INTRODUCTION

Version, Trademarks, and Copyrights

Version

This printed document is generated from the online help document included with the product. This version of the document describes 6.16 of the product. Since we continuously update our product, sometimes the printed document becomes out of phase with the shipping product. When in doubt, please refer to the online document for the most up to date information. This document was last updated on May 1st, 2000.

Trademarks and Copyrights

IMPORTANT: Registering the Software

PLEASE READ THIS BEFORE INSTALLATION!

The software **must** be unlocked within 30 days of installation or it will stop operating after the 30 days has ended. Downloading the software and attempting to reinstall it for another 30 day trial period will fail. When you register, we will provide you with an unlock code. The primary reason we copy-protected the software in this fashion is so that you can download the latest version of our software at anytime from our website. Once you register, the updates do not needed to be registered again, providing you install the update over the existing version.

You must unlock this software whether you purchase the product from us or from one of our distributors or through the website. Otherwise, the software will behave as if it is a demo version that expires in 30 days. Once you register, you should save and restore the licensing information if you ever want to move the installation to a different machine or different directory. This also means that updates should be installed in the same directory as the previous version. Otherwise, the new copy will behave once again as only a demo version.

If some accident occurs and you need to reinstall the product and have lost the licensing number, contact us and we will give you a new copy. We feel that the ability to obtain easy updates from our website outweighs the minor annoyances that the registration process causes.

Note that the licensing code is written to a floppy disk in a special format and the floppy disk cannot be copied. For updating to the latest versions, if you install the later version in the same directory as the older version, you will not have to do anything special regarding the licensing.

Unlocking with a Key (Floppy) Disk

If you receive a key disk with your package, then you can use the licensing code in the disk to unlock your software. Follow the directions on "Importing a License from a Floppy Disk" below.

Register without a Key Disk

In the Help menu, select "Register." A dialog box with a string of hexadecimal digits appear. Email, FAX, or call us with the string. We will then reply with an unlock code. In the unlikely event that you wish to uninstall and return the product to us, you will need to select the same dialog box and obtain an unlock code from us to render the software useless.

Transferring Your License to a Floppy Disk

You can transfer your license to a floppy disk temporarily if you want to reinstall the product on a different directory or different machine. In the Help menu, select "Transfer License to Floppy Disk." Follow the program directions to proceed. Once the license is transferred to a floppy disk, then the old copy becomes unlicensed so only do this if you are moving the license to a different copy.

Importing a License from a Floppy Disk

After transferring your license to a floppy disk (or if you receive a key disk), you may register a new installation by clicking on the "Import License from Floppy" button in the register dialog box. You can also do this by selecting "Import License from Floppy Disk" in the Help menu.

Using the Product on Multiple Computers

If you need to use the product on multiple computers, e.g. on an office PC and a laptop, and if you are the only user of the product, then you may obtain a separate license from us. Contact us for details. Alternatively, you may transfer the license back and forth using the floppy disk as described above.

Software License Agreement

This is a legal agreement between you, the end user, and ImageCraft. If you do not agree to the terms of this Agreement, please promptly return the package for a full refund.

GRANT OF LICENSE. This ImageCraft Software License Agreement permits you to use one copy of the ImageCraft software product ("SOFTWARE") on any computer provided that only one copy is used at a time.

COPYRIGHT. The SOFTWARE is owned by ImageCraft and is protected by United States copyright laws and international treaty provisions. You must treat the SOFTWARE like any other copyrighted material (e.g., a book). You may not copy written materials accompanying the SOFTWARE.

OTHER RESTRICTIONS. You may not rent or lease the SOFTWARE, but you may transfer your rights under this License on a permanent basis provided that you transfer this License, the SOFTWARE and all accompanying written materials, you retain no copies, and the recipient agrees to the terms of this License. If the SOFTWARE is an update, any transfer must include the update and all prior versions.

LIMITED WARRANTY

LIMITED WARRANTY. ImageCraft warrants that the SOFTWARE will perform substantially in accordance with the accompanying written materials and will be free from defects in materials and workmanship under normal use and service for a period of ninety (90) days from the date of receipt. Any implied warranties on the SOFTWARE are limited to 90 days. Some states do not allow limitations on the duration of an implied warranty, so the above limitations may not apply to you. This limited warranty gives you specific legal rights. You may have others, which vary from state to state.

CUSTOMER REMEDIES. ImageCraft's entire liability and your exclusive remedy shall be, at ImageCraft's option, (a) return of the price paid or (b) repair or replacement of the SOFTWARE that does not meet ImageCraft's Limited Warranty and that is returned to ImageCraft. This Limited Warranty is void if failure of the SOFTWARE has resulted from accident, abuse, or misapplication. Any replacement SOFTWARE will be warranted for the remainder of the original warrant period or 30 days, whichever is longer.

NO OTHER WARRANTIES. ImageCraft disclaims all other warranties, either express or implied, including but not limited to implied warranties of merchantability and fitness for a particular purpose, with respect to the SOFTWARE, the accompanying written materials, and any accompanying hardware.

NO LIABILITY FOR CONSEQUENTIAL DAMAGES. In no event shall ImageCraft or its supplier be liable for any damages whatsoever (including, without limitation, damages for loss of business profits, business interruption, loss of business information, or other pecuniary loss) arising out of the use of or inability to use the SOFTWARE, even if ImageCraft has been advised of the possibility of such damages. The SOFTWARE is not designed, intended, or authorized for use in applications in which the failure of the SOFTWARE could create a situation where personal injury or death may occur. Should you use the SOFTWARE for any such unintended or unauthorized application, you shall indemnify and hold ImageCraft and its suppliers harmless against all claims, even if such claim alleges that ImageCraft was negligent regarding the design or implementation of the SOFTWARE.

About the ImageCraft Development Environment

The ImageCraft C Development Environment is a program for developing microcontroller applications using the ANSI standard C language. Its main features are:

- An intuitive Windows 95/NT native Integrated Development Environment (IDE) with integrated editor and project manager. Source files are organized into projects. Editing and building can be done wholly within the environment. Compile time errors are displayed in the status window, and with a simple click of the mouse button, you can jump to the lines that cause the errors in the editor window. The integrated project manager generates a standard makefile that you can view and use directly if desired.
- The IDE drives an ANSI C command line compiler that is normally transparent in operation. However, if you wish, you can interact with the compiler directly using the command prompt program. The compiler is a set of native 32-bit programs and understands long file names.

With some exceptions, this document does not describe C in detail nor does it contain C tutorials in general. Since the compiler implements the standard ANSI C language, there are many excellent books on C available in your local bookstores or from online booksellers such as Amazon (although we highly recommend you support your local independent bookstores if possible).

You can find a list of books that we recommend on our website. There are many more fine books available, so browse around.

Support

Before contacting us, find out the version number of the software by selecting "About ICCAVR" in the Help menu. If you believe you have found a bug, please create the smallest **complete** example program that exhibits the behavior so that we can duplicate the problem.

Internet and email are the preferred methods of support. Program updates are available free of charge for the first six months. Files are available from our website:

```
http://www.imagecraft.com
```

Email support questions to

```
support@imagecraft.com
```

We have a mailing list called icc-avr pertinent to our AVR product users. To subscribe, email

```
minimalist@imagecraft.com
```

with the subject

```
subscribe icc-avr
```

The mailing list should not be used for general support questions. Those are best handled by the `support@imagecraft.com` account.

Our postal address and telephone numbers are

```
ImageCraft
706 Colorado Ave.
Suite 10-88
Palo Alto, CA 94303
U.S.A.

(650) 493-9326
(650) 493-9329 (FAX)
```

If you purchased the product from one of our international distributors, you may wish to query them for support first.

Product Updates

The product version number consists of a major number and a minor number. For example, V6.10 consists of the major number of 6 and the minor number of .10. Within the initial six months of purchase, you may update to the latest minor version free of charge. To receive updates afterward, you may purchase the low cost annual maintenance plan. Upgrade to a new major version may require additional cost.

With the software protection scheme used in the product, you get the upgrades by downloading the latest "demo" available on the website and installing it in the same directory as your current installation. Your existing license will work on the newly installed files.

Note that if you wish to install the new version on a different directory, or to a different machine, then you must transfer your license by using the floppy disk, as explained in Registering the Software.

Frequently Asked Questions

- Where do I start? I just want to write a simple program!

  First, you have to have a basic understanding of C. You use Project to organize your files, and you Build them to an output that you could use to program your device. You can also compile a single file to an output, but a Project is the preferred method of organizing your files.

- I try to compile one of the example programs, but something about the "paths" is incorrect.

  The project files contain the paths to the include files and the libraries. You need to modify them to match the locations where you install the tools. You can do this by selecting Project->Options.

- Does your compiler support ATMega? Tiny? or ??? devices?

  Currently, we support all devices that have data SRAM. We (will) have a campanion product ICCtiny for programming the Tiny devices and the ATS901200. We constantly update our device support list so the IDE will support newer devices as they become available. In addition, you can enter device characteristics by hand for any unsupported devices.

- I don't know C, what should I do?

  You should definitely get a good tutorial book. You can also use a PC ANSI C compiler (e.g. Borland/Inprise or Microsoft) to test out your algorithms.

- How do I use the ??? susbsystems of the AVR?

  You need the relevant AVR data book from Atmel. We also provide a set of simple interface library functions to the subsystems.

- I use printf, and nothing comes out.

  You need to initialize the UART. For example,

  ```
  UART_TRANSMIT_ON();
  UBRR = BAUD9600;
  ```

  Initializes the UART to 9600 baud and turns the transmitter on. These macros are defined in the file avr.h (the baudrate is for a 4MHz clock).

- What do I need to do to get interrupts working?

  See Interrupt Handlers.

- How do I use the Terminal IO window in AVR Studio? How come (XXXX) features do not work under AVR Studio?

  See COFF and Working With AVR Studio.

- How do I build my own libraries?

  See Librarian.

- Can I write in assembly code?

  Yes, both inline assembly and assembler modules (files).

- What library functions do you support?

  A subset of the Standard C library, and some AVR specific routines.

- Is the library source code available?

  Yes. See C Library General.

- Can I keep non-source files (e.g. documentation) in the Project Manager Window?

  Yes.

- Can you support ISP of the STK-200 or STK-300 boards?

Kanda Systems has advised us that we should not include ISP support within our IDE since the algorithms are still being revised. Please use the Kanda software instead.

File Types and File Extensions

Files types are determined by their extensions. The IDE and the compiler base their actions on the file types of the input.

- .c - specifies a C source file.
- .s - specifies an assembly source file.
- .h - specifies a header file.
- .prj - a project file. This is created and maintained by the IDE to store information about a project.
- .a - library file. The package comes with several libraries. libcavr.a is the basic library containing the Standard C library and AVR specific routines. The linker only links in modules (or files) from a library if the module is referenced. You may create or modify libraries as needed.

Output Files

- .s - for each C source file, an assembly output is generated by the compiler.
- .o - an object file, produced by assembling an assembly file. An output executable file is the result of linking multiple object files.
- .hex - an Intel HEX output file.
- .cof - a COFF format output file.
- .lst - a listing file. The object code and final addresses for your program files are gathered into a single listing file.
- .mp - a map file. It contains the symbol and size information of your program in a concise form.

The IDE also creates other files in the project output directory.

Pragmas
#pragma

The compiler accepts the following pragmas:

- #pragma interrupt_handler <func1>:<vector number> <func2>:<vector> ...

  Declare functions as interrupt handlers so that the compiler generates a `reti` instead of `ret` instruction, and save and restore all the registers that the functions use. Also generates the interrupt vectors based on the vector numbers. See `Interrupt` Handlers. This pragma must precede the function definitions.

The following pragmas do not have much use for the AVR targets.

- #pragma text:<name>

  Change the name of a text area. Corresponds to the -text:<text> command line option.

- #pragma data:<data>

  Change the name of a data area. Corresponds to the -data:<data> command line option.

- #pragma abs_address:<address>

  Do not use relocatable areas for the functions and global data but allocate them from the absolute address starting at <address>. This is useful for accessing interrupt vectors and other hard-wired items. This does not apply to uninitialized global data.

- #pragma end_abs_address

  Use the normal relocatable areas for objects.

Converting from Other Compilers

Converting between IAR or other ANSI C Compilers

As the first C compiler available for the AVR, there is quite a bit of source code written for the IAR C compiler. This page examines some of the issues you are likely to see when you are converting source code written for the IAR compiler to the ImageCraft compiler. Fortunately, most of your code should compile without any problem since both compilers are ANSI C compilers. Even the IO registers references are the same.

- Interrupt handler declaration. Our product uses a pragma to declare a function as an interrupt handler whereas IAR introduces syntactic extension. The mapping is one to one:

```
ICCAVR:
#pragma interrupt_handler func:4    // 4 is the vector number

IAR:
interrupt [vector_name] func()
```

- Extended keyword. IAR introduces the `flash` keyword to allocate an item in the program memory. ICCAVR uses the `const` keyword for similar purposes.
- Calling convention. The registers used to pass arguments to functions are different between the compilers. This only affect hand-written assembly functions.
- Inline assembly, macros etc. IAR does not support symbolic forms of inline assembly, whereas ICCAVR does.

Converting Between ICCtiny and ICCAVR

ICCtiny is our development product for the tiny AVR devices. A list of the compatabilities between ICCAVR and ICCtiny can be found in the ICCtiny help file and manual.

