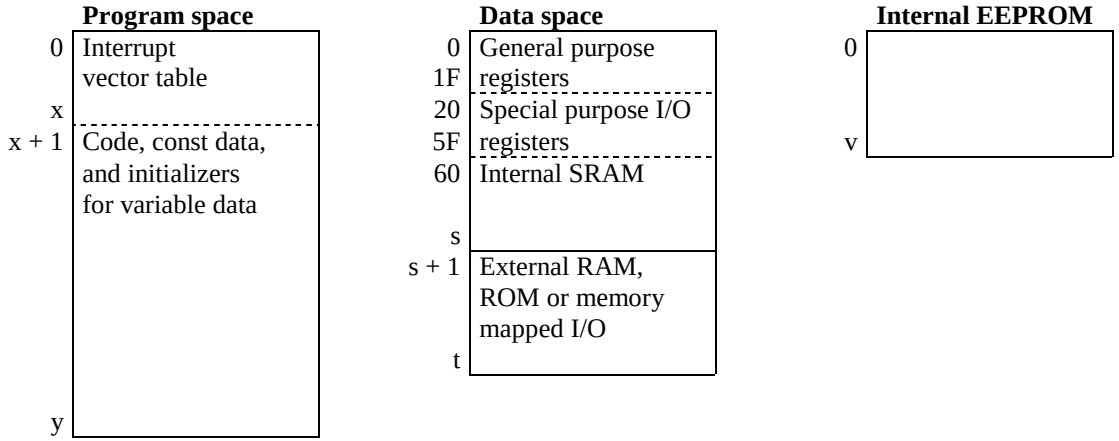
The IAR Embedded Workbench™
IAR C Compiler for AVR (ICCA90) utilization of AVR microprocessor memory space

This application note intends to clarify how the IAR C compiler - ICCA90 - utilizes the memory areas of the AVR microprocessor. The starting points of this application note are the AVR architecture, the available microprocessor versions, and fragments of C source code as it could be written by a developer of an embedded application.

The Harvard architecture, and the AVR architecture

The Harvard architecture is characterized by the separated program and data address spaces, both starting with address 0. There are separate, independent, address busses for the two memory areas. Atmel has taken this architecture one step further in the AVR architecture, and uses the low address range of both areas for special purposes. The first few bytes of the program space are used to keep the interrupt vector table (the size of the vector table is depending on processor version). The first 32 bytes of the data space are used for internal general purpose registers, and the following 64 bytes of the data space are used for internal special purpose I/O registers. The program space is completely held within the microcontroller's flash memory. There is no option to extend the program space with external memory. The data space may be extended with external memory in some of the versions of the AVR microcontroller. The AVR architecture also includes an internal EEPROM, which has an address range of its own, starting with 0. This EEPROM is only possible to access for reading and writing, one byte at a time, using certain special purpose I/O registers.



The following are depending on the version of the AVR microcontroller:

x End of interrupt vector table.
y End of internal flash.
s End of internal SRAM.
t End of data address range. Extending the data space with external memory is only possible for some microcontroller versions.
v End of internal EEPROM.

Currently available AVR microcontroller versions (autumn 1998)

All AVR microcontroller versions comes with an internal flash in program space, and an internal SRAM in data space (AT90S1200 has no SRAM). They also comes with an internal EEPROM, which is accessible through certain special purpose I/O registers. The following table shows the currently available microcontroller version and their memory configurations and options.

