Yrkes
Akademin
Vi hjälper dig att lyckas!

Programming of safety critical systems

Programming and Developing of Embedded Systems

Safety Critical Systems

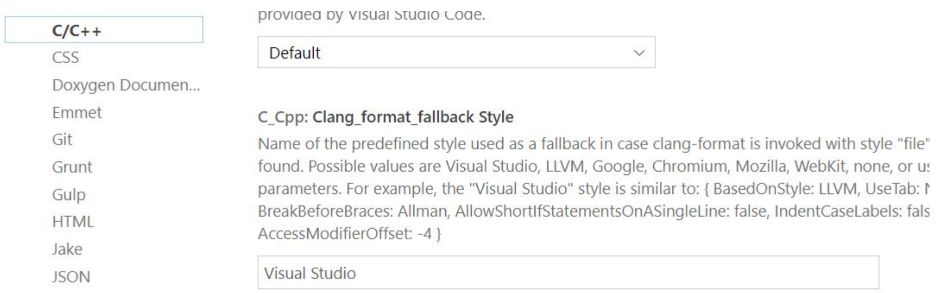
- Safety Critical Systems
  - Systems whose failure or malfunction may result in one (or more) of the following
    - Loss of life or serious injury to people
    - Damage to properties/equipments
    - Damage to the environment
- Examples of safety critical systems
  - Aircraft control systems
  - Robotic surgery machines
  - Railway signal control systems
  - Braking system in vehicles
- Safety-critical systems are increasingly computer-based

Development of Safety Critical Systems

- Creating systems that operate safely by minimizing, controlling and reducing hazards
  - And even if they fail, they are still capable of entering in a controlled safe operation mode
- Different Types of Reliability in Safety Systems
  - **Fail-operational** systems: They continue to operate even if their control systems fail
    - Remaining part of the system can complete the operation; E.g. a remote keyless system
  - **Fail-safe** systems: They become safe if they fail
    - They switch to a safe mode and usually inform an operator; E.g. Windows, Insulin pumps and etc.
  - **Fail-secure** systems: They become secure when they fail
    - Usually by locking up to minimize harm; E.g. Electronic doors, lock during power failures
  - **Fail-passive** systems: They continue to operate in the event of a system failure
    - By becoming passive and handing controls over to an operator; E.g. An aircraft autopilot
  - **Fault-tolerant** systems: They continue to operate in the event of failure
    - Usually by detecting at risk components and replace them before they result in any risk
    - E.g. cooling system in a nuclear reactor

Safety Critical Software Systems - Guidelines

- ❖ Modelling and coding guidelines
  - ➢ Enforcement of low complexity
  - ➢ Use of language subsets
  - ➢ Enforcement of strong typing
  - ➢ Use of defensive implementation techniques
  - ➢ Use of well-trusted design principles
  - ➢ Use of unambiguous graphical representation
  - ➢ Use of naming conventions
  - ➢ Use of style guides
    - ■ Like Visual Studio Style Guides for C/C++
    - ■ Activate it in Visual Studio Code



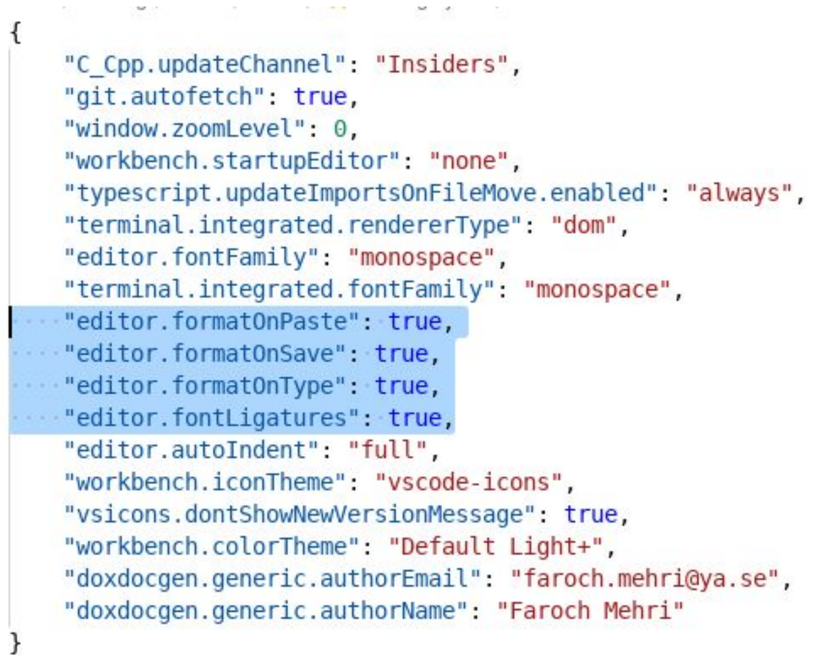
provided by Visual Studio Code.

Default

C_Cpp: **Clang_format_fallback Style**

Name of the predefined style used as a fallback in case clang-format is invoked with style "file" found. Possible values are Visual Studio, LLVM, Google, Chromium, Mozilla, WebKit, none, or us parameters. For example, the "Visual Studio" style is similar to: { BasedOnStyle: LLVM, UseTab: N BreakBeforeBraces: Allman, AllowShortIfStatementsOnASingleLine: false, IndentCaseLabels: fals AccessModifierOffset: -4 }

Visual Studio



```
{
    "C_Cpp.updateChannel": "Insiders",
    "git.autofetch": true,
    "window.zoomLevel": 0,
    "workbench.startupEditor": "none",
    "typescript.updateImportsOnFileMove.enabled": "always",
    "terminal.integrated.rendererType": "dom",
    "editor.fontFamily": "monospace",
    "terminal.integrated.fontFamily": "monospace",
    "editor.formatOnPaste": true,
    "editor.formatOnSave": true,
    "editor.formatOnType": true,
    "editor.fontLigatures": true,
    "editor.autoIndent": "full",
    "workbench.iconTheme": "vscode-icons",
    "vsicons.dontShowNewVersionMessage": true,
    "workbench.colorTheme": "Default Light+",
    "doxdocgen.generic.authorEmail": "faroch.mehri@ya.se",
    "doxdocgen.generic.authorName": "Faroch Mehri"
}
```

