CLEARSY
Safety Solutions Designer

TUTORIAL

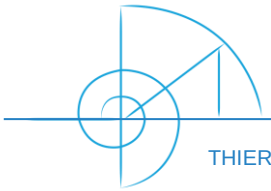
ABZ 2021

JUNE 2021

AIX
LYON
PARIS
STRASBOURG

WWW.CLEARSY.COM

Using B to program the CLEARSY Safety Platform



THIERRY.LECOMTE@CLEARSY.COM

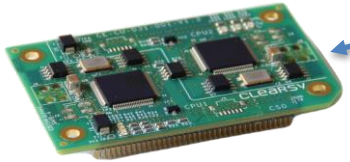
PART III

THE CLEARSY SAFETY PLAFORM FOR INDUSTRY

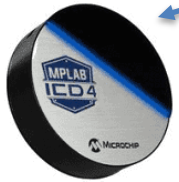
Table of content

- Industrial platform presentation
  - Required components
  - Typical development cycle
- Scope of the conference example
  - Definition of vital and non-vital interface
  - Interfaces and scope limitation
- Implementation
  - Hardware
  - Software
- Real implementation

Required components



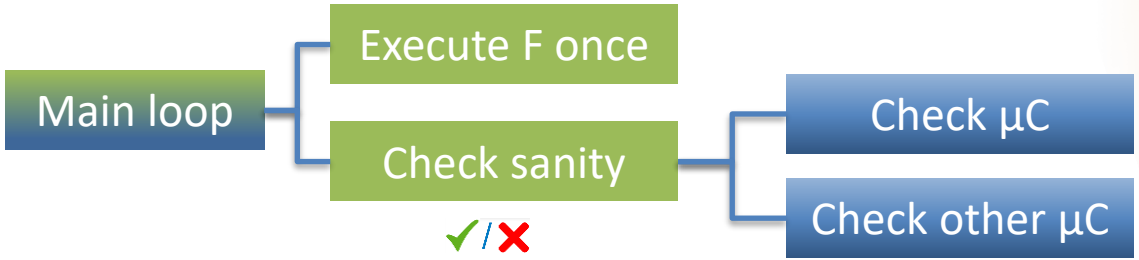
- 1x CS0
- 1x Starter kit
- 1x ICD3 or ICD4 kit
- 1x programmation cable
- 1x USB cable
- 1x laptop
- A few dupont wires



Main Principles: software

Legend

- to develop
- to complement
- developped or generated



Processed by interrupts

- Data acquisition
- Outputs control

CLEARSY Safety Platform for Industry

- B and C used for sequential and interrupted code
- More adaptable to specific needs
- All required verifications implemented
- I/O hosted on a motherboard to develop

compilation toolchain + core computer + SK motherboard
32 inputs, 32 outputs

Development cycle

- Design of safety critical system are made with strict design cycle to ensure a high level of quality and reduce the risk of systematic failure
  - First split features between vital and non-vital features
  - Follow standard prescriptions for design of vital feature
  - Meet the Safety Related conditions of the CLEARSY Safety Platform

    Allows to rely on the built-in mechanism of the CLEARSY Safety Platform → Execution is correct (granted with a SIL4 level).
- Design steps of a typical project with the CLEARSY Safety Platform
  - Design of custom hardware interface (vital or non-vital)
  - Implementation of business logic software
  - Safety demonstration and assessment of the designed product

Safety Related Application Conditions

- A list of 24 exported constraints that the final design shall meet to claim that the execution is ensured to be correct by the platform
  - 5x related to the system design
  - 11x related to the software
  - 3x related to the hardware
  - 5x related to exploitation of the designed product
- During this talk we will focus only on a few of them
  - SRAC-1 Restrictive value
  - SRAC-3 B0 proof
  - SRAC-4 Composite safety
- Exhaustive list is available in the user manual (C_D720 – extract available on github)

Safety Related Application Conditions

SRAC ID	Text	Implication
01	Given a product using the CSP for one or several features, it is assumed that the designer and the safety team define a property P that sums up all the product's safety (P is usually a conjunction of conditions). They shall ensure that P holds when all outputs (hardware, messages, …) are restrictive. This implies to define a restrictive state for each output.	We define the restrictive state by the absence of energy provided by the vital computer board. In this case the system shall be safe so the signal shall be red and the barrier down.
03	The CLEARSY Safety Platform does not provide any mechanism against the systematic design error of the vital duplicated software (the one written in B0). The end user is sole responsible for managing any bug inside the B0 written software. The end user shall demonstrate that the vital software written in B0 is free of any systematic error that may lead to a hazard. In addition, the end user shall validate the units and the value of the associated constants (i.e. data validation is in the user's scope).	The vital software will be formally proven to ensure that it is bug free.

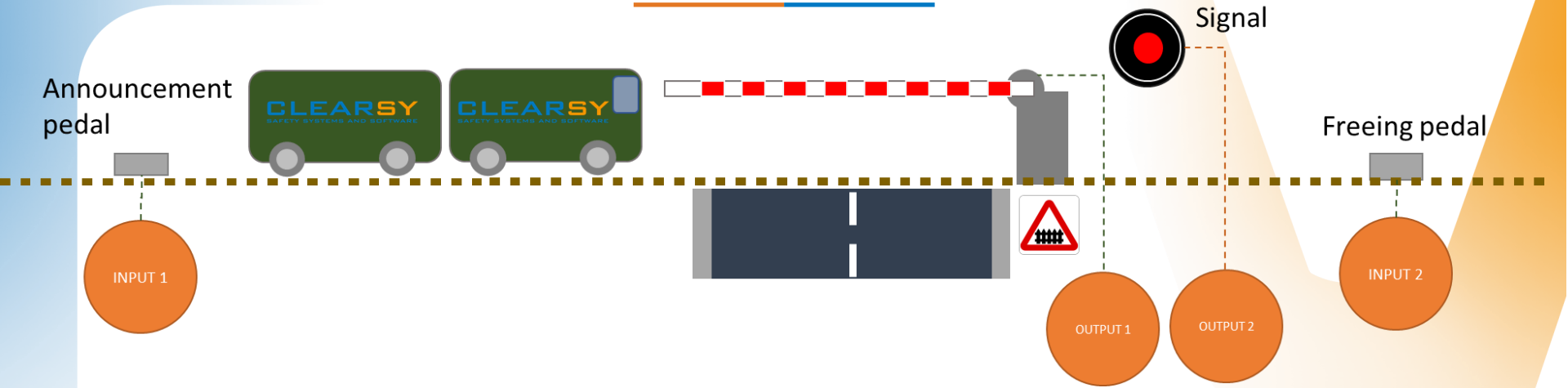
Safety Related Application Conditions (continued)

