

## Topics

<b>2</b>	<b>Introduction to Common Object File Format</b>	<b>2-3</b>
2.1	Sections	2-4
2.2	How the Assembler Handles Sections	2-6
2.2.1	Uninitialized Sections	2-6
2.2.2	Initialized Sections	2-7
2.2.3	Named Sections	2-7
2.2.4	Section Program Counters	2-8
2.2.5	An Example That Uses Sections Directives	2-8
2.3	How the Linker Handles Sections	2-11
2.3.1	Default Memory Allocation	2-12
2.3.2	Placing Sections in the Memory Map	2-13
2.4	Relocation	2-14
2.5	Runtime Relocation	2-15
2.6	Loading a Program	2-16
2.7	Symbols in a COFF File	2-17
2.7.1	External Symbols	2-17
2.7.2	The Symbol Table	2-17

## Figures

<b>Fig.</b>	<b>Title</b>	<b>Page</b>
2.1	Partitioning Memory Into Logical Blocks	2-5
2.2	Using Sections Directives	2-9
2.3	Generated Object Code according to previous source code example	2-10
2.4	Combining Input Sections to Form an Executable Object Module	2-12
2.5	An Example of Code That Generates Relocation Entries	2-14

## Notes

<b>Title</b>	<b>Page</b>
2.1	
Default Section Directive	2-6



## 2 Introduction to Common Object File Format

The assembler and linker create object files that can be executed by a MSP430 device. The format that these object files are in is called *common object file format* (COFF).

COFF makes modular programming easier because it encourages you to think in terms of *blocks* of code and data when you write an assembly language program. These blocks are known as **sections**. Both the assembler and the linker provide directives that allow you to create and manipulate sections.

For more information about COFF object file structure refer to the Appendix.

## 2.1 Sections

The smallest unit of an object file is called a **section**. A section is a block of code or data that will ultimately occupy contiguous space in the MSP430 memory map. Each section of an object file is separate and distinct from the other sections. COFF object files always contain three default sections:

<b>.text section</b>	usually contains executable code.
<b>.data section</b>	usually contains initialized data.
<b>.bss section</b>	usually reserves space for uninitialized variables.

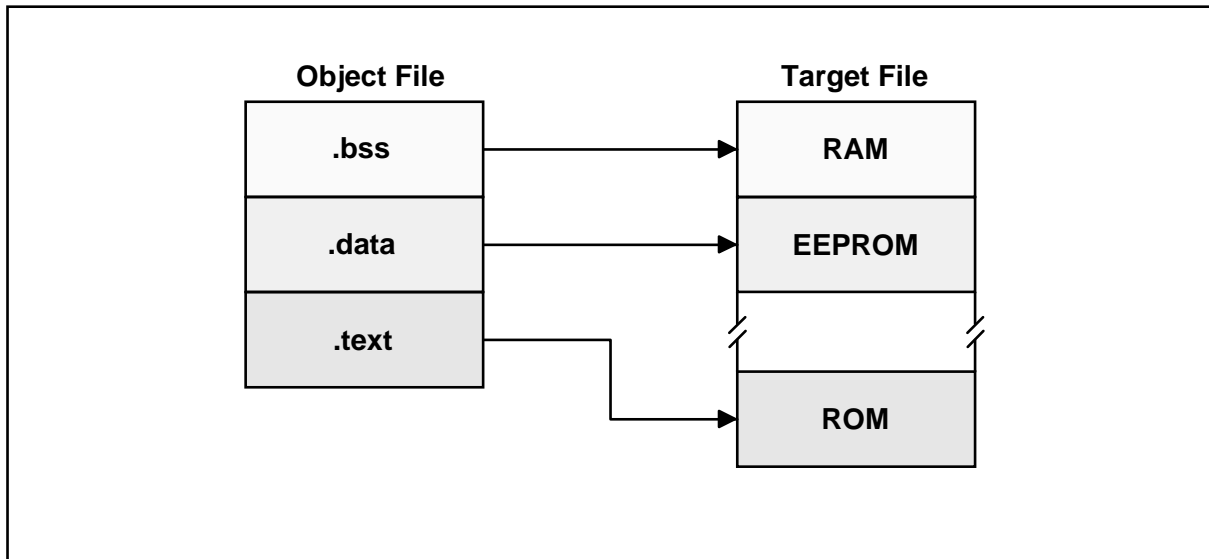
In addition, the assembler and linker allow you to create, name, and link **named** sections that are used like the .data, .text, and .bss sections.

