# AVR032: Linker Command Files for the IAR ICCA90 Compiler

## Features

- **XLINK commands**
- **Segment explanation and location**
- **Linker file examples for:**
  - **AT90S2313**
  - **AT90S8515**
  - **AT90S8515 with external RAM and Memory mapped I/O**

## Introduction

This application note describes how to make a linker command file for use with the IAR ICCA90 C-compiler for the AVR® microcontroller.

## Background

The C-compiler converts the source code to object code, which can be executed by a microcontroller. This code is divided in modules with blocks of code and data. The output from the compiler is relocatable, which means it has no absolute memory addresses.

When the code is linked with XLINK, the code is placed at actual addresses in memory. The linker also adds pre-compiled code from external libraries.

When XLINK reads an external library module only the modules referred by the user program will be loaded.

The output from XLINK is executable code that can be downloaded to the flash controller, or simulated in AVR Studio.

To instruct XLINK what to do, the user writes a command file. The command file contains a series of instructions to XLINK. The IAR Embedded Workbench includes a set of standard XLINK command files for different memory sizes. These files are based upon assumptions about the target system which may not be valid. Even if the assumptions are valid, modifying the command file will in most cases lead to better memory utilization.

This application note describes how to make a custom command file for the AVR controllers. To make a customized XLINK linker file, use a text editor to write the code. Save the file in the project directory with .XCL extension, e.g. mylink.xcl.

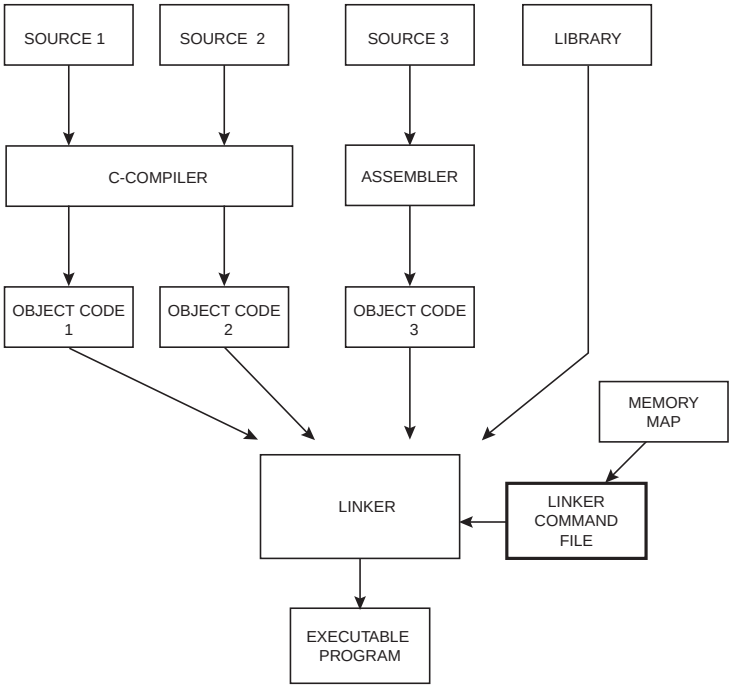
To use the file in the Embedded Workbench, select project ⟶ options, XLINK ⟶ include. Select override default at XCL file name, and select the new .xcl file.

**Linker Command Files for the IAR ICCA90 Compiler**

**Application Note**

Rev. 1079A–10/98

**Figure 1.** The linker places the executable object code in memory




## XLINK Commands

The XLINK linker commands which are used in the command file are briefly described in the following section. All addresses and sizes are given as hexadecimal values. For a complete reference see IAR Assembler Users Guide, XLINK options summary.

**Comments:**

```
-!  This is a comment     -!
Comments starts and stops with the -! sign
```

**Define CPU type:**

```
-c<cpu>
```

Example:

```
-ca90
```

Defines AVR as CPU type. Always start the XLINK file with this command.

**Define segments:**

```
-Z(memory type)segment name,……,segment name=
start(Hex)- end(Hex)
```

Example:

```
-Z(CODE)RCODE,CDATA0=1C-1FFF
```

Defines segments in flash memory. The RCODE segment here starts at address 1C immediately followed by the CDATA0 segment. If the total size of these segments are larger than the space offered, an error message will be given.

