Embedded Programming in C for an ARM7

- ✓ **Brief Introduction to C**
- ✓ **Directives**
- ✓ **Embedded Program in C**

Brief Introduction to C

Why C?

- C is popular
- C influenced many languages
- C is considered close-to-machine
  - Language of choice when careful coordination and control is required
  - Straightforward behavior (typically)
- Typically used to program low-level software (with some assembly)
  - Drivers, runtime systems, operating systems, schedulers, …

Introduction to C

- C is a high-level language
  - Abstracts hardware
  - Expressive
  - Readable
  - Analyzable
- C is a *procedural language*
  - The programmer explicitly specifies steps
  - Program composed of procedures
    - Functions/subroutines
- C is compiled (not interpreted)
  - Code is analyzed as a whole (not line by line)

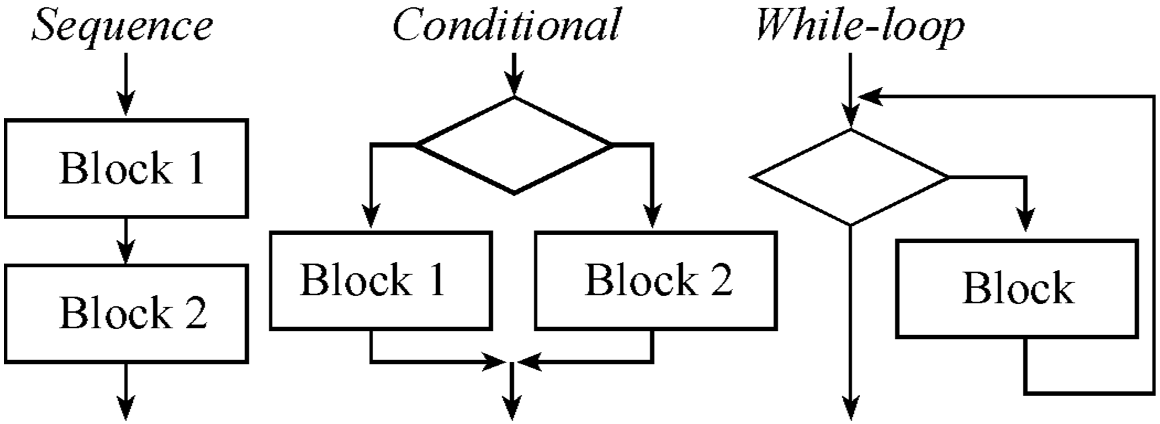
Introduction to C

- Program structure
  - Subroutines and functions
- Variables and types
- Statements
- Preprocessor

- DEMO

C Program (demo)

- ❑ Preprocessor directives (e.g., constants)
- ❑ Variables
- ❑ Functions
- ❑ Statements
- ❑ Expressions
- ❑ Names
- ❑ Operators
- ❑ Comments
- ❑ Syntax



Important Notes

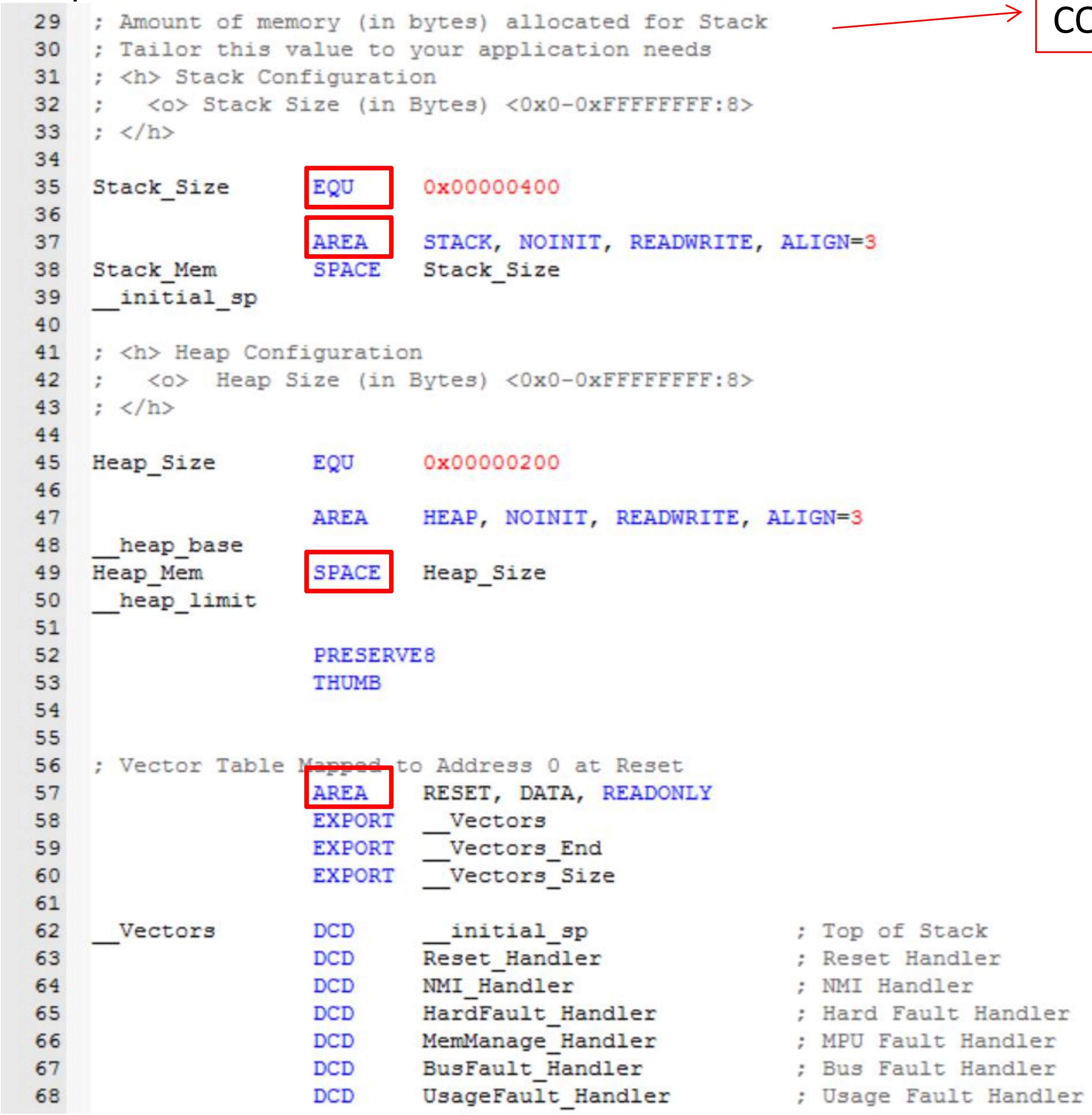
- C comes with a lot of “built-in” functions
  - printf() is one good example
  - Definition included in *header files*
  - #include<header_file.h>

