Embedding it better...

µTasker Document

i.MX RT 1011 – the µTasker way

Table of Contents

1. Introduction

The i.MX RT 1011 is presently the smallest, lowest feature embedded processor in NXP's i.MX RT range. It is however by no means to be underestimated since it is fast (up to 500MHz Cortex-M7 with single-precision FPU), available in user friendly 80 pin LQFP housing and contains a powerful range of internal peripherals including HS USB . It has 128kByte of internal RAM and needs only two major external components to be able to operate:
- a 24MHz oscillator or crystal
- an external program source, usually an SPI Flash connected on its QSPI interface

On top of this, it is very attractively prices, meaning that it competes with NXP Kinetis parts as economic solution - even with Cortex M0+ based KL parts.

The reader should use this document together with the i.MX RT 1021 document:
http://www.utasker.com/docs/iMX/i.MX_RT_1021_uTasker.pdf

since the this document described the differences with regards to the i.MX RT 1011's bigger brother, the i.MX RT 1021.

The **MIMXRT1010-EVK** is used as test vehicle throughout the document but any board using the part can be easily configured based on the contained details.

2. Clocking

The clocking of the i.MX RT 1011 is both flexible and complicated (at first glance). NXP offers a clock configuration tools to help set up clock domains but this is not used in the µTasker project since the method described here not only allows better understanding of the settings but also allows simple and efficient configuration to give lowest power consumption for whichever setup is needed. The µTasker simulator also checks settings to ensure nothing illegal is performed and shows the internal speeds actually achieved as the simulated device operates. Internal clocks that are not required are generally disabled by the µTasker project code and only enabled when required by a particular driver, thus ensuring optimal power efficiency without further higher level programming effort.

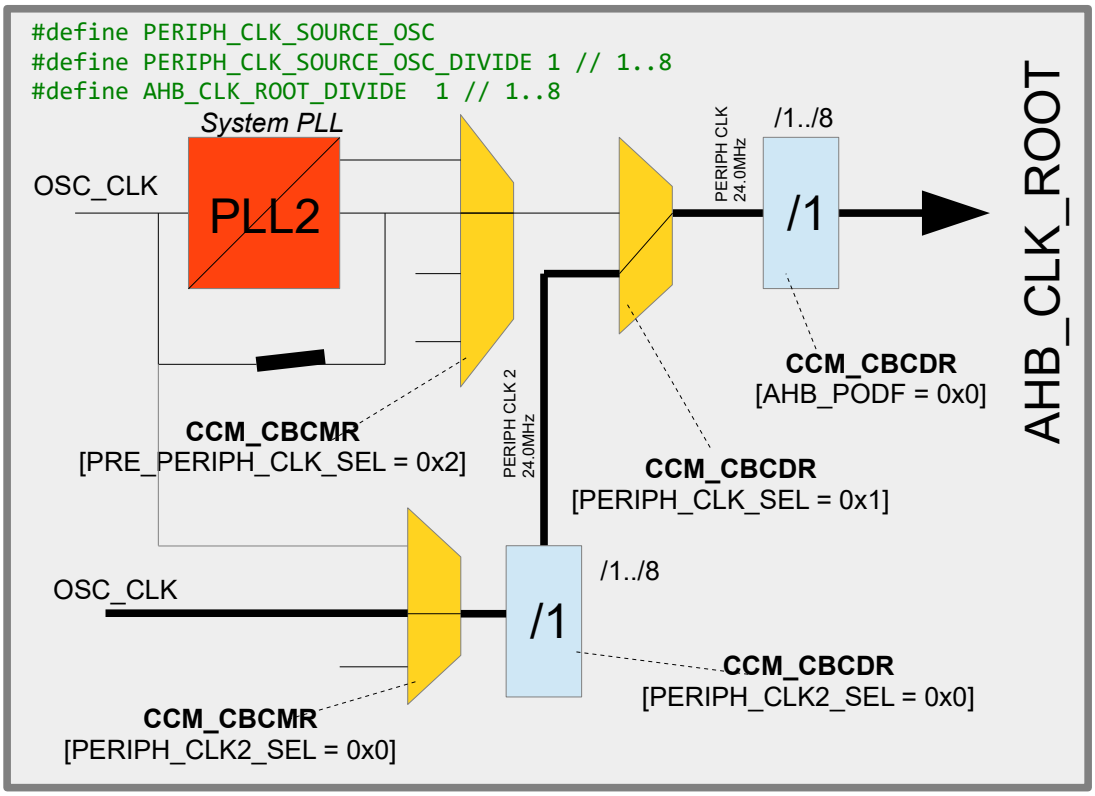
This means that anyone can quickly configure optimal settings for the project in hand, has a fast understand and continuous overview throughout the process.

This chapter describes just differences, when existing, between the clocking and its setup in comparison to that of the i.MX RT 1021, discussed in detail at
http://www.utasker.com/docs/iMX/i.MX_RT_1021_uTasker.pdf

1.1. ARM Core Clock

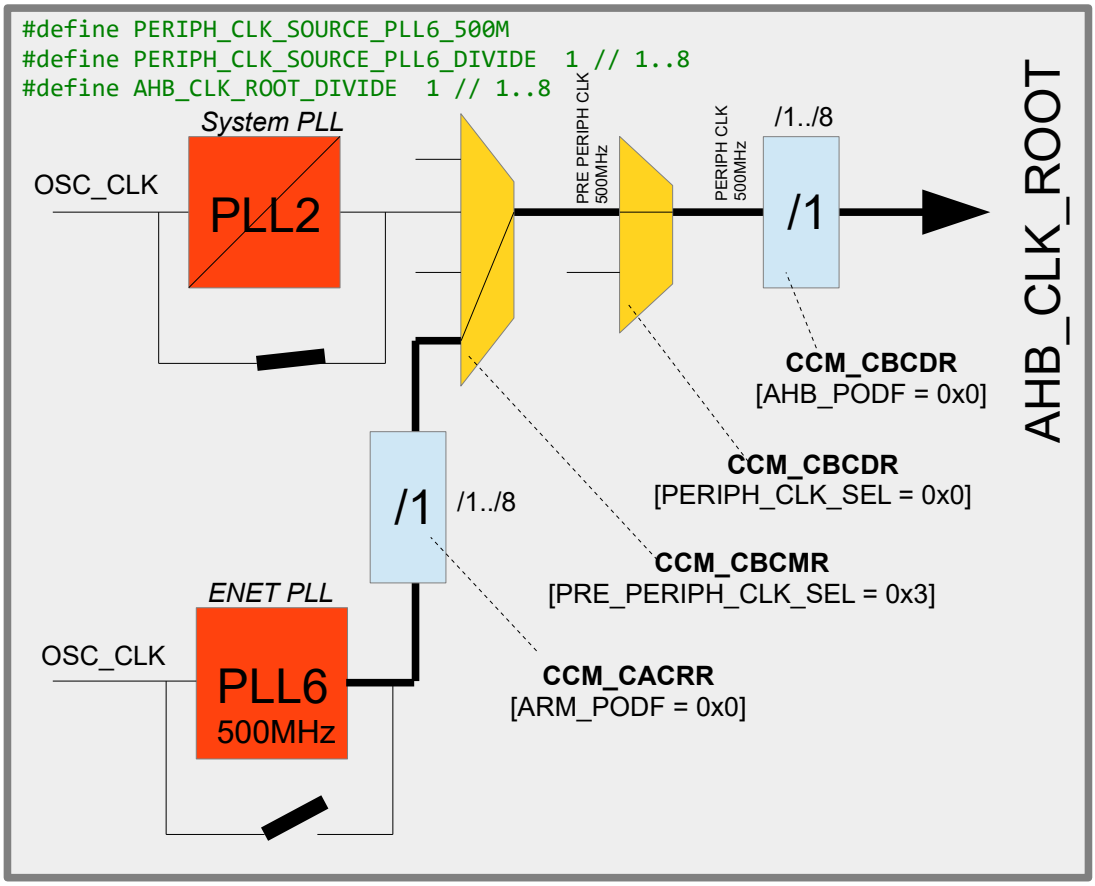
No difference, although the user's manual refers to this as CORE_CLK_ROOT rather than AHB_CLK_ROOT. In this document AHB_CLK_ROOT is used.

```
#define PERIPH_CLK_SOURCE_OSC
```