Example:

```
-Z(DATA)IDATA1,UDATA1,ECSTR,CSTACK+40=120-25F
```

Defines segments in RAM memory. IDATA1 will start at address 120, followed by UDATA1 and ECSTR. CSTACK + 40 means that the CSTACK segment will start 40 bytes (hex) higher than the end of ECSTR. (The stack grows backwards)

**Define replace names for external symbols:**

```
-ereplace_name
```

Example:

```
-e_small_write=_formatted_write
```

Replaces the external standard _formatted_write routine with the reduced _small_write. This is often done with the read and write routines scanf() and printf(), since the standard ANSI input and output routines are very comprehensive and result in large code.

**Disable warnings:**

```
-wno
```

Example:

```
-w29
```

Disable warning number 29.

## Segments

The AVR microcontroller can use several types of memory:

- **Program memory.** Flash memory that holds read-only segment
- **Internal RAM.** On-Chip SRAM, read-write segments
- **External Memory.** Connected to the external data bus. Can be e.g. SRAM, EPROM, EEPROM or memory-mapped I/O.

The various memory types and segments are described below. The user may also define segments, and place variables at a specific location.

### Program Memory Segments

Segments in program memory are read only.

Note: XLINK always counts segments in bytes, while the AVR program address counter counts words.

**INTVEC**

Holds the reset and interrupt vectors for the controller. For devices with less than 8K bytes memory each interrupt vector holds an RJMP (Relative Jump) instruction which is 2 bytes long. For devices with more than 8K bytes memory each interrupt vector holds a JMP (Jump) instruction which is 4 bytes long. See the AVR databook, reset and interrupt handling for details.

The size of this segment must be given by the user.

For AT90S8515 this segment is located at address 0 - 1B. This gives 28 locations which is sufficient to hold the RESET vector and the 13 interrupt vectors:

Example:

```
-Z(CODE)INTVEC=0-1B
```

**RCODE**

Holds code reachable with the RJMP instruction from INTVEC segment. C-STARTUP is placed in the RCODE segment. C-STARTUP performs low level initialization of the processor:

- Initialization of stack pointers for data and program
- Initializes static variables
- Calls the C function **main** ( )

Normally, C-STARTUP should be left unchanged. See ICCA90 Users Guide for instructions on how to modify the default C-STARTUP routine. RJMP can reach the entire address space for controllers up to 8K bytes of program memory (e.g., AT90S8515).

For devices with more than 8K bytes program memory, the interrupt vectors are 2 words (4 bytes). This means each interrupt vector can hold a JMP instruction which reaches the entire memory space.

The size of this segment is deduced by XLINK.

**CDATA0, CDATA1, CDATA2, CDATA3**

Holds initialization constants for tiny, small, far and huge data. At startup these segments are copied to the RAM segments IDATA. The sizes of these segments are deduced by XLINK.

Example:

```
char i = 0; /* GLOBAL C VARIABLE */
```

**CCSTR**

Contains C string literals. At startup this segment is copied to the ECSTR segment in SRAM. The size of this segment is deduced by XLINK.

**FLASH**

Contains constants declared as type flash. The constants are accessed in the program with the LPM instruction. The size of this segment is deduced by XLINK.

Example:

```
flash char mystring[ ] = “String in flash memory” ;
```

This C-code to declares a constant array which is stored in flash memory.

**SWITCH**

Contains jump tables generated by switch statements. The size of this segment is deduced by XLINK.

**CODE**

Contains the program code. The size of this segment is deduced by XLINK.

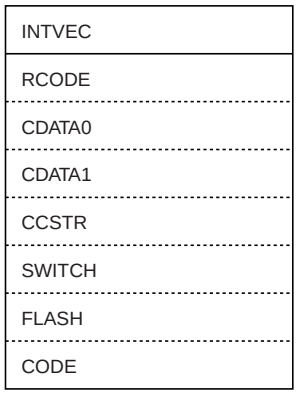
Declaring segments in program memory is straightforward. Two parameters are important: Size of interrupt vector table, and size of program memory on the device.

