

Embedded Software / ESS2

Master CORO / WET

Jean-Luc Béchenne - jean-luc.bechenne@ls2n.fr

1. The C language is not your friend - pitfalls and programming rules
2. Good practices
3. Models and model checking - how to validate a design
4. Application design

The C language is not your friend

Introduction

- C is an old language (developed between 1969 and 1973 by Denis Ritchie at Bell labs)
 - ANSI C - C90 (1990) standard
 - C99 (1999) standard
 - C11 (2011) standard
- It is the most widely used programming language of all times
- Fits well with the architectural features of computers: close to the machine
- Language of choice of embedded systems



Data types in C : integer (1)

- Classic

	8 bits	16 bits	32 bits	64 bits
signed	signed char	short / int	long / int	long long
unsigned	unsigned char	unsigned short / int	unsigned long / int	unsigned long long
Not specified	char	-	-	-

- C99 scheme defined in header file <stdint.h>

	8 bits	16 bits	32 bits	64 bits
signed	int8_t	int16_t	int32_t	int64_t
unsigned	uint8_t	uint16_t	uint32_t	uint64_t

Data types in C : integer (2)

- **uint8_t**. 8 bits, 1 byte, 256 values **from 0 to 255** (0 to 2^8-1)
- **int8_t**. 8 bits, 1 byte, 256 values **from -128 to 127** (-2^7 to 2^7-1)
- **uint16_t**. 16 bits, 2 bytes, 65536 values **from 0 to 65535** (0 to $2^{16}-1$)
- **int16_t**. 16 bits, 2 bytes, 65536 values **from -32768 to 32767** (-2^{15} to $2^{15}-1$)
- **uint32_t**. 32 bits, 4 bytes, 4 294 967 296 values **from 0 to 4 294 967 295** (0 to $2^{32}-1$)
- **int32_t**. 32 bits, 4 bytes, 4 294 967 296 values **from -2 147 483 648 to 2 147 483 647** (-2^{31} to $2^{31}-1$)
- min and max values are defined in header file `<limits.h>`
- size in bytes can be got by using `sizeof()`
- if a constant is wanted instead of a variable, use keyword **const**

const uint8_t myConst = 128;

Data types in C : integer (3)

- A scalar variable declaration has the following syntax (what appear between **[]** is optional)

*type var **[= value]**;*


- Example:

```
uint16_t  b;  
int       a = -1;  
int32_t   c;
```

- A literal integer value can be:
 - in decimal, it begins with a number $\neq 0$ except if 0 alone. Continues with numbers
 - in octal, it begins with 0, continues with numbers $\in [0:7]$
 - in hexadecimal, it begins with 0x, continue with numbers of letters $\in [A:F,a:f]$

Data types in C : integer (4)

- What is happening when **casting** from a type to another type ?



	uint8_t	uint16_t	uint32_t	int8_t	int16_t	int32_t
uint8_t	-	0 extended	0 extended	signed	signed	signed
uint16_t	truncated	-	0 extended	truncated	signed	signed
uint32_t	truncated	truncated	-		truncated	signed
int8_t	unsigned	0 extended	0 extended	-	sign ext.	sign ext.
int16_t	truncated	unsigned	unsigned	truncated	-	sign ext.
int32_t	truncated	truncated	unsigned	truncated	truncated	-

- Explicit casting is done by prefixing an expression by the wanted type in parenthesis : (uint16_t)a means cast 'a' to an uint16_t

Demo

Data types in C : integer (5)

- **Implicit casting**, beware !
 - When doing computation, implicit casting are performed

```
uint16_t a = 20;  
int16_t b = -1;  
  
if (a > b) a = 0;
```

What is the value of a at the end ?

- **Overflow** and **underflow**, beware: nothing is done to signal an overflow/underflow

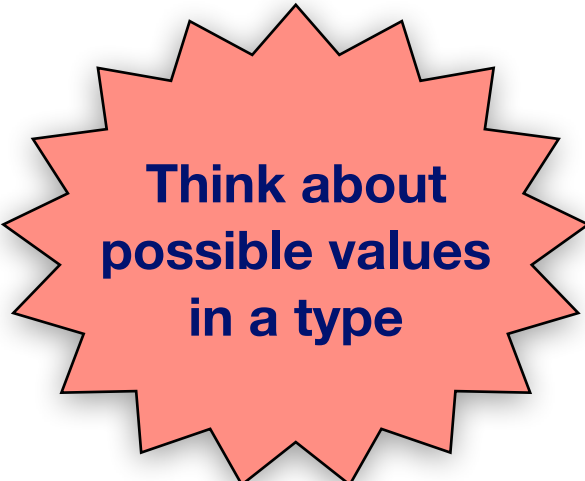
- Classical case

```
uint16_t i = 200;  
  
while (i >= 0) i--;
```

How many times the while loops ?



**Do not
mismatch
signed and
unsigned !**



**Think about
possible values
in a type**

Data types in C : boolean

- There is **no boolean in C**
- The result of a comparison is an integer : 0 (false) or 1 (true)
- But an integer $\neq 0$ is also true and different « true » values cannot be compared

```
int a = 20;
```

```
if (a) { /* same as a != 0 */  
    /* We get here since a is «true» */  
}
```



**Always use
explicit testing !**

```
int a = 20;
```

```
if (a != 0) {  
  
}
```

Data types in C : structures

- Collection of heterogenous data

```
struct my_struct {  
    uint8_t    header;  
    uint16_t   data;  
}; /* declaration of a structure */
```

```
/* a is an instance of  
   struct my_struct */  
struct my_struct a;
```

or

```
typedef struct {  
    uint8_t    header;  
    uint16_t   data;  
} my_type; /* declaration of a new type */
```

```
/* a is an instance of  
   struct my_struct */  
my_type a;
```

- Accessing members of a struct

```
a.header = 3;  
a.data = 4;
```

Data types in C : Pointers

- A pointer is a scalar
- A pointer is variable that stores the address of something:
 - another variable
 - a function
 - a register of a device in memory
- A pointer has a type related to the type of pointed data

```
uint8_t * a;    /* pointer to an uint8_t */  
uint16_t * b;   /* pointer to an uint16_t */  
uint32_t * c;   /* pointer to an uint32_t */
```

Constant pointer and pointer to constant

- The pointer is a variable and the data it points to is also a variable

```
/* pointer to an uint8_t */  
uint8_t * a;
```

- The pointer is a variable and the data it points to is a constant

```
/* pointer to a const uint8_t */  
const uint8_t * a;
```

- The pointer is a constant and the data it points to is a variable

```
/* const pointer to an uint8_t */  
uint8_t * const a;
```

- The pointer is a constant and the data it points to is also a constant

```
/* const pointer to a const uint8_t */  
const uint8_t * const a;
```

Operators related to pointer

- * means value pointed by the pointer or value at address : **dereferencing**
- & means address of a variable

```
uint16_t    a = 3;  
uint16_t * p;    /* pointer to an uint16_t */  
uint16_t    c;
```

```
p = &a; /* p points to a. It contains the address of a */  
c = *p; /* *p gets what is in variable a */
```

Pointers, variables and memory

- Usually, variables are allocated in RAM, constants and code (functions) are allocated in ROM.
- Memory is usually **addressable at byte level**. That is addresses are addresses of bytes and a memory location is a byte
- A variable needs as many memory locations as its size in number of bytes
- **Endianess**. an integer variable can be stored in 2 ways:
 - Big Endian : high order bytes are stored in low addresses
 - Little Endian : high order bytes are stored in high addresses
 - Let's store **0x11223344** (4 bytes integer) at addresses **3000**

Big Endian	3000	11
	3001	22
	3002	33
	3003	44

Little Endian	3000	44
	3001	33
	3002	22
	3003	11

**It depends on
the hardware,
not a choice !**

Data alignment

- Depending on the hardware the data should be **aligned** in memory, or not.
 - Rule of thumb: if the machine is a N bits machine
 1. data of size $\leq N$ can only be located (that is: start) at addresses that can be divided by the size of the data in bytes.
 2. data size $> N$ can only be located at addresses that can be divided by N in bytes
- Example: on a 32 bits computer,
 - a 32 bits data (4 bytes) is aligned to addresses that can be divided by 4
 - a 16 bits data (2 bytes) is aligned to addresses that can be divided by 2
 - a 64 bits data (8 bytes) is aligned to addresses that can be divided by 4
- Accessing a 32 or 16 bits misaligned data may trigger an exception (depends on the machine)

Examples

- Let's consider the following declarations with data stored in little endian scheme on a 32 bits machines

```
uint32_t a = 0x11223344;  
uint16_t b = 0x7788;  
uint8_t c = 2;  
uint32_t * p = a;  
uint8_t * q = c;
```

10002000	44	a
10002001	33	
10002002	22	
10002003	11	
10002004	88	b
10002005	77	
10002006	2	c
10002007	?	
10002008	00	p
10002009	20	
1000200a	00	
1000200b	10	
1000200c	06	q
1000200d	20	
1000200e	00	
1000200f	10	
10002010	?	
10002011	?	
10002012	?	
10002013	?	
10002014	?	