It is important to note that there are two basic types of sections:

<b>Initialized sections</b>	contain data or code. The .text and .data sections are initialized; named sections created with the .sect assembler directive are also initialized.
<b>Uninitialized sections</b>	reserve space in the memory map for uninitialized data. The .bss section is uninitialized; named sections created with the .usect, reg, and .regvar assembler directive are also uninitialized.

The assembler provides several directives that allow you to associate various portions of code and data with the appropriate sections. The assembler builds these sections during the assembly process, creating an object file that is organized like the object file shown in the following figure.

One of the linker's functions is to relocate sections into the target memory map; this is called **allocation**. Because most systems contain several different types of memory, using sections can help you to use target memory more efficiently. All sections are independently relocatable; you can place different sections into various blocks of target memory. For example, you can define a section that contains an initialization routine and then allocate the routine into a portion of the memory map that contains ROM.



**Figure 2.1:** Partitioning Memory Into Logical Blocks

## 2.2 How the Assembler Handles Sections

The assembler's main function related to sections is to identify the portions of an assembly language program that belong in a particular section. The assembler has seven directives that support this function:

- **.bss**
- **.data**
- **.sect**
- **.text**
- **.usect**

The **.bss** and **.usect** directives create *uninitialized sections*; the **.text**, **.data**, and **.sect** directives create *initialized sections*.

**Note: Default Section Directive**

If you don't use any of the sections directives, the assembler assembles everything into the **.text** section.

### 2.2.1 Uninitialized Sections

Uninitialized sections reserve space in MSP430 memory; they are usually allocated into RAM. These sections have no actual contents in the object file; they simply reserve memory. A program can use this space at runtime for creating and storing variables.

Uninitialized data areas are built by using the **.bss** and **.usect** assembler directives. The **.bss** directive reserves space in the **.bss** section. The **.usect** directive reserves space in a specific uninitialized named section. If the section name is specified, the space is reserved in the named section. Each time you invoke one of these directives, the assembler reserves more space in the appropriate section.

The syntaxes for these directives are:

**.bss** *name* [,*size in bytes*]

*symbol* **.usect** "*section name*", *size in byte*

*symbol* points to the first byte reserved by this invocation of the **.bss** or **.usect** directive. The symbol corresponds to the name of the variable that you're reserving space for. It can be referenced by any other section and can also be declared as a global symbol (with the **.global** assembler directive).

*size* is an absolute expression. The **.bss** directive reserves *size* bytes in the **.bss** section; the **.usect** directive reserves *size* bytes in *section name*. If the section name is specified, the space is reserved in the named section. The default size for **.bss** is one byte.

*section name* tells the assembler which named section to reserve space in.

The `.text`, `.data`, and `.sect` directives tell the assembler to stop assembling into the current section and begin assembling into the indicated section. The `.bss` and `.usect`, however, **do not** end the current section and begin a new one; they simply “escape” from the current section temporarily. The `.bss` and `.usect` directives can appear anywhere in an initialized section without affecting the contents of the initialized section.

### 2.2.2 Initialized Sections

Initialized sections contain executable code or initialized data. The contents of these sections are stored in the object file and placed in MSP430 memory when the program is loaded. Each initialized section is separately relocatable and may reference symbols that are defined in other sections. The linker automatically resolves these section–relative references.

Three directives tell the assembler to place code or data into a section. The syntaxes for these directives are:

**`.text`**

**`.data`**

**`.sect`** *“section name”*

When the assembler encounters one of these directives, it stops assembling into the current section (acting as an implied “end current section” command). It then assembles subsequent code into the respective section until it encounters another `.text`, `.data`, or `.sect` directive.