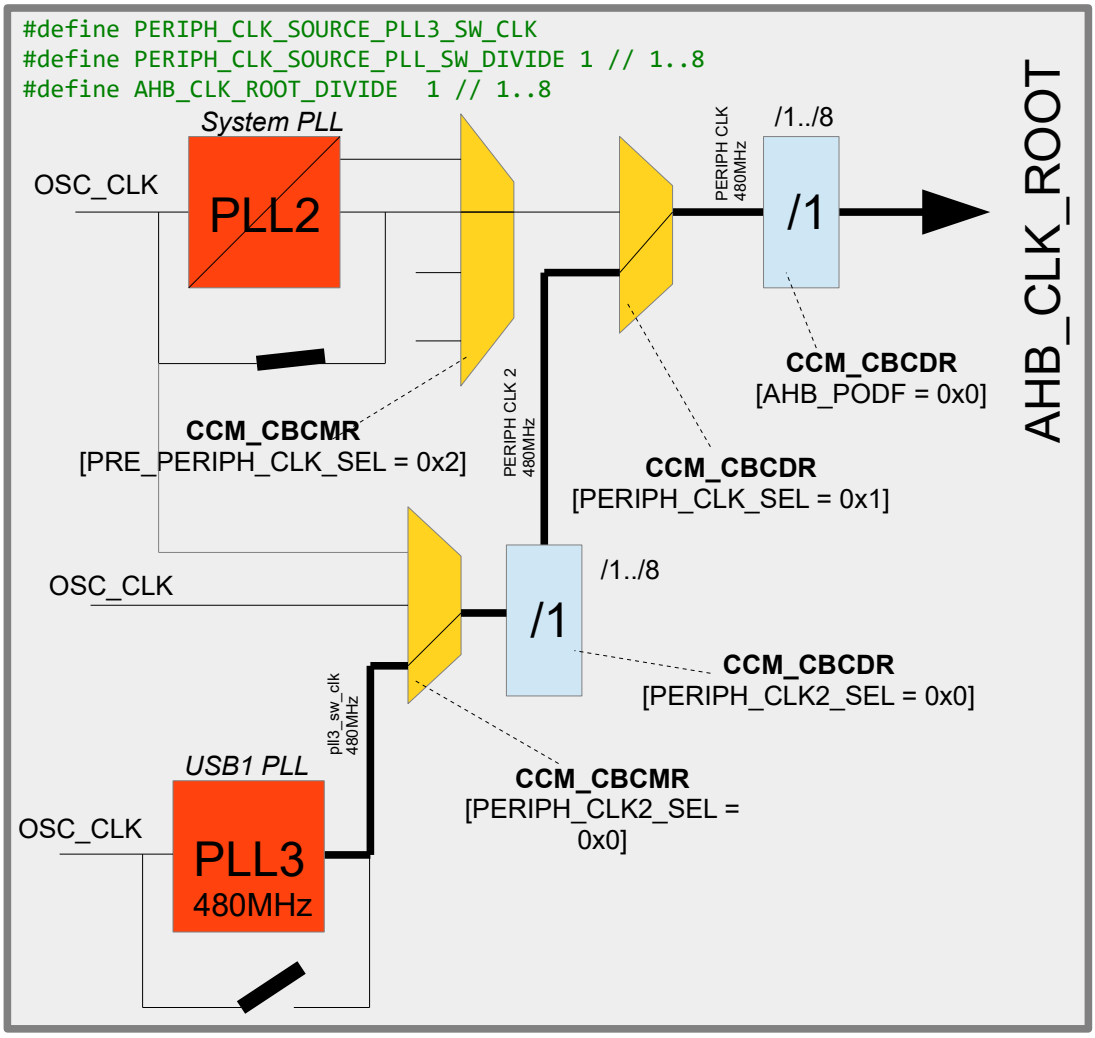
PERIPH_CLK_SOURCE_OSC_DIVIDE is not available in the I.MX RT 1011 and is thus fixed at 1 and any other value is ignored (define doesn't need to be used).

```
#define PERIPH_CLK_SOURCE_PLL6_500M
```



PERIPH_CLK_SOURCE_PLL6_DIVIDE is not available in the I.MX RT 1011 and is thus fixed at 1 and any other value is ignored (define doesn't need to be used).

```
#define PERIPH_CLK_SOURCE_PLL3_SW_CLK
```



PERIPH_CLK_SOURCE_PLL_SW_DIVIDE is not available in the I.MX RT 1011 and is thus fixed at 1 and any other value is ignored (define doesn't need to be used).

The choice of the core clock represents the major work of setting up the clocks. The following details are then specific to peripherals used in the system. *Peripherals of no interest don't need to be specifically configured since they will use defaults and be gated off by the the control code.*

This chapter describes just differences, when existing, between the clocking and its setup in comparison to that of the i.MX RT 1021, discussed in detail at *http://www.utasker.com/docs/iMX/i.MX_RT_1021_uTasker.pdf*

1.2. IPG Clock – used by ADC and XBAR

See i.MX RT 1021 document since the clock details are identical.

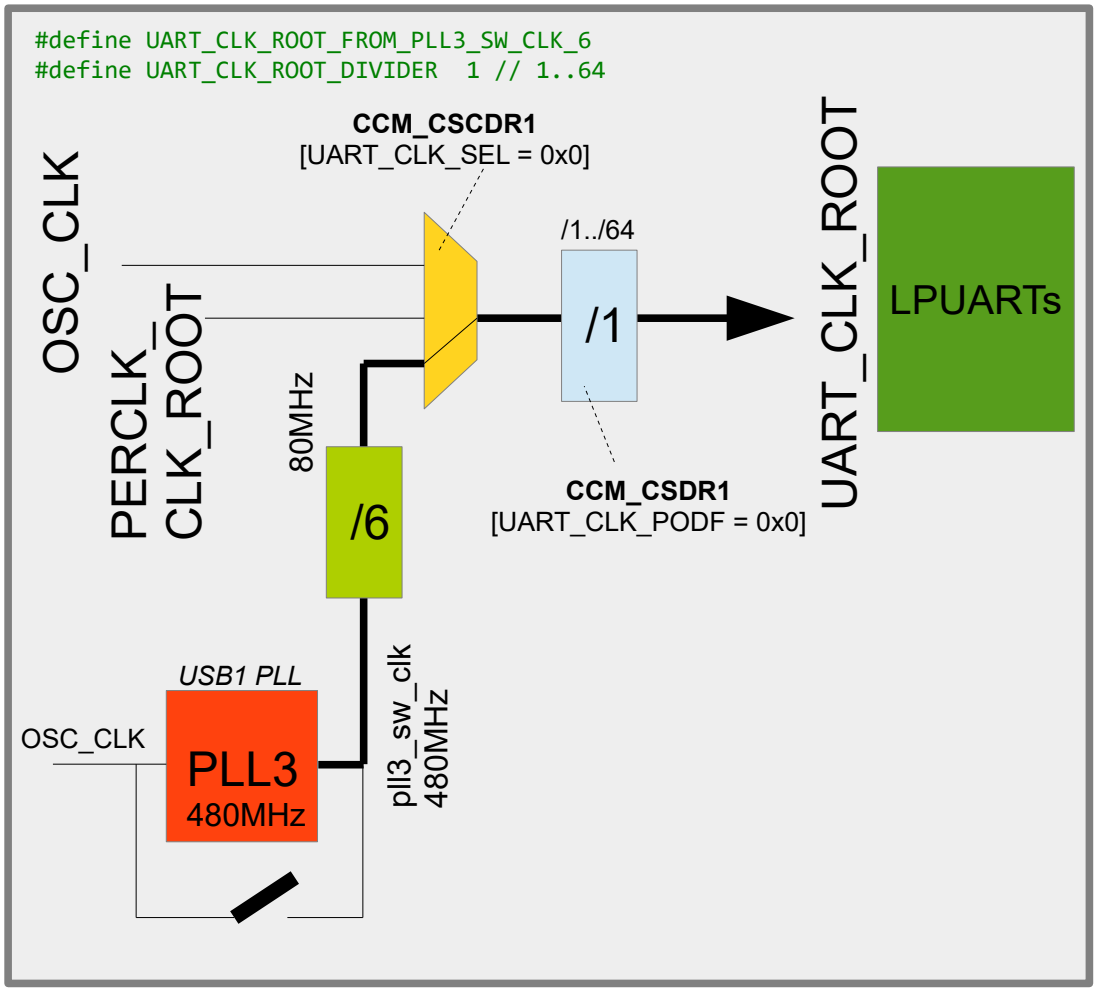
1.3. PERCLK – used by PIT and GPT

See i.MX RT 1021 document since the clock details are identical.

1.4. UART_CLK_ROOT – used by all LPUARTs

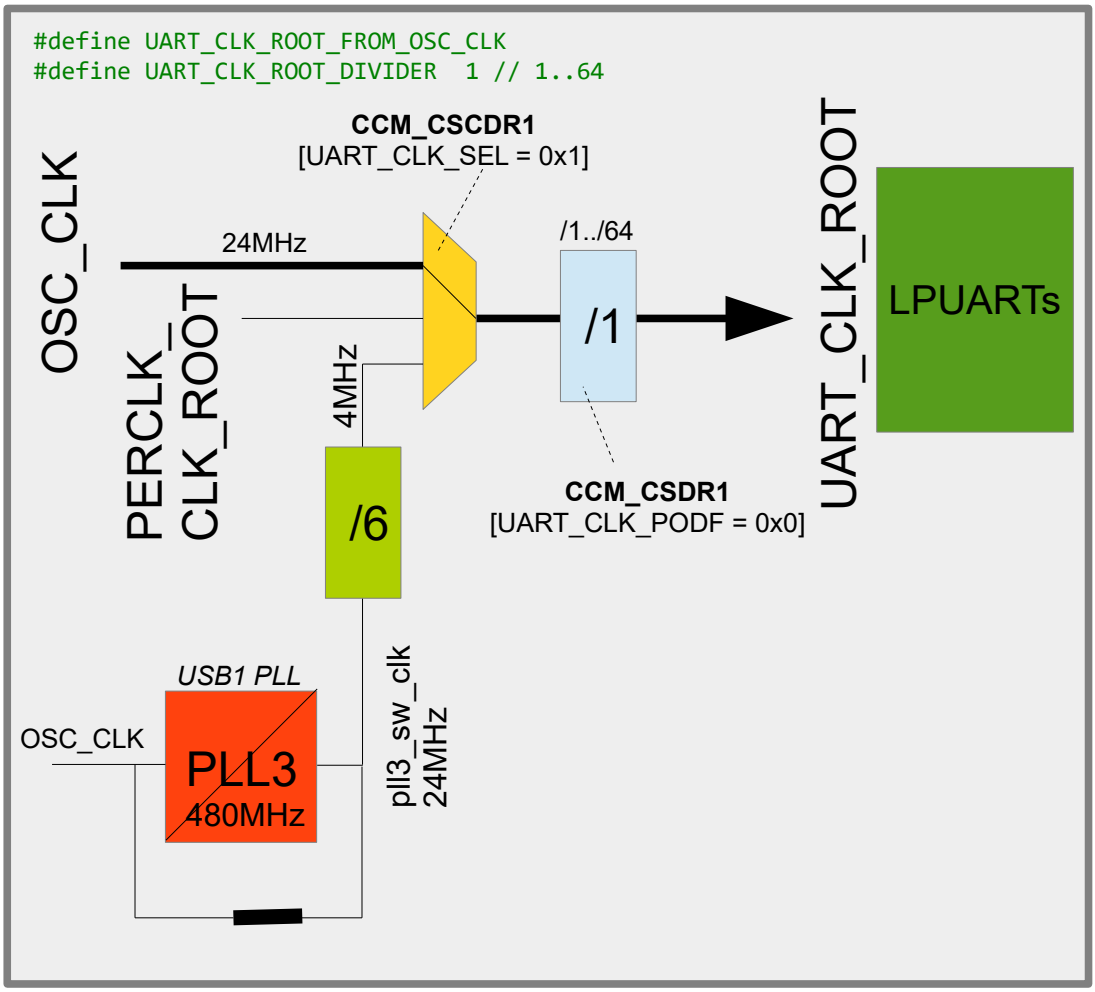
The LPUARTs in the i.MX RT 1011have a common clock that can be derived from either `OSC_CLK`, `pll_sw_clk/6` or `PER_CLK_ROOT`. A common pre-scaler allows the frequency to be divided by 1..64. *`PER_CLK_ROOT` source is an option that doesn't exist in the i.MX RT 1021,*

```
#define UART_CLK_ROOT_FROM_PLL3_SW_CLK_6
```