Safety Critical Software Systems - Guidelines

- Principles for software architectural design
  - Hierarchical structure of software components
  - Restricted size and complexity of software components
  - Restricted size of interfaces
  - Strong cohesion within each software component
  - Loose coupling between software components
  - Appropriate scheduling properties (timing property of the components)
  - Appropriate management of shared resources
  - Appropriate isolation of the software components
  - Restricted use of interrupts
  - Use of design notation, like pseudocode, flow charts and structure charts

Safety Critical Software Systems - Guidelines

- Design principles for software unit design and implementation
  - Use of semi-formal notation; like UML, timing diagram and etc.
  - One entry and one exit point in subprograms and functions
  - No dynamic objects or variables, or else online test during their creation
  - Initialization of variables
  - No multiple use of variable names
  - Avoid global variables or else justify their usage
  - Restricted use of pointers
  - No implicit type conversions
  - No hidden data flow or control flow
  - No unconditional jumps
  - No recursions

Safety Critical Software Systems - Guidelines

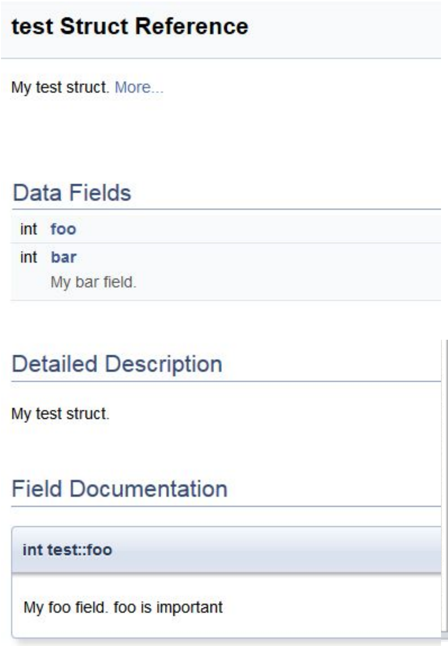
- Methods for software verification
  - Walk-through
  - Pair-programming
  - Semi-formal verification
  - Control flow and data flow analysis
  - Static code analysis; like IAR, CodeSonar, Parasoft C/C++test and etc.
  - Requirement-based test
  - Interface test
  - Fault injection test
  - Resource usage evaluation
  - Analysis of boundary values

Safety Critical Software Systems - Guidelines

- Structural coverage metrics
  - Statement coverage
  - Branch coverage
  - Modified condition/decision coverage (MC/DC)
    - Each entry and exit point is invoked
    - Each decision takes every possible outcome
    - Each condition in a decision takes every possible outcome
    - Each condition in a decision is shown to independently affect the outcome of the decision.
    - Independence of a condition is shown by proving that only one condition changes at a time.
  - Function coverage
  - Call coverage

Generate documentation from source code (Doxygen)

- Doxygen
  - The de facto standard tool for generating documentation from C, C++ and etc. source files
  - It generates the documents based on the comments in the source files
  - Install **Doxygen Documentation Generator extension** in Visual Studio Code



test Struct Reference

My test struct. More...

Data Fields

int **foo**
int **bar**
My bar field.

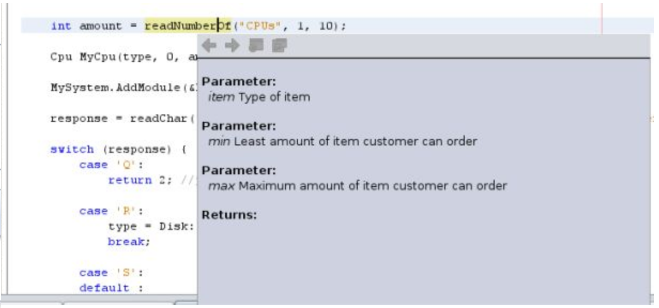
Detailed Description

My test struct.

Field Documentation

int test::foo

My foo field. foo is important



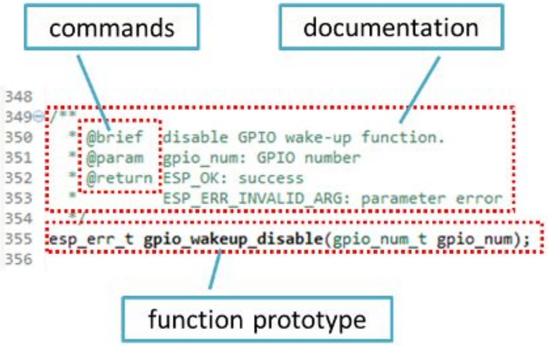
```
int amount = readNumberOf("CPUs", 1, 10);

Cpu MyCpu(type, 0, a

MySystem.AddModule(&

response = readChar(

switch (response) {
    case 'Q':
        return 2; //

    case 'R':
        type = Disk:
        break;

    case 'S':
    default :
```

Parameter:
item Type of item

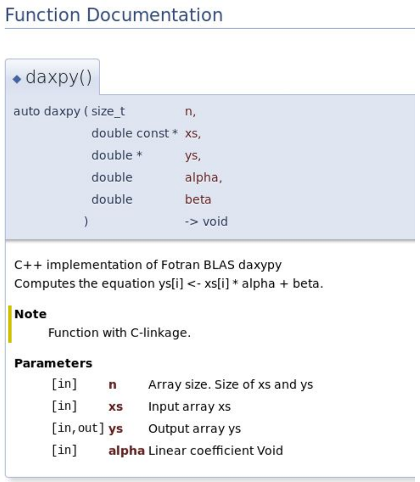
Parameter:
min Least amount of item customer can order

Parameter:
max Maximum amount of item customer can order

Returns:



```
/**
  * @brief   disable GPIO wake-up function.
  * @param   gpio_num: GPIO number
  * @return  ESP_OK: success
  *          ESP_ERR_INVALID_ARG: parameter error
  */
esp_err_t gpio_wakeup_disable(gpio_num_t gpio_num);
```



Function Documentation

◆ daxpy()

```
auto daxpy ( size_t          n,
             double const *  xs,
             double *        ys,
             double          alpha,
             double          beta
           )                 -> void
```

C++ implementation of Fotran BLAS daxypy
Computes the equation ys[i] <- xs[i] * alpha + beta.