Sections are built up through an iterative process. For example, when the assembler *first* encounters a `.data` directive, the `.data` section is empty. The statements following this first `.data` directive are assembled into the `.data` section (until the assembler encounters a `.text` or `.sect` directive). If the assembler encounters subsequent `.data` directives, it *adds* the statements following these `.data` directives to the statements that are already in the `.data` section. This creates a single `.data` section that can be allocated contiguously into memory.

### 2.2.3 Named Sections

Named sections are sections that **you** create. You can use them like the default `.text`, `.data`, and `.bss` sections, but they are assembled separately from the default sections.

For example, repeated use of the `.text` directive builds up a single `.text` section in the object file. When linked, this `.text` section is allocated into memory as a single unit. Suppose there is a portion of executable code (perhaps an initialization routine) that you don't want allocated with `.text`. If you assemble this segment of code into a named section, it will be assembled separately from `.text`, and you will be able to allocate it into memory separately from `.text`. Note that you can also assemble initialized data that is separate from the `.data` section, and you can reserve space for uninitialized variables that is separate from the `.bss` section.

Two directives let you create named sections:

- The **.usect** directive creates sections that are used like the `.bss` section. These sections reserve space in RAM for variables.
- The **.sect** directive creates sections that can contain code or data, similar to the default `.text` and `.data` sections. The `.sect` directive creates named sections with *relocatable* addresses.

The syntaxes for these directives are:

```
symbol .usect "section name", size  
      .sect "section name"
```

The *section name* parameter is the name of the section. Section names are significant to 8 characters. You can create up to 32,767 separate named sections.

Each time you invoke one of these directives with a new name, you create a new named section. Each time you invoke one of these directives with a name that was already used, the assembler assembles code or data (or reserves space) into the section with that name. *You cannot use the same names with different directives.* That is, you cannot create a section with the `.usect` directive and then try to use the same section with `.sect`.

#### 2.2.4 Section Program Counters

The assembler maintains a separate program counter *for each section*. These program counters are known as *section program counters*, or **SPCs**.

An SPC represents the current address within a section of code or data. Initially, the assembler sets each SPC to 0. As the assembler fills a section with code or data, it increments the appropriate SPC. If you *resume* assembling into a section, the assembler remembers the appropriate SPC's previous value and continues incrementing the SPC at that point.

The assembler treats each section as if it begins at address 0; the linker relocates each section according to its final location in the memory map.

#### 2.2.5 An Example That Uses Sections Directives

The figure on the next page shows how you can build COFF sections incrementally, using the sections directives to swap back and forth between the different sections. You can use sections directives to:

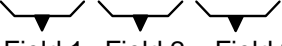
- Begin assembling into a section for the first time.
- Continue assembling into a section that already contains code. In this case, the assembler simply appends the new code to the code that is already in the section.




The SPCs are modified during assembly. A line in a listing file has four fields:

- Field 1** contains the source code line counter.
- Field 2** contains the section program counter.
- Field 3** contains the object code.
- Field 4** contains the original source statement.

1			*****
2			** Assemble an initialized table into .data **
3			*****
4	0000		.data
5	0000	0011	coeff .word 011h, 022h
	0002	0022	
6			
7			*****
8			** Reserve space in .bss for a variable **
9			*****
10			
11	0000		.bss buffer, 10
12			*****
13			** Still in .data **
14			*****
15			
16	0004	0123	ptr .word 0123h
17			*****
18			** Assemble code into the .text section **
19			*****
20			
21	0000		.text
22	0000	5504	addl add R5,R4
23	0002	4304	clr R4
24	0004	'23fd	jnz addl
25			*****
26			** Assemble more data into the .data section**
27			*****
28			
29	0006		.data
30	0006	00aa	ivals .word 0aah, 0bbh
	0008	00bb	
31			*****
32			** define another section for more variables**
33			*****
34			
35	0000		var2 .usect "newvars",1
36	0001		inbuf .usect "newvars",7
37			*****
38			** Assembler more code into .text **
39			*****
40			
41	0006		.text
42	0006	4524	acode mov @R5, R4
43	0008	4407	mov R4, R7
44			*****
45			** Define a named section for int. vectors **
46			*****
47			
48	0000		.sect "vectors"
49	0000	'0000	.word addl, acode
	0002	'0006	



Field 1   Field 2   Field 3



Field 4

**Figure 2.2:** Using Sections Directives

The listing file creates five sections:

- .text** contains 10 bytes of object code.
- .data** contains 10 bytes of object code.
- vectors** is a named section created with the `.sect` directive; it contains 4 bytes of initialized data.
- .bss** reserves 10 bytes in memory.
- newvars** is a named section created with the `.usect` directive; it reserves 8 bytes in memory.

