Using STM32MP1 Cortex-M with Keil MDK

MDK Tutorial

AN319, February 2021, V1.2

feedback@keil.com

Abstract

This application note describes how to create new projects and debug applications for the Arm® Cortex®-M4 core of STMicroelectronics' heterogeneous multi-core device STM32MP1 in Arm Keil MDK. Two examples using different boot modes are explained step by step. The examples are present in the STM32MP1xx device family pack. Additionally, it is explained how to generate an OpenSTLinux device tree for customized peripheral assignments.

Contents

Prerequisites

MDK v5.27 provides support for creating and debugging applications for Arm Cortex-M based cores of heterogeneous multi-core devices, including STM32MP1 devices from STMicroelectronics.

To be able to use the examples provided in this application note, you need to have a valid MDK license (not MDK-Lite). You can use the built-in seven days trial version of MDK-Professional.

Using PackInstaller, make sure that the following software packs are available:

- ARM.CMSIS.5.5.1.pack (or higher)
- ARM.AMP.1.1.0.pack (or higher)
- Keil. STM32MP1xx.1.1.0.pack (or higher)

A Bash shell is required to run shell scripts. Install for example Git for Windows and make sure the installation folder (C:\Program Files\Git\bin) is added to the Windows path. In Windows, check **Control Panel – System – Advanced System Settings – Environment Variables – System Variables – Path**

Introduction

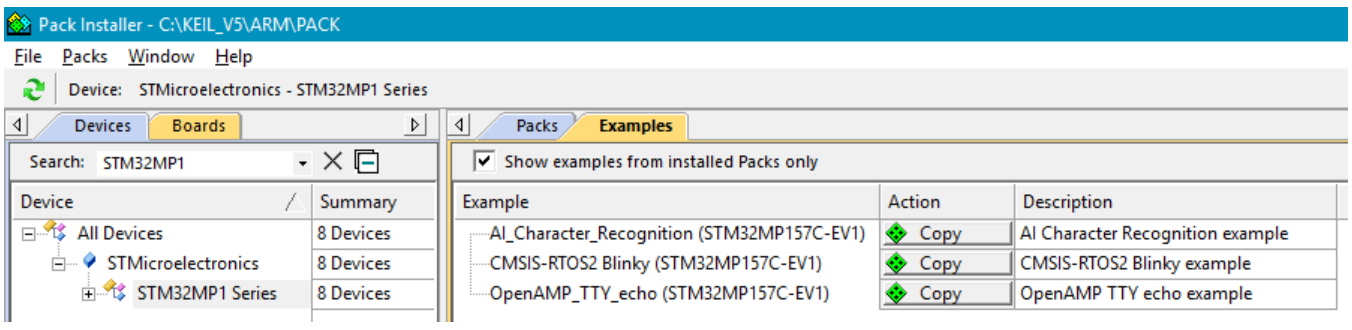
The STM32MP1 is a heterogeneous device based on a single- or dual-core Arm Cortex-A7 (CA7) and an Arm Cortex-M4 (CM4) core. The Arm Cortex-A7 core provides access to open-source operating systems (such as Linux or Android) while the Arm Cortex-M4 core leverages the STM32 MCU ecosystem and usually runs bare-metal code or a real-time operating system such as Keil RTX5.

This device offers two different boot modes: engineering and production.

- The *engineering* boot mode allows the user to connect a debugger to an opened chip, so that any program can be loaded on either the CA7 or the CM4. When the ROM code detects engineering boot mode, it reopens CA7 secure debug and starts the CM4 to run an infinite loop. In this boot mode, the CM4 application can be loaded directly with the debug adapter. This is the recommended setup to start a new CM4 project.
- In *production* boot mode, the master processor CA7 loads the application and starts the coprocessor CM4. In this case, system resources, such as clock and regulators, are managed by OpenSTLinux, which loads the executable file and launches the coprocessor through Linux remoteproc commands.

For the examples demonstrated in this tutorial, it's assumed you are familiar with the STM32MP157x-EV1 evaluation board or the STM32MP157C-DK2 discovery kit and you have already loaded and booted the OpenSTLinux image. Otherwise please follow the Getting Started section of the STM32 MPU Wiki, which is the entry point for many references in this document.

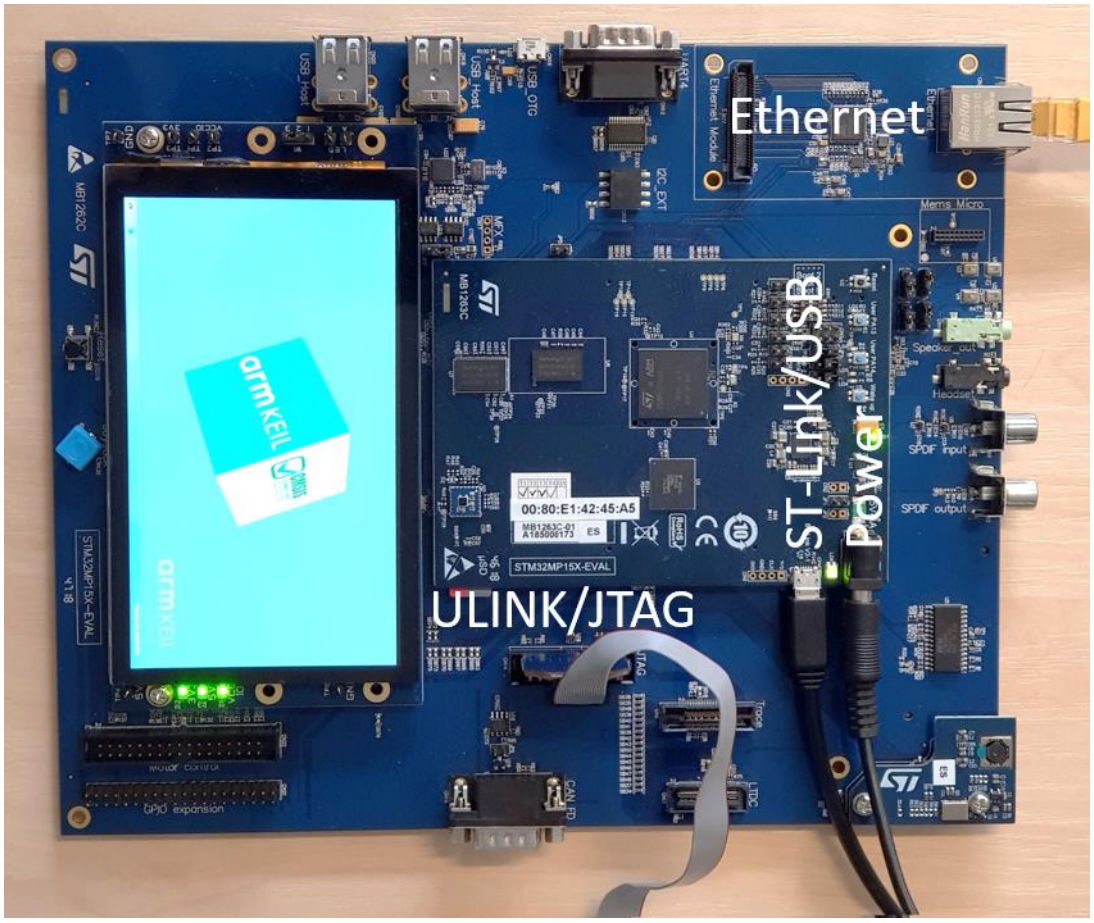
The examples are also part of the STM32MP1xx device family pack. They can be copied using the Pack Installer tool. Open it by clicking on the Pack Installer icon in the µVision toolbar: 



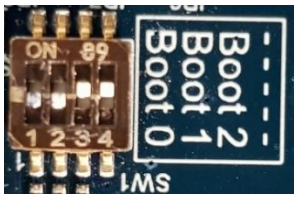
Hardware setup for the STM32MP157x-EV1 evaluation board

The following connections must be made to be able to debug the hardware successfully:

- Connect the power supply to the CN1 jack.
- Using a Micro-USB cable, connect to the ST-Link (CN4); this can be used for debug and a serial terminal
- [Optional] Connect a ULINK debug adapter using an adapter board to the JTAG connector (CN14)
- Ethernet connection (network with DHCP server or fixed IP address)



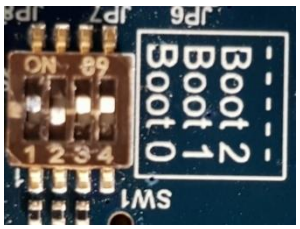
Boot in engineering mode



For engineering mode, you need to set the boot pins in the following order:

- Boot 0: Off
- Boot 1: Off
- Boot 2: On

Boot in production mode



For production mode, you need to set the boot pins in the following order:

- Boot 0: On
- Boot 1: Off
- Boot 2: On

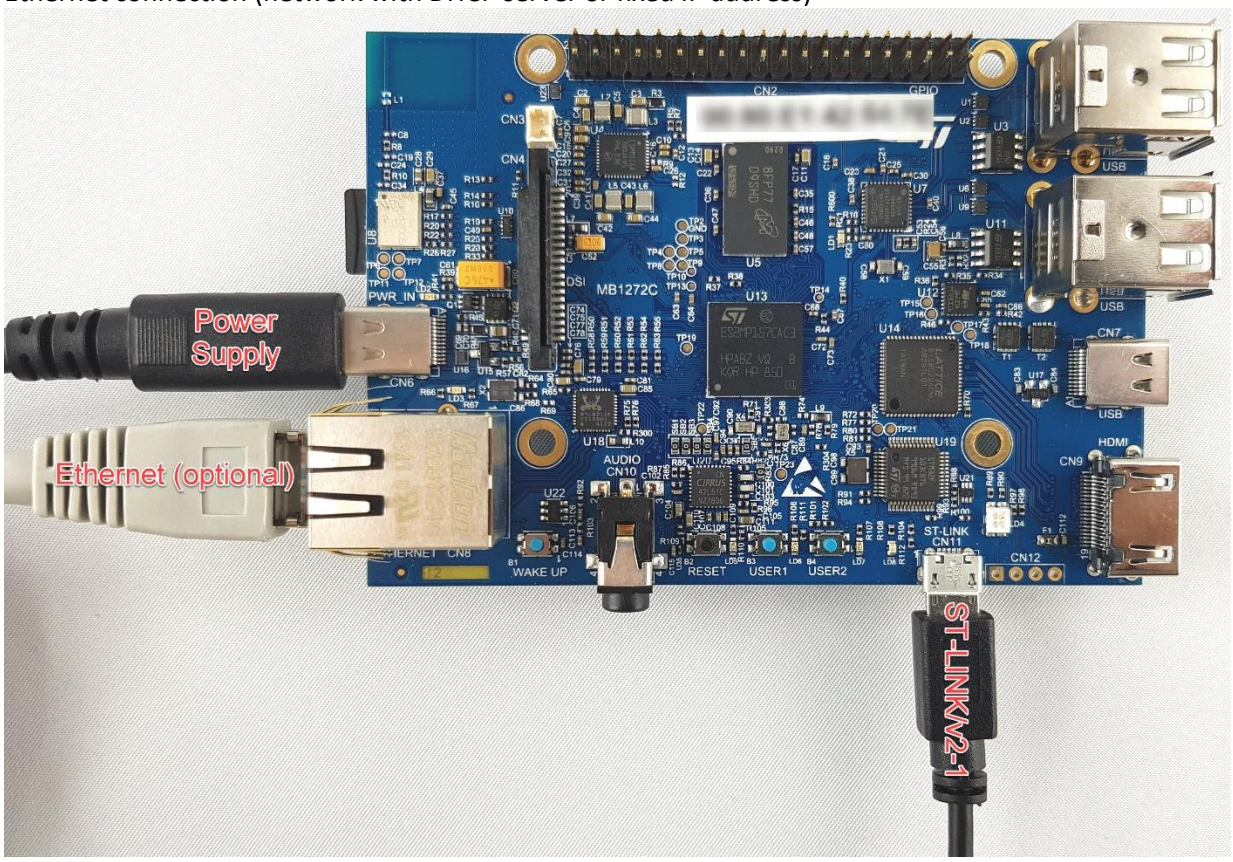
Debug connection

The STM32MP157x-EV1 evaluation board includes an on-board ST-Link debug adapter. While this is great to get started, the capabilities of the debug unit are limited. Using the 20-pin JTAG connector (with the optional adapter board HPI-0195C), you can connect any member of the ULINK family of debug adapters to get more debug visibility. The required connections and software settings are explained later.

Hardware setup for the STM32MP157C-DK2 discovery kit

The following connections must be made to be able to debug the hardware successfully:

- Connect the USB Type-C power supply to the CN6 connector.
- Using a Micro-USB cable, connect to the ST-Link (CN11); this can be used for debug and a serial terminal
- Ethernet connection (network with DHCP server or fixed IP address)



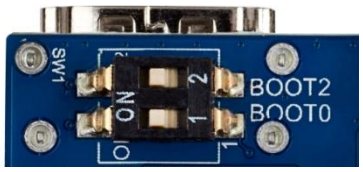
Boot in engineering mode

For engineering mode, you need to set the boot pins in the following order:

- Boot 0: Off
- Boot 2: On

Boot in production mode

For production mode, you need to set the boot pins in the following order:



- Boot 0: On
- Boot 2: On

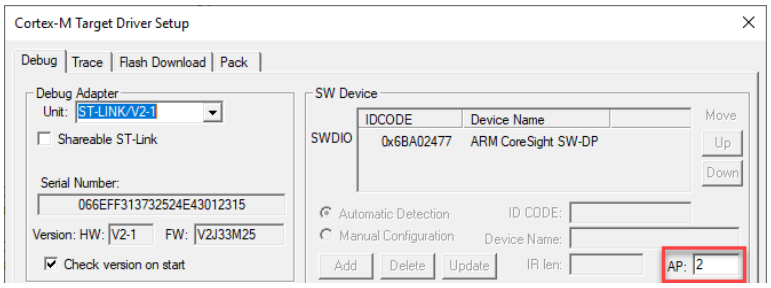
Debug connection

The STM32MP157C-DK2 discovery kit has only an on-board ST-Link debug adapter. If you need more debug capabilities, use the STM32MP157x-EV1 evaluation board instead. The required connections and software settings are explained in the following page.

ST-Link

Engineering mode

Go to **Project – Options for Target**. On the **Debug** tab, select “ST-Link” and then open the “Settings” dialog. On the next dialog’s **Debug** tab make sure that AP equals 2:

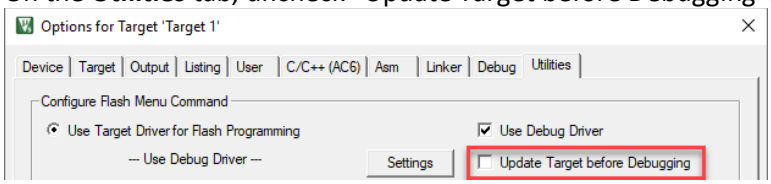


On the **Trace** tab, make sure that “Trace Enable” is unchecked:

On the **Pack** tab, make sure “Debug Description Enable” is unchecked:

Click OK.

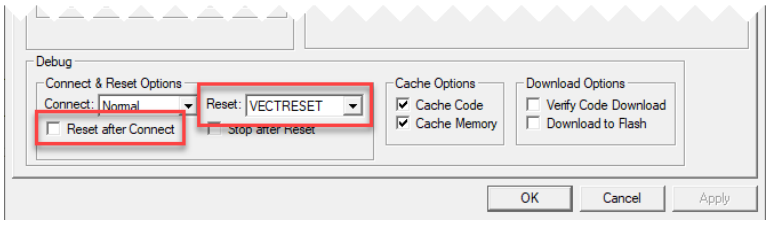
On the **Utilities** tab, uncheck “Update Target before Debugging”:



Click OK.

Production mode

In production mode, in addition to the setting used above, make sure that you are not resetting the target when connecting to it. On the **Debug** tab, make sure that “Reset” equals VECTRESET and that “Reset after Connect” is not used:



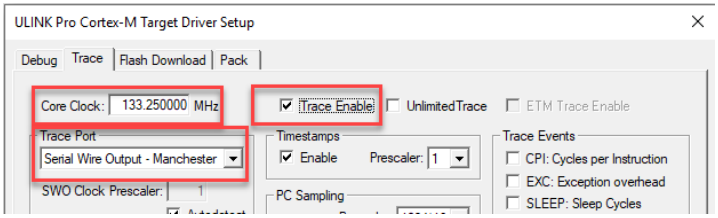
Note:

In production mode, the debug adapter cannot connect directly to the device. OpenSTLinux will establish the connection once the application for the Cortex-M4 is loaded. Thus, you need to have a binary image of the application ready and must follow the steps 15 – 17 on page 23.

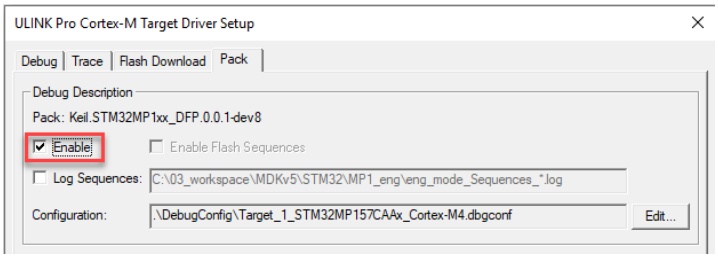
ULINK

Engineering mode

Go to **Project – Options for Target**. On the **Debug** tab, for example, select “ULINK Pro Cortex Debugger” and then open the “Settings” dialog.
On the next dialog’s **Trace** tab set the Core Clock to 133.25 and enable “Trace Enable”. Use “Serial Wire Output – Manchester” as the trace port:

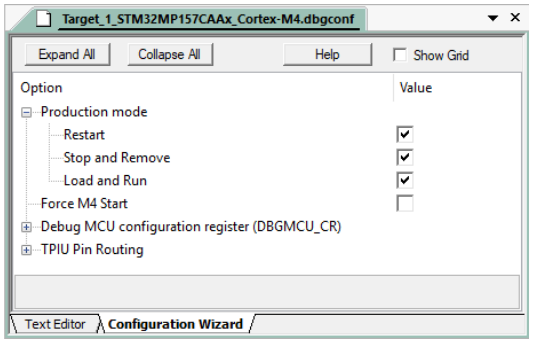


On the Pack tab, check that Debug Description is enabled:



Production mode

In production mode, use the same settings as above. In addition, click on the “Edit” button on the **Pack** tab (see screenshot above). The dbgconf configuration file opens in a new window. In Configuration Wizard mode, enable “Restart”, “Stop and Remove”, and “Load and Run”:
These options will enable the ULINK debug adapters to automatically use the right debug sequences to communicate with the target (not available for the ST-Link debug adapter).

