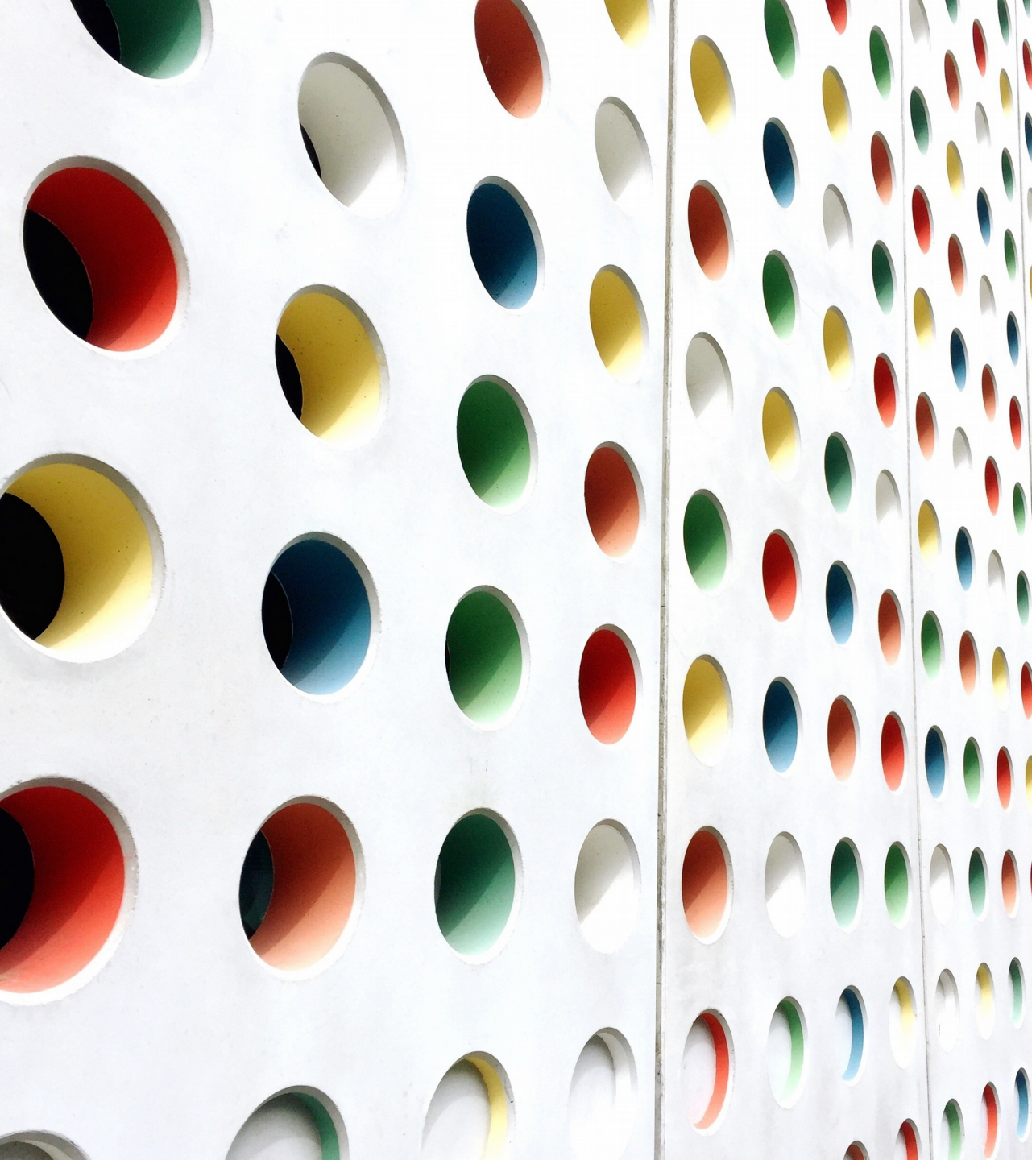


Workshop 3 Presentation: Implementation of Portenta H7 based CAN Bus Simulation & Fault Injection Tool