The following lines are sufficient to declare program memory segments:

```
-Z(CODE)INTVEC=0-Interrupt vector size(bytes)
-Z(CODE)RCODE,CDATA0,CDATA1,CCSTR,SWITCH,
FLASH,CODE=Interrupt vector size(bytes)-End of program memory(bytes)
```

This will set up the memory like Figure 2:

**Figure 2.** Program Memory Map

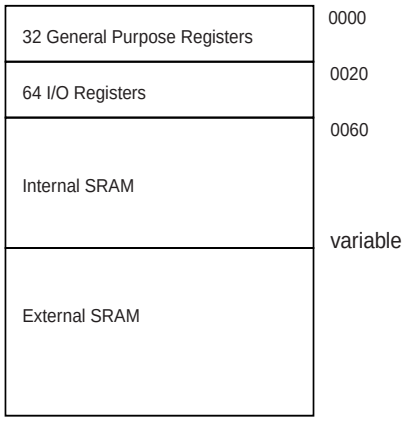


## Data Memory Segment:

Data memory consists of internal and external RAM. The 32 general purpose registers are mapped into RAM addresses 0-1F (hex), the 64 I/O registers are mapped into addresses 20-5F (hex). Internal RAM is starting at address 60 (hex). The start of the external RAM area is device dependent.

Variables in RAM are read-write variables.

**Figure 3.** Data Memory Map




**UDATA0, UDATA1, UDATA2, UDATA3**

Uninitialized data for tiny, small, far and huge variables respectively. Contains space for variables which are not initialized at declaration. The size of this segment can either be given by the user or deduced by XLINK. The latter is recommended.

**IDATA0, IDATA1, IDATA2, IDATA3**

Initialized data for tiny, small, far and huge variables. Holds data that are initialized at declaration. Variables in IDATA are copied from the corresponding CDATA segment in the code at startup. The size of this segment can either be given by the user or deduced by XLINK. The latter is recommended.

If the compiler option -y (writable strings) is active, const objects will be copied to IDATA segment from CDATA at startup.

Note: Variables declared as tiny are placed in the IDATA0 and UDATA0 segments. Tiny variables can be reached by using 8-bit address. This give them a address reach of 256 bytes (0-FF). Due to the fact that the general purpose registers and I/O registers are memory mapped, tiny variables must not be placed on addresses below 60 (hex), and they must not be placed at addresses higher than 255 (hex) (FF).

Example:

```
tiny int temp;
```

C-declaration of a variable placed in the UDATA0 segment.

There are several ways of setting the segments for tiny variables.

Example:

```
-Z(DATA)IDATA0,UDATA0=60-FF
```

This allocates the address space between addresses 60-FF (hex) for tiny variables. It allocates the entire address space from address 60 to address FF, even if the program does not use tiny variables! If the program uses more tiny variables than there is space for the user will get an error message.

Example:

```
-Z(DATA)IDATA0,UDATA0,RSTACK+20,IDATA1,
UDATA1,ECSTR,CSTACK+60=60-25F
```

This places the tiny variables in the lower part of the internal RAM address space, immediately followed by the RSTACK segment. No RAM space will be lost if there is few tiny variables, but no warning will be given if the program contains so many tiny variables that the IDATA0/UDATA0 exceed address FF (hex).

Watch out for unpredictable behavior of the program caused by this possibility, read the linker map file listing carefully to investigate the actual space required by the tiny variables.

**RSTACK**

Return stack. This segment holds the return addresses of function calls. The stack pointer is used to access this stack. The size of RSTACK is application dependent. Each call to a function requires 2 bytes on the stack for return addresses. Return addresses for interrupt routines are also stored on the return stack. If the stack size is declared too small, the stack will overwrite another segment in the data area.

**ECSTR**

Holds writable copies of C string literals if the compiler option -y (writable strings) is active. This segment is copied from CCSTR segment in CODE at startup. If there is a shortage of data memory, check whether the strings are constants and use flash declarations instead to minimize data memory usage.

**CSTACK**

Data stack. This segment holds the return stack for local data. The Y-pointer (R28-R29) is used to access this stack. The size of CSTACK is application dependent. The CSTACK is used to store local variables and parameters, temporary values and storing of registers during interrupt. If the stack size is too small, the stack will overwrite another segment in the data area.