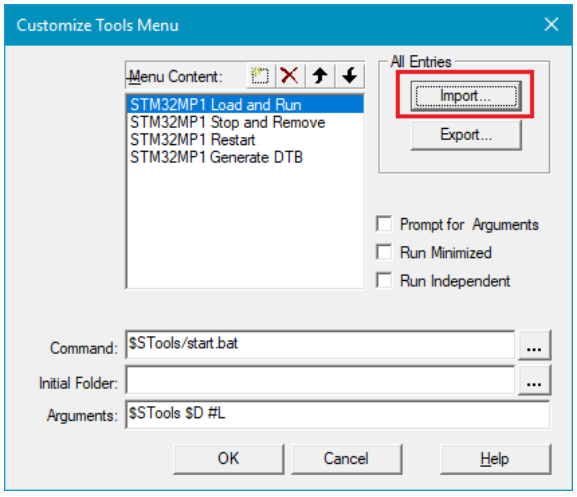


Customizing the µVision Tools Menu

Alternatively, to the method above (and the only automated way when using the ST-Link), you can add shell scripts to the µVision Tools Menu. These shell scripts load, run, stop and remove the application in production mode. An additional one generates the device tree files.

The scripts are in the STM32MP1’s pack subfolder 'Tools' and can be added to MDK menu:
Tools – Customize Tools Menu – Import...
{PACK_FOLDER}\Tools\tools.cfg

(for example C:\KEIL_V5\ARM\PACK\Keil\STM32MP1xx_DFP\0.0.1\Tools\tools.cfg)



Creating a new Blinky project for engineering mode

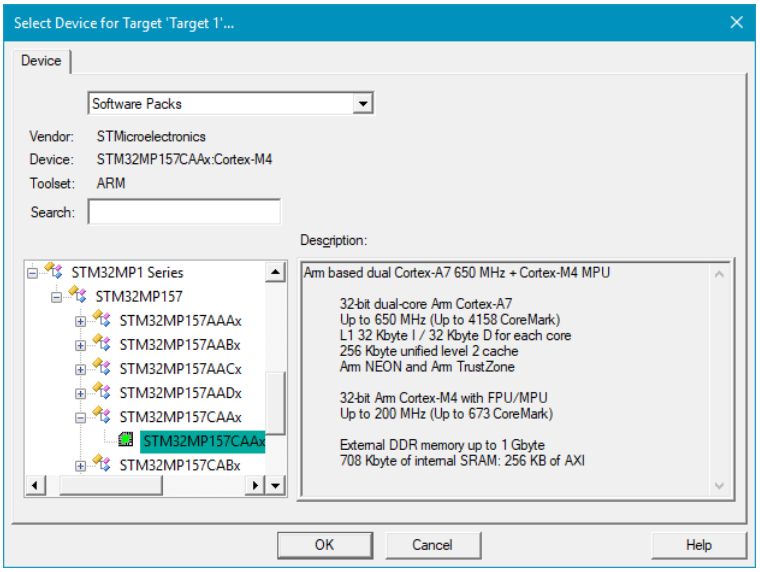
The ROM code detects *engineering* boot and reopens CA7 secure debug and starts CM4 to run an infinite loop, allowing the user to connect a debugger to an opened chip, so that any program can be loaded directly with the debug adapter.

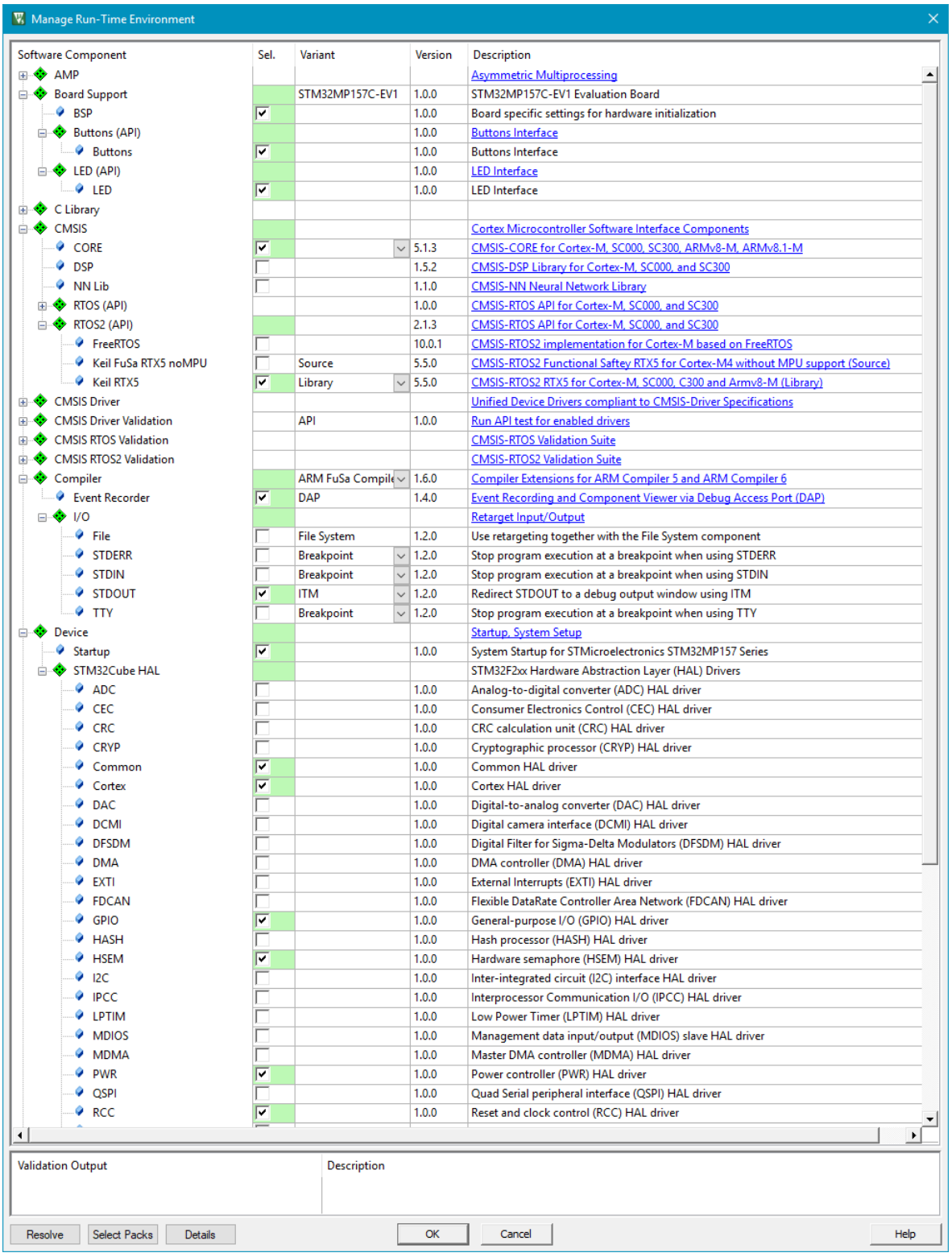
The boot switches are described in the Hardware setup section.

In this example, Keil RTX5 threads are used to control LEDs and a user button. LD3 and LD4 blink as running lights, while blinking is paused if the UserPA13/B2 button is pressed.

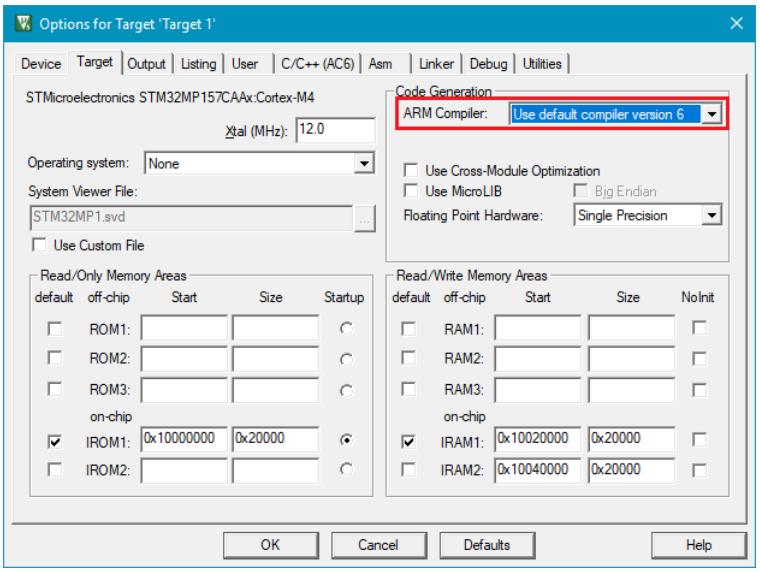
Project creation step by step

1. In µVision, select Project – New µVision Project… and give a name to the new project.
2. From the device list, select STMicroelectronics STM32MP157CAAx:Cortex-M4 device.
3. In the Manage Run-Time Environment window, select the following components:
   - Board Support:BSP
   - Board Support:Buttons
   - Board Support:LED
   - CMSIS:CORE
   - CMSIS:RTOS2 (API):Keil RTX5
   - (Variant Source)
   - Compiler:Event Recorder
   - Compiler:IO:STDOUT
   - (Variant ITM)
   - Device:Startup
   - Device:STM32Cube HAL:Common
   - Device:STM32Cube HAL:Cortex
   - Device:STM32Cube HAL:GPIO
   - Device:STM32Cube HAL:HSEM
   - Device:STM32Cube HAL:PWR
   - Device:STM32Cube HAL:RCC