In this configuration the USB1 PLL is used as reference (called `pll_sw_clk`) and divided by a fixed value of 6, resulting in 80MHz. An optional pre-scaler defined by `UART_CLK_ROOT_DIVIDER` can divide this by 1 to 64 (when not defined, the default is 1)

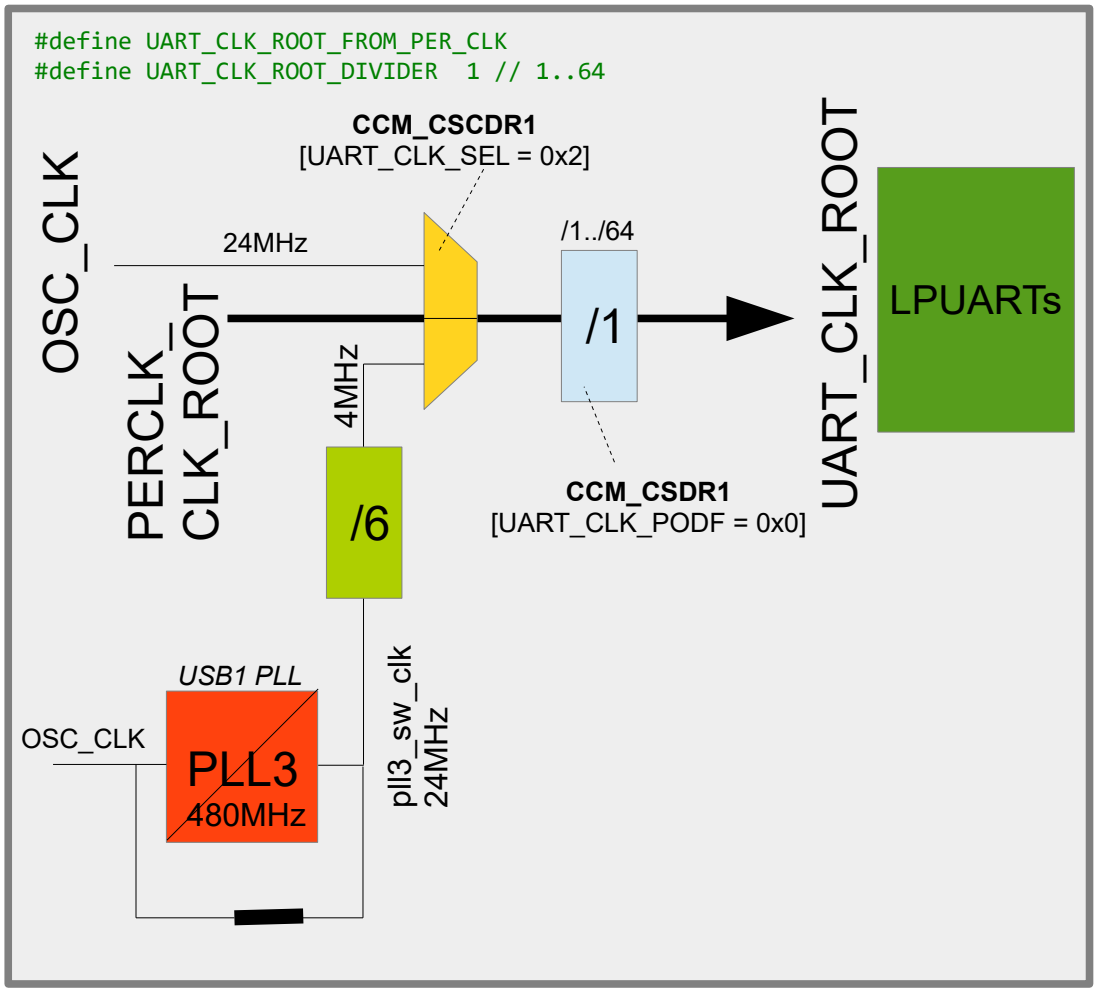
```
#define UART_CLK_ROOT_FROM_OSC_CLK
```



In this configuration the `OSC_CLK` (24MHz) is used as reference, with the optional pre-scaler of 1 to 64.

Note that when PLL3 is not enabled it is left in its powered down, bypassed mode and the alternative clock would be 4MHz instead of 80MHz. This does in fact correspond to the default for the LPUART clock out or reset but this is not used as configuration option since it has no advantage.

```
#define UART_CLK_ROOT_FROM_PER_CLK
```



This configuration is not available to the i.MX RT 1021

The maximum frequecy allowed for `UART_CLK_ROOT` is 80MHz.

The µTasker project driver code will signal a build error if it detects that such a frequency has been exceeded. The µTasker simulator also checks run-time derived frequencies and will exception if it detects such violations.

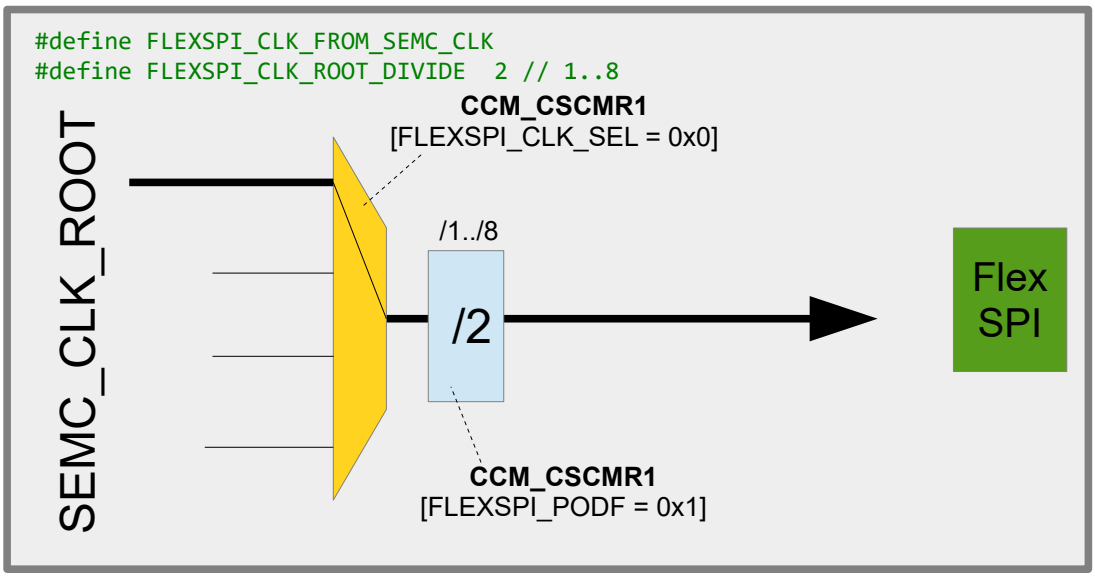
`UART_CLK_ROOT` supplies all LPUARTs but the clock is automatically disabled at each individual LPUART input when the corresponding LPUART is not used.

