

Jennaron Research

**SCCAVR - A SmallC Compiler
for
AVR Microprocessors**

Manual revision 3 (23-9-99)

Ron Kreymborg
Compiler Revision 2
September 1999

Table of Contents

TABLE OF CONTENTS	2
INTRODUCTION	3
COMPILER QUICK START	4
COMMAND LINE FLAGS	5
SOME HELPFUL HINTS	6
INSTALLING THE SOFTWARE	7
<i>Installing SCCAVR</i>	7
<i>Installing the IAR Assembler</i>	7
<i>Installing AVR Studio</i>	8
THE C IMPLEMENTATION.....	8
<i>Floating Point</i>	8
<i>Structures</i>	9
<i>Global Initialisation</i>	10
<i>Functions</i>	11
<i>Interrupt Functions</i>	11
<i>The register Storage Qualifier</i>	12
<i>The long Data Type</i>	12
<i>Arithmetic</i>	12
<i>Input Output</i>	13
CODE EFFICIENCY	14
THE RUNTIME INITIALIZATION FILE	16
MONITORING STACK USAGE.....	19
THE RUNTIME LIBRARY	20
SOME TEST PROGRAMS	22
<i>A Simple LED Flasher</i>	22
<i>External Memory on an 8515</i>	24
<i>Driving a Two Line LCD Display</i>	26
<i>Using a Multi-processing Task Dispatcher</i>	30
CALLING ASSEMBLY FUNCTIONS	34
USING INTERRUPTS WITH SMALLC.....	36
<i>Program One</i>	37
<i>Program Two</i>	39
<i>Program Three</i>	40

Introduction

The original Smallc provided a basic K&R version of C without structures, unions, and the more complex data types (only unsigned chars and ints). It also lacked many features such as unsigned variables, function prototyping, casts, longs, structures, register variables, etc. It also showed its lineage in the days of single instruction/data spaces - there was no mechanism to use different instructions for different data sources. This was a requirement for most microprocessors using a Harvard architecture like the Atmel AVRs.

However, Smallc looked like what I wanted, a compiler I could modify as necessary to suit my requirements, so I decided to convert it to an ANSI C compiler, albeit with perhaps a few bits still missing.

For some time I persevered with the Atmel WAVRASM program as my assembler, using all sorts of contortions to include special assembler files whenever a library function was called for. Finding the IAR relocating DOS Assembler-Linker-Librarian in the public domain solved all these problems and now all the little snippets of code required for C runtime support could be defined as modules in an already assembled relocatable library and simply linked in at link time. This makes for very efficient run-time code and simplicity itself to modify. It also provides all the advantages of being able to modularise a project.

The AVR architecture is not really aimed at emulating a stack machine, whereas SmallC assumes such a machine. I initially wrote the AVR code generator to use the AVR push and pop instructions. It worked but was not code space efficient. Using the Y register pair as the C runtime stack pointer and leaving the hardware stack pointer to manage subroutines and interrupts is far more efficient. However this requires two stacks - one for subroutines and one for the C code. SmallC for AVR now by default implements the top of sram as the start of the hardware stack, with the C stack (usually) starting lower down. The command line includes options to set the addresses of these hardware and software stacks. You must at least estimate how many levels of hardware stack you will need and set that aside. It is then fairly simple to write some code that monitors hardware stack usage during program development, typically using a sentinel byte at the bottom of the stack. With no standard IO path for reporting this is difficult to include in the compiler.

The SmallC compiler, now called **sccavr**, has been modified to take as input a C source file and produces as output an assembly code file using IAR syntax. The input file is expected to use a .c extension. The output file will have the same name as the source file but with an .s90 extension. The IAR tools link the various relocatable objects, thus allowing multi-module projects with all the simplicity and encapsulation that implies. The IAR linker can be programmed to produce a runtime file in .r90 format for input to the Atmel AVR Studio debugger, or in Intel .hex format for downloading to the target AVR microprocessor.

Compiler Quick Start

Assuming the compiler and IAR tools are installed (see *Installing the Software*), a typical command line would be:

```
sccavr testing.c
```

An error free compile will produce an output file having the same name with an s90 extension as required by the IAR assembler. You then run the IAR assembler (called aa90) with the command:

```
aa90 testing -L -v1
```

You do not need to type the s90 extension on the input file name. The **-L** produces a listing. The **-vn** option specifies a peculiar IAR convention about code and data sizes, where **-v1** defines 64Kbytes of data and 8Kbytes of code. The relocatable output file will have an r90 extension. If there are any include files, you can specify their path using the **-I** parameter. If the assemble is successful you then run the IAR linker (called xlink) with the one line command:

```
xlink crti testing libcavr -ca90 -Z(DATA)sdata=60 -Z(CODE)scode=0  
-Fintel-standard -o testing.hex -xsem -l map
```

This links the C run time initialisation file crti.r90 as the first object and then the assembled compiler output object file testing.r90. It then examines the runtime library libcavr.r90 for unresolved external names and links their associated modules in as well. The **-ca90** specifies the AVR products, and the **-Z(DATA)** and **-Z(CODE)** parameters specify the starting address of the data and code segments respectively. The compiler names these two segments internally as **sdata** and **scode**, and they represent the AVR SRAM and flash PROM areas respectively. The **-Fintel-standard** specifies an Intel hex file as the required output format, and **-o testing.hex** specifies the output file name.

The **-xsem -l map** parameters are optional and produce a cross-referenced linker listing called map.lst showing exactly where all modules will be loaded within memory.

Like the **aa90** command, you can specify any paths to libraries with the **-I** parameter. You are not limited to compiling and linking a single program. One of the nice features of the IAR assembler/linker programs are relocatable modules. You can link any number of files simply by entering the above xlink command line and including the additional file names after the startup filename. More typically you would be using a *make* file to simplify all this and these commands would be in the Makefile file.

Before running the **xlink** command, you will need to select a startup file to suit your processor (see *The Runtime Initialisation File*), rename it crti.s90, and assemble it using the command:

```
makeinit
```

You will also need to assemble the runtime library (see *The Runtime Library*) with the command:

```
make1ib
```

Both these commands and their use is explained further in the section describing *Some Test Programs*.

Command Line Flags

The compiler will automatically run the assembler and linker stages with the parameters described above if you use the additional **-a** (run assembler) and **-l** (run linker) command line flags.

```
sccavr -al testing.c
```

This uses a canned set of parameters to compile the C code, run the assembler, and then run the linker twice. The first link produces a hex file ready to download, the second produces a file called debug.r90 ready for debugging in AVR Studio. It is the same as typing the commands

```
aa90 testing -r -L -v1
xlink crti testing libcavr -ca90 -Z(DATA)sdata=60 -Z(CODE)scode=0
      -Fintel-standard -o testing.hex -xsem -l map
xlink crti testing libcavr -r -ca90 -Z(DATA)sdata=60 -Z(CODE)scode=0
      -o debug
```

The compiler will include for **aa90** an **-I** parameter for each path in your **INCLUDE** environment, and an **-I** parameter for **xlink** for each path in your **LIB** environment. As you can see from the previous section, using the **-l** flag to force a link will only have meaning if your program is contained in one C file. Otherwise just use the **-a** flag for each module and link separately, embedding the commands in either a make or batch file. A batchfile example would be:

```
sccavr -can robot.c
sccavr -can motors.c
sccavr -can nav.c
xlink crti robot nav motors libcavr -ca90 -Z(DATA)sdata=60
      -Z(CODE)scode=0 -Fintel-standard -o robot.hex -xsem -l map
```

If you include the flag **-c** in the **sccavr** command line, the output assembly code will include the original C code as assembler comments. This can be helpful when reviewing or debugging at the assembler level. Use the **-n** flag to prevent including any *include* files in the assembler output listing.

By default the assembler **-v** flag is set to **-v1** (see the IAR document *AT90S Assembler, Linker, and Librarian*) You can enter any of the four possible values 0, 1, 2 or 3 in the command line and it will be passed verbatim to the assembler. For the mega103 for example:

```
sccavr -can -v3 robot.c
```

In addition the **-k** flag enables stack checking, the **-h** flag defines the address of the top of the hardware stack, and the **-s** flag defines the address for the top of the software stack. These are explained in detail below.

Some Helpful Hints

My recommendation for initial experimenting in AVR Studio is to compile with the command

```
sccavr -clan -h0xdf -s0xbf filename.c
```

By default both the assembler and linker are run with the **-r** flag to include debugging information in the default **debug.r90** output file, ready for loading into Studio. Ignore this file if you are just using the **.hex** file.

The **-h** flag value sets the AVR stack pointer to 0xdf, the top of sram for a 2313 processor. The **-s** flag value defines the top of the software stack, with the difference here defining a hardware stack size of 0xdf - 0xbf = 0x20 or 32 bytes. What really happens is the **h** and **s** values become the **HARDSTK** and **SOFTSTK** values respectively. The compiler outputs these as public constants in the module containing the *main* function. The startup code (crti) loads the **HARDSTK** value into the processor hardware stack pointer (SPH and SPL), and the **SOFTSTK** value is loaded into the r29-28 (Y) registers as the top of the C software stack. In this example the **h-s** value defines the space available for subroutine, interrupt and push operations. It is your responsibility to ensure this is a large enough space for any particular program.