µ-controller version name	Compiler processor option	ICCA90 header file	Program space				Data space				Internal EEPROM, size and address range (0 - v)
			Size of internal flash	No. of interrupt vectors (incl. reset vector)	Interrupt vector width	Available address range for code (byte addr, x+1 - y)	Registers	Remarks	Internal SRAM, size and address range (60 - s)	External option, RAM, ROM or memory mapped I/O (s+1 - t)	
AT90S1200	-v0	(1)	1 Kbytes	4	2 bytes	8 - 3FF	(2)		None.	None.	64 bytes: 0 - 3F
AT90S2313	-v0	io2313.h	2 Kbytes	11	2 bytes	16 - 7FF	(2)		128 bytes: 60 - DF	None.	128 bytes: 0 - 7F
AT90S2323 AT90LS2323	-v0	io2323.h	2 Kbytes	3	2 bytes	6 - 7FF	(2)		128 bytes: 60 - DF	None.	128 bytes: 0 - 7F
AT90S2343 AT90LS2343	-v0	io2323.h	2 Kbytes	3	2 bytes	6 - 7FF	(2)		128 bytes: 60 - DF	None.	128 bytes: 0 - 7F
AT90S4414	-v1	io4414.h	4 Kbytes	13	2 bytes	1A - FFF	(2)		256 bytes: 60 - 15F	~ 64 Kbytes: 160 - FFFF	256 bytes: 0 - FF
AT90S8515	-v1	io8515.h	8 Kbytes	13	2 bytes	1A - 1FFF	(2)		512 bytes: 60 - 25F	~ 64 Kbytes: 260 - FFFF	512 bytes: 0 - 1FF
AT90S8535 AT90LS8535	-v1	io8535.h	8 Kbytes	17	2 bytes	22 - 1FFF	(2)		512 bytes: 60 - 25F	~ 64 Kbytes: 260 - FFFF	512 bytes: 0 - 1FF
ATmega103 ATmega103L	-v3	iom103.h	128 Kbytes	24	4 bytes	60 - 1FFFF	(2)	Has RAMPZ	~ 4 Kbytes: 60 - FFF	60 Kbytes: 1000 - FFFF	4 Kbytes: 0 - FFF

(1) It is actually possible to write applications in C for the AT90S1200, using the ICCA90 C compiler, but it is not recommended unless the developer has a deep knowledge in how the IAR tools work. Instead the recommendation is to implement the application by writing it in assembler source code, utilizing the AA90 and XLINK tools to generate the downloadable code.

(2) All microcontroller versions have the following internal registers: 32 general purpose registers, data address range: 0 - 1F
64 special purpose I/O registers, data address range: 20 - 5F

Corresponding IAR compiler processor options:

Compiler processor option	Max program address range (byte address)	Interrupt vector width	Max data address range	The compiler assumes the existence of RAMPX	RAMPY	RAMPZ	RAMPD
-v0	0 - 1FFF (8 Kbytes)	2 bytes	0 - FF (256 bytes)	no	no	no	no
-v1	0 - 1FFF (8 Kbytes)	2 bytes	0 - FFFF (64 Kbytes)	no	no	no	no
-v2	0 - 1FFFF (128 Kbytes)	4 bytes	0 - FF (256 bytes)	no	no	yes	no
-v3	0 - 1FFFF (128 Kbytes)	4 bytes	0 - FFFF (64 Kbytes)	no	no	yes	no
-v4	0 - 1FFFF (128 Kbytes)	4 bytes	0 - FFFFFF (16 Mbytes)	yes	yes	yes	yes (3)
-v5	0 - 7FFFFF (8 Mbytes)	4 bytes	0 - FFFF (64 Kbytes)	no	no	yes	no
-v6	0 - 7FFFFF (8 Mbytes)	4 bytes	0 - FFFFFF (16 Mbytes)	yes	yes	yes	yes (3)

(3) Some microcontroller derivates don't have RAMPD, but are still related to compiler option -v4 or -v6. In these cases, utilize command line option `-u_no_rampd`.

How ICCA90 utilizes the memory spaces

The following table contains a trace of how the ICCA90 compiler utilizes the program and data memory for different purposes. The main starting point of a table entry is a fragment of C source code, as it could be written by a developer of an embedded application. The table contains the following columns:

Item to put in memory	Contains the type of item that needs to be "put" somewhere in an appropriate part of an appropriate memory space.
Actual source	Contains a C source code fragment example, or equal.
Com. (comment)	Contains a comment number which applies to the rest of the table row. The actual comment is found below the table.
Memory space	States which memory space in the AVR architecture that is used for this item. If nothing else is said, then "Program" means internal flash, and "Data" means internal or external RAM.
Address range (hex)	Specifies the valid *logical* address range of the specified memory space (byte addresses in data space as well as in program space), within which the item shall be put. The valid *logical* range for different items differs between the AVR microcontroller versions: x = End of the interrupt vector table. s = End of the internal SRAM. v = End of the internal EEPROM. Note, these *logical* address ranges indicates the "soft" limits, handled within the IAR toolkit. The real, "hard", address ranges of the program and data memories depends on the actual hardware configuration, i.e. on the actual AVR microcontroller version and optionally added external (data) memory. It's recommended to let the "hard" range be reflected by the customized content of the used linker file (*.xcl). The "hard" range must always be within the "soft" range.
Segment	Specifies the segment name chosen by the ICCA90 compiler. The name may be altered by the programmer, by the use of `#pragma` directives or compiler command line options or XLIB utilities.
Linker memory type	Tells which directive to use to indicate to the linker which type of memory a certain segment or group of segments belong to. Example: `-Z(FAR)IDATA2,UDATA2=10000` where `FAR` is the linker memory type, and `IDATA2` and `UDATA2` are the segment names. Linker memory types in program space: CODE, FARCODE, HUGECODE. Linker memory types in data space: DATA, NEAR, FAR, HUGE, NEARC, NEARCONST, FARC, FARCONST, HUGEC, HUGECONST, NPAGE, ZPAGE.
C-Spy simulator memory section	Indicates in which display section of the memory window the segment is presented in the C-Spy simulator.

