Interworking of MA35D1 RTP and Linux

Application Note for 64-bit MPU Family

Document Information

Abstract	This document introduces the interworking between programs running at MA35D1 RTP M4 and Linux kernel. It also introduces the development environment of RTP M4.
Apply to	NuMicro® MA35D1 series.

For additional information or questions, please contact: Nuvoton Technology Corporation.

www.nuvoton.com

Table of Contents

1. Overview

The NuMicro MA35D1 series microprocessor (MPU) platform is based on dual 64/32-bit Arm® Cortex-A35 cores at Armv8-A architecture and an Arm Cortex-M4 core at ARMv7-M architecture. The MA35D1 series microprocessor system allows users to run independent firmware on each CPU core according to different needs.

The software currently provided by MA35D1 runs Linux on the Cortex-A35, and can run RTOS or bare-metal application on the RTP M4 which is a Cortex-M4 core. In this document, RTP M4 refers to the Cortex-M4 core in MA35D1.

The RTP M4 is not equipped with any non-volatile storage, such as Flash. The Cortex-M4 executable image always executes from the internal SRAM of the RTP M4. The MA35D1 supports mapping RTP M4 memory space 0x20000 ~ 0x3FFFFF to DDR offset 0x2000 ~ 0x3FFFFF. But in Linux, RTP M4 can only use 0x20000 ~ 0x7FFFF because Linux kernel image starts from DDR 0ffset 0x80000. The maximum size of RTP M4 executable area is 0x80000 which includes the run-time memory.

The Cortex-A35 can load executable images into the dedicated 128 KB SRAM of RTP M4 simply by memory copy. For images larger than 128 KB, the portion over 128 KB will be loaded to DDR offset 128 KB.

Nuvoton provides Nu-Link2-Pro ICE plugin for Keil, Eclipse, and IAR, in which users can also load executable images into RTP M4 and perform debug or run code. MA35D1 adopts the Linux remoteproc framework to load the RTP M4 firmware to RTP M4 SRAM. From the Cortex-A35 view, RTP M4 SRAM is mapped to address space 0x24000000 ~ 0x2401FFFF. From the RTP M4 view, the SRAM is mapped to address space 0x00000000 ~ 0x0001FFFF. Related instructions will be detailed in section 2.1.1.

The MA35D1 Wormhole Controller (WHC) supports bi-dir4ectional data exchange between Cortex-A35 and RTP M4. WHC is a controller provided by MA35D1 to handle the inter-processor communication between Cortex-A35 and RTP M4. MA35D1 Linux RTP ecosystem uses WHC to transfer simple commands, and uses share memory to transfer a larger amount of data. However, it should be noted that the current MA35D1 software only provides a way to use share memory to transfer data, not provide a way to directly let the user control WHC to send command.

The MA35D1 Linux BSP supports the driver for Linux rpmsg framework for Cortex-A35 to send and receive data to RTP M4. And MA35D1 RTP BSP supports for building software code running under the OpenAMP framework, which can support send and receive data to Cortex-A35. OpenAMP is a standardized embedded multi-core framework, whose purpose is to enable RTOS and bare metal programs developed in AMP (Asymmetric Multiprocessing) system design to communicate with the interface provided by the open source Linux community. Related instructions detail in section 2.1.2 and 2.2.1.

In addition, due to the memory space shared between different cores, it is necessary to implement a synchronization mechanism to prevent different cores from accessing the same memory space and causing unpredictable behavior. The Hardware Semaphore module of MA35D1 provides eight hardware semaphores, by which users can manage the synchronization between different processors with semaphore keys. Related instructions detail in section 2.1.4 and 2.2.2.

The MA35D1 also provides a system security peripheral configuration controller (SSPCC), which can configure the security attribute of all peripherals, such as SRAM and GPIO. The user determines a peripheral resource be allocated to Cortex-A35 or Cortex-M4 by programming SPPCC. In the MA35D1 Linux platform, only Secure Monitor of TF-A can program SSPCC because SSPCC is secure-only while Linux kernel in running under non-secure mode. Related instructions detail in section 2.1.3. For detailed introduction of Trusted Firmware-A (TF-A), please refer to ”NuMicro® Family MA35D1 TF-A User Manual”.

2. Cortex-A35 and Cortex-M4 Management

2.1 Cortex-A35

2.1.1 Loading Cortex-M4 Firmware

Since the RTP M4 does not have its own Flash, the required firmware needs to be loaded from outside of RTP M4. In addition to using Nu-Link2-Pro ICE, the MA35D1 supports to load executable images into RTP M4 SRAM from the Cortex-A35 side using the Linux built-in remoteproc framework. Of course, the user needs to copy the compiled firmware to the Linux file system before using the Cortex-A35 to load the RTP M4 images. Currently Linux only supports loading firmware in ELF format. Users can also use the remoteproc driver to start/stop RTP M4 CPU execution.

Figure 2-1 is a schematic diagram of Cortex-A35 loading Cortex-M4 executable image and starting/stopping RTP M4 by Cortex-A35.

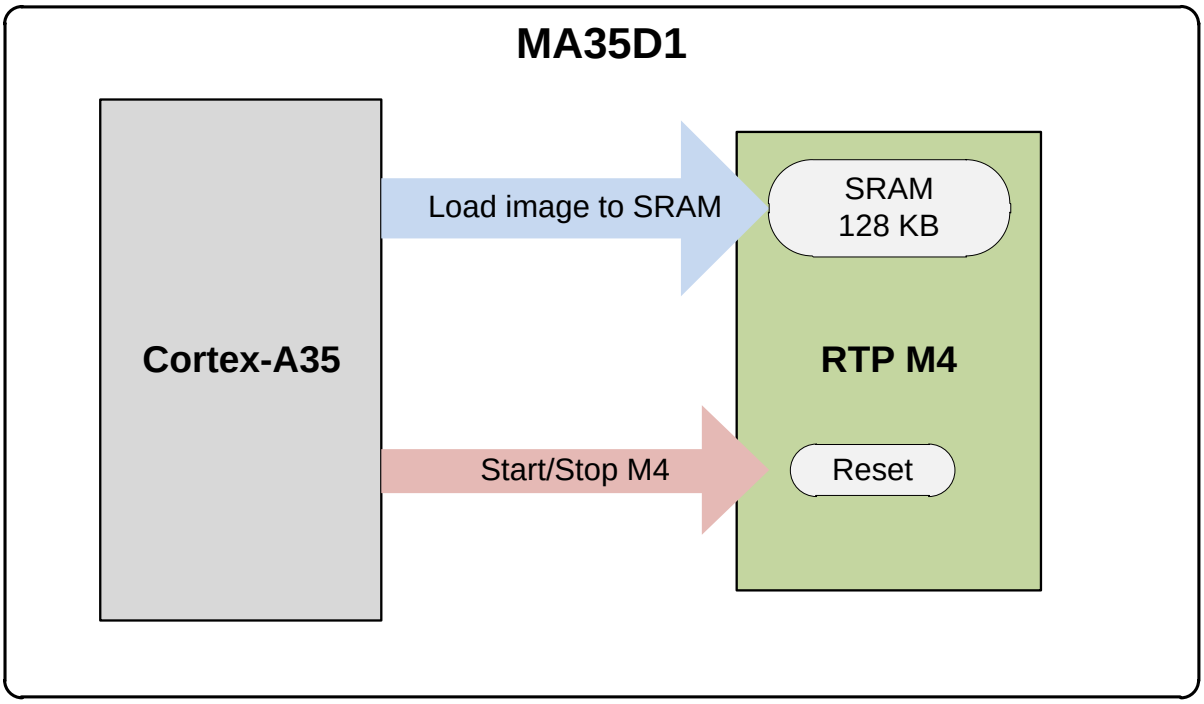

Figure 2-1 Load and Start/Stop RTP M4 Image

To support RTP M4 applications that require memory larger than 128 KB, the MA35D1 supports to map RTP M4 memory space 0x20000 ~ 0x3FFFFF to DDR offset 0x20000 ~ 0x3FFFFF. However, due to Linux kernel image locates at DDR offset 0x80000, available DDR memory is reduced to 0x20000 ~ 0x7FFFF under Linux. As a result, the maximum memory available for a RTP M4 application is 512 KB under MA35D1 Linux platform.

Figure 2-2 shows the memory map of MA35D1 Cortex-A35 and RTP M4. For an image small than 128 KB, Linux will always load it to RTP M4 internal SRAM. For an image larger than 128 KB, Linux will load the first 128 KB to RTP M4 SRAM, and load the remaining to DDR offset 0x20000, which is 0x80020000 from Linux view.