Acknowledgments

The front end of the compiler is lcc: "lcc source code (C) 1995, by David R. Hanson and AT&T. Reproduced by permission." The assembler/linker is a distant descendent of Alan Baldwin's public domain assembler/linker package. Some of the 16-bit arithmetic multiply/divide/modulo routines were written by Atmel. The 32-bit floating point and long arithmetic routines were written by Jack Tidwell. Check out http://www3.igalaxy.net/~jackt for Jack's AVR BASIC compiler, utilities and AVR circuits. The Make utility is by Jacob Navia. Check out Jacob's low cost Win32 compiler http://www.cs.virginia.edu/~lcc-win32.

The PROFESSIONAL version includes the GNU RCS utilities and the grep program. The GNU copyleft license specifies that you may redistribute the GNU programs. This does not apply to any other software in this package that is not GNU based. ImageCraft has not modified the GNU programs.

The PROFESSIONAL version also includes the Application Builder from Kanda Inc. (http://www.kanda.com) and Jack Tidwell's AvrCalc program.

All code used with permission. Please report **ALL bugs to us directly**.

TUTORIALS

Getting Started

Once you invoke the IDE, choose Project->Open from the menu system. Navigate to the \icc\examples directory and select the project "led." The project manager displays the filename led.c indicating there is one file in this project. Select the project compilation option by choosing Project->Options. Under the "Target" tab, select the target processor. **Make changes to the "Paths" tab to match the location where you install the compiler**.

Now select Project->Make. The IDE invokes the compiler to compile the project files and display any messages in the Status Window.

Assuming there is no error, an output file called led.hex is produced in the same directory as your source file; in this case, \icc\examples. This is a file in Intel HEX format. Most AVR programmers and simulators understand this format, and you can load this program into your target. That's all there is to build a program!

If you want to test your program on a tool that accepts COFF debug information, for example the AVR Studio, then you need to select COFF as the output file format under Project->Options. Notice that the often used functions are also available in the button bar and as context sensitive right mouse pop-up menus. For example, you can choose the compiler options by right clicking on the Project Window.

Double clicking a file name in the Project Window opens it in the Editor. Go ahead and open led.c this way. For experimentation, try to introduce errors, for example, delete a semicolon ";" from a line. Now select Project->Build. The IDE will ask you if you want to save the changes first. Select yes and the compilation will commence. This time there should be an error displayed in the Status Window. Clicking on the error line, or clicking on the error symbol on the left on the error line will move the cursor to the offending line in the Editor.

Anatomy of a C Program

A C program must define a function called `main`. The compiler links your program with the startup code and the library functions into an "executable" file, so called because you can execute it on your target. The purpose of the startup code is described in detail in Startup File. In summary, a C program needs the target environment to be set up in certain ways and the startup code initializes the target to satisfy these requirements.

In general, your `main` routine performs some initialization stuff and then executes an infinite loop. As an example, let's examine the file `led.c` in the `\icc\examples` directory:

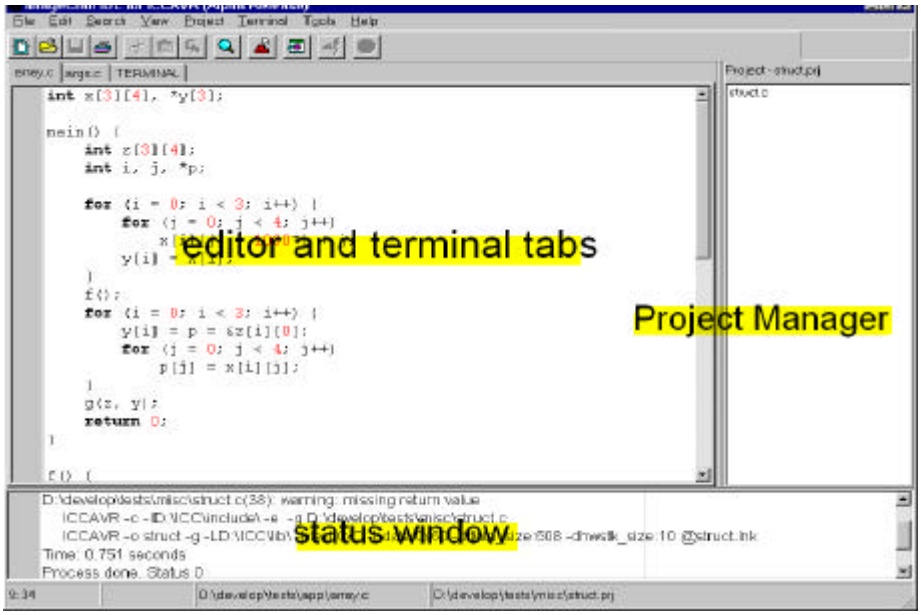
```
#include <avr.h>
/* This seems to produce the right amount of delay for the LED to be
 * seen
 */
void Delay()
   {
   unsigned char a, b;

   for (a = 1; a; a++)
         for (b = 1; b; b++)
               ;
   }

void LED_On(int i)
   {
   PORTC = ~BIT(i);  /* low output to turn LED on */
   Delay();
   }

void main()
   {
   int i;
   DDRC = 0xFF;      /* output */
   PORTC = 0xFF;     /* all off */

   while (1)
         {
         /* forward march */
         for (i = 0; i < 8; i++)
               LED_On(i);
         /* backward march */
         for (i = 8; i > 0; i--)
               LED_On(i);
         /* skip */
         for (i = 0; i < 8; i += 2)
               LED_On(i);
         for (i = 7; i > 0; i -= 2)
               LED_On(i);
         }
   }
```

The main routine is very straightforward. After initializing some IO registers, it executes an infinite loop and changes the LEDs in walking patterns. The LEDs are changed in the routine LED_On, which simply writes the correct values to the IO port. Since the CPU runs very fast, LED_On calls a delay loop so that the patterns can be seen. Since the actual amount of delay is not critical, a pair of nested loops seems to give the right amount of delay. If the actual timing is important, then the routine should use the timer register to count time.

The other example, `8515intr.c`, is very similar but also shows how simple it is to write an interrupt handler in C. While small, these two programs can serve as a starting point for your programs.

IDE Overview

The IDE is divided into three window panes:



The top left window pane is the editor and incudes the terminal tabs. The editor is capable of syntax highlighting C elements, plus bookmarks and other features. When opened, the built in terminal emulator also displays as one of the tabbed windows in the editor pane. The top right pane is the Project Manager window. This pane contains the list of C and assembly files in a project. The bottom pane is the Status Window. Any compilation status is displayed on this pane. In addition, the bottom status bar displays useful information such as the full file name of the currently active editor, the cursor position, and the full file name of the project file.

The panes are resizable and you may hide the status window or the project window from view to maximize the editor display. This is done by toggling the appropriate menu item in the View menu.

User operations are input through the menus. Frequently used operations are also available on the button bar and as context sensitive right mouse button pop-up menus. The IDE is highly configurable. Browse the choices available from the Option menus.

Using the Project Manager

Once you create your program file(s), either with the IDE's built-in editor, or some other tool, you can add the files to the Project Manager. The Project Manager keeps track of all the files in your project, including non-source code files such as project documentation. Only the source code files are important to the Project Manager. When you select a Build command, the Project Manager deduces the header file dependencies and invokes the compiler to rebuild only the files that have been changed. Using a project manager greatly simplifies your programming task.

Sometimes for quick and dirty prototyping, you may want to forgo the steps of setting up a project. The IDE allows you to compile a single file into an output file. See Compiling a Single File.

IDE

Compiling a Single File

Normally in order to create an output file, you should create a project and define all the source files belonging to that project. Nevertheless, sometimes it is convenient to compile a single file to either an object file or into a final output. You can use the IDE File->Compile File... command to perform either of these tasks. When you invoke this command, the file in the current active editor is compiled.

Compiling a single file into an object file is useful for checking syntax errors, or if you are compiling a new startup file. Compiling a single file into an output file is useful if your program is small and can be kept as a single file. Note that the default compiler options are used.

Creating a New Project

To create a new project, use Project->New. This brings up a dialog box that allows you to specify the name of the project, which is also used as the name of your output file. Since you can't save an empty project, you must add some source files to the project file list. If you have already created some source files, you can add them using the Project->AddFile(s) command. Otherwise, you can create source files by invoking File->New, typing in your code and then invoking File->Save or File->SaveAs. You can then add the newly created file by invoking Project->AddFile(s). You can also add a file that is currently being edited to the project file list by right clicking on the editor window and invoking "Add to Project." Usually you put the source files in the same directory as the project file, but that is not a requirement.

Be sure to save the project by invoking Project->Save or Project->SaveAs. Compiler options are specified using Project->Options.

Project Manager

The Project Manager allows you to group a list of files into a project and defines compilation options for them. This allows you to break down your program into small modules. When you perform a project Build function, only source files that have been changed are recompiled. Header file dependencies are automatically generated. That is, if a source file includes a header file, then the source file will be automatically recompiled if the header file changes. The Project Manager creates a makefile in standard format. You may examine the generated makefile if you wish.

A source file can be written in either C or in assembly. C files must have the .c extension and assembly files must have the .s extension. You may keep any files in the project list. For example, you may keep project documentation files in the Project Manager window. The Project Manager ignores non-source files when performing a Build.

Compiler options are kept with the project files so you can have different projects with different targets. When you start a new project, a default set of options is used. You may set the current options as the default or load the default options into the current option set. The default options are kept in the file `default.prj` in the executable directory where the compiler is located.

To ease cluttering up your project directory, you may specify that the output file and the intermediate files that the tool generates reside in a separate directory. Usually this is a subdirectory under your project directory. See Compiler Options: Paths.

Editor Windows

The Editor Windows section is the main area of interaction between you and the IDE. It contains a list of open files in a tabbed notebook control. If you invoke the IDE built-in terminal emulator, it is opened in this section as well.

The editor is highly configurable. See Environment Options and Editor Options. If you choose not to use the IDE's built-in editor, you can use your editor of choice. The compiler's output is fairly simple and should be parseable by most advanced editors. Contact us or the editor vendors for details. To enable concurrent viewing and editing of the same file in the IDE's editor and an external editor, you can direct the IDE to detect if any opened files have been changed on the disk (e.g. by an external editor) and to reload the file if changed.

To the left of the editor is the gutter where informational glyphs are displayed. These include line numbers and bookmarks. Up to 10 bookmarks can be set in each editor window.

Status Window

The Status Window displays the status information from the IDE, such as from a Project Build. Compiler error messages start with "!E.." and are tagged with a small red sign on the gutter. Clicking on an error status line brings the cursor to the offending line in the editor.

The contents of the Status Window can be selected and copied into the Windows Clipboard. When you are performing a Build function, the last line indicates the status of the build. If there is any error, you may scroll backward to find the source of the error(s).

Terminal Emulator

The IDE contains a built in terminal emulator. Note that it does not contain any ISP (in System Programming) functions but is a simple terminal, perhaps for your target device to display debugging messages. An ASCII file downloader is also provided.

REFERENCES

Pop Up Menus

Context sensitive popup menus are available in most places by right-clicking on the mouse. Usually they are a subset of the most popular commands for that window.

File Menu

This menu contains file and program related operations. The active editor is the editor tab with the focus. If you select to open a file that is already opened, its editor tab will be made active. The status bar at the bottom gives information about the current active editor tab, including cursor location, whether the file is READONLY or modified, and the full path name of the file.

- **New** - Creates an empty editor tab where you can enter text.
- **Reopen...** - Contains a list of recently opened files. Select file to reopen it.
- **Open...** - Opens a file for editing.
- **Reload... from Disk** - Abandons all changes and reloads the active file from the disk.
- **Reload... from Back Up** - Reloads the active file from the last backed up file. See "Save."
- **Save** - Saves the active file to the disk. It optionally creates a backup file, of the form <file>.~<ext> before the new contents are saved. See Environment Options.
- **Save As...** - Saves the active file to the disk with a new name.
- **Close** - Closes the active file. Prompts if the file contains unsaved changes.
- **Compile File... to Object** - Compiles the active file to an object file with .o extension. Note that an object file is not directly loadable into a device programmer or simulator such as AVR Studio. See File Types. This is useful to check for any compilation errors with the file, or to create an object file for a library, or a new startup file.
- **Compile File... to Output** - Compiles the active file to an output, suitable to be loaded into a device programmer or AVR Studio. Normally you would use the Project Manager to manage a list of the files for your project, but if your project is small, you can simply use this command to create an output. The compiler uses the default Compiler Options.
- **Save All** - Saves all the currently open files.
- **Close All** - Closes all the currently open files and the Terminal Tab if open. Prompts for unsaved changes.
- **Print** - Prints the active file. See Print Options for options.
- **Exit** - Quits the program. Prompts for unsaved changes.

Edit Menu

This menu contains editing operations for the editor.

- **Undo** - Undoes last editing changes.
- **Redo** - Undoes the last "undo." That is, reapplies the changes you have undone.
- **Cut** - Cuts the selected text into the Windows Clipboard.
- **Copy** - Copies the selected text into the Windows Clipboard. Note that this works for the Status Window content.
- **Paste** - Pastes the Window Clipboard content into the cursor point.
- **Delete** - Deletes the selected text.
- **Select All** - Selects the entire contents of the active editor.
- **Block Indent** - Indents (i.e. shift right) the selected text by the amount specified in the *Environment Options -> Editor*.
- **Block Outdent** - Outdents (i.e. shift left) the selected text by the amount specified in the *Environment Options -> Editor*.