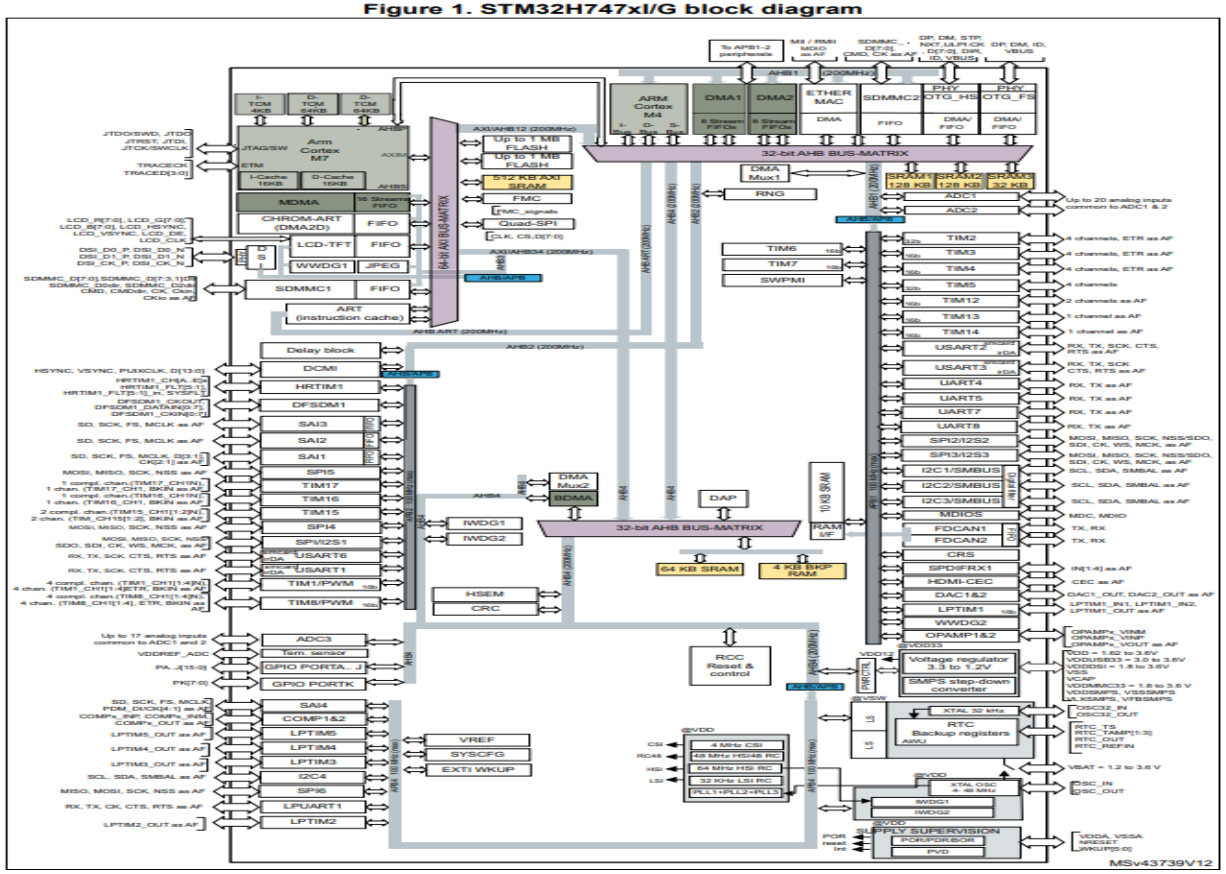
NGUYEN DANG TIEN LOI - 21009

Tool Specifications

- CAN Bus Simulation:
  - Classic CAN
  - Bit rate 250000
- Fault Injection
  - Manipulating Parameters
  - Adding Delay
  - Ignore