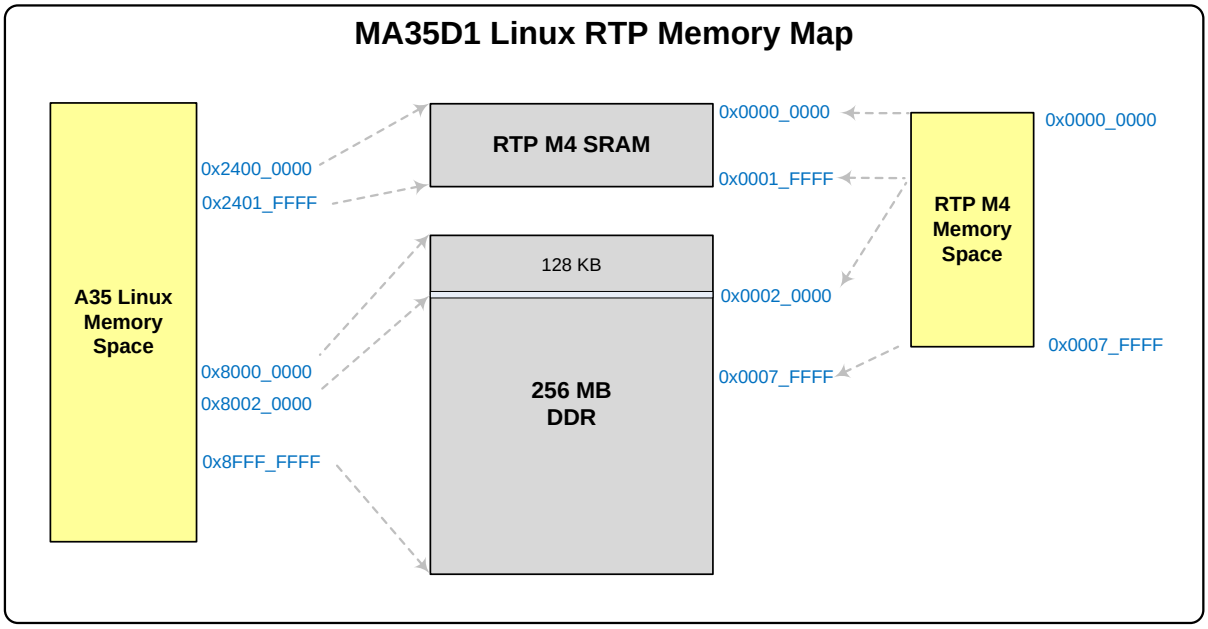

Figure 2-2 Linux RTP Memory Map

The following introduces how users can use Linux's remoteproc framework to load RTP M4 images and start/stop RTP M4.

Before starting to run the Linux kernel, users need to set up the kernel configuration and device tree. After the setting is complete, you need to recompile and run the Linux kernel.

The following are the kernel configuration and device tree settings:

1. Set the Linux kernel configuration to enable the MA35D1 remoteproc driver.

```
Device Drivers  --->
  Remoteproc drivers  --->
       [*] Support for Remote Processor subsystem
       <*>   MA35D1 remoteproc support
```

2. Device tree configuration.

The device tree node setting of remoteproc:

```
rproc {
           compatible = "nuvoton, ma35d1-rproc";
           mboxes = <&wormhole 1>;
           resets = <&reset MA35D1_RESET_CM4>;
     };
```

After starting the Linux kernel, user can load and start/stop the remote processor firmware through the sysfs interface. (The sysfs filesystem is a pseudo-filesystem which provides an interface to kernel data structures).

The sysfs interface description is as follows:

1. Add a new firmware path

The firmware components are stored in the file system, by default in the /lib/firmware/ folder. Optionally another location can be set. In this case, the remoteproc framework parses this new path in priority. The below command adds a new path for firmware parsing:

```
$> echo -n <firmware_path> > /sys/module/firmware_class/parameters/path
```

2. Rename the firmware file name

If the firmware elf filename differs from the default one (rproc-%s-fw), set the name with the following command: (replace X with remoteproc instance number: 0 by default)

```
$> echo -n <firmware_name.elf> > /sys/class/remoteproc/remoteprocX/firmware
```

3. Load and start the firmware

Users can use the following command to load and start the firmware:

```
$> echo start > /sys/class/remoteproc/remoteprocX/state
```

4. Stop the firmware

Users can use the following command to stop the firmware:

```
$> echo stop > /sys/class/remoteproc/remoteprocX/state
```

2.1.2 Communication with Cortex-M4

The MA35D1 can load RTP M4 firmware from Cortex-A35, and provide WHC and shared memory. Cortex-A35 and RTP M4 perform command handshaking through WHC and exchange mass data through the shared memory. The MA35D1 provides (4M - 128K) bytes share memory of which memory address space is 0x8002_0000 ~ 0x803F_FFFF from the Cortext-A35 view. From the RTP M4 view, shared memory address space is 0x0002_0000 ~ 0x003F_FFFF.

The MA35D1 and RTP M4 communicate with each other using the message passing mechanism of Linux RPMsg framework. The Linux RPMsg framework can set the address and size of shared memory and use the file system for messages that the user wants to receive/transmit to remote CPUs via shared memory. That is to say, users can easily communicate with the remote processor by means of read/write files.

Figure 2-3 is a schematic diagram of Cortex-A35 using RPMsg framework to communicate with RTP M4.

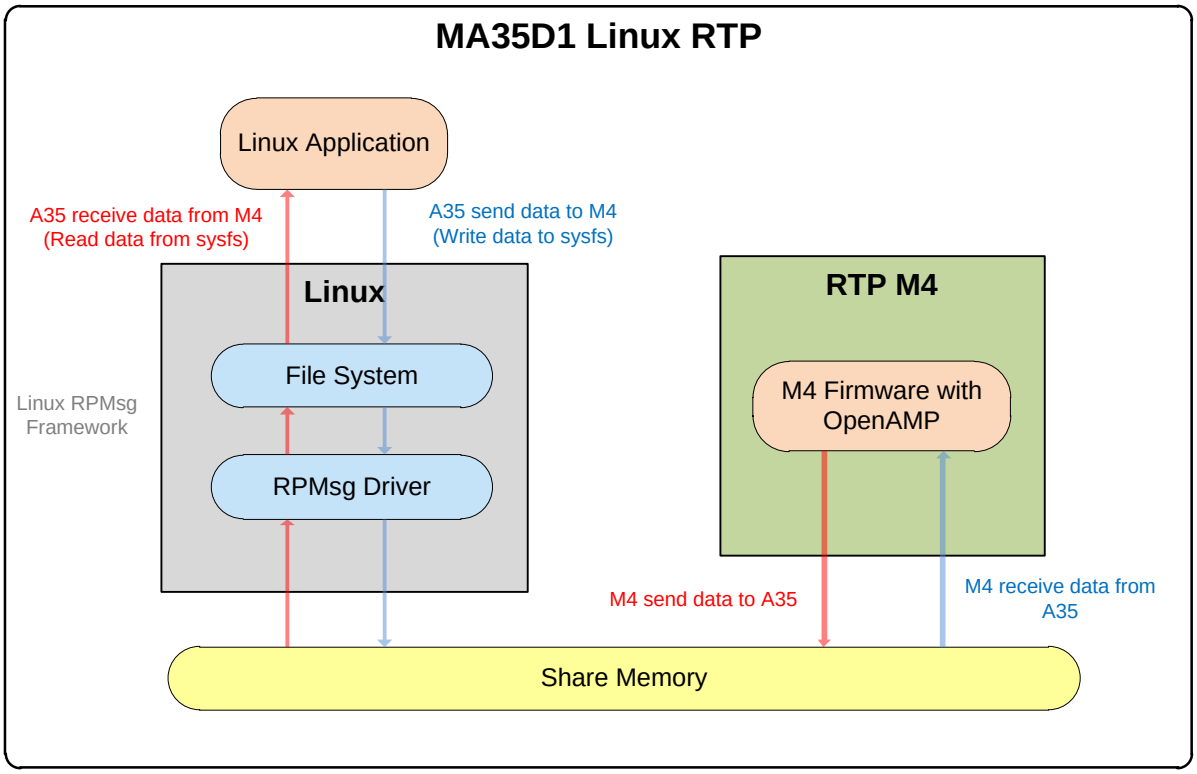

Figure 2-3 Linux RPMsg Framework

The following introduces how users can use Linux's RPMsg framework to communicate with RTP M4.

Before starting to run the Linux kernel, users need to set up the kernel configuration and device tree. After the setting is complete, you need to recompile and run the Linux kernel.

The following are the kernel configuration and device tree settings:

1. kernel configuration to enable the MA35D1 rpmsg driver

```
Device Drivers  --->
  Rpmsg drivers  --->
       -*-   RPMSG device interface
       <*>   MA35D1 Shared Memory Driver
```

2. Device tree configuration

The device tree node setting of rpmsg:

“compatible” should be set to "nuvoton, ma35d1-rpmsg", which can load the rpmsg driver.

```
rpmsg {
      compatible = "nuvoton, ma35d1-rpmsg";
```

“share-mem-addr” defines share memory address and should be 0x2401FF00.

```
      share-mem-addr = <0x2401FF00>;
```

“tx-smem-size” defines tx share memory size and should be 128.

```
        tx-smem-size = <128>;
```

“rx-smem-size” defines rx share memory size and should be 128.

```
        rx-smem-size = <128>;
};
```

After starting the Linux kernel, users can follow the steps below to send/receive messages to/from a remote CPU via shared memory:

1. Open rpmsg control device, for example:

```
        open("/dev/rpmsg_ctrl0", O_RDWR | O_NONBLOCK);
```

2. Create rpmsg end point, for example:

```
        ioctl(rpfd, RPMSG_CREATE_EPT_IOCTL, eptinfo);
```

and open rpmsg endpoint, for example:

```
        open("/dev/rpmsg0", O_RDWR | O_NONBLOCK);
```

3. Use write/read function to send/receive messages to/from a remote CPU via shared memory, for example:

```
        write(fd, Tx_Buffer, 128); // send message
        read(fd, Rx_Buffer, 128); // read message
```

4. In addition to the above steps, the user can also use the polling function to confirm that the write function has sent a message to the shared memory and RTP M4 has received the message, or use the polling function to confirm that there is a message from RTP M4 in the shared memory. Shared memory, and then use the read function to read the message.