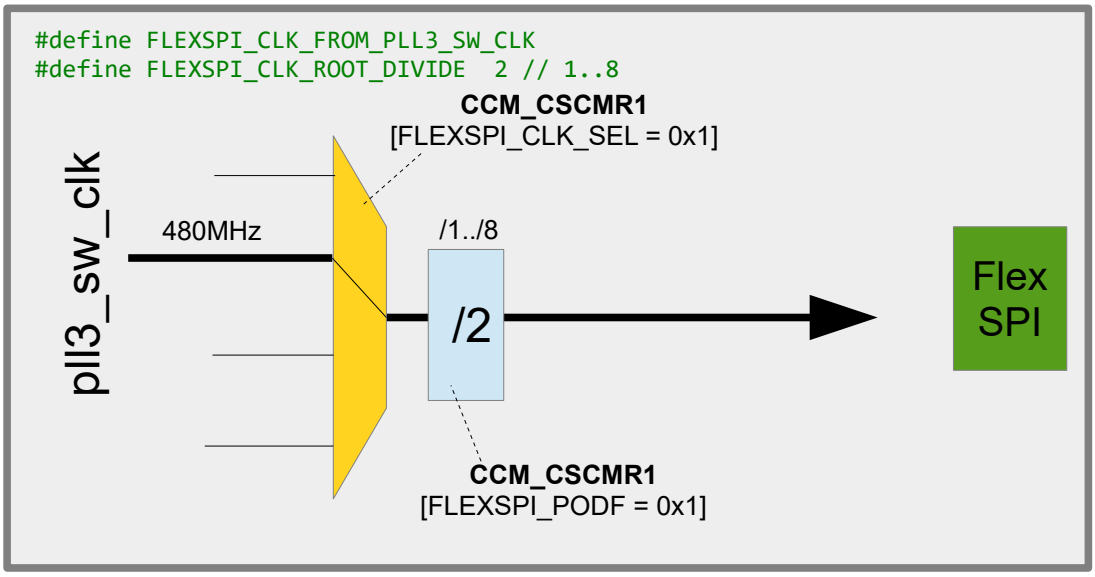
1.5. USDHC1_CLK_ROOT/USDHC2_CLK_ROOT

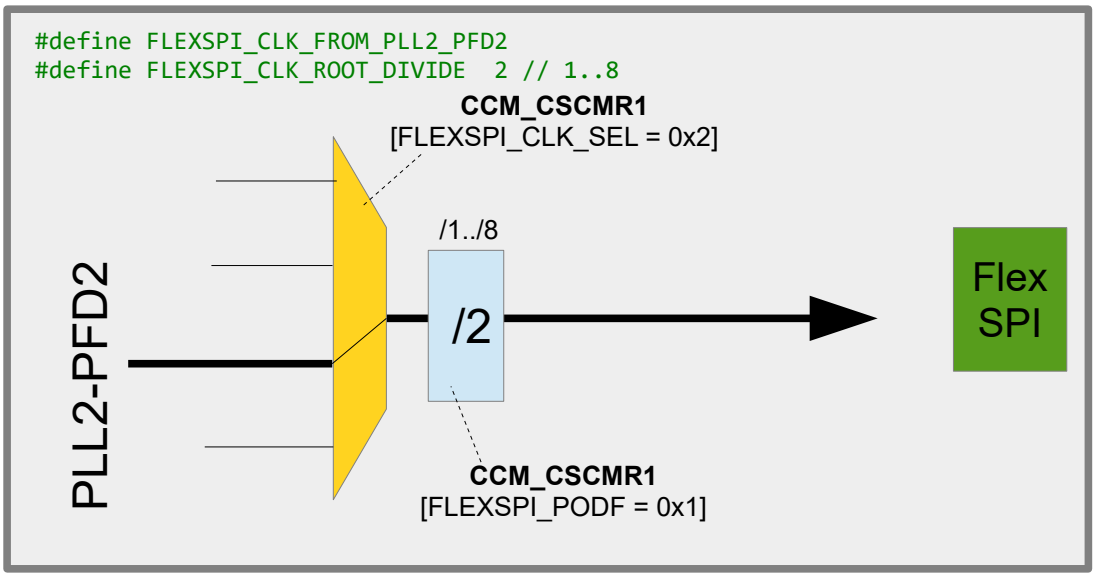
To add..

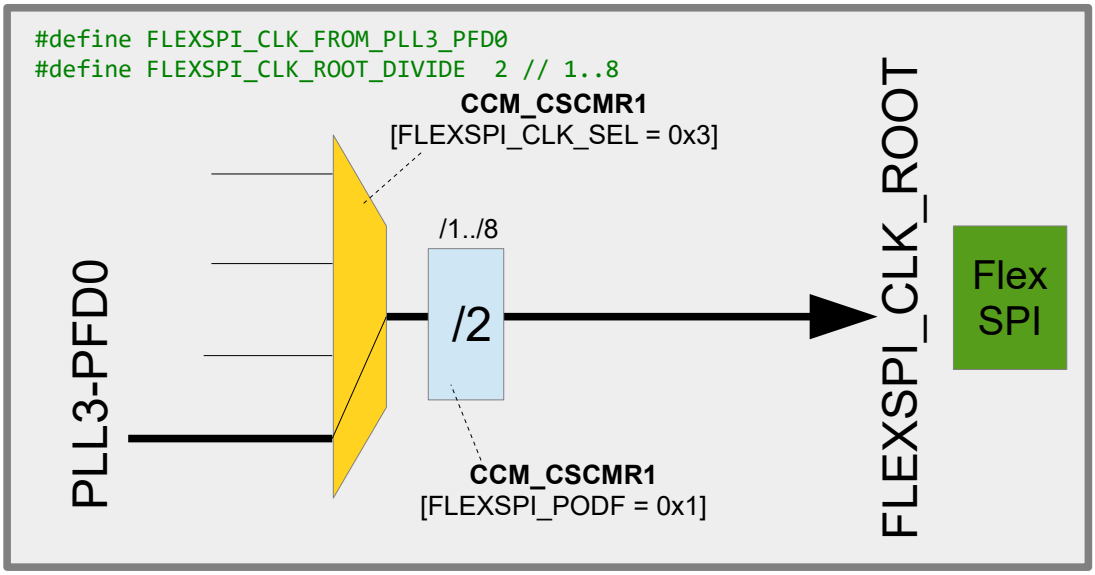
1.6. FLEXSPI_CLK_ROOT

The default source of the FLEXSPI_CLK_ROOT is from the SEMC_CLK_ROOT (see previous section) with a pre-scaler of 2. As discussed in the previous section, the SEMC_CLK_ROOT is per default the PERIPH CLK (the clock root supplying the core clock pre-scaler) divided by 3. FLEXSPI_CLK_ROOT frequency must not exceed 322MHz.






#define FLEXSPI_CLK_FROM_PLL2_PFD2
#define FLEXSPI_CLK_ROOT_DIVIDE 2 // 1..8
CCM_CSCMR1
[FLEXSPI_CLK_SEL = 0x2]
/1../8
/2
PLL2-PFD2
Flex SPI
CCM_CSCMR1
[FLEXSPI_PODF = 0x1]


#define FLEXSPI_CLK_FROM_PLL3_PFD0
#define FLEXSPI_CLK_ROOT_DIVIDE 2 // 1..8
CCM_CSCMR1
[FLEXSPI_CLK_SEL = 0x3]
/1../8
/2
PLL3-PFD0
FLEXSPI_CLK_ROOT
Flex SPI
CCM_CSCMR1
[FLEXSPI_PODF = 0x1]

1.7. LPSPI_CLK_ROOT

To add..

1.8. TRACE_CLK_ROOT

To add..

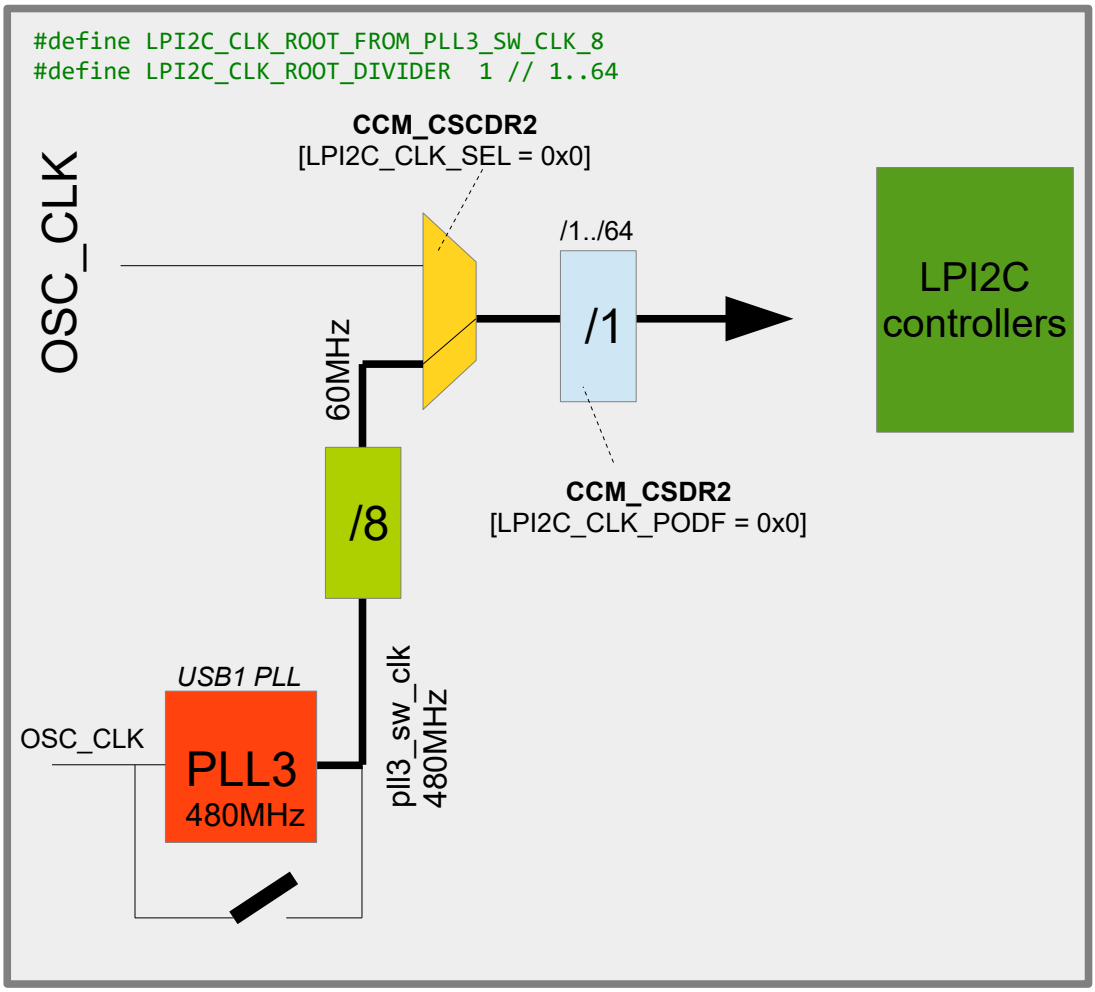
1.9. SAI1_CLK_ROOT/SAI2_CLK_ROOT/SAI2_CLK_ROOT

To add..

