AN14597

RT Series Memory Relocation in Zephyr

Rev. 2.0 — 21 July 2025 **Application note**

Document information

Information	Content
Keywords	AN14597, i.MX RT, i.MX RT Crossover MCUs, Zephyr, Memory, Memory Management, Memory Relocation
Abstract	This document describes methods for managing memory within Zephyr on RT Series. These methods are similar for the other Zephyr enabled devices.

1 Introduction

Memory relocation is necessary when there are application constraints regarding memory placement. This is important for flashless i.MX RT Series devices to optimize performance. But, the flash-based MCUs can also benefit from memory relocation. By default, many Zephyr Samples and Demos are placed into external QSPI flash. Most of the applications perform well while executing from external QSPI Flash. But to achieve maximum performance, the code and data can be in internal SRAM. For more information, see AN12437.

Zephyr OS has multiple methods to relocate the memory. This document introduces methods to relocate the memory using an i.MX RT1060 device as an example.

2 Hardware and software requirements

This section provides the required hardware and software information.

2.1 Hardware

In this application note, an i.MX RT1060 evaluation kit is used. But, the steps are closely aligned to the other i.MX RT Crossover MCUs with similar memory regions. For more information, see Zephyr Supported Devices.

2.2 Software

The required list of softwares is given below:

- MCUXpresso for Visual Studio Code
- Zephyr Revision v4.0.0 - Zephyr Lab Installation and Preparation
- Zephyr Lab RT1060 Hello World

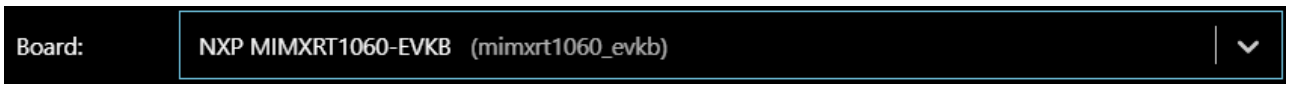


Figure 1. i.MX RT1060 evaluation kit board

3 Relocating all code and data

This section provides a method of instructions to relocate all the memory.

If relocation of all the code/data to a particular memory region is required, perform the simplest form of relocation in Zephyr as given below:

- To relocate the code to a different region, change the chosen `flash` node.
- To relocate the data to a different region, change the chosen `sram` node.

3.1 Limitations

Some memory regions must be configured prior to relocation. In this context, the SDRAM by the ROM bootloader, or reconfiguring the FlexRAM.

When performing the memory configuration, consider the following cases:

- In the case of SDRAM, Zephyr uses the DCD (in flash) and the boot ROM to configure SDRAM.
- If not booting from the flash, SDRAM is not initialized. By default, Zephyr places the data for RT1060 into the SDRAM.

For more information, see SDRAM Examples on RT1060 and MCUXpresso SDK.

3.2 Relocating to TCM

The following steps explain how to perform relocation of all code (`.text`) and read-only data to ITCM. In addition, the following steps demonstrate the relocation of all `.data` and `.bss` to DTCM:

1. Create a folder called `boards` within the *hello_world* folder.
   Note: *Some Zephyr samples already have this folder.*

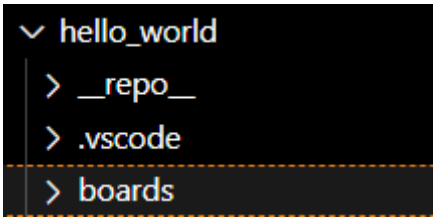


Figure 2. Boards folder in VS code explorer

2. Create an `overlay` file for the chosen board within the *boards* folder with a name that ending in `overlay`.
   An `overlay` file modifies the device tree for that application. For specific information, see setting devicetree overlays.
   At the time of writing this document, the overlay for MIMXRT1060-EVKB must be named `mimxrt1060_evkb.overlay` (Zephyr 4.0).
   Note: *After Zephyr 4.0, this filename is changed to* `mimxrt1060_evk_mimxrt1062_qspi_B.overlay`.

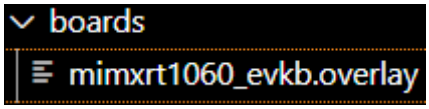


Figure 3. Overlay file within boards folder

3. Within the **.overlay* file, place the following code:

```
/{
        chosen {
            zephyr,flash = &itcm;
            zephyr,sram  = &dtcm;
        };
};
```

   The naming of `zephyr,flash` and `zephyr,sram` can be confusing.
   - `zephyr,flash` points to the node the linker uses for all the code (`.text`) and read-only data sections. Typically, this points to physical flash memory, but it can be in memory that is not flash.
   - `zephyr,sram` points to the node the linker uses for all the `.data` and `.bss` sections. This must be in RAM, but it is not required to be in SRAM.
4. Add the configuration to the configuration file provided in Step 5 and Step 6. It enables the i.MX RT device to load from the flash memory to RAM on boot. This step allows the application to persist in nonvolatile memory and run again after a power-on reset.
   A bootloader executes from the ROM and manages the boot process.
   The RAM loader feature configures the bootloader to copy the application image from the flash memory to RAM and then jumps to RAM for execution.
5. Create a configuration file for the chosen board within the *boards* folder with a name ending in `.conf`.
   For more information about configuration, see Setting Kconfig configuration values.
   For MIMXRT1060-EVKB, place the configuration file (*mimxrt1060_evkb.conf*) in the *boards* folder.
   Note: *After Zephyr 4.0, this filename is changed to mimxrt1060_evk_mimxrt1062_qspi_B.conf.*

Figure 4. Configuration file

6. In the *mimxrt1060_evkb.conf* file, add the following configuration to enable the RAMLoader:
   `CONFIG_NXP_FLEXSPI_ROM_RAMLOADER=y`
7. To perform a pristine build of the `hello_world` sample, perform the following steps:
   a. Navigate to **MCUXpresso Extension**.
   b. Under the **projects** tab, click the right arrow of the project. For example in this instance, click right arrow at `hellow_world Zephyr 4.0.0`
   c. Click to select **Pristine Build/Rebuild Selected**.