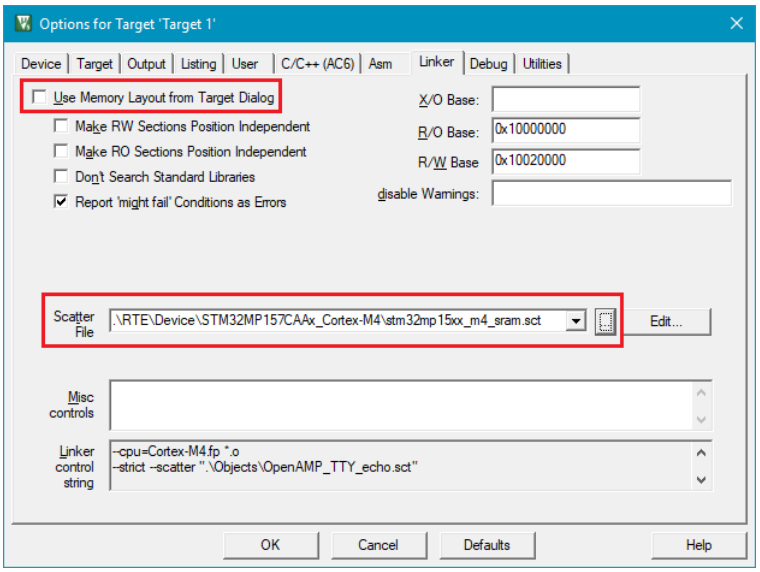
4. Go to Project – Options for Target (Alt+F7). On the **Target** tab, set “Use default compiler version 6” as the ARM compiler.



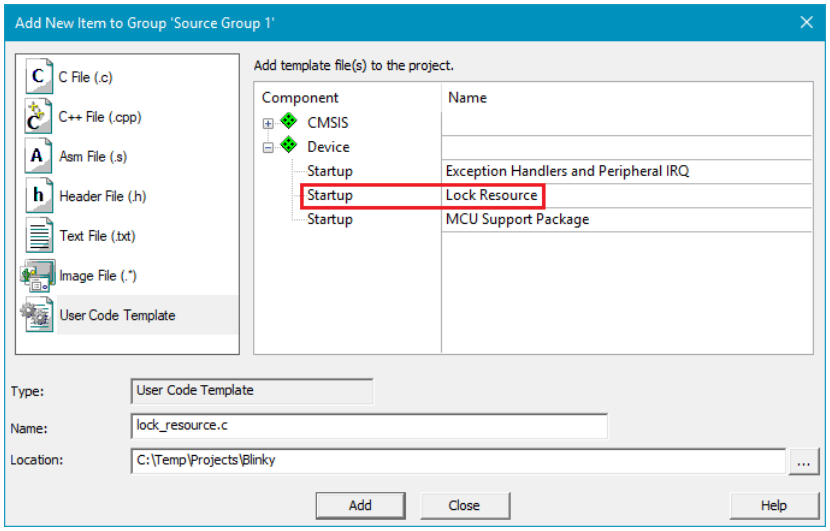
On the **Linker** tab, uncheck “Use Memory Layout from Target Dialog”, click on the ellipsis button “…” and select as Scatter File:

```
.\RTE\Device\STM32MP157CAAx_Cortex-M4\stm32mp15xx_m4_sram.sct
```

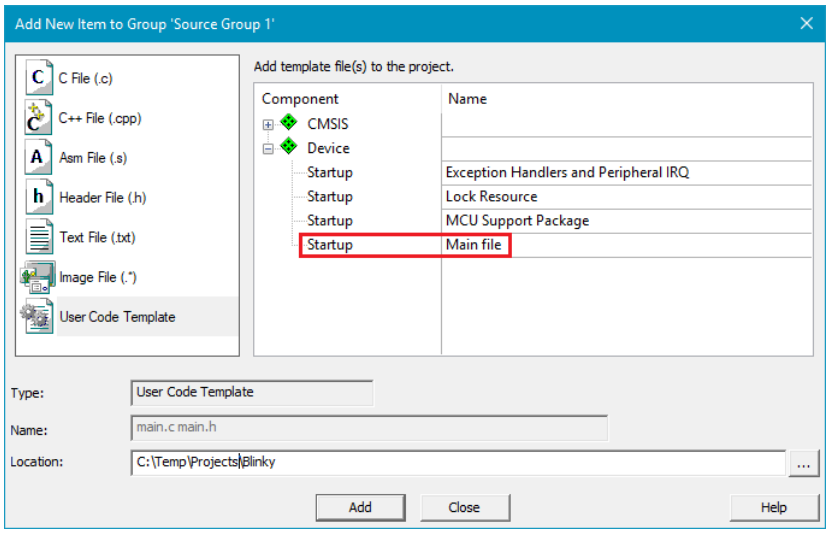
On the **Debug** tab, select your debug adapter. Refer to the section Debug connection for more information.



5. Add user code templates: Right click on “Source Group 1” and then “Add new Item to Source Group 1”.
Add “Lock Resource” from Device:Startup component.



6. Add main file: Right click on “Source Group 1” and then “Add new Item to Source Group 1”. Add “Main file” from Device:Startup component.



In the main.c file add the following lines into the respective sections:

Includes

```
#ifdef _RTE_
#include "RTE_Components.h"             // Component selection
#endif
#ifdef RTE_CMSIS_RTOS2                  // when RTE component CMSIS RTOS2 is
used
#include "cmsis_os2.h"                  // :CMSIS:RTOS2
#endif
```

Function prototypes

```
extern void app_main (void *arg);
```

Kernel initialization, before while loop in the main function

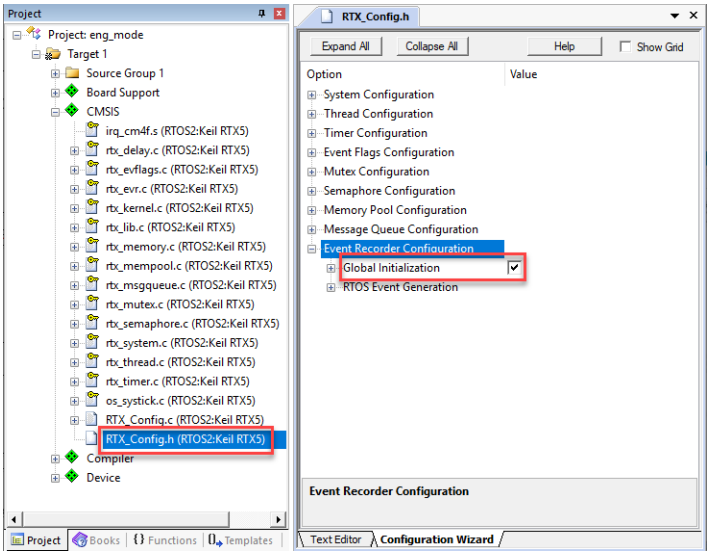
```
#ifdef RTE_CMSIS_RTOS2
  /* Initialize CMSIS-RTOS2 */
  osKernelInitialize ();

  /* Create application main thread */
  osThreadNew(app_main, NULL, NULL);

  /* Start thread execution */
  osKernelStart();
#endif
```

7. Add a new Blinky.c file with the following content:

```
/*----------------------------------------------------------------------------
 * Name:    Blinky.c
 * Purpose: LED Flasher
 *----------------------------------------------------------------------------
 * This file is part of the uVision/ARM development tools.
 * This software may only be used under the terms of a valid, current,
 * end user licence from KEIL for a compatible version of KEIL software
 * development tools. Nothing else gives you the right to use this software.
 *
 * This software is supplied "AS IS" without warranties of any kind.
 *
 * Copyright (c) 2019 Keil - An ARM Company. All rights reserved.
 *----------------------------------------------------------------------------*/

#include <stdio.h>
#include "main.h"
#include "Board_LED.h"                  /* :Board Support:LED     */
#include "Board_Buttons.h"              /* :Board Support:Buttons */
#include "cmsis_os2.h"                  /* :CMSIS:RTOS2           */
#include "RTE_Components.h"             /* Component selection     */
#include CMSIS_device_header

#ifdef RTE_Compiler_EventRecorder
#include "EventRecorder.h"
#endif

static osThreadId_t tid_thrLED;         /* Thread id of thread: LED */
static osThreadId_t tid_thrBUT;         /* Thread id of thread: BUT */

/*----------------------------------------------------------------------------
  thrLED: blink LED
 *----------------------------------------------------------------------------*/
__NO_RETURN static void thrLED(void *argument) {
  uint32_t led_max    = LED_GetCount();
  uint32_t led_num    = 0U;

  (void)argument;

  printf("thrLED is running\n");

  for (;;) {
    osThreadFlagsWait(0x0001U, osFlagsWaitAny ,osWaitForever);
    LED_On(led_num);                            /* Turn specified LED on */
    printf("Blink LED number %d\n", led_num);
    osThreadFlagsWait(0x0001U, osFlagsWaitAny ,osWaitForever);
    LED_Off(led_num);                           /* Turn specified LED off */

    led_num++;                                  /* Change LED number */
    if (led_num >= led_max) {
      led_num = 0U;                             /* Restart with first LED */
    }
  }
}
```