The following is an example of a program that uses rpmsg to communicate between the master and slave processors.

```
struct rpmsg_endpoint_info {
        char name[32];
        __u32 src;
        __u32 dst;
};

static int rpmsg_create_ept(int rpfd, struct rpmsg_endpoint_info *eptinfo)
{
        int ret;
        ret = ioctl(rpfd, RPMSG_CREATE_EPT_IOCTL, eptinfo);
        if (ret)
                perror("Failed to create endpoint.\n");
        return ret;
}
```

```
int main(int argc, char **argv)
{
	char *dev[10]={"/dev/rpmsg_ctrl0", " "};
	unsigned int i;
	int rev1, rev2;
	struct rpmsg_endpoint_info eptinfo;
	int ret;
	int err;
	unsigned char Tx_Buffer[130];
	unsigned char Rx_Buffer[130];

	fd[0] = open("/dev/rpmsg_ctrl0", O_RDWR | O_NONBLOCK);
	if (fd[0] < 0) {
		perror("Failed to open \n");
		return 0;
	}

	strcpy(eptinfo.name, "rpmsg-test");
	eptinfo.src = 0;
	eptinfo.dst = 0xFFFFFFFF;

	ret = rpmsg_create_ept(fd[0], &eptinfo);
	if (ret) {
		perror("Failed to create RPMsg endpoint.\n");
		return -EINVAL;
	}

	fd[1] = open("/dev/rpmsg0", O_RDWR | O_NONBLOCK);

	if (fd[1] < 0) {
		perror("Failed to open rpmsg0 \n");
		return -EINVAL;
	}

	while(1)
	{
		struct pollfd fds[] = {
			{
				.fd   = fd[1],
				.events    = POLLOUT,
			},
```

```
        };

         for(i = 0; i < 128; i++)
         {
               Tx_Buffer[i] = (255-i);
         }

         rev1 = write(fd[1], Tx_Buffer, 10);
         if (rev1 < 0) {
               perror("Failed to write \n");
               return -EINVAL;
         }

         while(1) // wait send message finish
         {
               err = poll(fds, 1, 10000);
               if((err == -1) || (err == 0))
               {
               }
               else
               {break;}
         }
       break;
    }

    while(1)
    {
         struct pollfd fds[] = {
               {
                     .fd   = fd[1],
                     .events    = POLLIN,
               },
         };

         while(1) // wait share memory receive message
         {
               err = poll(fds, 1, 10000);
               if((err == -1) || (err == 0))
               {
               }
               else
```

```
                {break;}
            }

            rev1 = read(fd[1], Rx_Buffer, 128);
            if (rev1 < 0) {
                perror("Failed to read \n");
                return -EINVAL;
            }

            printf("\n Receive %d bytes data from M4: \n", rev1);

            for(i = 0; i < rev1; i++)
            {
                printf(" 0x%x, \n", Rx_Buffer[i]);
            }
        }
        return 0;
}
```

2.1.3 Assigning Internal Peripheral Resources

Users can assign internal peripheral resources to the master processor or slave processor by programming SSPCC with TF-A (Trusted Firmware-A).

On assigning the ownership of a peripheral device, the user must also assign the corresponding ownerships of GPIO pins used by that device.

The following is the device node sample, which describes the attribute of SSPCC.

```
sspcc: sspcc@404F0000 {
```

“compatible” must set to “nuvoton,ma35d1-sspcc”. Register base address of System security peripheral configuration controller (SSPCC) is 0x404F0000.

```
        compatible = "nuvoton,ma35d1-sspcc";
        reg = <0x0 0x404F0000 0x0 0x1000>;
```

“config” sets all peripherals’ attribution. It includes NAND, SDH, UART, Timer, SPI, Crypto, etc. The secure attribute of each peripheral is one of TZS (secure), TZNS (non-secure), and subM (RTP M4). The secure attribute of all peripherals are defined in *plat/nuvoton/ma35d1/includes/sspcc.h*. This header file lists all the peripherals whose attribute to be changed. The attribute of peripherals not listed here remains power-on by default. The following example assigns UART1 secure attribute to subM (RTP M4).

```
        config = <UART1_SUBM>;
        gpio_s = <PA2_SUBM>,
                 <PA3_SUBM>;
```

```
};
```

As to the configuration of SRAM0, the 128 KB SRAM0 is for RTP M4 to retain code and data. By default, the whole SRAM0 is shared between RTP M4 and Cortex-A35. To prevent the SRAM0 from being unexpectedly modified by Cortex-A35 after RTP M4 is powered up and executed, SSPCC provides a feature that can configure certain range of SRAM to be that only RTP M4 has access right.

The SR0BOUND(SSPCC_SRAMSB[4:0]) is used to set a 16 Kbytes-aligned boundary region starting from offset 0 of SRAM0 that can only access by RTP M4. The region above the boundary will be the share memory between Cortex-A35 and RTP M4.

2.1.4 Hardware Semaphore

Since some memory area are shared by different cores, it is necessary to implement a synchronization mechanism to prevent from concurrent accessing to the same memory area and resulting in memory inconsistence. The MA35D1 Hardware Semaphore provides eight hardware semaphores, which can support synchronization between different cores by the semaphore keys.

The driver for MA35D1 hardware semaphore is:

- *drivers/hwspinlock/ma35d1_hwsem.c*

Please follow the setting below to enable MA35D1 hardware semaphore support.

```
Device Drivers  --->
  [*] Hardware Spinlock drivers  --->
    <*>   MA35D1 Hardware Semaphore support
```

The followings describe the hardware semaphore node in the MA35D1 device tree.

The base address of hwsem controller is 0x40380000.

```
hwsem: hwspinlock@40380000 {
```

“compatible” must be set to “nuvoton,ma35d1-hwsem”.

```
      compatible = "nuvoton,ma35d1-hwsem";
```

“reg” defines the base address and size of hardware semaphore control register.

```
      reg = <0x0 0x40380000 0x0 0x1000>;
```

Set “okay” to enable hardware semaphore; otherwise, set to “disable”.

```
      status = "okay";
};
```

Also, users can learn more about Hardware Spinlock Framework using the following links.

https://www.kernel.org/doc/Documentation/hwspinlock.txt

2.1.5 Power Down / Wakeup

The MA35D1 can make Cortex-A35 and RTP M4 enter Power-down mode respectively. Under Linux command shell, using the following command can make Cortex-A35 enter the Power-down mode, which writes “mem” into /sys/power/state.

```
$ echo mem > /sys/power/state
```

After Cortex-A35 core enters Power-down mode, internal interrupts generated from MA35D1 peripheral devices, for example WHC and RTC, and external interrupts generated from external devices, for example USB, can wake up the Cortex-A35 core. Users need to refer to the MA35D1 user manual to know how to use different devices to wake up the Cortex-A35 core. Likewise, after the RTP M4 enters Power-down mode, interrupts generated by those MA35D1 devices with “subM” security attribute can wake up RTP M4.

It is also worth noting that when the Cortex-A35 enters Power-down mode, the DDR will enter self-refresh mode. At this time, the DDR cannot be accessed. Therefore, the firmware running on the RTP M4 should never access the shared memory located in the DDR. However, when the RTP-M4 goes into Power-down mode, the Cortex-A35 does not have this limitation.

2.2 RTP M4

2.2.1 Using OpenAMP Control Share Memory

OpenAMP (Open Asymmetric Multi-Processing) is a framework providing the software components needed to enable the development of software applications for AMP systems. It allows operating systems to interact within a broad range of complex homogeneous and heterogeneous architectures and allows asymmetric multiprocessing applications to leverage parallelism offered by the multicore configuration.

The MA35D1 RTP M4 supports OpenAMP framework for users to use this framework to communicate with Cortex-A35.

The OpenAMP architecture on the RTP M4 side of the MA35D1 is very similar to that of the Linux rpmsg framework. Users can refer to Figure 2-3.

Follow the steps below to send data to or receive data from share memory:

1. Execute MA35D1_OpenAMP_Init(int RPMsgRole, rpmsg_ns_bind_cb ns_bind_cb) to initialize mailbox (WHC) and share memory.

   The parameter “RPMsgRole” should be “RPMSG_REMOTE” and “ns_bind_cb” be “NULL”, for example,

```
MA35D1_OpenAMP_Init(RPMSG_REMOTE, NULL);
```

2. Execute OPENAMP_create_endpoint(struct rpmsg_endpoint *ept, const char *name, uint32_t dest, rpmsg_ept_cb cb, rpmsg_ns_unbind_cb unbind_cb) to create rpmsg endpoint.

   Users create a struct rpmsg_endpoint and set the address of this struct to the parameter “*ept”. Users can also set endpoint name to parameter “*name”. The parameter “dest” should be “RPMSG_ADDR_ANY“. The parameter “cb” is the call back function. When the endpoint data received, the call back function will be called. Then users can get share

memory data in this call back function.

```
static int rx_callback(struct rpmsg_endpoint *rp_chnl, void *data, size_t len, uint32_t
src, void *priv)
{
      uint8_t received_rpmsg[128];
      //Users can get shared memory data from address “src” and size is “len”
      memcpy((void *)received_rpmsg, (const void *)src, len);
}
int32_t main (void)
{
    struct rpmsg_endpoint resmgr_ept;
      OPENAMP_create_endpoint(&resmgr_ept, "rpmsg-sample", RPMSG_ADDR_ANY, rx_callback,
NULL);
}
```

3. Use OPENAMP_check_for_message() to wait for new data from linux.

```
{
    struct rpmsg_endpoint resmgr_ept;
      OPENAMP_check_for_message(&resmgr_ept);
}
```

4. Use OPENAMP_send_data() to send new data to Linux and use OPENAMP_check_TxAck() to wait for a response from Linux, which means Linux has received new data.

```
{
      struct rpmsg_endpoint resmgr_ept;
      uint8_t transmit_rpmsg[128];
      // Send data to Cortex-A35
      OPENAMP_send_data(&resmgr_ept, transmit_rpmsg, 5);

      while(1)
    {
            // check Cortex-A35 response ack
         if(OPENAMP_check_TxAck(&resmgr_ept) == 1)
         break;
    }
}
```