Micro controller

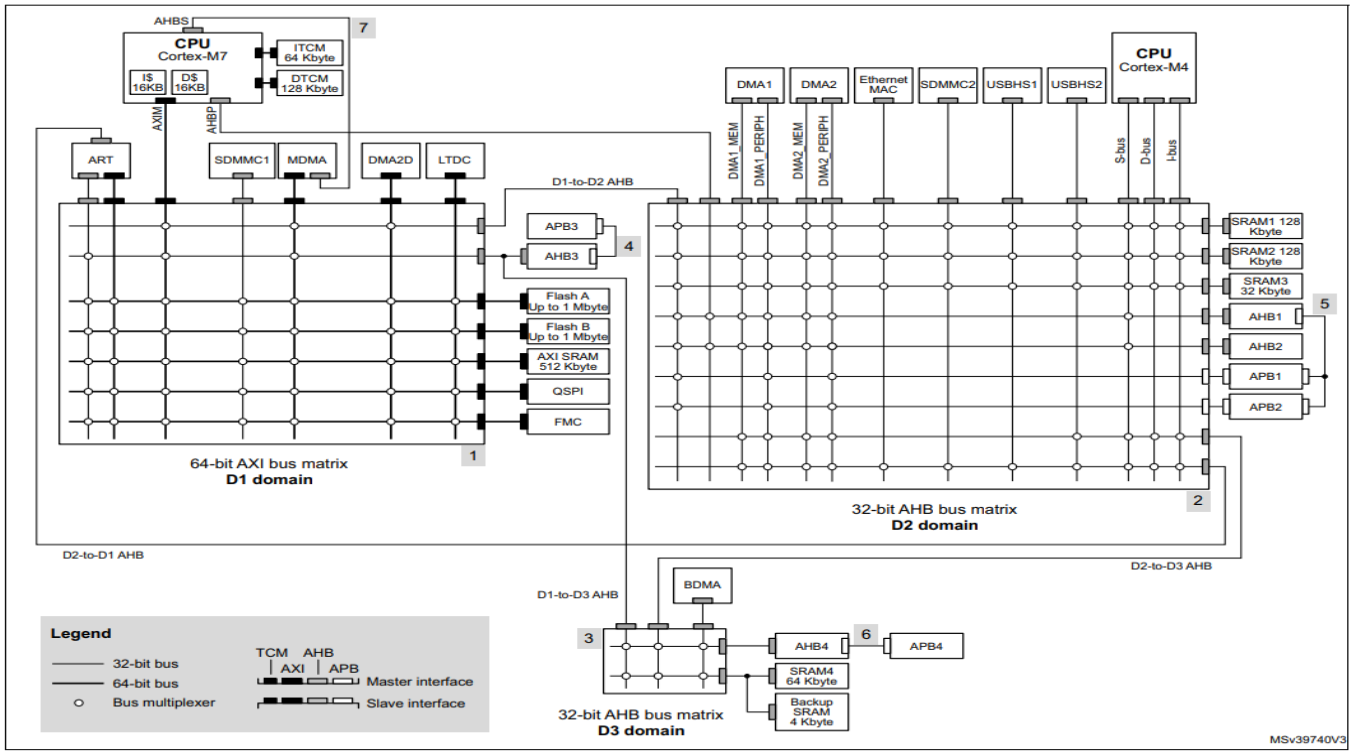
Figure 1. STM32H747xI/G block diagram



Source: (2)

Bus Matrix

Figure 5. STM32H747xI/G bus matrix



Source: (2)

Clock generation block

Clock generation block

- Generation and dispatching of clocks for the complete device
- 3 separate PLLs using integer or fractional ratios
- Possibility to change the PLL fractional ratios on-the-fly
- Smart clock gating to reduce power dissipation
- 2 external oscillators:
  - High-speed external oscillator (HSE) supporting a wide range of crystals from 4 to 48 MHz frequency
  - Low-speed external oscillator (LSE) for the 32 kHz crystals
- 4 internal oscillators
  - High-speed internal oscillator (HSI)
  - 48 MHz RC oscillator (HSI48)
  - Low-power Internal oscillator (CSI)
  - Low-speed internal oscillator (LSI)

Source: (3)

Arduino Setup

Magic Number (validation): a0
Bootloader version: 22
Clock source: External oscillator
USB Speed: USB 2.0/Hi-Speed (480 Mbps)
Has Ethernet: Yes
Has WiFi module: Yes
RAM size: 8 MB
QSPI size: 16 MB
Has Video output: Yes
Has Crypto chip: Yes

Linker Script at /home/tienloi/.arduino15/packages/arduino/hardware/mbed_portenta/2.3.1/variants/PORTENTA_H7_M7/linker_script.ld

Vector Table at /home/tienloi/.arduino15/packages/arduino/hardware/mbed_portenta/2.3.1/cores/arduino/mbed/targets/TARGET_STM/TARGET_STM32H7/STM32Cube_FW/CMSIS/stm32h747xx.h