Figure 5. Hello world sample

To clean the project before building (instead of doing a pristine build), perform step 38 of the Zephyr Kconfig Lab.

The successful build displays a similar log:

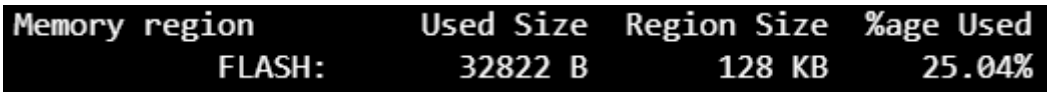


Figure 6. Successful build log

8. In addition to checking the region usage percentage, it is valuable to see where the memory regions are resolved in the `*.map` file (*zephyr.map*).
   The `*.map` file is created in the generated *build* folder, which defaults to the *hello_world/build/zephyr* directory.

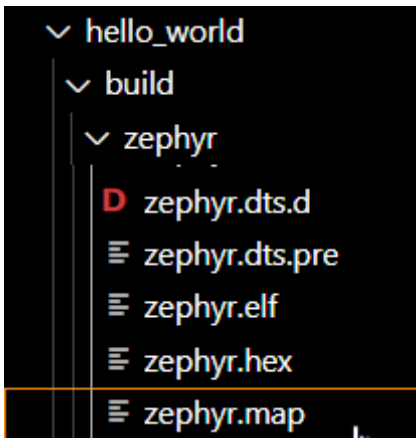


Figure 7. Memory regions in Zephyr map file

In the i.MX RT1060 Processor Reference Manual, you can find the System memory map.

Start Address	End Address	Size	Description
0000_0000	0007_FFFF	512KB	ITCM
2000_0000	2007_FFFF	512KB	DTCM

Figure 8. System memory map

The text and rodata are relocated to ITCM:

- To view that the start address for text is within the ITCM, search `__text_region_start` in the map file (*zephyr.map*).

Figure 9. Start address of text

- To view that the rodata start address is within ITCM, search the map file (*zephyr.map*) for `__rodata_region_start`.

Figure 10. Start address of rodata

The data and bss start addresses are relocated to DTCM:

- To view that the data start addresses are relocated to DTCM, search the map file (*zephyr.map*) for `__data_region_start`.

Figure 11. Start address of data

- To view that the bss start addresses are relocated to DTCM, search the map file (*zephyr.map*) for `__bss_start`.

Figure 12. Start address of bss

It is evident that the relocation is successful. If desired, you can search the end addresses of text, rodata, data, and bss in the *zephyr.map* file.

9. To flash and debug the device, perform the following steps:
   a. Navigate to VSCode MCUXpresso Extension.
   b. Click the debug button as shown in Figure 13.

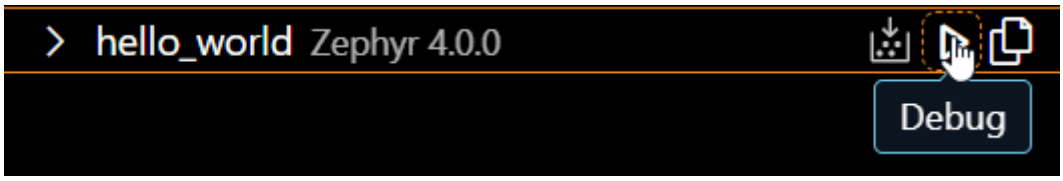


Figure 13. Debugging the device

3.3 Relocating to SDRAM

As mentioned in Section 3.1, the SDRAM requires configuration for its usage. Typically, the ROM bootloader initializes this memory during reset operation, but it requires the DCD programmed to the XIP header code in flash.

This section provides the options for using the SDRAM. It also explains how to split the existing sdram0 node, create a partition for code and data within SDRAM, and relocate all code and data to SDRAM.

3.3.1 Options for SDRAM Configuration

In Section 3.2, the configuration placed data into DTCM instead of SDRAM (default). By relocating all code to ITCM and all data to DTCM, SDRAM is not used at all. When SDRAM is used in the application, configure it first using an option given below:

- Using the RAMLoader is a viable option for configuring the SDRAM because the DCDs are stored in flash. This option provides an additional functionality of allowing the image to persist after reset. Because of this, code and data can be placed in SDRAM.
- When debugging an application in SDRAM, another option is to program a project containing the DCD into flash. As far as the DCD data is concerned, this option is not different from using the RAMLoader option. In this method, an image is stored in flash prior to programming the `hello_world` sample to RAM using the debugger. Most Zephyr samples contain this data. The `hello_world` sample contains this data, but to avoid confusion caused by having the same application running out of the flash and RAM, it is advisable to use a different sample in the flash.

An example is the blinky sample. The DCD data is found in the dcd.c file.

If using Segger Probes, an additional debugger option **.jlinkscript* file is used to initialize the SDRAM by directly writing to the hardware registers. For example, evkbmimxrt1060_sdram_init.jlinkscript.

This script is used by the debugger to initialize the SDRAM, and then the debugger downloads the code to SDRAM. For more information, see the GitHub page: SDRAM Examples on RT1060 and MCUXpresso SDK.

3.3.2 Relocating all code and data

To relocate all the code and data, perform the following steps:

1. This guide uses the RAMloader option. Add the following configuration, which enables the RAMLoader to *boards\mimxrt1060_evkb.conf*:

```
CONFIG_NXP_FLEXSPI_ROM_RAMLOADER=y
```

2. Split the SDRAM into code and data section for the linker.
   To avoid overlap of code and data, divide SDRAM into two partitions `sdram_code`, and `sdram_data` and then add the following code in the *mimxrt1060_evkb.overlay* file:

```
&sdram0 {
        ranges = <0x00000000 0x80000000 DT_SIZE_M(32) >;
        #address-cells = < 0x1 >;
        #size-cells = < 0x1 >;
        /* Divide SDRAM into two partitions for Code and Data */
        sdram_code: memory@0 {
                device_type = "memory";
                reg = <0x00000000 DT_SIZE_M(16)>;

        };
        sdram_data: memory@1000000 {
                device_type = "memory";
                reg = <0x01000000 DT_SIZE_M(16)>;
        };
};
```