SRAC ID	Text	Implication
04	The end user shall consider that one of the 2 microcontrollers in any CS0 board may encounter a dangerous random failure […]	We should find a design in which the output cannot be controlled by a single microcontroller (2 out of 2 architecture)

Safety properties offered by the CLEARSY Safety Platform

- If all the Safety Related Application conditions are met then the user is protected against
  - Clock drift (i.e. time counter can be trusted with 105ppm accuracy)
  - Software corruption (i.e. firmware is identical as the one loaded)
  - Memory corruption (i.e. runtime variable are not corrupted)
  - RAM corruption Damage on the ALU (i.e. error on arithmetic operation)
  - Error on configuration of the MPU internal registers
- Without the CLEARSY Safety Platform you will need to prove that your design is robust against these hazards

Yet another level crossing – industrial version (partial)



- Interfaces of the system
  - Announcement (input 1)
  - Freeing pedal (input 2)
  - Red signal output (output 2)
  - Barrier output (output 1)
- Associated requirements
  - This input shall not miss rising edge but may see more rising edges than in reality
  - Ensure that reaction time is less than constant (to close barrier before train arrival)
  - This input shall not wrongly acquire rising edge but may miss some.
  - This output shall not be wrongly energized (i.e. light is dark due to positive logic)
  - This output shall not be wrongly energized (i.e. unwanted barrier up)

Scope limitation

- All the interfaces of the system have at least one safety critical component
- To keep this presentation accessible we limit the scope

Interface	Safety level required	Safety level of SW within the example	Safety level of HW within the example
Announcement pedal	Safety critical	Only maximum response time performed in safety	Non-vital implementation
Freeing pedal	Safety critical	Only maximum response time performed in safety	Non-vital implementation
Red signal	Safety critical	Non vital implementation	Non-vital implementation
Barrier	Safety critical	Vital implementation	Vital implementation

Safety property guaranteed by the example

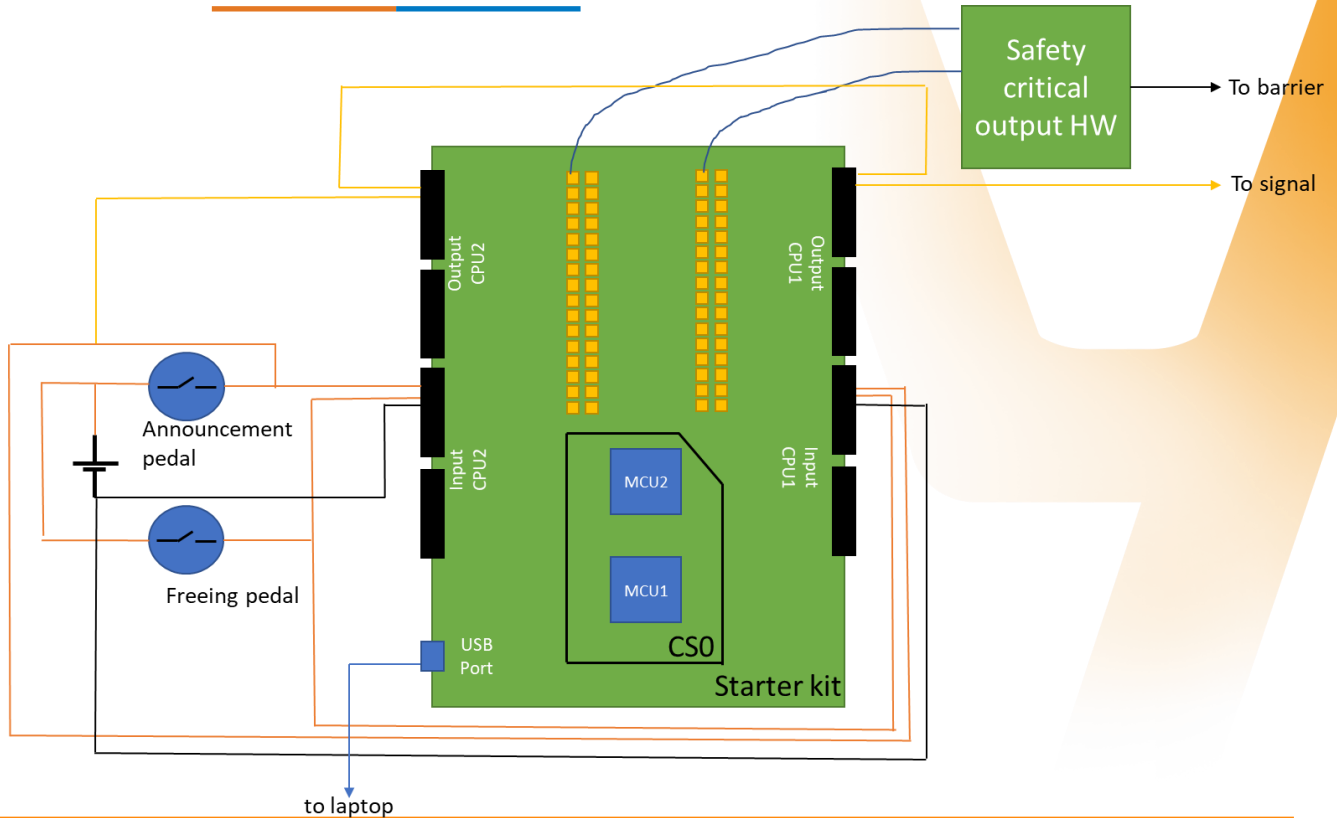
- If the barrier is open (permissive state) then the system has determined that the area was free from train within the last 1 second (duration for the fastest train to travel between announcement pedal and the level crossing)

- If there are more than 8 axles entered in the area then the system shall shutdown and barrier is closed. In this case only a power recycle can restore the system.