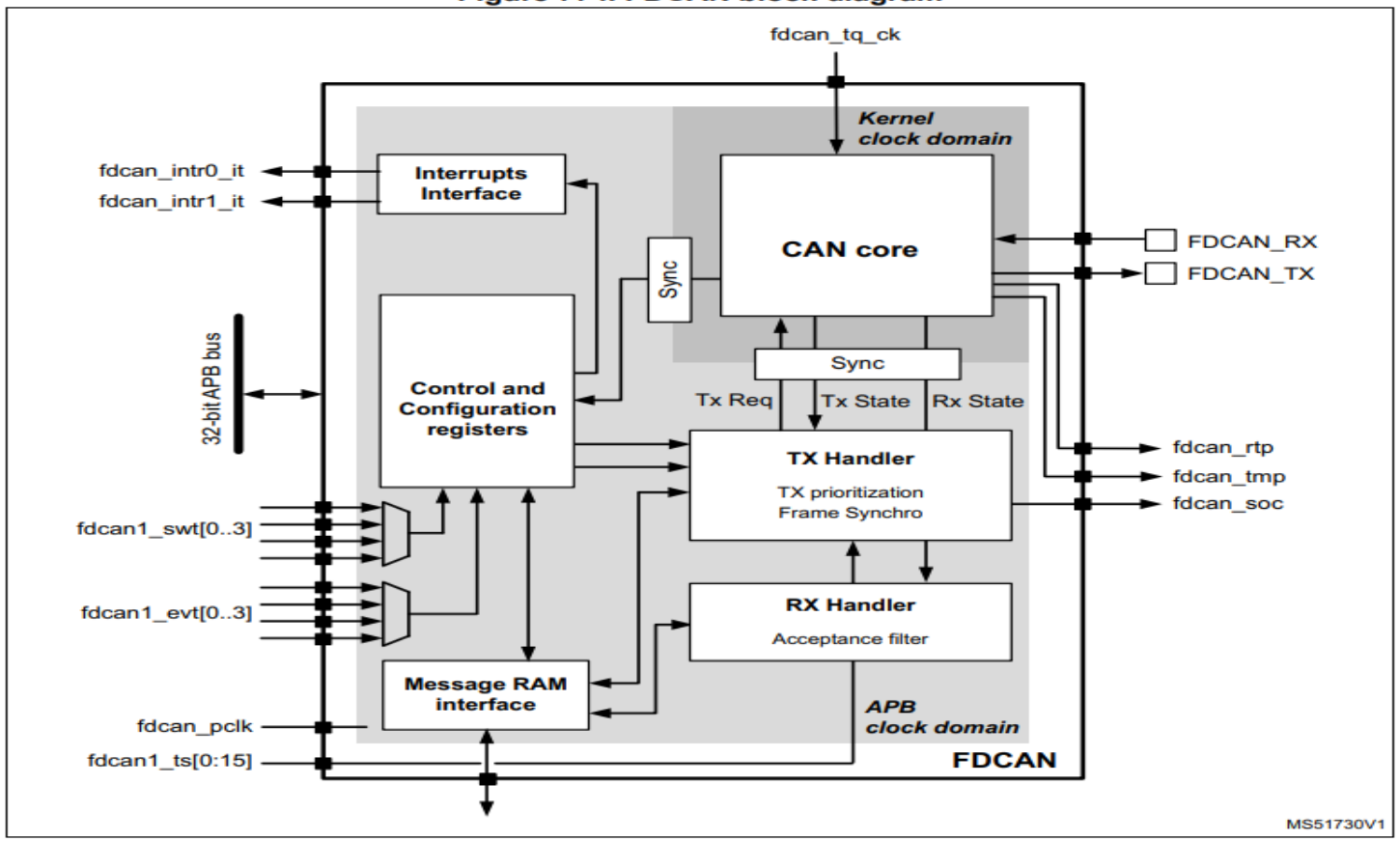
FDCAN Implementation

- CAN Pin: PB_8, PH_13 (CAN0)
- Clock Source: PLL
- Enable TimestampCounter & TimeoutCounter
- IT: RX_FIFO0_NEW_MSG, TIMEOUT_OCCURRED, TIMESTAMP_WRAPAROUND, TX_COMPLETE

https://github.com/meomotminh/CAN-HAL/blob/main/HAL CAN/CAN INIT/CAN_Init/src/FDCAN/FDCAN.cpp

FDCAN peripheral

Figure 774. FDCAN block diagram



Source: (3)

FDCAN HAL Instruction

31.2.1 **How to use this driver**

1. Initialize the FDCAN peripheral using HAL_FDCAN_Init function.
2. If needed , configure the reception filters and optional features using the following configuration functions:
   - HAL_FDCAN_ConfigClockCalibration
   - HAL_FDCAN_ConfigFilter
   - HAL_FDCAN_ConfigGlobalFilter
   - HAL_FDCAN_ConfigExtendedIdMask
   - HAL_FDCAN_ConfigRxFifoOverwrite
   - HAL_FDCAN_ConfigFifoWatermark
   - HAL_FDCAN_ConfigRamWatchdog
   - HAL_FDCAN_ConfigTimestampCounter
   - HAL_FDCAN_EnableTimestampCounter
   - HAL_FDCAN_DisableTimestampCounter
   - HAL_FDCAN_ConfigTimeoutCounter
   - HAL_FDCAN_EnableTimeoutCounter
   - HAL_FDCAN_DisableTimeoutCounter
   - HAL_FDCAN_ConfigTxDelayCompensation
   - HAL_FDCAN_EnableTxDelayCompensation
   - HAL_FDCAN_DisableTxDelayCompensation
   - HAL_FDCAN_EnableISOMode
   - HAL_FDCAN_DisableISOMode
   - HAL_FDCAN_EnableEdgeFiltering
   - HAL_FDCAN_DisableEdgeFiltering
   - HAL_FDCAN_TT_ConfigOperation
   - HAL_FDCAN_TT_ConfigReferenceMessage
   - HAL_FDCAN_TT_ConfigTrigger
3. Start the FDCAN module using HAL_FDCAN_Start function. At this level the node is active on the bus: it can send and receive messages.
4. The following Tx control functions can only be called when the FDCAN module is started:
   - HAL_FDCAN_AddMessageToTxFifoQ
   - HAL_FDCAN_EnableTxBufferRequest
   - HAL_FDCAN_AbortTxRequest
5. After having submitted a Tx request in Tx Fifo or Queue, it is possible to get Tx buffer location used to place the Tx request thanks to HAL_FDCAN_GetLatestTxFifoQRequestBuffer API. It is then possible to abort later on the corresponding Tx Request using HAL_FDCAN_AbortTxRequest API.
6. When a message is received into the FDCAN message RAM, it can be retrieved using the HAL_FDCAN_GetRxMessage function.
7. Calling the HAL_FDCAN_Stop function stops the FDCAN module by entering it to initialization mode and re-enabling access to configuration registers through the configuration functions listed here above.
8. All other control functions can be called any time after initialization phase, no matter if the FDCAN module is started or stoped.