5. The following is an example of controlling shared memory:

```
#define M4_COMMAND_ACK 0x81
static uint32_t rx_status = 0;
#define tx_rx_size 128
uint8_t received_rpmsg[tx_rx_size];
```

```
uint8_t transmit_rpmsg[tx_rx_size];


static int rx_callback(struct rpmsg_endpoint *rp_chnl, void *data, size_t len, uint32_t
src, void *priv)
{
    uint32_t *u32Command = (uint32_t *)data;
    uint32_t i;


    if(*u32Command == COMMAND_RECEIVE_A35_MSG)
    {
        memcpy((void *)received_rpmsg, (const void *)src, len > sizeof(received_rpmsg) ?
sizeof(received_rpmsg) : len);


        printf("\n Receive %d bytes data from Cortex-A35: \n", len);
        for(i = 0; i < len; i++)
        {
            printf(" 0x%x \n", received_rpmsg[i]);
        }


        rx_status = 1;
    }
    else
    {
        printf("\n unknow command!! \n");
    }


    return 0;
}


int32_t main (void)
{
    struct rpmsg_endpoint resmgr_ept;
    uint32_t i;
    int ret;


    MA35D1_OpenAMP_Init(RPMSG_REMOTE, NULL);
    OPENAMP_create_endpoint(&resmgr_ept, "rpmsg-sample", RPMSG_ADDR_ANY, rx_callback,
NULL);


    while(1) // wait message from A35
    {
        OPENAMP_check_for_message(&resmgr_ept);
```

```
        if(rx_status)
        {
            rx_status = 0;
            break;
        }
    }

    for(i = 0; i < tx_rx_size; i++)
    {
        transmit_rpmsg[i] = i;
    }

    // send message to Cortex-A35
    ret = OPENAMP_send_data(&resmgr_ept, transmit_rpmsg, 5);
    if (ret < 0)
    {
        printf("Failed to send message\r\n");
    }

    printf("\n Transfer %d bytes data to Cortex-A35 \n", ret);

    while(1) // wait send message finish and Cortex-A35 response ack
    {
        if(OPENAMP_check_TxAck(&resmgr_ept) == 1)
        break;
    }
    while(1);
}
```

3. Hardware Semaphore

The hardware semaphores on the RTP M4 side can be controlled by setting the HWSEM_SEM0 ~ HWSEM_SEM7 registers.

Registers HWSEM_SEM0 ~ HWSEM_SEM7 can be accessed by both Cortex-A35 and RTP M4, and are used to control the eight semaphores for synchronization. A write to these register will try to hold the semaphore. ID (HWSEM_SEMx[3:0]) indicates the current owner of the semaphore and a write to these with the same KEY (HWSEM_SEMx[15:8]) unlock the semaphore.

4. Debugging RTP M4 with Nu-Link2-Pro

In addition to loading RTP M4 firmware using the method described in section 2.1.1, the MA35D1 can also use Nu-Link2-Pro to program and debug M4 firmware.

Nu-Link2-Pro supports using Keil uVision, IAR Embedded Workbench and NuEclipse as firmware debugging environment.

4.1 Preliminary Preparation

Before debugging, use the Nu-Link2-Pro and enable the CMSIS-DAP feature by the following steps:

1. Upgrade the Nu-Link2-Pro firmware with version later than v7174.
2. Open NU_CFG.txt file located in the NuMicro MCU disk folder.
3. Set CMSIS-DAP=1 and re-plug the Nu-Link2-Pro.

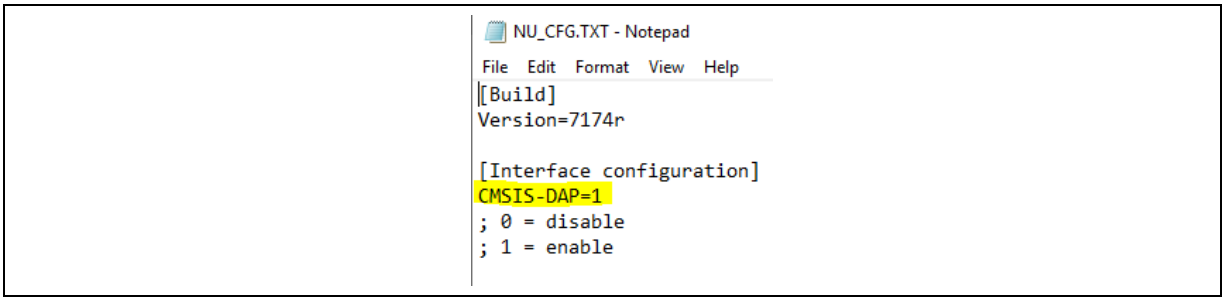

Figure 4-1 Enable CMSIS-DAP Feature

4.2 Keil uVision

To open the Keil project, double-click the uvproj file under the Keil/ directory, or open the uvproj file from the “Project” pull-down menu after launching the Keil uVision as shown in Figure 4-2.

Figure 4-2 Open Keil Project

In “**Options for Target – Linker**” tab, modify “**R/O Base**” and “**R/W Base**” value as shown in Figure 4-3.

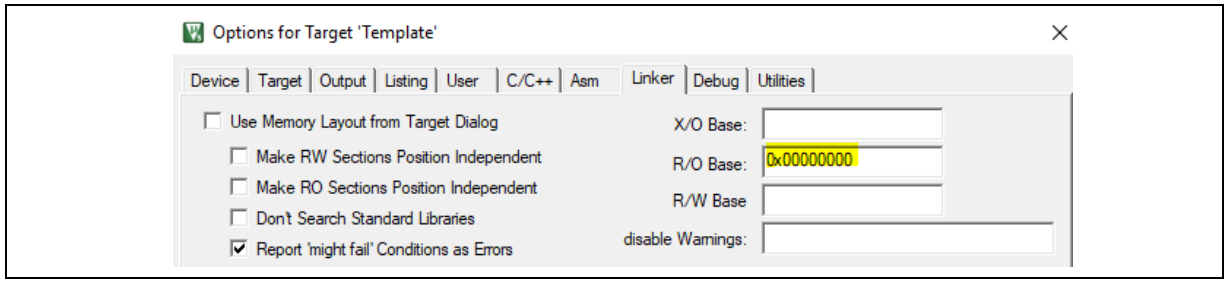

Figure 4-3 Linker Configuration

To build the project click the “Build” icon shown in Figure 4-4.

Figure 4-4 Build Keil Project

In “Options for Target – Debug” tab, select CMSIS-DAP Debugger Driver as shown in Figure 4-5 and click settings.

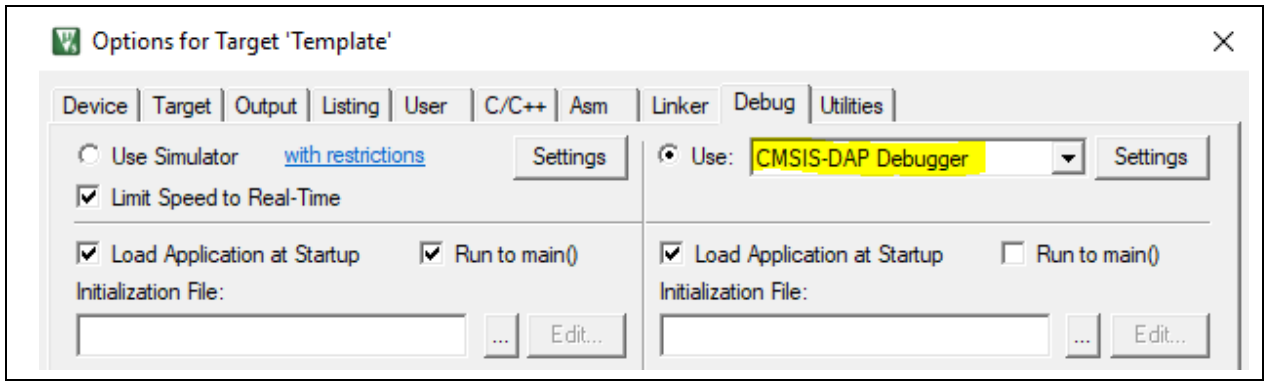

Figure 4-5 Select Debugger Driver

Select SW adapter “Nu-Link 2 CMSIS-DAP” and check IDCODE for device connection and set AP to 0x02 as shown in Figure 4-6.

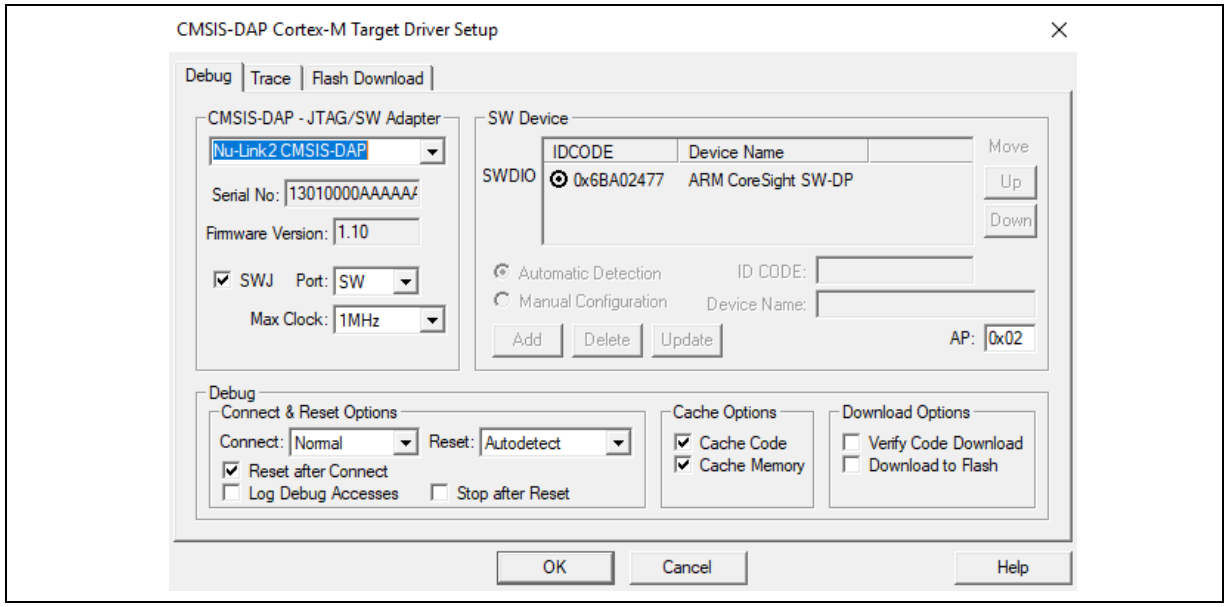

Figure 4-6 CMSIS-DAP Debug Settings

In “**Options for Target – Utilities**” tab, uncheck “**Update Target before Debugging**” option as shown in Figure 4-7.