Pointer Arithmetic (1)

- Pointer can be compared when same type

```
uint32_t * p;  
uint32_t * q;  
...  
if (p == q) ...
```

`p == q` : true if pointers **point to the same location**

`p != q` : true if pointers **do not point to the same location**

`p < q` : true if p points to a location which has a **lower address** than the location on which q points to

`p > q` : true if p points to a location which has a **higher address** than the location on which q points to

`>=` and `<=` allowed too

Pointer Arithmetic (2)

- An integer can be added to or subtracted from a pointer

```
uint32_t  a = 0x11223344;  
uint16_t  b = 0x5566;  
uint8_t   c = 0x77;  
uint32_t * p = &a;  
uint16_t * q = &b;  
uint8_t  * r = &c;
```

```
p = p+1; /* p points to an uint32_t that follow a in memory */  
q = q+1; /* q points to an uint16_t that follow b in memory */  
r = r+1; /* r points to an uint8_t that follow c in memory */
```

if the type of a pointer is a `datatype *`, then adding (resp. subtracting) `n` to the pointer adds (resp. subtract) `n * sizeof(datatype)` to the actual address.

Demo

`void *` and `NULL`

- `void *` is a generic pointer type. **Cannot be dereferenced**
- Pointer arithmetic should not be used on a `void *` pointer since the size of the type it points to is not specified. However some compiler allow it (assuming 1 for the size).
For gcc/clang, use `-pedantic-errors` option
- Comparison and assignments are allowed between `void *` and other pointer types
- `NULL` is a convention, a value meaning « this pointer points to nothing »
In fact `NULL` is equal to 0.
Dereferencing a pointer set to `NULL` means accessing memory location of address 0
⇒ No variable is allocated at this address of course

Demo

Pointeurs and structures

- A pointer may point to a structure.

```
typedef struct {  
    uint8_t    header;  
    uint16_t   data;  
} my_type;  
  
/* a is an instance of  
   struct my_struct */  
my_type a;  
my_type * p = &a;
```

- Accessing members through p

```
*p.header = 3;  
*p.data = 4;
```

Or

```
p->header = 3;  
p->data = 4;
```

Accessing anywhere in memory with pointers (1)

- A pointer can be initialized with an integer value: the address
- Useful to access peripheral hardware registers (in fact C was designed for that)
- Example: timer registers of an ATMega 328
 - 5 registers (each a byte)
 - TCCR0A, address 0x24
 - TCCR0B, address 0x25
 - TCNT0, address 0x26
 - OCR0A, address 0x27
 - OCR0B, address 0x28

Accessing anywhere in memory with pointers (2)

```
typedef struct {  
    uint8_t    TCCR0A;  
    uint8_t    TCCR0B;  
    uint8_t    TCNT0;  
    uint8_t    OCR0A;  
    uint8_t    OCR0B;  
} CounterStruct;
```

```
CounterStruct * const TIMER0 = (CounterStruct *)0x24;
```

```
/* get the current value of the timer */  
uint8_t value = TIMER0->TCNT0;
```

Data types in C : arrays (1)

- Declaring an array

```
/* Array of 10 elements */
uint8_t myArray1[10];
/* Initialized array of 3 elements */
uint8_t myArray2[] = { 1, 2, 3 };
/* Oops, may compile */
uint8_t myArray3[];
/* Incompletely initialized array of 10 elements */
uint8_t myArray3[10] = { 1, 2, 3 };
```

- Accessing an array

```
uint8_t a = myArray2[0]; /* value: 1 */
uint8_t b = myArray2[1]; /* value: 2 */
uint8_t c = myArray2[2]; /* value: 3 */
uint8_t d = myArray2[3]; /* Oops, out of bounds */
uint8_t e = myArray2[-1]; /* Oops, out of bounds */
```



**No bounds
checking**

Data types in C : arrays (2)

- Shocking news: arrays do not really exist in C.
 - When accessing an array, the `[]` notation is only syntactic sugar

```
uint8_t myArray2[] = { 1, 2, 3 };
```

- `myArray2` (alone) is a constant pointer to an `uint8_t` (almost) and points to `myArray2[0]` so `myArray2 == &myArray2[0]`
- The compiler translates `myArray2[n]` to `*(myArray2 + n)`
- Proof: `uint8_t myArray2[] = { 1, 2, 3 };`

```
uint8_t a = 0[myArray2]; /* value: 1 */
```

**Of course it's
only to explain,
DO NOT USE
THIS !**

<code>myArray2[n]</code>	\Leftrightarrow
<code>*(myArray2 + n)</code>	\Leftrightarrow
<code>*(n + myArray2)</code>	\Leftrightarrow
<code>n[myArray2]</code>	

**This explains
why no bounds
checking !**

Data types in C : strings (1)

- Strings do not actually exist in C
- Again syntactic sugar to ease the writing of programs
- Declaring a string:

```
char * const myStr = "Hello world !";
```

- The string is an array of char stored in memory and terminated by 0
- `myStr` is a pointer to the first letter, here letter 'H'

3000	'H'	
3001	'e'	
3002	'l'	
3003	'l'	
3004	'o'	
3005	' '	
3006	'w'	
3007	'o'	
3008	'r'	
3009	'l'	
300a	'd'	
300b	' '	
300c	'!'	
300d	0	
300e		
300f		
3010	0	myStr
3011	30	

Demo

Functions (1)

- Syntax

```
<return_type> function_name(<list of arguments>)  
{  
    ...  
}
```

- Examples

```
uint16_t multiply(uint8_t x, uint8_t y)  
{  
    return x * y;  
}
```

```
void Hello()  
{  
    Serial.print("Hello");  
}
```

Functions (2)

- Arguments are passed by value: copied
- No argument passing by reference
 - So there is no way for a callee to modify a variable of the caller
 - If such thing wanted, pass a pointer to the variable
- The content of an array or string cannot be passed

```
void multiply(uint8_t x, uint8_t y, uint8_t * result)
{
    *result = x * y;
}
```

- Instead the pointer is passed

```
uint16_t sum(uint8_t * x, uint16_t size)
{
    uint16_t result = 0;
    for (uint16_t i = 0; i < size; i++) {
        result = result + x[i];
    }
}
```

Functions (3)