- C has one special function called *main*()
  - This is where execution starts (reset vector)

- C development process
  - Compiler translates C code into assembly code
  - Assembler (e.g. built into uVision4) translates assembly code into object code
  - Object code runs on machine

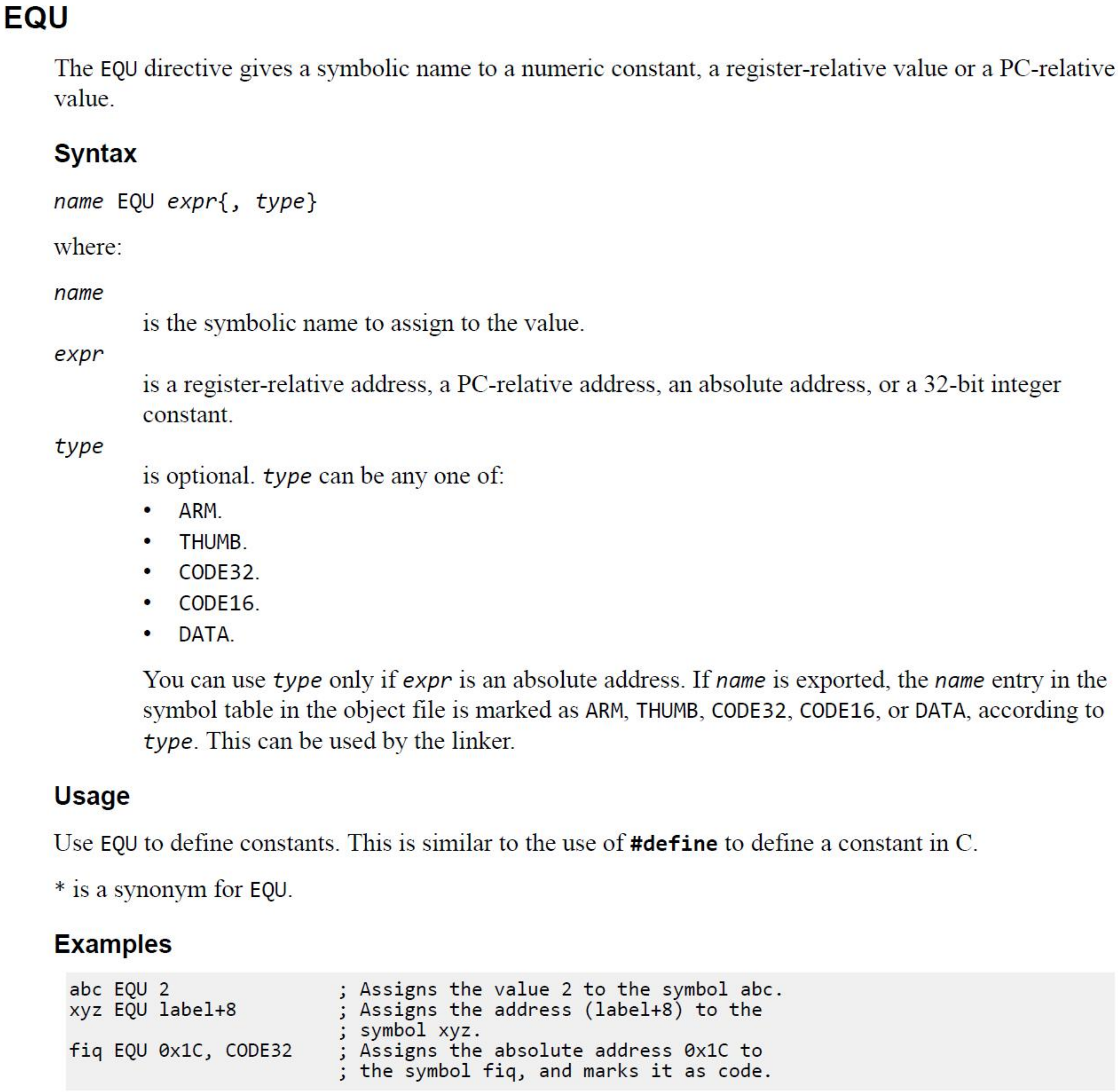
Directives

startup.s



```
; Amount of memory (in bytes) allocated for Stack
; Tailor this value to your application needs
; <h> Stack Configuration
;   <o> Stack Size (in Bytes) <0x0-0xFFFFFFFF:8>
; </h>

Stack_Size      EQU     0x00000400

                AREA    STACK, NOINIT, READWRITE, ALIGN=3
Stack_Mem       SPACE   Stack_Size
__initial_sp

; <h> Heap Configuration
;   <o>  Heap Size (in Bytes) <0x0-0xFFFFFFFF:8>
; </h>

Heap_Size       EQU     0x00000200

                AREA    HEAP, NOINIT, READWRITE, ALIGN=3
__heap_base
Heap_Mem        SPACE   Heap_Size
__heap_limit

                PRESERVE8
                THUMB


; Vector Table Mapped to Address 0 at Reset
                AREA    RESET, DATA, READONLY
                EXPORT  __Vectors
                EXPORT  __Vectors_End
                EXPORT  __Vectors_Size

__Vectors       DCD     __initial_sp               ; Top of Stack
                DCD     Reset_Handler              ; Reset Handler