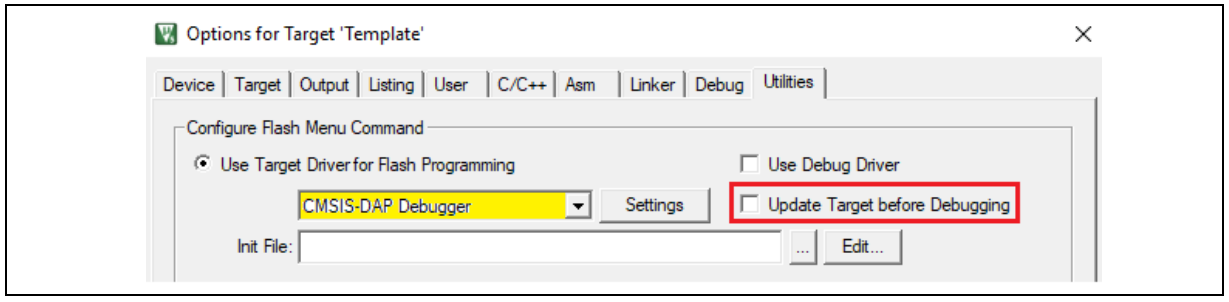

Figure 4-7 CMSIS-DAP Utilities Settings

uVision will load the built image and enter debug mode after click the “**Start/Stop Debug Session**” icon as shown in Figure 4-8.

Figure 4-8 Debug Keil Project

4.3 IAR Embedded Workbench

To open the IAR workspace, double-click the eww file under IAR/ directory or open the eww file from the “File” pull-down menu after launching the IAR Embedded Workbench as shown in Figure 4-9.

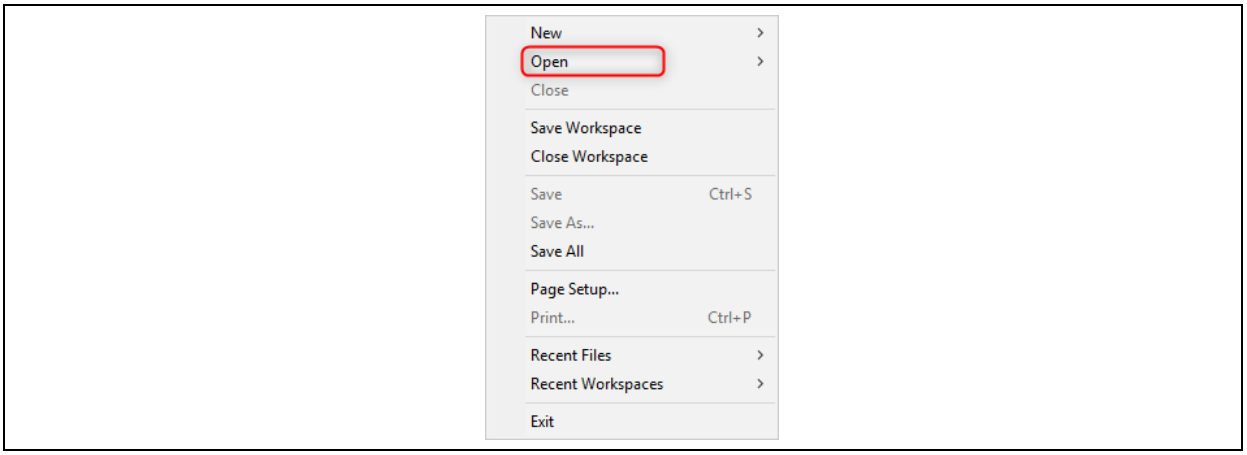

Figure 4-9 Open IAR Workspace

Open Options to modify setting as shown in Figure 4-10.

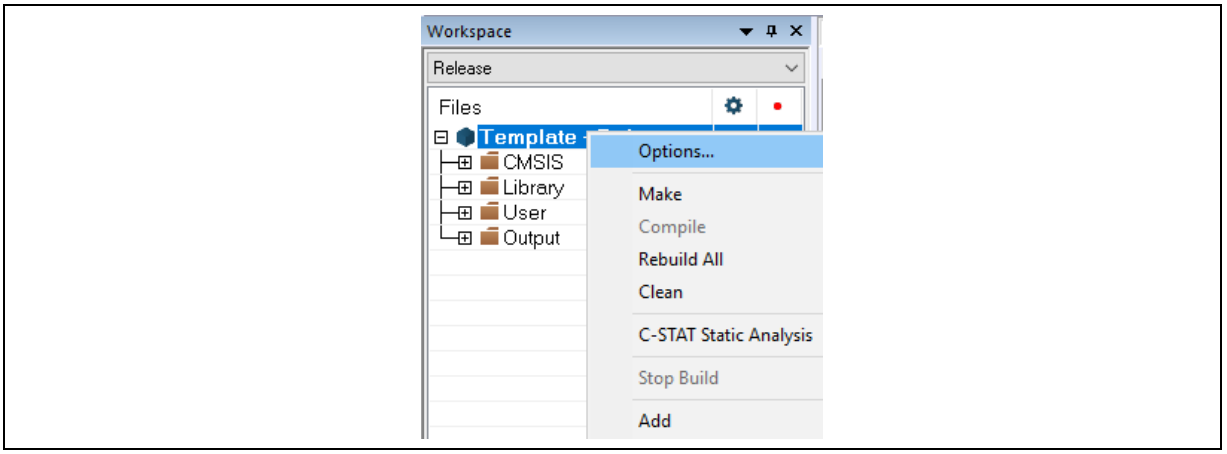

Figure 4-10 Workspace Options

Select Cortex-M4F or Cortex-M4 with floating point option as shown in Figure 4-11.

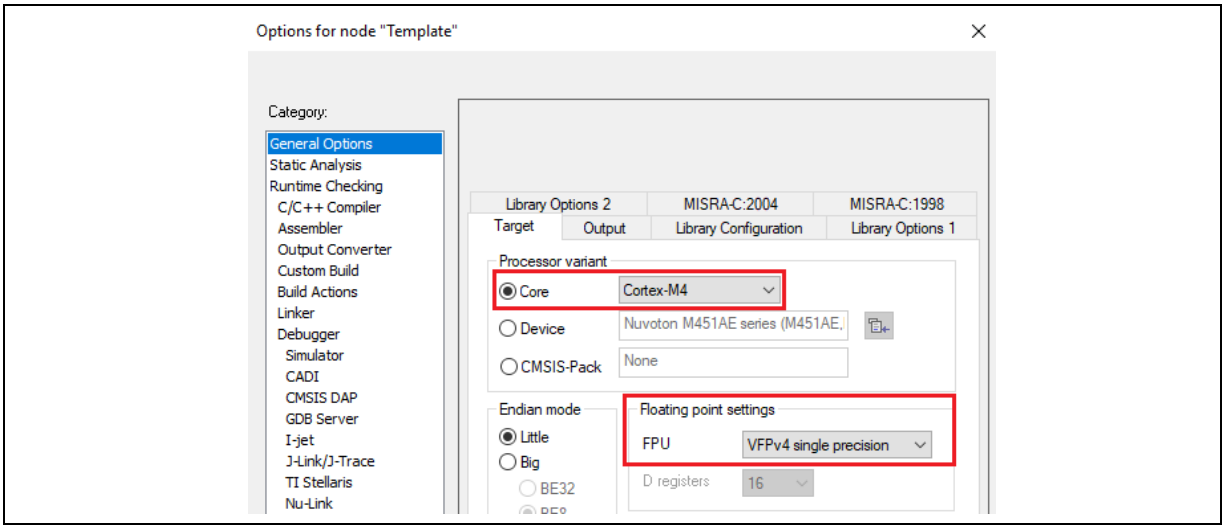

Figure 4-11 General Options

Click “**Override default**” option to edit linker configuration for debugging on SRAM address as shown in Figure 4-12.

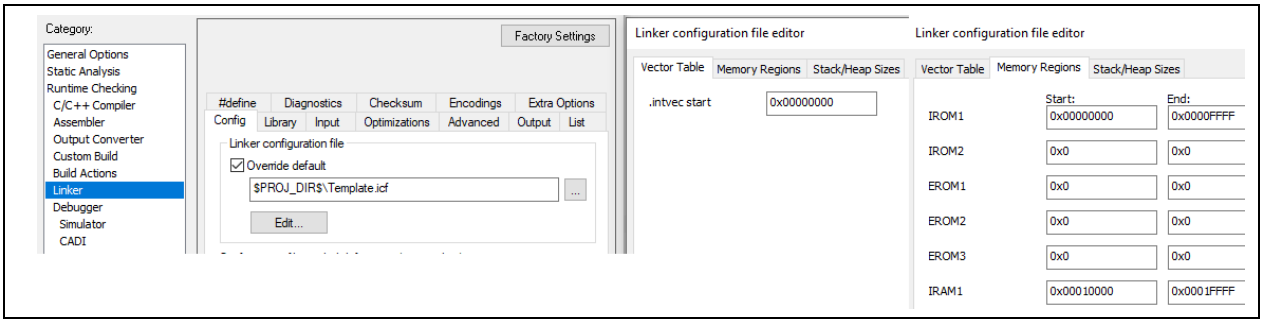

Figure 4-12 Linker Options

Select CMSIS-DAP driver and uncheck “**Use flash loader**” option as shown in Figure 4-13.