In the case above, the small sram space of the 2313 usually allows you to have on screen in the Studio debugger a Register window, a Processor window, a Memory window, and the Program window. This set of windows allows you to see all machine states without using slide bars, etc.

The default for sccavr is the ram space of the 8515 processor.

When the AVR Studio debugger runs, it checks it can find the source code (s90) for each module in the executable (usually debug.d90). This means that for each module your program has included from the libcavr.r90 library, Studio will ask for a path to the source file (cruntime.s90). This quickly becomes very dreary, so the simple solution is to copy the cruntime.s90 file into your working directory if you will be using AVR Studio.

The compiler stores numerical values with the most significant bytes in lower memory addresses. The reason for this was simply to allow easy interpretation when examining memory in a debugger. When stored in cpu registers, the most significant byte will be in the higher numbered registers.

Installing the Software

All the software is available for download from www.jennaron.com/smallc.html. This includes the runtime libraries, include files, and the code examples discussed later. For convenience the IAR tools and AVR Studio debugger are also available. The sccavr compiler downloaded from this site is restricted in the number of global and local variables that can be defined. The fully featured version is available for purchase.

Installing SCCAVR

1. The compiler as downloaded is a zipped format file.
2. Unzip the `sccavr.zip` file into a new directory called `sccavr`. Either specify this while running a program like WinZip95 or create the directory beforehand. This will unpack the SmallC files into the `\sccavr` directory.
3. Now for some text editing. You must add the following lines at the end of your `\autoexec.bat` file in the root of your boot disk. Use any plain text editor. These lines must be *after* any other lines where these variables are defined.

```
set path=%path%;c:\sccavr;c:\iar\exe
set include=%include%;c:\sccavr;c:\iar\aa90
set lib=%lib%;c:\sccavr
```

This appends to your existing environment variables the additional paths required by the SmallC tools, including the IAR tools (see later).

4. If you intend using long file names (and who doesn't), the C runtime library used by sccavr requires an additional environment variable. Also add the following line to your `\autoexec.bat` file:

```
set lfn=128
```

Installing the IAR Assembler

1. Download the IAR Assembler tools `iarasm13.zip` file into a temporary directory.
2. Unzip the file using a tool like WinZip95 or `unzip`.
3. Click on the now unpacked `install.exe` file to run the DOS based install utility. Accept all the defaults. This will install the IAR tools in a directory called `\iar`. Note the intent of the reference to the `autoexec.iar` file has been completed during sccavr installation above.
4. Make sure you copy the two pdf files from the temporary directory to the `\iar` directory. These are the manuals for the IAR tools and very informative.

Installing AVR Studio

1. Download the **astudio2.exe** file into a temporary directory.
2. Click on this file to run the DOS based unpacking program.
3. Click on the **setup.exe** program to run the installation tool. Accept all the defaults.

When the installations are complete, reboot your machine so the changes to the environment take effect. You are now ready to use sccavr.

The C Implementation

The SmallC compiler for AVR adds to standard SmallC the following features:

- the **long** data type
- structures and typedef structures
- casts
- **unsigned chars** and **ints**
- stronger typing
- function definitions
- the **register** storage qualifier
- support for the AVR Harvard architecture
- global variable initialisation

The optimisations in Release 2 of the compiler address many of the stack based issues associated with SmallC. However, the stack paradigm still applies to arithmetic and automatic variables. For code space efficiency you should use global or register variables wherever possible.

SmallC for AVR does not support:

- multi-dimensional arrays
- the float and double data types
- the const data qualifier (ignored)
- enumerated types

Floating Point

The 32-bit data type **float** should be fairly easy to implement as the symbol table now allows setting 4 byte variables. As I need transcendental functions in the near future, and have a number of algorithms ready, this will happen fairly quickly.

Structures

Both global and local structures are implemented. Arrays of structures are allowed, as are arrays within structures and pointers to structures. Other structures and structure pointers are currently not allowed within structures. The syntax for structures is standard C. You can define a structure prototype like:

```
struct t_entry {
    int x, y;
    char note[6];
    int *pntx;
};
```

This produces no code, just a template that may be used subsequently by declarations like:

```
struct t_entry ThisOne;
struct t_entry *eptr;
```

You could also use **typedef** to define the structure and its implementations:

```
typedef struct t_entry {
    int x, y;
    char note[6];
    int *pntx;
} ENTRY;

ENTRY ThisOne;
ENTRY *eptr;
```

Note that a function definition like this must be global to allow passing function pointers between functions. You can define a structure prototype and create versions of that prototype in the same declaration. A structure pointer would be declared as for the first case above and can be used with the standard C syntax

```
eptr = ThisOne;
...
eptr->x = 1256;
```

Structure pointers can be incremented and decremented. However in this implementation it is probably easier to use structure arrays.

If you put the structure definition in a global include file it will be available to all C modules in your program. You can instance this template both globally and locally. You can of course, define a structure without a template using the standard C syntax. For example, to define two structures, both with the same internal structure and one an array, you could write:

```
struct {
    int x,y;
    int mode;
} A_Set, B_Set[6];
```

For the moment structures within structures are not allowed. The possible future addition of a *malloc* function would allow dynamic structures like lists, and will require structure pointers within structures.

Global Initialisation

Strings are not implemented in C as a type, but by convention are represented as a null terminated character array. Currently there are no library functions to support strings (strcpy, strcat, etc), but these are easily added (see *The Runtime Library*). You can declare global strings in two ways. Global definitions for a string constant use the syntax:

```
#define pumpWarn "Warning: Pump #231"
```

The compiler would assign a label to this, add a trailing null, and store it in AVR program memory (in the scode segment) using the declaration:

```
L0      db      87,97,114,110,105,110,103,58
        db      32,80,117,109,112,32,35,50,51,49,0
```

Note the compiler has added the trailing null. Subsequent string constants declared in the same way are appended to this `db` structure, separated by the nulls. The addressing is handled from within the compiler. You could also declare this same string as:

```
char pumpWarn = "Warning: Pump #231";
```

This would also be stored in program memory but would use the defined label for access and would declare the character array as a string:

```
pumpWarn db      "Warning: Pump #231"
```

Note that here the assembler automatically adds a trailing null to the string - one is not supplied by the compiler. Unsigned chars can be defined using the syntax:

```
unsigned char keys = { 1,2,0x3c, 4, 56,2 };
```

The types `int`, `unsigned int` and `long` can all be declared using similar syntax:

```
int CrossData = { 12, 234, 876, 237, 345, 1211,
                 980, 30, 356, 1587, 66 };
```

Note that while no action is taken if the `const` keyword is used (it is simply skipped), variables declared as above are very much constant and read only, as they are written into the eeprom along with the executable code!

The changes I have made allow access to this data via the `lpm` instruction for accessing constant data in program memory. You need not worry about this of course and can subsequently treat the declaration like any other global variable:

```
char *pt;
pt = pumpWarn;
```

I have not yet found a low cost way to flag the fact that a pointer passed between functions may point to either data or program memory at run time. However the compiler keeps track of this difference within the function in which the pointer is assigned. **Thus to use a pointer that you have pointed at a constant in program memory you must use it within that same function.** In all other cases the pointer is assumed to point to data memory. To simplify handling this

special case, a library function called `romcpy` exists for moving a string constant out of prom into ram for subsequent processing. Using the constant defined above as an example, the function will copy the string from prom to a ram based `char` array, eg

```
char RamTextString[20];
...
romcpy(RamTextString, pumpWarn);
```

You can define the string within the function if required:

```
romcpy(RamTextString, "Closing door");
```

Remember to include the `avrstdio.h` header file. String constants are collected by the compiler and output as a block at the end of `scode` space.

Functions

Functions require prototypes. If the compiler encounters a function without a previous definition it assigns a return type of `int` and assumes there will be no parameters. If there are parameters or it is subsequently used as another type the use is flagged as an error. It is good programming practice to use function prototypes, either as a list at the start of a single file program, or in an include file for multi-file programs. In the latter case use the include file for global function prototypes, and in each program the `static` qualifier for the list of local function prototypes that are only used in that file. This is C's small contribution to data hiding.

A function prototype should type both the return value and any parameters. Named parameters are optional in declarations, ie both these declarations are equivalent:

```
long Accumulate(long *sum, int count);
long Accumulate(long*, int);
```

The syntax of a function declaration has been changed to the more modern form:

```
int GetTime(int (*timer)(), int offset) {
    <body of function>
}
```

Interrupt Functions

I have added the `interrupt` qualifier to function names:

```
interrupt clock0(void);
```

The function cannot have a type and it makes no sense to include function parameters. You will need to add this function name at the correct place in the startup file `crti.s90`, either as a vector in the interrupt table, or via a jump from the interrupt handler. An interrupt routine can have local variables and can address global variables. However a C interrupt routine is an expensive device, as the entire

C register set is pushed at the beginning and popped at the end for a total of around 60 additional instructions, or about 16µS with a 4Mhz crystal.

I have added two pseudo C instructions `gintoff` and `ginton` for issuing the AVR global interrupt control instructions `cli` and `sei` respectively.

The *register* Storage Qualifier

The compiler can use the fourteen registers r2 through r15 for local variable storage. All supported data types are allowed. This offers a considerable speed advantage over local variables on the stack, particularly for variables that are accessed repeatedly like loop counters. The only disadvantage is that they are pushed onto the hardware stack whenever the function in which they are defined calls another function. This presents an additional hardware stack size requirement, particularly if you are using recursive functions.

