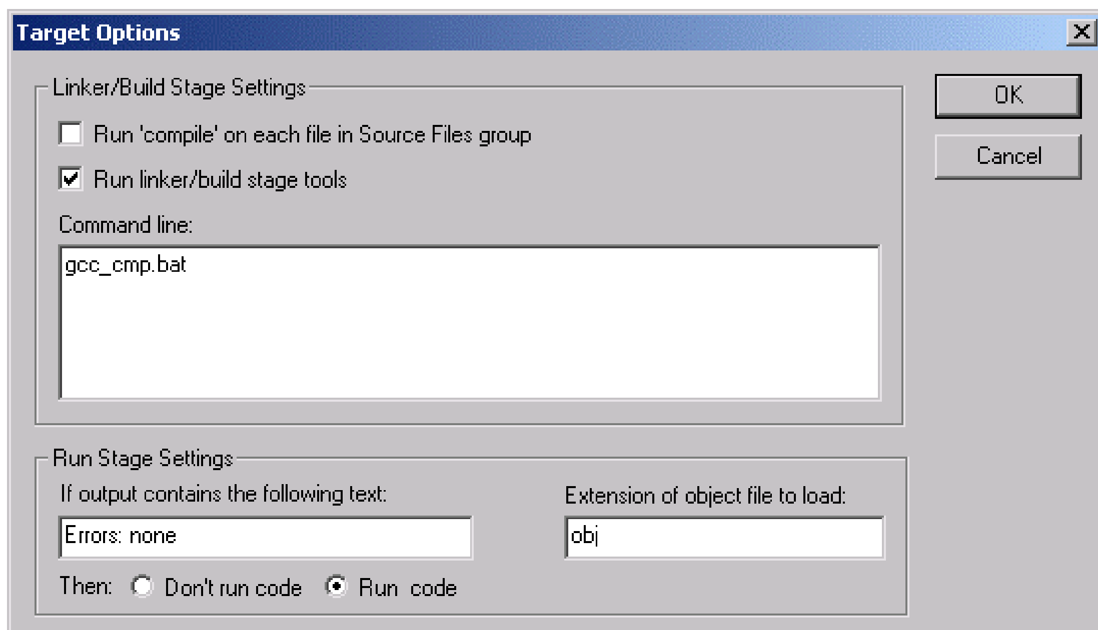
For Win2000, you need an additional "start"-file to kick off the compilation. Both files are available in the "`\avrfreaks\win2000`" subdir of your avrgcc installation.

Save "`gcc_cmp.bat`" and "`gcc_cmp2.bat`" in "`c:\winnt`", or some other folder you know is included in your Windows path.

This setup will start make.exe, which is instructed to direct its screen dump to a file that is displayed as the compiler output inside AVRstudio, and then deleted.

Target Options for AVR Studio

Right-click "Target:debug" in the Project window, and select "Settings". Verify the following:
- "Run 'compile'…" is unchecked.
- "Run linker…" is checked.
- The command line window says "`gcc_cmp.bat`".
- "Run stage settings": opt for "Run code". The first text box says "`Errors: none`", the second says "`obj`" for object file extension.



Building the ieasy project

To build the project:

- Right-click "Target: debug" in the project window of AVRstudio, and select "build" from the bottom of the menu.