3. Place the code and data in the `sdram_code` and `sdram_data` respectively by changing the `zephyr,flash` and `zephyr,sram` nodes:

```
         chosen {
                zephyr,flash = &sdram_code;
                zephyr,sram = &sdram_data;
        };
```

4. After steps 2 and 3, the entirety of the `*.overlay` file is as follows:

```
/{
        chosen {
                zephyr,flash = &sdram_code;
                zephyr,sram = &sdram_data;
```

```
        };


};
&sdram0 {
        ranges = <0x00000000 0x80000000 DT_SIZE_M(32) >;
        #address-cells = < 0x1 >;
        #size-cells = < 0x1 >;
        /* Divide SDRAM into two partitions for Code and Data */
        sdram_code: memory@0 {
                device_type = "memory";
                reg = <0x00000000 DT_SIZE_M(16)>;
        };
        sdram_data: memory@1000000 {
                device_type = "memory";
                reg = <0x01000000 DT_SIZE_M(16)>;
        };
};
```

5. After a pristine build, both the FLASH (code, rodata), and RAM (data, bss) regions show a 16MB size with a small amount of usage:

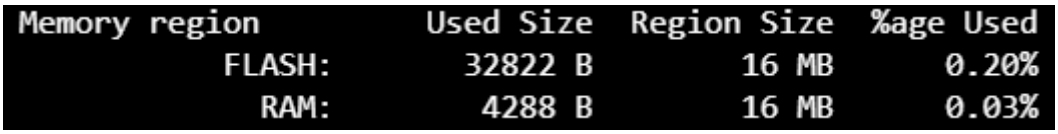


Figure 14. FLASH and RAM regions

6. For confirmation about the placement of code, rodata, data, and bss, look into the **.map* file (*build/zephyr/zephyr.map*).
   - The text region starts and ends in the partition allotted for `sdram_code` as shown below:

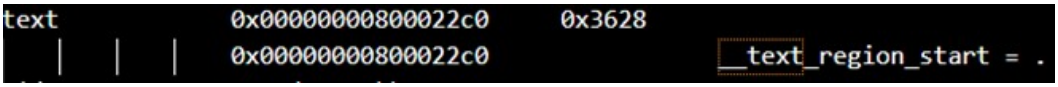


Figure 15. Text start region

Figure 16. Text end region

   - The data region starts and ends in the partition allotted for `sdram_data` as shown below:

Figure 17. Data start region

Figure 18. Data end region

Therefore, it provides adequate information that relocation is successful.

4 Zephyr code relocation usage

Compared to the previous method of code/data relocation explained in Section 3, the Zephyr code relocation feature allows greater flexibility at the cost of adding other complexity. For more information, see Zephyr Code and Data Relocation.

Zephyr code relocation feature is useful for relocating specific files or libraries to the desired memory region. It is also useful for relocating the text, rodata, data, and bss sections from specific files or libraries. For more information on this feature, see Zephyr project.

The following steps start from the unmodified Hello World Sample. They demonstrate using the Zephyr code relocation feature to relocate the files and libraries. Also, they demonstrate relocating specific memory sections from the files and libraries.

4.1 Limitations

The usage of the Zephyr feature in relocating the code and data has the following limitations:

- Zephyr startup code performs the relocation of code and data. Therefore, any code and data used before the Zephyr relocation process cannot be relocated.
- Some code in the kernel files and libraries can be relocated. However, some code cannot be relocated since it executes before relocation.
- If relocating to memory has to be configured, then it must be configured before relocation. For example, SDRAM by the ROM bootloader, or reconfiguring FlexRAM (Section 6).
- Code relocation can only be used to move memory to regions which have been defined for the linker. For information on adding the custom memory regions, refer Section 4.4.

4.2 Enabling configuration file to use code and data relocation feature

To use the code and data relocation feature, perform the following steps:

1. Enable `CONFIG_CODE_DATA_RELOCATION` in the configuration file *prj.conf* file.
2. Edit the *prj.conf* file and add `CONFIG_CODE_DATA_RELOCATION=y`.

Figure 19. Configuration file

Note: *After enabling this feature, add some relocation to the CMakeLists.txt file. Otherwise, an error occurs when trying to build.*

```
Disable CONFIG_CODE_DATA_RELOCATION if no file needs relocation
ninja: build stopped: subcommand failed.
build finished with error(s).
```

Figure 20. Build error when no relocation is detected

4.3 Relocating files

This section demonstrates the relocation of files to ITCM.

4.3.1 Relocating code and data files

To relocate the code and data from files, perform the following steps:

1. To relocate the *main.c* file (text, rodata, data, and bss) to ITCM, add the following line to *CMakeLists.txt:*:
   `zephyr_code_relocate(FILES src/main.c LOCATION ITCM)`

Figure 21. Relocating files

When added to *CMakeLists.txt*, it must appear as given below:

```
# SPDX-License-Identifier: Apache-2.0
cmake_minimum_required(VERSION 3.20.0)
find_package(Zephyr REQUIRED HINTS $ENV{ZEPHYR_BASE})
project(hello_world)
# add this line to relocate main.c to ITCM
zephyr_code_relocate(FILES src/main.c LOCATION ITCM)
target_sources(app PRIVATE src/main.c)
```

2. After completing a pristine build, the ITCM usage must increase from 0.00% (from the default `Hello_World Sample`) to ~0.10%.

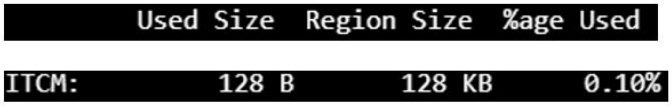


Figure 22. ITCM usage

The main.c file has been successfully relocated to ITCM.

4.3.2 Relocate selected memory section from files

This option specifies the section of memory to be relocated in addition to the location.

To add the text section from the chosen file, perform the following steps:

1. To relocate only the text section from the chosen file, add the following line in the *CMakeLists.txt* file:

```
zephyr_code_relocate(FILES src/main.c LOCATION ITCM_TEXT)
```

2. After a pristine build, usage of ITCM must change to ~0.02%:

Figure 23. ITCM usage in build output

4.4 Relocating libraries

This section demonstrates the relocation of a library to a specific memory region. It also explains how to move a particular type of memory from a library to a specific memory region.