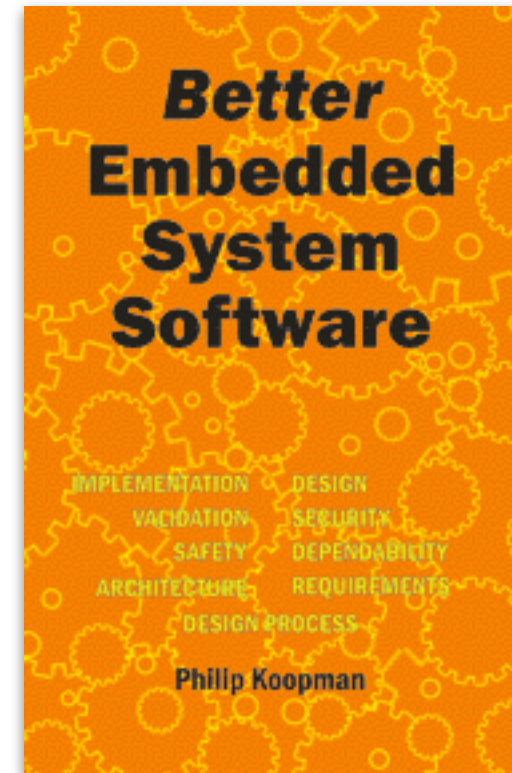
- How to prevent modifying the data of the caller when passing an array or a string ?
 - Use **const**

```
uint16_t sum(const uint8_t * const x, const uint16_t size)
{
    uint16_t result = 0;
    for (uint16_t i = 0; i < size; i++) {
        result = result + x[i];
    }
}
```

Good practices

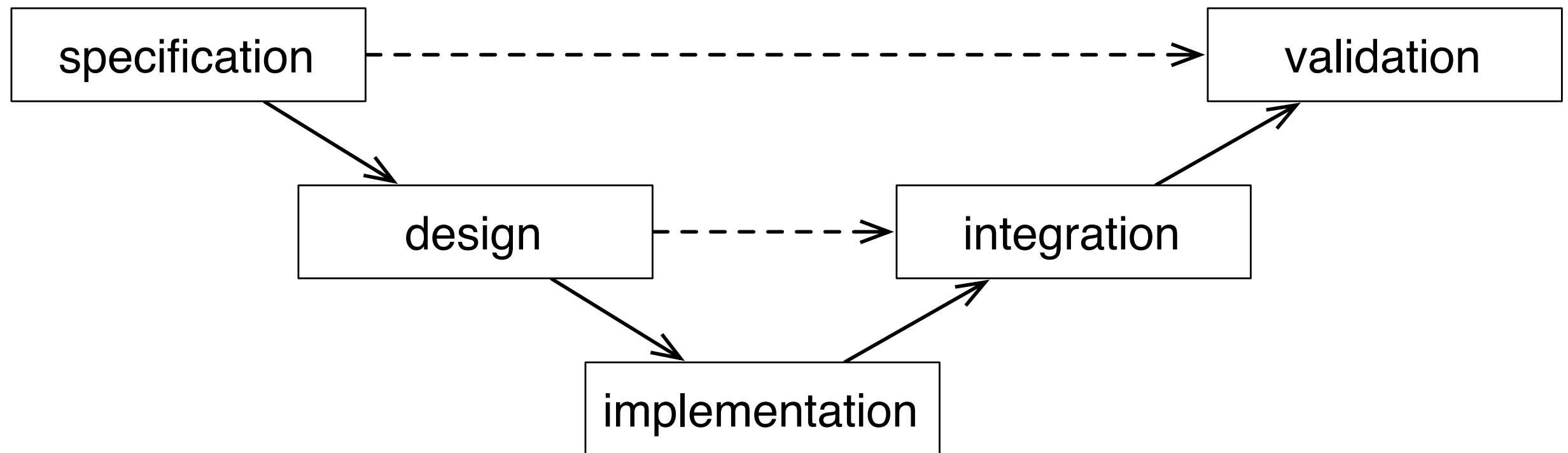
Better Embedded System Software
Philip Koopman, Ph.D.
Carnegie Mellon University

Drumnadrochit Education LLC, 2010
Hardcover, 397 pages, acid free paper
ISBN-13: 978-0-9844490-0-2
ISBN-10: 0-9844490-0-0



Topic Quick Reference	
1. Introduction	1
SOFTWARE DEVELOPMENT PROCESS	
2. Written Development Plan	9
3. How Much Paper Is Enough?	17
4. How Much Paper Is Too Much?	29
REQUIREMENTS & ARCHITECTURE	
5. Written Requirements	37
6. Measurable Requirements	47
7. Tracing Requirements To Testing	57
8. Non-Functional Requirements	65
9. Requirements Churn	77
10. Software Architecture	85
11. Modularity	93
DESIGN	
12. Software Design	107
13. Statecharts and Modes	117
14. Real Time	125
15. User Interface Design	147
IMPLEMENTATION	
16. How Much Assembly Language Is Enough?	157
17. Coding Style	167
18. The Cost of Nearly Full Resources	175
19. Global Variables Are Evil	185
20. Mutexes and Data Access Concurrency	199
VERIFICATION & VALIDATION	
21. Static Checking and Compiler Warnings	219
22. Peer Reviews	227
23. Testing and Test Plans	237
24. Issue Tracking & Analysis	255
25. Run-Time Error Logs	265
CRITICAL SYSTEM PROPERTIES	
26. Dependability	275
27. Security	295
28. Safety	315
29. Watchdog Timers	333
30. System Reset	341

Reminder: system design process (V model)



- Most faults are introduced at design stage
- The later a fault is detected, the higher the cost of the correction
- It is not always possible to “patch” a running system

Use of system/software engineering to maximize early fault detection

Development process: Written development plan (1)

- Tasks that should be performed along the development process
 - requirements
 - acceptance testing
 - test plans
- Inputs and outputs of each each task
 - « Paper » : report, achievement
 - Software or pieces of software
- Risk management
 - identify if things can get wrong
 - list alternative

Development process: Written development plan (2)

- Keep the development process on its path
- Benefits:
 - Resist the management: works better if the development plan has been agreed by the management
 - Easier to face reality: decision about changing or killing the project are easier
 - Keep things consistent across the project.

Development process: Written development plan (3)

- Should address
 - Development approach: scheduling of development steps
 - Requirements: requirements document
 - Architecture: diagrams
 - Testing: test plan
 - Design: flowcharts and statecharts
 - Implementation: commented source code
 - Reviews: test reports, peer reviews
 - Maintenance: bug report and updates

Development process: Paper (1)

Any project that is not appropriately documented should be abandoned when the original developer switches to working on a new project.

- Create enough paper
 - Having zero paper is bad
 - It Cost effort
 - Allow someone else to understand the project
- If not enough ?
 - A developer leaves \Rightarrow project fails
 - Someone has to do reverse engineering to understand what has been programmed
 - You find the product does not fit some requirements

Development process: Paper (2)

- Examples
 - Architecture
 - List of objects, functions, with their interface
 - If there is a network: a message dictionary
 - Design
 - flowcharts, statecharts, pseudo-code
 - Implementation
 - code comments
 - Test
 - list of tests and if passed, failed or not yet run
 - record of peer reviews results

Development process: Paper (3)

- Do not create useless paper
 - If a document is created near the end of the project as « documentation » it is useless: it did not support the development process.
 - A document which is not traced in other documents is useless. Example: a statechart which is not related to the corresponding source code.
 - A document which is not updated during the development process.

We will create the documentation at the end of the project

NO !

Requirements (1)

- Requirements
 - Everything the product must do
 - Everything the product must NOT do
 - Constraints
- Allows to
 - make sure nothing has been forgotten
 - understand the big picture
 - anchor for the design process

Requirements (2)

- Types
 - Functional requirements
 - functions
 - behaviors
 - features
 - Non-functional requirements
 - performance
 - security
 - safety
 - Constraints
 - standards
 - design rules
 - cost
 - hardware

« When the button is pressed the motor should stop »

« Response time to a button press shall be less than 200ms »

« A Cortex-M MCU should be used »

Requirements (3)

~~« When the timer expire, the software shall increment the 16 bits integer variable RollOverCount »~~

« The system shall count how many timer expirations occurred with the ability to tally at least 25000 expirations »

~~« The system shall be fast »~~

« The system shall have a worst case response time of 200ms »

Requirements (4)

Requirement must be easily verified and, if practical to do so, directly measurable

- Tolerance
 - Numerical values should have a tolerance associated with them if single value

~~« When the system is activated, the output signal shall be set high for 500ms and then set low »~~

« When the system is activated, the output signal shall be set high for 500ms **+/- 5ms** and then set low »

Requirements (5)

- Consistency
 - use the same word to name the same thing: we are not doing literature.