Table borders:

Separator between different types of storage items

Separator between alternatives within one type of storage item

Help line, to improve readability

Item to put in memory	Actual source	Com.	Memory space	Address range (hex) (byte address)	Segment	Linker memory type	C-Spy simulator memory section

Auto variables	`void foo(void)`
	`{ int i;`
	`  int ii = 1; ... }`
		7, 10.a	Data (gen.purp.reg)	`04- 0F, 18- 1B`	-	-	DATA

				`R4-R15, R24-R27`	-	-	Register window
		7, 10.b, 11	Data (see data stack below)				
Function parameters	`void foo(int p)` `{ ... }`	10.a	Data (gen.purp.reg)	`10 - 17`	-	-	DATA
				`R16 - R23`	-	-	Register window
		10.b, 11	Data (see data stack below)				
Return values	`int foo(void)` `{ return 1;  }`	10.a	Data (gen.purp.reg)	`10 - 13`	-	-	DATA
				`R16 - R19`	-	-	Register window
		10.b, 11	Data (see data stack below)				

0-initialized global and static variables	`tiny int t;`
	`static tiny int st;`
	`void foo(void)`
	`{ static tiny int lst; ... }`
		13	Data	`60 - FF`	`UDATA0`	`DATA`	DATA

	`near int n;`
	`static near int sn;`
	`void foo(void)`
	`{ static near int lsn; ... }`
		13	Data	`60 - FFFF`	`UDATA1`	`DATA`	DATA

	`far int f;`
	`static far int sf;`
	`void foo(void)`
	`{ static far int lsf; ... }`
		13	Data	`60 - FFFFFF`	`UDATA2`	`FAR`	DATA

	`huge int h;`
	`static huge int sh;`
	`void foo(void)`
	`{ static huge int lsh; ... }`
		13	Data	`60 - FFFFFF`	`UDATA3`	`HUGE`	DATA

Initialized global and static variables	`tiny int ti = 1;`
	`static tiny int sti = 1;`
	`void foo(void)`
	`{ static tiny int lsti = 1; ... }`
		2, 15	Program	*x+1* - See com. 15	`CDATA0`	`CODE`	CODE

			Data	`60 - FF`	`IDATA0`	`DATA`	DATA

	`near int ni = 1;`
	`static near int sni = 1;`
	`void foo(void)`
	`{ static near int lsni = 1; ... }`
		2, 15	Program	*x+1* - See com. 15	`CDATA1`	`CODE`	CODE

			Data	`60 - FFFF`	`IDATA1`	`DATA`	DATA

	`far int fi = 1;`
	`static far int sfi = 1;`
	`void foo(void)`
	`{ static far int lsfi = 1; ... }`
		2, 15	Program	*x+1* - See com. 15	`CDATA2`	`FARCODE`	CODE

			Data	`60 - FFFFFF`	`IDATA2`	`FAR`	DATA

	`huge int hi = 1;`
	`static huge int shi = 1;`
	`void foo(void)`
	`{ static huge int lshi = 1; ... }`
		2, 15	Program	*x+1* - See com. 15	`CDATA3`	`CODE`	CODE

			Data	`60 - FFFFFF`	`IDATA3`	`HUGE`	DATA
Non-volatile variables, and	`no_init int ni;`	9	Data (non-volatile memory or memory mapped I/O)	*s+1* - `FFFFFF`	`NO_INIT`	`DATA`	DATA