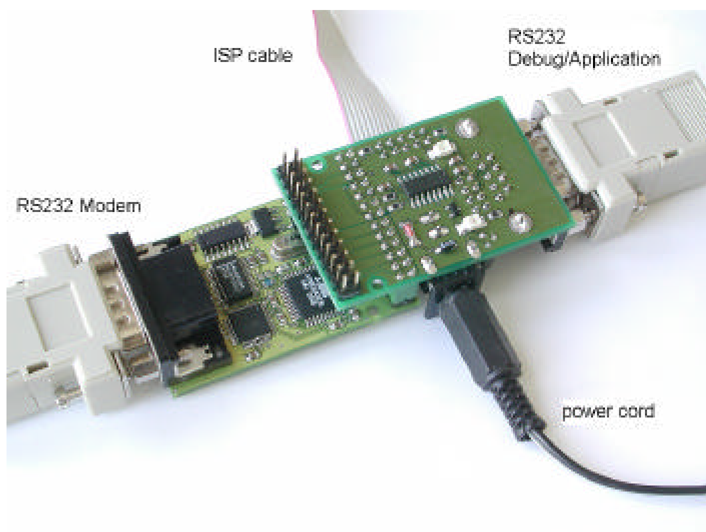
Provided you completed all we went through so far as you should, this project should build just fine.
Watch the the *make* output appearing in a window inside AVRstudio. As long as it doesn't report any errors, all is OK.

```
-------- begin --------
c:\avrgcc\avrfreaks\avr_make:91: warning: overriding commands for target `clean'
makefile:86: warning: ignoring old commands for target `clean'
avr-objcopy -j .eeprom --set-section-flags=.eeprom="alloc,load" --change-section-lma .eeprom=0 -O ihex° demo.elf demo.eep
avr-size demo.elf
  text°  data°  bss°  dec°  hex°filename
  4222°  264°  171°  4657°  1231°demo.elf
Errors: none
-------- end --------
```

Now you can transfer (flash) the built hex-file to your i-easy module.

Flashing the i-easy pro with letATwork Programmer and Debug Adpater (included in the Starterkit)

Make sure the i-easy is connected and powered like in the picture. Connect the letATwork Programming Dongle to the Debug Adapter ISP Port.



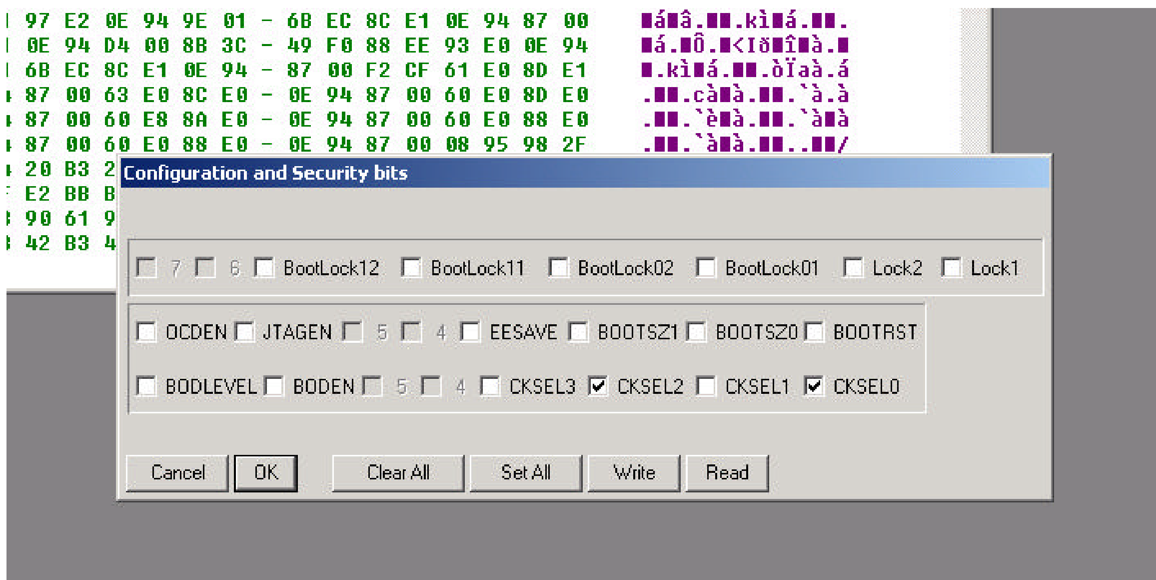
To perform the first tests it make sense to connect to a Hayes compatible Modem and via the Debug RS232 Port to a Computer (running a terminal software). So you can see function and results of the demo application.

Connect the letATwork ISP to your computers printer port (bidirectional Port).