4.4.1 Relocating to a specific memory region

To relocate a library to a specific memory region, perform the following steps:

1. Adding the following code in the *CMakeLists.txt* file relocates the serial drivers to ITCM:
   `zephyr_code_relocate(LIBRARY drivers__serial LOCATION ITCM)`
2. Find the driver name. The name of the driver can be found in the *zephyr.map* file. The driver libraries within the file, **.map* are commonly named as `libdrivers__x.a`.
   Where, x is the name of the driver.
   In this context, the library name is `libdrivers__serial.a`. To find the library name, open the *zephyr.map* file and search for `serial`.
   An alternate method to find the library name is by searching the name of a source file in the **.map* file. For example, the line below shows the source file *uart_mcux_lpuart.c* is part of the `drivers__serial` library.

```
.text.mcux_lpuart_config_get
   |   |   |    0x000000006000538e       0x10 zephyr/drivers/serial/libdrivers__serial.a(uart_mcux_lpuart.c.obj)
```

Figure 24. Relocating to a specific memory region

Prior to relocation, the*.map* file shows the text elements of this driver library in the flash.

3. In the *CMakeLists.txt* file, the `CMake` driver name is used. After conversion, the `CMake` driver name is changed to `drivers__serial`.
   The *CMakeLists.txt* file must appear as given below:

```
# SPDX-License-Identifier: Apache-2.0

cmake_minimum_required(VERSION 3.20.0)

find_package(Zephyr REQUIRED HINTS $ENV{ZEPHYR_BASE})
project(hello_world)

#add this line to relocate main.c TEXT to ITCM
zephyr_code_relocate(FILES src/main.c LOCATION ITCM_TEXT)

#add this line to relocate serial drivers to ITCM
zephyr_code_relocate(LIBRARY drivers__serial LOCATION ITCM)

target_sources(app PRIVATE src/main.c)
```

4. Now, the ITCM usage must be ~0.79%, as we have relocated TEXT from *main.c* and the `drivers__serial` library to this region.

```
ITCM:        1032 B       128 KB      0.79%
```

Figure 25. ITCM usage

4.4.2 Relocating a specific memory section of the library

For performance optimization, it is best to place code and rodata into ITCM, while placing data and bss to DTCM.

To relocate the code, perform the following steps:

1. Change the following line:

```
zephyr_code_relocate(LIBRARY drivers__serial LOCATION ITCM)
```

   By changing the above line to the line given below moves only the text and rodata to ITCM:

```
zephyr_code_relocate(LIBRARY drivers__serial LOCATION ITCM_TEXT_RODATA)
```

2. To move DATA and BSS to DTCM, add the following line:

```
zephyr_code_relocate(LIBRARY drivers__serial LOCATION DTCM_DATA_BSS)
```

   With these two lines added to *CMakeLists.txt*, the code and rodata from the serial drivers library are placed into ITCM. Meanwhile, the data and bss from the serial drivers library are placed into DTCM.
3. With this change. ITCM usage will be at ~0.78%, while DTCM usage must change from 0.00% to ~0.01% (after rebuilding).

```
ITCM:          1 KB        128 KB      0.78%
DTCM:          8 B         128 KB      0.01%
```

Figure 26. ITCM and DTCM usage

4. Confirm relocation. You can take steps to ensure that the relocation is successful. A successful relocation can be confirmed by ensuring that the text from *main.c*, the text and rodata from the serial driver library, and the data and bss from the serial driver library are relocated.
   In the *zephyr.map* file, search for `.itcm_text_reloc`. This section of the MAP file displays all the text that is relocated to ITCM.
   Notably, the text from *main.c* is relocated as shown below:

```
.itcm_text_reloc
                0x0000000000000000      0x2f0 load address 0x00000000600022c0
                0x0000000000000000                . = ALIGN (0x4)
 *main.c.obj(SORT_BY_ALIGNMENT(.rel.text.main))
 *main.c.obj(SORT_BY_ALIGNMENT(.text.main))
 .text.main     0x0000000000000000       0x18 app/libapp.a(main.c.obj)
                0x0000000000000000                main
```

Figure 27. relocation confirmation

   In addition, the text from the serial driver library is placed as shown below:

```
.text.mcux_lpuart_config_get
                0x0000000000000018       0x10 zephyr/drivers/serial/libdrivers__serial.a(uart_mcux_lpuart.c.obj)
*uart_mcux_lpuart.c.obj(SORT_BY_ALIGNMENT(.text.mcux_lpuart_configure))
.text.mcux_lpuart_configure
                0x0000000000000028       0x30 zephyr/drivers/serial/libdrivers__serial.a(uart_mcux_lpuart.c.obj)
*uart_mcux_lpuart.c.obj(SORT_BY_ALIGNMENT(.text.mcux_lpuart_configure_init))
.text.mcux_lpuart_configure_init
                0x0000000000000058      0x144 zephyr/drivers/serial/libdrivers__serial.a(uart_mcux_lpuart.c.obj)
```

Figure 28. Text relocation

   To confirm whether the relocation is successful, search the MAP file as given below:
   - `.itcm_rodata_reloc`: rodata relocated to itcm
   - `dtcm_bss_reloc`: bss relocated to dtcm
   - `dtcm_bss_reloc`: bss relocated to dtcm

   Note: *This driver library does not have anything in its data section to relocate.*

4.5 Relocating to custom memory region

This section explains how to create a custom memory region. This is useful if the default linker file does not contain the memory region we must relocate using the code relocation feature. The example in this section relocates some code and data from files and libraries to SDRAM. This involves modifying the **.overlay* file to split SDRAM into partitions for code and data. It also involves adding other properties to the memory nodes to enable a custom linker section.

To relocate files and libraries to a custom memory region, perform the following steps:

1. Partition SDRAM for Code and Data
   This step is like Section 3.3.2, and the following steps start from the unmodified `Hello World Sample`. `zephyr,flash` is not relocated, so most of the application remains in flash. `zephyr,sram` is changed, as `sdram0` is the default for `zephyr,sram`. Here, `zephyr,sram` is changed to `dtcm`. Therefore, any data that is not relocated using Code relocation exists in `dtcm`.