Item to put in memory	Actual source	Com.	Memory space	Address range (hex) (byte address)	Segment	Linker memory type	C-Spy simulator memory section
memory mapped I/O							
	`_EEPUT(eeprom_address, value)` `_EEGET(variable, eeprom_address)`		Internal EEPROM	`00 - v`	-	-	EEPROM
Constant data	`flash int fi = 1;`	8	Program	*x+1* - `FFFF`	`FLASH`	`CODE`	CODE
	`tiny const int t = 1;`	2, 15	Program	*x+1* - See com. 15	`CDATA0`	`CODE`	CODE
			Data	`60 - FF`	`IDATA0`	`DATA`	DATA
	`near const int n = 1;`	1, 2, 15	Program	*x+1* - See com. 15	`CDATA1`	`CODE`	CODE
			Data	`60 - FFFF`	`IDATA1`	`DATA`	DATA
		3	Data (external PROM)	*s+1* - `FFFF`	`CONST`	`DATA`	DATA
	`far const int f = 1;`	1, 2, 15	Program	*x+1* - See com. 15	`CDATA2`	`FARCODE`	CODE
			Data	`60 - FFFFFF`	`IDATA2`	`FAR`	DATA
		3	Data (external PROM)	*s+1* - `FFFFFF`	`CONST`	`FAR`	DATA
	`huge const int h = 1;`	2, 15	Program	*x+1* - See com. 15	`CDATA3`	`CODE`	CODE
			Data	`60 - FFFFFF`	`IDATA3`	`HUGE`	DATA
	`"string literal"`	2, 4, 14, 15	Program	*x+1* - See com. 15	`CCSTR`	See com. 14	CODE
			Data	`60` - See com. 14	`ECSTR`	See com. 14	DATA
		5, 14	Data (external PROM)	*s+1* - See com. 14	`CSTR`	See com. 14	DATA
Interrupt functions	`interrupt [XYZ]` `  void xyz_handler(void) {...}`	6.a	Program	`00 - x`	`INTVEC`	`CODE`	CODE
		6.b, 6.c	Program	*x+1* - `7FFFFE`	***`RCODE`***	`CODE`	CODE
Normal function bodies	Common C code...		Program	*x+1* - `7FFFFE`	`CODE`	`CODE`	CODE
	`switch` statements		Program	*x+1* - `FFFF`	`SWITCH`	`CODE`	CODE
Stacks	Data stack, implicit	10, 11.a	Data	`60 - FF`	`CSTACK`	`DATA`	DATA
		10, 11.b	Data	`60 - FFFF`	`CSTACK`	`DATA`	DATA
		10, 11.c	Data	`60 - FFFFFF`	`CSTACK`	`FAR`	DATA
	Return stack, implicit	12	Data	`60 - FFFF`	`RSTACK`	`DATA`	DATA

Internal special purpose I/O registers	`sfrb ABCB = 0x0A;  // One byte`
	`void foo(void)`
	`{ char c;`
	`  c = ABCB;  }`
			Data (I/O reg space)	Data `20 - 5F`
				I/O reg. `00 - 3F`
					-	-	DATA
							I/O SPACE

	`sfrw ABCW = 0x0A; // Two bytes`
	`void foo(void)`
	`{ int i;`
	`  i = ABCW;  }`
			Data (I/O reg space)	Data `20 - 5F`
				I/O reg. `00 - 3F`
					-	-	DATA
							I/O SPACE

Table comments:

(*) Each combination of processor option and memory model has a default memory type, and also one or more additionally available memory types. All constant and variable data which are put into the segments CCSTR, ECSTR, CSTR, CONST and CSTACK are always implicitly belonging to the default memory type. Note, CSTR and CONST are never used if the tiny memory model (-mt) is chosen (see comment 14).

Processor option	Memory model	Default memory attribute	Additionally available mem. attributes	CCSTR (-y option)		ECSTR (-y option)		CSTR (not -y option)		CONST (not -y option)	
				lnk mem type	address range (byte address)	lnk mem type	address range	lnk mem type	address range	lnk mem type	address range
-v0, -v1, -v2, -v3, -v5	-mt (Tiny)	`tiny`	`near`	`CODE`	*x+1* - See com. 15	`DATA`	`60 - FF`	-	-	-	-
-v1, -v3, -v5	-ms (Small)	`near`	`tiny`	`CODE`	*x+1* - See com. 15	`DATA`	`60 - FFFF`	`DATA`	*s+1* - `FFFF`	`DATA`	*s+1* - `FFFF`
-v4, -v6	-ms (Small)	`near`	`far, huge`	`CODE`	*x+1* - See com. 15	`DATA`	`60 - FFFF`	`DATA`	*s+1* - `FFFF`	`DATA`	*s+1* - `FFFF`
-v4, -v6	-ml (Large)	`far`	`near, huge`	`FARCODE`	*x+1* - See com. 15	`FAR`	`60 - FFFFFF`	`FAR`	*s+1* - `FFFFFF`	`FAR`	*s+1* - `FFFFFF`