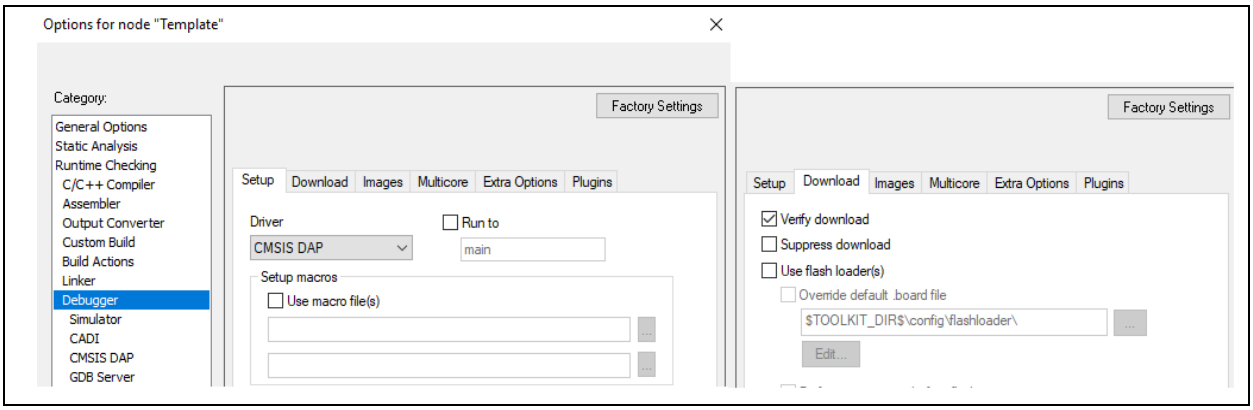

Figure 4-13 Debugger Options

Select SWD interface for CMSIS-DAP setting as shown in Figure 4-14.

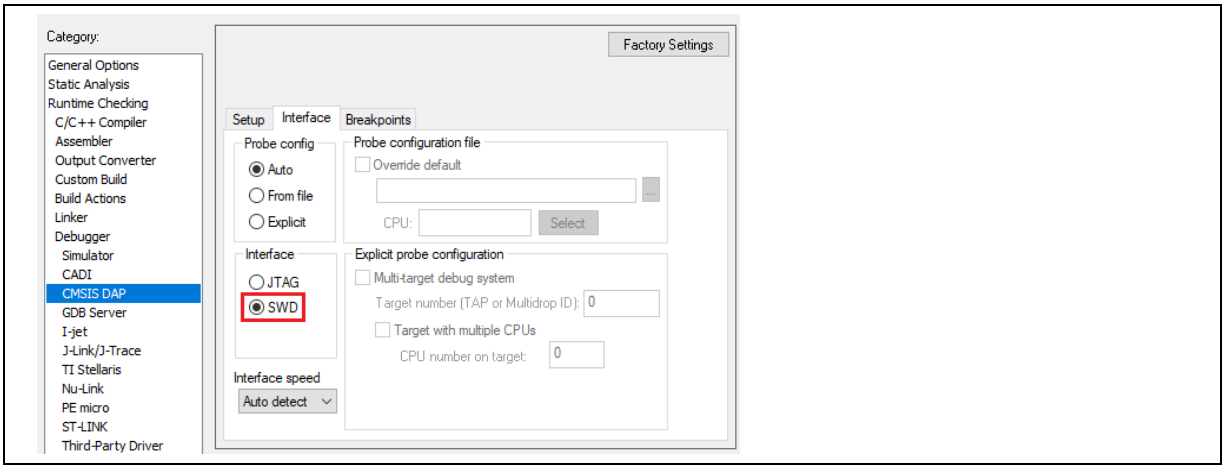

Figure 4-14 CMSIS-DAP Interface Setting

To build the workspace click the “Make” icon as shown in Figure 4-15.

Figure 4-15 Build IAR Workspace

Modify “CMSIS-DAP Memory Configuration” based on SRAM address range as shown in Figure 4-16.

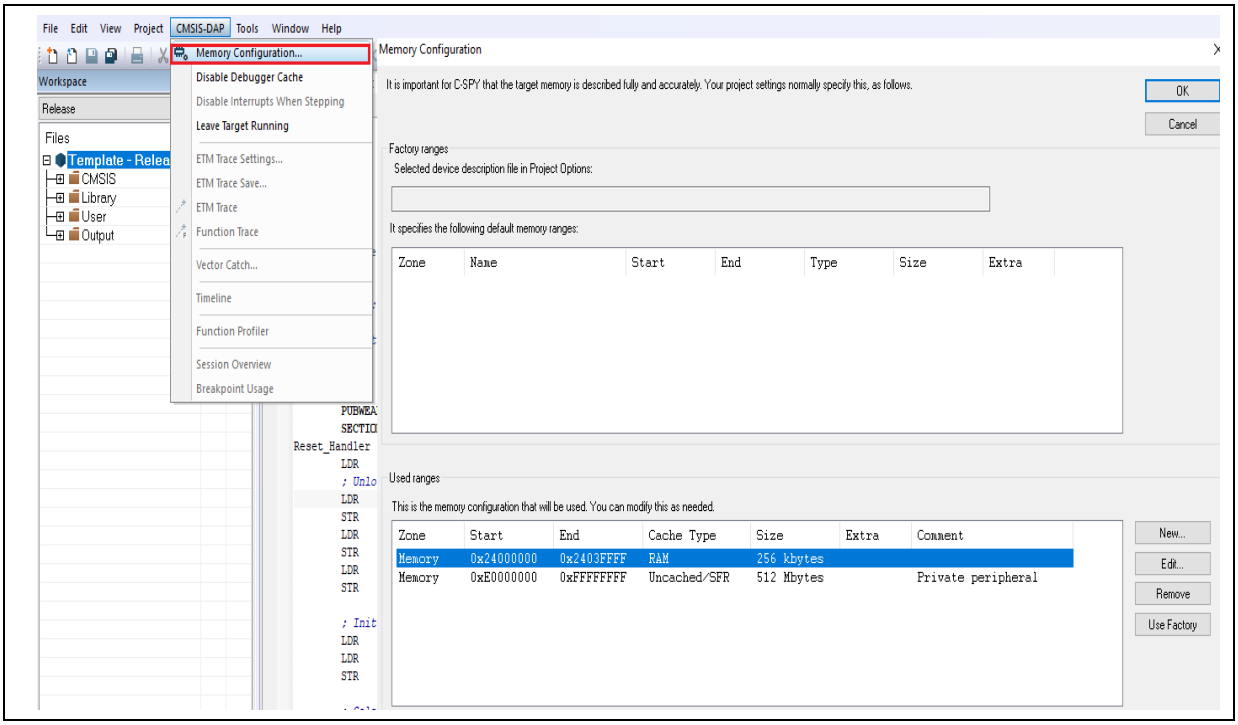

Figure 4-16 CMSIS-DAP Memory Configuration

IAR will load the built image and enter debug mode after click the “Download and Debug” icon as shown in Figure 4-17.

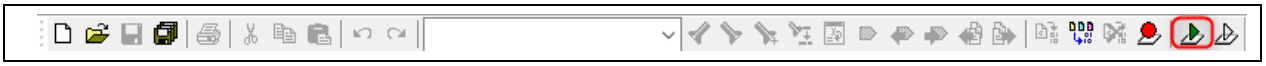

Figure 4-17 Debug IAR Project

4.4 NuEclipse

To use open the NuEclipse project, first launch the NuEclipse tool, select “**Import…**” from the File menu as shown in Figure 4-18.

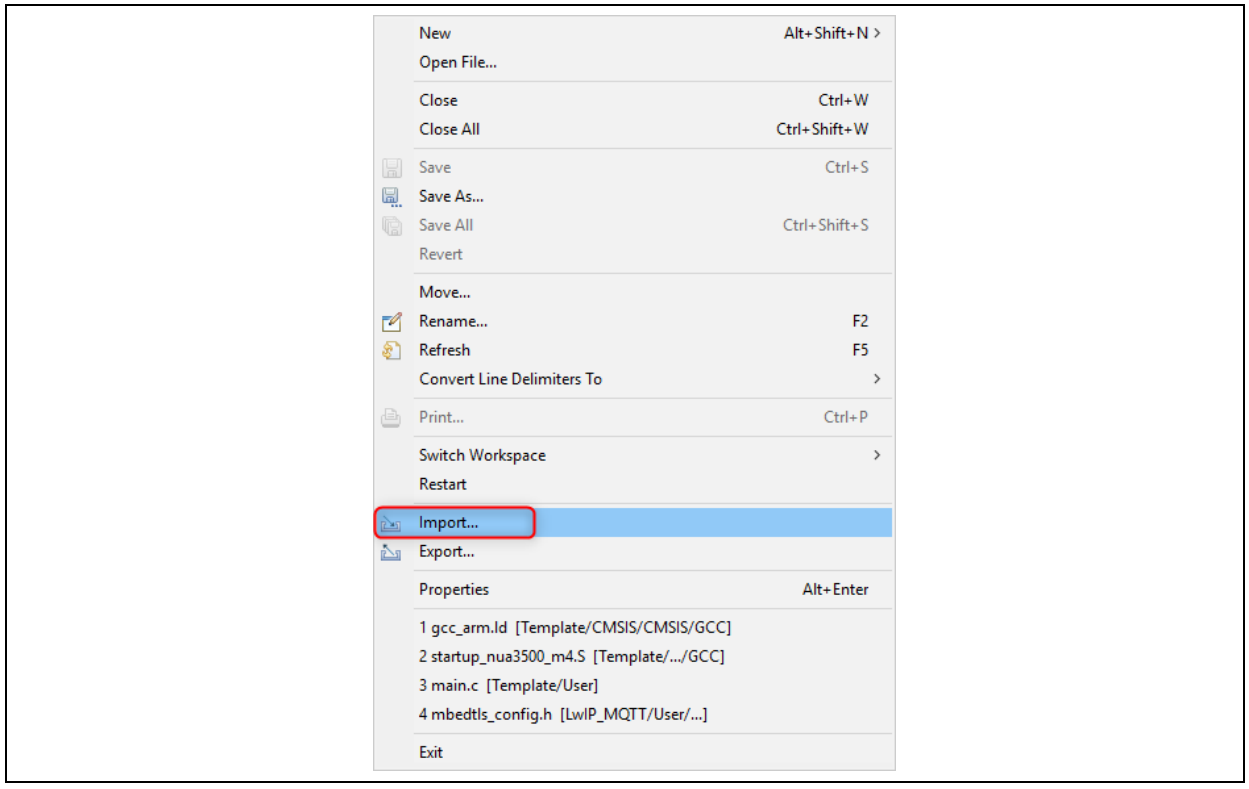

Figure 4-18 NuEclipse Import Project

Next, select import existing project to workspace and click "**Next >**" button as shown in Figure 4-19.