Partition SDRAM by changing the **.overlay* file (`mimxrt1060_evkb.overlay`) as given below:

```
&sdram0 {
        ranges = <0x00000000 0x80000000 DT_SIZE_M(32) >;
        #address-cells = < 0x1 >;
        #size-cells = < 0x1 >;
        /* Divide SDRAM into two partitions for Code and Data */
        sdram_code: memory@0 {
                device_type = "memory";
                reg = <0x00000000 DT_SIZE_M(16)>;

        };
        sdram_data: memory@1000000 {
                device_type = "memory";
                reg = <0x01000000 DT_SIZE_M(16)>;
        };
};
```

2. Add other properties to both the subnodes (`sdram_code`, and `sdram_data`) as given below:

```
compatible = "zephyr,memory-region";
zephyr,memory-region = "CUSTOMNAME";
```

The compatible property is used to mark the node as a memory region. The name given to `zephyr,memory-region` is used to generate a linker region of the given name. For this document, `SDRAMCODE`, and `SDRAMDATA` are used.
The full **.overlay* file displays as given below:

```
/{
        chosen {
                zephyr,sram = &dtcm;
        };


};
&sdram0 {
        ranges = <0x00000000 0x80000000 DT_SIZE_M(32) >;
        #address-cells = < 0x1 >;
        #size-cells = < 0x1 >;
        /* Divide SDRAM into two partitions for Code and Data */
        sdram_code: memory@0 {
                device_type = "memory";
                reg = <0x00000000 DT_SIZE_M(16)>;
                compatible = "zephyr,memory-region";
                zephyr,memory-region = "SDRAMCODE";
        };
        sdram_data: memory@1000000 {
                device_type = "memory";
                reg = <0x01000000 DT_SIZE_M(16)>;
                compatible = "zephyr,memory-region";
                zephyr,memory-region = "SDRAMDATA";
        };
};
```

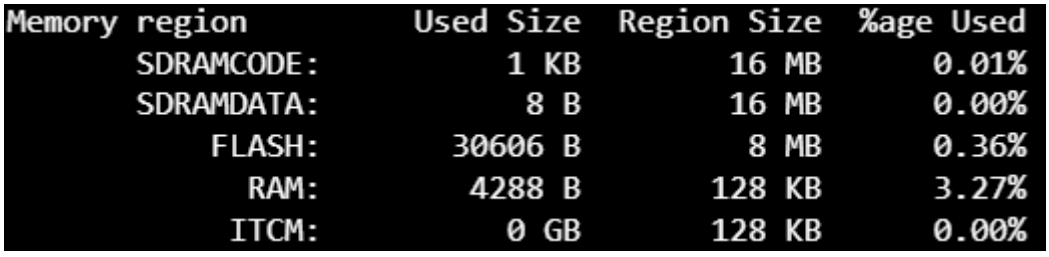
3. Enable CONFIG_CODE_DATA_RELOCATION in the *prj.conf* file.
   Edit the *prj.conf* file and add `CONFIG_CODE_DATA_RELOCATION=y`.

Figure 29. Configuration file

4. Use Code Relocation APIs to Move Objects Into SDRAM.
   This step uses the previously covered Code Relocation APIs. We can use them to relocate to the newly created memory regions in the *CMakeLists.txt* file.

```
# SPDX-License-Identifier: Apache-2.0

cmake_minimum_required(VERSION 3.20.0)

find_package(Zephyr REQUIRED HINTS $ENV{ZEPHYR_BASE})
project(hello_world)
#add this line to relocate main.c text and rodata to SDRAM_CODE
zephyr_code_relocate(FILES src/main.c LOCATION SDRAMCODE_TEXT_RODATA)

#add this line to relocate main.c data and bss to SDRAM_DATA
zephyr_code_relocate(FILES src/main.c LOCATION SDRAMDATA_DATA_BSS)

#add this line to relocate serial drivers text and rodata to SDRAM_CODE
zephyr_code_relocate(LIBRARY drivers__serial LOCATION SDRAMCODE_TEXT_RODATA)

#add this line to relocate data and bss to SDRAM_DATA
zephyr_code_relocate(LIBRARY drivers__serial LOCATION SDRAMDATA_DATA_BSS)

target_sources(app PRIVATE src/main.c)
```

5. Confirm relocation.
   The output from the build looks like Figure 30.



Memory region	Used Size	Region Size	%age Used
SDRAMCODE:	1 KB	16 MB	0.01%
SDRAMDATA:	8 B	16 MB	0.00%
FLASH:	30606 B	8 MB	0.36%
RAM:	4288 B	128 KB	3.27%
ITCM:	0 GB	128 KB	0.00%

Figure 30. Build output

DTCM serves as the RAM region, and it is the default memory region for data and bss that is not relocated.
Note: *The SDRAMDATA region contains data and bss from main.c and the serial library.*
The Flash region is an external flash and it is the default memory region for text and rodata that is not relocated.
Note: *The SDRAMCODE region contains text and rodata from main.c and the serial library.*
Navigate to *build/zephyr/zephyr.map*.
To confirm that the text from *main.c* has been relocated to `SDRAMCODE`, search for `sdramcode_text_reloc`:

- The `.sdramcode_text_reloc` section contains text that has been relocated to `SDRAMCODE`.
- The `.sdramcode_rodata_reloc` section contains rodata that has been relocated to `SDRAMCODE`.
- The `.sdramdata_bss_reloc` section contains bss that has been relocated to `SDRAMDATA`.
- The `.sdramdata_data_reloc` section is not generated because there is no data available for relocation.