Note: You should always carefully consider the hardware and software stack size requirements in your design.

The *long* Data Type

The 32-bit signed **long** data type is available for all operations. Long constants use the "L" affix (123456L). Data typing is strictly enforced and you must use casts to convert between types. In some cases variables are promoted and demoted based on context, but it is always wise to check. If in doubt, cast.

Unless otherwise specified, all variables are signed, so casts to a larger data type automatically do sign extend.

```
char a;  
long b;  
a = -5;  
b = (long)a;
```

Thus **b** will have the value -5L.

Arithmetic

The language supports signed and unsigned add, subtract, multiply and divide on chars, integers and longs. There is no automatic error handling. For situations where you are uncertain of the range of variables, four C library functions allow your program to dig itself out of arithmetic errors.

`Int CheckDivZero()`

Call immediately after a division. Returns zero for no error.

Int CheckLongOverflow()	Call immediately after a multiply operation using longs . Returns non-zero if the result exceeded 32 bits.
Long GetLongOverflow()	If the CheckLongOverflow function returns true, this function will return the upper 32-bits of the 64-bit product. You could thus define an array: long value[2] and implement a subset of 64-bit integer arithmetic.
int CheckIntOverflow()	Call immediately after a multiply operation using ints . Returns non-zero if the result exceeded 16 bits. Just do it again after casting to longs.

The 8-bit variables **char** and **unsigned char** are always promoted to 16-bits for arithmetic, including the arithmetic in comparisons.

Input Output

Port input and output is one of the main functions of microcontrollers such as the AVRs. I have implemented the IO standard **in**, **out**, **sbi** and **cbi** functions as **inp**, **outp**, **sbip** and **cbip** respectively. The port addresses and function definitions are defined in the avrstdio.h header file. Although their implementation is of necessity indirect, their use in C is close to the hardware and is somewhat easier to remember than a specific function per IO function. Typical use is

```
outp(DDRB, 0xff);
outp(PORTB, 0xc7);
sbip(PORTB,1);
val = inp(PORTB);
```

The function definitions are:

```
char inp(int addr);
void outp(int addr, int val);
void sbip(int addr, int bit);
void cbip(int addr, int bit);
```

The addresses associated with the mnemonics are the IO addresses. The conversion to a memory mapped address is done at run time. Thus you can mix assembler with C code when speed is necessary without address worries and the compiler will manage the offsets, ie:

```
code = 0x05;
outp(PORTB, code); // send command
#asm
    ldi    r31,0x01 ; motion stop command
    in     r30,PINB ; wait for bit 7 set
    sbrs  r30,7    ; skip when it is
    rjmp  $-4     ; otherwise loop
    out   PORTC,r31 ; must stop quickly
#endasm
val = inp(PINB); // get input byte
```

You may need to ensure the correct header definition file for the processor you are using is included for the assembler. Use the compiler `-I` flag for this:

```
sccavr -clan -Iio8515.h myprogram.c
```

This will insert an include line for this file in the assembly output.

Code Efficiency

SmallC writes the output assembler file as it processes each C token. It therefore has no memory of what has been done and thus cannot review its representation at the end of the line or function. SmallC originally used the *primary register* concept, which was fine in the days where registers were scarce but hardly applies to processors like the AVR series. In this version I have incorporated some look ahead to take sccavr out of the simple stack machine class. It also freely uses the index registers for variable management. However sccavr still uses the stack paradigm for automatic stack variables and all simple arithmetic. For most control applications the AVR processors are fast enough that code efficiencies are not a problem. When they are I simply move to assembler. In this section I will show three examples of how sccavr manages a simple integer assignment, where the variables are stack, global and register based.

The first example shows a function with two local integer variables and an assignment between them.

```
test {
    int a, b;
    ...
    a = b;
}
```

The resulting assembler code with my comments is:

```
;test() {
test
; int a, b;
;
; a = b;
    sbiw r28,4           ; create a local stack frame
    mov  r26,r28         ; stack pointer to X index
    mov  r27,r29
    adiw r26,2           ; offset to a
    st  -y,r26           ; store address on stack
    st  -y,r27
    mov  r26,r28         ; stack pointer to X index
    mov  r27,r29
    adiw r26,2           ; offset to b
    ld  r31,x+           ; get b value into Z
    ld  r30,x+
    ld  r27,y+           ; get a address into X
    ld  r26,y+
    st  x+,r31           ; store b value into a address
    st  x+,r30
;}
    adiw r28,4           ; clear stack frame
ret
```

This is the most expensive method and you can see how the code is built up by scanning from left to right in the line. Now the same function but with global variables:

```
int a, b;
...
test {
    ...
    a = b;
}
```

This uses the AVR direct access instructions `lds` and `sts` for a considerable saving in code space.

```
;test() {
test
;
; a = b;
lds r27,_b           ; get b contents into X
lds r26,_b+1
sts _a,r27           ; store X into a
sts _a+1,r26
;}
ret
```

This is about as efficient as it gets. When used in a module and flagged with the `static` qualifier, the main problem with global variables (ie name conflicts and that anyone can alter them) tend to go away, as this makes them invisible from outside the module. So this can sometimes be a good solution although recursion is not possible and internal module name conflicts are still a problem. The last example uses the `register` qualifier on local variables.

```
test {
    register int a, b;
    ...
    a = b;
}
```

And this produces the quite efficient code:

```
;test() {
test
; register int a, b;
;
; a = b;
mov r31,r13         ; b to Z
mov r30,r12
mov r15,r31         ; Z to a
mov r14,r30
;}
ret
```

There are 14 bytes of register storage available to every function (r15-r2). However, their use comes with the time and space penalty associated with pushing and popping the relevant registers when the owning function calls another.

For me (and I'm sure for others) the main advantage of C in the embedded world is that the algorithms get written quickly, without worrying about the minutiae of data movement associated with programming in assembler. In nearly every case I have found the C implementation is all I need, and that is the way the product gets shipped. Only occasionally, typically when handling data streams originating from

or destined for external peripherals, was it necessary to move to assembler in the interests of speed.

The Runtime Initialization File

SmallC is intended for those people who are quite happy working with assembly code but just want the convenience of writing the majority of processing code in C. Thus the startup code is external to the compiler and can be modified and elaborated on as required to suit each new project. The compiler is shipped with a number of processor specific startup files as demonstrations. The `xxxx_1.s90` versions assume that interrupts will not be used. The version for the 8515 processor is `i8515_1.s90`:

```
; startup.s90 for no interrupts AVR micro 8515.

        col      120
        lstexp-           ; no macro listing

#include <avr.inc>
#include <io8515.h>

; CODE SEGMENT
        name      startup
        rseg      scode
        extern    SOFTSTK, HARDSTK
        extern    _main

        rjmp      _prep

_prep   ldiz      HARDSTK
        out       SPH,r31
        out       SPL,r30
        rcall     _rmclr
        ldiy      SOFTSTK+1
        rcall     _main
        rjmp      $

_rmclr  clr       r27
        sbiw      r30,1
_rc11   st        -z,r27
        cpi       r30,0x60
        brne     _rc11
        tst       r31
        brne     _rc11
        ret

        end
```

By convention `sccavr` prefixes the names of all functions, globals and compiler related and library functions with an underscore. The main reason for this is to prevent the assembler confusing a variable like `x` with the register `x`.

Note: The startup file eventually jumps to a label called `_main`. Thus somewhere in the files that make up your application you must use `main` as the name of your initialisation function.

`Sccavr` uses a two stack paradigm for the C code, where interrupts, calls, pushes and pops use the hardware stack, and C code uses the Y register pair (r29-28) as a

more code efficient software stack pointer. Both stack pointers decrement down into memory for pushes, and increment for pops. It is worth noting that the software stack pointer (Y) will always be pointing at the last byte pushed – ie, it decrements before pushing.

By default the software and hardware stacks are initialised to 575 (0x23F) and 607 (0x25F) respectively (the 8515). The difference allows for a hardware stack depth of 32 bytes or a function call depth of 16 before the hardware stack starts over-writing the software stack.

Note: This default is not enough room if somewhere in your code you use the *interrupt* qualifier for a C based interrupt function.

All interrupt stack pushes and programmatic pushes also go on this stack. The default values can be easily changed in the compiler command line. For a 2313 for example, using a hardware stack call depth of only 8 (16 bytes), the command would be:

```
sccavr -h223 -s207 testing.c
```

This sets the hardware stack top to 0xdf and the software stack top to 0xcf. Note you can use either hex or decimal notation.

There is no reason why the software stack should be lower in memory than the hardware stack. For example if you are using external ram you could keep the hardware stack constrained to the bottom section of the internal sram and leave the software stack at the top of external ram, allowing lots of room for recursive functions and saving cycles on hardware stack access. As an example, suppose you had 32K of external ram connected to an 8515, and you wanted 128 bytes of hardware stack at the bottom of internal ram, and the C stack at the top of the external ram. To set the hardware stack top at 0xdf (223) in the internal sram, and the C software stack to the top of external ram, the command would be:

```
sccavr -h223 -s0x7fff test.c
```

Remember you would need to reset the xlink parameter **-Z(DATA)sdata=** to the next byte after the hardware stack top so global data starts there rather than at the bottom of sram. In this example it would be **-Z(DATA)sdata=e0** (ie 224). You would also need to modify the global memory clear routine in the startup code. One of the example programs demonstrates a similar configuration.