Search Menu

This menu contains the searching functions of the editor.

- **Find...** - Finds text in the editor. The search options are:
  - **Match Case** - If checked, match the case exactly
  - **Whole Word** - If checked, find match only if the search string is surrounded by white spaces or punctuation.
  - **Direction Up/Down** - Selects whether to perform the search upward or downward from the cursor.
- **Find in Files...** - Available only in the PROFESSIONAL version. Finds text in all the open files, or all files in the project, or all the files matching the specified filemask (default is *.* or all files). The result of the search is displayed in the Status Window. The search options are:
  - **Case Sensitive** - If checked, match the case exactly
  - **Whole Word** - If checked, find match only if the search string is surrounded by white spaces or punctuation.
  - **Regular Expression** - If checked, then allow `grep` regular expressions. Some of the more commonly used expressions are:
    - . (dot) - any character
    - ^ - the beginning of a line
    - $ - the end of a line
    - [0-9] - any digit
    - [a-z] - any lower case letter
    - (<expr1>|<expr2>) - match either "expr1" or "expr2"
    - ? - match 0 or 1 occurrence of the previous expression
    - * - match 0 or more occurrences of the previous expression
- **Replace...** - Replaces text in the editor.
- **Find Again** - Performs another search using the last search string.
- **Jump To Matching Brace** - if the cursor is on (i.e. in front of) a brace character, the cursor is moved forward or backward to go the matching brace character. For example, '(' is the matching brace character for ')'. The brace characters are:

  `(, ), [, ], {, }, <, and >`
- **Goto Line Number** - Prompts for a line number to jump to. Note that you can also have the editor display line numbers in the gutter.
- **Add Bookmark** - Adds a bookmark to the line. Note that it is often quicker to simply click on the gutter of a line to add or delete a bookmark.
- **Delete Bookmark** - Deletes a bookmark on a line.
- **Next Bookmark** - Searches forward until a line with a bookmark is encountered.
- **Goto Bookmark** - Jumps to a specific bookmark.

View Menu

- **Project File List** - If checked, the Project File List Window (the right pane) is displayed. Uncheck it to maximize the editor window size.
- **Status Window** - If checked, the Status Window (the bottom pane) is displayed. Uncheck it to maximize the top pane display.
- **Project Makefile** - Opens the makefile in READONLY mode. When you Build a project, the Project Builder creates a makefile that describes the dependencies of the project files. Dependencies between the header files (.h files) are determined automatically by the Project Builder.
- **Output Listing File** - Opens the listing file (.lst) in READONLY mode. The listing file contains final code addresses for all your program code, excluding the library routines. See Listing File.

Project Menu

The menu contains the interface to the Project Builder. Only one project may be open at a time. If there is an open project with unsaved changes and you try to create a new project or open another project, you will be prompted to save the changes.

- **New...** - Creates a new project file. Prompts for a directory and file name to store the project file. Usually you would keep your project file in the same directory as your source files, although it is not required. Long file names are supported except that paths with embedded spaces are not allowed. The name you give to the project will be used as the name of the program output. For example, if your project is named "foo.prj," then the output is called foo.hex, or foo.cof, depending on the output file format.
- **Open** - Opens an existing project file.
- **Open All Files...** - Opens all the project source files.
- **Close All Files** - Closes all project files that are open.
- **Reopen...** - Contains a list of recently opened projects; select to reopen one.
- **Make Project** - Determines the project file dependencies and compiles changed files to output.
- **Rebuild All** - Recompiles all project files. Useful if somehow things get out of sync, but if you find that your files are not being recompiled even if they are changed, there may be other causes such as incorrect system dates.
- **Add File(s)** - Opens a dialog box and adds files to the project. You may add any files to your project but source files must either be C files (with .c extension) or assembly files (with .s extension). Non-source files are kept on the project file list but are otherwise ignored by the Project Builder.
- **Remove Selected Files** - Removes selected files in the project file list window from the project.
- **Option...** - Opens the Compiler Options dialog box. See Compiler Options.
- **Close** - Closes the project. Prompts to save changes if needed.
- **Save** - Saves the project, including list of project files and compiler options.
- **Save As...** - Saves the project to a different name.

RCS Menu

Please see Configuration Management with RCS for a general overview of RCS.

IDE RCS functions

- **Checkin Selected File(s)** - checks in all the selected files in the project list window. A dialog box is displayed for you to enter the check in message (or in the case of initial check in, the file description and an optional label). The files are checked out immediately afterward. This uses the "ci" command with the -l option to check out the files.
- **Checkin Project** - checks in all the files in the project. You will get an error message from "ci" if a file has not been changed, you may ignore those errors. The files are checked out immediately afterward.
- **Diff Selected File** - displays the differences between different revisions of a file. The default is to compare the last revision with the version that you are currently working on. You can also compare any two versions of the file. The output is displayed in the Status Window. This uses the "rcsdiff" command.
- **Show Log of the Selected File(s)** - shows the log entries of the selected files. This is useful for finding out the different revision numbers and labels of the files. This uses the "rlog" command.

Tools Menu

- **Environment Options** - Opens up the Environment Options and the Terminal Options dialog box.
- **Editor and Print Options** - Opens up the Editor and Print Options dialog box.
- **App Builder** - Available only in the PROFESSIONAL version. Invokes the Application Builder, which is a program that creates AVR initialization routines based on the selections you make in a series of dialog boxes. Currently only the MEGA device is supported.
- **AVR Calc** - Available only in the PROFESSIONAL version. Invokes the AVRCalc program, which is a handy program that displays 32-bits floating point as hexidecimal, and calculates the UART constants and timer registers for a given baudrate or crystal frequency.
- **Configure Tools** - Allows you to add tools to the Tools Menu.

  When invoked, you can use the dialog box to change the Tools Menu content. The fields of this dialog box are:

  - **Menu Name** - Name to use in the Tools menu.
  - **Program** - Full path to the executable. You may use the Browse button to select an executable.
  - **Parameters** - Parameters to the program. If you specify %f in the content, it will be replaced by the filename of the topmost file being edited.
  - **Initial Directory** - The directory the IDE switches to before running the program.
  - **Capture Output** - If checked, the IDE captures the output (standard output and standard error) of the program and displays it on the Status Window. This should only be checked for Win32 Console Mode or DOS programs.
- **Run** - Simple interface to run a program. Similar to Windows "Run" function on the Start menu.

Any tools that you configure will appear after the "Run" command.

Terminal Menu

This menu interacts with the terminal emulator.

- **View** - Toggles whether to display the terminal emulator.
- **Clear Window** - Clears the terminal window.
- **ASCII Download... Last File** - Downloads the last file.
- **ASCII Download... Browse** - Prompts for a file to download.
- **Capture...** - Toggles capturing of the terminal output. Prompts for file name when turned on.

Compiler Options

There are three tabs in the option dialog box: Paths, Compiler, and Target. There are two buttons in addition to the standard OK, Cancel, and Help:

- Set As Default - Writes the options to the default option file \icc\bin\default.prj. When you start the IDE or create a new project, it is loaded with the default options.
- Load Default - Loads the default options into the current setting.

Compiler Options: Paths

- **Include Path(s)** - You may specify the directories where the compiler should search for include files. You may specify multiple directories by separating the paths with semicolons or spaces. If a path contains a space, then you should enclose it within double quotes.
- **Library Path** - The compiler automatically links in the startup file (see Startup File), and the standard library. This edit box specifies the location of these files. Only a single directory may be specified. If a path contains a space, then you should enclose it within double quotes.
- **Output Directory** - Typically the source files are kept in the project directory, along with the project files. Compilation creates a number of files and to avoid cluttering up the project directory, you may want to put all the output files in their own directory. Typically this is a subdirectory under the project directory.

Compiler Options: Compiler

- **Strict ANSI C Checking** - The ANSI C standard still allows certain operations that may be unsafe. If this check box is selected, the compiler warns about declarations and casts of function types without prototypes, assignments between pointers to integers and pointers to enums, and conversions from pointers to smaller integral types. It also warns about unrecognized control lines, non-ANSI language extensions and source characters in literals, unreferenced variables and static functions, declaring arrays of incomplete types, and exceeding some ANSI environmental limits, such as more than 257 cases in switch statements.
- **Accept C++ Comments** - If selected, the compiler treats everything up to the newline after the character pair // as comments.
- **Macro Define(s)** - You define macros separated by spaces or commas. Each macro definition is in the form

  ```
  name[:value] or name[=value]
  ```

  For example:

  ```
  DEBUG:1;PRINT=printf
  ```

  defines two macros, DEBUG and PRINT. DEBUG has the value 1 by default and PRINT is defined as printf. This is equivalent to writing

  ```
  #define DEBUG 1
  #define PRINT printf
  ```

  in the source code. A common usage is to use conditional preprocessor directives to include or exclude certain code fragments.
- **Macro Undefine(s)** - same syntax as Macro Define(s) but has the opposite meaning.
- **Output File Format** - You select COFF/HEX, Intel HEX or COFF. Intel HEX is a simple format that is basically a memory image of the program. COFF contains debugging information that can be used by tools that accept that format for source level debugging. For example, the AVR Studio understands COFF output format. COFF/HEX specifies that both types of files will be generated.
- **Strings in FLASH** - By default, literal strings are allocated in both FLASH ROM and SRAM to allow easy mixing of strings and char pointers. (See Strings.) If you want to eliminate the overhead of allocating strings in SRAM, you can check this option. You must be careful to use the correct library functions. For example, to copy a literal string allocated this way into a buffer, you must use the cstrcpy() function instead of the standard strcpy() function. (See String Functions.)
- **Optimizations** - You control the levels and types of optimizations. Currently, the choices are
  - **Default** - basic optimizations such as register allocation, basic block common subexpression elimination etc.
  - **Maximize Code Size Reduction** - enabled for the PROFESSIONAL version only. This invokes the Code Compressor (tm) optimizer to eliminate duplicate object code fragments. While the operation of the Code Compressor is generally transparent, you should read the description in the Code Compressor page to familiarize yourself with its operations and limitations.

  The STANDARD and demo versions of the product are allowed a number of free trial runs of the optimizer. Once the number of trial runs is over, the optimizer will no longer be operative in these versions.

  As we improve the product, we will add more optimization levels to both the STANDARD and PROFESSIONAL versions of the tools, and the compiler option settings will be augmented to provide the users with flexible control.

Compiler Options: Target

- **Device Configuration** - Select the target device. If your device is not on the list, as long as it has data SRAM, the compiler will work with it. If your target device is not on the list, select "**Custom**" and enter the relevant parameters described below.
- **Memory Sizes** - Specify the amount of program and data memory in the device. Changeable only if you select "Custom" in the device selector list. The data memory refers to the internal SRAM. Note that the compiler cannot use more than 64K of program memory so there is only one entry for both Mega603 and Mega103.
- **Text Address** - Normally text (the code) starts right after the interrupt vector entries. For example, code starts at 0xD (word address) for 8515 and 0x30 for the Mega devices. However, if you do not use all of the interrupts, you may start your code earlier - as long as it does not conflict with the vector table. Changeable only if you select "Custom" in the device selector list.
- **Data Address** - Specify the start of the data memory. Normally this is 0x60, right after the CPU and IO registers. Changeable only if you select "Custom" in the device selector list. Ignore if you choose external SRAM.
- **Use Long JMP/CALL** - Specify that the device supports long jmp and long call instructions.
- **Enhanced Core** - Specify that the device supports the enhanced core instructions such as hardware multiply, lpm z+, movw etc.
- **IO Registers Offset Internal SRAM** - Specify whether or not the IO registers offset the start of the internal SRAM. For example, 8515's SRAM starts at 0x60, after the IO registers space and extends for 512 bytes. For the Mega603 device, the IO registers overlays the SRAM space and therefore SRAM starts at 0 (but not usable until after 0x60) and extends for 4096 bytes. Changeable only if you select "Custom" in the device selector list.
- **Internal vs. External SRAM** - Specify the type of data SRAM on your target system. If you select external SRAM, the correct MCUCR bits will be set.
- **PRINTF Version** - This radio group allows you to choose which version of the printf your program is linked with. More features obviously use up more code space:
  - Small or Basic: only `%c, %d, %x, %X, %u`, and `%s` format specifier without modifiers are accepted.
  - Long: the long modifier: `%ld, %lu, %lx`, %lX are supported, in addition to the width and precision fields.
  - Floating point: `%f` for floating point are supported. Note that for non-mega targets, due to large code space requirement, long support is not presented. For mega targets, all format and modifiers are accepted.
- **AVR Studio Simulator IO** - If selected, use the library functions that interface to the Terminal IO Window in the AVR Studio. Note that you must copy the file iostudio.s into your source directory. See COFF and Working With AVR Studio.
- **Additional Libraries** - You may use other libraries besides the standard ones provided by the product. For example, on our website is a library called libstk.a for accessing STK-200 peripherals. To use other libraries, copy the files to the library directory and specify the names of the library files without the "lib" prefix and the ".a" extension in this box. For example, "stk" refers to the libstk.a library file. All library files must end with the .a extension.
- **Advanced** - the options here allow you finer control over the target customization:
  - **Return Stack Size** - the compiler uses two stacks, one for the return addresses for the function calls and interrupt handler (known as the hardware stack), and one for parameter passing and local storage (known as the software stack). This option allows you to control the size of the return stack. The size of the software stack does not need to be specified. See Stacks.