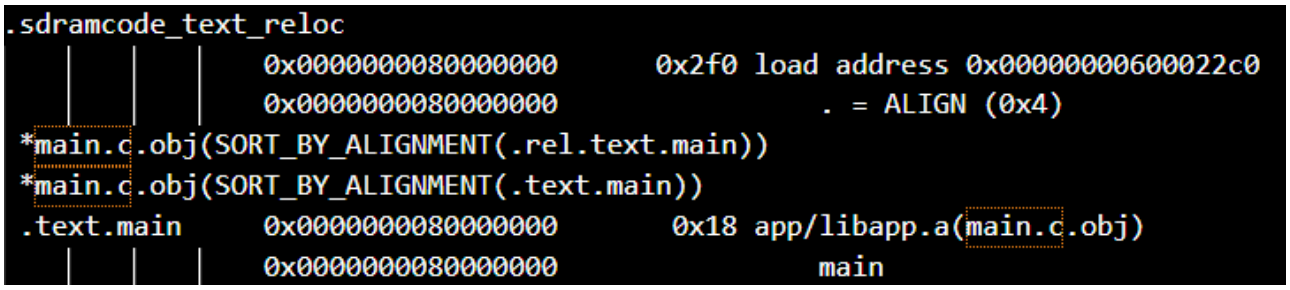


Figure 31. Relocation confirmation

4.6 Relocating Using KConfigs

Some Zephyr samples, such as the Zperf Sample, use a combination of Kconfigs and Code Relocation APIs to achieve memory relocation.

In the case of the Zperf Sample, Kconfigs are placed in the application Kconfig file, which selects `CODE_DATA_RELOCATION` if `NET_SAMPLE_CODE_RELOCATE` is enabled.

```
config NET_SAMPLE_CODE_RELOCATE
    bool "Relocate networking code into RAM"
    select CODE_DATA_RELOCATION
    help
      Relocate networking code into RAM when running the zperf
      sample. Can improve performance on platforms with fast code
      RAM.
```

If `NET_SAMPLE_CODE_RELOCATE` is enabled in this sample, it also enables usage of `NET_SAMPLE_CODE_RAM_NAME` to specify the region to relocate to.

```
if NET_SAMPLE_CODE_RELOCATE

config NET_SAMPLE_CODE_RAM_NAME
    string "Networking code RAM location"
    default "RAM"
    help
      Region to relocate networking code to

endif # NET_SAMPLE_CODE_RELOCATE
```

Both configs (`NET_SAMPLE_CODE_RELOCATE`, and `NET_SAMPLE_CODE_RAM_NAME`) are placed into the board specific .config file for the application. (Zephyr v4.0.0 included the board-specific file mimxrt1060_evk.conf).

```
CONFIG_NET_SAMPLE_CODE_RELOCATE=y
CONFIG_NET_SAMPLE_CODE_RAM_NAME="ITCM"
```

Within the, these configurations are used with code relocation APIs to relocate specific code to ITCM:

```
if (CONCMakeLists.txt fileFIG_NET_SAMPLE_CODE_RELOCATE)
  # Relocate key networking stack components and L2 layer to RAM
  zephyr_code_relocate(LIBRARY subsys__net__ip
    LOCATION "${CONFIG_NET_SAMPLE_CODE_RAM_NAME}_TEXT" NOKEEP)
  zephyr_code_relocate(LIBRARY subsys__net
    LOCATION "${CONFIG_NET_SAMPLE_CODE_RAM_NAME}_TEXT" NOKEEP)
if (CONFIG_NET_L2_ETHERNET)
  zephyr_code_relocate(LIBRARY drivers__ethernet
```

```
    LOCATION "${CONFIG_NET_SAMPLE_CODE_RAM_NAME}_TEXT" NOKEEP)
  zephyr_code_relocate(LIBRARY subsys__net__l2__ethernet
    LOCATION "${CONFIG_NET_SAMPLE_CODE_RAM_NAME}_TEXT" NOKEEP)
endif()
endif()
```

5 Configuring The FlexRAM

This section provides information regarding using the FlexRAM within Zephyr.

For more information, see AN12077. To understand the key considerations when dynamically changing the FlexRAM configuration within the application, refer to this application note. It is not recommended to execute from any FlexRAM when changing the FlexRAM configuration.

5.1 Configuring FlexRAM using boot fuses

RT1060 has 16 possible configurations for OCRAM, DTCM, ITCM, available using OTP fuses. For more information, see Table 3 in Section 2 of AN12077.

Note: *Changes to fuses are permanent and cannot be undone. An example of this method is using fuse configuration 1 instead of the default value (0).*

	FUSE FlexRAM Configuration Value	IOMUXC_GPR_GPR17 (FLEXRAM_BANK_CFG) (binary)	Bank																OCRAM [kB]	DTCM [kB]	ITCM [kB]
	0x6D0 [19:16]		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15			
0	0b0000	01010101101011111111101001010101	O	O	O	O	D	D	I	I	I	I	D	D	O	O	O	O	256	128	128
1	0b0001	01010101011010111101001010101	O	O	O	O	D	D	I	I	D	D	O	O	O	O	O	O	320	128	64

Figure 32. FlexRAM configuration

After changing the boot fuses, the following changes must be made in the **.overlay* file:

```
&itcm {
        reg = < 0x0 DT_SIZE_K(64) >;
};
&dtcm {
        reg = < 0x20000000 DT_SIZE_K(128) >;
};
&ocram {
        reg = < 0x20280000 DT_SIZE_K(320) >;
};
&flexram {
        flexram,bank-spec = <FLEXRAM_OCRAM>,
                            <FLEXRAM_OCRAM>,
                            <FLEXRAM_OCRAM>,
                            <FLEXRAM_OCRAM>,
                            <FLEXRAM_DTCM>,
                            <FLEXRAM_DTCM>,
                            <FLEXRAM_ITCM>,
                            <FLEXRAM_ITCM>,
```

```
                                <FLEXRAM_DTCM>,
                                <FLEXRAM_DTCM>,
                                <FLEXRAM_OCRAM>,
                                <FLEXRAM_OCRAM>,
                                <FLEXRAM_OCRAM>,
                                <FLEXRAM_OCRAM>,
                                <FLEXRAM_OCRAM>,
                                <FLEXRAM_OCRAM>;
};
```

These changes update the size of the itcm, dtcm, and ocram nodes. They also update the flexram, bank-spec property of the flexram node to match the boot fuses.

Advantage of programming fuses: You can run the application entirely out of FlexRAM regions, as there is no longer a need to configure FlexRAM within the application.

