# **I. Embedded C general**

## **I.1 Data types**

General data types and how they are used in automotive embedded C.

### **Primitive data types:**

|  |  |  |
| --- | --- | --- |
| **Type** | **Storage size** | **Value range** |
| char | 1 byte | -128 to 127 or 0 to 255 |
| unsigned char | 1 byte | 0 to 255 |
| signed char | 1 byte | -128 to 127 |
| int | 2 or 4 bytes | -32,768 to 32,767 or -2,147,483,648 to 2,147,483,647 |
| unsigned int | 2 or 4 bytes | 0 to 65,535 or 0 to 4,294,967,295 |
| short | 2 bytes | -32,768 to 32,767 |
| unsigned short | 2 bytes | 0 to 65,535 |
| long | 8 bytes | -9223372036854775808 to 9223372036854775807 |
| unsigned long | 8 bytes | 0 to 18446744073709551615 |
| float | 4 byte | 1.2E-38 to 3.4E+38 |
| double | 8 byte | 2.3E-308 to 1.7E+308 |
| long double | 10 byte | 3.4E-4932 to 1.1E+4932 |

Size of primitive types is variate and depend on the target platform (i.e. used compiler, processor)

Because AUTOSAR required sw app to be platform and compiler independent (help to avoid type changes when moving and software module from platform A to B):

=> AUTOSAR sw shall not use the C standardized libraries since it shall not guaranties the compatibilities with different platforms.

=> The primitive data types used shall be defined in a common header file (Platform\_Types.h), which shall be included by all source file(different platform shall provide its own platform definition type).

Ex:

typedef unsigned char boolean;  
typedef signed char sint8;  
typedef unsigned char uint8;  
typedef signed short sint16;  
typedef unsigned short uint16;  
typedef signed long sint32;  
typedef signed long long sint64;  
typedef unsigned long uint32;  
typedef unsigned long long uint64;

typedef float float32;  
typedef double float64;

### **Enumerated types:**

User define data type: used to assign names to integral constants, the names make a program easy to read and maintain.

Define enum\_tag type.

enum enum\_tag { value1 = 1, value2, value3}; // value1 = 1 to specify the value1 tag equal 1 or else the value shall be begin at the default value 0, other tag value2 if not specify shall be equal value of previous tag + 1

Create variable “var” base on “enum\_tag” type:

enum enum\_tag var = value1; // var shall has value = 1 and size of var shall be 4bytes(default type used if the target processor is 32bit type)

The size of enum type could be redefined by compiler.

Ex: Typical usage of enumerated types (define a custom types with specifics list of value).

typedef enum Std\_ReturnTypes\_ETag

{

E\_OK,

E\_NOT\_OK

} Std\_ReturnTypes;

Std\_ReturnTypes Len\_returnValue;

### **Structure types:**

User define data type: use to create a data type that can be used to group items of possibly different types into a single type.

Define struct\_tag type.

struct struct\_tag

{

int element1;

char element2;

};

Create variable “var” base on “struct\_tag” type:

struct\_tag var={100, 5};

Ex: typical usage of structure types:

1. *Define initialized configuration data structure*

typedef struct

{

uint16 CanHTHobjectsCount;

uint16 CanHRHobjectsCount;

const Can\_ControllerConfigType \*CanControllerConfiguration;

} Can\_ConfigType;

CONST(Can\_ConfigType, CAN\_CONFIG\_DATA) Can\_Config;

* Above Can\_Config constant structure store the input configuration data of Can Driver and will be input to Can\_Init API to initialize module CAN.

1. *Define group of related register for HW mapping*

/\* Receive rule register \*/

typedef struct Can\_Filter\_STag {

uint32 ulFMReg; /\* Filter mask register \*/

uint32 ulFM1Reg; /\* Filter mask register1 \*/

uint32 dummy0; /\* dummy address value \*/

uint32 ulFS1Reg; /\* Filter status register \*/

uint32 dummy1; /\* dummy address value \*/

uint32 ulFFA1Reg;

uint32 dummy2; /\* dummy address value \*/

uint32 ulFA1Reg;

uint32 dummy3[8]; /\* dummy address value \*/

uint32 ulFReg[28][2];

} Can\_Filter;