                DCD     NMI_Handler                ; NMI Handler
                DCD     HardFault_Handler          ; Hard Fault Handler
                DCD     MemManage_Handler          ; MPU Fault Handler
                DCD     BusFault_Handler           ; Bus Fault Handler
                DCD     UsageFault_Handler         ; Usage Fault Handler
```

CORTEX-M3



EQU

The EQU directive gives a symbolic name to a numeric constant, a register-relative value or a PC-relative value.

Syntax

name EQU *expr*{, *type*}

where:

name

is the symbolic name to assign to the value.

expr

is a register-relative address, a PC-relative address, an absolute address, or a 32-bit integer constant.

type

is optional. *type* can be any one of:

- ARM.
- THUMB.
- CODE32.
- CODE16.
- DATA.

You can use *type* only if *expr* is an absolute address. If *name* is exported, the *name* entry in the symbol table in the object file is marked as ARM, THUMB, CODE32, CODE16, or DATA, according to *type*. This can be used by the linker.

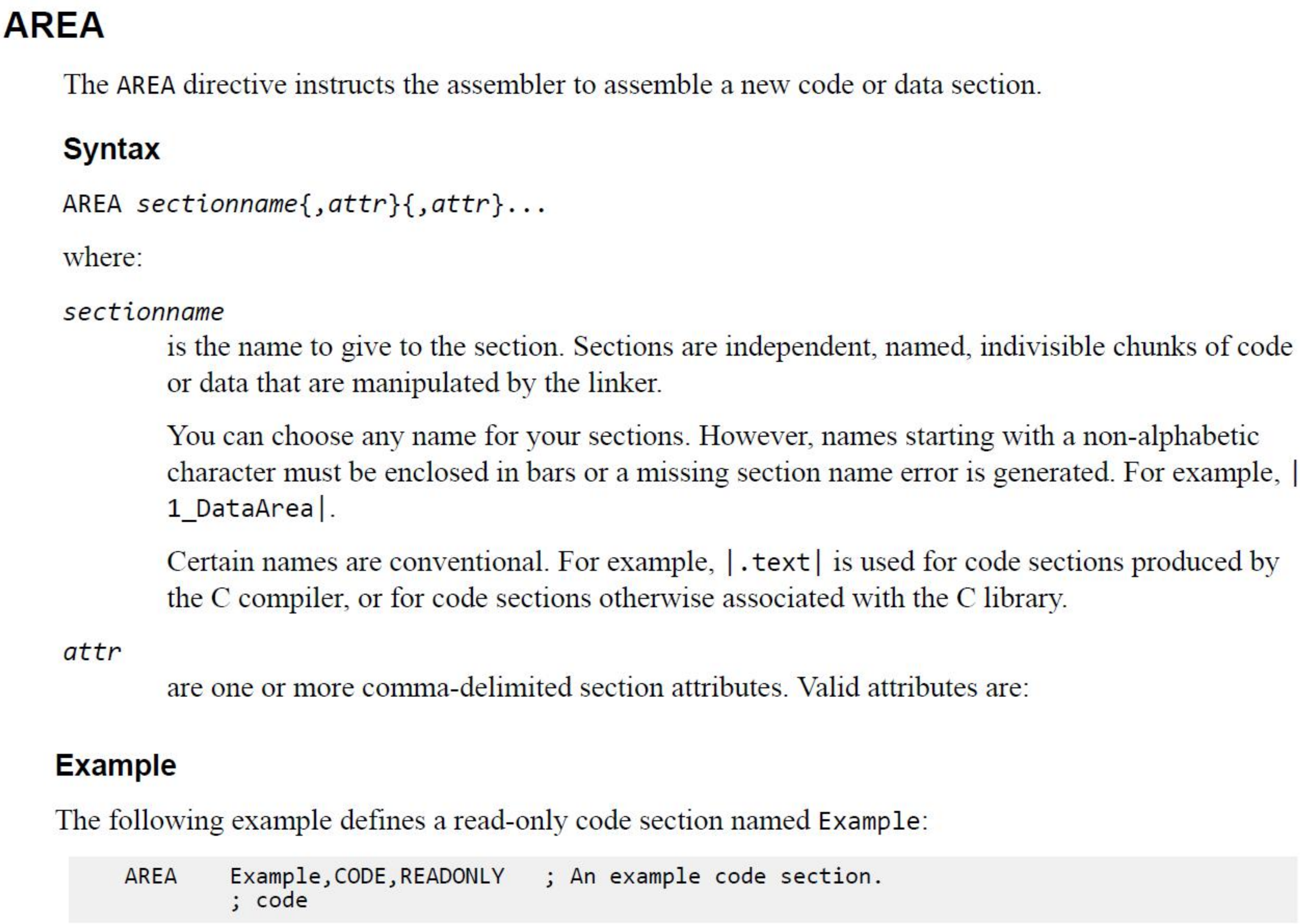
Usage

Use EQU to define constants. This is similar to the use of **#define** to define a constant in C.