Using **PonyProg2000** (found on the CD as ISP burning tool) go to the Setup menu select Parallel Port , AVR ISP/I/O Driver and the used LPT Port. Now go to the Device Menu and select the AVR micro/Atmega323 MCU.

Load the compiled binary hex file with File/Open Device File, select Filetype .hex and load the demo.hex file from the demo source directory.

Check the fuses of the Mega323 MCU at Command/Security and Configuration Bits with READ. Now you should see the fuses setting like in the picture. Very important are the CKSEL bits and the JTAGEN bit.



Pony Prog 2000, Mega323 Fuses for i-easy pro

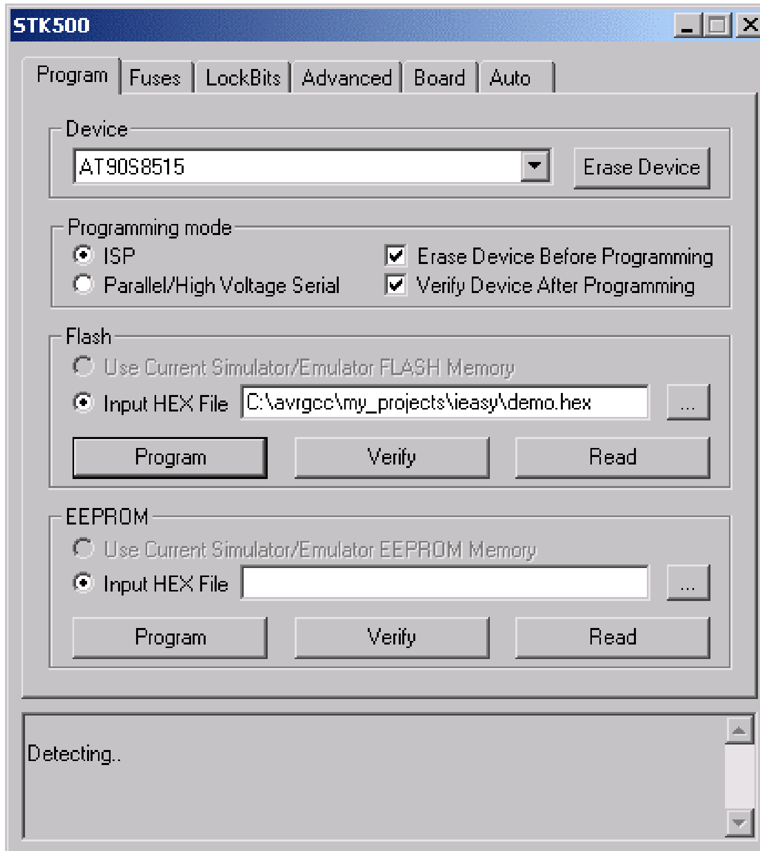
An alternative programmer setup is the ATAVRISP Programmer by Atmel, the next chapter describes the use of this device (not needed with the starterkit).

Flashing the i-easy with optional STK500 compatible Programmer (ATAVRISP)

Make sure the i-easy is connected and powered.

Connect the ATAVRISP to the Debug Adapter ISP Port.

To load your newly built code onto the chip, click the little AVR chip icon on the toolbar of AVRstudio, or open the *STK500 tool menu* by selecting “STK500/… (Alt+8)” from the “Tools” menu:



You can probably leave most of the settings to their default values.

- Select the “ATmega323” device.
- In the “Flash” section of the dialog box; browse to the file “demo.hex” in your project directory.
- Click the “Program” button.
- Make sure the output textbox at the bottom of the dialog says everything went OK.
- **Fuses**: Please be carefull when burning the fuses for ATmega323 – do not enable JTAG as some JTAG ports are used for the iChip bus and switch the clk source to external crystal.

i-easy Programming

Introduction

After installation of the AVR Studio (IDE) and GCC (Compiler) and hopefully first success with the demo application we can start the programming of the i-easy module now.
Starting programming with i-easy needs an introduction on how internet connections work (next chapter), the functional structure of the i-easy module and an instruction overview on the I-easy lib.