You need not worry if your global data definitions spill over into the external ram. All that will happen is access to these variables will take one extra cycle. As global memory is assigned by the linker on a first come first served basis, you can assign highly accessed global data early, perhaps in a special data.c file.

A more elaborate startup file where interrupts are used could include the initialisation code initially described by David VanHorn. Here all interrupt vectors are included along with the individual handlers, all bogus reset generated interrupts harmlessly cleared, and the processor state completely set up before calling the C main function. The examples have the name format **xxxx_2.s90** with the version for the 8515 called **i8515_2.s90**. This starts off like:

```

; Interrupt enabled startup code for sccavr.
; David VanHorn & Ron Kreymborg

#include <avr.inc>
#include <avrmacro.inc>

        name      startup
        rseg      scode           ; must be first code segment
        extern    SOFTSTK         ; defined in compiler
        extern    HARDSTK
        extern    _main           ; entry point to user's code

        rjmp     _start          ; reset jump

; Interrupt vectors
rjmp     ext_int0
rjmp     ext_int1
rjmp     tim1_capt
rjmp     tim1_compa
rjmp     tim1_compb
rjmp     tim1_ovf
rjmp     tim0_ovf
rjmp     spi_handler
rjmp     uart_rxc
rjmp     uart_dre
rjmp     uart_txc
rjmp     ana_comp

;*****
; Startup routine begins

_start  ldiz     HARDSTK           ; init hardware stack
        out     0x3e,r31
        out     0x3d,r30
        rcall   _init             ; go clear all interrupts, setup the
                                   ; ports, start the timers, all that
                                   ; hardware stuff
        rcall   _ramclear         ; global data area must be zeroed
        ldiy   SOFTSTK+1         ; init the C runtime stack
        sei
        rjmp   _main             ; and hello world

```

If you use interrupts, register r16 has been set aside as the only register not used by the C runtime environment. Use this to save and restore the flags register (SREG) on entry and exit. Many interrupt routines can be serviced with just this register. Other registers can be used after first pushing them to the hardware stack.

Note: If you use the `-l` option in sccavr to run xlink, it assumes the startup file is called `crti.r90`. Thus you must be sure to name your startup file `crti.s90`, or else copy one of the examples from the `\sccavr` directory, and then run the batch file called `makeinit.bat`. Read the comments in this batch file concerning the `aa90 -v` flag.

You can have any number of startup files, one for each project if required, all with different names. You just need to change the xlink line in your make file. A much simpler technique is to use a separate directory for each project, a system I highly recommend.

Monitoring Stack Usage

Stack overflow is an ever present problem in embedded systems. Stack usage is always difficult to estimate and there is little that can be done in the way of error recovery when it occurs. To guard against the situation occurring, the `sccavr` compiler provides a flag that if set will provide code to check both stacks prior to and immediately after calling every function (except the library `outp` function). While this does not cover events that obliterate the return address on the hardware stack or events which overwrite data neighbours by a few bytes, it will catch most other events. It works by writing a two byte integer sentinel (0x5555) into user specified addresses at the bottom of the software and hardware stacks. While these bytes continue to match normal processing proceeds. Otherwise one of two possible outputs on one of four possible output ports will be driven low and the processor will stop in a loop. If connected to LEDs, these port outputs can be used to indicate a hardware or software stack overflow respectively.

You enable stack checking with the `-k` command line flag

```
sccavr -clank myprogram.c
```

This flag works with a new file you must create (or add to) in the `/sccavr` directory called `sccavr.ini` that defines a number of stack checking parameters. These define the addresses to write the sentinel words to in memory and the port definitions for overflow indication. You must always define both stack limits. The three associated variables are called `hardstklimit`, `softstklimit` and `stkerr`. Typical entries for an 8515 processor using the default `sccavr` stack settings would be:

```
softstklimit=0x212
hardstklimit = 0x242
stkerr=portb, 3,0
```

Blank lines and spaces are ignored. Recapping, the `sccavr` default stack settings give a hardware stack top of 0x25f and a software stack top of 0x23f. The hardware stack limit setting above is therefore three bytes above the top of the software stack. At startup this would look like:

```
00
55
55 <- sentinel address (0x242)
00
00
00 <- Y register address (0x23f)
00
```

The address for these sentinels will depend on your application. Perhaps in this case an address of 0x240 would be adequate. If you are particularly worried you might put it sixteen or so bytes above the top of the software stack. The sentinel for the software stack is constrained to be at least above the top of the globally defined variable area.

The `stkerr` entry defines the port and bit positions that will be used to indicate an overflow. In the above example, bit position 3 of PORTB will be used for hardware

stack indication, and bit position 0 for the software stack. In both cases the code that is executed is:

```

in      r20,<port>
cbr     r20,<respective_bit_position>
out     <port>,r20
rjmp    $

```

For a hardware stack overflow in the case defined above, the actual code would be:

```

in      r20,PORTB           ; get PORTB levels
cbr     r20,4               ; 1<<3
out     PORTB,r20          ; light the LED
rjmp    $                  ; error

```

Note this means the other bit positions on the port are unaffected, providing a minimum impact on other peripherals connected to this port when an error occurs.

The check functions are inserted just before the `main` function. The code to insert the sentinels is inserted immediately after the `main` label and before your code is started. Before calling any functions (apart from `output`) you should configure the port for the two chosen output positions.

There is a speed penalty of course, but if your application is not affected by this, there is nothing wrong with leaving the code in production equipment, with the two LEDs mounted on the board and used for system debugging when and if an error occurs.

The Runtime Library

The C runtime library is supplied as both a source file and a ready to use IAR format *library* file. The files are `cruntime.s90` and `libcavr.r90` respectively. I recommend that you create your own specialist libraries for your own functions, rather than adding to `libcavr`, and that you precede the library name with the letters *lib*. However, if the new function is a C standard it would make sense to add it to the standard library. Remember to add the function definition to the `avrstdio.h` file.

As an example we will add the assembly code for the standard `strcpy` function to the library to show the steps involved. The C code for `strcpy` is:

```

void strcpy(char *dest, char *source) {
    while ((*dest++ = *source++) != '\0');
}

```

SmallC pushes arguments from left to right, so the `source` pointer will be on the top of the stack with `dest` below it. Or viewed as it will be on the actual decrementing stack:



Anything at a lower stack address is unknown. We need to get the two pointers into the two indexing registers X and Z. Remember that the Y register is the C stack

pointer – you can use it if you must but always restore it before returning. Using the AVR indexed with offset instructions makes unloading the pointers easy:

```

strcpy  ldd    r31,y+0      ; MSB of source
        ldd    r30,y+1      ; LSB of source
        ldd    r0,y+2       ; MSB of dest
        ldd    r26,y+3      ; LSB of dest
        mov    r27,r0       ; dest in X

```

The copy loop and exit is straightforward. The calling routine does the stack fixup so there is nothing to do prior to returning. I use the convention that labels internal to a library function are preceded by an underscore.

```

_strcpy1 ld    r0,z+       ; get next char
        st    x+,r0       ; store in dest
        tst   r0          ; was it null?
        brne  _strcpy1    ; .. no, loop
        ret                   ; ..yes, done

```

Once you have tested this as a stand alone assembly function, you simply wrap it in an IAR *module definition* and edit it into the cruntime.s90 code. The complete definition looks like:

```

        module  strcpy
        rseg    scode
        public  _strcpy
_strcpy  ldd    r31,y+0      ; MSB of source
        ldd    r30,y+1      ; LSB of source
        ldd    r0,y+2       ; MSB of dest
        ldd    r26,y+3      ; LSB of dest
        mov    r27,r0       ; dest in X
_strcpy1 ld    r0,z+       ; get next char
        st    x+,r0       ; store in dest
        tst   r0          ; was it null?
        brne  _strcpy1    ; .. no, loop
        ret                   ; ..yes, done
        endmod

```

The module name need not be the same as the entry label, the code segment is always **scode**, and the entry label needs to be declared as public. Note the function name must be prefixed by an underscore because sccavr will prefix the **strcpy** name it finds in your C code.

Run the batch file called makelib.bat to re-create the libcavr.r90 library file now including your new function and you are ready to go. This batch file looks like:

```

@echo off
if exist libcavr.r90 del libcavr.r90
aa90 cruntime -r -v1 > t
if errorlevel 1 goto bad
xlib < makelib.ins >> t
if errorlevel 1 goto bad
del cruntime.r90
del t
goto end
:bad
echo ***** At least one error occurred
:end

```

The input command file makelib.ins for the xlib librarian is:

```

define-cpu a90
fetch-modules cruntime libcavr
quit

```

The **define-cpu** defines the AVR processors, and **fetch-modules** defines the input and output files respectively (see the IAR *AT90S Assembler, Linker and Librarian* manual). You could copy and edit these build files to suit your own libraries.

Some Test Programs

A number of sample programs are available to download from www.jennaron.com/smallc.html. These range from simple to quite complex. Each one demonstrates how you can use C to manage the considerable resources available in the AVR range of microprocessors.