« When the button is pressed once, the output signal shall turn on. When the switch is pressed twice, the output signal shall turn off »

« When the button is pressed once, the output signal shall turn on. When the button is pressed a second time, the output signal shall turn off »

Requirements (6)

- use **shall** and **should**
 - **shall**: mandatory behavior
 - **should**: desirable behavior. Might not happen under certain conditions.
- Trackability
 - Each requirement shall have a unique identifier.

Tracing requirements to test

- If there is no test or set of test to cover a requirement then it is not actually tested
- Tracing requirement to test means checking that each requirement is covered by a test and each test corresponds to one or more requirements
- **Traceability matrix**

	R1	R2	R3	R4	R5	R6
T1	x	x				
T2		x				
T3			x		x	
T4					x	x

Non-functional requirements (1)

- Every technical aspect of the system that is required for success should be included in the requirements
- Performances
 - processing deadlines
 - (re)boot speed
 - energy consumption
 - heat dissipation
- Resource usage
 - CPU
 - Memory
 - I/O
 - Network bandwidth

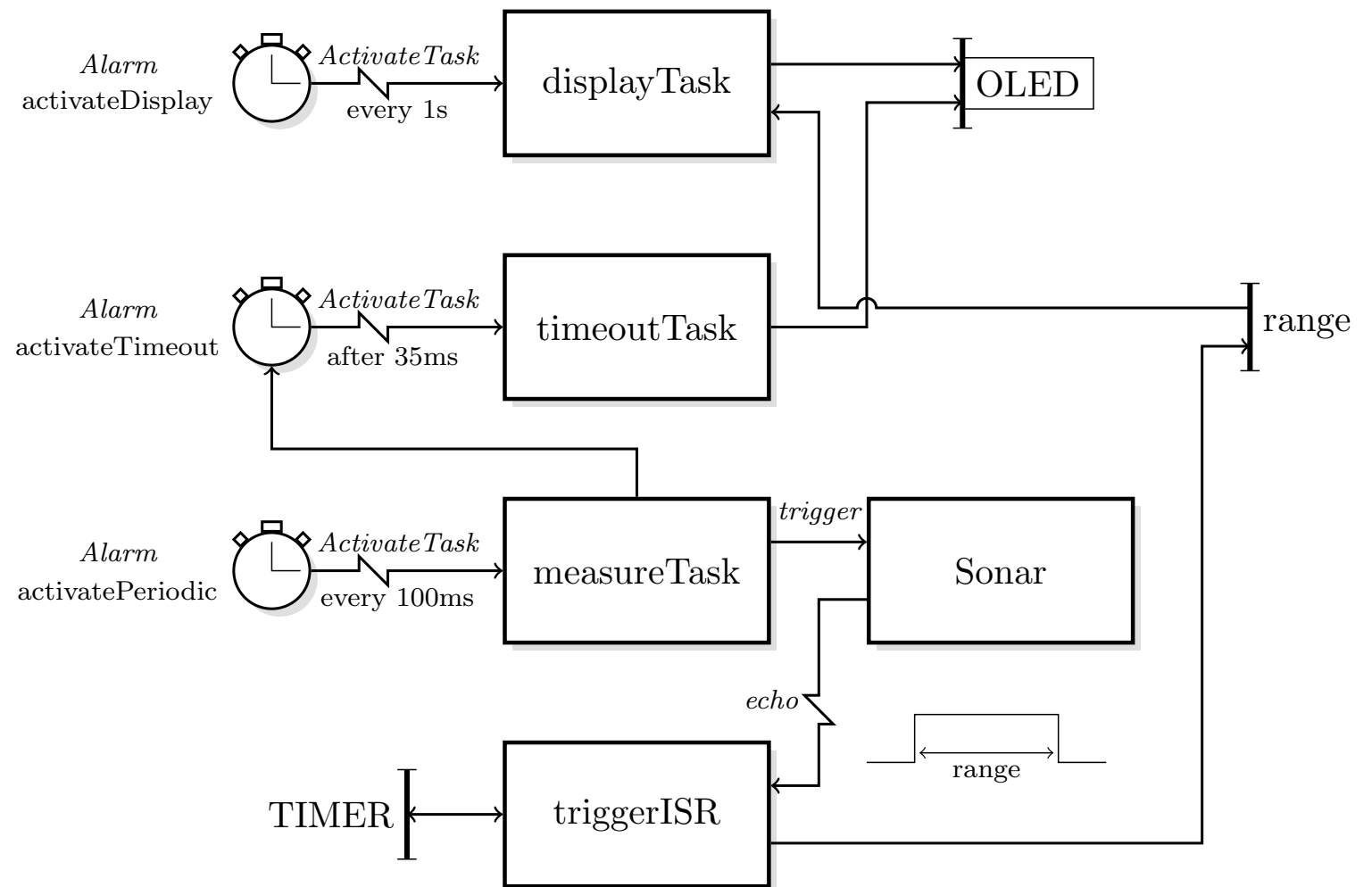
Non-functional requirements (2)

- Dependability
 - Reliability (MTBF)
 - Availability (fraction of « up-time »)
 - Maintainability
- Fault handling
 - Fault detection
 - Fault recovery
 - Fault logging
- Security
 - Authentication
 - Privileges
 - Secrecy
 - Integrity and tamper evidence

Software Architecture

**A messy architecture drawing means
you have a messy architecture**

- Set of components, their behavior and their interconnections within the system.
- At minimum a boxes and arrows diagram describing the overall architecture.
- **Master plan of the overall organization**



Every system should have an architectural diagram which:

- 1. has boxes and arrows with well-defined meaning**
- 2. fits on a single letter-size sheet of paper**
- 3. is homogeneous, clean and relevant**

Common architecture representations

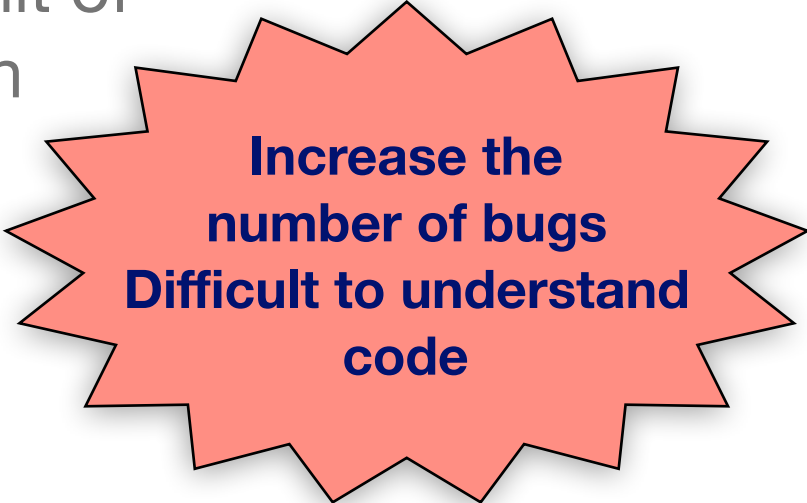
	Boxes	Arrows	Strengths
Call graph	subroutines or objects with functions they provide	subroutines or methods call	show flow of control
Class diagram	Object classes with methods	inheritance	show relationship of classes
Dataflow diagram	Computation or data transformation	streams of data	show data flowing across the system
Hardware allocation diagram	CPUs with software modules	Communication Messages	show allocation and distribution of software
Control hierarchy diagram	Controllers or control loop software	inputs and outputs	show Interactions in hierarchical control

Other architecture representations

- Message dictionary
 - message type
 - payload
 - frequency of transmission
- Real time schedule
 - tasks
 - estimated maximum execution time (WCET)
 - period
 - deadlines
- Memory map
 - program
 - global variables
 - stack(s)
 - heap
 - I/O