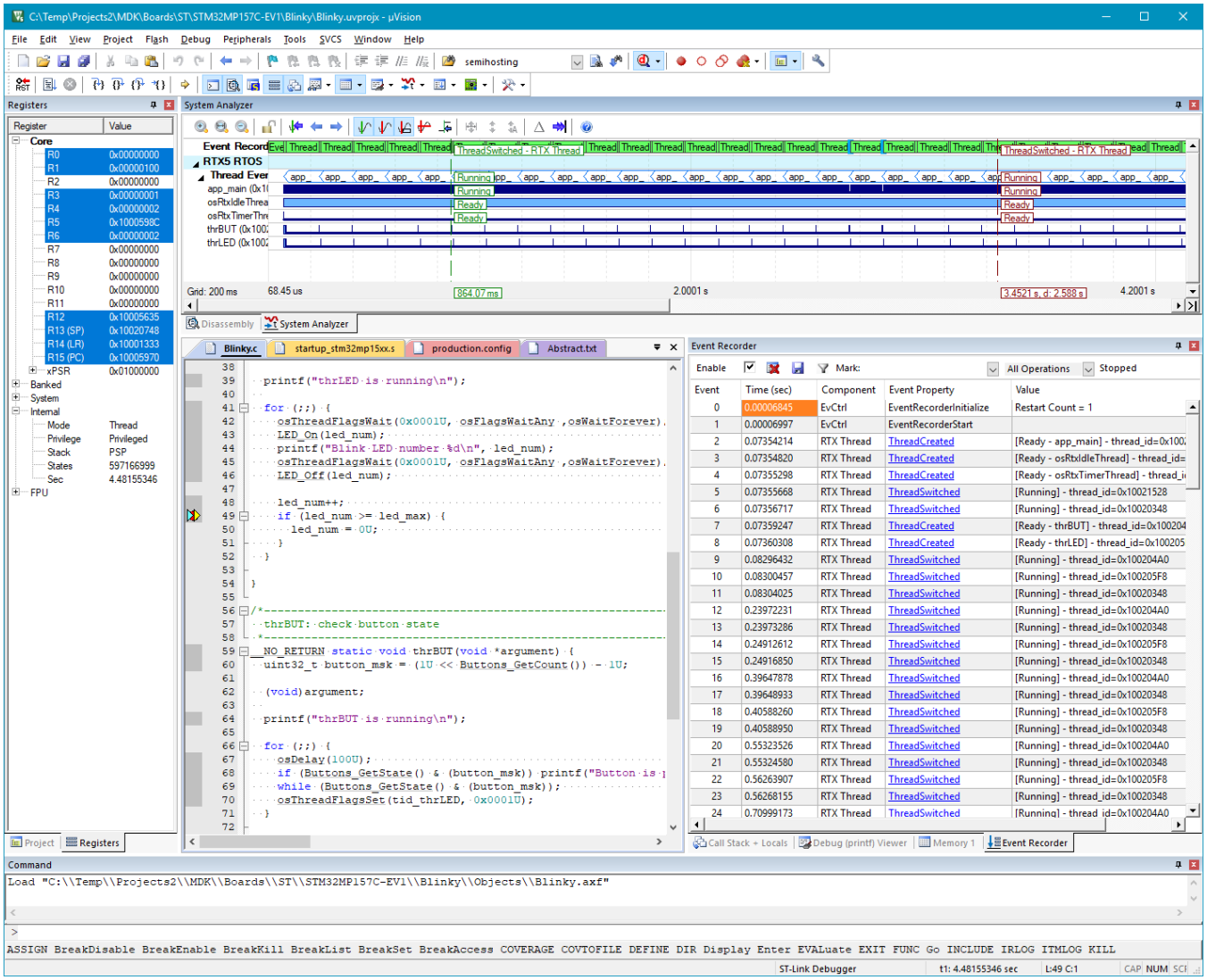
```
/*----------------------------------------------------------------------------
  thrBUT: check button state
 *----------------------------------------------------------------------------*/
__NO_RETURN static void thrBUT(void *argument) {
  uint32_t button_msk = (1U << Buttons_GetCount()) - 1U;

  (void)argument;

  printf("thrBUT is running\n");

  for (;;) {
    osDelay(100U);                                  /* Wait */
    if (Buttons_GetState() & (button_msk)) printf("Button is pressed\n");
    while (Buttons_GetState() & (button_msk)); /* Wait while holding user
button */
    osThreadFlagsSet(tid_thrLED, 0x0001U);
  }
}

/*----------------------------------------------------------------------------
 * Application main thread
 *----------------------------------------------------------------------------*/
__NO_RETURN void app_main (void *argument) {

  (void)argument;

  LED_Initialize();                                 /* initialize LEDs */
  Buttons_Initialize();                             /* initialize Buttons */

  tid_thrBUT = osThreadNew (thrBUT, NULL, NULL);    /* create BUT thread */
  if (tid_thrBUT == NULL) { /* add error handling */ }

  tid_thrLED = osThreadNew (thrLED, NULL, NULL);    /* create LED thread */
  if (tid_thrLED == NULL) { /* add error handling */ }

  for (;;) {}
}
```

8. To see threads switching in the System Analyzer window, enable Event Recorder for RTX5. In the **Project** window, under CMSIS, double-click the "RTX_Config.h" file.
   In Configuration Wizard mode, open "Event Recorder Configuration" and enable "Global Initlialization":



9. Build the target (F7). The program is compiled without errors or warnings.

10. Start debugging: if you have not yet selected your debug adapter in your µVision project, refer to the section Debug connection how to do so. When done, go to **Debug – Start/Stop Debug Session** or press Ctrl+F5.
    The application is loaded by the debug adapter into the RAM memory space specified in the linker scatter file. Run and debug as usual, using µVision features like System Analyzer and Event Recorder as illustrated here:



Creating an OpenAMP project for production mode

In *production mode*, the OpenSTLinux runs on the CA7 as master and is responsible to load the application and to start the CM4 as coprocessor, using the remoteproc framework. MDK integrates such remote coprocessor commands in the debug engine, so the developer can load, run and debug the CM4 application as usual.

This example demonstrates the inter-processor communication between the CM4 and the CA7 cores. It uses a STMicroelectronics variant of the OpenAMP distribution.

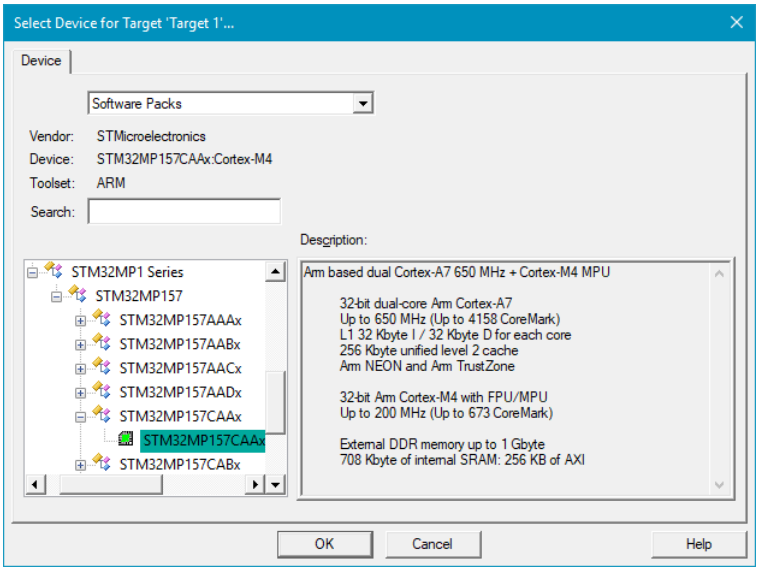
What the example does:

- CM4 initializes the OpenAMP component, configures IPCC peripheral through HAL and sets up the openamp-rpmsg framework infrastructure, which is the first level of communication between CA7 and CM4.
- CM4 creates two rpmsg channels for two virtual UART instances.
- CM4 waits for messages from the master core CA7 on these two channels.
- When the CM4 receives a message on one virtual UART instance, it sends the message back to CA7 on the same virtual UART instance.