Industrial implementation

Hardware setup

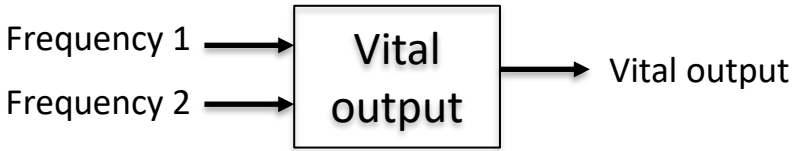
- Composite safety
  - Input are read two time
  - Vital output are controlled by two switches in serial
- Non-vital input
  - Acquired by the starter kit
- Signal output
  - Performed by non vital output
- Barrier output
  - Performed by vital output dedicated interface



Safety design constraints related to the safe output (barrier)

- In our example the barrier is safely controlled through a dedicated hardware board.
  - This board ensures that one MCU cannot bypass the command from the other MCU (double cut output)

  - This board can be seen as a regular output which is controlled by two frequencies. Output is set only if the two frequencies are simultaneously applied



Overview

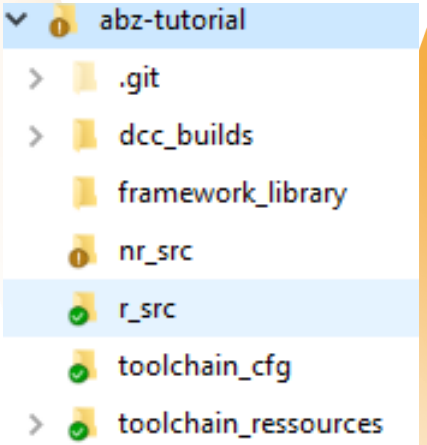
Video « Overview»

Software

- The software of the CLEARSY Safety Platform is bundled in a single tar file
  - Contains documentation
  - Ready to compile examples
  - Official CLEARSY Safety Platform software library (SIL4)
- The user only needs to append its business software. It consists of
  - Appending non-vital source code C within the nr_src folder
    - Driver interface for the starter kit and potential custom interface
    - Scheduling of the application
  - Appending vital software written in B formal method within the r_src folder
    - Business logic (in our case the level crossing behavior)
      - Model for counting number of wheel in the section
      - Generation of the output frequency
      - Deadline for ensuring the pace of the input polling
  - Running a single command for the generation

Structure of the CLEARSY Safety Platform project

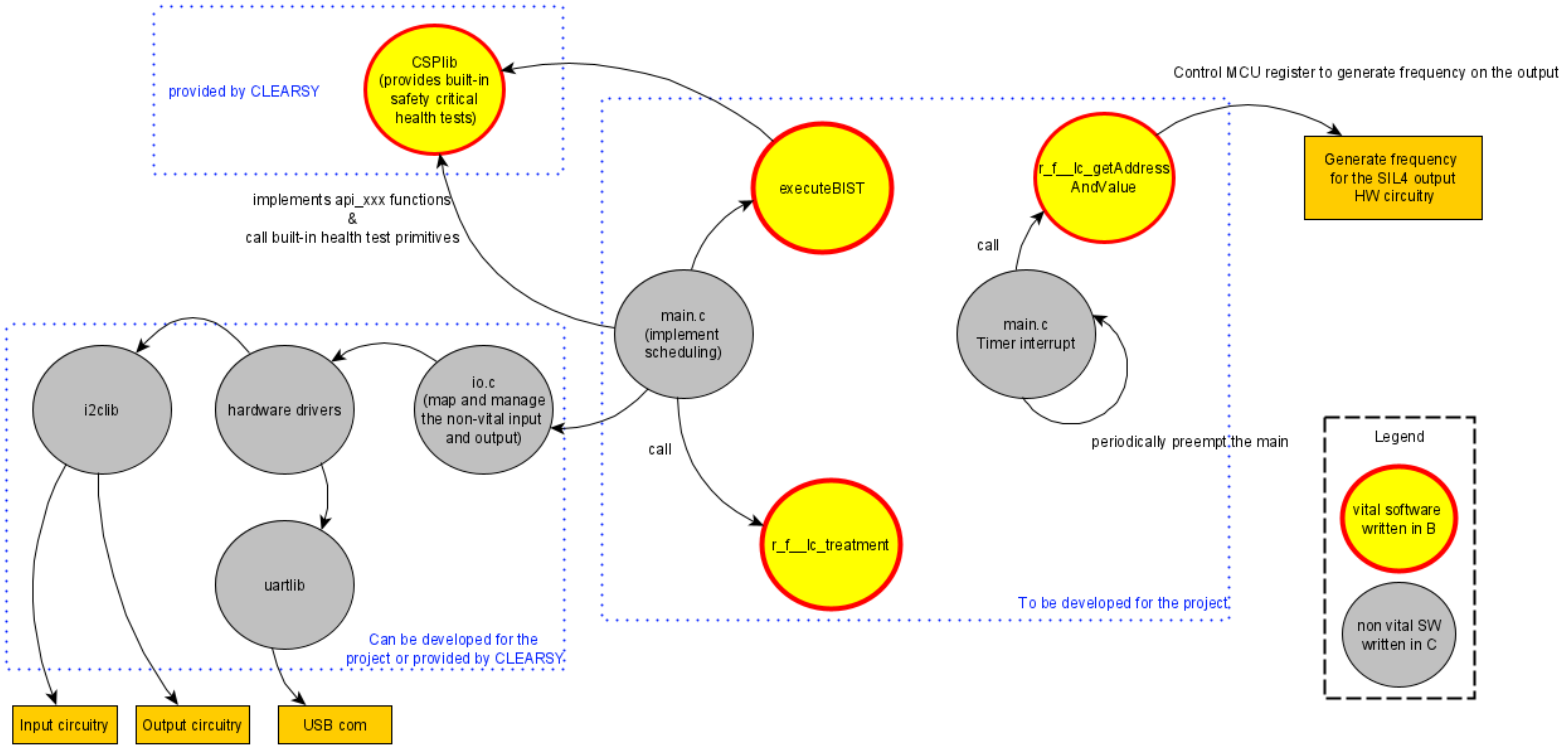
- nr_src: Non-replicated C source
- r_src: Replicated B0 source (vital software)
- Dcc_builds: generated folder (binary)
- Framework_library: contains a single .tar file with the certified precompiled binary
- Toolchain_cfg: contains configuration of Atelier B
- Toolchain_ressources: various files used for parametrization of the project
- Config file : located at the root of the project structure, provides the name and list of files and resources.