#define CAN1\_BASEADDR 0x40006400UL

#define CAN2\_BASEADDR 0x40006800UL

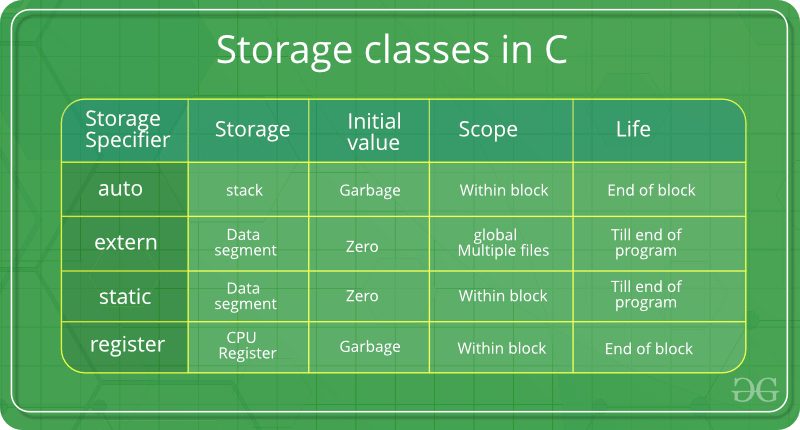
#define Can1\_Filter\_Reg (\*(volatile Can\_Filter \*)(CAN1\_BASEADDR + 500UL))

#define Can2\_Filter\_Reg (\*(volatile Can\_Filter \*)(CAN2\_BASEADDR + 500UL))

* Above structure Can1\_Filter\_Reg shall be mapped to CAN driver hardware register. To access register “Filter mask” of CanController1: Can1\_Filter\_Reg->ulFMReg = 0x0000000F; // set mask for first 4 bits of filter rule.

### **Union types:**

## **I.2 Storage class**



### **The auto Storage Class:**

- The **auto** storage class is the default storage class for all **local variables**

{

int mount; // same as auto int month

auto int month;

}

### **The register Storage Class:**

- The **register** storage class is used to define **local variables** that should be stored in a **register instead of RAM** (i.e. local variables store directly in register R0-R12(ARM arch registers) instead of STACK(RAM)).

{

register int miles;

}

### **The static Storage Class:**

The **static** storage class instructs the compiler to **keep a local variable** in existence during the life-time of the program instead of creating and destroying it each time it comes into and goes out of scope.

The **static** modifier may also be **applied to global variables**. When this is done, it causes that variable's **scope to be restricted to the file in which it is declared**.

Ex: static to retain value

{

static int miles;

}

int incfunc(void)

{

static int count = 9;

count++;

return count;

}

call incfunc 1st time return = 10, 2nd time return = 11, … value of count retain after each call of incfunc

Ex: static to restricted scope of variable and function to file(module), which it is decleared.

In file Can.h

static int global\_var1;

In file Can.c

#include “Can.h”

static void static\_func(void)

{

printf(“global\_var1 in Can.c before %d” , global\_var1)

global\_var1 = 100;

printf(“global\_var1 in Can.c after %d” , global\_var1)

}

void test\_func\_can(void)

{

static\_func();

}

In file Can\_MainFunc.c

#include “Can.h”

static void static\_func(void)

{

printf(“global\_var1 in Can\_MainFunc.c before %d” , global\_var1)

global\_var1 = 200;

printf(“global\_var1 in Can\_ MainFunc.c after %d” , global\_var1)

}

void test\_func\_can\_main(void)

{

static\_func();

}

In Main.c

int main(void)

{

test\_func\_can(); //>> print: global\_var1 in Can.c before 0

//>> print: global\_var1 in Can.c after 100

test\_func\_can\_main(); //>> print: global\_var1 in Can\_ MainFunc.c before 0

//>> print: global\_var1 in Can\_ MainFunc.c after 200

}

Above compilation shall be pass even though exist same function name “static\_func” and variable “global\_var1” in different source file because these definition is restricted in its own module file.

* there will be different instance global\_var1, static\_func in each source file Can.c and Can\_MainFunc.c

### **The extern Storage Class:**

**Extern** storage class simply tells us that the variable is defined elsewhere and not within the same block where it is used (extend the visibility of variables/functions).

extern void func(void); // is equal void func(void) : by default the declaration of a function is extern

extern int a; // declare that variable a and variable a is defined somewhere else

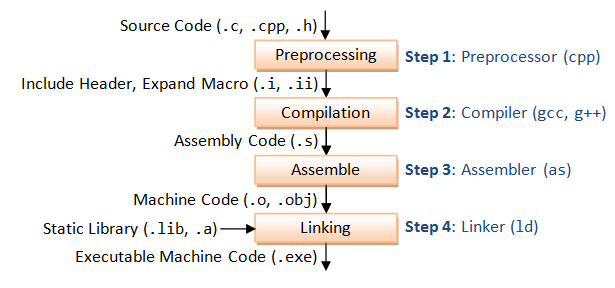
We could re-declare function/variable multiple time but only able to define function/variable only one time.