    Each function call or interrupt handler uses two bytes of the return stack. Therefore, you need to estimate the deepest level of your call trees (i.e. the maximum number of nested routines your program may call, possibly any interrupts), and enter the appropriate size here. **Programs using**

floating points or longs should specify a hardware stack size of at least 30 bytes.

- **Non Default Startup** - a startup file is always linked with your program (see Startup File). In some cases, you may have different startup files based on the project. This option allows you to specify the name of the startup file. If the filename is not an absolute pathname, then the startup file must be in the directory specified in the "Paths" tab.

Environment Options

This dialog box controls the general environment settings of the IDE:

- **Beep on Completing Build** - Emits a beep when a build is completed.
- **Verbose Compiler Output** - Directs the compiler driver to print out each pass as it processes the file. This shows the exact command line switches passed to each compiler pass.
- **Multiple Row Editor Tabs** - Changes the appearance of the editor tabs to use multiple row display instead of single row with scroll arrow when the number of editor labels grows too large to fit in the display.
- **Auto Save Files Before Compiling** - Saves the project files automatically when you request a Build. Normally, you are prompted to save unsaved changes for each changed file.
- **Create Backup on Save** - When saving a file, copies the last version to a file of the form <file>.~<ext> before overwriting it with the latest modifications.
- **Undo Across Save** - Allows undoing of changes even if the file has been saved.
- **Scan for Changes in Opened Files** - Periodically scans the opened files to see if the disk version has been changed. This is useful if you are using an external editor while keeping a file open in this IDE as well.
- **Close Files on Project Close** - Automatically closes all the project files when a project is closed.
- **Printer Setup** - Invokes the Printer Setup dialog box.

Editor and Print Options

There are three tabs that control the Editor and Print operations. File related editor options (e.g. whether to create backup on save), are part of the Environment Options. Some options are described "unused and is ignored."

Options

Print

- Wrap long lines - wrap long lines when printing
- Line numbers - print line numbers
- Title in header - print file name on header
- Date in header - print current date and time on header
- Page numbers - print page numbers

General

- Word wrap - word wrap text display
- Override wrapping - unused and is ignored
- Auto indent - When not in word-wrap mode, indent caret to first non-space character in line above
- Smart TAB - When not in word-wrap mode, TAB keys move to next non-space character in line above
- Smart fill - If use TAB character is on (below), this option causes the editor to use the minimum number of characters made up of TABs and spaces to fill a required gap. Otherwise, it uses spaces only.
- Use TAB character - Insert the TAB character into the text. If this checkbox is not checked, then the correct number of spaces are inserted in place of the TAB character.
- Line numbers in gutter - Show line numbers in the gutter
- Mark wrapped lines - Show a black triangle in the gutter for wrapped lines
- Title as filename - unused and is ignored
- Block cursor for overwrite - Show a block cursor when the editor is in overwrite mode
- Word select - Double clicking selects word nearest to mouse position
- Syntax highlight - Turn on Syntax Highlighting
- Cursor beyond EOL - Allow caret/cursor to move beyond the end of the line
- Show all chars - Show hidden white space characters as glyphs (applies to TAB, SPACE, NEWLINE and wrapped lines)

Others

- Right Margin - Display a right margin (vertical gray line) on specified column
- Gutter - Display a gutter of specified size. Note that the gutter is used for displaying bookmarks and line numbers, among other things.
- Block indent step size - Number of steps to use when indenting using the Block Indent and Outdent commands
- TAB Columns - Specify the position of the tab columns. If not specified, then the TAB stop value is used to calculate the positions of the TABs.
- TAB stop - number of characters to use per TAB if "TAB columns" is not used

Highlighting

This page allows you to control the highlighting display.

Key Assignments

The page allows you to modify the key assignments for editing commands.

Code Templates

This page allows you to define and edit "code templates" that you can access with a hotkey combination (defaults to Control-J, or ^J). Code templates are useful for filling out the basic syntactic elements without you doing all the typing. A list of templates for commonly used elements such as the C control structures is provided.

Terminal Options

Changing the COM port or the baud rate closes the COM port and reopens it if it was already open.

- **COM Port** - Specifies which COM port the terminal emulator should use.
- **Baudrate** - Specifies the Baudrate to use. All Windows standard Baudrates are supported.
- **Flow Control** - Controls the method of flow control.
- **ASCII Transfer Protocol** - If using ASCII Downloading to transfer an output file to a programming device, certain target programmers need certain protocols to be used. Note that the terminal emulator does not support the AVR ISP (In System Programming) function.

C LIBRARY AND THE STARTUP FILE

Startup File

The linker links in the startup file before your files, and links the standard library libcavr.a with your program. The startup file is either crtavr.o or crtatmega.o depending on the target device. The startup file defines a global symbol __start which is the starting point of your program. The startup file functions are:

I. Initialize the hardware and software stack pointers.

II. Copy the initialized data from the idata area to the data area.

III. Initialize the bss area to zero.

IV. Call the user main routine.

V. Define the entry point for exit, which is defined as an infinite loop. If main ever returns, it will enter exit and gets stuck there (so much for "exit").

The startup also defines the reset vector. You do not need to modify the startup file to use other interrupts, see Interrupt Handlers.

To modify and use a new startup file:

```
cd \icc\libsrc.avr          ; or wherever you install the compiler
<edit crtavr.s>
<open crtavr.s using the IDE>    ;
<Choose "Compile File To->Object">     ;generate a new crtavr.o
copy crtavr.o ..\lib        ; copy to the library directory
```

Substitute "crtatmega" for "crtavr" if you are targeting the Mega devices. Note that Mega devices use two words per interrupt entry whereas non-Mega devices only use one word per entry.

You can also have multiple startup files. You simply specify the name of the startup file to use in your Project Options dialog box. Note that you must either specify the startup with an absolute pathname or it must be located in the directory specified by the Project Options Library path.

C Library General Description

Library Source

The library source code (`c:\icc\libsrc.avr\libsrc.zip` by default) is a password protected zip file. Many unzip programs are available on the web if you do not already have one. The password is shown on the "About" box if the software has been unlocked. For example:

```
cd \icc\libsrc
unzip -s libsrc.zip
; unzip prompts for password
```

io*.h (io2313.h, io8515.h, iom603.h, ... etc.)

These files are converted from the official Atmel source files that define the IO registers. These files should be used instead of the older `avr.h`.

```
PORTB = 1;
uc = PORTA;
```

macros.h

This file contains useful macros and defines.

Other Header Files

The following standard C header files are supported. In general, it is a good practice to include the header files if you use the listed functions in your program. In the case of floating point or long functions, you must include the header files since the compiler must know about their prototypes. See Functions Returning Non-Integer Values.

- assert.h - assert(), the assertion macros
- ctype.h - character type functions.
- float.h - floating point characteristics.
- limits.h - data type sizes and ranges
- math.h - floating point math functions.
- stdarg.h - support for variable argument functions.
- stddef.h - standard defines.
- stdio.h - standard IO (input/output) functions.
- stdlib.h - standard library including memory allocation functions.
- string.h - string manipulation functions.

Character Type Functions

The following functions categorize input according to the ASCII character set. Use "#include <ctype.h>" before using these functions.

- int **isalnum**(int c)

  returns non zero if c is a digit or alphabetic.

- int **isalpha**(int c)

  returns non zero if c is an alphabetic.

- int **iscntrl**(int c)

  returns non zero if c is a control character (e.g. FF, BELL, LF ..etc.).

- int **isdigit**(int c)

  returns non zero if c is a digit.

- int **isgraph**(int c)

  returns non zero if c is a printable character and not a space.

- int **islower**(int c)

  returns non zero if c is a lower case alphabetic.

- int **isprint**(int c)

  returns non zero if c is a printable character.

- int **ispunct**(int c)

  returns non zero if c is a printable character and is not a space or a digit or an alphabetic.

- int **isspace**(int c)

  returns non zero if c is a space character including space, CR, FF, HT, NL, and VT.

- int **isupper**(int c)

  returns non zero if c is an upper case alphabetic.

- int **isxdigit**(int c)

  returns non zero if c is a hexadecimal digit.

- int **tolower**(int c)

  returns the lower case version of c if c is an upper case character. Otherwise it returns c.

- int **toupper**(int c)

  returns the upper case version of c if c is a lower case character. Otherwise it returns c.

Floating Point Math Functions

The following floating point math routines are supported. You must #include <math.h> before using these functions.

- float **asin**(float x)

  returns the arcsine of x for x in radians.

- float **acos**(float x)

  returns the arccosine of x for x in radians.

- float **atan**(float x)

  returns the arctangent of x for x in radians.

- float **atan2**(float x, float y)

  returns the angle whose tangent is y / x, in the range [-pi, +pi] radians.

- float **ceil**(float x)

  returns the smallest integer not less than x.

- float **cos**(float x)

  returns the cosine of x for x in radians.

- float **cosh**(float x)

  returns the hyperbolic cosine of x for x.

- float **exp**(float x)

  returns e to the x power.

- float **exp10**(float x)

  returns 10 to the x power.

- float **fabs**(float x)

  returns the absolute value of x.

- float **floor**(float x)

  returns the largest integer not greater than x.

- float **fmod**(float x, float y)

  returns the remainder of x / y.

- float **frexp**(float x, int *pexp)

  returns a fraction f and stores a base-2 integer into *pexp that represents the value of the input x. The return value is in the interval of [1/2, 1) and x equals f * 2**(*pexp).

- float **fround**(float x)

  rounds x to the nearest integer.

- float **ldexp**(float x, int exp)

  returns x * 2**exp

- float **log**(float x)

  returns the natural logarithm of x.

- float **log10**(float x)

  returns the base-10 logarithm of x.

- float **modf**(float x, float *pint)

  returns a fraction f and stores an integer into *pint that represents x. f + (*pint) equal x. abs(f) is in the

interval [0, 1) and both f and *pint have the same sign as x.

- float **pow**(float x, float y)
  returns x raised to the power y.
- float **sqrt**(float x)
  returns the square root of x.
- float **sin**(float x)
  returns the sine of x for x in radians.
- float **sinh**(float x)
  returns the hyperbolic sine of x for x.
- float **tan**(float x)
  returns the tangent of x for x in radians.
- float **tanh**(float x)
  returns the hyperbolic tangent of x.

Standard IO functions

Since standard file IO is not meaningful for an embedded microcontroller, much of the standard stdio.h content is not applicable. Nevertheless, some IO functions are supported. Use "#include <stdio.h>" before using these functions. You will need to initialize the output port. The lowest level of IO routines consists of the single character input (getchar) and output (putchar) routines. Thus, if you want to use a higher level function on a different device, for example printf to an LCD, all you need to do is to redefine the low level function. See Librarian.

To use the standard IO functions with the AVR Studio simulator (the Terminal IO Window), select the appropriate radio button in Compiler Options.

- int **getchar**()

  returns a character from the UART using polled mode.

- int **printf**(char *fmt, ..)

  prints out formatted text according to the format specifiers in the fmt string. The format specifiers are a subset of the standard formats:

  %d - prints the next argument as a decimal integer

  %o - prints the next argument as an unsigned octal integer

  %x - prints the next argument as an unsigned hexadecimal integer

  %X - the same as %x except that upper case is used for 'A'-'F'

  %u - prints the next argument as an unsigned decimal integer

  %s - prints the next argument as a C null terminated string

  %c - prints the next argument as an ASCII character

  %f - prints the next argument as a floating point number

  %S - prints the next argument as a constant string in FLASH

  If a '#' character is specified between '%' and 'o' or 'x', then a leading 0 or 0x is printed respectively. If you specify 'l' (letter el) between % and one of the integer format characters, then the argument is taken to be long, instead of int.

  printf is supplied in three versions, depending on your code size and feature requirements (the more features, the higher the code size):

  - Basic: only %c, %d, %x, %u, and %s format specifiers without modifiers are accepted.
  - Long: the long modifiers %ld, %lu, %lx are supported, in addition to the width and precision fields.
  - Floating point: all formats including %f for floating point are supported.

  The code size is significantly larger as you progress down the list. You select the version to use in the Compiler Options dialog box.

- int **putchar**(int c)

  prints out a single character. The library routine uses the UART in polled mode to output a single character. For convenience, writing a '\n' newline character causes a '\r' carriage return character to be output first.

- int **puts**(char *s)

  prints out a string followed by NL.

- int **sprintf**(char *buf, char *fmt)

  prints a formatted text into buf according to the format specifiers in fmt. The format specifiers are the same as in printf().

"**const char *" support functions**
`cprintf` and `csprintf` are exactly the same as `printf` and `sprintf` respectively except that the format string takes a FLASH string.

Standard Library And Memory Allocation Functions

The Standard Library header file <stdlib.h> defines the macros NULL and RAND_MAX and typedefs size_t and declares the following functions. Note that you must initialize the heap with the `_NewHeap` call before using any of the memory allocation routines (i.e.. `calloc, malloc,` and `realloc).`

- int **abs**(int i)

  returns the absolute value of i.

- int **atoi**(char *s)

  converts the initial characters in "s" into an integer, or returns 0 if an error occurs.