How does an internet connection work?

To get an overview of the special fuctions and possibilities of the iChip you need a rudimentary overview on the principles of a client server connection via TCP/IP.
If you know all about it you can skip this chapter.

Transmission Control Protocol

When two computers wish to exchange information over a network, there are several components that must be in place before the data can actually be sent and received. Of course, the physical hardware must exist, which typically includes network interface cards (NICs) and wiring of some type to connect them. Beyond this physical connection, however, computers also need to use a protocol which defines the parameters of the communication between them. In short, a protocol defines the "rules of the road" that each computer must follow so that all of the systems in the network can exchange data. One of the most popular protocols in use today is TCP/IP, which stands for Transmission Control Protocol/Internet Protocol.

In case you have a modem dial-up connection (e.g. to an internet service provider) you can't use TCP/IP directly. You need a tunneling protocol called PPP which tunnels generated TCP/IP packages to the internet gateway of your provider. This job is done by the Seiko iChip for you. So we only need to talk about TCP/IP.

By convention, TCP/IP is used to refer to a suite of protocols, all based on the Internet Protocol (IP). Unlike a single local network, where every system is directly connected to each other, an internet is a collection of networks, combined into a single, virtual network. The Internet Protocol provides the means by which any system on any network can communicate with another as easily as if they were on the same physical network. Each system, commonly referred to as a host, is assigned a unique 32-bit number which can be used to identify it over the internetwork. Typically, this address is broken into four 8-bit numbers separated by periods. This is called dot-notation, and looks something like "192.43.19.64". Some parts of the address are used to identify the network that the system is connected to, and the remainder identifies the system itself. Without going into the minutia of the Internet addressing scheme, just be aware that there are three "classes" of addresses, referred to as "A", "B" and "C". The rule of thumb is that class "A" addresses are assigned to very large networks, class "B" addresses are assigned to medium sized networks, and class "C" addresses are assigned to smaller networks (networks with less than approximately 250 hosts).

When a system sends data over the network using the Internet Protocol, it is sent in discrete units called datagrams, also commonly referred to as packets. A datagram consists of a header followed by application-defined data. The header contains the addressing information which is used to deliver the datagram to it's destination, much like an envelope is used to address and contain postal mail. And like postal mail, there is no guarantee that a datagram will actually arrive at it's destination. In fact, datagrams may be lost, duplicated or delivered out of order during their travels over the network. Needless to say, this kind of unreliability can cause a lot of problems for software developers. What's really needed is a reliable, straight-forward way to exchange data without having to worry about lost packets or jumbled data.

To fill this need, the Transmission Control Protocol (TCP) was developed. Built on top of IP, TCP offers a reliable, full-duplex byte stream which may be read and written to in a fashion similar to reading and writing a file. The advantages to this are obvious: the application programmer doesn't need to write code to handle dropped or out-of-order datagrams, and instead can focus on the application itself. And because the data is presented as a stream of bytes, existing code can be easily adopted and modified to use TCP.

TCP is known as a connection-oriented protocol. In other words, before two programs can begin to exchange data they must establish a "connection" with each other. This is done with a three-way handshake in which both sides exchange packets and establish the initial packet sequence numbers (the sequence number is important because, as mentioned above, datagrams can arrive out of order; this number is used to ensure that data is received in the order that it was sent). When establishing a connection, one program must assume the role of the client, and the other the server. The client is responsible for initiating the connection, while the server's responsibility is to wait, listen and respond to incoming connections. Once the connection has been established, both sides may send and receive data until the connection is closed.

User Datagram Protocol

Unlike TCP, the User Datagram Protocol (UDP) does not present data as a stream of bytes, nor does it require that you establish a connection with another program in order to exchange information. Data is exchanged in discrete units called datagrams, which are similar to IP datagrams. In fact, the only features that UDP offers over raw IP datagrams are port numbers and an optional checksum.