The second column shows the object code that is assembled into these sections; the first column shows the source statements that generated the object code.

Line Numbers	Object Code	Section
22 23 24 42 43	5504 4304 23FD 4524 4407	.text
5 5 16 30 30	0011 0022 0123 00AA 00BB	.data
49 49	0000 0006	vectors
11	No data 10 bytes reserved	.bss
35 36	No data 8 bytes reserved	newvars

**Figure 2.3:** Generated Object Code according to previous source code example

### 2.3 How the Linker Handles Sections

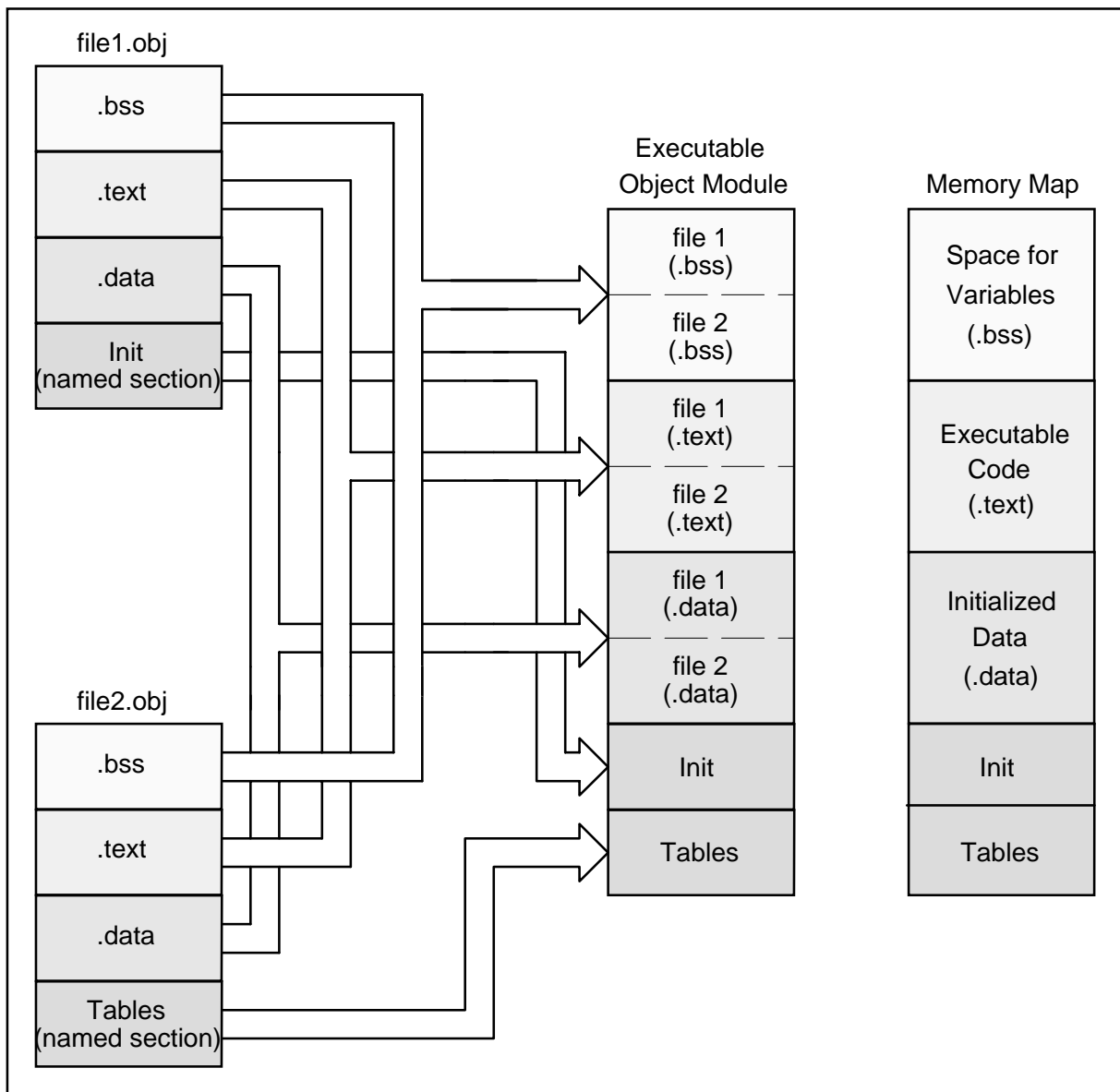
The linker has two main functions related to sections. First, the linker uses the sections in COFF object files as building blocks; it combines input sections (when more than one file is being linked) to create output sections in an executable COFF output module. Second, the linker chooses memory addresses for the output sections.