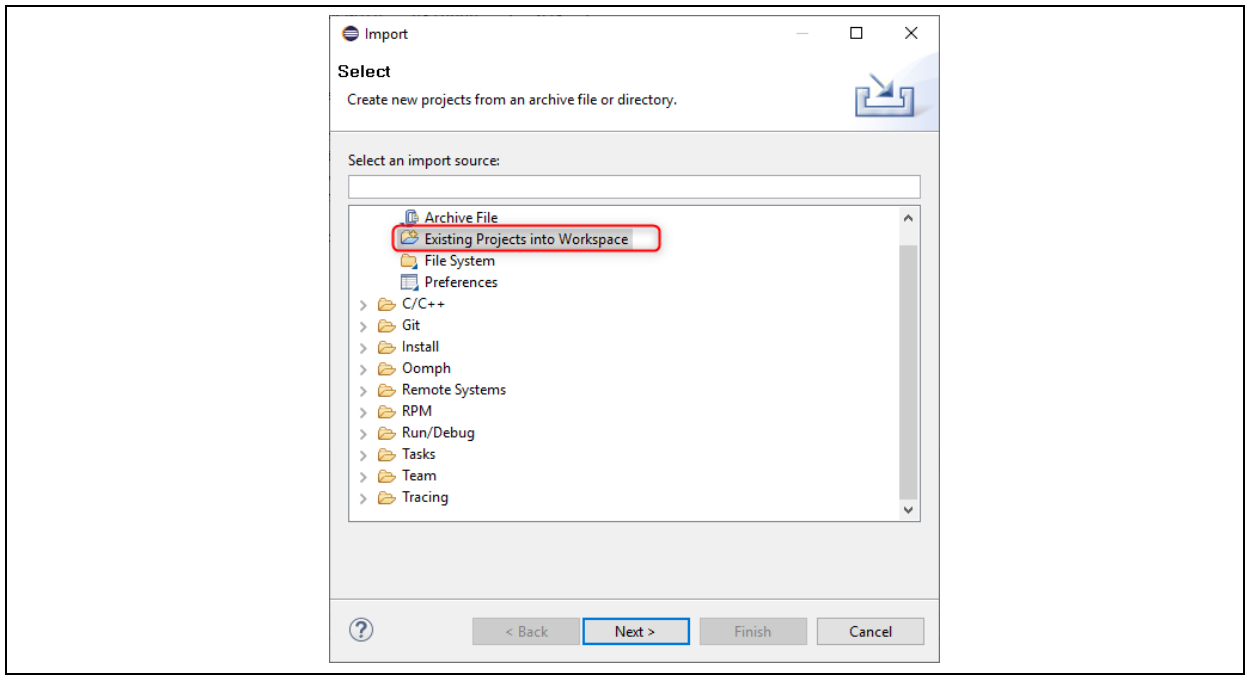

Figure 4-19 NuEclipse Import Existing Project

The last step is to select the project file, and then click “**Finish**” button.

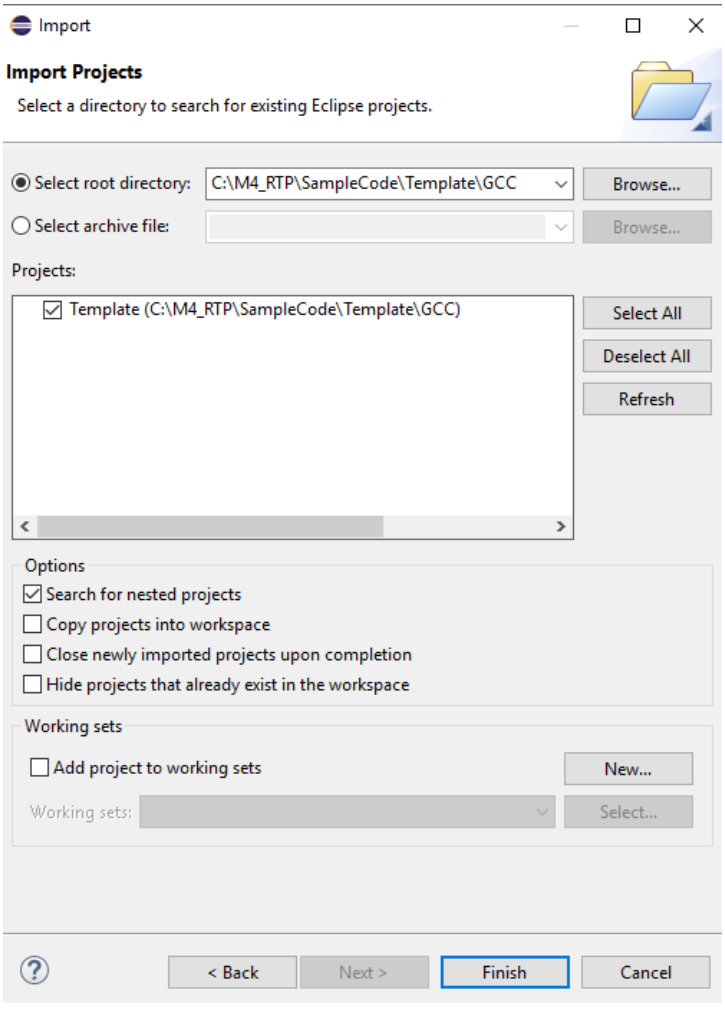

Figure 4-20 NuEclipse Select Project

To build NuEclipse project, click the Build icon as shown in Figure 4-21 or use the Ctrl + B hotkey.

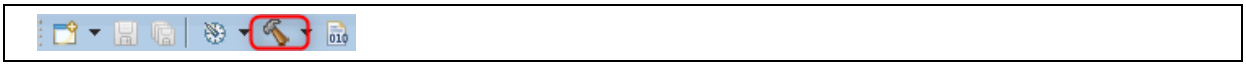

Figure 4-21 Build NuEclipse Project

Click “**Debug Configurations**” as shown in Figure 4-22.

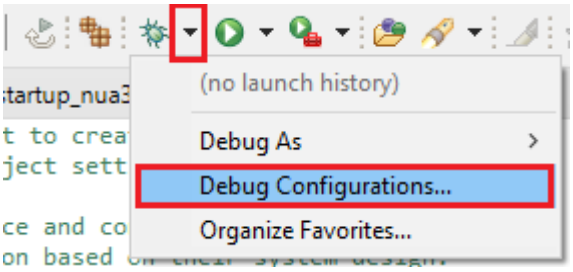

Figure 4-22 Debug Configuration

Double-click on the “**GDB Nuvoton Nu-Link Debugging**” group. The Nuvoton Nu-Link debug configuration appears on the right-hand side.

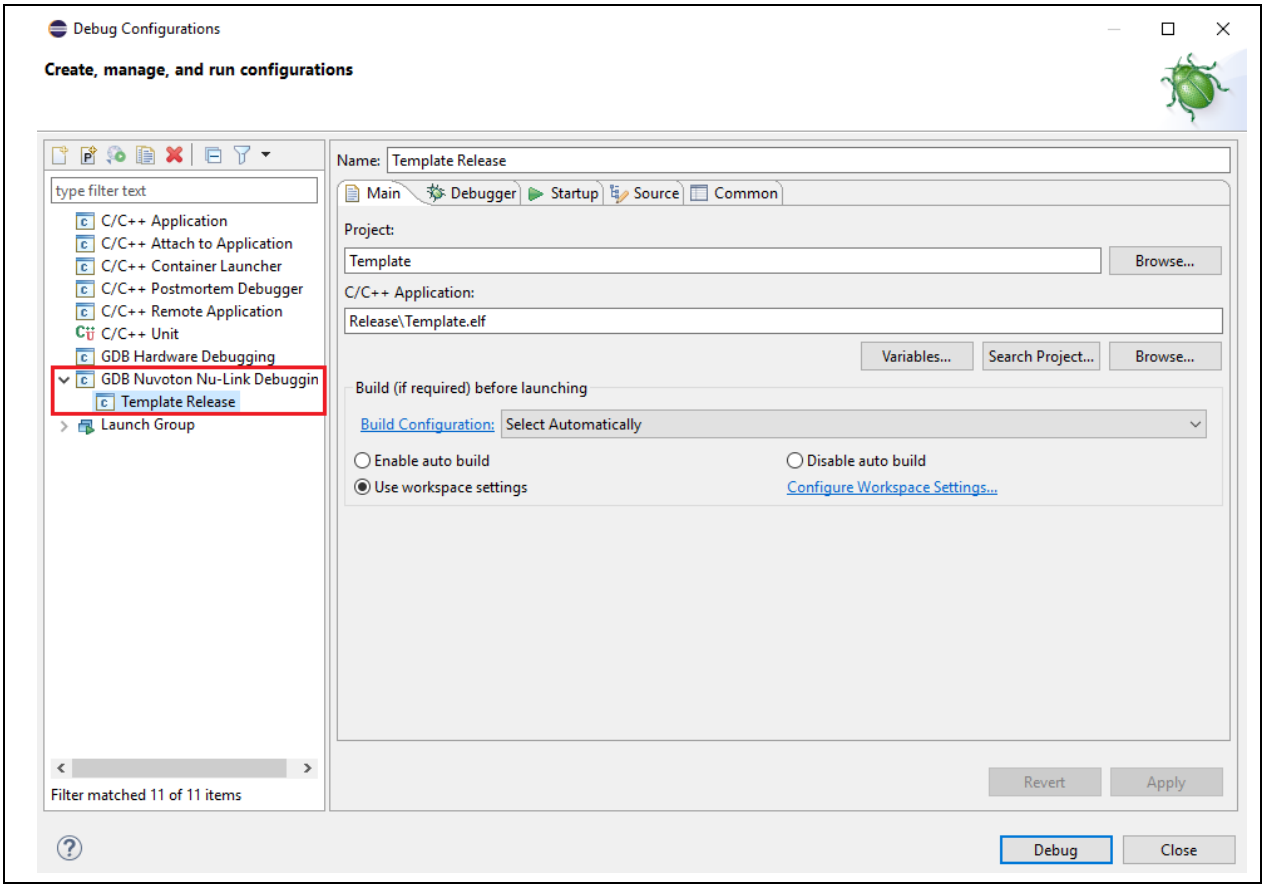

Figure 4-23 GDB Nuvoton Nu-Link Debugging

The value of GDB client port is set to 3334.