I strongly suggest you unpack each example into separate directories. If you use just one directory, it can get very frustrating trying to remember which startup file belongs to which example.

A Simple LED Flasher

The first one we will use is called **chase.c**. This program sequentially and repeatedly moves a low level from bit 7 to bit 0 and back again on PORTB. If you have the STK200 development board this port has eight LEDs connected to this port. The program listing is:

```

// Demonstration program for sccavr: chase.c
// Continously moves a low from bit 7 to bit zero
// and back again on the output of PORTB. Turns on the
// LEDS When used with the STK200 development board.
//
// Ron Kreymborg

#include <avrstdio.h>

int flip;

main() {
    int i;
    register unsigned int led;

    outp(DDRB,0xff); // set PORTB for all outputs
    led = 0xff7f; // initial pattern (bit 7 low)
    flip = 1; // used to signal direction
    while (1) { // do forever
        outp(PORTB, led); // output the pattern
        if (flip) {
            led = led >> 1; // right shift the low
            if (led == 0x1fe) // reverse if at the end
                flip = !flip;
        }
        else {
            led = (led << 1) + 1; // left shift and feed in 1s
            if (led == 0xff7f) // reverse if at the end
                flip = !flip;
        }
    }
}

```

```

        }
        for (i=0; i<9000; i++)      // leave it there for a while
        ;
    }
}

```

This is a good example as it contains the three storage types, global, automatic and register. Make sure you have installed the compiler and adjusted the path variables as detailed in *Installing the Software*. I will assume throughout these example you are using an 8515 processor. Copy the no-interrupts startup file from \sccavr and re-assemble:

```

copy \sccavr\i8515_1.s90 crt.i.s90
makeinit

```

This produces the relocatable file `crt.i.r90` ready for linking in your working directory. If you do not have a local copy the linker will look for it in the \sccavr directory, but it is usual to use a local copy in your work directory for project based modification.

If you make any changes to the C library, regenerate the relocatable version (in the \sccavr directory) with the command:

```
makelib
```

To try out this example program, copy the `chase.c` file to a working directory and give the command:

```
sccavr -clan chase.c
```

This command generates an assembler file with the C code embedded and with stacks set for an 8515. You will need to use the `-s` and `-h` flags if the destination is a different micro. The automatic `xlink` command creates an Intel-standard hex file called `chase.hex` ready for downloading into the 8515. It also creates a file called `map.lst` which is a listing of where the code, data and any library modules will be located in memory.

If you want to step through the program and watch how it works in AVR Studio, you can use the `debug.d90` output file.

You should examine the `chase.s90` file produced by `sccavr` to see and understand how it converts the C code to AVR assembler. Note the simplicity of how register and global variables are accessed. Compare this with how the integer variable `i` is handled on the software stack. You might like to go back later and make this a register variable too and re-compile to see the difference.

Now run the Atmel ISP program supplied with the STK200. If you are not using this hardware, use whatever software you normally use to download the hex file `chase.hex` to your processor. You should see the LEDs immediately begin flashing.

If you have them, try the same program with a 2313 in the STK200 and with a mega103 in the STK300.

External Memory on an 8515

The STK200 has the tracks and sockets ready to mount 32Kbytes of external memory to an 8515 AVR. Once this is installed and the 8515 configured, internal ram still runs from 0x0060 to 0x025f, but external ram is contiguous with it from 0x0260 to 0x7fff. One point to note is that the 8515 adds an additional clock cycle when accessing external memory. Note also that ports A and C are sacrificed to managing address and data when configured for external ram.

The program extram.c is really a memory tester. It defines a global **unsigned char** array variable called **data** with a size of 32,000. Once initialisation is complete, this array is filled from start to finish with an incrementing pattern, and then read to check the value written is correct. The current pattern is displayed on the STK200 LEDs during both cycles. After a successful check LED 7 is blinked and the pattern incremented. If the check fails the processor is stopped with all LEDs on. This example shows how the stack configuration can be changed to suit the application. We will again use the no-interrupts startup file but must modify the global data clear routine required by the C specification to now cover the external ram. We must also switch the 8515 into external ram mode *before* accessing external ram, and for C code this means *before* calling the **main** function, as global data must be cleared to zero. The changes to the 8515 no-interrupt startup file **i8515_1.s90** are

```

        rjmp     _prep
_prep   ldiz     HARDSTK
        out      SPH,r31
        out      SPL,r30
        ldi      r20,0x80
        out      MCUCR,r20           ; external ram enable
        rcall    _rmclr
        ldiy     SOFTSTK+1
        rcall    _main
        rjmp     $

_rmclr  clr      r27
        ldiz     SOFTSTK
_rcll   st       -z,r27
        cpi      r30,low(HARDSTK+1)
        brne    _rcll
        cpi      r31,high(HARDSTK+1)
        brne    _rcll
        ret
```

Note the **out** instruction to the MCUCR register, setting the 8515 into external ram mode. Note also the changes to the ram clear routine. In this example the compile and link commands are

```
sccavr -can -s0x7fff -h0xbf extram.c
xlink crti extram libcavr -ca90 -Z(DATA)sdata=c0 -Z(CODE)scode=0
        -Fintel-standard -o extram.hex -xsem -l map
```

The command line changes are:

-s0x7fff	Sets the software stack to the top of external ram.
-h0xbf	Sets the top of the hardware stack.

`-Z(DATA)sdata=c0` Sets C global memory to start immediately after the hardware stack.

After you have copied the `i8515_1.s90` startup file to `crti.s90` and edited it as above, prepare it for compiling with the command:

`makeinit`

The program listing for `extram.c` is quite simple. After initialising `PORTB` for output it begins a forever loop in which the current value is inverted and output to `PORTB` (remember a *low* lights a LED). It then fills the global array with the current value, then attempts to read it back. If something has changed the program will halt with all LEDs on. If all is well the program blinks LED 7 then increments the test value and does it all again.

```
// Demonstration program for sccavr: extram.c
// Configures an STK200 fitted with the external ram
// components. Writes a incrementing number to an array
// of bytes, then reads it back checking for errors.
// Use a modified crt1_1.s90.
//
// Ron Kreymborg

#include <avrstdio.h>

#define BYTE unsigned char
#define SIZE 32000

void main(void);
void blink(void);
void delay(void);

BYTE data[SIZE];

void main(void) {
    register BYTE value;
    register int i;

    outp(DDRB, 0xff); // PORTB is output
    value = 1;
    while (1) { // do forever
        outp(PORTB, ~value); // show value
        for (i=0; i<SIZE; i++) // fill data array
            data[i] = value;
        for (i=0; i<SIZE; i++) {
            if (data[i] != value) { // check all is ok
                outp(PORTB, 0); // turn all on
                while (1) // and hang if not
                    ;
            }
        }
        value++; // step to next value
        blink(); // blink LED 7
    }
}

// Blink LED 7 on PORTB for a few mSecs.

void blink(void) {
    register BYTE b;

    b = inp(PORTB); // get current value
    outp(PORTB, b | 0x80); // ensure 7 off
    delay();
    outp(PORTB, b & 0x7f); // turn it on
}
```

```

    delay();
    outp(PORTB, b);                // restore whatever
}

void delay(void) {
    int i;
    for (i=0; i<4000; i++)        // about 70ms @ 4MHZ
        ;
}

```

Now compile this example. As ever, you should make sure there is enough room for the software stack at the end of global memory. If you are unsure or think it will be close, look at the *Segments in Address Order* entry in the listing file produced by xlink (map.lst). It lists exactly the extent of both the scode and sdata segments in memory.

Download the hex file to the STK200 using the ISP program. You should see the LED display slowly count up in binary. The 7 LED should blink briefly at the end of each cycle. You might like to experiment with choosing a random number from one of the 8515 counters.

Driving a Two Line LCD Display

This is a good example in that it shows how large chunks of assembler can be written directly into a C module using the `#asm` key word. It also shows examples of how the additional C input/output instructions can be used. It assumes the LCD connected to the STK200 is a two line by sixteen-character display. The program function is fairly simple in that it writes a message to the top line and a `long` number to the bottom, then after a short delay, displays the number again incremented by one. It assumes a 4Mhz clock.