* is a synonym for EQU.

Examples

```
abc EQU 2               ; Assigns the value 2 to the symbol abc.
xyz EQU label+8         ; Assigns the address (label+8) to the
                        ; symbol xyz.
fiq EQU 0x1C, CODE32    ; Assigns the absolute address 0x1C to
                        ; the symbol fiq, and marks it as code.
```



AREA

The `AREA` directive instructs the assembler to assemble a new code or data section.

Syntax

`AREA` *`sectionname{,attr}{,attr}...`*

where:

`sectionname`

is the name to give to the section. Sections are independent, named, indivisible chunks of code or data that are manipulated by the linker.

You can choose any name for your sections. However, names starting with a non-alphabetic character must be enclosed in bars or a missing section name error is generated. For example, `|1_DataArea|`.

Certain names are conventional. For example, `|.text|` is used for code sections produced by the C compiler, or for code sections otherwise associated with the C library.

`attr`

are one or more comma-delimited section attributes. Valid attributes are:

Example

The following example defines a read-only code section named `Example`:

```
    AREA    Example,CODE,READONLY   ; An example code section.
            ; code
```

SPACE or FILL

The `SPACE` directive reserves a zeroed block of memory. The `FILL` directive reserves a block of memory to fill with a given value.

Syntax

`{`*`label`*`} SPACE `*`expr`*

`{`*`label`*`} FILL `*`expr{,value{,valuesize}}`*

where:

`label`

is an optional label.

`expr`

evaluates to the number of bytes to fill or zero.

`value`

evaluates to the value to fill the reserved bytes with. *`value`* is optional and if omitted, it is 0. *`value`* must be 0 in a `NOINIT` area.

`valuesize`

is the size, in bytes, of *`value`*. It can be any of 1, 2, or 4. *`valuesize`* is optional and if omitted, it is 1.