Software architecture

- The software will be scheduled by a single infinite-loop
  - Acquire non-vital input (non-vital SW)
  - Execute the B model
  - Apply the output of the signal (non-vital SW)
  - Execute the built-in self tests
- In addition some asynchronous events can modify the execution flow
  - Interrupt for managing non-vital communication (between processor)
  - Interrupt for non-vital IO management (I2C interface)
  - Interrupt for calling the B model in charge of the output

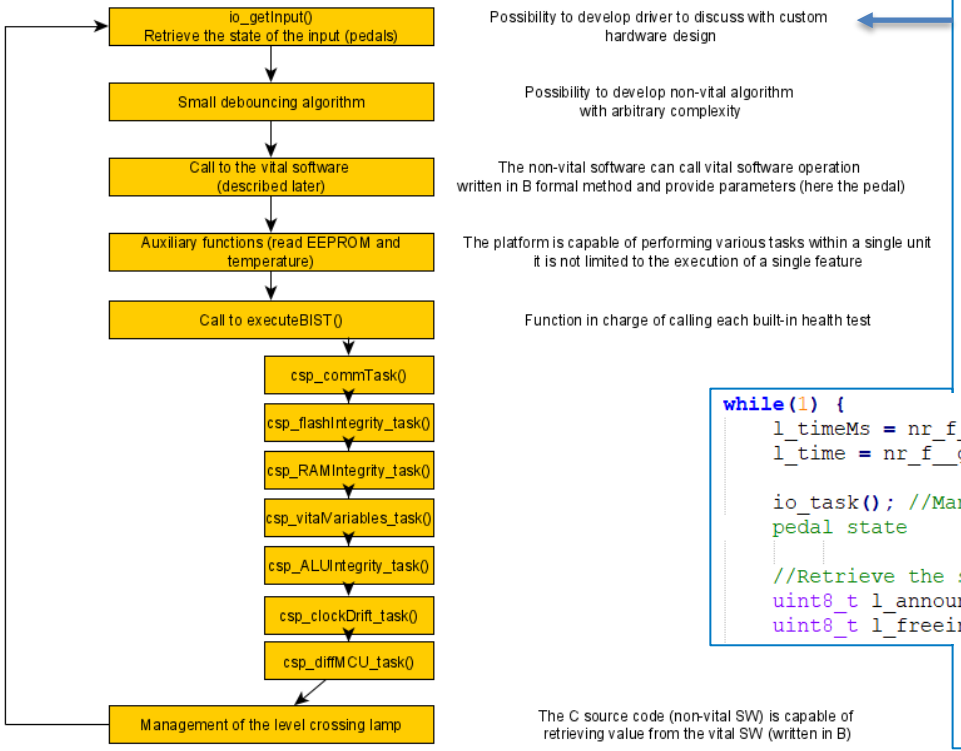
Component view



A few metrics

Type100	Number of line of code / model	Number of proof obligation	Overall contribution
CSPlib non vital software	9 029	-	78,4%
CSPlib vital software	9 392	2020	92%
Level crossing vital software	762	185	8%
Level crossing non vital software	218	-	1,6%
Level crossing non-vital driver	2 263	-	20,0%
TOTAL C source code	11 510		
TOTAL B source code	10 154	2205	

Scheduling (api_main function in main.c)