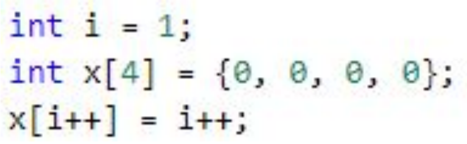
Note
Function with C-linkage.

Parameters

[in]	**n**	Array size. Size of xs and ys
[in]	**xs**	Input array xs
[in, out]	**ys**	Output array ys
[in]	**alpha**	Linear coefficient Void

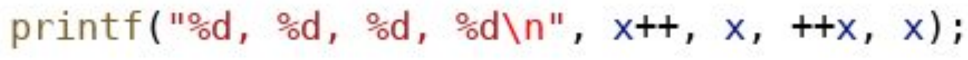
C programming - Side Effects and Sequence Points

- The order of evaluation of expressions is undefined for most of the operators in C
  - Therefore some unexpected results may occur when using certain operators
  - The results may be different with different compilers and different machines
  - The undefined behaviours can make the program unportable
  - These unexpected results are caused by **side effects**



```
int i = 1;
int x[4] = {0, 0, 0, 0};
x[i++] = i++;
```

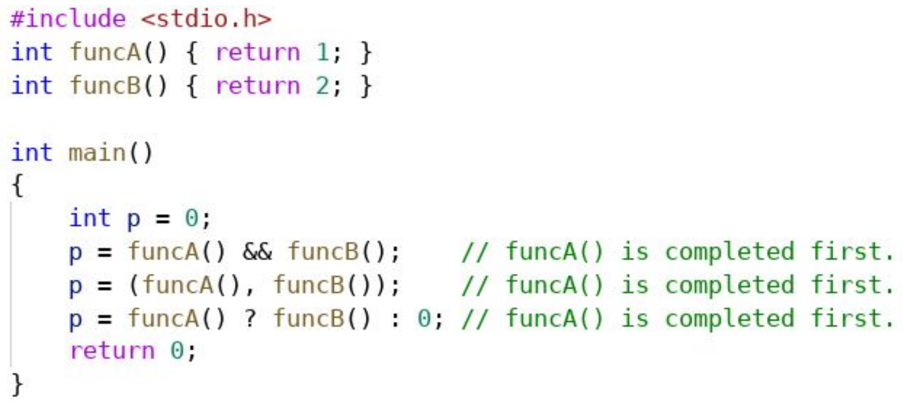
- Any operation that affects storage of an operand has a **side effect**
- Any operation has a **persistent side effect** if it does anything beyond its life
- Conflicting **side effects** are a kind of undefined behavior.
- C guarantees that all side effects of an expression will be concluded by the next **sequence point** in a program.



```
printf("%d, %d, %d, %d\n", x++, x, ++x, x);
```

C programming - Side Effects and Sequence Points

- **Sequence points** are checkpoints in a program at which the compiler ensures that operations in an expression are concluded.
- The most known sequence point is the **semicolon** marking the end of a statement.
- Other sequence points are as follows:
  - expr1, expr2 (the comma operator)
  - expr1 && expr2 (the logical AND operator)
  - expr1 || expr2 (the logical OR operator)
  - expr1 ? expr2 : expr3 (the conditional operator)

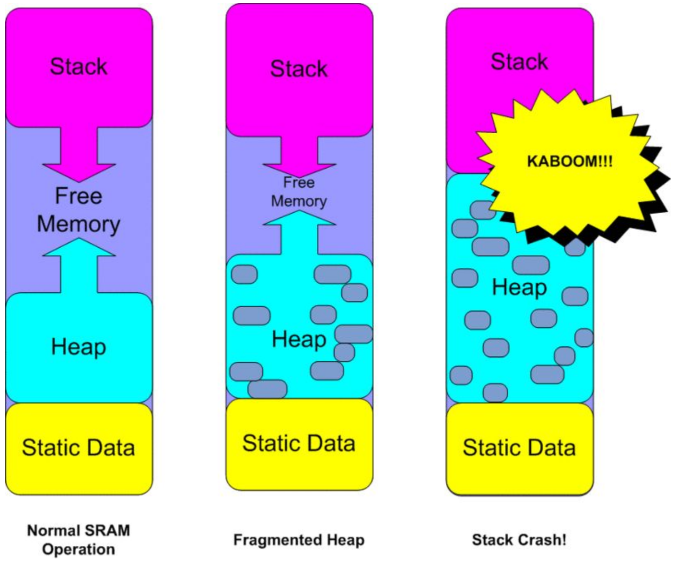


```c
#include <stdio.h>
int funcA() { return 1; }
int funcB() { return 2; }

int main()
{
    int p = 0;
    p = funcA() && funcB();    // funcA() is completed first.
    p = (funcA(), funcB());    // funcA() is completed first.
    p = funcA() ? funcB() : 0; // funcA() is completed first.
    return 0;
}
```

- Using the execution order of side effects, when none is guaranteed, is risky
- Don’t use a variable which has more than one side effect in a given statement.