The linker provides two directives that support these functions:

- The **MEMORY directive** allows you to define the memory map of a target system. You can name portions of memory and specify their starting addresses and their lengths.
- The **SECTIONS directive** tells the linker how to combine input sections and where to place the output sections in memory.

It is not always necessary to use linker directives. If you don't use them, the linker uses the target processor's default allocation algorithm. When you *do* use linker directives, you must specify them in a linker command file.

## 2.3.1 Default Memory Allocation



**Figure 2.4:** Combining Input Sections to Form an Executable Object Module

In the figure, file1.obj and file2.obj have been assembled to be used as linker input. Each contains the .text, .data, and .bss default sections; in addition, each contains a named section. The executable output module shows the combined sections. The linker combines file1.text with file2.text to form one .text section, then combines the .data sections, then the .bss sections, and finally places the named sections at the end. The memory map shows how the sections are put into memory.

### 2.3.2 Placing Sections in the Memory Map

The figure also illustrates the linker's default methods for combining sections. Sometimes you may not want to use the default setup. For example, you may not want all of the .text sections to be combined into a single .text section. Or you might want a named section placed where the .data section would normally be allocated. Most memory maps comprise various types of memories (RAM, ROM, EPROM, etc.) in varying amounts; you may want to place a section in a particular type of memory.

- The **MEMORY** directive allows you to define the memory map for your particular system.
- The **SECTIONS** directive lets you build sections and place them into memory.

## 2.4 Relocation

The assembler treats each section as if it began at address 0. All relocatable symbols (labels) are relative to address 0 in their sections. Of course, all sections can't actually begin at address 0 in memory, so the linker **relocates** sections by:

- allocating sections into the memory map so that they begin at the appropriate address.
- adjusting symbol values to correspond to the new section addresses.
- adjusting references to relocated symbols to reflect the adjusted symbol values.

The linker uses *relocation entries* to adjust references to symbol values. The assembler creates a relocation entry each time a relocatable symbol is referenced. The linker then uses these entries to patch the references after the symbols are relocated.

```

1                               .global x
2                               .text
3 0000    !40300000             br    #x      ;uses an external relocation
4 0004    '12B00008             call   #y      ;uses an internal relocation
5 0008    5504                  y:    add    R5, R4 ;defines internal relocation

```

**Figure 2.5:** An Example of Code That Generates Relocation Entries

Both symbols *x* and *y* are relocatable. *y* is defined in the *.text* section of this module; *x* is defined in some other module. When the code is assembled, *x* has a value of 0 (the assembler assumes all undefined external symbols have values of 0), and *y* has a value of 8 (relative to address 0 in the *.text* section). The assembler generates two relocation entries, one for *x* and one for *y*. The reference to *x* is an external reference (indicated by the *!* character in the listing). The reference to *y* is to an internally defined relocatable symbol (indicated by the *'* character in the listing).