UDP is sometimes referred to as an unreliable protocol because when a program sends a UDP datagram over the network, there is no way for it to know that it actually arrived at it's destination. This means that the sender and receiver must typically implement their own application protocol on top of UDP. Much of the work that TCP does transparently (such as generating checksums, acknowledging the receipt of packets, retransmitting lost packets and so on) must be performed by the application itself.

With the limitations of UDP, you might wonder why it's used at all. UDP has the advantage over TCP in two critical areas: speed and packet overhead. Because TCP is a reliable protocol, it goes through great lengths to insure that data arrives at it's destination intact, and as a result it exchanges a fairly high number of packets over the network. UDP doesn't have this overhead, and is considerably faster than TCP. In those situations where speed is paramount, or the number of packets sent over the network must be kept to a minimum, UDP is the solution.

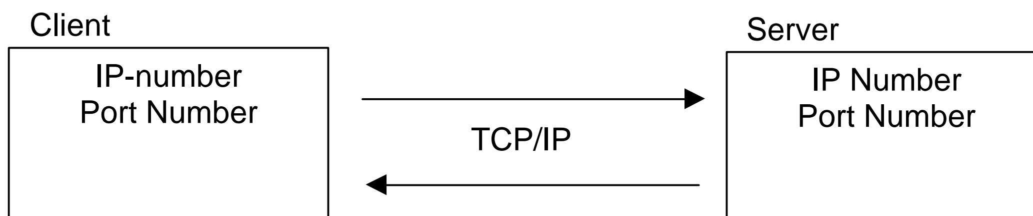
Service ports

In addition to the IP address of the remote system, an application also needs to know how to address the specific program that it wishes to communicate with. This is accomplished by specifying a *service port*, a 16-bit number that uniquely identifies an application running on the system. Instead of numbers, however, service names are usually used instead. Like hostnames, service names are usually matched to port numbers through a local file, commonly called *services*. This file lists the logical service name, followed by the port number and protocol used by the server. In case of the i-easy the `ichip.h` includes some predefined service ports.

A number of standard service names are used by Internet-based applications and these are referred to as *well-known services*. Some common services are:

Service Name	Port	Function
echo	7	Used to echo data back to the program that sent it. This is commonly used to test an application to make sure that a network connection can be established successfully.
POP3	10	Post Office Protocol - Version 3 (receiving mails)
daytime server	13	
ftp	20 and 21	Used to transfer files between computer systems using the File Transfer Protocol.
telnet	23	Used to provide terminal emulation services for the remote host.
smtp	25	Used to send electronic mail to a remote host using the Simple Mail Transfer Protocol.
HTTP	80	world wide web http (hypertext transfer protocol)

Remember that a service name or port number is a way to *address* an application running on a remote host. Because a particular service name is used, it doesn't guarantee that the service is available, just as dialing a telephone number doesn't guarantee that there is someone at home to answer the call.



Sockets

The previous sections described what information a program needs to communicate over a TCP/IP network. The next step is for the program to create what is called a *socket*, a communications end-point that can be likened to a telephone. However, creating a socket by itself doesn't let you exchange information, just like having a telephone in your house doesn't mean that you can talk to someone by simply taking it off the hook. You need to establish a

connection with the other program, just as you need to dial a telephone number, and to do this you need the *socket address* of application that you want to connect to. This address consists of three key parts: the *protocol family*, *Internet Protocol (IP) address* and the *service port number*.

We've already talked about the IP address and service port, but what's the protocol family? It's a number which is used to logically designate the group that a given protocol belongs to. Since the socket interface is general enough to be used with several different protocols, the protocol family tells the underlying network software which protocol is being used by the socket. With the protocol family, IP address of the system and the service port number for the program that you want to exchange data with, you're ready to establish a connection.

On i-easy you can use two sockets at a time.