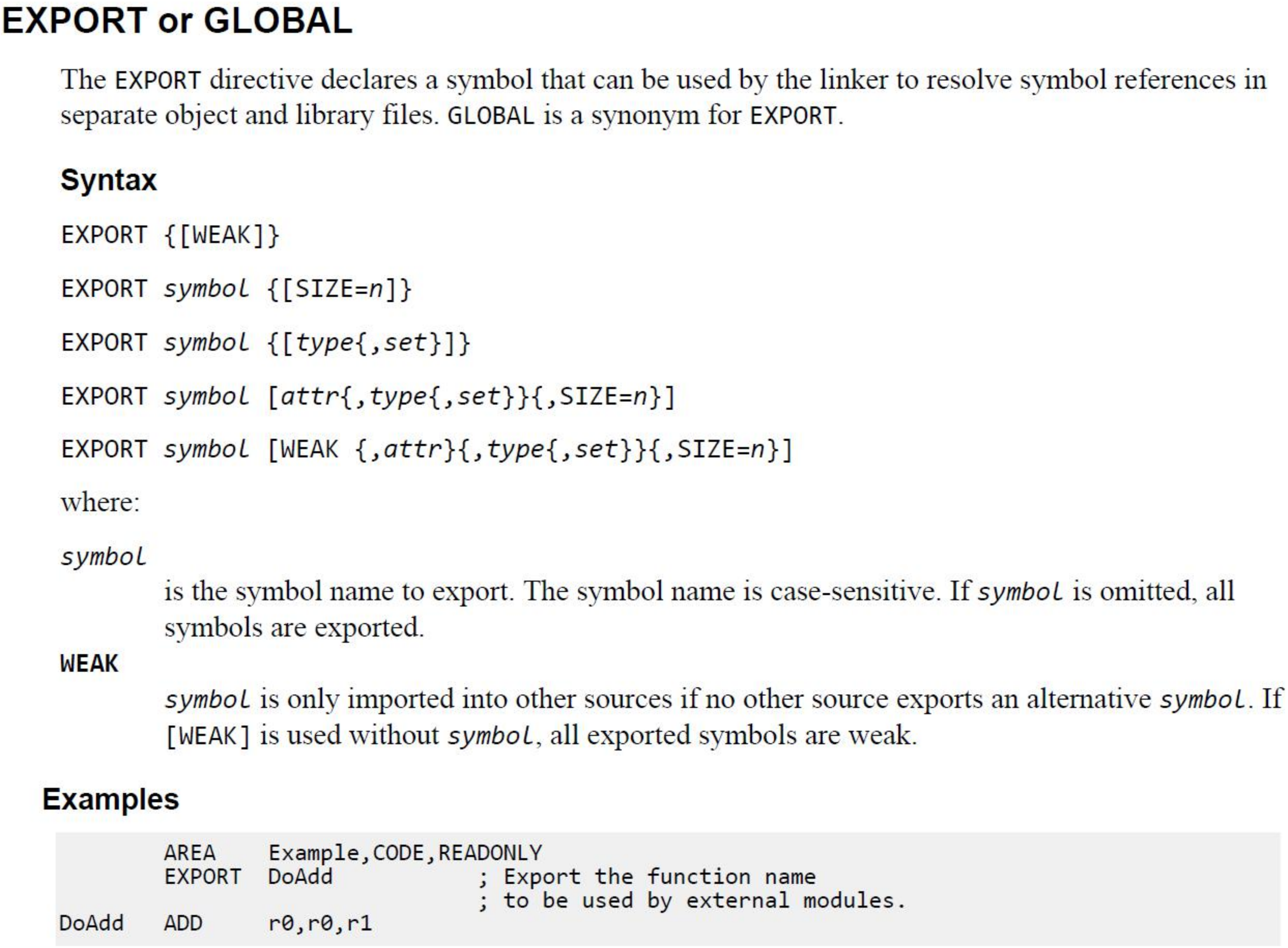
Usage

Use the `ALIGN` directive to align any code following a `SPACE` or `FILL` directive.

`%` is a synonym for `SPACE`.

Example

```
        AREA    MyData, DATA, READWRITE
data1   SPACE   255        ; defines 255 bytes of zeroed store
data2   FILL    50,0xAB,1 ; defines 50 bytes containing 0xAB
```



EXPORT or GLOBAL

The `EXPORT` directive declares a symbol that can be used by the linker to resolve symbol references in separate object and library files. `GLOBAL` is a synonym for `EXPORT`.

Syntax

`EXPORT {[WEAK]}`

`EXPORT` *`symbol`* `{[SIZE=`*`n`*`]}`

`EXPORT` *`symbol`* `{[`*`type`*`{,`*`set`*`}]}`

`EXPORT` *`symbol`* `[`*`attr`*`{,`*`type`*`{,`*`set`*`}}{,SIZE=`*`n`*`}]`

`EXPORT` *`symbol`* `[WEAK {,`*`attr`*`}{,`*`type`*`{,`*`set`*`}}{,SIZE=`*`n`*`}]`

where:

`symbol`

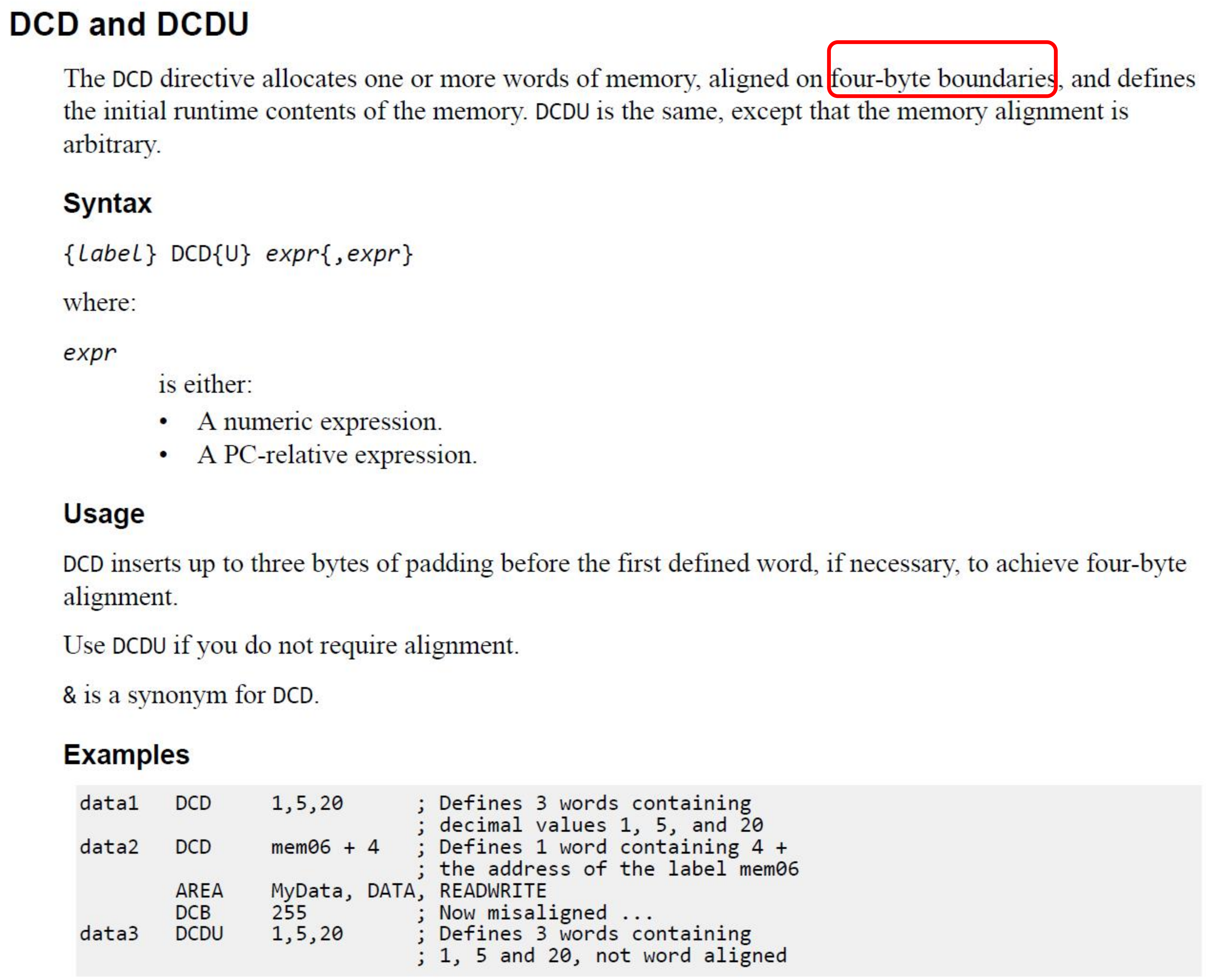
is the symbol name to export. The symbol name is case-sensitive. If *`symbol`* is omitted, all symbols are exported.