1.10. LPI2C_CLK_ROOT – used by all LPI2C controllers

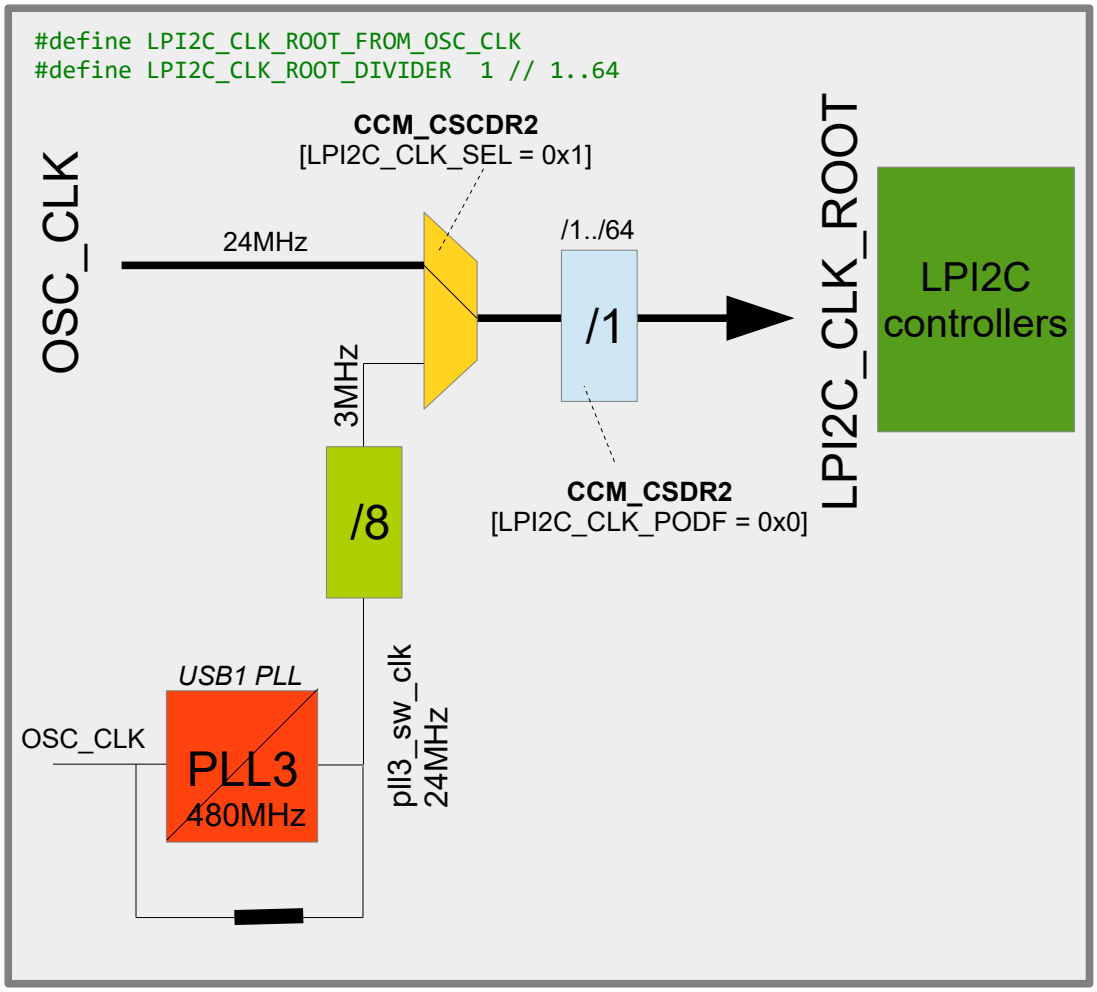
The LPI2C controllers in the i.MX RT 1021have a common clock that can be derived from either `OSC_CLK` or from `pll_sw_clk/8`. A common pre-scaler allows the frequency to be divided by 1..64.

```
#define LPI2C_CLK_ROOT_FROM_PLL3_SW_CLK_8
```



In this configuration the USB1 PLL is used as reference (called `pll_sw_clk`) and divided by a fixed value of 8, resulting in 60MHz. An optional pre-scaler defined by `LPI2C_CLK_ROOT_DIVIDER` can divide this by 1 to 64 (when not defined, the default is 1)

```
#define LPI2C_CLK_ROOT_FROM_OSC_CLK
```



In this configuration the OSC_CLK (24MHz) is used as reference, with the optional pre-scaler of 1 to 64.

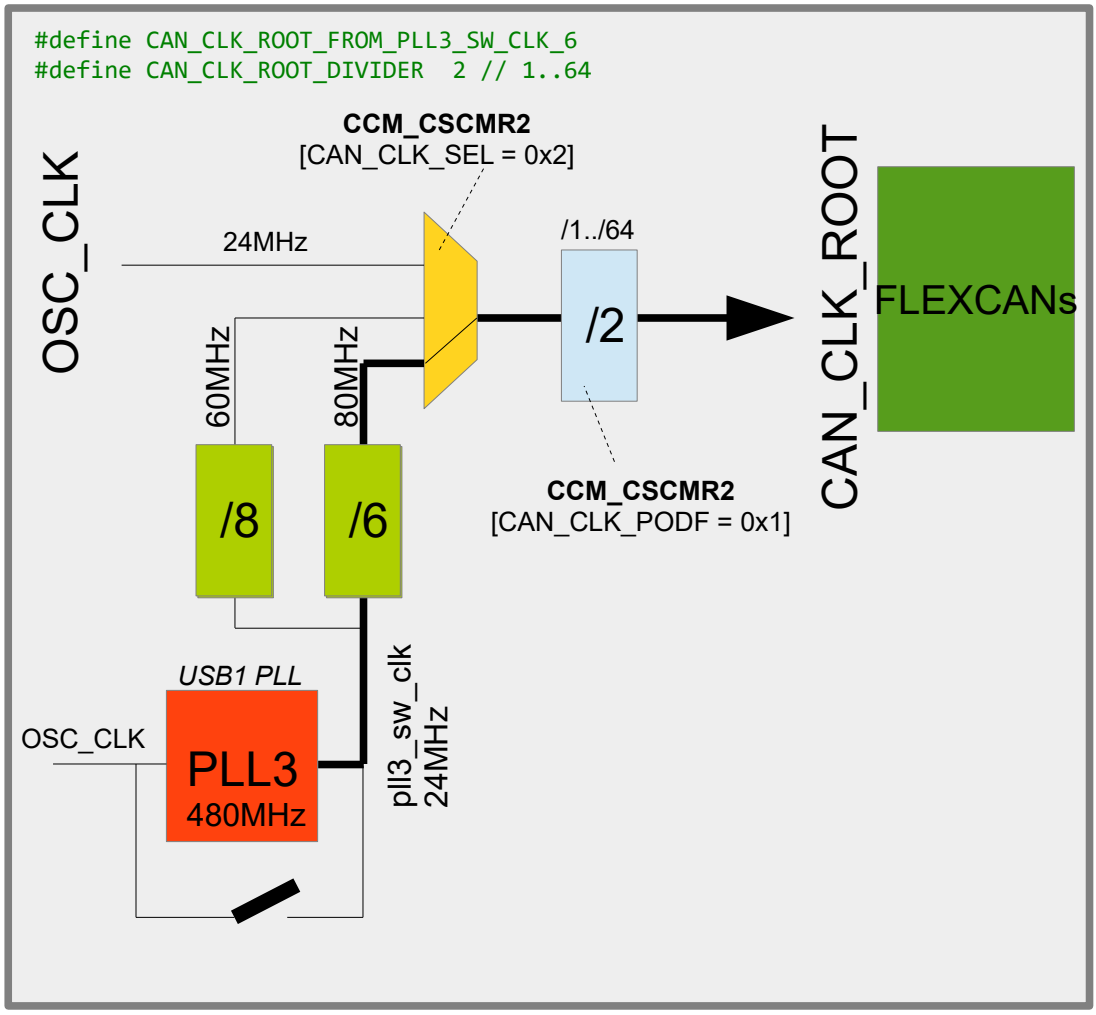
Note that when PLL3 is not enabled it is left in its powered down, bypassed mode and the alternative clock would be 3MHz instead of 60MHz. This does in fact correspond to the default for the LPI2C clock out or reset but this is not used as configuration option since it has no advantage.

LPI2C_CLK_ROOT supplies all LPI2C controllers but the clock is automatically disabled at each individual LPI2C input when the corresponding LPI2C controller is not used.

1.11. CAN_CLK_ROOT

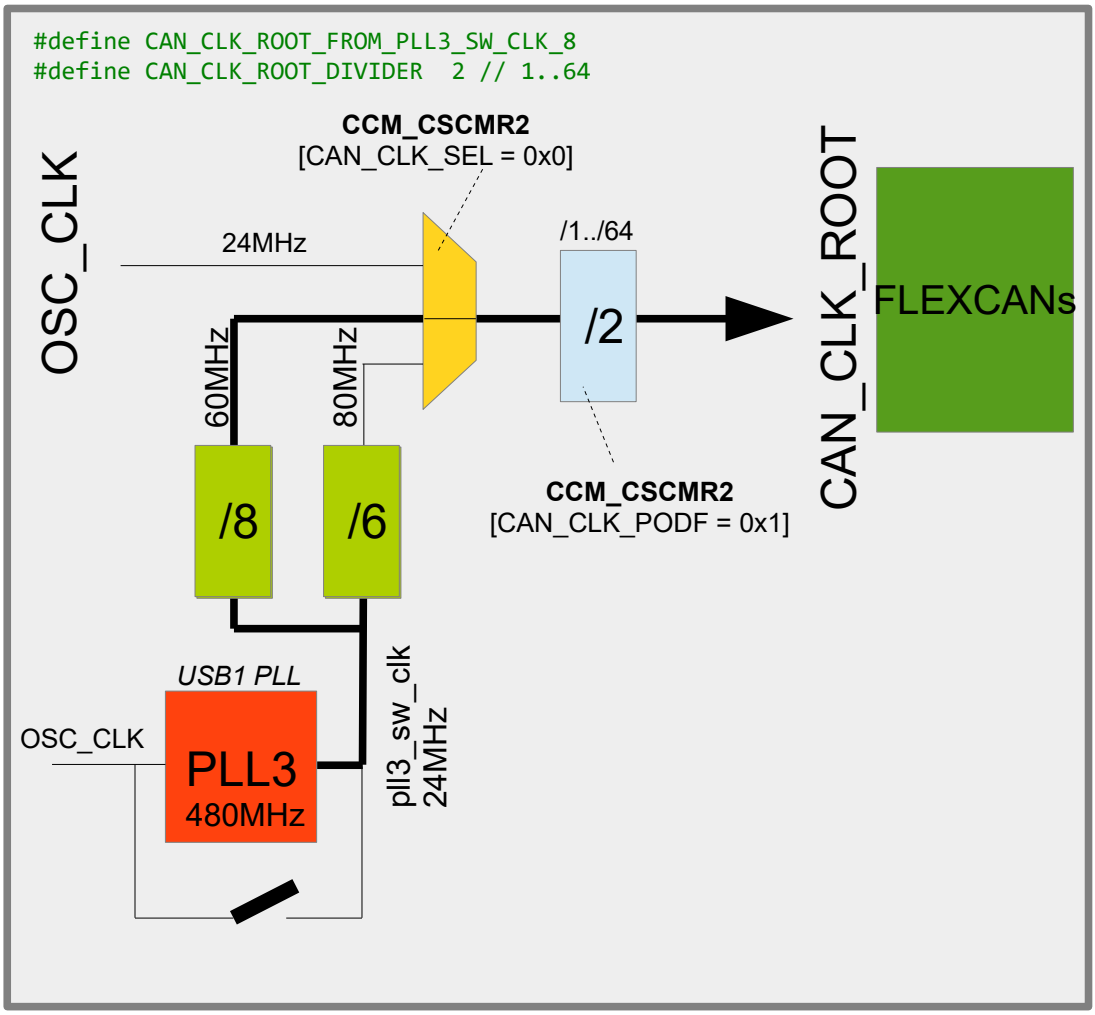
The FLEXCAN controllers in the i.MX RT 1021have a common clock that can be derived from either `OSC_CLK`, `pll_sw_clk/6` or from `pll_sw_clk/8`. A common pre-scaler allows the frequency to be divided by 1..64.

```
#define CAN_CLK_ROOT_FROM_PLL3_SW_CLK_6
```