- double **atof**(const char *s)

  converts the initial characters in "s" into a double and returns it.

- long **atol**(char *s)

  converts the initial characters in "s" into a long integer, or returns 0 if an error occurs.

- void ***calloc**(size_t nelem, size_t size)

  returns a memory chunk large enough to hold "nelem" number of objects, each of size “size.” The memory is initialized to zeros. It returns 0 if it cannot honor the request.

- void **exit**(status)

  terminates the program. Under an embedded environment, typically it simply loops forever and its main use is to act as the return point for the user "main" function.

- void **free**(void *ptr)

  frees a previously allocated heap memory.

- void ***malloc**(size_t size)

  allocates a memory chunk of size “size” from the heap. It returns 0 if it cannot honor the request.

- void _**NewHeap**(void *start, void *end)

  initializes the heap for memory allocation routines. `Malloc` and related routines manage memory in the heap region. See Data Memory Usage for information on memory layout. A typical call uses the address of the symbol `_bss_end` as the "start" value. `_bss_end` is defined in the `endavr.o` file linked in automatically by the linker. It defines the end of the data memory used by the compiler for global variables and strings. The "end" value must not run into the stacks.

  ```
  extern void _bss_end;
  _NewHeap(&_bss_end, (char *)(&_bss_end) + 200); // 200 bytes heap
  ```

  Beware that for a microcontroller with a small amount of data memory, it is often not feasible or wise to use dynamic allocation due to its overhead and potential for memory fragmentation. Often a simple statically allocated array serves ones needs better.

- int **rand**(void)

  returns a pseudo random number between 0 and RAND_MAX.

- void ***realloc**(void *ptr, size_t size)

  reallocates a previously allocated memory chunk with a new size.

- void **srand**(unsigned seed)

  initializes the seed value for subsequent rand() calls.

- long **strtol**(char *s, char **endptr, int base)

  converts the initial characters in "s" to a long integer according to the "base." If base is 0, then "strtol" chooses the base depending on the initial characters (after the optional minus sign, if any) in s: 0x or 0X indicates a hexadecimal integer, 0 indicates an octal integer, with a decimal integer assumed

otherwise. If "endptr" is not NULL, then *endptr will be set to where the conversion ends in s.

- unsigned long **strtoul**(char *s, char **endptr, int base)

  is the same thing as "strtol" except that the number to be converted is an unsigned long and the return value is unsigned long.

String Functions

The following String functions are supported. Use "#include <string.h>" before using these functions. <string.h> defines NULL and typedefs size_t, and the following string and character array functions:

- void ***memchr**(void *s, int c, size_t n)

  searches for the first occurrence of "c" in the array "s" of size "n." It returns the address of the matching element or the null pointer if no match is found.

- int **memcmp**(void *s1, void *s2, size_t n)

  compares two arrays, each of size "n." It returns 0 if the arrays are equal and greater than 0 if the first different element in "s1" is greater than the corresponding element in "s2." Otherwise, it returns a number less than 0.

- void ***memmove**(void *s1, void *s2, size_t n)

  copies "s2" into "s1," each of size "n." The routine works correctly even if the inputs overlap. It returns s1.

- void ***memset**(void *s, int c, size_t n)

  stores "c" in all elements of the array "s" of size "n." It returns s.

- char ***strcat**(char *s1, char *s2)

  concatenates "s2" into "s1." It returns s1.

- char ***strchr**(char *s, int c)

  searches for the first occurrence of "c" in "s," including its terminating null character. It returns the address of the matching element or the null pointer if no match is found.

- int **strcmp**(char *s1, char *s2)

  compares two strings. It returns 0 if the strings are equal, and greater than 0 if the first different element in "s1" is greater than the corresponding element in "s2." Otherwise, it returns a number less than 0.

- char ***strcpy**(char *s1, char *s2)

  copies "s2" into "s1." It returns s1.

- size_t **strcspn**(char *s1, char *s2)

  searches for the first element in "s1" that matches any of the elements in "s2." The terminating nulls are considered part of the strings. It returns the index where the match is found.

- size_t **strlen**(char *s)

  returns the length of "s." The terminating null is not counted.

- char ***strncat**(char *s1, char *s2, size_t n)

  concatenates up to "n" elements, not including the terminating null, of "s2," into "s1." It then copies a null character onto the end of s1. It returns s1.

- int **strncmp**(char *s1, char *s2, size_t n)

  is the same as the `strcmp` function except it compares at most "n" characters.

- char ***strncpy**(char *s1, char *s2, size_t n)

  is the same as the `strcpy` function except it copies at most "n" characters.

- char ***strpbrk**(char *s1, char *s2)

  does the same search as the `strcspn` function except that it returns the pointer to the matching element in "s1" if the element is not the terminating null. Otherwise, it returns a null pointer.

- char ***strrchr**(char *s, int c)

  searches for the last occurrence of "c" in "s" and returns a pointer to it. It returns a null pointer if no

match is found.

- size_t **strspn**(char *s1, char *s2)

  searches for the first element in "s1" that does not match any of the elements in "s2." The terminating null of s2 is considered part of s2. It returns the index where the condition is true.

- char ***strstr**(char *s1, char *s2)

  finds the substring of "s1" that matches "s2." It returns the address of the substring if found and a null pointer otherwise.

"const char *" support functions

These functions perform the same processing as their counterparts without the 'c' prefix except that they operate on constant strings in FLASH.

- size_t cstrlen(const char *s)
- char *cstrcpy(char *dst, const char *src);
- int cstrcmp(const char *s1, char *s2);

Variable Argument Functions

<stdarg.h> provides support for variable argument processing. It defines the pseudo-type va_list, and three macros:

- **va_start**(va_list foo, <last-arg>)

  initializes the variable “foo.”

- **va_arg**(va_list foo, <promoted type>)

  accesses the next argument, cast to the specified type. Note that type must be a "promoted type," e.g. int, long, or double. Smaller integer types such as "char" should not be used.

- **va_end**(va_list foo)

  ends the variable argument processing.

For example, printf() may be implemented using vfprintf() as follows:

```
#include <stdarg.h>
int printf(char *fmt, ...)
   {
   va_list ap;

   va_start(ap, fmt);
   vfprintf(fmt, ap);
   va_end(ap);
   }
```

PROGRAMMING THE AVR

Accessing AVR Features

The strength of C is that while it is a high level language, it allows you to access low level features of the target devices. With this capability, there are very few reasons to use assembly except in cases where optimized code is of utmost importance. Even in cases where the target features are not available in C, usually inline assembly and preprocessor macros can be used to access these features transparently.

The header file `io*.h (e.g. io8515.h, iom603.h etc.)` define device specific AVR IO registers. These files are converted from the official Atmel release to match our compiler syntax. The file `macros.h` defines many useful macros. For example, the macro UART_TRANSMIT_ON() can be used to turn on the UART (when available on the target device).

The compiler is smart enough to generate the single cycle instructions such as `in, out, sbis, sbi,` etc. when accessing memory mapped IO. See `IO registers`.

Note: our older header file `avr.h` defines IO register bits as a bit mask, whereas `io*.h` defines them as a bit position. Thus to use `io*.h` and IO register bits, most of the time you will need to use the BIT() macro defined in `macros.h`. For example:

```
avr.h:

#define SRE       0x80  // external RAM enable

... (in your C program)

MCUCR |= SRE;


io8515.h

#define SRE       7

... (in your C program)

#include <macros.h>

MCUCR |= BIT(SRE);
```

Bit Twiddling

A common task in programming the microcontroller is to turn on or off some bits in the IO registers. Fortunately, Standard C is well suited to bit twiddling without resorting to assembly instructions or other non standard C constructs. C defines some bitwise operators that are particularly useful.

a | b - bitwise or, the expression denoted by "a" is bitwise or'ed with the expression denoted by "b." This is used to turn on certain bits, especially when used in the assignment form |=. For example:

```
PORTA |= 0x80;    // turn on bit 7 (msb)
```

a & b - bitwise and. This operator is useful for checking if certain bits are set. For example:

```
if ((PORTA & 0x81) == 0)      // check bit 7 and bit 1
```

Note that the parathesis is needed around the expressions of an & operator since it has lower precedence than the == operator. This is a source of many programming bugs in C programs.

a ^ b - bitwise exclusive or. This operator is useful for complementing a bit. For example, in the following case, bit 7 is flipped:

```
PORTA ^= 0x80;    // flip bit 7
```

~a - bitwise complement. This operator performs a ones-complement on the expression. This is especially useful when combined with the bitwise and operator to turn off certain bits:

```
PORTA &= ~0x80;   // turn off bit 7
```

The compiler generates optimal machine instructions for these operations. For example, the `sbic` instruction might be used for a bitwise and operator for conditional branching based on bit status.

Program Memory and Constant Data

The AVR is a Harvard architecture machine, separating program memory from data memory. There are several advantages to such a design. For example, the separate address space allows an AVR device to access more total memory than a conventional architecture. For example, the ATmega series allows up to 64K words of program memory and 64K bytes of data memory, and future devices with possibly even greater amounts of program memory may be available later. However, the program counter still remains at 16 bits.

Unfortunately, C was not invented on such a machine. In particular, C pointers are either data pointers or function pointers, and C rules already specify that you cannot assume data and function pointers can be converted back and forth. However, with a Harvard architecture machine like the AVR, even a data pointer may point to either data memory or to program memory.

There are no standard C rules for how to handle this. The ImageCraft AVR compiler uses the "const" qualifier to signify that an item is in the program memory. Note that for a pointer declaration, the const qualifier may appear in different places, depending whether it is qualifying the pointer variable itself or the items that it points to. For example:

```
const int table[] = { 1, 2, 3 };

const char *ptr1;

char * const ptr2;

const char * const ptr3;
```

"table" is a table allocated in the program memory. "ptr1" is an item in the data memory that points to data in the program memory. "ptr2" is an item in the program memory that points to data in the data memory. Finally, "ptr3" is an item in the program memory that points to data in the program memory. In most cases, items such as "table" and "ptr1" are probably the most typical. The C compiler generates the LPM instruction to access the program memory.

Note that the C Standard does not require "const" data to be put in the read-only memory, and in a conventional architecture, this would not matter except for access rights. So, this use of the const qualifier is unconventional, but within the allowable parameters of the C standard. However, this does introduce conflicts with some of the standard C function definitions.

For example, the standard prototype for "strcpy" is `strcpy(char *dst, const char *src)`, with the const qualifier of the second argument signifying that the function does not modify the argument. However, under ICCAVR, the const qualifier would indicate that the second argument points to the program memory which is likely not the case. Thus these functions are defined without the const qualifiers.

Finally, note that only const variables with file storage class will be put into FLASH. For example, variables that are defined outside of a function body or variables that have the static storage class have file storage class. **If you declare local variables with the const qualifier, they will not be put into FLASH and undefined behaviors may result.**

Strings

As explained in Program Memory and Constant Data, the separation of program and data memory in the AVR's Harvard architecture introduces some complexity. This page explains this complexity as it relates to literal strings.

Strings

The compiler places switch tables and items declared as const into program memory. The last thorny issue is the allocation of literal strings. The problem is that, in C, strings are converted to char pointers. If strings are allocated in the program memory, then either all the string library functions must be duplicated to handle different pointer flavors, or the strings must also be allocated in the data memory.

The ImageCraft compiler offers two options for dealing with this:

Default String Allocation

The default is to allocate the strings in both the data memory and the program memory. All references to the strings are to the copies in data memory. To ensure that their values are correct, at program startup the strings are copied from the program memory to the data memory. Thus, only a single copy of the string functions are needed (this is also exactly how the compiler implements initialized global variables).

If you wish to conserve space, you can allocate strings only in the program memory by using `const` character arrays. For example:

```
const char hello[] = "Hello World";
```

In this example, `hello` can be used in all contexts that a literal string can be used, except as an argument to the standard C library string functions, as explained above.

printf has been extended with the %S format character for printing out FLASH only strings. In addition, new string functions have been added to support FLASH only strings. (See String Functions.)

Allocating All Literal Strings to FLASH Only

You can direct the compiler to place all literal strings in FLASH only by selecting the **"Project->Options->Target->**Strings In FLASH Only" checkbox. Again, be aware that you must be careful when calling library functions. When this option is checked, effectively the type for a literal string is "`const char *`" and you must ensure that the function takes the appropriate argument type. Besides new "`const char *`" related **string functions**, `cprintf` and `csprintf` have been created to accept "`const char *`" as the format string type. See **Standard IO Functions**.

Stacks

The generated code uses two stacks: a hardware stack that is used by the subroutine calls and interrupt handlers, and a software stack for allocating stack frames for parameters, temporary and local variables. Although it may seem cumbersome, using two stacks instead of one allows the most transparent use of the data RAM.

The hardware stack is allocated at top of the data memory, and the software stack is allocated a number of bytes below that. The size of the hardware stack and the size of the data memory are controlled by settings in the target tab of Compiler Options. The data area is allocated starting at 0x60, right after the IO space. This allows the data area and the software stack to grow toward each other. You must not allow the hardware stack to grow into the software stack, otherwise Bad Things Can Happen (tm). A later release of the linker may automatically choose an optimal size for the hardware stack, unless recursive functions are involved.