1. If the memory type isn't the default memory type (*), or if compiler command line option -y is present.
2. The initialized information is copied from program to data memory at startup. This is done in the ?C_STARTUP function in CSTARTUP.S90. The application accesses the variable in the data space. The used program space is never accessed after startup is fulfilled.
3. If the memory type is the default memory type (*), and if compiler command line option -y isn't present.
4. If compiler command line option -y is present.
5. If compiler command line option -y isn't present.
6. 6.a is the interrupt vector, 6.b is the interrupt function, both must be present. Note 6.c, the compiler doesn't put the interrupt function in the RCODE segment automatically, instead the programmer has to do it explicitly (CODE is used by default). One way to accomplish this is to use the `#pragma codeseg(RCODE)` directive in the C source file which contains the interrupt function(s). Another way is to utilize the compiler command line option `-RRCODE` when compiling the said file.
7. Auto variable, i.e. a variable which is transient and local within a function. Only the default memory type (*) is allowed for an auto variable. Initialization of an auto variable is done in the generated application code, i.e. each time the function is called.
8. To reach constant data in program space, use the intrinsic function `_LPM()`, or the specific string handling functions defined in `pgmspace.h`. Note, the `FLASH` segment must reside within the first 64 Kbytes of program memory (the `LPM` assembler instruction takes a 16 bit address as argument).
9. Note, the `no_init` keyword cannot be used to define variables within the internal EEPROM.
10. General purpose registers (10.a) are used to keep some auto variables, some function parameters and some return values. The data stack (10.b) is used to keep other auto variables, other function parameters and other return values, as well as saved local registers. The decision which storage to use is made internally by the compiler, and is depending on the number of and sizes of said variables and parameters. Auto variables, function parameters and return values always belong to the default memory type (*).
11. (See comment 10.b as well.) The Y register (R28-R29) is used as data stack pointer. Because of the fact that auto variables, function parameters and return values always belong to the default memory type (*), the data stack must consequently be handled as an area which keeps variables of the default memory type. Therefore, if the default memory type is `tiny`, then the range of the data stack must be 60-FF (11.a), if `near` is default, then the range is 60-FFFF (11.b), and if `far` is default, then the range is 60-FFFFFF (11.c). The data stack must in all cases be kept completely within a 64 Kbytes block. In the 11.c case (where `far` is default), a RAMPY register must be present in the microcontroller to let the data stack be positioned above address FFFF.

12. The stack pointer SP is used as the return stack pointer. The SP is 16 bits wide and kept in two special purpose I/O registers. The return stack always has to be positioned completely within the first 64 Kbytes of data space.
13. All 0-initialized variables are initialized with 0 (0x00) at startup. This is done in the ?C_STARTUP function in CSTARTUP.S90.
14. The CCSTR, ECSTR and CSTR segments shall be related to linker memory types according to the table above (*). The table also indicates the valid address range in each case. Note, the -y option is always implicit for memory model -mt (tiny), i.e. the CSTR segment is never used if -mt is chosen - instead the CCSTR and ECSTR segments are chosen in this case to keep string literals.
15. The segments that contains initialized data in program space, CDATAx and CCSTR, must be kept within a range which is reachable by the LPM assembler instruction, or by the ELPM instruction, if that is supported by the microcontroller. If the ELPM instruction is supported, then the complete program space is reachable, hence the valid range for the segments is the complete program address range above the interrupt vector table. If LPM is the only supported instruction, then the segments with initialized data must be kept within the first 64 Kbytes of program address range (still above the interrupt vector).

Address ranges

Memory attribute	Address range (byte address)	Comment
`tiny`	0 - FF	
`near`	0 - FFFF	
`far`	0 - FFFFFF	A far object is not allowed to cross a 64 Kbytes boundary. I.e. the 8 most significant address bits must be the same for all bytes in a far object.
`huge`	0 - FFFFFF	A huge object may cross a 64 Kbytes boundary.

A more extensive example

The following sample code intends to clarify how the keywords may be used to tell the compiler to put different data in different address ranges:

```
tiny char near *p;      // Declares p as a pointer to a near character, which is found in the range 0-FFFF.
                        // p itself is put in the tiny range, 0-FF.

near char a[] = "ABC";  // Declares an array of characters which is put in the near range 0-FFFF.

void foo(void)
{
      p = a;            // p points to the first character in the array a.
}
```