C programming - Dynamic Memory Allocation & Deallocation

- The main purpose of dynamic memory allocation & deallocation (e.g. malloc & free)
  - Multiple processes can dynamically share all the available RAM memory
    - E.g. a multi-process system with a large and variable or unknown amount of memory
- The problems with dynamic memory allocation
  - It might fail
  - Memory fragmentation
  - It is slow and time taking
    - Calling into the OS to get more memory or scan free space
    - Memory management is required
  - Its execution time is non-deterministic and varying
    - In the case of real-time systems it is inacceptable



C programming - Dynamic Memory Allocation & Deallocation

- Dynamic memory allocation is avoided in embedded programming. Because
  - It can be harmful and hard to handle failures
  - Behavior of embedded systems are completely deterministic
    - Especially in the case of safety critical systems
  - You always know the amount of RAM needed for the worst case
- Alternatives for dynamic memory allocation
  - Static memory allocation
  - Static memory preallocation and implement a safe memory management
    - E.g. A Stack-based memory allocation and deallocation
  - One-time allocation at startup is generally safe; but not freeing
  - Convert multiple-instances modules to single-instances modules by determining the number of required instances and allocate memory statically for them

Safety Critical Software Systems (MISRA C Guidelines)

- Guideline categories in MISRA C
  - Mandatory guidelines - C code which is claimed to conform to MISRA C
    - Shall comply with every mandatory guideline
    - Deviation from mandatory guidelines is not permitted
  - Required guidelines - C code which is claimed to conform to MISRA C
    - Shall comply with every required guideline
    - In the case of deviation, a formal deviation is required
      - Such deviations are properly recorded and authorized
      - Individual programmers can be prevented from deviating guidelines by a formal authorization
  - Advisory guidelines
    - These are recommendations. It does not mean that these items can be ignored, but rather that they should be followed as far as they are reasonably practical.
  - *The Standard Libraries code is not required to comply with MISRA C.*

Safety Critical Software Systems (MISRA C Guidelines - Required)

- The essential type model
  - Operands shall not be of an inappropriate essential type

Operator	Operand	Essential type category of arithmetic operand					
		Boolean	character	enum	signed	unsigned	floating
[ ]	integer	×	×				×
+ (unary)		×	×	×			
- (unary)		×	×	×		×	
+ -	either	×		×			
* /	either	×	×	×			
%	either	×	×	×			×
< > <= >=	either	×					
== !=							

Safety Critical Software Systems (MISRA C Guidelines - Required)

- The essential type model ...

Operator	Operand	Essential type category of arithmetic operand					
		Boolean	character	enum	signed	unsigned	floating
! && ||	any		×	×	×	×	×
<< >>	left	×	×	×	×		×
<< >>	right	×	×	×	×		×
~ & | ^	any	×	×	×	×		×
?:	1st		×	×	×	×	×
?:	2nd and 3rd						

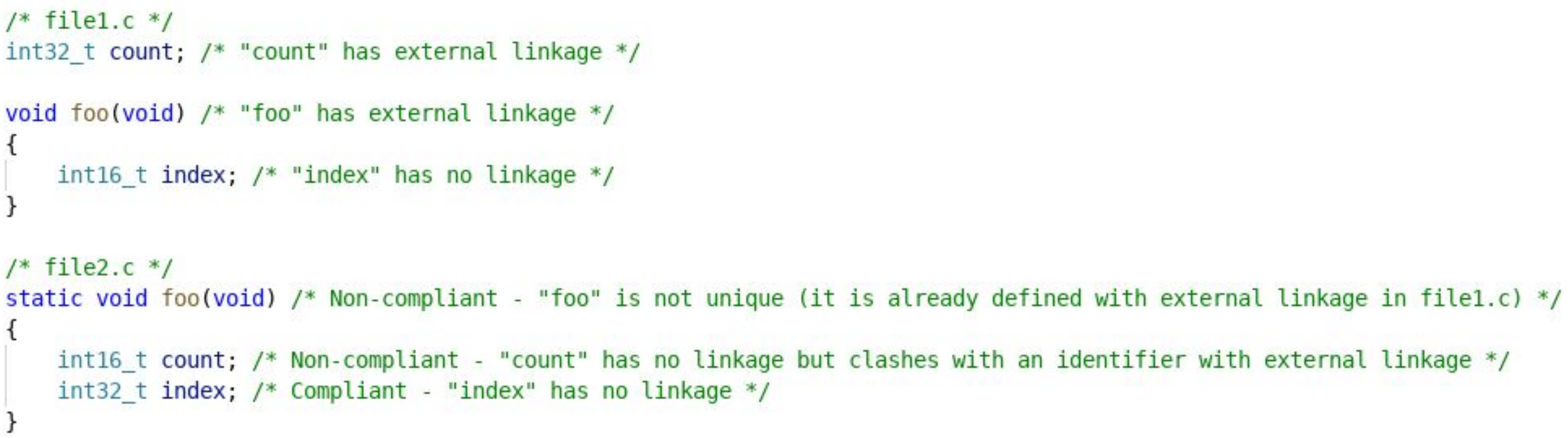
- Expressions of essentially character type shall not be used inappropriately in addition and subtraction
  - Exceptions: convert between digits in the range ‘0’ to ‘9’ and the corresponding ordinal value
  - Convert a character from lowercase to uppercase and vice versa

Safety Critical Software Systems (MISRA C Guidelines - Required)

- If a function returns error information, then that error information shall be tested
- Contents of a header file shall not be included more than once
- The validity of values passed to functions shall be checked
- Dynamic memory allocation shall not be used
- A program shall contain no violations of the standard C
- There shall be no occurrence of undefined or critical unspecified behaviour
- The character sequences /* and // shall not be used within a comment
    - Exception: The sequence // is permitted within a // comment
- Functions and objects shall not be defined with external linkage if they are referenced in only one translation unit (file)

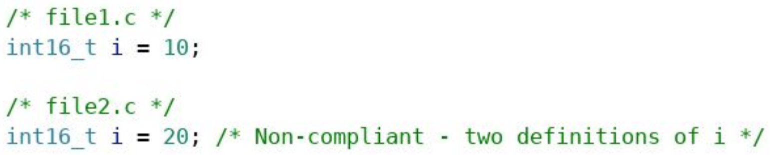
Safety Critical Software Systems (MISRA C Guidelines - Required)

- Identifiers that define objects or functions with external linkage shall be unique