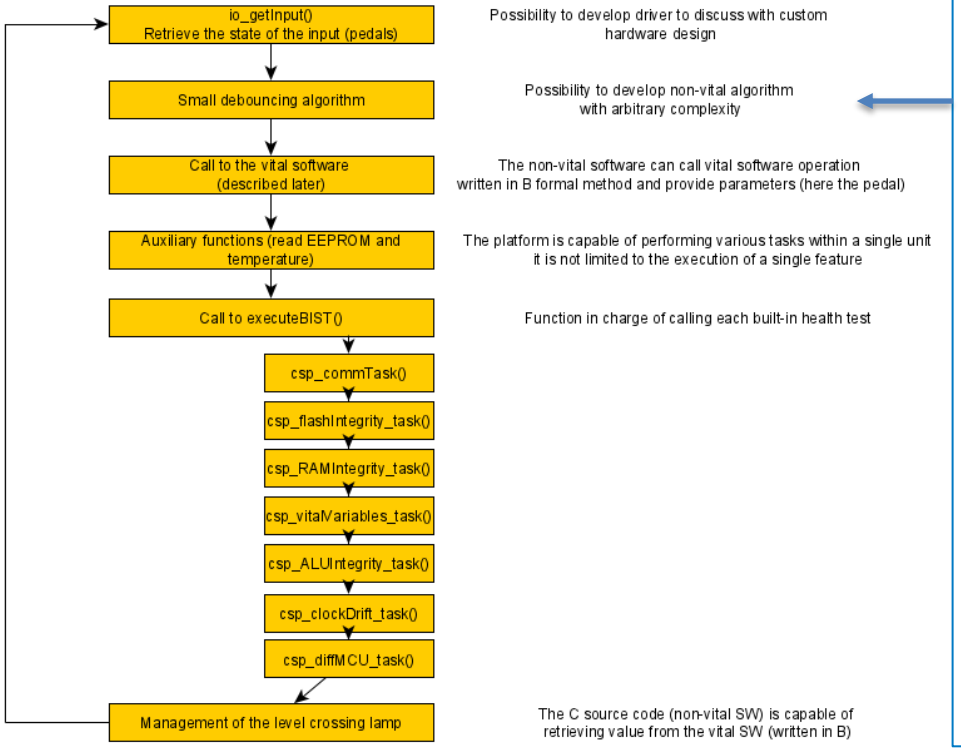
- Call the board specific package to communicate with I2C peripherals to retrieve the state of inputs (pedals)
- CLEARSY Safety Platform library provides primitive to get the system time in millisecond or ticks (125µs)

```
while(1) {
    l_timeMs = nr_f__getTimeMs();
    l_time = nr_f__getTime();

    io_task(); //Manage communication with the input device for acquiring the raw
    pedal state

    //Retrieve the state of the raw input
    uint8_t l_announcementPedalStateRaw = io_getInput(ANNOUNCEMENT_PEDAL_CHANNEL);
    uint8_t l_freeingPedalStateRaw = io_getInput(FREEING_PEDAL_CHANNEL);
```

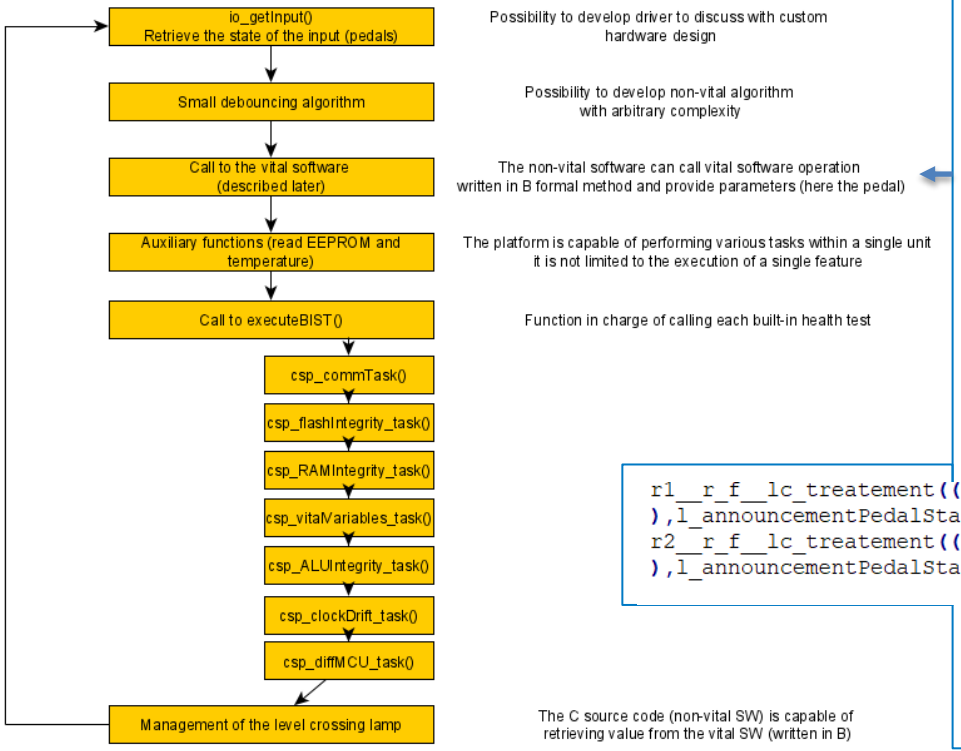
Scheduling (api_main function in main.c)




```
//Management of the filtering of the push button bouncing.
if(l_announcementPedalState == 0)
{
    if(l_announcementPedalStateRaw == 1)
    {
        l_announcementPedalState = l_announcementPedalStateRaw;
        l_lastAnnoucementPedalEdgeTime = l_timeMs;
    }
}
else
{
    if((l_announcementPedalStateRaw == 0) && ((l_timeMs -
    l_lastAnnoucementPedalEdgeTime) > BOUNCING_FILTER_TIME_MS))
    {
        l_announcementPedalState = l_announcementPedalStateRaw;
    }
}

if(l_freeingPedalState == 0)
{
    if(l_freeingPedalStateRaw == 1)
    {
        l_freeingPedalState = l_freeingPedalStateRaw;
        l_freeingPedalStateEdgeTime = l_timeMs;
    }
}
else
{
    if((l_freeingPedalStateRaw == 0) && ((l_timeMs -
    l_freeingPedalStateEdgeTime) > BOUNCING_FILTER_TIME_MS))
    {
        l_freeingPedalState = l_freeingPedalStateRaw;
    }
}
```