Polling mode operation

1. Reception and transmission states can be monitored via the following functions:
   - HAL_FDCAN_IsRxBufferMessageAvailable
   - HAL_FDCAN_IsTxBufferMessagePending
   - HAL_FDCAN_GetRxFifoFillLevel
   - HAL_FDCAN_GetTxFifoFreeLevel

Interrupt mode operation

1. There are two interrupt lines: line 0 and 1. By default, all interrupts are assigned to line 0. Interrupt lines can be configured using HAL_FDCAN_ConfigInterruptLines function.
2. Notifications are activated using HAL_FDCAN_ActivateNotification function. Then, the process can be controlled through one of the available user callbacks: HAL_FDCAN_xxxCallback.

Source: (1)

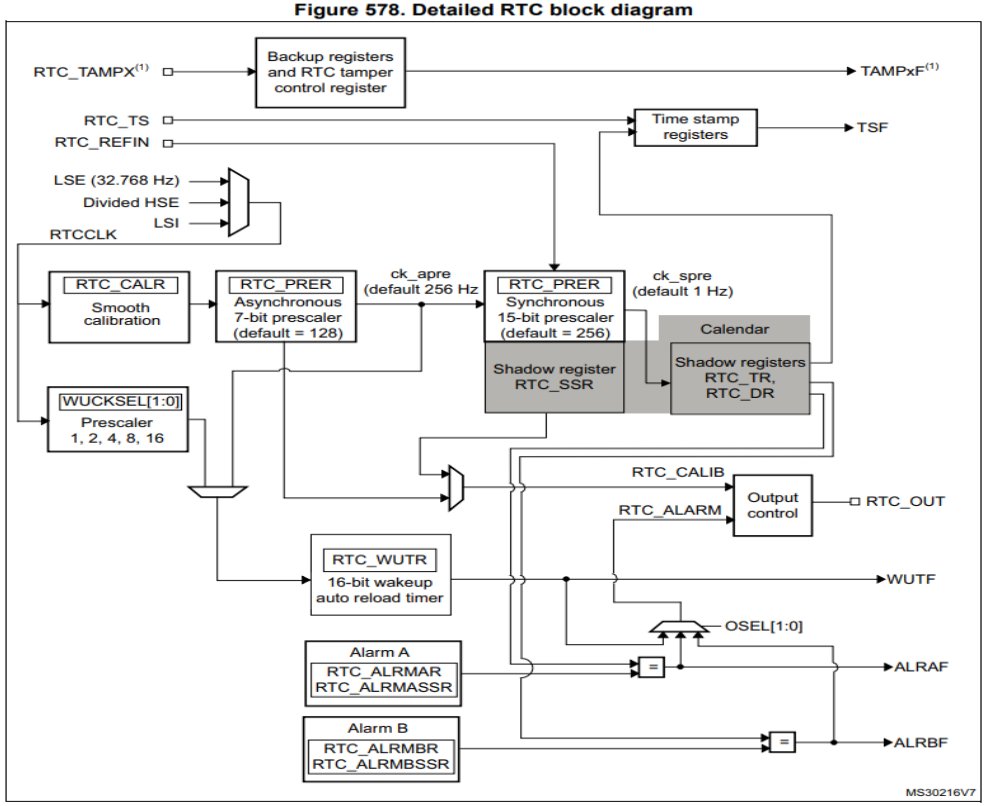
FDCAN Implementation

- CAN Pin: PB_8, PH_13 (CAN0)
- Clock Source: PLL
- Enable TimestampCounter & TimeoutCounter
- IT: RX_FIFO0_NEW_MSG, TIMEOUT_OCCURRED, TIMESTAMP_WRAPAROUND, TX_COMPLETE

https://github.com/meomotminh/CAN-HAL/blob/main/HAL CAN/CAN INIT/CAN_Init/src/FDCAN/FDCAN.cpp

RTC peripheral

Figure 578. Detailed RTC block diagram



1. x is an integer index starting from 1, the number of tampers depends on devices.

Source: (3)

RTC HAL Instruction

72.2.4 How to use RTC Driver

- Enable the RTC domain access (see description in the section above).
- Configure the RTC Prescaler (Asynchronous and Synchronous) and RTC hour format using the HAL_RTC_Init() function.

Time and Date configuration

- To configure the RTC Calendar (Time and Date) use the HAL_RTC_SetTime() and HAL_RTC_SetDate() functions.
- To read the RTC Calendar, use the HAL_RTC_GetTime() and HAL_RTC_GetDate() functions.