The program is not meant to show the most efficient way to drive the lcd display, but to highlight a number of sccavr features. While rather long I will list it here and then go into some detail.

```

/*****
Outputs messages to a 2-line LCD connected to
the STK200 - lcd.c
The board has the following connections hard wired
Port A          - data
PORTD R/W bit 6 - low write, high read
PORTC RS bit 6 - low command, high data
      E bit 7 - high strobe
This example uses a 4 data wire interface. Note
you will need to include the IAR ioXXXX.h file in the
compile line for the particular chip you are using,
as the constants in #asm blocks are handled by the
aa90 assembler, not by the compiler. ie
    sccavr -clan -Iio8515.h lcd.c
Use crti_1.s90.

Ron Kreymborg
*****/

#include <avrstdio.h>

#define UNSIGNED unsigned int
#define BYTE      unsigned char

```

```

void main(void);
void toascii(long n, char *p);
void send(BYTE *p, UNSIGNED addr);
void init(void);
void LCDinit(void);
void Command(BYTE val);
void Data(BYTE val);
void strobe(void);
void Delay(int prescale, int count);
void Command_8(int val);
void wait(void);
void ChkBusy(void);
void SendC(BYTE val);

char *mess = "Stepping numbers:";

BYTE string[20];

void main(void) {
    int i;
    char *p1;
    long n;

    init(); // init everything
    p1 = mess;
    romcpy(string, p1); // put message 1 in ram
    send(string, 0x00); // send to line 1
    n = 0L; // number starts from zero
    while (1) {
        toascii(n++, string); // convert long to ascii
        send(string, 0x40); // pprint it
        for (i=0; i<10; i++) // leave there for a while
            Delay(5, 1);
    }
}

// A by-the-book method for producing decimal numbers
// from a C binary long. Enter with number in <n> and a
// pointer to the output char buffer in <p>.
void toascii(long n, char *p) {
    register long m, s;
    register int i, flag;

    i = 10; // places
    s = 1000000000L; //divisor
    flag = 1; // number found flag
    while (i--) {
        m = n / s; // get next msd
        *p = (char)(m + 48L); // convert to ascii
        if (flag && (*p == '0') && (i > 0)) // zap if leading zero
            *p = ' ';
        else
            flag = 0;
        p++; // next char
        n = n - m * s; // shift left
        s /= 10L; // reduce divisor
    }
    *p = '\0'; // add trailing null
}

// Send the string pointed to by <p> to the lcd
// line starting at <addr>.
send(BYTE *p, UNSIGNED addr) {
    while (*p) {
        Command(addr + 0x80);
        Data(*p++);
        addr++;
    }
}

// Init the SDK200 and the LCD.
void init(void) {

```

```

    outp(DDRA, 0xff);      // PORTA all outputs
    outp(PORTA, 0);       // data all low
    outp(DDRB, 0xff);     // PORTB all outputs
    outp(PORTB, 0xff);    // data all high
    outp(DDRC, 0xff);     // PORTC all outputs
    outp(PORTC, 0x3f);    // E low, RS low
    outp(DDRD, 0xff);     // PORTD all outputs
    outp(PORTD, 0xbf);    // R/W low
    outp(PORTC, 0x3f);    // RS low for instructions
    outp(PORTD, 0xbf);    // R/W to write
    Delay(5, 60);        // power on wait 100mS
    Command_8(0x30);      // 8-bit mode
    Delay(4, 178);       // 4200uS
    Command_8(0x30);      // 8-bit mode
    Delay(2, 181);       // 150uS
    Command_8(0x30);      // 8-bit mode
    Command_8(0x20);     // 4-bit mode
    Command(0x28);       // 4-bits, 2-lines, 5x7
    Command(0x08);       // display off, cursor off, blink off
    Command(0x0c);       // display on
    Command(0x06);       // increment, no display shift
}

void Command(BYTE val) {
    cbip(PORTC,6);       // RS pin to command (lo)
    SendC(val);
}

void Data(BYTE val) {
    sbip(PORTC,6);       // RS pin to data (hi)
    SendC(val);
}

void SendC(BYTE val) {
    Delay(1, 96);
    cbip(PORTD, 6);     // R/W to write (lo)
    outp(PORTA, val);   // send high nibble
    strobe();           // strobe E pin
    outp(PORTA, val<<4); // send low nibble
    strobe();
}

void strobe(void) {
    #asm
    sbi  PORTC,7         // pulse E strobe high
    cbi  PORTC,7         // port low again
    #endasm
}

void Delay(int prescale, int count) {
    #asm
    ldd  r20,y+3         // port value
    out  TCCR0,r20
    ldi  r20,2
    out  TIFR,r20
    ldd  r20,y+1         // count
    out  TCNT0,r20
lpl
    in  r20,TIFR
    andi r20,2          // isolate timer0 overflow
    breq lpl           // loop until set
    #endasm
}

// 8-bit command, assumes both R/W and RS are low.
void Command_8(int val) {
    Delay(1, 96);
    outp(PORTA, val);   // PORTA is data
    strobe();
}

Clear() {
    Command(0x01);     // clear display
}

```

```

    Delay(5, 197);          // 15 msec wait
}

```

The **main** routine uses the **romcpy** library function to copy the message to ram so it can be addressed in other functions. The long number is initialised to zero and subsequently incremented after passing it to the **toascii** function for conversion from a binary number to an ascii string. After configuring the ports, the **init** function follows the standard for initialisation of Hitachi and clone LCD controllers.

The **send** function precedes each character with its destination address. The command set requires address commands to have bit-7 set. Both the address and character pointers are then incremented and the results sent until the string terminating null occurs.

The **Command** and **Data** functions differ only in the setting of the RS pin, low for a command and high for data. Both then call the 4-bit send command function **sendC**. This sends first the high nibble then the low to the LCD, both followed by the E line strobe. The STK200 LCD control was meant to use the extended addressing capability of the 8515, and generates the E strobe using capacitive coupled logic. On my old Tektronix 453 the strobe pulse is a little over 300uSecs, whereas the specification states 450uSec minimum. If you experience trouble, try replacing the 33pF capacitor with a 64pF part. These three functions are written entirely in C and demonstrate satisfactory solutions where time is not critical.

The **Delay** function is just the opposite, it is written entirely in assembler. You can use all assembler instructions as well as C style comments. The compiler does nothing with these lines - they are passed on to the assembler verbatim. The compiler has no idea which processor you are using, so you will need to include a reference to the respective IAR definition file in the **sccavr** command line. Eg for the 8515:

```
sccavr -clan -Iio8515.h lcd.c
```

The function itself demonstrates a method of using timer0 in a non-interrupt environment. You send it pre-defined prescale and count values which are loaded into the counter. The terminal count bit TOV0 is cleared in the TIFR register and a loop is entered waiting for this bit to be set. The formula for deciding on a count value is

$$count = 256 - \frac{uSecs \times Mhz}{divider}$$

Where *divider* is from the set 1, 8, 64, 256, 1024 (See *Clock0 Prescale Select* in the respective processor manual). Results less than zero mean go to a larger divider, while results close to 255 mean go to a smaller divider. If results with two or more dividers are within this range, go for the smaller result, as this will minimise round-off errors. For example, if the desired delay is 1000uSec and with a clock of 4Mhz, three solutions are within this range:

Divider	Count	Usec/tick	Ticks	delay
---------	-------	-----------	-------	-------

64	193.5	16	62	992
256	240.375	64	16	1024
1024	252.094	256	4	1024

A divider of 64 and register load value of 194 (ie the smallest result) give the closest solution.

Using a Multi-processing Task Dispatcher

I wrote an article in the August '99 DDJ describing a multi-task dispatcher for small embedded systems. The example used here uses a modified version (no status) with a main program consisting of a number of independent tasks that are responsible for flashing various LEDs and monitoring buttons on the STK200 board. While the example is not very useful in itself, it is a good example of a multi-module program and shows many of the features of SmallC and the IAR tools. It also shows the use of the interrupt startup file `crti_2.s90`.

The complete set of files required are in the `multi.zip` file available for download from www.jennaron.com/smallc.html. As with the other examples, unpack this file into a directory of its own.

The main program sets up the two 8515 ports used (B and D), starts the five tasks, and then sits in a scheduling loop. The `multi.c` dispatcher maintains two queues – a ready queue and a delayed queue. For every call to `Dispatch`, the next ready task is run. `Dispatch` does not return until that task concludes. Note that the `ReRunMe` function allows a task to give up control but ensure it will be restarted. Delays on the delay queue are measured in clock ticks. Every clock tick counts down the delays for tasks on the delay queue. When the delay time for the task with the shortest delay goes to zero, that task is taken off the delayed queue and put on the ready queue. A feature of this dispatcher is that only the task at the head of the delay queue has its delay decremented. While this requires time be spent on inserting a new entry, time is saved during the considerably more frequent event of a clock tick.

The tasks in this example are started by being put on the run queue, so they start at once in the order they were queued.

```

/*****
multest.c
Initialise the ports B (output) and D (input). Initialise
the task dispatcher (multi.c), then call the startup
function for each of the 5 tasks. Then, in a forever loop:
1. Check if the timer interrupt has ticked (I_Timer in
crti_2.s90 non-zero). If it has, clear it and call
the DecrementDelay function in multi.c.
2. Call Dispatch in multi.c to schedule any waiting tasks.
3. Check if a switch has been pressed.
Use crt_i_2.s90.

Ron Kreymborg
*****/

```

```

#include "avrstdio.h"
#include "multi.h"
#include "multest.h"

void main(void);

extern UNSIGNED I_Timer;

void main(void) {

    outp(DDRB, 0xff);      // portB is output
    outp(PORTB, 0xff);
    outp(DDRD, 0);        // port D is input
    outp(PORTD, 0xff);
    InitMulti();          // initialise dispatcher
    QueTask(FlashOn);     // queue each task in turn
    QueTask(SequenceStart);
    QueTask(BucketIn);
    QueTask(Continous);
    QueTask(CheckSwitch);

    while (1) {
        if (I_Time) {      // if the timer has ticked
            I_Time = 0;
            DecrementDelay(); // decrement any delayed tasks
        }
        Dispatch();        // run any pending tasks
        CheckChanged();    // show any switch
    }
}