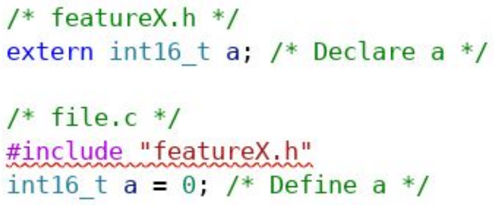


```c
/* file1.c */
int32_t count; /* "count" has external linkage */

void foo(void) /* "foo" has external linkage */
{
    int16_t index; /* "index" has no linkage */
}

/* file2.c */
static void foo(void) /* Non-compliant - "foo" is not unique (it is already defined with external linkage in file1.c) */
{
    int16_t count; /* Non-compliant - "count" has no linkage but clashes with an identifier with external linkage */
    int32_t index; /* Compliant - "index" has no linkage */
}
```

- An external object or function shall be declared once in one and only one file
- An identifier with external linkage shall have exactly one external definition



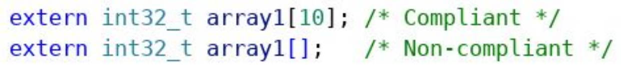
```c
/* file1.c */
int16_t i = 10;

/* file2.c */
int16_t i = 20; /* Non-compliant - two definitions of i */
```



```c
/* featureX.h */
extern int16_t a; /* Declare a */

/* file.c */
#include "featureX.h"
int16_t a = 0; /* Define a */
```

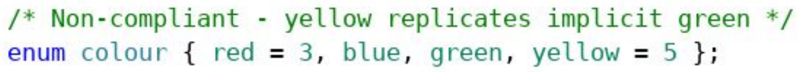
Safety Critical Software Systems (MISRA C Guidelines - Required)

- The **static** storage class specifier shall be used in all declarations of objects and functions that have internal linkage
- An **inline function** shall be declared with the static storage class

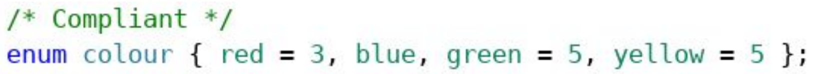


```
extern int32_t array1[10]; /* Compliant */
extern int32_t array1[];   /* Non-compliant */
```

- When an array with external linkage is declared, its size shall be explicitly specified
- Within an enumerator list, the value of an implicitly-specified enumeration constant shall be unique



```
/* Non-compliant - yellow replicates implicit green */
enum colour { red = 3, blue, green, yellow = 5 };
```



```
/* Compliant */
enum colour { red = 3, blue, green = 5, yellow = 5 };
```

- Functions shall not call themselves, either directly or indirectly
- A value returned by a function having non-void return type shall be used
- A macro shall not be defined with the same name as a keyword
- All #else, #elif and #endif preprocessor directives shall reside in the same file as the #if, #ifdef or #ifndef directive to which they are related

Safety Critical Software Systems (MISRA C Guidelines - Required)

- A reserved identifier or macro name shall not be declared
- The standard library memory allocation/deallocation functions of <stdlib.h> shall not be used
- The standard header file <setjmp.h> shall not be used
- Standard header file <signal.h> shall not be used
- The standard library input/output functions shall not be used
- The atof, atoi, atol and atoll functions of the standard library <stdlib.h> shall not be used
- The standard library functions abort, exit, getenv and system of <stdlib.h> shall not be used
- The standard library functions bsearch and qsort of <stdlib.h> shall not be used
- The standard library time and date functions shall not be used
- All resources obtained dynamically by the standard library functions shall be explicitly released
- The same file shall not be open for read and write access at the same time on different streams

Safety Critical Software Systems (MISRA C Guidelines - Advisory)

- The essential type model
  - The value of an expression should not be cast to an inappropriate essential type
  - Casting from **void** to any other type is not permitted as it results in **undefined behaviour**.

Essential type category	**from**					
to	Boolean	character	enum	signed integer	unsigned integer	floating
Boolean		×	×	×	×	×
character	×					×
enum	×	×	×	×	×	×
signed integer	×					
unsigned integer	×					
floating	×	×				

Safety Critical Software Systems (MISRA C Guidelines - Advisory)

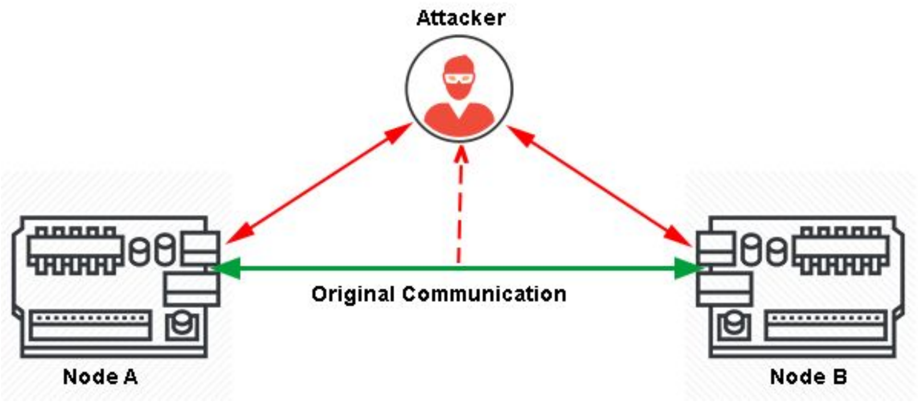
- If a pointer to a structure or union is never dereferenced within a translation unit, then the implementation of the object should be hidden
- Functions which are designed to provide operations on a resource should be called in an appropriate sequence. e.g. opening a file; reading from a file; closing a file;
- A project should not contain unused type declarations
- A project should not contain unused macro declarations
- An object should be defined at block scope if it's identifier only appears in a single function
- A pointer should not be converted into an integer and vice versa.
  - If casting between integers and pointers is used, care should be taken. E.g. addressing memory mapped registers or other hardware specific features

Safety- and security-critical systems

- In an embedded software, both safety and security are important.
  - Safety - Ability of a system to function
    - Without causing harm to objects or people with a sufficient level of reliability
  - Security - Ability of a system to to protect
    - Against external attacks e.g. hackers and leakage of of sensitive data