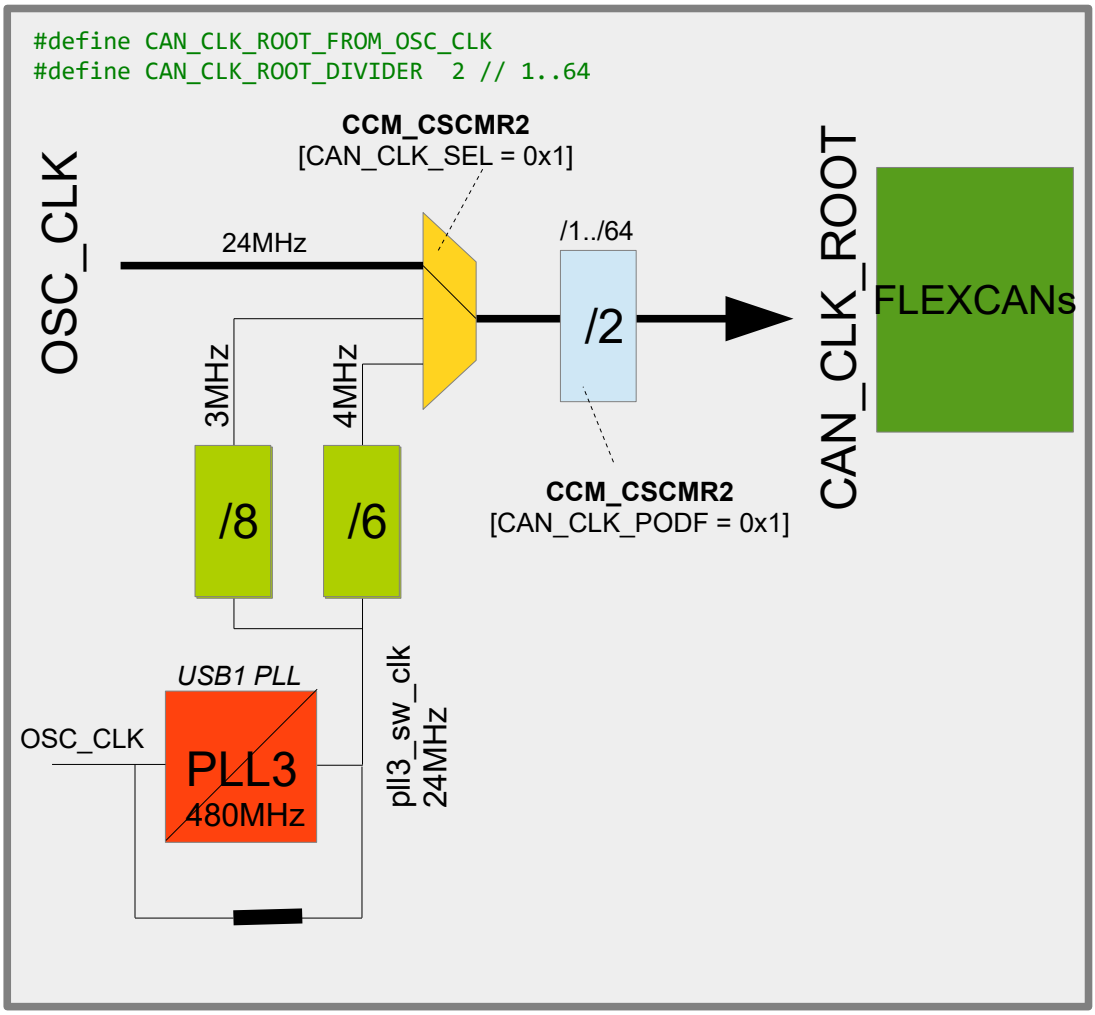
In this configuration the USB1 PLL is used as reference (called `pll_sw_clk`) and divided by a fixed value of 6, resulting in 80MHz. An optional pre-scaler defined by `CAN_CLK_ROOT_DIVIDER` can divide this by 1 to 64 (when not defined, the default is 2)

```
#define CAN_CLK_ROOT_FROM_PLL3_SW_CLK_8
```



In this configuration the USB1 PLL is used as reference (called `pll_sw_clk`) and divided by a fixed value of 8, resulting in 60MHz. An optional pre-scaler defined by CAN_`CLK_ROOT_DIVIDER` can divide this by 1 to 64 (when not defined, the default is 2)

```
#define CAN_CLK_ROOT_FROM_OSC_CLK
```



In this configuration the OSC_CLK (24MHz) is used as reference, with the optional pre-scaler of 1 to 64.

Note that when PLL3 is not enabled it is left in its powered down, bypassed mode and the alternative clocks would be 4/3MHz instead of 80/60MHz. This is not used as configuration option since it has no advantage.

CAN_CLK_ROOT supplies all FLEXCAN controllers but the clock is automatically disabled at each individual FLEXCAN input when the corresponding FLEXCAN controller is not used.

1.12. SPDIF0_CLK_ROOT

To add..

1.13. FLEXIO1_CLK_ROOT

To add..

3. Internal Clock Monitoring

The i.MX RT 1021 has two peripheral outputs called CCM_CLKO1 (on GPIO_SD_B1_02, or GPIO3-22) and CCM_CLKO2 (on GPIO_SD_B1_03, or GPIO3-23) which can be attached to various interal clocks. This can be useful to verify that these clocks really have the frequencies that are expected, as well as generating signals for external usage. Fast internal signals can also be divided down by a pre-scaler with a value between 1 and 8.

These are the clocks that can be selected:

CCM_CLKO1

```
PLL3_SW_CLK_DIV2
PLL2_DIV2
ENET_PLL_DIV2
SEMC_CLK_ROOT
AHB_CLK_ROOT
IPG_CLK_ROOT
PERCLK_ROOT
PLL4_MAIN_CLK
```

CCM_CLKO2

```
USDHC1_CLK_ROOT
LPI2C_CLK_ROOT
OSC_CLK_ROOT
LPSPI_CLK_ROOT
USDHC2_CLK_ROOT
SAI1_CLK_ROOT
SAI2_CLK_ROOT
SAI3_CLK_ROOT
TRACE_CLK_ROOT
CAN_CLK_ROOT
FLEXSPI_CLK_ROOT
UART_CLK_ROOT
SPDIF0_CLK_ROOT
```

Two macros are made available to configure the pin and output the desired signals:

```
fnSetClock1Output(CLK, div)
fnSetClock2Output(CLK, div)
```

whereby examples of utilisation are:

```
    fnSetClock1Output(ENET_PLL_DIV2, 4);
                              // output ENET_PLL/2 with pre-scaler 4 on CCM_CLKO1
    fnSetClock2Output(UART_CLK_ROOT, 1);
                              // output UART_CLK_ROOT with no pre-scaler on CCM_CLKO2
```

4. LPUART

The i.MX RT 1021 LPUART driver is shared with the Kinetis LPUART driver and supports interrupt and DMA driven modes. Minor differences due to the i.MX RT 1021 hardware are controlled by the platform definition `_iMX`. Sharing the driver is possible due to the high compatibility between the LPUART used in the Kinetis parts and i.MX RT 1021 and improves maintenance since only one source needs to be managed and the i.MX RT 1021 inherits the features from the mature Kinetis driver.

See the UART user's manual for general details of usage:
http://www.utasker.com/docs/uTasker/uTaskerUART.PDF

It should be kept in mind that i.MX RT 1021 peripherals tend to be numbered 1..n and not 0..n-1, as is the case with Kinetis peripherals. To avoid confusion it is recommended to use the defines

```
iMX_LPUART_1
iMX_LPUART_2
iMX_LPUART_n
```

instead of channel numbers, whereby `iMX_LPUART_1` is in fact 0.

It is to be noted that the µTasker project is often chosen due to its immediate support for free-running UART Rx DMA on all serial interfaces, which is something that is generally not found in other solutions. The i.MX RT 1021 thus could immediately inherit this operation.