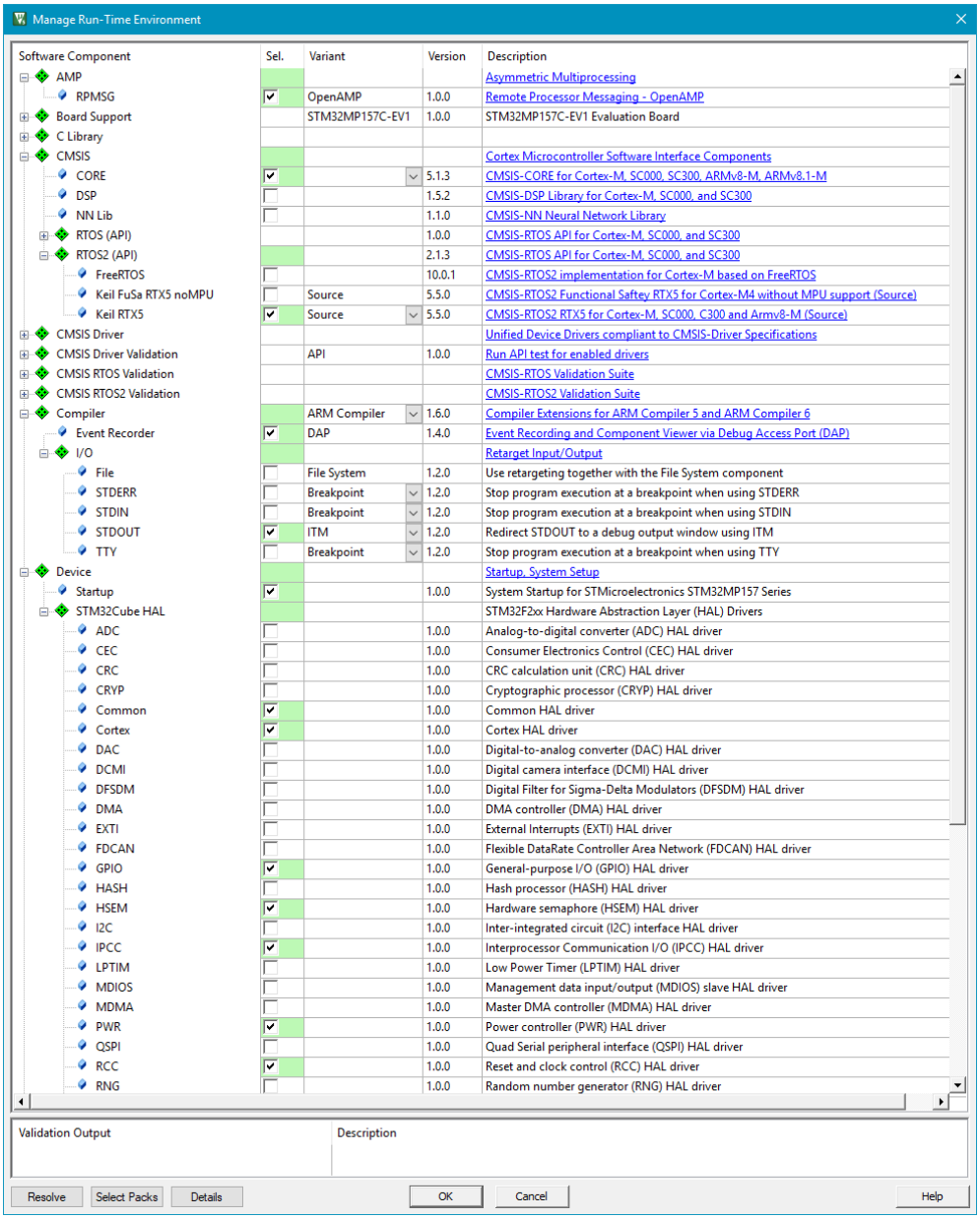
For more info about STM32MPU coprocessor handling, resource management, RPMsg and remoteproc framework, please refer to Coprocessor management Linux and Coprocessor management STM32Cube.

Project creation step by step

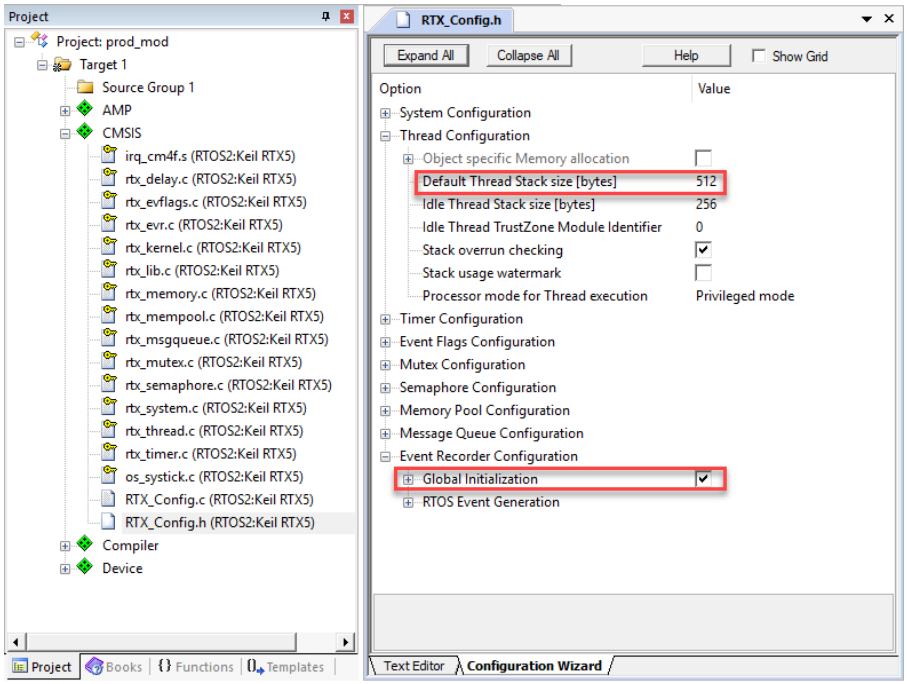
1. In µVision, select Project – New µVision Project… and give a name to the new project.
2. From the device list, select STMicroelectronics STM32MP157CAAx:Cortex-M4 device.



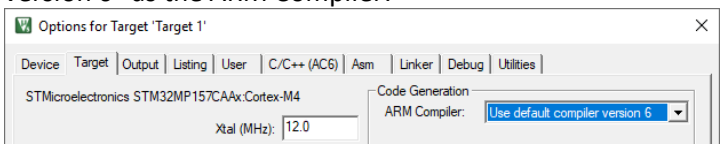
3. In the Manage Run-Time Environment window, select the following components:
   - AMP:RPMSG
   - CMSIS:CORE
   - CMSIS:RTOS2 (API):Keil RTX5 (Variant Source)
   - Compiler:Event Recorder
   - Compiler:IO:STDOUT (Variant ITM)
   - Device:Startup
   - Device:STM32Cube HAL:Common
   - Device:STM32Cube HAL:Cortex
   - Device:STM32Cube HAL:GPIO
   - Device:STM32Cube HAL:HSEM
   - Device:STM32Cube HAL:IPCC
   - Device:STM32Cube HAL:PWR
   - Device:STM32Cube HAL:RCC



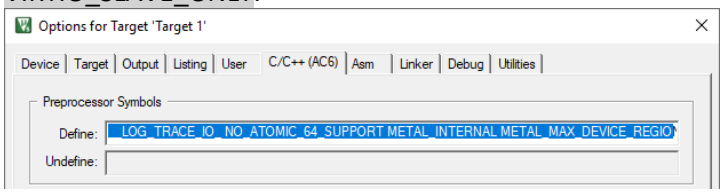
4. Configure RTX5: open the RTXConfig.h file under CMSIS component, expand the “Thread Configuration” option and change the “Default Thread Stack size” to 512. Expand Event Recorder Configuration and enable “Global Initialization”:



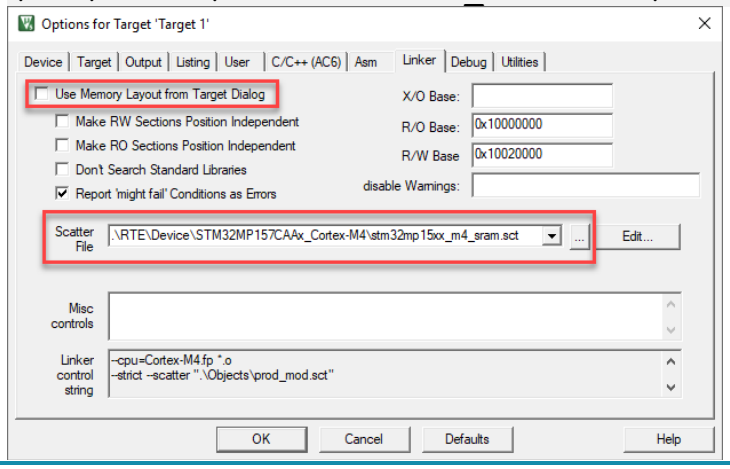
5. Open the menu Project – Options for Target (Alt+F7). On the **Target** tab, set “Use default compiler version 6” as the ARM Compiler:



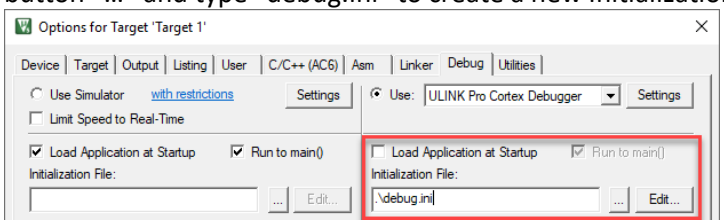
6. On the **C/C++ (AC6)** tab, add the following preprocessor defines:
__LOG_TRACE_IO_ NO_ATOMIC_64_SUPPORT METAL_INTERNAL METAL_MAX_DEVICE_REGIONS=2 VIRTIO_SLAVE_ONLY:



7. On the **Linker** tab, uncheck “Use Memory Layout from Target Dialog”, click on the ellipsis button “…” and select as Scatter File:
`\RTE\Device\STM32MP157CAAx_Cortex-M4\stm32mp15xx_m4_sram.sct`



8. On the **Debug** tab: select your debug adapter, uncheck “Load Application at Startup”, click on the ellipsis button “…” and type “debug.ini” to create a new Initialization File:



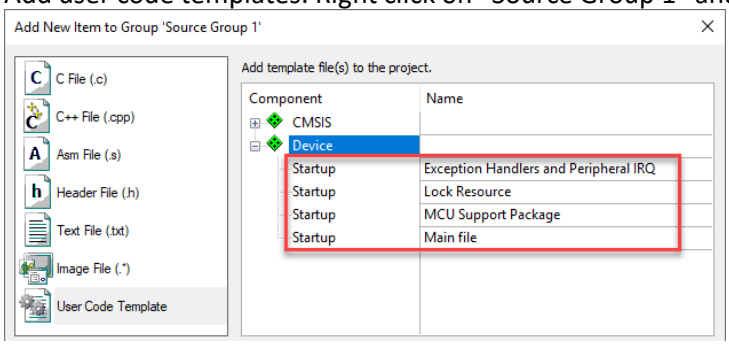
Click on “Edit…” to open the debug.ini file and add the following line:

```
LOAD %L NOCODE INCREMENTAL
```

9. Edit the linker scatter file to add Resource Table, OpenAMP, and Event Recorder buffer sections:

```
LR_VECTORS 0x00000000 0x00000400  {          ; Vector table
  .isr_vector +0 {
    startup*.o (RESET, +First)
  }
}

LR_IROM1 0x10000000 0x00060000  {            ; SRAM
  ER_IROM1 0x10000000 0x00020000  {          ; Code
    *(InRoot$$Sections)
    .ANY (+RO)
    .ANY (+XO)
  }
  RW_IRAM1 0x10020000 0x00020000  {          ; Data
    .ANY (+RW +ZI)
  }
  .resource_table +0 ALIGN 4 {               ; Resource Table
    rsc_table.o (+RW +ZI)
  }
  RW_OPENAMP 0x10040000 EMPTY 0x00010000 {   ; OpenAMP Buffer

  }
  RW_ER 0x10050000 UNINIT 0x00010000  {      ; Event Recorder
    EventRecorder.o (+ZI)
  }
}
```

10. Add user code templates: Right click on “Source Group 1” and then “Add new Item to Source Group 1”:



Add the all available templates from the Device:Startup component.

11. Add the IPCC functions to the stm32mp1xx_hal_msp.c file as reported below.

```
/**
* @brief IPCC MSP Initialization
* This function configures the hardware resources used in this example
* @param hipcc: IPCC handle pointer
* @retval None
*/
void HAL_IPCC_MspInit(IPCC_HandleTypeDef* hipcc)
{

  if(hipcc->Instance==IPCC)
  {
  /* USER CODE BEGIN IPCC_MspInit 0 */

  /* USER CODE END IPCC_MspInit 0 */
    /* Peripheral clock enable */
    __HAL_RCC_IPCC_CLK_ENABLE();
  /* IPCC interrupt Init */
    HAL_NVIC_SetPriority(IPCC_RX1_IRQn, 0, 0);
    HAL_NVIC_EnableIRQ(IPCC_RX1_IRQn);
  /* USER CODE BEGIN IPCC_MspInit 1 */

  /* USER CODE END IPCC_MspInit 1 */
}

}

/**
* @brief IPCC MSP De-Initialization
* This function freeze the hardware resources used in this example
* @param hipcc: IPCC handle pointer
* @retval None
*/

void HAL_IPCC_MspDeInit(IPCC_HandleTypeDef* hipcc)
{

  if(hipcc->Instance==IPCC)
  {
  /* USER CODE BEGIN IPCC_MspDeInit 0 */

  /* USER CODE END IPCC_MspDeInit 0 */
    /* Peripheral clock disable */
    __HAL_RCC_IPCC_CLK_DISABLE();

    /* IPCC interrupt DeInit */
    HAL_NVIC_DisableIRQ(IPCC_RX1_IRQn);
  /* USER CODE BEGIN IPCC_MspDeInit 1 */

  /* USER CODE END IPCC_MspDeInit 1 */
  }
}
```

12. The interrupt handlers SVC_Handler, PendSV_Handler and SysTick_Handler are part of the RTX5 implementation and shall be removed from the stm32mp1xx_it.c file. Add the IPCC handlers as reported below.

```
extern IPCC_HandleTypeDef hipcc;

void IPCC_RX1_IRQHandler(void)
{
  HAL_IPCC_RX_IRQHandler(&hipcc);
}

void IPCC_TX1_IRQHandler(void)
{
  HAL_IPCC_TX_IRQHandler(&hipcc);
}
```

13. In the main.c file add the following lines to the respective sections:

Includes

```
#ifdef _RTE_
#include "RTE_Components.h"             // Component selection
#endif
#ifdef RTE_CMSIS_RTOS2                  // when RTE component CMSIS RTOS2 is
used
#include "cmsis_os2.h"                  // :CMSIS:RTOS2
#endif
#include "virt_uart.h"                  // ARM:AMP:RPMSG
```

Defines

```
#define MAX_BUFFER_SIZE RPMSG_BUFFER_SIZE
```

Variables

```
IPCC_HandleTypeDef hipcc;

VIRT_UART_HandleTypeDef huart0;
VIRT_UART_HandleTypeDef huart1;

__IO FlagStatus VirtUart0RxMsg = RESET;
uint8_t VirtUart0ChannelBuffRx[MAX_BUFFER_SIZE];
uint16_t VirtUart0ChannelRxSize = 0;

__IO FlagStatus VirtUart1RxMsg = RESET;
uint8_t VirtUart1ChannelBuffRx[MAX_BUFFER_SIZE];
uint16_t VirtUart1ChannelRxSize = 0;
```

Function prototypes

```
extern void app_main (void *arg);
static void MX_IPCC_Init(void);
void VIRT_UART0_RxCpltCallback(VIRT_UART_HandleTypeDef *huart);
void VIRT_UART1_RxCpltCallback(VIRT_UART_HandleTypeDef *huart);
```

Kernel initialization, before while loop in the main function

```
#ifdef RTE_CMSIS_RTOS2
  /* Initialize CMSIS-RTOS2 */
  osKernelInitialize ();

  /* Create application main thread */
  osThreadNew(app_main, NULL, NULL);

  /* Start thread execution */
  osKernelStart();
#endif
```

Main thread

```
/*---------------------------------------------------------------------------
 * Application main thread
 *---------------------------------------------------------------------------*/
void app_main (void *argument) {

  /* IPCC initialisation */
  MX_IPCC_Init();

  /* OpenAmp initialisation */
  MX_OPENAMP_Init(RPMSG_REMOTE, NULL);

  /*
   * Create Virtual UART device
   * defined by a rpmsg channel attached to the remote device
   */
  log_info("Virtual UART0 OpenAMP-rpmsg channel creation\r\n");
  if (VIRT_UART_Init(&huart0) != VIRT_UART_OK) {
    log_err("VIRT_UART_Init UART0 failed.\r\n");
    Error_Handler();
  }

  log_info("Virtual UART1 OpenAMP-rpmsg channel creation\r\n");
  if (VIRT_UART_Init(&huart1) != VIRT_UART_OK) {
    log_err("VIRT_UART_Init UART1 failed.\r\n");
    Error_Handler();
  }

  /*Need to register callback for message reception by channels*/
  if(VIRT_UART_RegisterCallback(&huart0, VIRT_UART_RXCPLT_CB_ID,
VIRT_UART0_RxCpltCallback) != VIRT_UART_OK)
  {
   Error_Handler();
  }
  if(VIRT_UART_RegisterCallback(&huart1, VIRT_UART_RXCPLT_CB_ID,
VIRT_UART1_RxCpltCallback) != VIRT_UART_OK)
  {
    Error_Handler();
  }

  /* Infinite loop */
  while (1)
```

```
  {

    OPENAMP_check_for_message();

    if (VirtUart0RxMsg) {
      VirtUart0RxMsg = RESET;
      VIRT_UART_Transmit(&huart0, VirtUart0ChannelBuffRx,
VirtUart0ChannelRxSize);
    }

    if (VirtUart1RxMsg) {
      VirtUart1RxMsg = RESET;
      VIRT_UART_Transmit(&huart1, VirtUart1ChannelBuffRx,
VirtUart1ChannelRxSize);
    }

    osDelay(100);
  }
}
```