`WEAK`

`symbol` is only imported into other sources if no other source exports an alternative *`symbol`*. If `[WEAK]` is used without *`symbol`*, all exported symbols are weak.

Examples

```
        AREA    Example,CODE,READONLY
        EXPORT  DoAdd               ; Export the function name
                                    ; to be used by external modules.
DoAdd   ADD     r0,r0,r1
```



DCD and DCDU

The DCD directive allocates one or more words of memory, aligned on four-byte boundaries, and defines the initial runtime contents of the memory. DCDU is the same, except that the memory alignment is arbitrary.

Syntax

`{label} DCD{U} expr{,expr}`

where:

expr

is either:

- A numeric expression.
- A PC-relative expression.

Usage

DCD inserts up to three bytes of padding before the first defined word, if necessary, to achieve four-byte alignment.

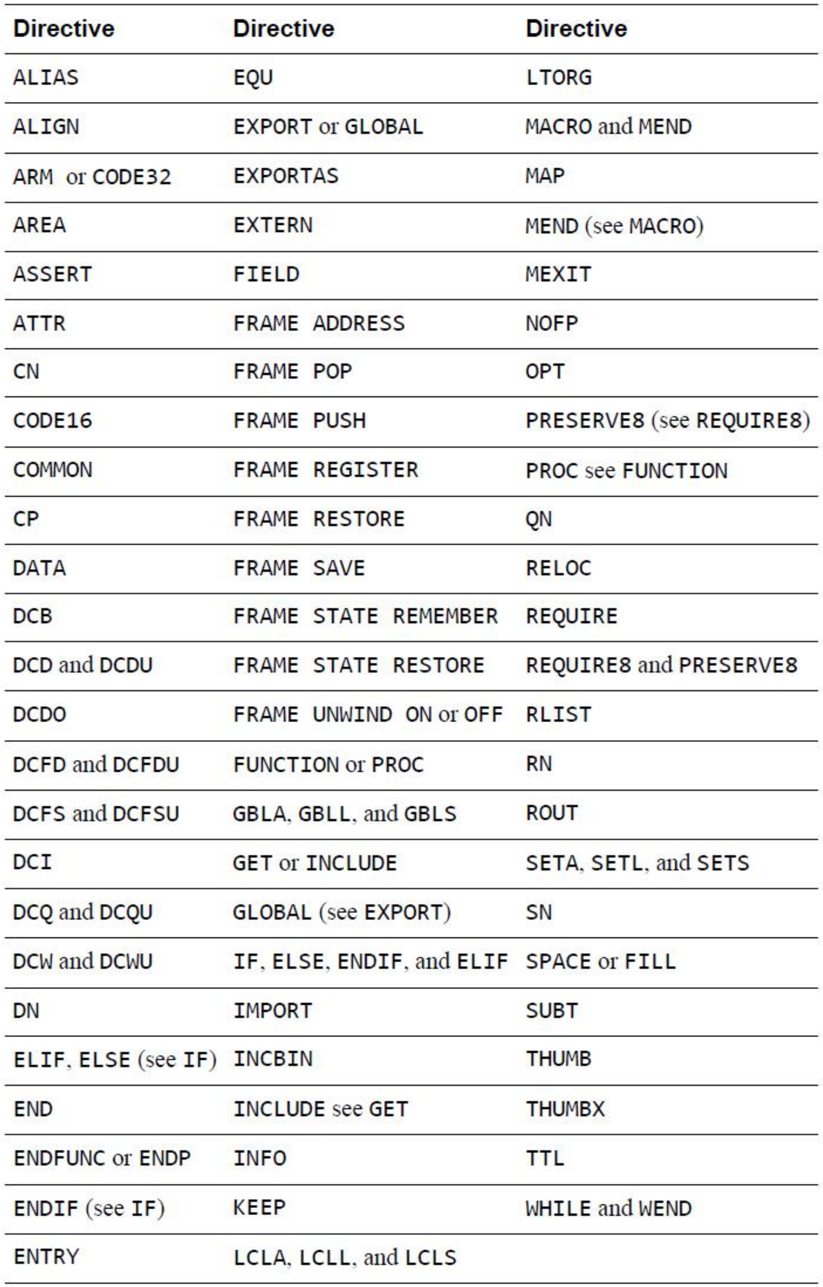
Use DCDU if you do not require alignment.

& is a synonym for DCD.