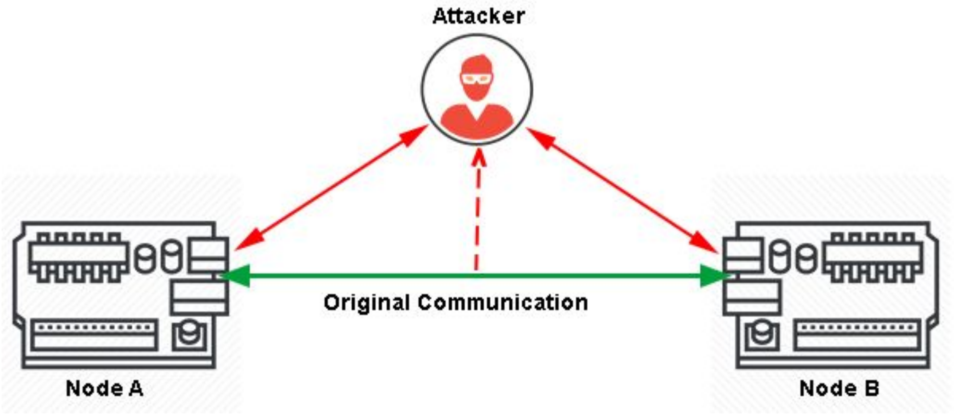
🔗 [Keeping Embedded Secure]()

- Security in Communication: It is all about **C**.**I**.**A**
  - **C**onfidentiality - allows the sender and receiver to share information only in the way they intend
  - **I**ntegrity – allows the receiver to verify that the data has not been altered during transmission
  - **A**uthentication - allows the receiver of the information to be certain where it came from



Security in Communication

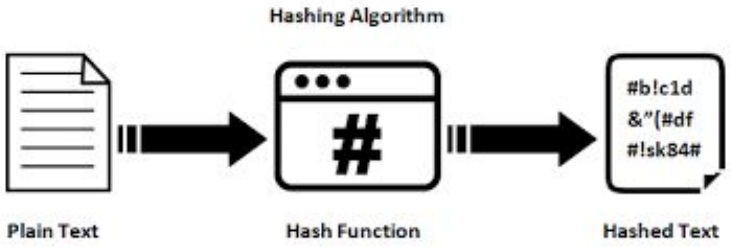
- Note that the attacker can listen to the communication and can send and receive data
- If the communication is not secure
  - Sensitive information is leaked
  - The attacker can act as man in the middle
    - Pretends it is the sender and receiver



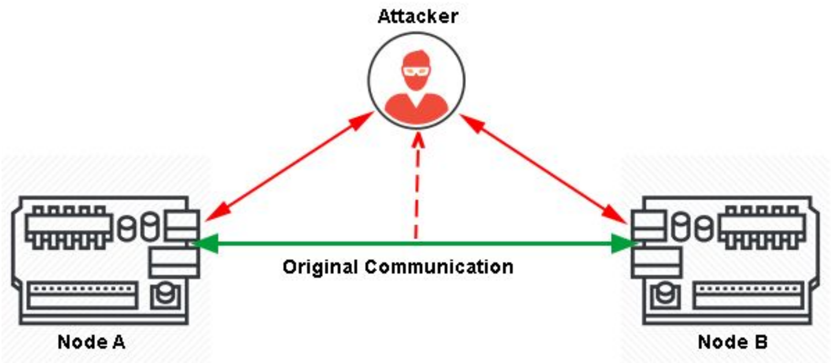
- Data Integrity
  - A checksum be used to check if a message has been altered during transmission or not
  - There are [different algorithms] to checksum a message. E.g. CRC, MD, SHA and etc.
  - [A hash function] is a one-way function which takes a message with any length and generates a unique and fixed size hash value from the message.
    - There is no way to reverse the process and convert the hash value to the original message

Security in Communication

- **Hash** algorithms are used in data integrity and creating digital signatures
- SHA (Secure Hash Algorithm) are cryptographic hash functions used to verify data integrity.
  - There are SHA1, SHA2 and SHA3 types
  - E.g. SHA1 generates a 160-bit hash value(20 bytes)
  - E.g. SHA256 is a SHA2 type which takes a message with any length and generates a 256-bit hash value (32 bytes)

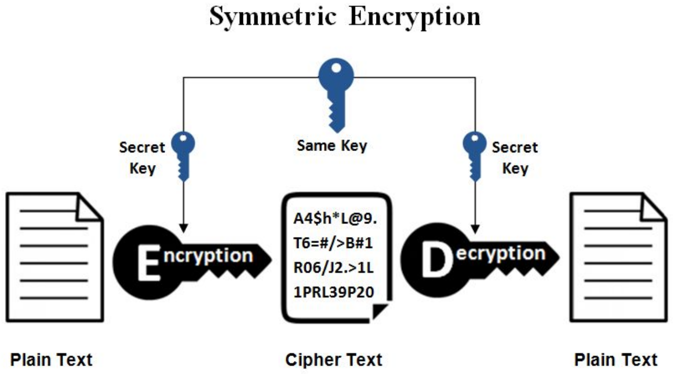


- **Cryptography** is the practice and study of techniques for secure communication in the presence of third parties called adversaries.
  - **Key**: A secret like a password used to encrypt and
  - decrypt information. There are a few different types of keys used in cryptography.