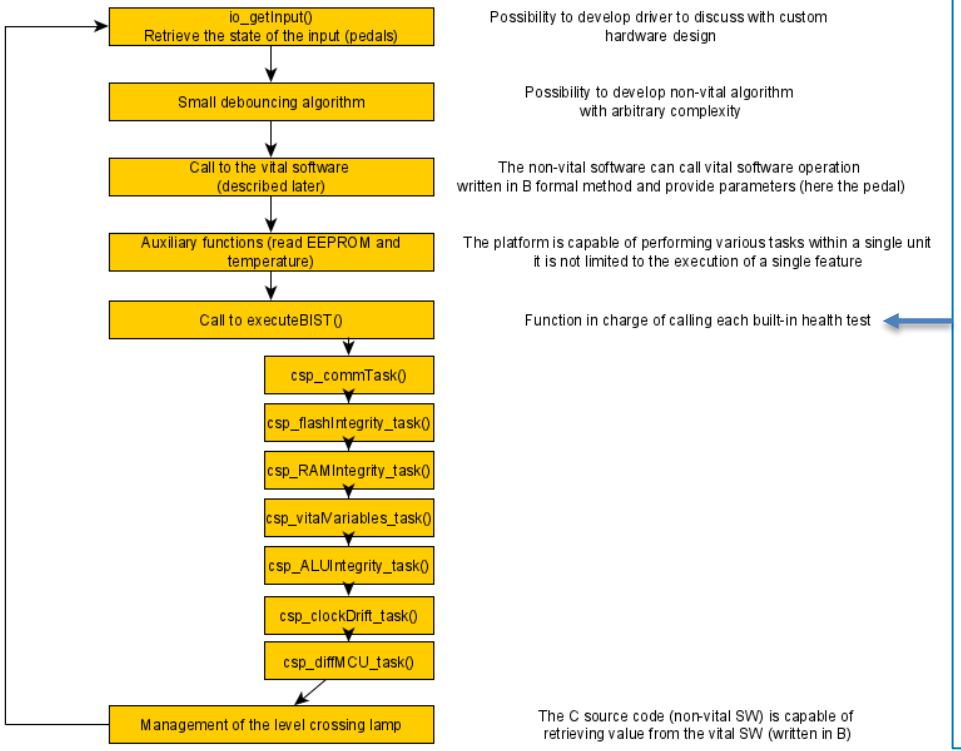
Scheduling (api_main function in main.c)



- Vital software is replicated so we need to call r1_function and r2_function in sequence.
- Time is encoded on 64 bits and need to be split into 2x32bits to the vital software.

```
r1__r_f__lc_treatement((uint32_t)(l_time >> 32),(uint32_t)(l_time & 0xffffffff
),l_announcementPedalState,l_freeingPedalState);
r2__r_f__lc_treatement((uint32_t)(l_time >> 32),(uint32_t)(l_time & 0xffffffff
),l_announcementPedalState,l_freeingPedalState);
```

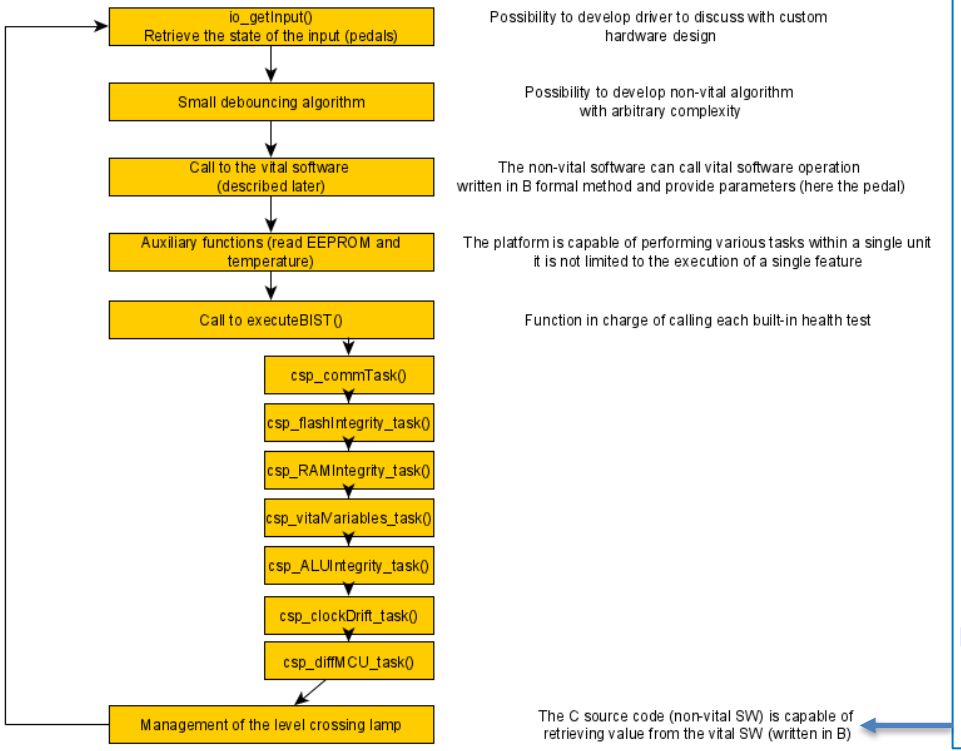
Scheduling (api_main function in main.c)



- The end user has to call frequently enough each built-in self test
- In this example all the built-in self test are called on every loop of the main function
- It is possible to call these functions with various patterns

```
void SECTION_NON_REPLICATED_NON_SHARED_FUNCTION execute_BIST()
{
    csp_commTask();
    csp_flashIntegrity_task();
    csp_RAMIntegrity_task();
    csp_vitalVariables_task();
    csp_ALUIntegrity_task();
    csp_clockDrift_task();
    csp_diffMCU_task();
}
```

Scheduling (api_main function in main.c)



- The lamp is controlled through regular C source code (non-vital)
- The non-vital software is capable of retrieving data from the B model with getter
- As it is non-vital we do not need to call getter of each replica

```
r1__r_f__lc_get_freeZone(&l_areaIsFree);
if(l_areaIsFree != 1) // The area is occupied the barrier shall be closed
(unpowered)
{
    io_setOutput(SIGNAL_OUTPUT_CHANNEL,true);
}
else
{
    io_setOutput(SIGNAL_OUTPUT_CHANNEL,false);
}
```