IPCC Initialization and Virtual UART callbacks

```
/**
  * @brief IPPC Initialization Function
  * @param None
  * @retval None
  */
static void MX_IPCC_Init(void)
{

  hipcc.Instance = IPCC;
  if (HAL_IPCC_Init(&hipcc) != HAL_OK)
  {
     Error_Handler();
  }
}

void VIRT_UART0_RxCpltCallback(VIRT_UART_HandleTypeDef *huart)
{

    log_info("Msg received on VIRTUAL UART0 channel:  %s \n\r", (char *)
huart->pRxBuffPtr);

    /* copy received msg in a variable to sent it back to master processor
in main infinite loop*/
    VirtUart0ChannelRxSize = huart->RxXferSize < MAX_BUFFER_SIZE? huart-
>RxXferSize : MAX_BUFFER_SIZE-1;
    memcpy(VirtUart0ChannelBuffRx, huart->pRxBuffPtr,
VirtUart0ChannelRxSize);
    VirtUart0RxMsg = SET;
}

void VIRT_UART1_RxCpltCallback(VIRT_UART_HandleTypeDef *huart)
{

    log_info("Msg received on VIRTUAL UART1 channel:  %s \n\r", (char *)
huart->pRxBuffPtr);

    /* copy received msg in a variable to sent it back to master processor
in main infinite loop*/
    VirtUart1ChannelRxSize = huart->RxXferSize < MAX_BUFFER_SIZE? huart-
>RxXferSize : MAX_BUFFER_SIZE-1;
    memcpy(VirtUart1ChannelBuffRx, huart->pRxBuffPtr,
VirtUart1ChannelRxSize);
    VirtUart1RxMsg = SET;
}
```

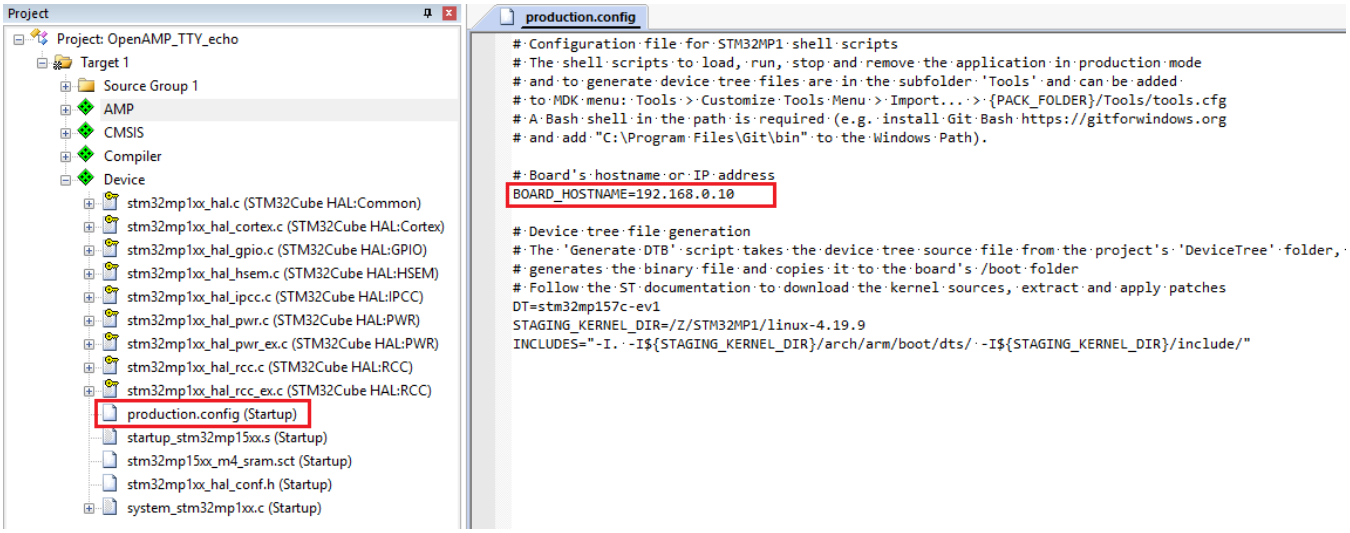
14. Build the target (F7). The program compiles without errors or warnings.

15. Before connecting to the target, open a terminal viewer (such as PuTTY or TeraTerm) and connect to the COM port that is provided by the ST-LINK (check Windows Device Manager), using 115200 Baud. Once Linux is up and running, use the command ifconfig to get the IP address of the device:

```
root@stm32mp1:~# ifconfig
eth0      Link encap:Ethernet  HWaddr 00:80:FF:FF:FF:A5
inet addr:192.168.0.10  Bcast:192.168.0.255  Mask:255.255.255.0
inet6 addr: fe80:ff0:e1ff:feef:45a5/64 Scope:Link
UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
RX packets:35 errors:0 dropped:0 overruns:0 frame:0
TX packets:52 errors:0 dropped:0 overruns:0 carrier:0
collisions:0 txqueuelen:1000
RX bytes:7081 (6.9 KiB)  TX bytes:8135 (7.9 KiB)
Interrupt:63 Base address:0xc000
```

You'll need this address in the next step, to communicate with the target.

16. Configure the production.config file: set BOARD_HOSTNAME with the board's hostname or IP address.

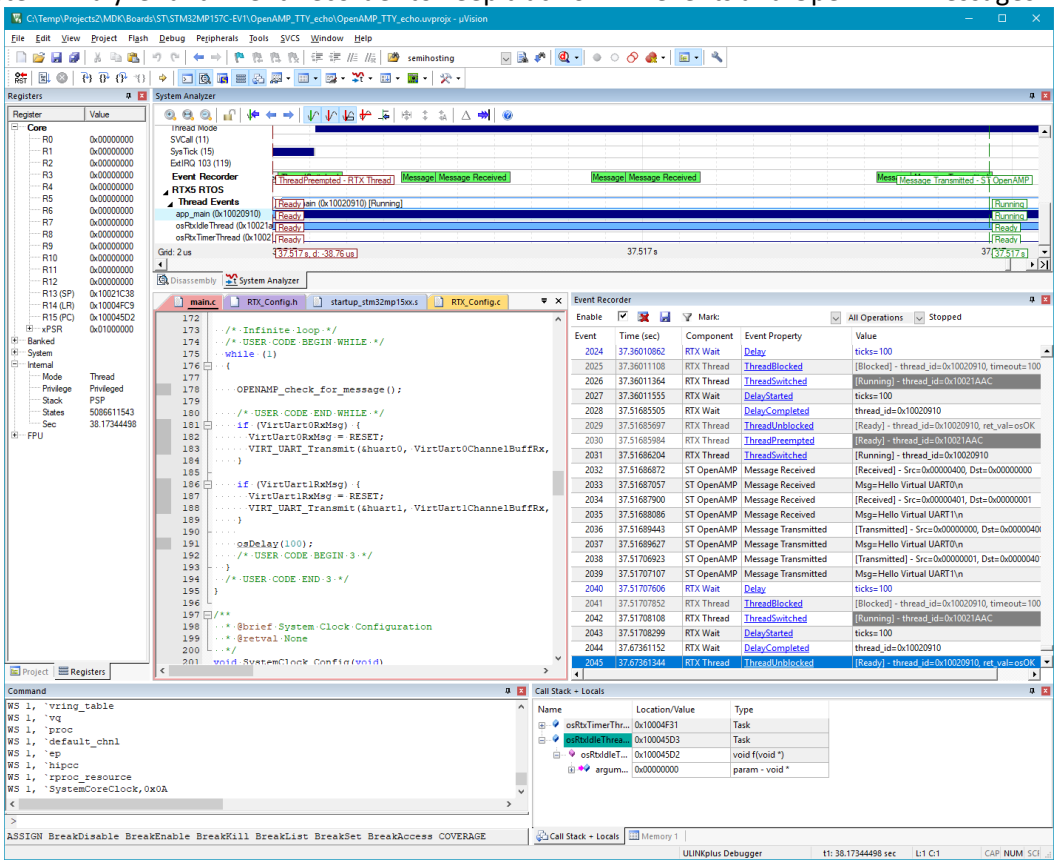


17. Start debugging: if you have not yet selected your debug adapter in your µVision project, refer to the section Debug connection how to do so.
    If you are using ST-Link, go to **Tools – STM32MP1 Load and Run**.
    For ULINKpro, ULINKplus, ULINK2 and CMSIS-DAP debug adapters, this command is run using debug sequences, as configured earlier. Go to **Debug – Start/Stop Debug Session** or press Ctrl+F5.
    The Linux console should show the following message:
    `remoteproc remoteproc0: remote processor m4 is now up`
    At this point, the image has been loaded to the embedded Linux file system. The remoteproc commands load the application into the specified memory location (as stated in the scatter file) and start the CM4. The debug symbols have been loaded as well and the debug adapter is connected to the target, so that you can run and debug as usual.

Run the following script in Linux to send messages every second:

```
if [[ $(ls -A /dev/ttyRPMSG*) ]]; then
  stty -onlcr -echo -F /dev/ttyRPMSG0
  cat /dev/ttyRPMSG0 &
  stty -onlcr -echo -F /dev/ttyRPMSG1
  cat /dev/ttyRPMSG1 &
fi

while [[ $(ls -A /dev/ttyRPMSG*) ]]; do
  echo "Hello Virtual UART0" >/dev/ttyRPMSG0
  echo "Hello Virtual UART1" >/dev/ttyRPMSG1
  sleep 1
done
```

Use System Analyzer and Event Recorder to keep track of RTX events and OpenAMP messages:



Useful links

- STM32 Arm® Cortex®-based MPUs user guide
- Reference Manual STM32MP157
- Programming Manual STM32
- Datasheet STM32MP157C
- Errata STM32MP151x/3x/7x
- OpenAMP workgroup