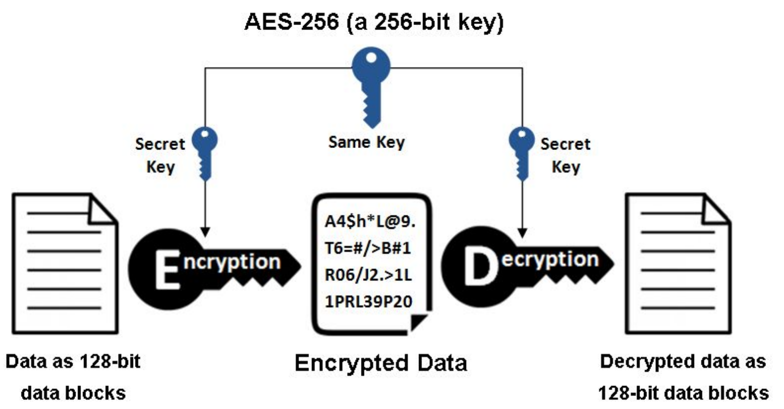
Security in Communication - Cryptography

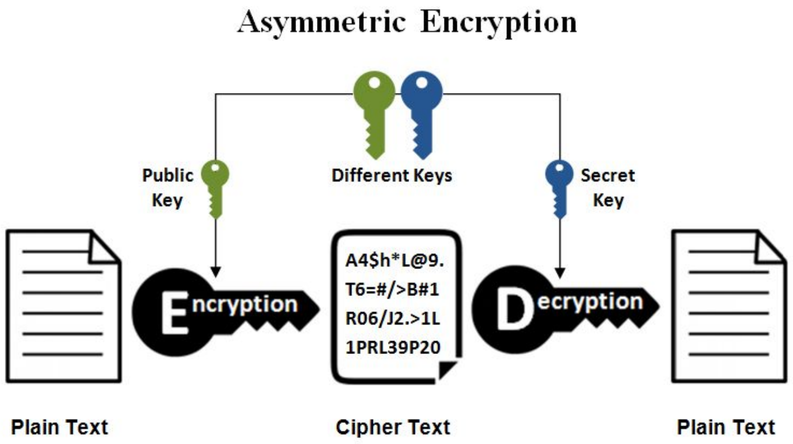
- **Encryption**: It is the process of locking up information using cryptography.
  - Data is turned into ciphers(encrypted data)
  - The encrypted data is not understandable by the third-party.
- **Decryption**: The process of unlocking the encrypted information.
- There are **Symmetric** and **Asymmetric** encryptions
- Symmetric encryption
  - In this method only one shared secret key is used by the parties to encrypt and decrypt information
  - The process is very fast and it only provides confidentiality.
  - Algorithms like DES, **AES**(Advanced Encryption Standard) and etc. are used
    - AES is a block cipher with a data block of 128 bits and three key lengths: 128, 192, or 256 bits.



Security in Communication - Cryptography

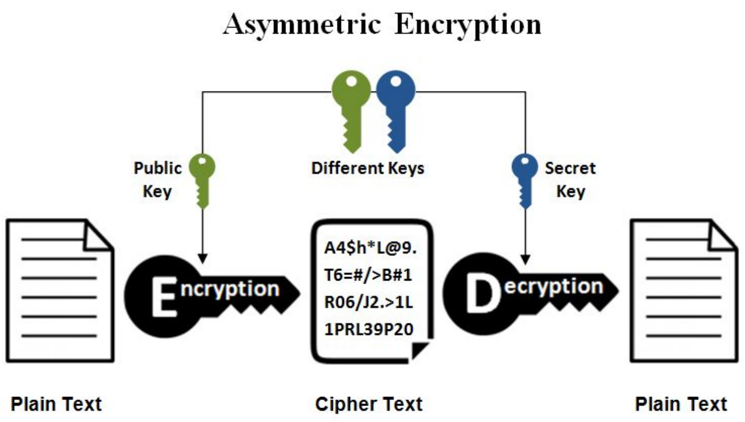
- **AES-256** as the strongest encryption standard uses
  - A 256-bit secret key to encrypt and decrypt blocks of 128-bit data. Means that the data shall be blocks of 128 bits and for each block the encrypted data is 128 bits. Padding is used.
- Asymmetric encryption
  - In this method a pair of keys (public and private) is used by the parties to encrypt and decrypt data
  - It is also called public key encryption. One of the keys is used to encrypt and the other one is used to decrypt information. The public key is shared.





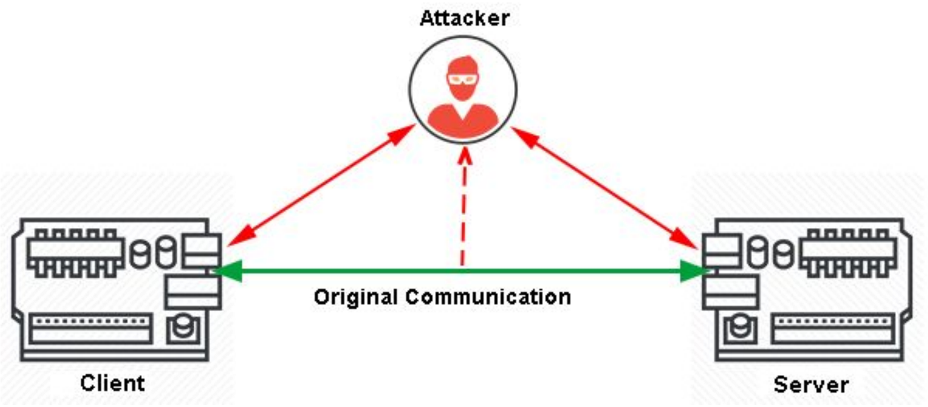
Security in Communication - Cryptography

- Asymmetric encryption …
  - The process is slow. It provides both confidentiality and authenticity. Usually it is used to sign and share secrets like AES keys. The private key is used to sign secrets (encrypt with the private key and decrypt with the public key)
  - Algorithms like ECC, DSA and **RSA** are used.
- **RSA** (Rivest–Shamir–Adleman) algorithm
  - A pair of big keys is used; usually 2048-bit keys
  - The data block size and the cipher size are equal and their sizes are equal to the size of keys
  - Padding is used to pad data block