Interrupt and output generation

- Frequency generation is managed by a timer interrupt which is always running
  - Periodically the execution flow is pre-empted and this function is executed
  - It simply call get « safetyAnd » on two getters from the B model
- Getters returns a register address and the value that a given replica wants to write in a register (for applying a state on a given pin of the MCU)
- The csp_stdlib_safetyAnd, will compare and apply the value only if both replica are aligned
- This is one example of a safety critical vote between the two replicas

```
//Interrupt Service Routine in charge of the generation of the frequency on the vital
output pin.
void __ISR(_TIMER_4_VECTOR,ipl3SOFT) safetyFlasher_timer4ISR(void)
{
    static uint32_t timerCounter = 0;
    timerCounter++;
    if ((timerCounter & 0b01) == 0) {
#ifndef __CSP_DEBUG__
        csp_stdLib_safetyAnd(r1__r_f__lc_getSetAddressAndValue,&
        r2__r_f__lc_getSetAddressAndValue);
#endif
#ifdef __CSP_DEBUG__
        csp_stdLib_safetyAnd(&r1__r_f__lc_getSetAddressAndValue,&
        r2__r_f__lc_getSetAddressAndValue);
#endif
    } else {
#ifndef __CSP_DEBUG__
        csp_stdLib_safetyAnd(r1__r_f__lc_getClearAddressAndValue,&
        r2__r_f__lc_getClearAddressAndValue);
#endif
#ifdef __CSP_DEBUG__
        csp_stdLib_safetyAnd(&r1__r_f__lc_getClearAddressAndValue,&
        r2__r_f__lc_getClearAddressAndValue);
#endif
    }

    IFS0CLR = _IFS0_T4IF_MASK;
}
```

Formal Model

- In the abstract model the safety property is directly expressed:

```
ASSERTIONS
    (v_lc_zoneFree = TRUE
        =>
        0 : v_lc_carCount[v_lc_clock-C_lc_watchdogTimeoutPermissive..v_lc_clock])
```

  - v_lc_zoneFree is a boolean that indicates wheter a train is still present or not between the announcement and the freeing pedal.
  - v_lc_carCount is a function from the time (tick system) to integer, that reflects the number of car that have been detected at each time step of the system
  - V_lc_clock: the current tick of the system
  - C_lc_watchdogTimeoutPermissive: constant which represent (as a first approximation) the time for the fastest train to travel between announcement pedal and level crossing (i.e. 1 second in our example).

B formal model : counting the axle

- Count occurs only if parameter are correct
- B model detects transitions of the pedals state
- Watchdog is updated only if the number of axle does not exceed 8.

```
r_f__lc_treatement(p_clockH, p_clockL, p_inLc, p_outLc) =
BEGIN
    SELECT p_inLc : 0..1 & p_outLc : 0..1
    THEN
        ANY n_in, n_out
        WHERE
            n_in : 0..1 &
            (n_in = 1 <=> (p_inLc = 1 & v_lc_inPrev = 0)) &
            n_out : 0..1 &
            (n_out = 1 <=> (p_outLc = 1 & v_lc_outPrev = 0))
        THEN
            SELECT
                v_lc_carCount(max(dom(v_lc_carCount))) + n_in - n_out : -8..8
            THEN
                v_lc_watchdogTimeout :(v_lc_watchdogTimeout : uint64_t
                    & v_lc_watchdogTimeout : v_lc_clock..v_lc_clock+C_lc_watchdogTimeoutPermissive) ||
                v_lc_carCount(v_lc_clock) := v_lc_carCount(max(dom(v_lc_carCount))) + n_in - n_out ||
                v_lc_zoneFree := bool(v_lc_carCount(max(dom(v_lc_carCount))) + n_in - n_out = 0) ||
                v_lc_inPrev := p_inLc ||
                v_lc_outPrev := p_outLc
            END
        END
    WHEN
        0=0
    THEN
        skip
    END
END;
```

Applying the state

- Provides the internal states (v_lc_zoneFree) of the model
  - Output inhibition is a flag from the library which is false until all the built in self test passed.

```
p_address, p_value <-- r_f__lc_getSetAddressAndValue =
BEGIN
    p_address := addrUint(LATESET) ||
    IF v_lc_zoneFree = TRUE
        & v_cs0_outputInhibition = SFALSE
    THEN
        p_value := C_lc_outputMask
    ELSE
        p_value := 0
    END
END;
```

Project configuration

- Single text file in Makefile format
  - Provides the list of the source
  - Provides the path to the Manifest of the B Project
  - Provides the list of the software library used by the project
  - Allows fine tunning of compilation options

```
# Project name
PROJET = ABZTUTORIAL

# Non replicated sources MCU dependant
SOURCES_NR_MCU = \
    nr_src/abztutorial_main.c \
    nr_src/abztutorial_nonvital_tmp_eeprom.h \
    nr_src/io.h \
    nr_src/csp_abztutorial_debug.h \
    nr_src/flash.c \
    nr_src/flash.h

# Non replicated sources specific to MCU1
SOURCES_NR_MCU1 =

# Non replicated sources specific to MCU2
SOURCES_NR_MCU2 =

# Non replicated sources
SOURCES_NR = \
    nr_src/csp_abztutorial_debug.h \
    nr_src/csp_abztutorial_debug.c \
    nr_src/csp_abztutorial_nrv.h \
    nr_src/csp_abztutorial_nrv.c \
    nr_src/csp_abztutorial_register.h \
    nr_src/csp_abztutorial_register.c \
    nr_src/pca6416a.c \
    nr_src/pca6416a.h \
    nr_src/io.c \
```

Mini summary

- Source code is structured between vital (B model) and non-vital (C language)
  - No limitation in term of algorithm complexity
  - Capable of adding driver for custom hardware
  - C source code can trigger B model operations (if prefixed by r_f__)
    - Be carreful these operations are replicated and need to be called twice
  - Scheduling design choices are free for the software designer
  - C source code can retrieve state from the B model through getter
  - B model can read variables from the C source code
    - By operation parameter
    - By naming convention : non replicated variable nr_v__ (not illustrated here)
- Interrupts are available and can be used with both vital and non-vital software

Compilation and loading

Video « Compile and load»