Alarm configuration

- To configure the RTC Alarm use the HAL_RTC_SetAlarm() function. You can also configure the RTC Alarm with interrupt mode using the HAL_RTC_SetAlarm_IT() function.
- To read the RTC Alarm, use the HAL_RTC_GetAlarm() function.

Source: (1)

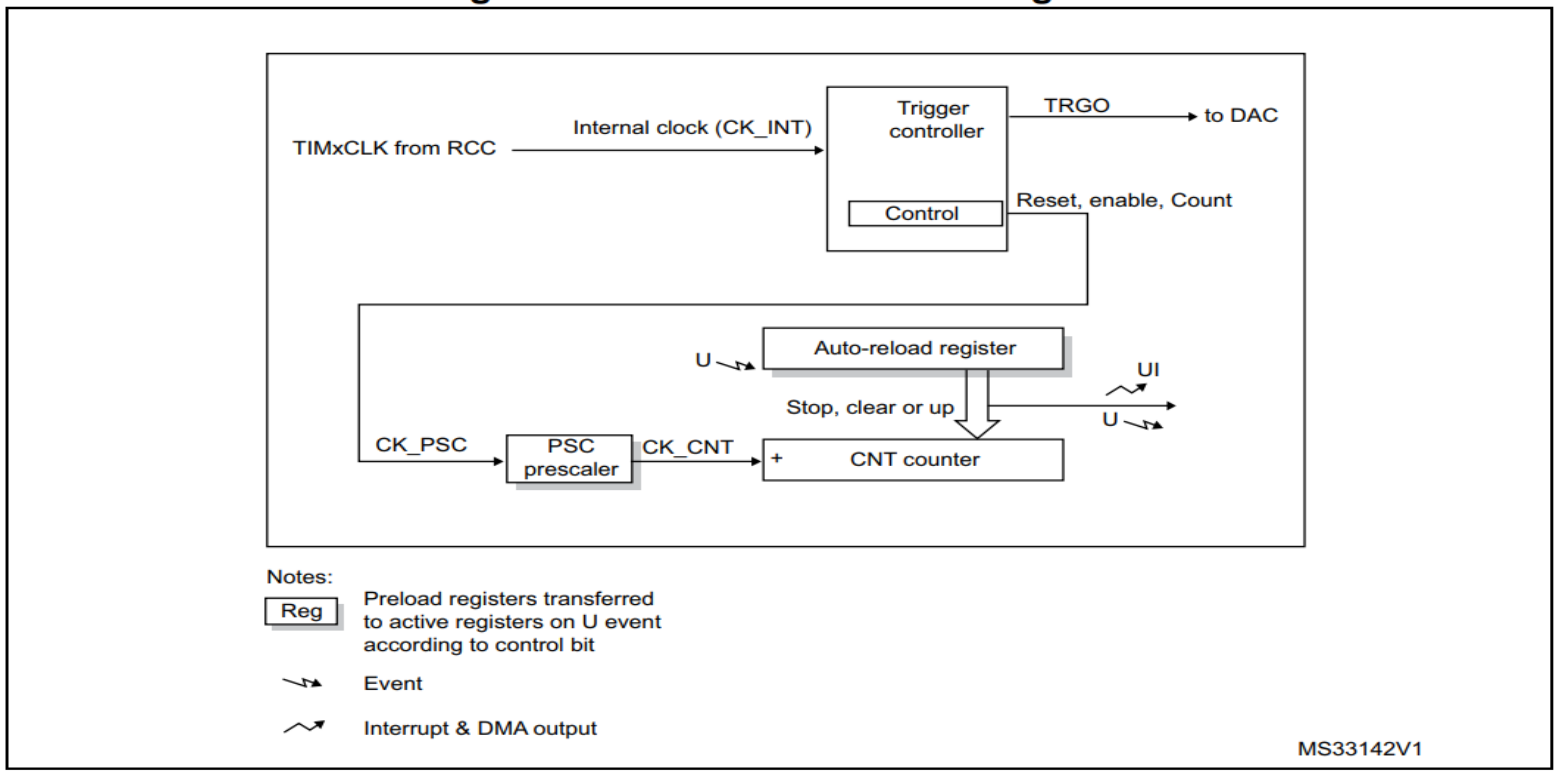
RTC Implementation

- HourFormat: 24
- Set Time: 12:45:00
- Set Date: 12 June 2018 (Tuesday)
- Clock Source: LSE
- Enable EXTI Line 17 for RTC Alarm
- IT: RTC_Alarm

https://github.com/meomotminh/CAN-HAL/blob/main/HAL CAN/CAN INIT/CAN_Init/src/RTC/RTC.cpp

TIMER peripheral

Figure 554. Basic timer block diagram



Source: (3)