## External PROM

Warning: If the compiler option -y (writable strings) is not active (default), the compiler assumes there is an external PROM in the system. In most cases the system does not have an external PROM, and the writable string should be active (checked). To minimize data memory usage it is recommended to use the flash keyword for constants.

Example:

```
flash char mystring[ ] = “String in flash memory”;
```

The following read-only segments are placed in external PROM.

**CONST**
Holds variables declared as const.

**CSTR**
Holds string literals when the -y (writable strings) is inactive.

Note: The CONST and CSTR should only be included in the XLINK file if there is an external PROM in the system.

## User Defined Segments

The user may define segments and place variables at absolute addresses in memory.

Example: Memory mapped real time clock placed at absolute address in external address space. Linker file command:

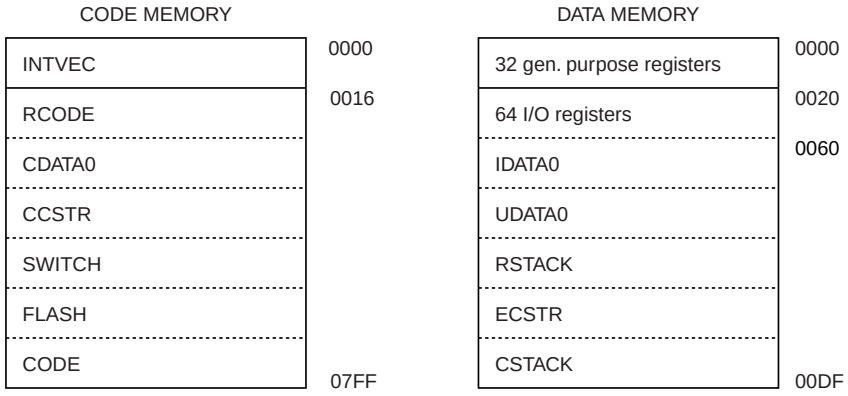
```
-Z(DATA)RTC=0F00-0F70
```

C-Code to place variables in this memory mapped I/O:

```
#Pragma Memory=DATASEG(RTC)
UNSIGNEDCHAR SEC,MIN,HOURS;
#PRAGMA MEMORY=DEFAULT
```