```

The clock is ticked by the clock0 hardware. This interrupt handler sets a public variable called `I_Timer` at every interrupt. The interrupt code segment from the startup file `crti_2.s90` looks like:

```

;*****
; TIMER0 interrupt. This runs at CLKTICK microseconds.

tim0_ovf
    push    r16
    in     r16,SREG        ; save status
    push   r16
    ldi    r16,CLOCK      ; reload tick
    out    TCNT0,r16
    ldi    r16,0x01       ; set wake up flag
    sts    _I_Time,r16
    pop    r16
    out    SREG,r16       ; restore status
    pop    r16
    reti

```

And the definition for `_I_Time` looks like:

```

;*****
; DATA SEGMENT IN SRAM
;*****

        rseg    sdata        ; data segment
        public  _I_Time      ; so C can get at it

I_Timer ds    1              ; 2 byte integer

```

The timer is initialised with the code:

```

; Set Timer0 running. This ticks at CLKTICK mSec intervals. Note
; relationship defining CLKTICK in mS depends on 1024 prescalar.

        ldi    r17,$05        ; set prescalar (1024)

```

```

out      TCCR0,r17
ldi      r17,CLOCK          ; load derived constant
out      TCNT0,r17
ldi      r17,$02           ; enable timer0 interrupts
out      TIMSK,r17

```

The associated constants are derived in the code:

```

;*****
; General constants

CLKTICK= 25000                ; 25uSec tick for clock0
CLOCK=   256-(CLKTICK*4/1024) ; value to load into TCNT0

```

Because the `crti` file must always be the first file in a link list, the `I_Timer` variable will be located at address 0x60 in sram. A more detailed explanation of this code and its workings is given in the *Using Interrupts with SmallC* section.

The `DecrementDelay` function in `multi.c` must be called at every clock tick to manage the internal delay queue. It can be either called directly from the interrupt routine by giving it an *interrupt* qualifier, or be called from within the main program loop as shown here. In the latter case you must be sure all functions complete within the clock tick time (25mSecs in this example – see above). The code in `multi.c` is fairly straightforward apart from the delay queue management and this is described fully in the original document in PDF format downloadable from www.jennaron.com/dispatch.html. A slight change in the `GetNewTask` function for this non-status version prevents the dangerous situation of a task appearing more than once on a queue.

The `flash.c` module is indicative of the LED drivers:

```

/*****
Controls LED 7 on the STK200.
Initially turns LED 7 on then queues FlashOff to run after
a delay. FlashOff turns LED 7 off and queues FlashOn to
run again some time later.
*****/

#include "avrstdio.h"
#include "multi.h"
#include "multest.h"

static void FlashOff(void);

void FlashOn(void) {
    UNSIGNED val;
    val = (UNSIGNED)inp(PINB);
    val &= 0x7f;
    outp(PORTB, val);
    QueDelay(FlashOff, 5);
}

static void FlashOff(void) {
    UNSIGNED val;
    val = (UNSIGNED)inp(PINB);
    val |= 0x80;
    outp(PORTB, val);
    QueDelay(FlashOn, 25);
}

```

The `FlashOn` function drives bit-7 of PORTB low, thus turning on the associated LED. It then queues the `FlashOff` task on the delay queue to start 5 clock ticks

later. When **FlashOff** eventually runs it drives bit-7 high again and re-queues **FlashOn** to start 25 ticks later. Thus LED7 cycles between an on time of 5 clock ticks and an off time of 25 clock ticks.

The switch.c module uses a number of static local variables to debounce and maintain the switch states. In operation you can press the PD3 pushbutton on the STK200 board and LED0 will light while the switch is down.

```

/*****
Debounces the STK200 pushbutton PD3 on port D.
If a switch change is detected by CheckSwitch, a 1 clock tick
delay is started. When that concludes the state is rechecked
and if not different, CheckSwitch is re-scheduled. If still
different the state is updated and the ChangedFlag set. This
is detected in the main program and the LED 0 state swapped.
*****/

#include "avrstdio.h"
#include "multi.h"
#include "multest.h"

static void CheckSwState(void);

static BYTE sw, state;
static BYTE ChangedFlag;
static BYTE val;

void CheckSwitch(void) {
    val = (BYTE)inp(PIND) & 0x08;
    if (val ^ sw)
        QueDelay(CheckSwState, 1);
    else
        ReRunMe(1);
}

static void CheckSwState(void) {
    val = (BYTE)inp(PIND) & 0x08;
    if (val ^ sw) {
        sw = val;
        ChangedFlag = 1;
    }
    QueDelay(CheckSwitch, 1);
}

void CheckChanged(void) {
    if (ChangedFlag) {
        state = ~state;
        val = (BYTE)inp(PORTB);
        val &= 0xfe;
        val |= state & 0x01;
        outp(PORTB, val);
        ChangedFlag = 0;
    }
}

```

This is not intended to show the most efficient C version of a switch debouncer. Its primary function is to demonstrate how events can be passed back to a main program for action. Here the switch is debounced during the one clock tick delay between a switch change being first detected in the **CheckSwitch** function and the subsequent re-check in the **CheckSwState** function. Naturally the delay must be long enough to ensure switch bouncing has ceased, and the 25mSec clock tick in this example should be more than adequate. If the current switch state still differs from the stored state the stored state variable **sw** is updated to reflect the new state

and the **ChangedFlag** set. The **CheckChanged** function can be repeatedly called externally with no effect unless the **ChangedFlag** is set, and in that case it will toggle the LED0 via the variable **state**. The **val** variable would normally be local, but because the dispatcher ensures that only one function can be running at a time it can be shared between functions.

Run the batch file called **make.bat** to compile and link this suite of programs. Use the ISP program to download the **multi.hex** file to the STK200. Alternately you can load the **debug.d90** file into the Studio debugger.

There is a code space cost in using structures as in the **multi.c** module, but their use simplifies many programming tasks where complex data structures are required. However, if you have a code space problem, try replacing any structures with a collection of arrays. You might like to try replacing the three variables in the **task** structure used here with three separate arrays and see what the space reduction is from the current size.

Calling Assembly Functions

Calling an assembly language function is easy. Unlike for interrupts, you can use the entire register set except the C stack pointer (Y), and you can use this too if you first push it onto the stack. In the first example we will assume a function to get the current timer0 count has been prototyped as:

```
extern int GetTimer0(void);
```

The function must return the value of the current Timer0 count. The C source showing the function use is

```
int Projector(void) {
    int retval;
    retval = GetTimer0();
    .....
    .....
}
```

The compiler creates an s90 assembly language file, part of which looks like the following sequence:

```
;          retval = GetTimer();
sbiw      r28,2          ; space for retval
mov       r26,r28       ; stack pointer into X
mov       r27,r29
st        -y,r26        ; push address of retval
st        -y,r27
rcall    _GetTimer0    ; call function
ld        r27,y+        ; pop retval address
ld        r26,y+
st        x+,r31        ; store returned value
st        x+,r30
```

The first instruction provides room on the software stack for the one integer local variable. The next four instructions move the address of the return value to the top of stack (TOS). SmallC subsequently accesses local variables by first copying the current stack pointer into either the X or Z registers, and then adding an offset from

its current position that makes that register a pointer to the lowest address (most significant byte) of the local variable. In this example, there is only one local variable and we will assume the stack is unchanged since entering this function. Thus there is nothing to add after moving the stack pointer to X and pushing the address onto the software stack.

The next instruction calls the function. It is the function's responsibility to load its return value into the primary register before returning. For chars and ints this is r31-r30 (Z), and for longs r31-r30-r27-r26 (most significant byte always in the higher numbered register). The last four instructions in the above integer example pop the `retval` address off the stack into X and store the returned value (in Z) into the `retval` variable.

Knowing this we can write the `GetTimer0()` function in assembler.

```

; Return the TIMER0 current value in Z.
        rseg      scode
        public    _GetTimer0
_GetTimer0
        in       r30,TCNT0          ; get timer 0 current value
        clr     r31                 ; return as an unsigned integer
        ret
        end

```

Of course, we could just as easily have stayed with C and written:

```

// Return the TIMER0 current value in Z.
int GetTimer0(void) {
    #asm
        in   r30,TCNT0    ; get timer 0 current value
        clr  r31         ; return as an unsigned int
    #endasm
}

```

When the function requires parameters to be passed in, you must work with the C stack. For example, suppose you have a device that requires a byte bit pattern and an integer value and, for speed reasons, must be in assembly. The C calling fragment is:

```

register char bits;
register int Zcount;

bits = 0x3e;
Zcount = 643;
SetGizmo(bits, Zcount);

```

The compiler creates the following assembly code:

```

;      bits = 0x3e;
        ldiz   62                ; immediate 16-bit load to primary
        mov    r15,r30           ; store char object in register
;      Zcount = 643;
        ldiz   643               ; immediate 16-bit load to primary
        mov    r14,r31           ; store int object in register
        mov    r13,r30
;      SetGizmo(bits, Zcount);
        mov    r30,r15           ; fetch char object from register
        clr   r31
        st    -y,r30            ; push primary
        st    -y,r31
        mov   r31,r14           ; fetch int object from register
        mov   r30,r13

```

```

st      -y,r30          ; push primary
st      -y,r31
rcall   _SetGizmo
adiw    r28,4           ; adjust stack frame

```

Note that chars are always converted to ints during function calls. The **ldiz** mnemonic is a macro that loads a number into the Z register pair (r31-30). Thus on entry to the **SetGizmo** function, the stack looks like this

0x3e
0x00
0x83
0x02

<- stack pointer (Y)

It is important that you do not alter the value in the C stack register. However you could unpack the stack with code like:

```

ldd     r31,y+0        ; get Zcount into z
ldd     r30,y+1
( go do something with it )
ldd     r20,y+3        ; get the bit pattern into r20
( go do something with that )
ret

```

Using Interrupts with SmallC

One of the nice features about the AVR architecture and a user controlled startup file is that you can define a vector to a specific interrupt routine at a specific address in code space. If you use something like David VanHorn's code in `crti_2.s90`, then you will have a dummy routine for each interrupt that normally is never used or just ensures a spurious interrupt will do no harm. While I would highly recommend you write interrupt handlers in assembler, you can define a C interrupt routine if some feature of C is absolutely necessary and time is not of the essence.