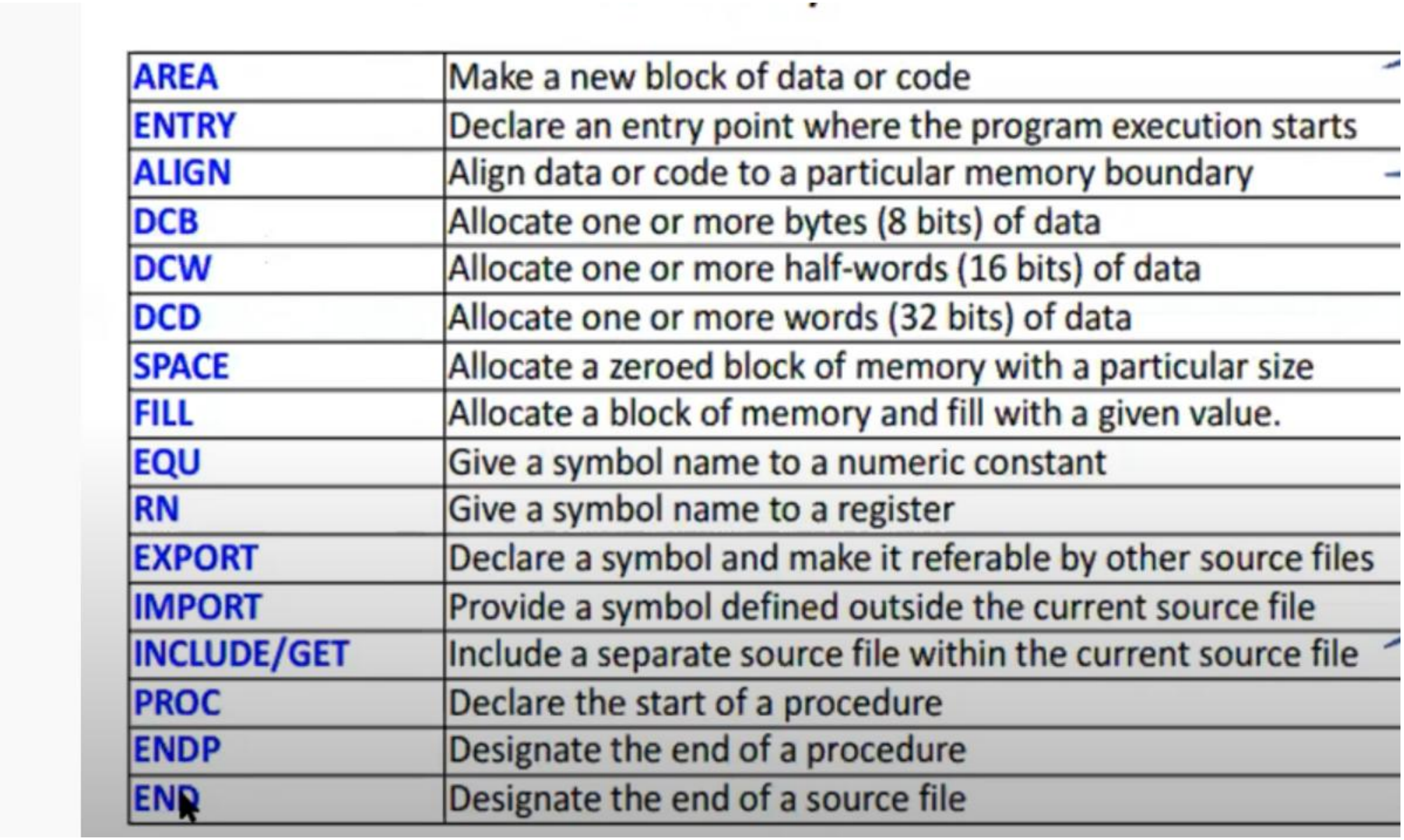
Examples

```
data1   DCD     1,5,20      ; Defines 3 words containing
                            ; decimal values 1, 5, and 20
data2   DCD     mem06 + 4   ; Defines 1 word containing 4 +
                            ; the address of the label mem06
        AREA    MyData, DATA, READWRITE
        DCB     255         ; Now misaligned ...
data3   DCDU    1,5,20      ; Defines 3 words containing
                            ; 1, 5 and 20, not word aligned
```