TIMER HAL Instruction

87.2.2 How to use this driver

1. Initialize the TIM low level resources by implementing the following functions depending on the selected feature:
   - Time Base : HAL_TIM_Base_MspInit()
   - Input Capture : HAL_TIM_IC_MspInit()
   - Output Compare : HAL_TIM_OC_MspInit()
   - PWM generation : HAL_TIM_PWM_MspInit()
   - One-pulse mode output : HAL_TIM_OnePulse_MspInit()
   - Encoder mode output : HAL_TIM_Encoder_MspInit()

2. Initialize the TIM low level resources :
   a. Enable the TIM interface clock using __HAL_RCC_TIMx_CLK_ENABLE();
   b. TIM pins configuration
      - Enable the clock for the TIM GPIOs using the following function: __HAL_RCC_GPIOx_CLK_ENABLE();
      - Configure these TIM pins in Alternate function mode using HAL_GPIO_Init();

   The external Clock can be configured, if needed (the default clock is the internal clock from the APBx), using the following function: HAL_TIM_ConfigClockSource, the clock configuration should be done before any start function.

   Configure the TIM in the desired functioning mode using one of the Initialization function of this driver:
   - HAL_TIM_Base_Init: to use the Timer to generate a simple time base
   - HAL_TIM_OC_Init and HAL_TIM_OC_ConfigChannel: to use the Timer to generate an Output Compare signal.
   - HAL_TIM_PWM_Init and HAL_TIM_PWM_ConfigChannel: to use the Timer to generate a PWM signal.
   - HAL_TIM_IC_Init and HAL_TIM_IC_ConfigChannel: to use the Timer to measure an external signal.
   - HAL_TIM_OnePulse_Init and HAL_TIM_OnePulse_ConfigChannel: to use the Timer in One Pulse Mode.
   - HAL_TIM_Encoder_Init: to use the Timer Encoder Interface.

   Activate the TIM peripheral using one of the start functions depending from the feature used:
   - Time Base : HAL_TIM_Base_Start(), HAL_TIM_Base_Start_DMA(), HAL_TIM_Base_Start_IT()
   - Input Capture : HAL_TIM_IC_Start(), HAL_TIM_IC_Start_DMA(), HAL_TIM_IC_Start_IT()
   - Output Compare : HAL_TIM_OC_Start(), HAL_TIM_OC_Start_DMA(), HAL_TIM_OC_Start_IT()
   - PWM generation : HAL_TIM_PWM_Start(), HAL_TIM_PWM_Start_DMA(), HAL_TIM_PWM_Start_IT()
   - One-pulse mode output : HAL_TIM_OnePulse_Start(), HAL_TIM_OnePulse_Start_IT()
   - Encoder mode output : HAL_TIM_Encoder_Start(), HAL_TIM_Encoder_Start_DMA(), HAL_TIM_Encoder_Start_IT().

6. The DMA Burst is managed with the two following functions: HAL_TIM_DMABurst_WriteStart() HAL_TIM_DMABurst_ReadStart()

Source: (1)

TIMER Implementation

- Timer 6
- Bus APB1
- Period : TIM6CLK / 1000
- IT: TIM6_DAC

https://github.com/meomotminh/CAN-HAL/blob/main/HAL CAN/CAN INIT/CAN_Init/CAN_Init.ino

SRAM

The embedded system SRAM is divided into up to five blocks over the three power domains:

- D1 domain, AXI SRAM:
  - AXI SRAM is mapped at address 0x2400 0000 and accessible by all system masters except BDMA through D1 domain AXI bus matrix. AXI SRAM can be used for application data which are not allocated in DTCM RAM or reserved for graphic objects (such as frame buffers)
- D2 domain, AHB SRAM:
  - AHB SRAM1 is mapped at address 0x3000 0000 and accessible by all system masters except BDMA through D2 domain AHB matrix. AHB SRAM1 can be used as DMA buffers to store peripheral input/output data in D2 domain, or as code location for Cortex®-M4 CPU (application code available when D1 is powered off.
  - AHB SRAM2 is mapped at address 0x3002 0000 and accessible by all system masters except BDMA through D2 domain AHB matrix. AHB SRAM2 can be used as DMA buffers to store peripheral input/output data in D2 domain, or as read-write segment for application running on Cortex®-M4 CPU.
  - AHB SRAM3 is mapped at address 0x3004 0000 and accessible by all system masters except BDMA through D2 domain AHB matrix. AHB SRAM3 can be used as buffers to store peripheral input/output data for Ethernet and USB, or as shared memory between the two cores.
- D3 domain, AHB SRAM:
  - AHB SRAM4 is mapped at address 0x3800 0000 and accessible by most of system masters through D3 domain AHB matrix. AHB SRAM4 can be used as BDMA buffers to store peripheral input/output data in D3 domain. It can also be used to retain some application code/data when D1 and D2 domain enter DStandby mode, or as shared memory between the two cores.