Modularity

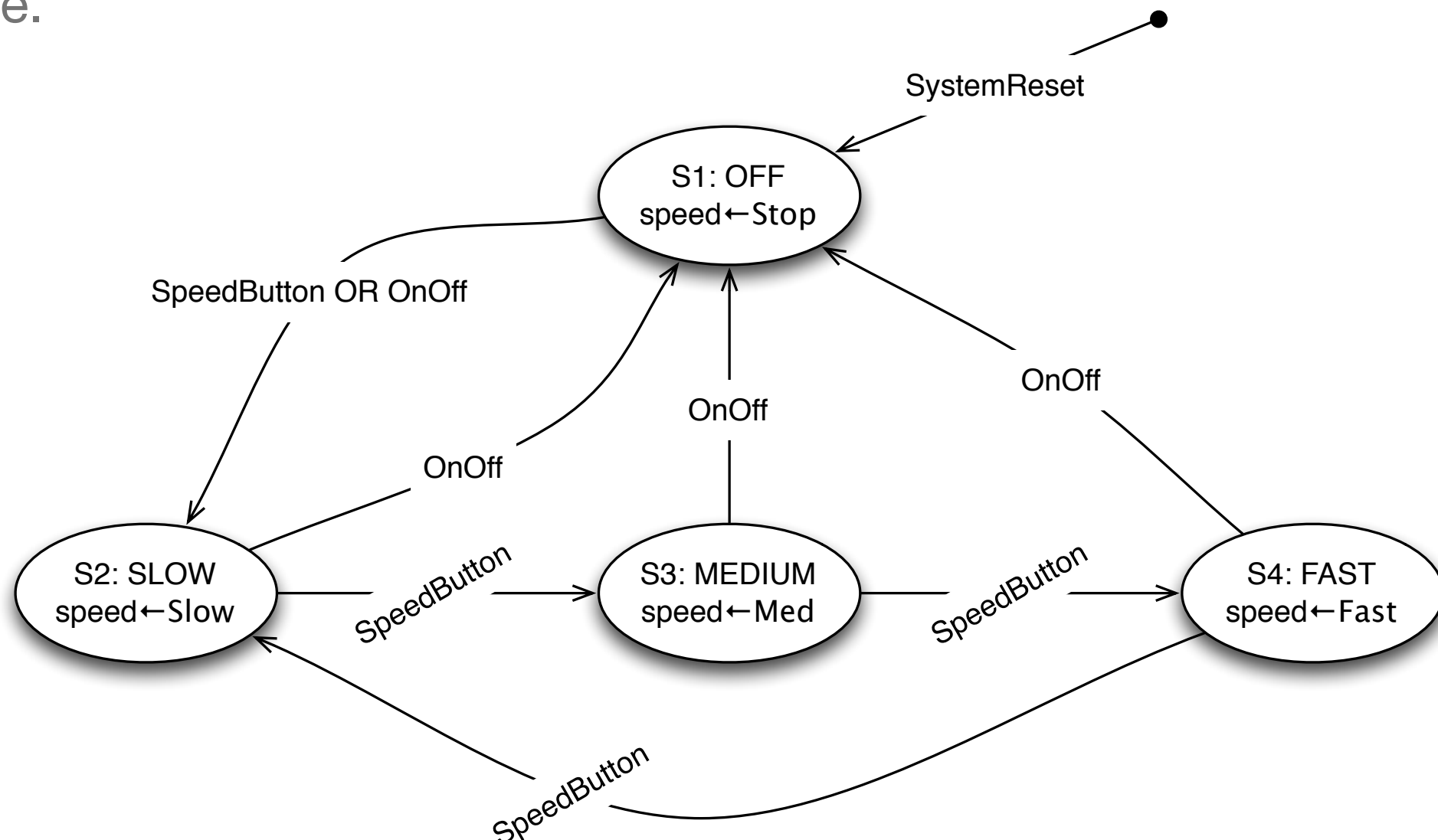
- Break software in modules to manage complexity
 - moderately short modules are better
 - few dependencies with other modules (low coupling)
 - hide implementation
 - well defined interfaces
- Modularity is not good if
 - source code for a single procedure/method/unit of code is > 1-2 pages. Better if it fits on a screen
 - unrelated operations combined in a single module
 - many unrelated parameters passed from one module to another
 - global variables used to pass informations from module to module



**Increase the
number of bugs
Difficult to understand
code**

Statecharts and modes

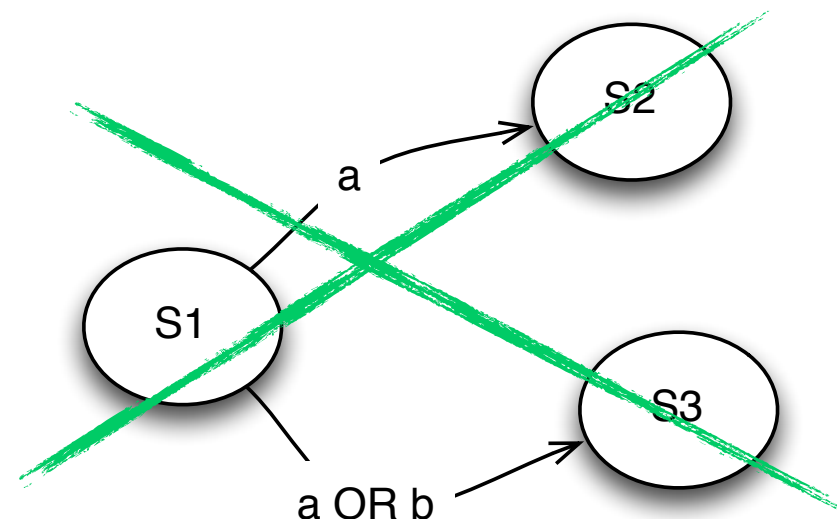
- Embedded software have often a modal or state-based behavior: response of the system depends on an internal finite state machine.
- Statecharts are finite state machine diagrams tailored for representing software.