A downside of allocating the stacks at the top of the data memory is that if external SRAM is used for additional data memory, then accesses to the stacks are slowed down by at least one additional clock cycle. A different method would place the stacks in the internal SRAM (which does not incur the additional clock penalty), and place global data in the external SRAM. Fortunately, since we provide the full source code to the startup file (and the library functions), you may modify the startup file to use this method instead.

Inline Assembly

Besides writing assembly functions in assembly files, inline assembly allows you to write assembly code within your C file. (Of course, you may use assembly source files as part of your project as well.) The syntax for inline assembly is:

```
asm("<string>");
```

Multiple assembly statements can be separated by the newline character \n. String concatenations can be used to specify multiple statements without using additional asm keywords. To access a C variable inside an assembly statement, use the %<name> format:

```
register unsigned char uc;
asm("mov %uc,R0\n"
    "sleep\n");
```

Any C variable can be referenced this way (excluding C goto labels). If you use an assembly instruction that requires a CPU register, then you must use the register storage class to force a local variable to be allocated in a CPU register.

In general, using inline assembly to reference local registers is limited in power: it is possible that no CPU registers are available if you have declared too many register variables in that function. In that case, you would get an error from the assembler. There is also no way to control allocation of register variables, so your inline instruction may fail. For example, using the `ldi` instruction requires the register to be one of the 16 upper registers, but there is no way to request that using inline assembly. There is also no way to reference the upper half of an integer register.

Inline assembly may be used inside or outside of a C function. The compiler indents each line of the inline assembly for readability. Unlike the AVR assembler, the ImageCraft assembler allows labels to be placed anywhere (not just at the first character of the lines in your file) so you may create assembly labels in your inline assembly code. You may get a warning on asm statements that are outside of a function. You may ignore these warnings.

IO Registers

IO registers, including the Status Register SREG, can be accessed in two ways. The IO addresses between 0x00 to 0x3F can be used with the IN and OUT instructions to read and write the IO registers, or the data memory addresses between 0x20 to 0x5F can be used with the normal data accessing instructions and addressing modes. Both methods are available in C:

- Data memory addresses. A direct address can be used directly through pointer indirection and type casting. For example, SREG is at data memory address 0x5F:

```
unsigned char c = *(volatile unsigned char *)0x5F; // read SREG

*(volatile unsigned char *)0x5F |= 0x80;  // turn on the Global
Interrupt bit
```

  Note that data memory 0 to 31 refer to the CPU registers! Extreme care must be taken not to change the CPU registers inadvertently.

  This is the **preferred method** since the compiler automatically generates the low level instructions such as `in, out, sbrs, sbrc,` etc. when accessing data memory in the IO register region.

- IO addresses. You may use inline assembly and preprocessor macros to access IO addresses:

```
register unsigned char uc;

asm("in %uc,$3F");      // read SREG

asm("out $3F,%uc");     // turn on the Global Interrupt bit
```

  In some early releases, this was the only way to use the faster low level `in and out` instructions but this is no longer the case.

Note: our older header file `avr.h` defines IO register bits as bit mask, whereas `io*.h` define them as bit position. Thus to use `io*.h` and IO register bits, most of the time you will need to use the BIT() macro defined in `macros.h`. For example:

```
avr.h:

#define SRE       0x80  // external RAM enable

... (in your C program)

MCUCR |= SRE;



io8515.h

#define SRE       7

... (in your C program)

#include <macros.h>

MCUCR |= BIT(SRE);
```

Addressing Absolute Memory Locations

Your program may need to address absolute memory locations. For example, external IO peripherals are usually mapped to specific memory locations. These may include LCD interface and dual port SRAM. Currently, you can use inline asm or a separate assembler file to declare data that are located in specific memory addresses. In a later release of the compiler, we may provide this capability in C.

In the following examples, assume there is a two-byte LCD control register at location 0x1000 and a two-byte LCD data register at the following location (0x1002), and there is a 100 byte dual port SRAM located at 0x2000.

Using an Assembler Module

In an assembler file, put the following:

```
.area memory(abs)

.org 0x1000

_LCD_control_register:: .blkw 1

_LCD_data_register:: .blkw 1

.org 0x2000

_dual_port_SRAM:: .blkb 100
```

In your C file, you must then declare them as:

```
extern unsigned int LCD_control_register, LCD_data_register;

extern char dual_port_SRAM[100];
```

Note the interface convention of prefixing an external variable names with an '_' in the assembler file and the use of two colons to define them as global variables.

Using Inline Asm

Inline asm is really just the same as regular assembler syntax, except that it is enclosed in the pseudo-function asm(). In a C file, the above assembly code becomes the following:

```
asm(".area memory(abs)"

      ".org 0x1000"

      "_LCD_control_register:: .blkw 1"

      "_LCD_data_register:: .blkw 1");

asm(".org 0x2000"

      "_dual_port_SRAM:: .blkb 100");
```

You still need to declare these as "extern" in C (as above), just as in the case of using a separate assembler file, since the C compiler does not really know what's inside the asm statements.

Interrupt Handling

C Interrupt Handlers

Interrupt handlers can be written in C. In the file where you define the function, before the function definition you must inform the compiler that the function is an interrupt handler by using a pragma:

```
#pragma interrupt_handler <name>:<vector number> *
```

"vector number" is the interrupt's vector number. Note that the vector number starts with one, which is the reset vector. This pragma has two effects:

- For an interrupt function, the compiler generates the `reti` instruction instead of the `ret` instruction, and saves and restores all registers used in the function.
- The compiler generates the interrupt vector based on the vector number and the target device.

For example,

```
#pragma interrupt_handler  timer_handler:4
...
void timer_handler()
   {
   ...
   }
```

The compiler generates the instruction

```
rjmp _timer_handler        ; for classic devices, or
jmp _timer_handler         ; for Mega devices
```

at location 0x06 (byte address) for the classic devices and 0xC (byte address) for the Mega devices (Mega devices use 2 word interrupt vector entries vs. 1 word for the classic non-Mega devices).

You may place multiple names in a single interrupt_handler pragma, separated by spaces. If you wish to use one interrupt handler for multiple interrupt entries, just declare it multiple times with different vector numbers. For example:

```
#pragma interrupt_handler timer_ovf:7 timer_ovf:8
```

Assembly Interrupt Handlers

You may write interrupt handlers in assembly. However, beware that if you call C functions inside your assembly handler, then the assembly routine must save and restore the volatile registers (see Assembly Interface) since the C functions do not do that (unless they are declared as interrupt handlers, but then they should not be called directly).

If you use assembly interrupt handlers, then you must define the vectors yourself. You use the "abs" attribute to declare an absolute area (see Assembler Directives) and use the ".org" statement to assign the rjmp or jmp instruction at the right location. Note that the ".org" statement uses byte address.

```
; for all but the ATMega devices
.area vectors(abs)         ; interrupt vectors
.org 0x6
rjmp _timer


; for the ATMega devices
.area vectors(abs)         ; interrupt vectors
.org 0xC
jmp _timer
```

Accessing EEPROM

The EEPROM can be accessed at runtime using library functions. Use #include <eeprom.h> before calling these functions:

- unsigned char **EEPROMread**(int location)

  Reads a byte from the specified EEPROM location.

- int **EEPROMwrite**(int location, unsigned char byte)

  Writes a byte to the specified EEPROM location. Returns 0 if successful.

- unsigned char **_256EEPROMread**(int location)

  Reads a byte from the specified EEPROM location for devices with 256 bytes of EEPROM, e.g. 2313 and 4414.

- int **_256EEPROMwrite**(int location, unsigned char byte)

  Writes a byte to the specified EEPROM location for devices with 256 bytes of EEPROM, e.g. 2313 and 4414. Returns 0 if successful.

The source code to the library is provided to licensed users, see C Library General. This allows you to tailor the device specific functions for your needs. Your may either add your new functions to the standard library `libcavr.c` (see Librarian), or add the new source file(s) as part of your project.

Accessing the SPI

A set of functions to access the SPI in polled master mode is provided. Please refer to `spi.h` for further information.

Relative Jump/Call Wrapping

On devices with 8K of program memory, all memory locations can be reached with the relative jump and call instructions (`rjmp` and `rcall`). To accomplish that, relative jumps and calls are wrapped around the 8K boundary. For example, a forward jump to byte location 0x2100 (0x2000 is 8K) is wrapped to the byte location 0x100.

This option is automatically detected by the Project Manager whenever the target Program Memory is exactly 8192 bytes.

C RUNTIME ARCHITECTURE

Data Type Sizes

TYPE	SIZE (bytes)	RANGE
unsigned char	1	0..256
signed char	1	-128..127
char (*)	1	0..256
unsigned short	2	0..65535
(signed) short	2	-32768..32767
unsigned int	2	0..65535
(signed) int	2	-32768..32767
unsigned long	4	0..4294967295
(signed) long	4	-2147483648..2147483647
float	4	1.175e-38..3.40e+38
double	4	1.175e-38..3.40e+38

(*) "char" is equivalent to "unsigned char"

floats and doubles are in IEEE standard 32-bit format with 7 bit exponent and 23-bit mantissa.

Bitfield types must be either signed or unsigned but they will be packed into the smallest space. For example:

```
struct {
   unsigned a : 1, b : 1;
};
```

Size of this structure is only 1 byte.

Assembly Interface and Calling Conventions

External Names

External C names are prefixed with an underscore, e.g. the function main is _main if referenced in an assembly module. Names are significant to 32 characters. To make an assembly object global, use two colons after the name. For example,

```
_foo::
   .word 1


(In the C file)
extern int foo;
```

Argument and Return Registers

The first argument is passed in registers R16/R17 if it is an integer and R16/R17/R18/R19 if it is a long or floating point. The second argument is passed in R18/R19. All other remaining arguments are passed on the software stack. Integer arguments smaller than ints (i.e. char) are extended to int size when passing to a function even if a function prototype is available. If R16/R17 is used to pass the first argument and the second argument is a long or float, the lower half of the second argument is passed in R18/R19 and the upper half is passed on the software stack.

Integer values are returned in R16/R17 and longs and floats are returned in R16/R17/R18/R19.

Preserved Registers

An assembly function must save and restore the following registers:

- R28/R29 or Y, this is the frame pointer.
- R10/R11/R12/R13/R14/R15/R20/R21/R22/R23

These registers are called preserved registers since their contents are unchanged by a function call.

Volatile Registers

The other registers:

- R0/R1/R2/R3/R4/R5/R6/R7/R8/R9/R24/R25/R26/R27/R30/R31
- SREG

can be used in a function without being saved or restored. These registers are called volatile registers since their contents may be changed by a function call.

Interrupt Handlers

Note that unlike a normal function, an interrupt handler must save and restore all registers that it uses. This is done automatically if you use the compiler capability to declare a C function as an interrupt handler. If you write a handler in assembly and if it calls normal C functions, then the assembly handler must save and restore the Volatile Registers since normal C functions do not preserve them. Since an interrupt handler operates asynchronous to the normal program operation, the interrupt handler or the functions it calls must not modify any machine registers.

Functions Returning Non-Integer Values

You must declare a function that returns a long, a floating point, or a structure value before you call it. For example, you should #include the header file <math.h> before calling any of the floating point functions. Otherwise, your program will not work since these routines return their values in different locations than functions returning integer values.

Long and Float Return Values

Long and float return values are in the same register set R16-R19.

Passing a Structure by Value

If passed by value, a structure is always passed through the stack, and not in registers. Passing a structure by reference (i.e. passing the address of a structure) is the same as passing the address of any data item, that is, a pointer (which is two bytes) is passed.

Returning a Structure by Value

When a function returning a structure is called, the calling function allocates a temporary storage and passes a secret pointer to the called function. When such a function returns, it copies the return value to this temporary storage.

C Machine Routines

Most C operations are translated into direct AVR instructions. However, there are some operations that are translated into subroutine calls since they involve many machine instructions and would cause too much code bloat if the translations are done inline. These routines are written in assembly language and can be distinguished by the fact that the routine names do not start with an underscore. Some of the commonly encountered routines with the following prefixes are:

- lsr16, lsr32, ... - perform shift operations on 16-bit and 32-bit data
- mpy, div, mod, neg, rmpy, rdiv, rmod, ... - 32-bit long and floating point routines

Program and Data Memory Usage

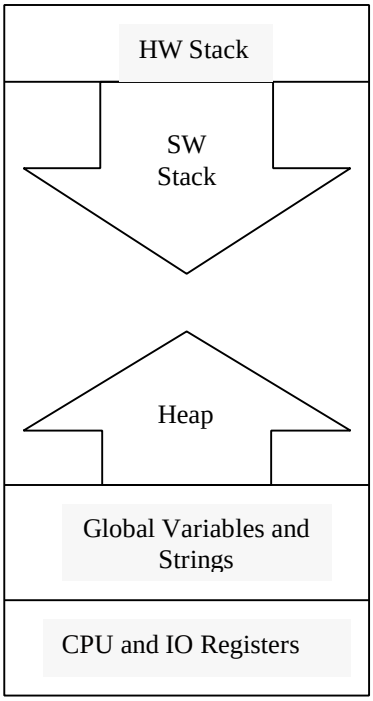
Program Memory

The program memory is used for storing your program code, constant tables, and initial values for certain data such as strings and global variables. See Program Memory and Constant Data. The compiler generates a memory image in the form of an output file that can be used by a suitable program such as an ISP (In System Flash) Programmer.