Client / Server applications

Programs written to use TCP are developed using the *client-server model*. When two programs wish to use TCP to exchange data, one of the programs must assume the role of the client, while the other must assume the role of the server. The client application initiates what is called an *active open*. It creates a socket and actively attempts to connect to a server program. On the other hand, the server application creates a socket and passively listens for incoming connections from clients, performing what is called a *passive open*. When the client initiates a connection, the server is notified that some process is attempting to connect with it. By *accepting* the connection, the server completes what is called a *virtual circuit*, a logical communications pathway between the two programs. It's important to note that the act of accepting a connection creates a new socket; the original socket remains unchanged so that it can continue to be used to listen for additional connections. When the server no longer wishes to listen for connections, it closes the original passive socket.

To review, there are five significant steps that a program which uses TCP must take to establish and complete a connection. The server side would follow these steps:

1. Create a socket.
2. Listen for incoming connections from clients.
3. Accept the client connection.
4. Send and receive information.
5. Close the socket when finished, terminating the conversation.

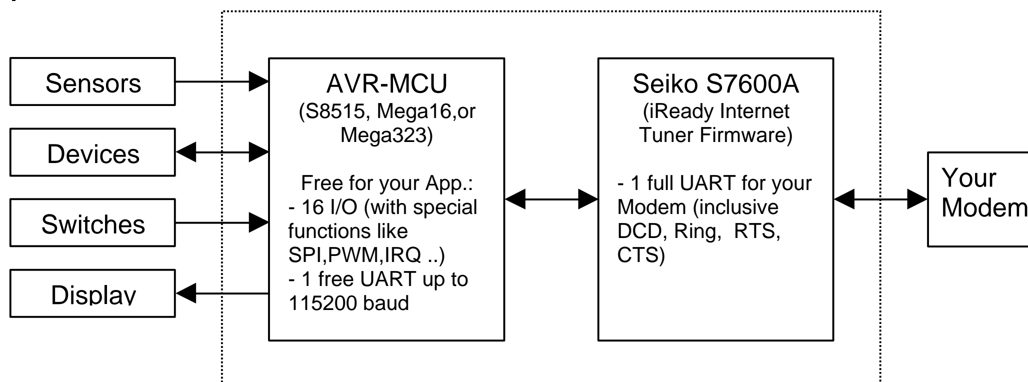
In the case of the client, these steps are followed:

1. Create a socket.
2. Specify the address and service port of the server program.
3. Establish the connection with the server.
4. Send and receive information.
5. Close the socket when finished, terminating the conversation.

Only steps two and three are different, depending on if it's a client or server application.

Functional overview

i-easy is equipped with two special low power consumption embedded controllers. One is an AVR microcontroller unit (MCU) from Atmel Corporation (depending on the module version: AT90S8515, ATmega16 or ATmega323) and the other one is the Seiko S-7600A controller (iChip) with specialized firmware for TCP/IP Protocol Stack, PPP, PAP.. (iReady Internet Tuner) .



The S7600A has a static firmware, which offers many functions to establish, open, close, send and receive data to or from the internet using a modem PPP dial-up connection.

The Atmel AVR is a highly integrated RISC based Flash Microcontroller. The flash has at least 1000 write cycles, so this MCU is ideal for fast developing and debugging your application at a short time to market and future-proof (firmware upgradable).

Programming the i-easy is the same task as programming a simple AVR controller, but you have many predefined special fuchtions. You can easily use the Seiko iChip with our library.

i-easy demo and i-easy lib v 0.4

In this short introduction we cannot give a detailed introduction to C coding / coding for AVR. We suggest reading an introduction to ANSI C or a beginners guide to C for AVR. Some of the information recources given on the last page may help.

The i-easy library is a set of special functions spread to different files, which you need to include to your project. In general one can find the low level instructions in `ichip.h` and higher level functions in `dial.h`, `socket.h` etc.