Statecharts construction

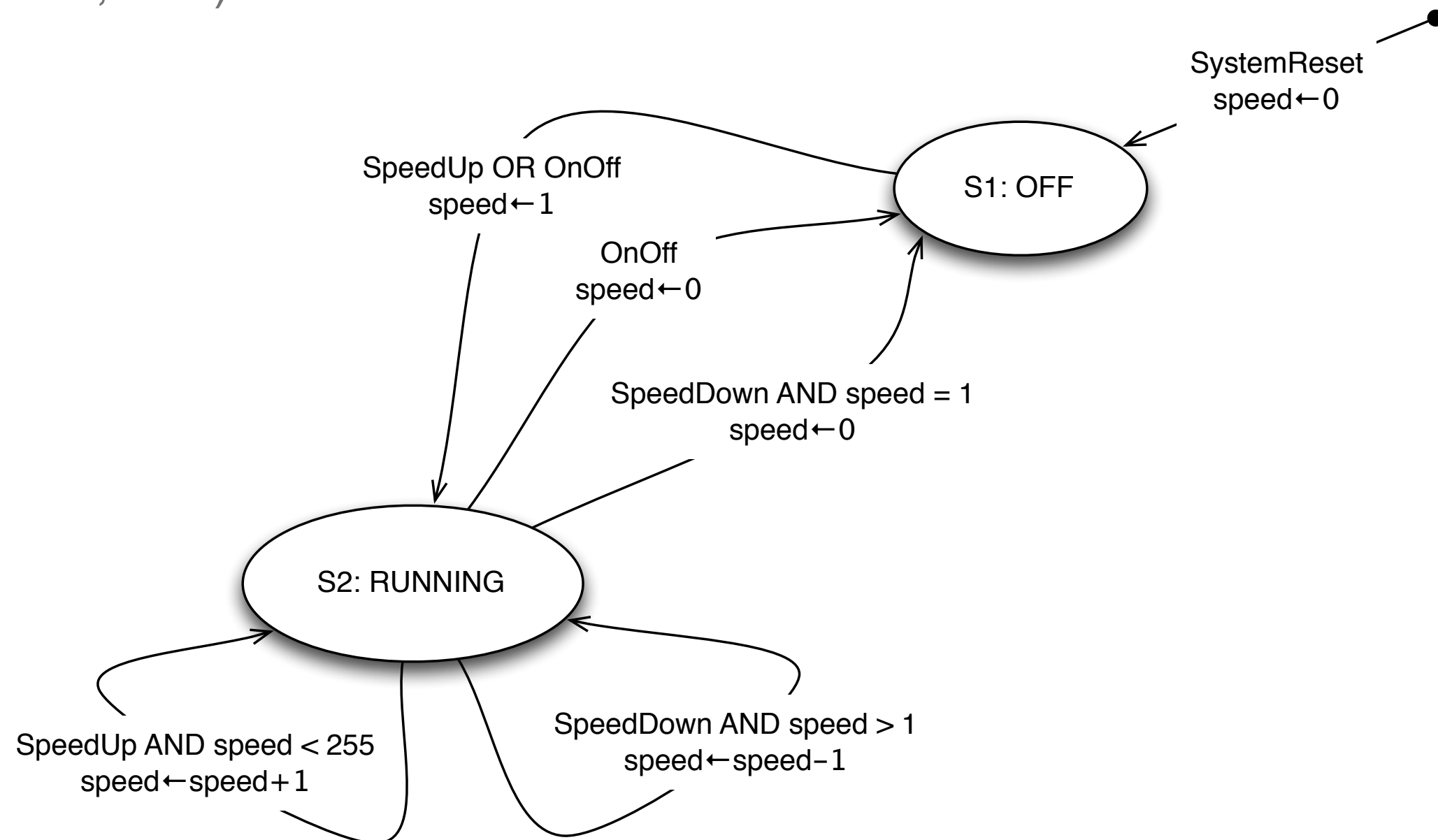
- Initialization arc
 - indicates which state is entered when the system is powered-up/reset
- A set of **states**
 - each state has a name and the actions related when entering in that state
- **Transition** between **states**
 - sport a **guard**: a boolean condition. **If true, the transition is taken**
 - may have actions executed when taking the transition (in this case no action on states).

Guards should be deterministic



Why use actions on transitions

- Not so simple case: speed from 0 to 255 instead of 0 to 3 (Stop, Slow, Medium, Fast) \Rightarrow 256 states !



Implementation of finite state machines

- Use a **switch ... case** statement
- A variable (currentState for instance) stores the current state.
- An enum gives the possible states.
- Straightforward

Coding style (1)

- Software is much easier to read and maintain if a consistent style is used throughout all the code in the project
- Consistent style
- Written style guidelines
 - indentation
 - variable naming
 - comments
 - language usage rules

Coding style (2)

- Every project should have a brief coding style guide
- Source file template : title, header, external interface, data type declarations, global variable if any, procedures, methods and other code definitions
- Comments :
 - interface and semantics of procedures/functions/methods
 - for traceability
 - explain only what is not obvious from looking at the code

```
static States currentState = OFF; /* set currentState to OFF */
```

```
static States currentState = OFF;    /* set speed state machine to  
                                     initial state. REQ 57 */
```


Coding style (3)

- Language usage rules. Examples:
 - All switch statement should have a default case with error handling if not used
 - An assignment shall not be used in conditional evaluation

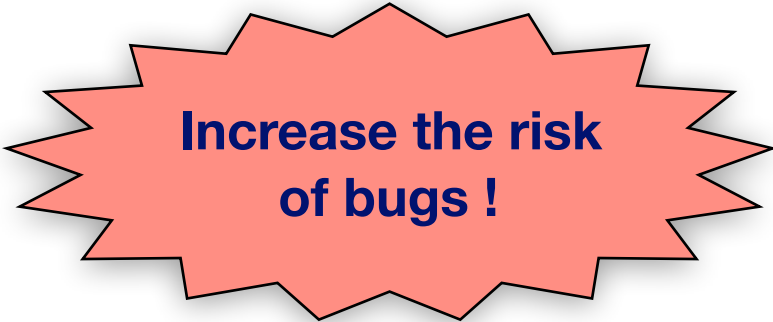
~~if (x = y) { ... }~~

if (x == y) { ... }

- All pointers passed to a subroutine shall be checked for NULL before first use
- Define constant instead of using literal numeric constants
- Non constant pointer to function shall not be used
- Test of a value against 0 shall be made explicit
- ...

Global variables are evil

- Memory allocations that are directly visible from the entire software
- Implicit coupling of all the software
- need mutex in concurrent software (Interrupts or operating system)
- Avoid globals whenever possible
- In C everything that is declared outside a function is potentially global
- Pointers to globals are even more evil



**Increase the risk
of bugs !**

If you still have to use a global, spend a lot of effort making it as clean and well documented as possible

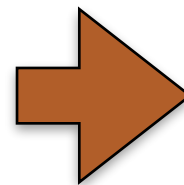
How to avoid or reduce the risk of globals

- Moves globals inside the function/procedure that needs it. Some globals are created to retain their value across multiple invocations of a procedure

static States currentState = 0FF;

```
int errorCount = 0;

int incrementErrorCount (void)
{
    errorCount++;
    return errorCount;
}
```



```
int incrementErrorCount (void)
{
    static int errorCount = 0;
    errorCount++;
    return errorCount;
}
```

How to avoid or reduce the risk of globals

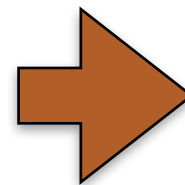
- If you must use a global variable shared by several subroutines, put these subroutines into a single source file.

```
int errorCount = 0;

int incrementErrorCount (void)
{
    errorCount++;
    return errorCount;
}

int errorCount (void)
{
    return errorCount;
}

void clearErrorCount(void)
{
    errorCount = 0;
}
```



```
static int errorCount = 0;

int incrementErrorCount (void)
{
    errorCount++;
    return errorCount;
}

int errorCount (void)
{
    return errorCount;
}

void clearErrorCount(void)
{
    errorCount = 0;
}
```

Static checking and compiler warning

- A static checker produces a warning when it sees something suspicious in source code. Example: splint
- Compiler warning emitted by gcc has greatly improve in past years.
- However useful warnings are turned off by default
 - -Wall : turn on warnings about constructions that some users consider questionable
 - -Werror : makes all warnings errors
 - -pedantic : checks conformity with ISO C/C++ standard
 - -Wextra : additional warnings
 - -Wimplicit-fallthrough : warn about possibly missing break in case
 - -Wmisleading-indentation : warn about wrong indentation
 - -Wshadow : warn if a local declaration shadows a global one

Compiler warnings cont.

- -Wcast-align : Warn whenever a pointer is cast such that the required alignment of the target is increased
- -Wswitch-default : Warn whenever a switch statement does not have a default case.
- -Wswitch-enum : Warn about a missing enumerated type case label or an outside range label.
- -Wconversion : Warn about implicit conversion that may alter a value.

Turn on as much checking options as possible !

Model-checking

What is model-checking ?

- We already saw state-charts
 - useful to represent software of state-based behavior
 - allows to specify formally a behavior (provided the model has an unambiguous semantics)
 - A state chart is a kind of model (but semantics may vary)
- With a clear semantics it is possible to automatically check property on a model.