Currently, the compiler does not use any program memory above 64K bytes. To access memory above the 64K bytes boundary (e.g. on the mega 103 devices), you need to call the instruction ELPM directly after setting up the RAMPZ register.

Data Memory (Internal SRAM Only)

The Data Memory is used for storing variables, the stack frames, and the heap for dynamic memory allocation. In general, they do not appear in the output file but are used when the program is running. A program uses data memory as follows:



High Address

Low Address

The bottom of the memory map is address 0. The first 96 (0x60) locations are CPU and IO registers. The compiler places global variables and strings from 0x60 on up. On top of the variables is the area where you can allocate dynamic memory. See Memory Allocation Functions. At the high address, the hardware stack starts at the end of the SRAM. Below that is the software stack which grows downward. It is up to you, the programmer, to ensure that the hardware stack does not grow past the software stack, and the software stack does not grow into the heap. Otherwise, unexpected behaviors will result (*oops*...).

Data Memory (External SRAM)

If you select a device target with 32K or 64K of external SRAM, then the stacks are placed at the top of the internal SRAM and grow downward toward the low memory addresses. The data memory starts on top of the hardware stack and grows upward. The allocations are done differently because the internal SRAM has faster access time than external SRAM and in most cases, it is more beneficial to allocate stack items to the faster memory.

Program Areas

The compiler generates code and data into different "areas." See Assembler Directives. The areas used by the compiler, ordered here by increasing memory address, are:

Read-Only Memory

- **interrupt vectors** - This area contains the interrupt vectors.
- **func_lit** - function table area. Each word in this area contains the address of a function entry. To be fully compatible with the Code Compressor, all indirect function references must be through an extra level of indirection. This is done automatically for you in C if you invoke a function by using a function pointer. In assembly, the following example illustrates:

```
; assume _foo is the name of the function
.area func_lit
PL_foo:: .word _foo     ; create a function table entry
.area text
ldi R30,<PL_foo
ldi R31,>PL_foo
rcall xicall
```

  You may use the library function `xicall` to call the function indirectly after putting the address of the function table entry into the R30/R31 pair.

- **lit** - This area contains integer and floating point constants.
- **idata** - The initial values for the global data and strings are stored in in this area.
- **text** - This area contains program code.

Data Memory

- **data** - this is the data area containing global and static variables, and strings. The initial values of the global variables and strings are stored in the "idata" area and copied to the data area at `startup` time.
- **bss** - this is the data area containing "uninitialized" C global variables. Per ANSI C definition, these variables will get initialized to zero at startup time.

The job of the linker is to collect areas of the same types from all the input object files and concatenate them together in the output file. See Linker Operations.

DEBUGGING

Testing Your Program Logic

Since the compiler implements the ANSI C language, a common method of program development is in fact to use a PC compiler such as Borland C or Visual C and debug your program logic first by compiling your program as a PC program. Obviously, hardware-specific portions must be isolated and replaced or stubbed out using dummy routines. Typically, 95% or more of your program's code can be debugged using this method.

COFF Debug and Working With AVR Studio

If you select the COFF output file format, source level debugging information is available for COFF capable debuggers such as AVR Studio(r), which you can download for free from Atmel's website http://www.atmel.com. **Note that due to the limitations of the COFF format and AVR Studio, all source files and the COFF output file must be in the same directory**.

Basic debugging is available with the STANDARD version of the compiler tool. The STANDARD version generates C source line numbers and variable addresses and basic data type information.

The PROFESSIONAL version of the compiler tool generates full debugging information including structure types.

Using Terminal IO Under AVR Studio in Simulator Mode

In order to use the Terminal IO window in AVR Studio in simulator mode, you must do the following:

I. Copy the file `\icc\libsrc.avr\iostudio.s` to your project directory.

II. Modify Compiler Options to select "AVR Studio Compatible IO."

Once this is done, the simulator uses its Terminal IO Window for UART IO.

Limitations of AVR Studio

You need AVR Studio 1.52 or higher to work effectively with output from the compiler. The latest version of AVR Studio as of this writing is V3 Beta.

Listing File

One of the output files produced by the compiler is a listing file of the name <exec file>.lst. **Filenames with .lis extensions are assembler output listing files and do not contain full information and should not be used.** The listing file contains your program's assembly code as generated by the compiler, interspersed with the C source code and the machine code and program locations. While global code and data are always displayed, due to the unfortunate nature of a relocatable linker, some of the static code or data will not be displayed with the final program locations. In addition, library code is not included.

Nevertheless, even with these limitations, the listing file is invaluable for debugging your program if you do not have debugging tools that understand the COFF debug format. Some low-cost In Circuit Emulators (ICE) may also use the listing file to drive their debugging, in addition to or in place of COFF.

Note that as of this release, when run, the Code Compressor does not update the content of the listing file. This will be fixed in a lter release.

COMMAND LINE COMPILER OVERVIEW

Compilation Process

[*Underneath the user friendly IDE is a set of command line compiler programs. While you do not need to understand this chapter to use the compiler, this chapter is good for those who want to find out "what's under the hood."*]

Given a list of files in a project, the compiler's job is to translate the files into an executable file in some output format. Normally, the compilation process is hidden from you through the use of the IDE's Project Manager. However, it can be important to have an understanding of what happens "under the hood":

I. The compiler compiles each C source file to an AVR assembly file.

II. The assembler translates each assembly file (either from the compiler or assembly files that you have written) into a relocatable object file.

III. After all the files have been translated into object files, the linker combines them together to form an executable file. In addition, a map file, a listing file, and debug information files are also output.

All these details are handled by the compiler driver. You give it a list of files and ask it to compile them into an executable file (default) or to some intermediate stage (e.g. to the object files). The driver invokes the compiler, the assembler and the linker as needed.

Actually, the IDE does not even interface with the compiler driver directly. It generates a makefile and invokes the "make" program to interpret the makefile, which causes the compiler driver to be invoked.

Driver

The compiler driver examines each input file and acts on the file based on the file's extension and the command line arguments it has received. .c files are C source files and .s files are assembly source files respectively. The design philosophy for the IDE is to make it as easy to use as possible. The command line compiler, though, is extremely flexible. You control its behavior by passing command line arguments to it. If you want to interface with the compiler with your own GUI (for example, the Codewright or Multiedit editor), here are some of the things you need to know.

- Error messages referring to the source files begin with "!E file(line):.."
- To bypass the command line length limit on Windows 95/NT, you may put command line arguments in a file, and pass it to the compiler as @file or @-file. If you pass it as @-file, the compiler will delete "file" after it is run.

The next page, Compiler Arguments, elaborates further on the subject.

Compiler Arguments

The IDE controls the behaviors of the compiler by passing command line arguments to the compiler Driver. Normally you do not need to know what these command line arguments do, but you can see them in the generated makefile and in the Status Window when you perform a build. Nevertheless, this page documents the options as used by the AVR IDE in case you want to drive the compiler using your own editor/IDE such as Codewright. All arguments are passed to the driver and the driver in turn passes the appropriate arguments down to different passes.

The general format of a command is

```
iccavr [ command line arguments ] <file1> <file2> ... [ <lib1> ... ]
```

where iccavr is the name of the compiler driver. As you can see, you can invoke the driver with multiple files and the driver will perform the operations on all of the files. By default, the driver then links all the object files together to create the output file.

For most of the common options, the driver knows which arguments are destined for which compiler passes. You can also specify which pass an argument applies to by using a -W<c> prefix. For example:

- -Wp is the preprocessor. e.g -Wp-e
- -Wf is the compiler proper. e.g. -Wf-atmega
- -Wa is the assembler
- -Wl (letter el) is the linker

Arguments Affecting the Driver itself

- -c Compile the file to the object file level only (does not invoke the linker.)
- -o <name> Name the output file. By default, the output file name is the same as the input file name, or the same as the first input file if you supply a list of files
- -v Verbose mode. Print out each compiler pass as it is being executed.

Preprocessor Arguments

- -D<name>[=value] Define a macro. See Compiler Options: Macro Define(s). Note that you should define ATMEGA if the target is a ATMega device since there are some definitions in the header file avr.h which depend on this being defined.
- -U<name> Undefine a macro. See Compiler Options: Macro Undefine(s).
- -e Accept C++ comments.
- -I<dir> (Capital letter i) Specify the location(s) to look for header files. Multiple -I flags can be supplied.

Compiler Arguments

- -l (letter el) Generate a listing file.
- -A -A (two -As) Turn on strict ANSI checking. Single -A turns on some ANSI checking.
- -g Generate debug information.
- -Mavr_mega Generate ATMega instructions such as call and jmp instead of rcall and rjmp.
- -Mavr_enhanced Generate enhanced core instructions and call and jmp
- -Mavr_enhanced_short Generate enhanced core instructions but not call and jmp
- -str_in_flash Allocate literal strings in FLASH only

Assembler Arguments

- -W Turn on relocation wrapping. See Relative Jump/Call Wrapping. Note that you need to use the -Wa prefix since the driver does not know of this option directly (i.e. -Wa-W).

Linker Arguments

- -L<dir> Specify the library directory. Only one library directory (the last specified) will be used.
- -O Invoke the Code Compressor (operative only for the PROFESSIONAL version).
- -m Generate a map file.
- -g Generate debug information.
- -u<crt> Use <crt> instead of the default startup file. If the file is just a name without path information, then it must be located in the library directory.
- -W Turn on relocation wrapping. See Relative Jump/Call Wrapping. Note that you need to use the -Wl prefix since the driver does not know of this option directly (i.e. -Wl-W).
- -fihx_coff Output format is both COFF and Intel HEX.
- -fcoff Output format is COFF
- -fintelhex Output format is Intel HEX
- -fmots19 Output format is Motorola S19 (not used on AVR - you can ignore this).
- -bfunc_lit:<address ranges> Assign the address ranges for the area named "func_lit." The format is <start address>[.<end address>] where addresses are word address. Any memory not used by this area will be consumed by areas that follow it, so this effectively declares the size of the FLASH memory. For example, some typical values are:

  ```
  -bfunc_lit:0x60.0x10000 for ATMega
  -bfunc_lit:0x1a.0x800   for 23xx
  -bfunc_lit:0x1a.0x2000  for 85xx
  ```

- -bdata:<address ranges> Assign the address ranges for the area or section named "data," which is the data memory on the AVR. For example, some typical values are:

  ```
  -bdata:0x60.0x800       for ATMega
  -bdata:0x60.0x80        for 23xx
  -bdata:0x60.0x200       for 85xx
  ```

- -dram_end:<address> Define the end of the data area. This is used by the startup file to initialize the value of the hardware stack. For the classic non-Mega devices without external SRAM, "ram_end" is the size of the SRAM plus 96 bytes of IO and CPU registers minus one. For the Mega devices, it is the size of the SRAM minus one. If a 64K external SRAM is attached, then it is 0xFFFF for all devices.
- -dhwstk_size:<size> Define the size of the hardware stack. The hardware is allocated at the top of SRAM, then the software stack follows it. See Stacks.
- -l<lib name> Link in the specific library files in addition to the default libcavr.a. This can be used to change the behavior of a function in libcavr.a since libcavr.a is always linked in last. The "libname" is the library file name without the "lib" prefix and without the ".a" suffix. For examples:

  ```
  -lstudio    "libstudio.a"     using withAVR Studio IO
  -llpavr     "liblpavr.a"      using full printf
  -lfpavr     "libfpavr.a"      using floating point printf
  ```

TOOLS REFERENCES

Code Compressor (tm)

The Code Compressor (tm) optimizer is a state-of-the-art optimization that reduces your final program size from 5-18%. It works on your entire program, searching across all files for opportunities to reduce the program size.

Advantages

- Code Compressor decreases your program size transparently. It does not interfere with traditional optimizations and can decrease code size even when aggressive traditional optimizations are done.
- Unlike other similar schemes, this is the first implementation that we are aware of in a commercial embedded compiler that optimizes the entire program.
- Code Compressor does not affect source level debugging with AVR Studio.

Disadvantage

- There is a slight increase in execution time due to function call overhead.

Compatibility Requirements

To be fully compatible with the Code Compressor, you should note the following:

- Indirect function references must be done through a function label entry in the "`func_lit`" output area. See `Program Areas`. This is done automatically if you are using C.
- Unlike the previous releases of the product, the program areas `func_lit, lit`, and `idata` now preceed the `text` program area. This allows the `text` area to shrink. However, this means that you should not create other named sections in the FLASH program area. (Contact ImageCraft if this capability is required.)
- If you are using the command line compiler directly and are upgrading from a pre-6.16 version, refer to `Compiler Arguments` for changes in the starting address specification.

Temporarily Deactivating the Code Compressor

Sometimes you may wish to disable the code compressor temporarily. For example, perhaps the code is extremely timing sensitive and it cannot afford to lose cycles by going through the extra function call and return overhead. You can do this by bracketing code fragments with an instruction pair:

```
tst R28

...

tst R29
```

If such a pair is found and if there is no return instruction between them, then the code compressor ignores the instructions in the fragment. Since these particular instructions test the stack pointer, it is unlikely that they would appear in the code accidentally.

The include file `macros.h` contains two new definitions:

```
COMPRESS_DISABLE();          // disable Code Compressor

COMPRESS_REENABLE();         // enable Code Compressor again
```

for use in C programs. Again, please note that there cannot be a return statement between the start and end of a fragment. Otherwise, these two instructions will be ignored.