5. LPI2C

The i.MX RT 1021 LPI2C driver is shared with the Kinetis LPUART driver and supports interrupt and DMA driven modes. Minor differences due to the i.MX RT 1021 hardware are controlled by the platform definition `_iMX`. Sharing the driver is possible due to the high compatibility between the LPI2C used in the Kinetis parts and i.MX RT 1021 and improves maintenance since only one source needs to be managed and the i.MX RT 1021 inherits the features from the mature Kinetis driver.

See the I²C user's manual for general details of usage:
http://www.utasker.com/docs/uTasker/uTasker_I2C.pdf

It should be kept in mind that i.MX RT 1021 peripherals tend to be numbered 1..n and not 0..n-1, as is the case with Kinetis peripherals. To avoid confusion it is recommended to use the defines

```
iMX_LPI2C_1
iMX_LPI2C_2
iMX_LPI2C_n
```

instead of channel numbers, whereby `iMX_LPI2C_1` is in fact 0.

6. FLEXCAN

The i.MX RT 1021 CAN driver is shared with the Kinetis CAN driver. Minor differences due to the i.MX RT 1021 hardware are controlled by the platform definition `_iMX`. Sharing the driver is possible due to the high compatibility between the FLEXCAN used in the Kinetis parts and i.MX RT 1021 and improves maintenance since only one source needs to be managed and the i.MX RT 1021 inherits the features from the mature Kinetis driver.

The FLEXCAN in the i.MX RT 1021 supports 64 receive buffers as opposed to the 16 in the FLEXCAN in the Kinetis parts.

See the CAN user's manual for general details of usage:
http://www.utasker.com/docs/uTasker/uTaskerCAN.PDF

It should be kept in mind that i.MX RT 1021 peripherals tend to be numbered 1..n and not 0..n-1, as is the case with Kinetis peripherals. To avoid confusion it is recommended to use the defines

```
iMX_FLEXCAN_1
iMX_FLEXCAN_2
```

instead of channel numbers, whereby `iMX_FLEXCAN_1` is in fact 0.

7. ADC

The i.MX RT 1011 integrates one successive approximation AD controller which achieves up to 12 bit resoluton (with 10..11 bit accuracy) and up to 1MS/s sampling rate.

The ADC has 15 input pins which can be multipled to.

8. PIT

The i.MX RT 1021 PIT driver is shared with the Kinetis PIT driver. Minor differences due to the i.MX RT 1021 hardware are controlled by the platform definition _iMX. Sharing the driver is possible due to the high compatibility between the PIT used in the Kinetis parts and i.MX RT 1021 and improves maintenance since only one source needs to be managed and the i.MX RT 1021 inherits the features from the mature Kinetis driver.

See the HW timer user's manual for general details of usage:
http://www.utasker.com/docs/uTasker/uTaskerHWTimers.PDF

(see `TEST_PIT` [`TEST_PIT_SINGLE_SHOT`, `TEST_PIT_PERIODIC` and `TEST_PIT_64_BIT` in ADC_Timers.h as reference to use).

9. DMA

The i.MX RT 1021 DMA driver is shared with the Kinetis eDMA driver. Minor differences due to the i.MX RT 1021 hardware are controlled by the platform definition `_iMX`. Sharing the driver is possible due to the high compatibility between the eDMA and DMA MUX used in the Kinetis parts and i.MX RT 1021 and improves maintenance since only one source needs to be managed and the i.MX RT 1021 inherits the features from the mature Kinetis driver.

10. GPIO

The i.MX RT 1021 GPIO / peripheral concept is quite different to the Kinetis concept. See the following video for an overview and also details concerning how the project was solved to ensure compatibility between Kinetis and i.MX RT:
https://www.youtube.com/watch?v=SmFTi8hlba0&list=PLWKlVb_MqDQFZAulrUywU30v869JBYi9Q&index=29

GPIOs can also be used as interrupts (see `IRQ_TEST` in `Port_Interrupts.h` as reference to use).

Each GPIO can be configured to generate an interrupt on low levels, high levels, falling edges or rising edges (or both falling and rising edges). The µTasker GPIO interrupt driver allows the user to assign an individual interrupt callback to each GPIO but it is useful to understand that the i.MX RT 1021 actually has the following interrupt vectors:

- `PORT1-0` – individual vector for these pins
- `PORT1-1`
- `PORT1-2`
- `PORT1-3`
- `PORT1-4`
- `PORT1-5`
- `PORT1-6`
- `PORT1-7`

- `PORT1-15..PORT1-8` – these 8 pins share a single vector

- `PORT1-31..PORT1-16` – these 16 pins share a single vector

- `PORT2-15..PORT2-0` – these 16 pins share a single vector
- `PORT2-31..PORT2-16` – these 16 pins share a single vector

- `PORT3-15..PORT3-0` – these 16 pins share a single vector
- `PORT3-31..PORT3-16` – these 16 pins share a single vector

- `PORT5-15..PORT5-0` – these 16 pins share a single vector
- `PORT5-31..PORT5-16` – these 16 pins share a single vector

`PORT1-0..PORT1-7` are the most efficient interrupts since the handler doesn't need to identify which port bits caused the interrupt before dispatching the user interrupt callback. Ports with an interrupt vector shared by more than one pin are slightly less efficient due to the need to identify which source or sources caused the interrupt and then dispatch one or more call-backs. When multiple GPIO input interrupts are pending at the same time the callbacks are dispatched in the order of the lower pin number up to the highest pin number.

11. RAM and Cache

The i.MX RT 1011 contains 128k of internal RAM which is constructed of 4 banks of 32k each. These banks can each be assigned to one of three areas (FlexRAM controller):
- OCRAM General RAM operates at 1/4 the core clock speed (32 bit wide). This is cacheable, meaning that if L1 cache is enabled data content that is already in cache is used to avoid needing to perform the OCRAM access.
- ITCM Instruction Tightly Coupled Memory (64 bit wide) that is optimised for instruction execution at the maximum core speed. Non-cacheable (also since already optimally fast) and so no potential cache synchronisation problems.
- DTCM Data Tightly Coupled Memory (64 bit wide) that is optimised for data access at the maximum core speed. Non-cacheable (also since already optimally fast) and so no potential cache synchronisation problems.

For full details concerning the FlexRAM and optimal configuration to match an applications memory requirements NXP has prepared the application note AN12077 which can be found at https://www.nxp.com/docs/en/application-note/AN12077.pdf

The i.MX RT 1021 has L1 cache with 16kBytes instruction cache and 8kBytes data cache. NXP has prepared the application note AN12042 which can be found at https://www.nxp.com/docs/en/application-note/AN12042.pdf

Use of the cache can ensure high speed operation even when the source of code or data is in a slower memory by avoiding to have to unnecessarily fetch the data when it has been loaded once to the cache.

When the cache is enabled it caches from OCRAM and QSPI-Flash; it neither caches ITCM nor DTCM, which are already tightly coupled to the core.

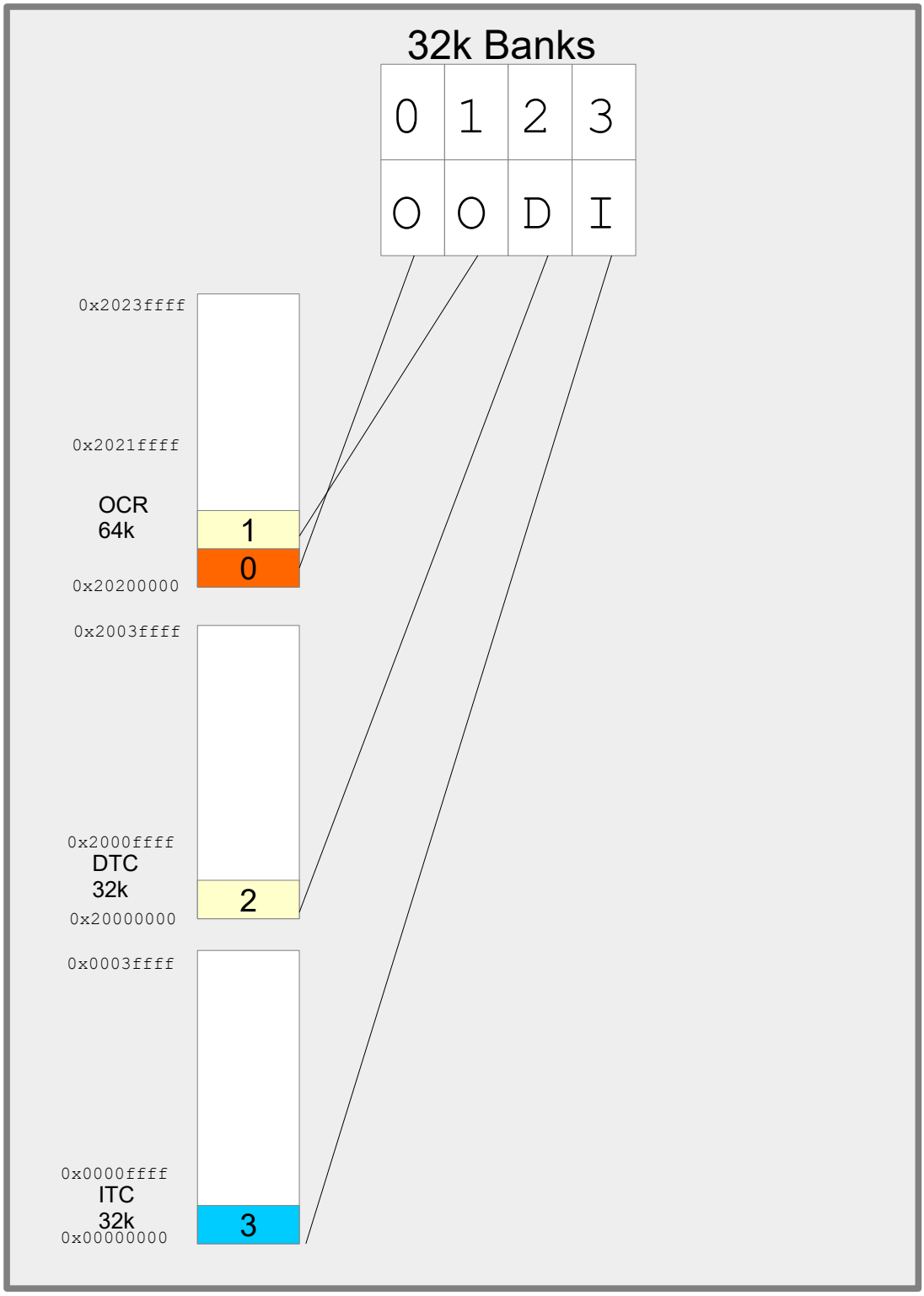
The application can decide whether it uses data or instruction cache with the defines

```
#define ENABLE_INSTRUCTION_CACHE
```