Modeling

- Model of functional requirements
- Model of non functional requirements
 - energy, memory, ...
 - time
- Requirements have to be verified
- A model is an abstraction

Model-checking

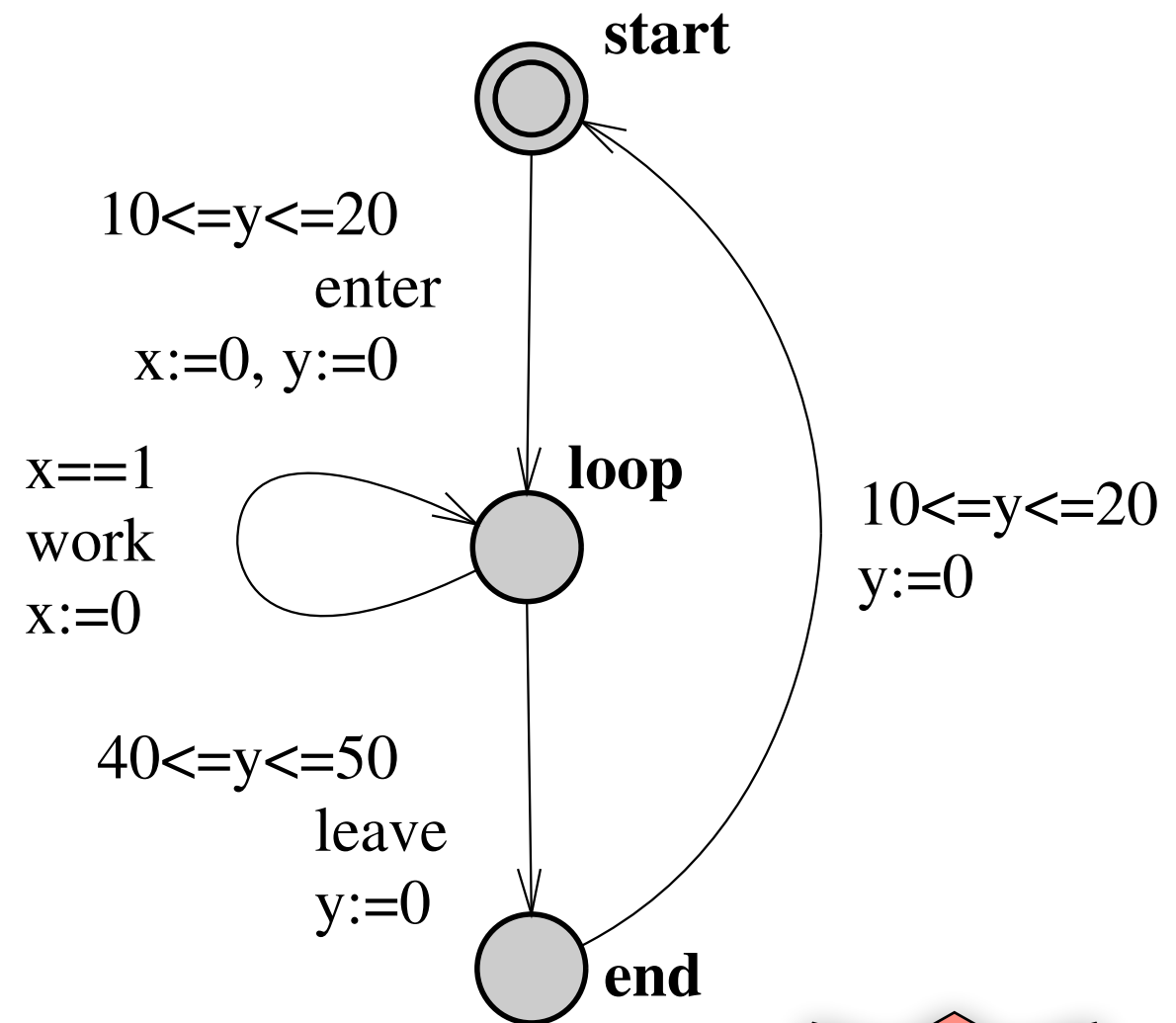
- Modeling of the system S
- formalization of the requirements φ
- Model-checking:
 - Modeling of the controlled system (S + environment): $S \parallel C$
 - Does $S \parallel C \models \varphi$?

Timed automata

- Alur and Dill (90-94)
- Finite Automata extended with clock having a real value
- Clock constraints, that is Guards on transitions are used to restrict the behavior
- Clock may be reset to 0 when a transition is taken

Semantics (1)

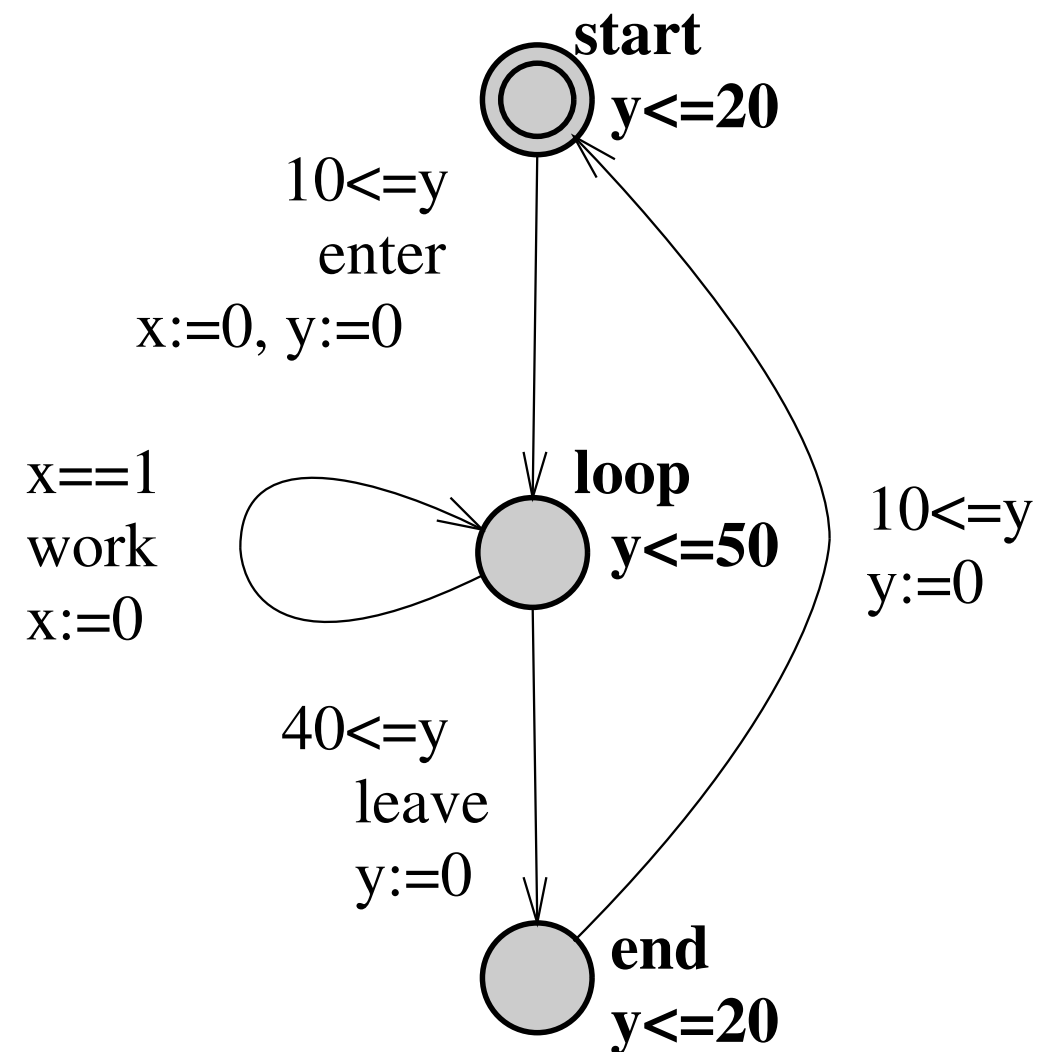
- x and y are clocks (initialized to 0)
- transition *enter* may be taken when y is within $[10, 20]$
- transition *work* may be taken when x is equal to 1
- transition *leave* may be taken when y is within $[40, 50]$
- At least we may go back to start when y is within $[10, 20]$
- If a temporal guard is true, it is not mandatory to take the transition.