## Example Code 1

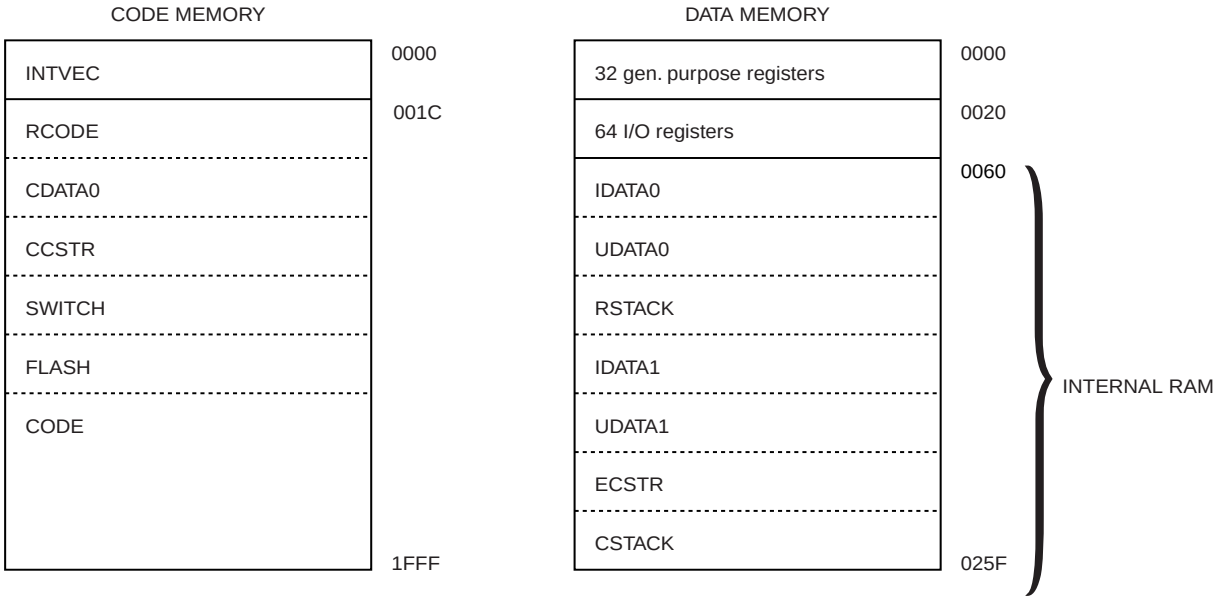
Example using AT90S2313 with 2K bytes flash memory and 128 bytes internal RAM. The segments will be set up like the memory map below. In code memory only the INTVEC segment has a specific address location. The other segments will be placed at the subsequent addresses in the order specified in the linker file. In RAM, only the order of the segments are specified, not the specific address locations.




```
-!    XLINK command file for the AT90S2313 128 bytes(60 - DF) data address
      space and 2 Kbytes(0- 7FF) program address space. -!
-!    Define CPU type (AVR) -!
-ca90
-! Define reset and I/O interrupt vector segment, requires 22(dec) locations -!
-Z(CODE)INTVEC=0-15
-! Define segments in flash memory -!
-Z(CODE)RCODE,CDATA0,CCSTR,SWITCH,FLASH,CODE=16-7FF
-! Define segments in RAM
The registers are in 0-1F, memory mapped I/O in 20-5F, Built-in SRAM in 60-DF.     -!
-! Return stack size is 10 bytes(hex), data stack size is 40 bytes(hex) -!
-Z(DATA)IDATA0,UDATA0,RSTACK+10,ECSTR,CSTACK+40=60-DF
-! Select reduced "printf" support to reduce library size.
See the configuration section of the IAR C-compiler Users Guide concerning use of printf/sprintf. -!
-e_small_write=_formatted_write
-e_small_write_P=_formatted_write_P
-! Disable floating-point support in "scanf" to reduce library size.
See the configuration section of the IAR C-compiler Users Guide concerning use of scanf/sscanf -!
-e_medium_read=_formatted_read
-e_medium_read_P=_formatted_read_P
-! Suppress one warning which is not relevant for this processor -!
-w29
-! Load the tiny 'C' library for processor option_VO -!
cl0t
```

## Example Code 2

Example using AT90S8515 with 8K bytes flash memory and 512 bytes internal RAM. The segments will be set up like the memory map below. In code memory only the INTVEC segment has a specific address location. The other segments will be placed at the subsequently addresses in the order specified in the linker file. In RAM, only the order of the segments are specified.

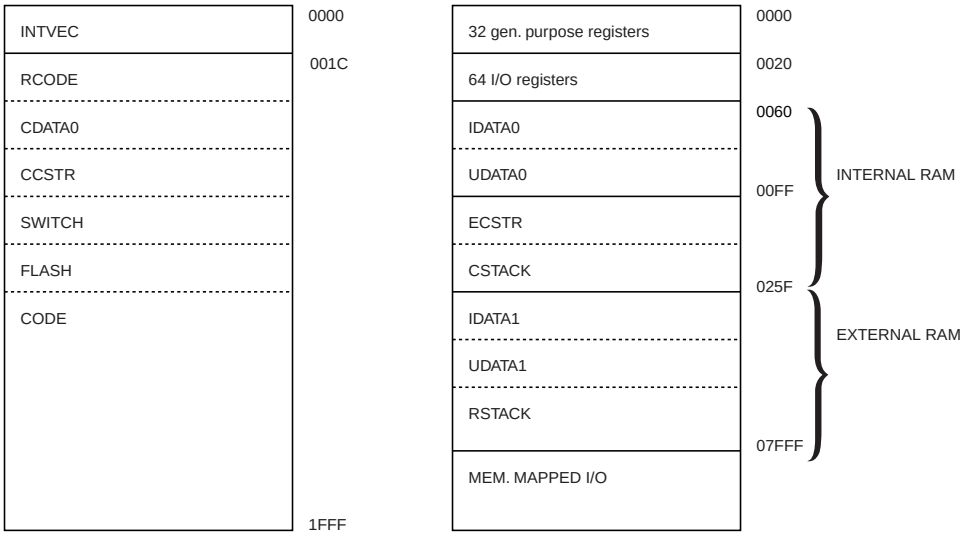



```
-!    XLINK command file for AT90S8515. 512 bytes data address
          space and 8 Kbytes program address space. -!
-!    Define CPU type (AVR) -!
-ca90
-! Define reset and interrupt vector segment, requires 28(dec) locations -!
-Z(CODE)INTVEC=0-1B
-! Define segments in flash memory -!
-Z(CODE)RCODE,CDATA0,CDATA1,CCSTR,SWITCH,FLASH,CODE=1C-1FFF
-! Define segments in RAM    -!
    -!  The registers are in addresses 0-1F and memory mapped I/O in addresses 20-5F, built-in SRAM in
        addresses 60-25F.
        Data stack(CSTACK) size is 60 bytes(hex), return stack(RSTACK) size is 20 bytes(hex) -!
-Z(DATA)IDATA0,UDATA0,RSTACK+20,IDATA1,UDATA1,ECSTR,CSTACK+60=60-25F
-! Select reduced "printf" support to reduce library size.
        See the configuration section of the IAR C-compiler Users Guide concerning use of printf/sprintf. -!
-e_small_write=_formatted_write
-e_small_write_P=_formatted_write_P
-!  Disable floating-point support in "scanf" to reduce library size.
    See the configuration section of the IAR C-compiler Users Guide concerning use of scanf/sscanf -!
-e_medium_read=_formatted_read
-e_medium_read_P=_formatted_read_P
-! Suppress one warning which is not relevant for this processor -!
-w29
-! Load the small 'C' library for processor option_v1-!
cl1s
```