DEMO.C	Sample Application for i-easy board V 1.0 (e-mail and time server test)
dial.c	Dialup functions for dialing and PPP login (i-easy library)
dial.h	Dialup definitions for dialing and PPP login (i-easy library)
GLOBAL.H	Global define and typedefs
ICHIP.C	Low level access functions (i-easy library)
ICHIP.H	Low level access definitions (i-easy library)
SOCKET.C	Socket and TCP/IP connection functions (i-easy library)
SOCKET.H	Socket and TCP/IP connection definitions (i-easy library)
TIMER.C	Timer library by Volker Oth (modified)
TIMER.H	Timer library by Volker Oth (modified)
UART.C	UART library by Volker Oth (modified)
UART.H	UART library by Volker Oth (modified)

In general your program has to fulfill the following tasks:

Establishing a connection

1. Initialise the Protocol Stack (iChip)

```
InitSeiko();
```

2. Connect to the internet via modem

```
status=connectModem("ATX0","ATDT0123456789");
```

3. Authorise your modem connection (PPP login with username and password)

```
PPP_open("username", "secret");
```

Now you have an established internet connection and can exchange information using the two i-easy sockets (see next subchapter).
After data exchange close the connection:

4. Exit the PPP connection

```
PPP_close();
```

5. Hang up the modem

```
disconnectModem();
```

Using i-easy socket for client connection

1. Prepare the socket:

1.1 Define IP variables and load IP addresses

```
IP_PORT my_ip = {MyIPAddr[3],MyIPAddr[2],MyIPAddr[1],MyIPAddr[0],0,0} ;
IP_PORT smtp_server_ip = {1,2,3,4,SMTP_PORT_H,SMTP_PORT_L};
```

1.2 Take a socket (here 0) and give him the source and destination defined above.

```
Init_Socket(TCP_CLIENT_MODE, SOCKET_0, my_ip, smtp_server_ip);
```

2. Establish the TCP connection and open the remote server port

```
Status=Tcp_Connect(CLIENT, SOCKET_0);
```

If `Status` is true than the connection has been established.

3. Exchange data using `Tcp_Receive` and `Tcp_Send` comand, e.g.

```
numb=Tcp_Receive(tcpbuffer);                    // receive data
if (strstr(tcpbuffer,code250)!=tcpbuffer) {
      return 1;}                                // check server response
```

```
numb=Tcp_Send(emailcmd0);                         // send predef. command
sec_delay(1);                                     // wait 1sec for server
numb=Tcp_Receive(tcpbuffer);                      // receive response
```

Most servers send portopening-messages so you can check protocol versions or control messages. You can read them out with the `Tcp_Receive` command which returns the number of received bytes and uses the TCP buffer array for the received message (don't forget to define a sufficiently large u08 array for the receivebuffer).

You can send comand or data to the server port using `Tcp_Send`. Here you need a sendbuffer array and the function returns the number of successfully sent bytes.

Talking to a server needs special knowledge about the server command sets, which are defined in so called RFCs (request for comment), e.g. POP3 comand set is defined in *RFC1939*. You can find a list of all RFCs at `http://www.faqs.org/rfcs/` .

4. Close the TCP connection

```
Tcp_Close(SOCKET_0);
```

A closer look to the ieasy lib and the included functions will be taken in the `refman.pdf`.

Good luck.

Information resources

For the **AVR architecture** or the **iChip instruction set** have a look at

`Datasheets/` on the CD-ROM,

where you will find datasheets as well as the AVR instruction set, application notes, iChip software manual …,

or in the web at

`http://www.atmel.com/atmel/products/prod23.htm`
Atmel AVR 8-bit RISC home

`http://www.seiko-usa-ecd.com/intcir/products/rtc_assp/s7600a.html`
Seiko S-7600A home
`http://www.iready.org/`
iChip developers' sebsite

AVR Studio and **AVR-GCC** information can be found at

`avrgcc_studio.pdf` on the CD-ROM (How-to for AVR-GCC with AVR Studio)

`http://www.avrfreaks.net/avrgcc/` (home of AVR-GCC for Windows)

`http://combio.de/avr/` AVR-GCC Linux(/Windows) Archives

`http://www.amelek.gda.pl/avr/libc/` Lib-C

RFCs
`http://www.faqs.org/rfcs/`

Have a look at

`Docs/` on the CD-ROM

to get the referenced `*.pdf` files.