**The system
may remain in any
location forever**

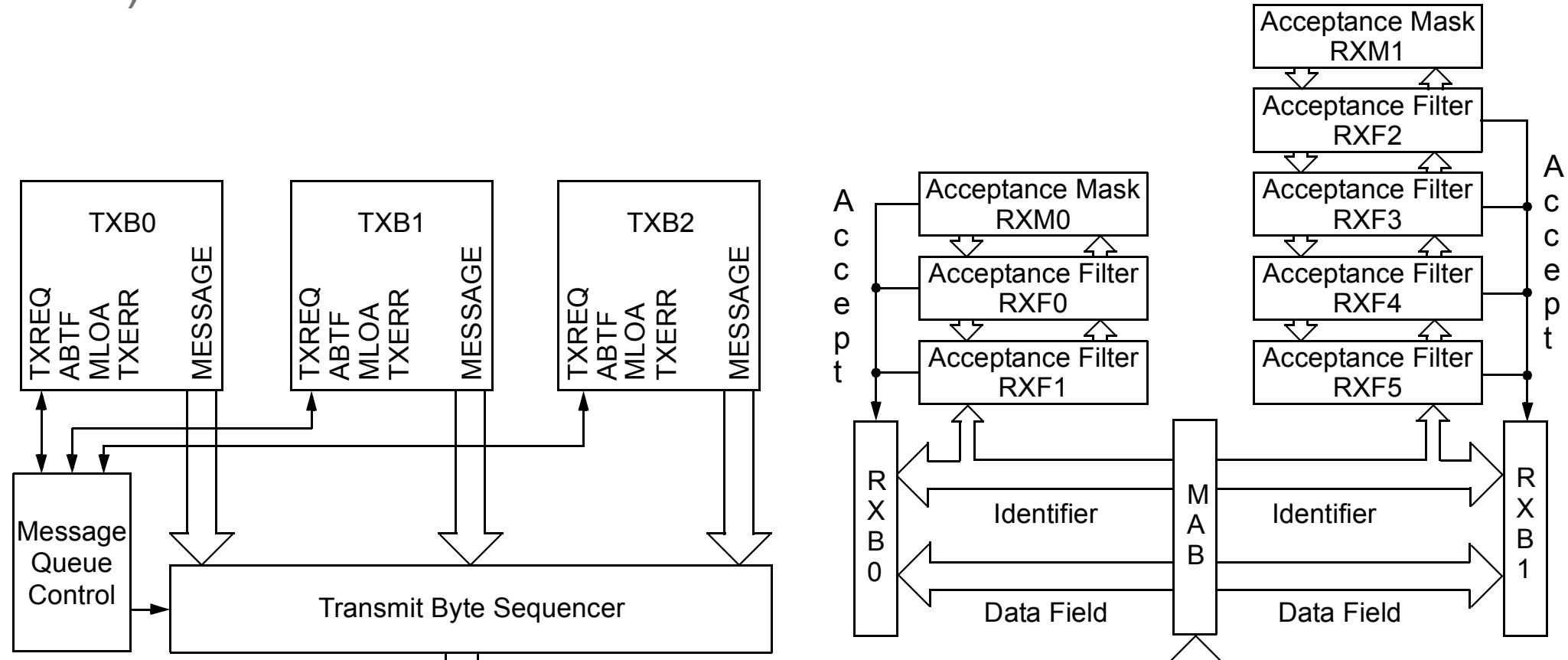
Semantics (2)

- Location invariant: a location must be leaved when the invariant is not satisfied
- *enter* must be taken when y is within $[10, 20]$
- *leave* must be taken when y is within $[40, 50]$
- We go back to start when y is within $[10, 20]$



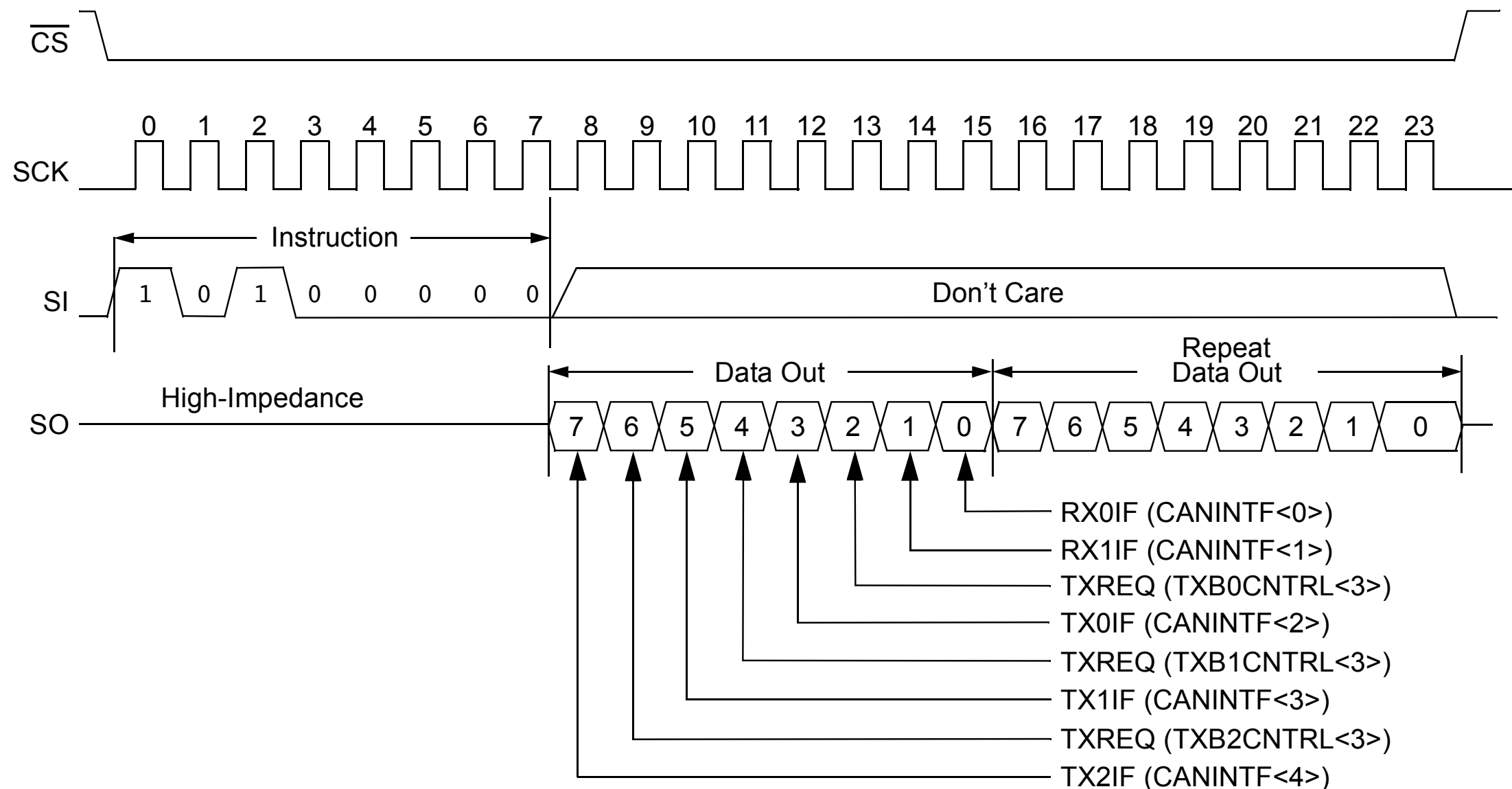
Example (1)

- We want to write a device driver for a MCP2515 CAN controller
 - CAN is a real-time field bus, maximum speed 1Mb/s
 - At 1Mb/s, a frame may take from 47 to 160 μ s to be transmitted
 - The MCP2515 has 2 receive buffers that may be configured in FIFO (almost)



Example (2)

- The MCP2515 is connected using the SPI bus. Let's say with a 8MHz clock
- Reading the status:



Example (3)

- Reading a receive buffer
- A full length buffer is 13 bytes long