Disadvantage of writing to OTP fuses: You cannot change them after programming. The result is that the changes made are not reversible.

5.2 Configuring FlexRAM Dynamically

Zephyr provides automatic configuration of the FlexRAM Dynamically based on the flexram, bank-spec property of the flexram node.

To perform Dynamic FlexRAM Configuration, make identical changes in the **.overlay* file, similar to Section 5.1.

```
&itcm {
        reg = < 0x0 DT_SIZE_K(64) >;
};
&dtcm {
        reg = < 0x20000000 DT_SIZE_K(128) >;
};
&ocram {
        reg = < 0x20280000 DT_SIZE_K(320) >;
};
&flexram {
        flexram,bank-spec = <FLEXRAM_OCRAM>,
                            <FLEXRAM_OCRAM>,
                            <FLEXRAM_OCRAM>,
                            <FLEXRAM_OCRAM>,
                            <FLEXRAM_DTCM>,
                            <FLEXRAM_DTCM>,
                            <FLEXRAM_ITCM>,
                            <FLEXRAM_ITCM>,
                            <FLEXRAM_DTCM>,
                            <FLEXRAM_DTCM>,
                            <FLEXRAM_OCRAM>,
                            <FLEXRAM_OCRAM>,
                            <FLEXRAM_OCRAM>,
                            <FLEXRAM_OCRAM>,
                            <FLEXRAM_OCRAM>,
                            <FLEXRAM_OCRAM>;
};
```

Advantage: The FlexRAM configuration does not need to match one of the 16 configurations available using fuses. Banks can be freely allocated as ITCM, DTCM, or OCRAM. This allows for greater flexibility.

Drawback: Unable to relocate the startup code (which performs the FlexRAM configuration) to FlexRAM regions. It means that the methods mentioned in Section 3 to relocate code to itcm and data to dtcm are not

recommended for usage with FlexRAM reconfiguration. If the startup code which reconfigures FlexRAM lies in a FlexRAM region, there is potential for failure. For stability, ensure that no code executes from the FlexRAM regions until completion of reconfiguration.

The recommended method to use with dynamic FlexRAM reconfiguration is Code Relocation APIs (Section 4), as this circumvents the potential issues with dynamic configuration.

A practical example of the usage of FlexRAM configuration can be found within the Facial Detect Demo. Where, the *.overlay file includes changes to make DTCM larger, and the Code Relocation APIs are used in the CMakeLists.txt file.

6 Board bring-up guidelines

This section describes board bring-up in general and not for one specific board.

In the Hello World Sample, Zephyr defaults to placing the code in the external flash, and data in SDRAM on the I.MX RT 1060 Evaluation kit.

Any other Zephyr samples which do not change the `zephyr,flash` or `zephyr,ram` node within the project *board/.overlay* file or Code Relocation API within the *CMakeLists.txt* file contains code in the external flash and data in SDRAM for this device.

When testing a new board, it helps to avoid an external memory initially. Use a debugger to load the simple code in the internal SRAM and confirm that the code executes. Then external memories can be enabled one at a time to verify each memory.

The instructions from Section 3.2 highlight the process for moving all code to ITCM and all data to DTCM. To ensure that the part boots up and runs `hello_world`, perform this exercise as an initial check. This is useful when using a custom board, which cannot have the same external flash or SDRAM.

Note: *If the MCU does not boot from flash, the Kconfig `NXP_FLEXSPI_ROM_RAMLOADER` must not be used.*

7 Conclusion

Methods for memory relocation in Zephyr provide practical solutions for managing memory in embedded systems. The framework of Zephyr for memory management allows developers to focus on building high-level applications rather than dealing with low-level memory concerns. While Zephyr is constantly growing and improving, the methods highlighted in this document have created more straightforward memory management. These methods can help create more stable systems.

8 Related documentation

Table 1 provides documentation on the related topics and the additional information you should have before setting up the hardware and software.

Table 1. Related documentation

Document	Document number	Description
Related Application notes	AN12437 (RT Series Performance Optimization)	Provides details on the memory types and performance details for each type on i.MX RT Series Devices.
	AN12077 (Using the i.MX RT FlexRAM)	Describes the flexible memory array available on the i.MX RT 4-digit crossover processors.
Zephyr documentation	Zephyr Project Landing Page	Zephyr project page

Table 1. Related documentation...*continued*

Document	Document number	Description
	Zephyr™ OS for Edge Connected Devices	Browse the NXP home page for Zephyr.
	Zephyr Code and Data Relocation	Provides information Code and Data Relocation feature of Zephyr.
	Lab Guides for MCUXpresso for VS Code	Provides importing and building Zephyr applications, debugging, devicetree, and Kconfig.

9 Glossary

Table 2 provides the acronyms used in this document and its description.

Table 2. Acronyms and description

Acronym	Abbreviation	Description
ITCM	Instruction Tightly Coupled Memory	Tightly Coupled Memory which is typically used to access critical functions, exception vector tables, and interrupt service routines.
DTCM	Data Tightly Coupled Memory	Tightly Coupled Memory which is typically used to store critical variables and frequently updated variables.
QSPI	Quad Serial Peripheral Interface	Peripheral designed to communicate with flash chips that can support this interface. The i.MX RT series can boot and execute from external QSPI flash.
DCD	Device Configuration Data	Data used for initializing and configuring peripherals on a device during the boot process.
FlexRAM	Flexible RAM	Highly configurable and flexible internal RAM memory array. This memory array contains memory banks, independently configured and accessed by different types of interfaces, such as I-TCM, D-TCM, or AXI (system). The memory bank can act as an ITCM, DTCM, or OCRAM memory. For more information, see AN12077.
OTP	One-Time Programmable Fuses	Nonvolatile memory elements that can be programmed only once and cannot be erased or reprogrammed. They permanently store critical device configuration, security parameters, and identification information.
XIP	eXecute In Place	A memory execution mode where program instructions are fetched and executed directly from non-volatile memory rather than being copied to RAM first.

Table 2. Acronyms and description*...continued*

Acronym	Abbreviation	Description
OCRAM	On-Chip RAM	General purpose memory, which is typically used for both code and data. However, it is expected to have worse performance when fetching instructions from ITCM, and accessing data compared to DTCM.