## Example Code 3

Example using AT90S8515 with 8 Kbytes flash memory, 512 bytes internal RAM, 32Kbytes external RAM and memory mapped I/O. The RSTACK (return stack) is placed in external memory.

In code memory only the INTVEC segment has a specific address location. The other segments will be placed at the subsequent addresses in the order specified in the linker file. In RAM, the addresses from 60-FF (hex) are reserved for tiny variables. The rest of the internal memory is reserved for ECSTR and CSTACK segment.




```
-!    XLINK command file for AT90S8515.  512 bytes internal data address
          space, 32Kbytes external SRAM, memory mapped I/O
          and 8 Kbytes program address space. -!
-!    Define CPU type (AVR) -!
-ca90
-! Define interrupt vector segment -!
-Z(CODE)INTVEC=0-1B
-! Define segments in flash memory -!
-Z(CODE)RCODE,CDATA0,CDATA1,CCSTR,SWITCH,FLASH,CODE=1C-1FFF
-! Define segments in RAM
    Built-in SRAM in 60-25F. The registers are in 0-1F and memory mapped I/O in 20-5F -!
-! The IDATA0 and UDATA0 segments contains tiny variables, the segments must be placed within the reach of a
tiny (8 bits) pointer. -!
-Z(DATA)IDATA0,UDATA0=60-FF
-! Data stack in internal memory, size is 100(hex)bytes     -!
-Z(DATA)ECSTR,CSTACK+100=100-25F
-! 32Kbytes external SRAM starting, using near variables        -!
    -! Return stack size is 40(hex) bytes in external RAM           -!
    -! First tell CSTARTUP that RSTACK is placed in External RAM    -!
-e?RSTACK_IN_EXTERNAL_RAM=?C_STARTUP
-Z(DATA)IDATA1,UDATA1,RSTACK+40=260-7FFF
-! External memory mapped IO is used -!
-Z(DATA)NO_INIT=8000-FFFF
-!  Select reduced "printf" support to reduce library size.
        See the configuration section of the IAR C-compiler Users Guide concerning use of printf/sprintf. -!
-e_small_write=_formatted_write
-e_small_write_P=_formatted_write_P
```

```
-!  Disable floating-point support in "scanf" to reduce library size.
    See the configuration section of the IAR C-compiler Users Guide concerning use of scanf/sscanf -!
-e_medium_read=_formatted_read
-e_medium_read_P=_formatted_read_P
-! Suppress one warning which is not relevant for this processor -!
-w29
-! Load the small 'C' library for processor option_v1-!
cl1s
```

## Reference

IAR C-Compiler Users Guide.
IAR Assembler Users Guide, XLINK section
AVR Microcontroller data book May 1997