A C function with the **interrupt** qualifier will first push r16 onto the hardware stack then copy the flags register **SREG** to r16 which is also pushed onto the stack. Next the entire set of registers are pushed.

Note: Do not use the default stack assignments if you use this feature – they do not reserve enough stack space.

You can define local variables in the interrupt function and/or use global variables. When your routine completes the registers are popped off the stack, the **SREG** register restored, and a **reti** instruction executed.

For example, suppose the requirement was to add 13 times the current **TIMER1** contents to a long global variable, every time a **TIMER0** interrupt occurs. In general this would look like the following. The interrupt vector in `crti_2.s90` file is:

```

external tim0_ovf
jmp     tim0_ovf

```

The C code for this version of **tim0_ovf** could be:

```

long total;          // global variable

interrupt tim0_ovf(void) {
    int tim1;
    tim1 = inp(TCNT1H)<<8 | inp(TCNT1L);
    total += (long)tim1 * 13L;
}

```

The following examples use the interrupt enabled startup file `crti_2.s90`. For hardware they use the STK200 starter kit as a vehicle, along with the Atmel ISP package to download the program to an 8515 processor. The example task is absolutely simple, it flashes the led connected to PORTB-7. However this demonstrates a number of ways a C program can interact with interrupt functions.

Program One

This example shows how a C program can detect that an interrupt has occurred. The interrupt enabled startup file is called `crti_2.s90`. First copy this to `crti.s90`. Have a look at the three relevant sections of this file:

```

CLKTICK= 25000      ; 25uSec tick for clock0
CLOCK=   256-(CLKTICK*4/1024)
TickBit= 0
-----
;*****
; TIMER0 interrupt. This runs at CLKTICK
; microsecs.

tim0_ovf
    push    r16
    in     r16,SREG        ; save status
    push   r16
    ldi    r16,CLOCK      ; reload tick
    out    TCNT0,r16
    lds    r16,_I_Timer    ; set done bit
    sbr    r16,1<<TickBit ; set wake up flag
    sts    _I_Time,r16
    pop    r16
    out    SREG,r16       ; restore status
    pop    r16
    reti

;*****
; DATA SEGMENT IN SRAM
;*****

        rseg    sdata        ; data segment
        public  _I_Time      ; so C can get at it
_I_Time ds     1

```

We have dealt with this in the previous *Using a Multi-processing Task Dispatcher* example, but I will re-iterate the details here. The `CLKTICK` equate sets the number of microseconds we want between Timer0 overflows. The second equate does a little sum with this value to arrive at the value that must be loaded into Timer0 to get this time delay. It assumes a clock speed of 4Mhz and a prescalar of 1024. Note there are no checks for over or underflow, so do the sum yourself to ensure your result is not negative, zero, or greater than 255.

The `tim0_ovf` function is fairly straightforward. It saves register `r16` then the status register `SREG`. In this case the latter is superfluous as no subsequent

instructions here effect **SREG**, but it is always good practise to save the flags in an interrupt routine. Next it reloads the Timer0 with the 25mSec value - the counter runs continuously so this sets up another interrupt 25mSecs in the future. So far there is nothing to tell an external routine the interrupt has occurred. However the next step sets a bit in the **_I_Time** byte in sram. Note that this variable has been declared public and so will be visible to other modules if declared as *extern*.

So, in the system we propose, the Timer0 interrupts will occur every 25mSecs and will result in bit-0 being set in a global variable called **_I_Time**. Note that unless something else happens, that bit-0 will never be reset. Now let's write some C code.

With a bit of thought we can see that the C code must declare **_I_Time** as an external, must somehow detect the bit-0 being set, and must flag this in some visible way, and in this case we have previously decided to do this by flashing the LED connected to PORTB-7 on the STK200 board. If we turn the led on when the interrupt occurs, the led on time could be defined by a do-nothing for loop that counts to a few thousand. However 25mSecs would be an adequate led on time, so we may as well use the interrupt again for the on-time delay. In addition, if we count 40 interrupts before doing anything, we will get a led that flashes about every second. Remember that sccavr prefixes all names with an underscore, so the variable **_I_Time** in assembler will be referenced as **I_Time** in C. The code could look like:

```
// Name: test1.c
#include "avrstdio.h"
void main(void);
void CheckInt(void);

extern char I_Timer;

main(void) {
    int n;
    outp(DDRB, 0xff);           // all outputs
    outp(PORTB, 0xff);         // all high
    n = 0;                      // zero counter
    while (1) {                // do forever
        CheckInt();            // wait for interrupt
        if (++n == 40) {        // if 40 have occurred
            cbip(PORTB, 7);     // bit 7 of PORTB low
            CheckInt();         // keep low for 25mSec
            sbip(PORTB, 7);     // bit 7 high again
            n = 0;              // ready for next cycle
        }
    }
}

void CheckInt(void) {
    while (I_Time == 0)        // wait
        ;
    I_Time = 0;                // must reset flag
}
```

Now let us compile all this and download it to the 8515 micro. First copy the sartup file and assemble it:

```
copy \sccavr\i8515_2.s90 crti.s90
makeinit
```

The C program above is called *intex1.c*, so compile, assemble and link it with:

```
sccavr -clan intex1.c
```

Now run the Atmel ISP program, setting it up so that pressing F5 re-reads the above hex file, erases the chip, confirms the erase, programs the program memory, and confirms the programming.

If all went well you should see the PORTB-7 led start blinking about every second. If you connect an oscilloscope to the PORTB pin 7 connector on the STK200 you should see a clean 25mSec low going pulse occurring about every second.

This is by far the best method to handle interrupts in C code - process the interrupt in assembler and notify the C code with some sort of flag.

Program Two

Use the new **interrupt** function qualifier where you must process the interrupt in C. Remember that to be sure the complete C environment is saved, SmallC pushes all the AVR registers on entry to an interrupt function and pops them off again before doing the **reti**, about a 16uSec overhead at 4MHz.

You need to make two small changes to the new `crti.s90` file. At the start somewhere add the line:

```
extern _CTimer0
```

At the `tim0_ovf` entry point add the line:

```
*****
; TIMER0 interrupt.
; This runs at CLKTICK microseconds.

tim0_ovf
    rjmp    _CTimer0        ; process in C
    push   r16
    in     r16,SREG        ; save status
    push   r16
    ldi    r16,CLOCK      ; reload tick
    out    TCNT0,r16
    lds    r16,I_Timer    ; set done bit
```

This could have been added in the interrupt jump table but as we are processing the interrupt in C, time obviously doesn't matter so neither will one more jump. Now we add the interrupt function to create the new `intex2.c` example:

```
// Name: intex2.c
#include "avrstdio.h"
#define CLOCK 158
void main(void);
void CheckInt(void);
interrupt CTimer0(void);
char IntFlag;
main(void) {
    int n;
    outp(DDRB, 0xff); // all outputs
    outp(PORTB, 0xff); // all high
    n = 0; // zero counter
    while (1) { // do forever
        CheckInt(); // wait for interrupt
```

```

        if (++n == 40) {           // if 40 have occurred
            cbip(PORTB, 7);       // bit 7 of PORTB low
            CheckInt();           // keep low for 25mSec
            sbip(PORTB, 7);       // bit 7 high again
            n = 0;                // ready for next cycle
        }
    }

void CheckInt(void) {
    while (IntFlag == 0)         // wait 25mSecs
        ;
    IntFlag = 0;                 // must reset flag
}

interrupt CTimer0() {
    outp(TCNT0, CLOCK);          // reset the timer
    IntFlag = 1;                 // flag 25mSecs done
}

```

Now compile and link this version:

```
sccavr -clan -s0x21f intex2.c
```

Note the change that specifies a hardware stack size of 64 bytes. The default 32 bytes would not be enough now the **interrupt** function has been added. Once this is run and the file downloaded to the 8515 you should see exactly the same operation as before.

Program Three

Finally, if you have a fairly fast oscilloscope, adding the following statements to the interrupt function should show a low going pulse a few uSecs wide on PORTB pin 5 and running at a 25mSec rate. Every 40 pulses you should see it immediately precede the 25mSec wide low going pulse on pin 7.

```

interrupt CTimer0() {
    register int pins;
    outp(TCNT0, CLOCK);          // reset the timer
    pins = (int)inp(PORTB);      // get PORTB output state
    pins &= 0xdf;                // set pin 5 low
    outp(PORTB, pins);
    I_Timer = 1;                 // flag done
    pins |= 0x20;                // set pin 5 high again
    outp(PORTB, pins);
}

```