10 Note about the source code in the document

Example code shown in this document has the following copyright and BSD-3-Clause license:

Copyright 2025 NXP Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
3. Neither the name of the copyright holder nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

11 Revision history

Table 3 summarizes the revisions to this document.

Table 3. Revision history

Document ID	Release date	Description
AN14597 v.2.0	21 July 2025	Modified code snippets in Section 3.3.2 and Section 4.5
AN14597 v.1.0	27 March 2025	Initial public release

Legal information

Definitions

Draft — A draft status on a document indicates that the content is still under internal review and subject to formal approval, which may result in modifications or additions. NXP Semiconductors does not give any representations or warranties as to the accuracy or completeness of information included in a draft version of a document and shall have no liability for the consequences of use of such information.

Disclaimers

Limited warranty and liability — Information in this document is believed to be accurate and reliable. However, NXP Semiconductors does not give any representations or warranties, expressed or implied, as to the accuracy or completeness of such information and shall have no liability for the consequences of use of such information. NXP Semiconductors takes no responsibility for the content in this document if provided by an information source outside of NXP Semiconductors.

In no event shall NXP Semiconductors be liable for any indirect, incidental, punitive, special or consequential damages (including - without limitation - lost profits, lost savings, business interruption, costs related to the removal or replacement of any products or rework charges) whether or not such damages are based on tort (including negligence), warranty, breach of contract or any other legal theory.

Notwithstanding any damages that customer might incur for any reason whatsoever, NXP Semiconductors' aggregate and cumulative liability towards customer for the products described herein shall be limited in accordance with the Terms and conditions of commercial sale of NXP Semiconductors.

Right to make changes — NXP Semiconductors reserves the right to make changes to information published in this document, including without limitation specifications and product descriptions, at any time and without notice. This document supersedes and replaces all information supplied prior to the publication hereof.

Suitability for use — NXP Semiconductors products are not designed, authorized or warranted to be suitable for use in life support, life-critical or safety-critical systems or equipment, nor in applications where failure or malfunction of an NXP Semiconductors product can reasonably be expected to result in personal injury, death or severe property or environmental damage. NXP Semiconductors and its suppliers accept no liability for inclusion and/or use of NXP Semiconductors products in such equipment or applications and therefore such inclusion and/or use is at the customer's own risk.

Applications — Applications that are described herein for any of these products are for illustrative purposes only. NXP Semiconductors makes no representation or warranty that such applications will be suitable for the specified use without further testing or modification.

Customers are responsible for the design and operation of their applications and products using NXP Semiconductors products, and NXP Semiconductors accepts no liability for any assistance with applications or customer product design. It is customer's sole responsibility to determine whether the NXP Semiconductors product is suitable and fit for the customer's applications and products planned, as well as for the planned application and use of customer's third party customer(s). Customers should provide appropriate design and operating safeguards to minimize the risks associated with their applications and products.

NXP Semiconductors does not accept any liability related to any default, damage, costs or problem which is based on any weakness or default in the customer's applications or products, or the application or use by customer's third party customer(s). Customer is responsible for doing all necessary testing for the customer's applications and products using NXP Semiconductors products in order to avoid a default of the applications and the products or of the application or use by customer's third party customer(s). NXP does not accept any liability in this respect.

Terms and conditions of commercial sale — NXP Semiconductors products are sold subject to the general terms and conditions of commercial sale, as published at https://www.nxp.com/profile/terms, unless otherwise agreed in a valid written individual agreement. In case an individual agreement is concluded only the terms and conditions of the respective agreement shall apply. NXP Semiconductors hereby expressly objects to applying the customer's general terms and conditions with regard to the purchase of NXP Semiconductors products by customer.

Export control — This document as well as the item(s) described herein may be subject to export control regulations. Export might require a prior authorization from competent authorities.

Suitability for use in non-automotive qualified products — Unless this document expressly states that this specific NXP Semiconductors product is automotive qualified, the product is not suitable for automotive use. It is neither qualified nor tested in accordance with automotive testing or application requirements. NXP Semiconductors accepts no liability for inclusion and/or use of non-automotive qualified products in automotive equipment or applications.

In the event that customer uses the product for design-in and use in automotive applications to automotive specifications and standards, customer (a) shall use the product without NXP Semiconductors' warranty of the product for such automotive applications, use and specifications, and (b) whenever customer uses the product for automotive applications beyond NXP Semiconductors' specifications such use shall be solely at customer's own risk, and (c) customer fully indemnifies NXP Semiconductors for any liability, damages or failed product claims resulting from customer design and use of the product for automotive applications beyond NXP Semiconductors' standard warranty and NXP Semiconductors' product specifications.

HTML publications — An HTML version, if available, of this document is provided as a courtesy. Definitive information is contained in the applicable document in PDF format. If there is a discrepancy between the HTML document and the PDF document, the PDF document has priority.

Translations — A non-English (translated) version of a document, including the legal information in that document, is for reference only. The English version shall prevail in case of any discrepancy between the translated and English versions.

Security — Customer understands that all NXP products may be subject to unidentified vulnerabilities or may support established security standards or specifications with known limitations. Customer is responsible for the design and operation of its applications and products throughout their lifecycles to reduce the effect of these vulnerabilities on customer's applications and products. Customer's responsibility also extends to other open and/or proprietary technologies supported by NXP products for use in customer's applications. NXP accepts no liability for any vulnerability. Customer should regularly check security updates from NXP and follow up appropriately.

Customer shall select products with security features that best meet rules, regulations, and standards of the intended application and make the ultimate design decisions regarding its products and is solely responsible for compliance with all legal, regulatory, and security related requirements concerning its products, regardless of any information or support that may be provided by NXP.

NXP has a Product Security Incident Response Team (PSIRT) (reachable at PSIRT@nxp.com) that manages the investigation, reporting, and solution release to security vulnerabilities of NXP products.

NXP B.V. — NXP B.V. is not an operating company and it does not distribute or sell products.

Trademarks

Notice: All referenced brands, product names, service names, and trademarks are the property of their respective owners.

NXP — wordmark and logo are trademarks of NXP B.V.

Contents