Running the example

Video « CS0 nominal»

Dysfunctional use case

- Example illustrated in this presentation
  - Safety built-in test not called frequently enough
  - Slowing down of the execution due to CPU overload
  - Corruption of flash (program) memory
  - Corruption of the RAM
  - Systematic design failure

Dysfunctional use case BIST not called

- What happen if the built in self tests are not called frequently enough
  - If the timing constraints are not met and at least one test timed out the system will shut down
- Scheduling is delegated to end user but safety is ensured whatever the implementation

```
void SECTION_NON_REPLICATED_NON_SHARED_FUNCTION execute_BIST()
{
    csp_commTask();
    csp_flashIntegrity_task();
    csp_RAMIntegrity_task();
    csp_vitalVariables_task();
    csp_ALUIntegrity_task();
    csp_clockDrift_task();
    csp_diffMCU_task();
}
```

```
void SECTION_NON_REPLICATED_NON_SHARED_FUNCTION failure_delayCallToBIST()
{


    if(nr_f__getTimeMs() <= 15000)
    {
        execute_BIST();
    }
    else
    {
        csp_commTask();
        csp_flashIntegrity_task();
        csp_RAMIntegrity_task();
        //csp_vitalVariables_task();
        csp_ALUIntegrity_task();
        csp_clockDrift_task();
        csp_diffMCU_task();
    }
}
```

Dysfunctional case : slowing down of the execution

- Scenario
  - A specific branch or external cause, requires extra processing and the vital software is not executed frequently enough
- Associated risk
  - Entering axle is detected and counted with a delay longer than the travel time from announcement pedal to level crossing → collision hazard
- Behavior of the system
  - If timing constraints are not met the system shutdown and reach the safe state (barrier closed). This is the result of both the platform design and the exported constraint SRAC01.

```
void SECTION_NON_REPLICATED_NON_SHARED_FUNCTION failure_delayProcessing(uint8_t
p_announcementPedal, uint8_t p_freeingPedal)
{
    static uint64_t s_nextExecutionTime = 0;
    static uint32_t s_delay = 100;
    uint64_t l_date = nr_f__getTimeMs();

    if(l_date >= s_nextExecutionTime)
    {
        uint64_t l_time = nr_f__getTime();
        r1__r_f__lc_treatement((uint32_t)(l_time >> 32),(uint32_t)(l_time & 0xffffffff
        ),p_announcementPedal,p_freeingPedal);
        r2__r_f__lc_treatement((uint32_t)(l_time >> 32),(uint32_t)(l_time & 0xffffffff
        ),p_announcementPedal,p_freeingPedal);
    }

    if(l_date <= 15000) //Normal execution during the first 15s.
    {
        s_nextExecutionTime = l_date;
    }
    else // delayed execution
    {
        if(l_date >= s_nextExecutionTime)
        {
            s_nextExecutionTime = l_date + s_delay;
            s_delay += 100;
        }
    }
}
```

Dysfunctional case : corruption of the FLASH

- Scenario
  - During the life of the product the program memory gets corrupted and the binary executed on the target does not correspond to the one designed and validated
- Associated risk
  - A corrupted binary may behave in any dangerous way
- Behavior of the system
  - The system will shutdown when it detects program memory (FLASH) corruption

Memory content before

```
20811 :10e44000ffffffffffffffffffffffffffffffdc
20812 :10e45000ffffffffffffffffffffffffffffffcc
20813 :10e46000ffffffffffffffffffffffffffffffbc
20814 :10e47000ffffffffffffffffffffffffffffffac
20815 :10e48000ffffffffffffffffffffffffffffff9c
```

Memory content after corruption

```
20811 :10e44000ffffffffffffffffffffffffffffffdc
20812 :10e45000ffffffffffffffffffffffffffffffcc
20813 :10e46000ffffffffffffffffffffffffffffffebd
20814 :10e47000ffffffffffffffffffffffffffffffac
20815 :10e48000ffffffffffffffffffffffffffffff9c
```

Dysfunctional case : corruption of the RAM

- Scenario
  - Cosmic ray or perturbation alter the RAM content during runtime
  - Or a systematic failure generate a divergent behavior on one replica
- Associated risk
  - The variable could be the Boolean flag used for determining if the area is occupied or not → collision hazard
- Behavior of the system
  - The system will detect the corruption and shutdown ensuring a safe state.

Modification of api_main.c

```
if(l_timeMs > 15000)
{
    failure_corruptRAM();
}

execute_BIST();
```

Simulation of the RAM perturbation

```
typedef struct _csp_abztutorial_struct_safety_vars
{
uint32_t rv_lc_watchdogTimeoutH;
uint32_t rv_lc_watchdogTimeoutL;
uint8_t rv_lc_carCount;
uint8_t rv_lc_zoneFree;
uint8_t rv_lc_inPrev;
uint8_t rv_lc_outPrev;
}safety_vars;


void SECTION_NON_REPLICATED_NON_SHARED_FUNCTION failure_corruptRAM()
{
    safety_vars * l_memory = (safety_vars *)(0xa0005ed8); //Determined by reverse
    engineering not available by default
    l_memory->rv_lc_zoneFree = 0xCA; //Arbitrary value that is not a boolean
}
```

Dysfunctional case : systematic design failure

- Scenario
  - Software is not formally proven and the 8 axles overflow condition is not properly managed.
  - Considering that a train has 6 axles, we can imagine that due to track maintenance operation 3 axles have been detected enterring the area but never leave. On the sunset a 6 axles train enter in the area causing the variable to overflow and result into a counter equal to 0 and an open barrier.
- Associated risk
  - The barrier is open whereas there is a train on the level crossing→ collision hazard
- Behavior of the system
  - The system will not react to this event. That is why the B0 proof asks for a safety guaranteed software implementation which is achieve in our case by a formal proof that cannot pass if the overflow case is not properly managed.

Dysfunctional case

Video « dysfunctional use case»