Configuration Management With RCS

The PROFESSIONAL version of the software provides a set of configuration management tools and an IDE interface to manage your source code. The command line software is the GNU Revision Control System (RCS) utilities (see Acknowledgements for remarks on GNU software). RCS manages multiple revisions of the source files, allowing you to look at an older revision of the file if needed. The IDE provides a simple interface to RCS which is sufficient for the most common tasks. To perform the more advanced tasks, you must use the command line RCS utilities directly. This page describes some of the more common RCS functions (visit http://www.gnu.org for full GNU RCS documentation).

RCS Repository

For each file under RCS control, RCS keeps a master record of the file, containing all changes made to the file at each revision. Typically the RCS repository is the subdirectory named RCS in the source file location. The IDE creates the repository automatically.

Each revision of the file has a revision number and an optional label. You refer to a particular revision by its number or label. A label is useful to snapshot a particular set of changes (for example, before you release your software).

In advanced usage, you may even modify an older revision and "merge" in your changes, or have multiple changes to the same file made by different people and have RCS reconcile the different changes (unless there are actual conflicts). These advanced topics will not be discussed further in this document.

Files Checkin and Checkout

To add a new revision of a file to the repository, you use the checkin command ("ci" utility). In order to modify a file in the repository, you use the checkout command ("co" utility). In the simplest case, a special option to "ci" checks in the file and then performs an immediate checkout so that you can continue to modify the file.

The IDE uses "ICCAVR" as the logname of the files in the repository.

Assembler Syntax

The assembler has the following syntax

Names

All names in the assembler must conform to the following specification:

```
('_' | [a-Z]) [ [a-Z] | [0-9] | '_' ] *
```

That is, a name must start with either an underscore (_) or an alphabetic character, followed by a sequence of alphabetics, digits, or underscores. In this document, names and symbols are synonyms for each other. A name is either the name of a symbol, which is a constant value, or the name of a label, which is the value of the Program Counter (PC) at that moment. A name can be up to 30 characters in length. Names are case sensitive except for instruction mnenomics and assembler directives.

Name Visibility

A symbol may either be used only within a program module or it can be made visible to other modules. In the former case, the symbol is said to be a **local** symbol, and in the latter case, it is called a **global** symbol.

If a name is not defined within the file it is referenced in, then it is assumed to be defined in another module and its value will be resolved by the linker. The linker is sometimes referred as a relocatable linker precisely because one of its purposes is to relocate the values of global symbols to their final addresses.

Numbers

If a number is prefixed with 0x or $, then the number is taken to be a hexadecimal number:

```
Example:    10

            0x10

            $10

            0xBAD

            0xBEEF

            0xC0DE

            -20
```

Input File Format

Input to the assembler must be an ASCII file that conforms to certain conventions. Each line must be of the form:

```
[label: [:]] [command] [operands] [;comments]

[] – optional field
```

Each field must be separated from another field by a sequence of one or more “space characters,” which are either spaces or tabs.

Labels

A name followed by one or two colons denotes a label. The value of the label is the value of the Program Counter (PC) at that point of the program. A label with two colons is a global symbol; that is, it is visible to other modules.

Commands

A command can be an AVR instruction, an assembler directive or a macro invocation. The “operands” field denotes the operands needed for the command.

Expressions

An instruction operand may involve an expression. For example, the direct addressing mode is simply an expression:

```
lds R10,asymbol
```

“asymbol” is an example of the simplest expression, which is just a symbol or label name. In general, an expression is described by:

```
expr: term |
      { expr }

      unop expr

      expr binop expr

term: .           <- current program counter

      name |

      #name
```

Parentheses () provide grouping. Operator precedents are given below. Expressions cannot be arbitrarily complex, due to the limitations of relocation information communicated to the linker. The basic rule is that for an expression, there can only be only one relocatable symbol. For example,

```
lds R10,foo+bar
```

is invalid if both foo and bar are external symbols.

Operators

The following is the list of the operators and their precedents. Operators with higher precedents are applied. Only the addition operator may apply to a relocatable symbol (i.e. an external symbol). All other operators must be applied to constants or symbols resolvable by the assembler (i.e. a symbol defined in the file).

OPERATOR	FUNCTION	TYPE	PRECEDENT
*	multiply	binary	10
/	divide	binary	10
%	modulo	binary	10
<<	left shift	binary	5
>>	right shift	binary	5
^	bitwise exclusive OR	binary	4
&	bitwise exclusive AND	binary	4
|	bitwise OR	binary	4
-	negate	unary	11
~	one’s complement	unary	11

<	low byte	unary	11
>	high byte	unary	11

“Dot” or Program Counter

If a dot (.) appears in an expression, then the current value of the Program Counter (PC) is used in place of the dot.

Assembler Directives

Assembly directives are commands to the assembler. Directives are case insensitive.

.area <name> [(attributes)]

Defines a memory region to load the following code or data. The linker gathers all areas with the same name together and either concatenates or overlays them depending on the area's attributes. The attributes are:

```
abs, or     <- absolute area

rel         <- relocatable area
```

followed by

```
con, or     <- concatenated

ovr         <- overlay
```

The starting address of an absolute area is specified within the assembly file itself whereas the starting address of a relocatable area is specified as a command option to the linker. For an area with the "con" attribute, the linker concatenates areas of that name one after another. For an area with the "ovr" attribute, for each file, the linker starts an area at the same address. The following illustrates the differences:

```
file1.o:

      .area text <- 10 bytes, call this text_1

      .area data <- 10 bytes

      .area text <- 20 bytes, call this text_2

file2.o:

      .area data <- 20 bytes

      .area text <- 40 bytes, call this text_3
```

text_1, text_2, and so on are just names used in this example. In practice, they are not given individual names. Lets assume that the starting address of the text area is set to zero. Then, if the text area has the "con" attribute, text_1 would start at 0, text_2 at 10, and text_3 at 30. If the text area has the "ovr" attribute, then text_1 and text_2 would again have the addresses 0 and 10 respectively, but text_3, since it starts in another file, would also have 0 as the starting address. All areas of the same name must have the same attributes, even if they are used in different modules. Here are examples of the complete permutations of all acceptable attributes:

```
.area foo(abs)

.area foo(abs,con)

.area foo(abs,ovr)

.area foo(rel)

.area foo(rel,con)

.area foo(rel,ovr)
```

.ascii "strings"

.asciz "strings"

These directives are used to define strings, which must be enclosed in a delimiter pair. The delimiter can be any character as long as the beginning delimiter matches the closing delimiter. Within the delimiters, any printable ASCII characters are valid, plus the following C-style escape characters, all of which start with a backslash ('\'):

```
\e    escape

\b    backspace

\f    form feed

\n    line feed

\r    carriage return

\t    tab

\<up to 3 octal digits> character with value equal to the octal digits
```

ASCIZ adds a NUL character (\0) at the end. It is acceptable to embed \0 within the string.

```
Examples:   asciz "Hello World\n"

            asciz "123\0456"
```

.byte <expr> [,<expr>]*

.word <expr> [,<expr>]*

.long <expr> [,<expr>]*

These directives define constants. The three directives denote byte constant, word constant (2 bytes) and long word constant (4 bytes) respectively. Word and long word constants are output in little endian format, the format used by the AVR microcontrollers. Note that **.long** can only have constant values as operands. The other two may contain relocatable expressions.

```
Example:    .byte 1, 2, 3

            .word label,foo
```

.**blkb <value>**

.blkw <value>

.blkl <value>

These directives reserve space without giving them values. The number of items reserved is given by the operand.

.define <symbol> <value>

Defines a textual substitution. Whenever "symbol" is used inside an expression (e.g. a register specifier or a label), it is replaced with "value." For example:

```
.define quot R15
mov   quot,R16
```

.else

Forms a conditional clause together with a preceding **".if"** and following **".endif."** If the "if" clause conditional is true, then all the assembly statements from the **".else"** to the ending **".endif"** (the else clause) are ignored. Otherwise, if the "if" clause conditional is false, then the "if" clause is ignored and the

'else' clause is processed by the assembler. See **.if**.

.endif

Ends a conditional statement. See **.if** and **.else**.

.endmacro

Ends a macro statement. See **.macro**.

<symbol> = <value>

Defines a constant value for a symbol.

```
Example:    foo = 5
```

.if <symbol name>

If the "symbol name" has a non zero value, then the following code, up to either the **".else**" statement or the **".endif**" statement (whichever occurs first), is assembled. Conditionals can be nested up to 10 levels. For example:

```
.if cond

lds R10,a

.else

lds R10,b

.endif
```

would load "a" into R10 if the symbol "cond" is non zero, and load "b" into R10 if "cond" is zero.

.include "<filename>"

Processes the contents in the file specified by "filename." If the file does not exist, then the assembler will try to open the file name created by concatenating the path specified via the –I command line switch with the specified filename.

```
Example:    .include "registers.h"
```

.macro <macroname>

Defines a macro. The body of the macro consists of all the statements up to the **.endmacro** statement. Any assembly statement is allowed in a macro body except for another macro statement. Within a macro body, the expression *@digit* where *digit* is between 0 and 9 is replaced by the corresponding macro argument when the macro is invoked. You cannot define a macro name that conflicts with an instruction mnemonic or an assembly directive. See **.endmacro** and **Macro Invocation**. For example,

Defines a macro named "foo:"

```
.macro foo

lds @0,a

mov @1,@0

.endmacro
```

Invoking foo with two arguments:

```
foo R10,R11
```

is equivalent to writing

```
lds R10,a

mov R11,R10
```

.org <value>

Sets the Program Counter (PC) to "value." This directive is only valid for areas with the "abs" attribute. Note that "value" is a byte address.

Example: .area interrupt_vectors(abs)
org 0xFFD0
dc.w reset

.globl <symbol> [, <symbol>]*

Makes the symbols defined in the current module visible to other modules. This is the same as having a label name followed by two periods (.). Otherwise, symbols are local to the current module.

<macro> [<arg0> [,<args>]*]

Invokes a macro by writing the macro name as an assembly command followed by a number of arguments. The assembler replaces the statement with the body of the macro, expanding expressions of the form @digit with the corresponding macro argument. You may specify more arguments than are needed by the macro body but it is an error if you specify less arguments than needed.

Example: foo bar,x

Invokes the macro named “foo” with two arguments, “bar” and “x.”

Linker Operations

The main purpose of the linker is to combine multiple object files into an output file suitable to be loaded by a device programmer or target simulator. The linker can also take input from a "library" which is basically a file containing multiple object files. In producing the output file, the linker resolves any references between the input files. In some detail, the linking steps involve:

I. Making the Startup File be the first file to be linked. The startup file initializes the execution environment for the C program to run.

II. Appending any libraries that you explicitly requested (or in most cases, as were requested by the IDE) to the list of files to be linked. Library modules that are directly or indirectly referenced will be linked in. All the user specified object files (e.g. your program files) are linked.

III. Appending the standard C library libcavr.a and a special object module endavr.o to the end of the file list.

IV. Scanning the object files to find unresolved references. The linker marks the object file (possibly in the library) that satisfies the references and adds to its list of unresolved references.It repeats the process until there are no outstanding unresolved references.

V. Combining all marked object files into an output file and generating map and listing files as needed.

Lastly, if this is the PROFESSIONAL version and if the Code Compressor optimization option is on, then the Code Compressor is called.

Librarian

A library is a collection of object files in a special form that the linker understands. When a library's component object file is referenced by your program directly or indirectly, the linker pulls out the library code and links it to your program. The standard supplied library is libcavr.a, which contains the standard C and AVR specific functions. Other libraries, such as libstudio.a, override some functions in libcavr.a so that a different behavior can be achieved without changing your program code. For example, by linking in `libstudio.a`, your program may use the Terminal IO window under AVR Studio. Of course, the linking process is mostly transparent to you - by choosing the appropriate Compiler Options, the IDE generates the correct switches to the compiler programs.

Nevertheless, there are times where you need to modify or create libraries. A command line tool called *ilibw.exe* is provided for this purpose (note: the PROFESSIONAL version *may* include capabilities that handle library files within the IDE, please inquire for details).

Note that a library file must have the .a extension. See Linker Operations.

Compiling a File into a Library Module

Each library module is simply an object file. Therefore, to create a library module, you need to compile a source file into an object file. This can be done by opening the file into the IDE, and invoking the **File->Compile File To Object** command.

Listing the Contents of a Library

On a command prompt window, change the directory to where the library is, and give the command "ilibw -t <library>". For example,

```
ilibw -t libcavr.a
```

Adding or Replacing a Module

I. Compile the source file into an object module.

II. Copy the library into the work directory

III. Use the command "ilibw -a <library> <module>" to add or replace a module.

For example, the following replaces the putchar function in libcavr.a with your version.

```
cd \icc\libsrc.avr
<modify putchar() in iochar.c>
<compile iochar.c into iochar.o>
copy \icc\lib\libcavr.a    ; copy library
ilibw -a libcavr.a iochar.o
copy libcavr.a \icc\lib    ; copy back
```

ilibw creates the library file if it does not exist, so to create a new library, just give ilibw a new library file name.

Deleting a Module

The command switch -d deletes a module from the library. For example, the following deletes iochar.o from the libcavr.a library:

```
cd \icc\libsrc.avr
copy \icc\lib\libcavr.a    ; copy library
ilibw -d libcavr.a iochar.o      ; delete
copy libcavr.a \icc\lib    ; copy back
```