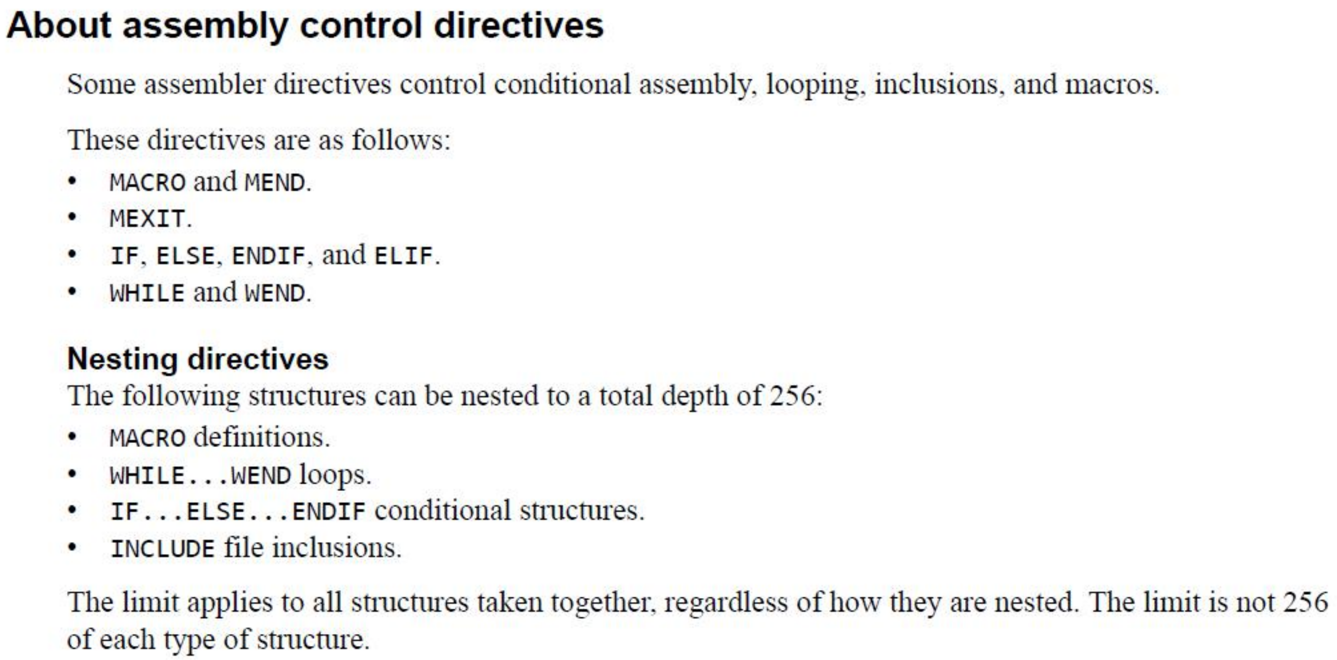
Table 12-1 List of directives



Directive	Directive	Directive
ALIAS	EQU	LTORG
ALIGN	EXPORT or GLOBAL	MACRO and MEND
ARM or CODE32	EXPORTAS	MAP
AREA	EXTERN	MEND (see MACRO)
ASSERT	FIELD	MEXIT
ATTR	FRAME ADDRESS	NOFP
CN	FRAME POP	OPT
CODE16	FRAME PUSH	PRESERVE8 (see REQUIRE8)
COMMON	FRAME REGISTER	PROC see FUNCTION
CP	FRAME RESTORE	QN
DATA	FRAME SAVE	RELOC
DCB	FRAME STATE REMEMBER	REQUIRE
DCD and DCDU	FRAME STATE RESTORE	REQUIRE8 and PRESERVE8
DCDO	FRAME UNWIND ON or OFF	RLIST
DCFD and DCFDU	FUNCTION or PROC	RN
DCFS and DCFSU	GBLA, GBLL, and GBLS	ROUT
DCI	GET or INCLUDE	SETA, SETL, and SETS
DCQ and DCQU	GLOBAL (see EXPORT)	SN
DCW and DCWU	IF, ELSE, ENDIF, and ELIF	SPACE or FILL
DN	IMPORT	SUBT
ELIF, ELSE (see IF)	INCBIN	THUMB
END	INCLUDE see GET	THUMBX
ENDFUNC or ENDP	INFO	TTL
ENDIF (see IF)	KEEP	WHILE and WEND
ENTRY	LCLA, LCLL, and LCLS	



AREA	Make a new block of data or code
ENTRY	Declare an entry point where the program execution starts
ALIGN	Align data or code to a particular memory boundary
DCB	Allocate one or more bytes (8 bits) of data
DCW	Allocate one or more half-words (16 bits) of data
DCD	Allocate one or more words (32 bits) of data
SPACE	Allocate a zeroed block of memory with a particular size
FILL	Allocate a block of memory and fill with a given value.
EQU	Give a symbol name to a numeric constant
RN	Give a symbol name to a register
EXPORT	Declare a symbol and make it referable by other source files
IMPORT	Provide a symbol defined outside the current source file
INCLUDE/GET	Include a separate source file within the current source file
PROC	Declare the start of a procedure
ENDP	Designate the end of a procedure
END	Designate the end of a source file



About assembly control directives

Some assembler directives control conditional assembly, looping, inclusions, and macros.

These directives are as follows:

- `MACRO` and `MEND`.
- `MEXIT`.
- `IF`, `ELSE`, `ENDIF`, and `ELIF`.
- `WHILE` and `WEND`.

Nesting directives

The following structures can be nested to a total depth of 256:

- `MACRO` definitions.
- `WHILE...WEND` loops.
- `IF...ELSE...ENDIF` conditional structures.
- `INCLUDE` file inclusions.

The limit applies to all structures taken together, regardless of how they are nested. The limit is not 256 of each type of structure.

Embedded Programming in C for an ARM7

Inline assembly

Keil uVision uses this syntax to embed assembly code into C programs

```
__asm {
                loop: LDR r1, [r3]
                      ADD r2, r1, #1
                      STR r2, [r3]
                      CMP r2, #76
                      BNE loop
      }
```

it is also possible to increase the performance of your code by inlining your functions. The inline keyword can be applied to any function, as shown below:

```
__inline void delay(void) {
                ....
}
```

When the inline keyword is used, the function will not be coded as a subroutine, but the function code will be **inserted** at the point where function is called, each time it is called. This removes the prologue and epilogue code which is necessary for a subroutine, making its execution time faster. However, you are duplicating the function every time it is called, so its is expensive in terms of your Flash memory.

A Simple Example

Write a program to add together 10 values from memory 0x00008000 onwards and store the summation to the memory at the address 0x00008080:

```
AREA  SUMMATION, CODE, READONLY
            ENTRY

            MOV r0, #9 ; set loop counter
            MOV r1, #0x08000 ; set address for data
            LDR r2, [r1], #4 ; load 1st value
  loop      SUBS r0, r0, #1 ; decrement counter
            LDR r3, [r1], #4 ; load next value
            ADD r2, r2, r3 ; add to running total
            BNE loop ; branch back to loop if zero flag is cleared because r0 does not hold zero.
            ;
            MOV r4, 0x00008080; set address for store
            STR  r2, [r4]; store result to memory
  idleLoop  B  idleLoop;

            END
```