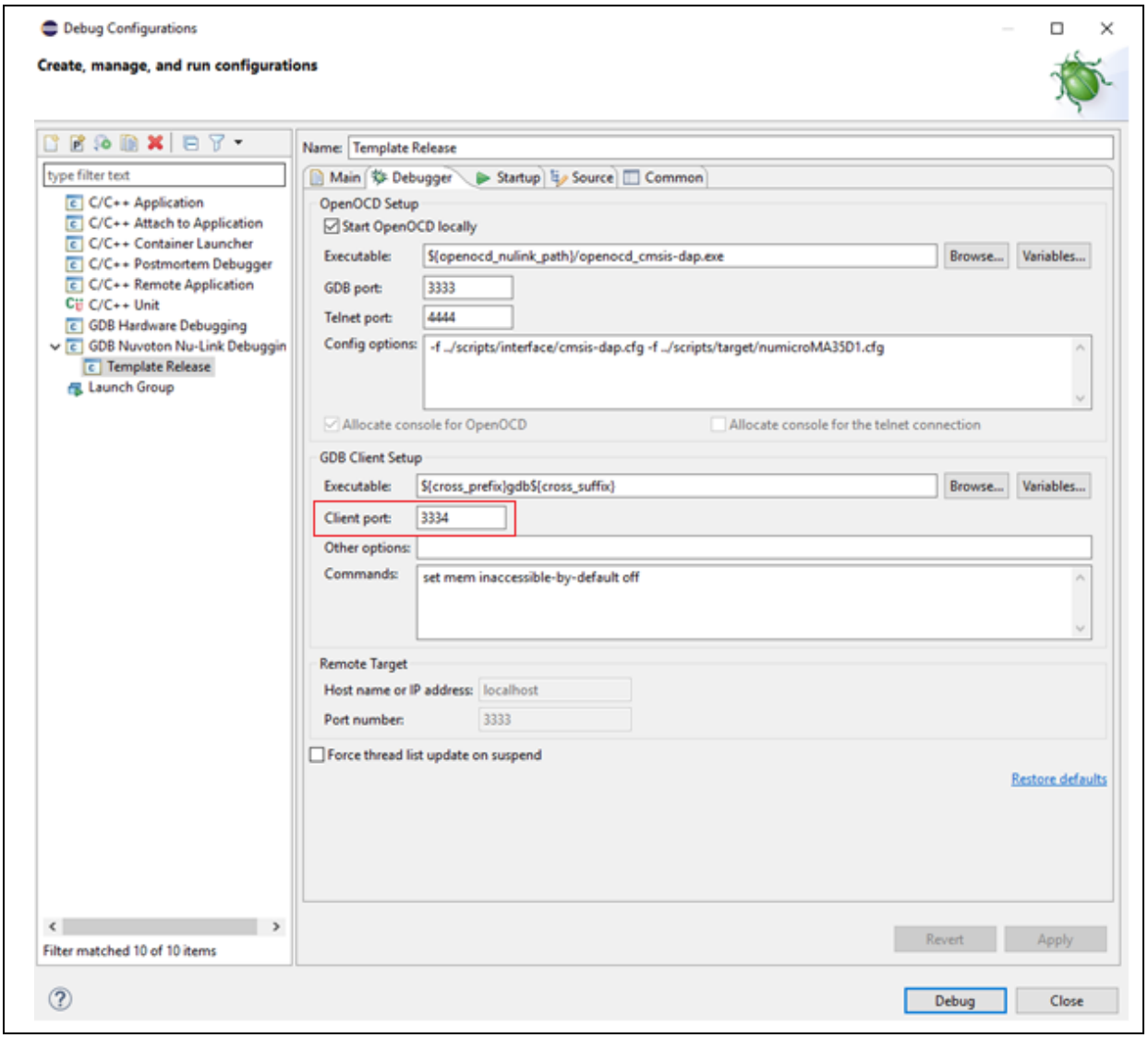

Figure 4-24 GDB Client Port

Check the startup debug setting shown in Figure 4-25, and then click "**Debug**" button. NuEclipse will load the built image and enter debug mode.

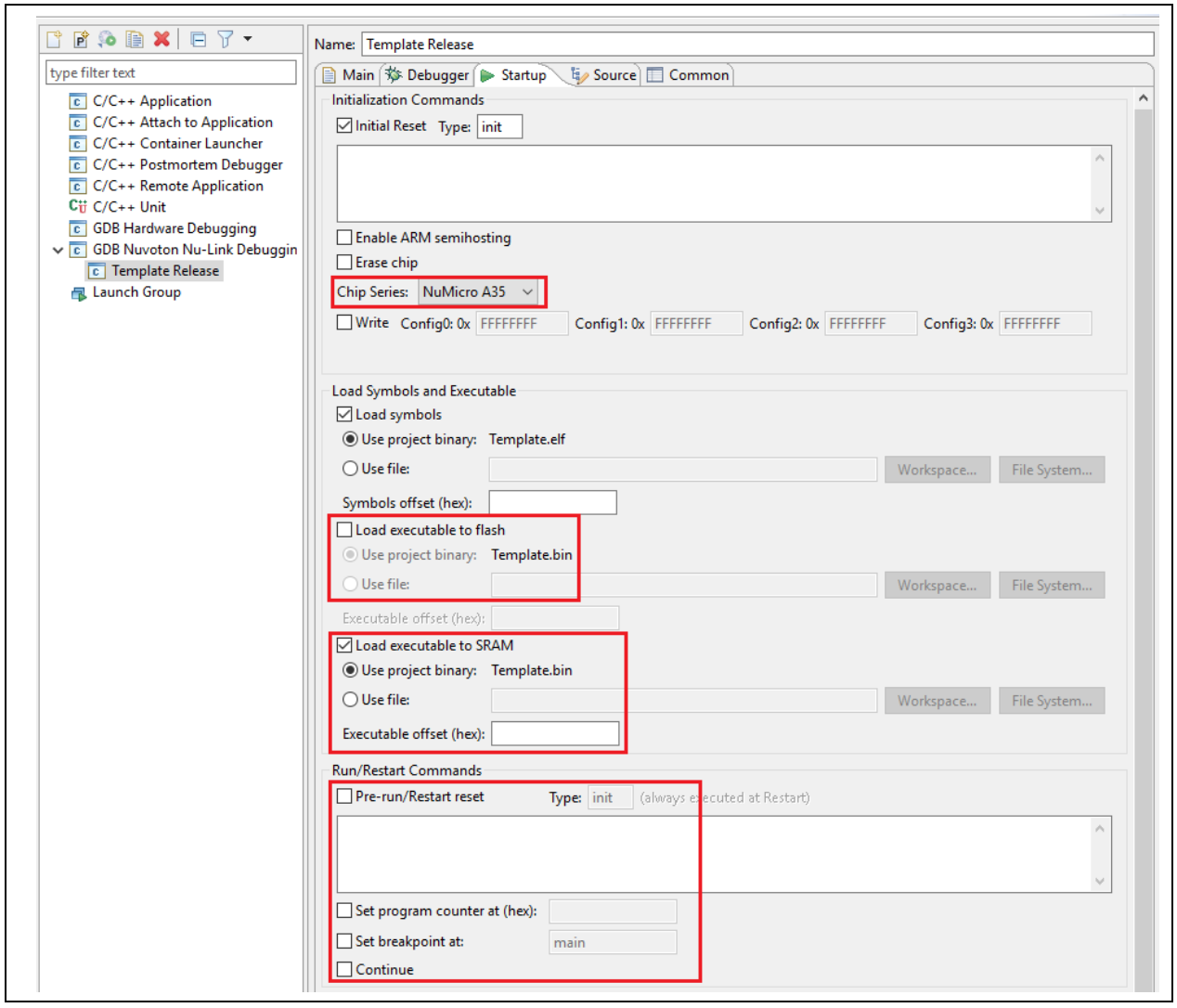

Figure 4-25 Startup Debug Setting

Revision History

Date	Revision	Description
2022.09.19	1.00	Initial version.

Important Notice

Nuvoton Products are neither intended nor warranted for usage in systems or equipment, any malfunction or failure of which may cause loss of human life, bodily injury or severe property damage. Such applications are deemed, "Insecure Usage".

Insecure usage includes, but is not limited to: equipment for surgical implementation, atomic energy control instruments, airplane or spaceship instruments, the control or operation of dynamic, brake or safety systems designed for vehicular use, traffic signal instruments, all types of safety devices, and other applications intended to support or sustain life.

All Insecure Usage shall be made at customer's risk, and in the event that third parties lay claims to Nuvoton as a result of customer's Insecure Usage, customer shall indemnify the damages and liabilities thus incurred by Nuvoton.