After the code is linked, suppose that *x* is relocated to address 7100h. Suppose also that the *.text* section is relocated to begin at address 7200h; *y* now has a relocated value of 7208h. The linker uses the two relocation entries to patch the two references in the object code:

```

40300000    br    #x    becomes 40307100
12B00008    call   #y    becomes 12B07208

```

Each section in a COFF object file has a table of relocation entries. The table contains one relocation entry for each relocatable reference in the section. The linker usually removes relocation entries after it uses them. This prevents the output file from being relocated again (if it is relinked or when it is loaded). A file that contains no relocation entries is an *absolute* file (all its addresses are absolute addresses). If you want the linker to retain relocation entries, invoke the linker with the *-r* option.

## 2.5 Runtime Relocation

It may be necessary or desirable at times to load code into one area of memory and run it in another. For example, you may have performance-critical code in a ROM-based system. The code must be loaded into ROM but would run much faster if it were in RAM.

The linker provides a simple way to specify this. In the `SECTIONS` directive, you can optionally direct the linker to allocate a section twice: once to set its load address, and again to set its run address.

Use the *load* keyword for the load address and the *run* keyword for the run address.

The load address determines where a loader will place the raw data for the section. Any references to the section (such as labels in it) refer to its run address. The application must copy the section from its load address to its run address; this does **not** happen automatically just because you specify a separate run address.

If you provide only one allocation (either load or run) for a section, the section is allocated only once and will load and run at the same address. If you provide both allocations, the section is actually allocated as if it were two different sections of the same size.

Uninitialized sections (such as `.bss`) are not loaded, so the only significant address is the run address. The linker allocates uninitialized sections only once: if you specify both run and load addresses, the linker warns you and ignores the load address.

## 2.6 Loading a Program

The linker produces executable COFF object modules. An executable object file has the same COFF format as object files that are used as linker input; however, the sections in an executable object file are combined and relocated to fit into target memory.

In order to run a program, the data in the executable object module must be transferred, or **loaded**, into target system memory.

Several methods can be used for loading a program, depending on the execution environment. Some of the more common situations are listed below.

- The MSP430 development tools (In-Circuit-Emulator and Evaluation Module) provide COFF object module loading capabilities.
- You can use the object format converter (the rom430, which is shipped as part of the assembly language package) to convert the executable COFF object module into one of several object file formats. You can then use the converted file with almost any EPROM programmer to burn the program into an EPROM.



## 2.7 Symbols in a COFF File

A COFF file contains a symbol table that stores information about symbols in the program. The linker uses this table when it performs relocation. Debugging tools can also use the symbol table to provide symbolic debugging.

### 2.7.1 External Symbols

External symbols are symbols that are defined in one module and referenced in another module. You can use the **.def**, **.ref**, or **.global** directives to identify symbols as external:

Defined (.def)	Defined in the current module and used in another module
Referenced (.ref)	Referenced in the current module, but defined in another module
Global (.global)	May be either of the above

The following code segment illustrates these definitions.

```
x:    ADD        #56h, R4    ;    Define x
      BR         #y         ;    Reference y
      .global    x          ;    DEF of x
      .global    y          ;    REF of y
```

The **.global** definition of **x** says that it is an external symbol defined in this module and that other modules can reference **x**. The **.global** definition of **y** says that it is an undefined symbol that is defined in some other module.

The assembler places both **x** and **y** in the object file's symbol table. When the file is linked with other object files, the entry for **x** defines unresolved references to **x** from other files. The entry for **y** causes the linker to look through the symbol tables of other files for **y**'s definition.

The linker must match all references with corresponding definitions. If the linker cannot find a symbol's definition, it prints an error message about the unresolved reference. This type of error prevents the linker from creating an executable object module.

### 2.7.2 The Symbol Table

The assembler always generates an entry in the symbol table when it encounters an external symbol (both definitions and references). The assembler also creates special symbols that point to the beginning of each section; the linker uses these symbols to relocate references to other symbols in a section.

The assembler does not usually create symbol table entries for any other type of symbol, because the linker does not use them. For example, labels are not included in the symbol table unless they are declared with **.global**. For symbolic debugging purposes, it is sometimes useful to have entries in the symbol table for each symbol in a program. To accomplish this, invoke the assembler with the **-s** option.