Source: (3)

SRAM HAL Instruction

85.2.1 How to use this driver

This driver is a generic layered driver which contains a set of APIs used to control SRAM memories. It uses the FMC layer functions to interface with SRAM devices. The following sequence should be followed to configure the FMC to interface with SRAM/PSRAM memories:

1. Declare a SRAM_HandleTypeDef handle structure, for example: SRAM_HandleTypeDef hsram; and:
   - Fill the SRAM_HandleTypeDef handle "Init" field with the allowed values of the structure member.
   - Fill the SRAM_HandleTypeDef handle "Instance" field with a predefined base register instance for NOR or SRAM device
   - Fill the SRAM_HandleTypeDef handle "Extended" field with a predefined base register instance for NOR or SRAM extended mode
2. Declare two FMC_NORSRAM_TimingTypeDef structures, for both normal and extended mode timings; for example: FMC_NORSRAM_TimingTypeDef Timing and FMC_NORSRAM_TimingTypeDef ExTiming; and fill its fields with the allowed values of the structure member.
3. Initialize the SRAM Controller by calling the function HAL_SRAM_Init(). This function performs the following sequence:
   a. MSP hardware layer configuration using the function HAL_SRAM_MspInit()
   b. Control register configuration using the FMC NORSRAM interface function FMC_NORSRAM_Init()
   c. Timing register configuration using the FMC NORSRAM interface function FMC_NORSRAM_Timing_Init()
   d. Extended mode Timing register configuration using the FMC NORSRAM interface function FMC_NORSRAM_Extended_Timing_Init()
   e. Enable the SRAM device using the macro __FMC_NORSRAM_ENABLE()
4. At this stage you can perform read/write accesses from/to the memory connected to the NOR/SRAM Bank. You can perform either polling or DMA transfer using the following APIs:
   - HAL_SRAM_Read()/HAL_SRAM_Write() for polling read/write access
   - HAL_SRAM_Read_DMA()/HAL_SRAM_Write_DMA() for DMA read/write transfer
5. You can also control the SRAM device by calling the control APIs HAL_SRAM_WriteOperation_Enable()/ HAL_SRAM_WriteOperation_Disable() to respectively enable/disable the SRAM write operation
6. You can continuously monitor the SRAM device HAL state by calling the function HAL_SRAM_GetState()

Callback registration

Source: (1)

SRAM Implementation

- SRAM1
- Config FMC_NORSTAM_BANK1 (not used)

https://github.com/meomotminh/CAN-HAL/blob/main/HAL CAN/CAN INIT/CAN_Init/src/loiTruck/loiTruck.h

Fault Injection Workflow

- Read Scenario Object
- Set RTC Alarm to trigger Scenario
    - Manipulating Parameters: Sample based on Start and Stop time, write on corresponding SRAM address
    - Adding Delay: Set loiTruck.delay value
    - Ignore: Set loiTruck.ignore
    - Set Alarm to signal end of Scenario
- Look for the next Scenario Object

Fault Injection Design

- Manipulating Parameter should be handled parallel using M4 and communicate via RPC if needed to ensure Real-Time communication

- M7 only handle CAN Bus Simulation

Fault Injection Implementation

```
// check if ignore or delay or send_predefined
if (!loiTruck->ignore){
  HAL_Delay(loiTruck->delay);

  if (HAL_FDCAN_AddMessageToTxFifoQ(&(loiTruck->my_can.CanHandle), &loiTruck->TxHeader, msg.data) != HAL_OK){
    Serial.println("Hier error addMessageToTx");
    return 0;
  }

  if (!loiTruck->fake_heart_beat){
    Serial.print("S :\t"); Serial.print(loiTruck->TxHeader.Identifier,HEX); Serial.print(" ");


    for (uint8_t i = 0; i < (msg.len); i++){
      Serial.print(" ");
      Serial.print(loiTruck->msg.data[i], HEX);
    }

    Serial.println();
  }
}
```

Implementation Approach

- Programming using HAL API for:
  - Ease of use
  - HAL driver available for all peripheral and board independent
  - Offer adequate controlling functionalities over the peripheral

- Designs:
  - M7 control CAN Bus simulation and all used Peripherals: RTC, FDCAN, TIMER...
  - M4 run only in Fault Injection Function need a parallel running method to manipulate data saved in SRAM
  - Specialization enable further development (internet connection, print TRACE...)

References

1. Description of STM32H7 HAL and low-layer drivers at
https://www.st.com/resource/en/user_manual/dm00392525-description-of-stm32h7-hal-and-lowlayer-drivers-stmicroelectronics.pdf

2. STM32H747xI datasheet at
https://www.st.com/resource/en/datasheet/stm32h747xi.pdf

3. RM0399 Reference Manual at

https://www.st.com/resource/en/reference_manual/dm00176879-stm32h745-755-and-stm32h747-757-advanced-arm-based-32-bit-mcus-stmicroelectronics.pdf