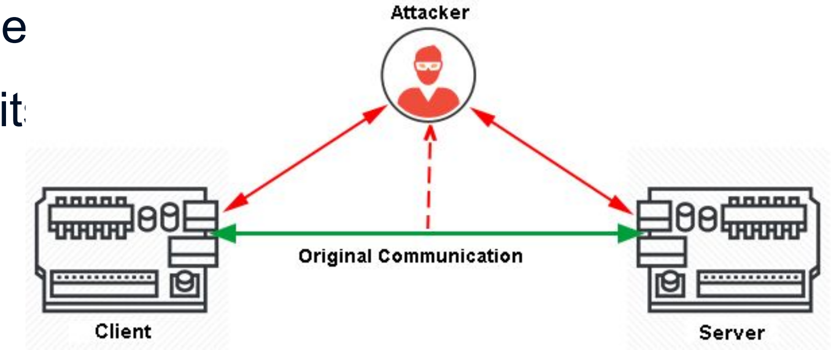
- Asymmetric Encryption - Simply explained
- Symmetric Key and Public Key Encryption
- RSA algorithm
- Symmetric vs. Asymmetric Key Encryption
- Symmetric vs. Asymmetric Encryption
- Public Key Cryptography
- RSA Key Generator

Security in Communication - Example



- Suppose that the server has a service restricted to some clients who have secret IDs shared with the server. Means that the server has a list of clients IDs and it gives the service to a client whose ID has been registered in the server's list. If a client wants to get the service, first it should send its ID in a secure way to the server. Using the SHA, AES and RSA make the communication between them secure.
- To make the communication secure
  - All the transactions shall be hashed using e.g. SHA-256
    - The sender calculate the hash for its message and append it to the message
    - The receiver recalculate the hash for the received message and compare it with the received hash value. If they are equal, it means that the message has not been altered.

Security in Communication - Example

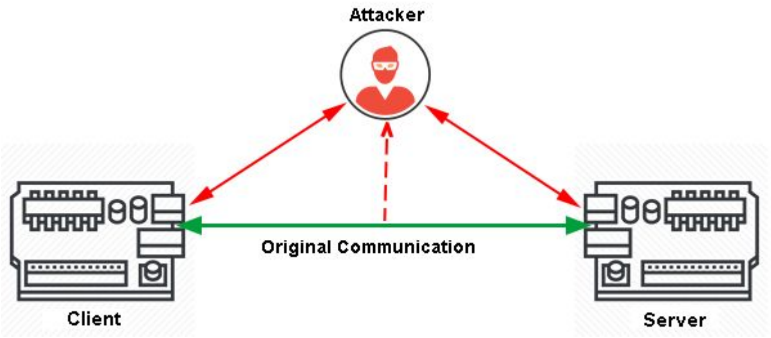


- The server sends its public key to the client. Note that the attacker can pretend it is the server and can also send it public key to the client.
  - In this case the client can not be sure that it has received the public key of the real server. To solve this problem a certificate is used. It means that an authorized third-party approves the identity of the server.
    - E.g. Using a [DigiCert TLS/SSL Certificate]

- The client receives server’s public key and generates a pair of keys using RSA.
- The client encrypts its public key with the public key of the server and sends it to the server.
- The server decrypt the message using its own private key and obtains the public key of the client
- Now the server and the client have exchanged their public keys
- The client signs its secret ID and encrypts it with server’s public key and sends it to the server

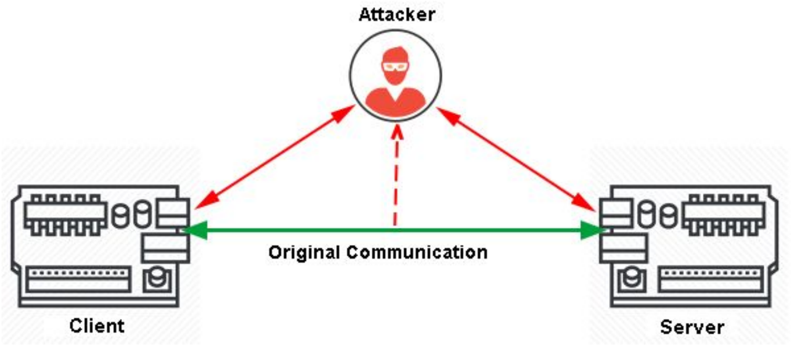
Security in Communication - Example

- The server decrypts the received message using its own private key and the client public key and gets the secret ID of the client.
- The server checks if client’s secret ID is valid or not.
  - If it is not valid e.g an unauthorized error message can be sent back
- The server generates a random AES key, a random unique session ID and a random sequence number.
  - A timestamp also can be used as the sequence number.
  - In any way, the server shall prevent using the same message twice. Note that the attacker can listen to the communication and capture the valid messages and send them to the server. The server can distinguish between the valid messages from the real client and the valid messages from the attacker using such a sequence number. In this way a message is valid only once.
- The server signs the AES key, the session ID and the sequence number in a message and then encrypts it using client’s public key and sends it to the client. The client decrypts the received message using its own private key and server’s public key to get the AES key, the session ID and initialize its own sequence number using the received sequence number from the server.



Security in Communication - Example

- Now there is a **session** between the client and the server and they have the same random AES key, session ID and random sequence number. For the rest of the communication the random AES key is used to encrypt and decrypt messages. The session is encrypted using the AES key and managed using the session ID and the sequence number.
- For each request, the client shall increment its sequence number and include the session ID and the sequence number in its request and encrypt the request using the shared AES and send it to the server
- The server shall decrypt the received message using the shared random AES and check if the received session ID and sequence number of the client match with its own session ID and sequence number or not.
  - If the sequence number does not match, means that the request is not valid
- If the received session ID and sequence number matches with the sequence number of the server, the server decodes the request and encrypt the response with the shared AES key and sends it to the client.
- The server shall increment its own sequence number.



Software Safety

- Some useful links
  - Safety Critical System
  - Safety critical sensors in cars
  - Automotive functional safety
  - Critical systems
  - Safe and Reliable Computer Control Systems Concepts and Methods
  - Functional Safety with ISO 26262 - Principles and Practice
  - GENIVI Alliance
  - Autonomous Vehicle Standards
  - CERT C Programming Language Secure Coding Standard
  - A Quick Guide to ISO 26262
  - Doxygen