## **I.3 Array and pointer**

## **I.4 Memory**

# **II. C compilation process**

C compilation process starts with the source code(C, asm) as input and converts the source code into machine-readable code(binary). In general the process could be broken down to 4 stage: Preprocessing, Compiling, Assembling and Linking.



Ex: compliation of SampleApp.c source file

arm-none-eabi-gcc -save-temps SampleApp.c -o SampleApp.o

create SampleApp.i (source c after preprocessing), SampleApp.s (source c compiled to source assembly), SampleApp.o (source compiled to binary code).

## **II.1 Preprocessing**

This is the first phase through which source code is passed. This phase include:

* Removal of Comments (strip down all comment text)
* Expansion of Macros (handle the expansion of defined macro value or macro function or string create by #define, #undef, …)

Ex:

#define VALUE 101

#define VAR(A) {#A, VAR\_##A}

#define WARN\_OVER100(EXP) \

{ if (EXP > 100) \

/\* do something \*/ \

}

Use macro(before expansion)

WARN\_OVER100(VALUE);

VAR(my\_var);

After expansion:

{ if (101 > 100)

/\* do something \*/

}

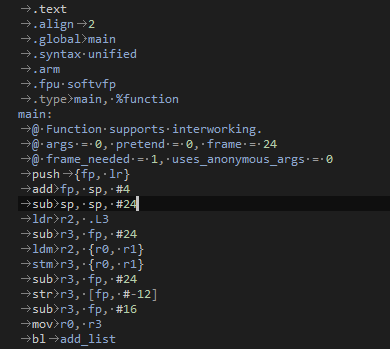
{“myvar”, VAR\_my\_var} << string and concatenated string

* Expansion of the included files (basically take the whole header file and placed where its included)
* Conditional compilation (check the #if, #else, #endif, #ifdef, #ifndef,… to enable compilation for selective code block)

## **II.2 Compilation**

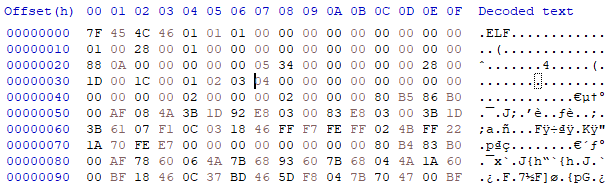
Compiler shall take the output of the preprocessor and generates assembly language, an intermediate human readable language, specific to the target processor.

* Check C program syntax
* Verification of function usage using prototype
* Generate assembly language
* Optionally optimize the translated code



## **II.3 Assembling**

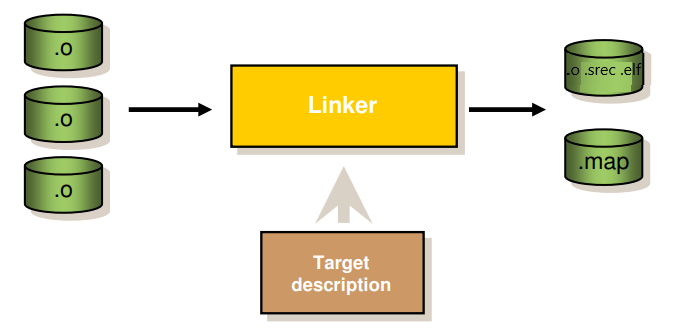
Assembler will convert the assembly code into pure binary code or machine code (object file).



## **II.4 Linking**

After compiling process is done, the linker combines a number of object and archive files (libraries), relocates their data and ties up symbol references in order to link the final application.

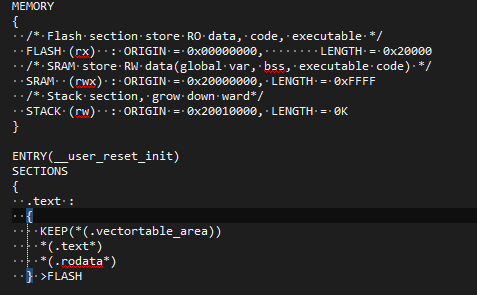
* Merging segments of code (ensure promises in compilation is kept)
* Allocate target memory (RAM, ROM, stack, special areas)
* Produce files for debugging (symbols, line numbers...)
* Produce files for target (mirror of memory)



The input file of linker shall be object files after compilation, linker script file (used to specify the target device memory section, where to place code and data).

The final output shall be executable file with additional debug information (.axf, .elf, .o,…), pure executable file (.bin, .srec,…), additional information file(.map, .lst,…).

Ex: Sample of ld script file



# **III. Thread-safe/Reentrancy/Asynchronous-Synchronous**