and

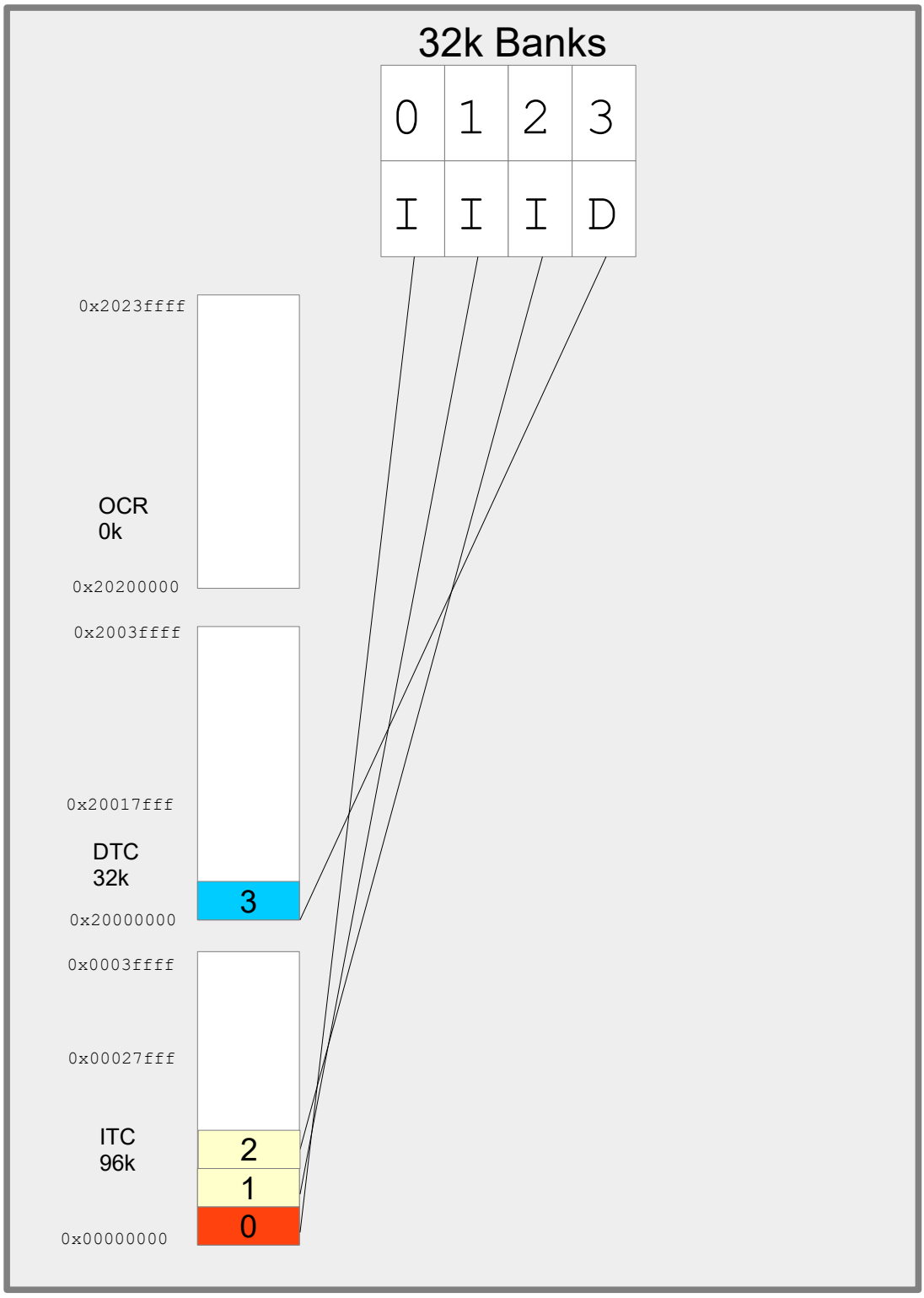
```
#define ENABLE_DATA_CACHE
```

The FlexRAM controller configures the RAM banks at reset based on eFuse settings. The standard setting (when nothing else has been programmed) is for 64k OCRAM, 32k DTCM and 32k ITCM; the ROM loader may use the first 64k of the OCRAM when it operates.



This default setting is assumed in the µTasker project to avoid special configuration requirements and therefore out of reset the banks are configured to give this memory map and layout:

Assuming it is decided that an application were best configured to have 3 banks for ITCM (so that the code could be completely located there – 96k) and 1 bank for data (so that up to 32k of data could be accesses at optimal speed) and no OCRAM the 4 banks could be configured as follows:



The µTasker FlexRAM driver always uses the bank order from instruction use to data use (if OCRAM were used it would be inserted between the two). The most important thing to understand is that when the bank use is modified the address range of the bank is also changed (it is moved in the memory map). This means that data that was in OCR before the change (in the default configuration) still exists in the bank memory but is now addressed in the ITC or DTC memory space instead.

This behaviour makes it complicated to change the memory configuration at run time because it means that any memory used before the change (eg. The stack or initialised variables) are usually at completely different locations after the change. Typically this will cause a program that simply changes the bank configuration without respecting the fact that its memory moves during the process to immediately fail. For this reason such changes are generally not performed during program operation; if such a configuration is changed it tends to be performed before any variable initialisation and also from code running in other sources and without stack dependency.

The µTasker concept assumes that code and variables fit in the internal RAM and so OCRAM is avoided. The division between ITC and OTC is performed at system initialisation automatically to allocate ITC banks to the code space and DTC banks to data space in such a way as to have as much DTC available as possible for heap and stack. If code of 82k were encountered it would thus assign 96k ITC and 32k DTC, as in the example. If less code were encountered additional banks would be assigned to DTC in order to maximise heap and stack availability. Code and data are automatically in the highest performance RAM areas and caching is not required to achieve optimal performance (without caching, no additional synchronisation of data is required).

There is an important reason for choosing the bank ordering: In the default configuration bank 3 is assigned to ITC and will not be used by the ROM loader (the ROM loader may use OCRAM only). After the bank swap is performed this bank is the last bank in DTM, whereby the stack pointer is located near the top, but leaving some additional space above it for 'preserved' variables. The advantage of this is that an application can always write values to the preserved area before a reset and these values will not be modified by the ROM loader. The µTasker boot loader or another application can then read these values, even if the application uses a different RAM bank configuration; as long as its stack pointer is put to near the end of the final bank it will automatically be referenced to the the preserved data area! The preserved area is used in the µTasker project for communicating between applications and the µTasker boot loaders, but can also be used by custom applications for holding data that is guaranteed to be preserved across warm resets.

Due to the nature of the memory operation of the i.MX RT 1011, its configuration requirement to achieve optimal performance and the desire to allow µTasker users to benefit from these with no additional effort the RAM bank management is an integral part of the µTasker boot strategy and the µTasker Boot Loader (see Boot Mode section) an integral part of every project (apart from when a stand-alone application is loaded in a debug environment for test purposes).

12. Boot Mode

The i.MX RT 1021 doesn't have internal flash and needs to boot from an external program source. This is controlled by an internal ROM which offers various boot loader capabilities which are controlled by two inputs (SRC_BOOT_MODE00/GPIO2-IO16 and SRC_BOOT_MODE01/GPIO2-IO17) as well as some eFUSEs (or further pins). *The µTasker project avoids the use of eFUSES and instead controls the operation in its code so that chips can be used in their default configuration*.

The i.MX RT 1021 supports three main modes:

SRC_BOOT_MODE01/GPIO 2-IO17/GPIO_EMC_17	SRC_BOOT_MODE00/GPIO2-IO16/GPIO_EMC_16	
0	0	Boot from fuses
0	1	Serial downloader (LPUART1 or USB1)
1	**0**	**Internal boot**

For simplicity the µTasker project assumes that internal boot option is used as standard, meaning that the ROM loader runs and the exact configuration is taken from eFUSEs or pin overrides, whereby the µTasker assumes that serial (QPI) NOR-Flash is used: the MIMXRT1020-EVK has an IS25LP064A-JBLE to this effect, which is an 8 Mbyte part. It is connected in QSPI mode on the primary FlexSPI interface of the i.MX RT 1021.

To ensure that the NOR-Flash mode is used the processor pins
*GPIO_EMC_25/*GPIO2_IO25

GPIO_EMC_24/GPIO2_IO24

GPIO_EMC_23/GPIO2_IO23

GPIO_EMC_22/GPIO2_IO22

should be pulled down at reset.

SRC_BT_CFG07/GPIO2-IO25/GPIO_EMC_25	SRC_BT_CFG06/GPIO2-IO24/GPIO_EMC_24	SRC_BT_CFG05/GPIO2-IO23/GPIO_EMC_23	SRC_BT_CFG04/GPIO2-IO22/GPIO_EMC_22	
0	**0**	**0**	**0**	**FlexSPI1 (Serial NOR)**
0	0	1	x	SD card
1	0	x	x	MMC/eMMC
0	1	x	x	SEMC (NAND)
0	0	0	1	SEMC (NOR)
1	1	x	x	FlexSPI1 (Serial NAND)

The MIMXRT1020-EVK has all configuration inputs pulled to ground by default and supplies a DIP switch with 4 switches to allow configuring the main boot mode and whether the FlexSPI or SD card is booted from.

The following setting is used:

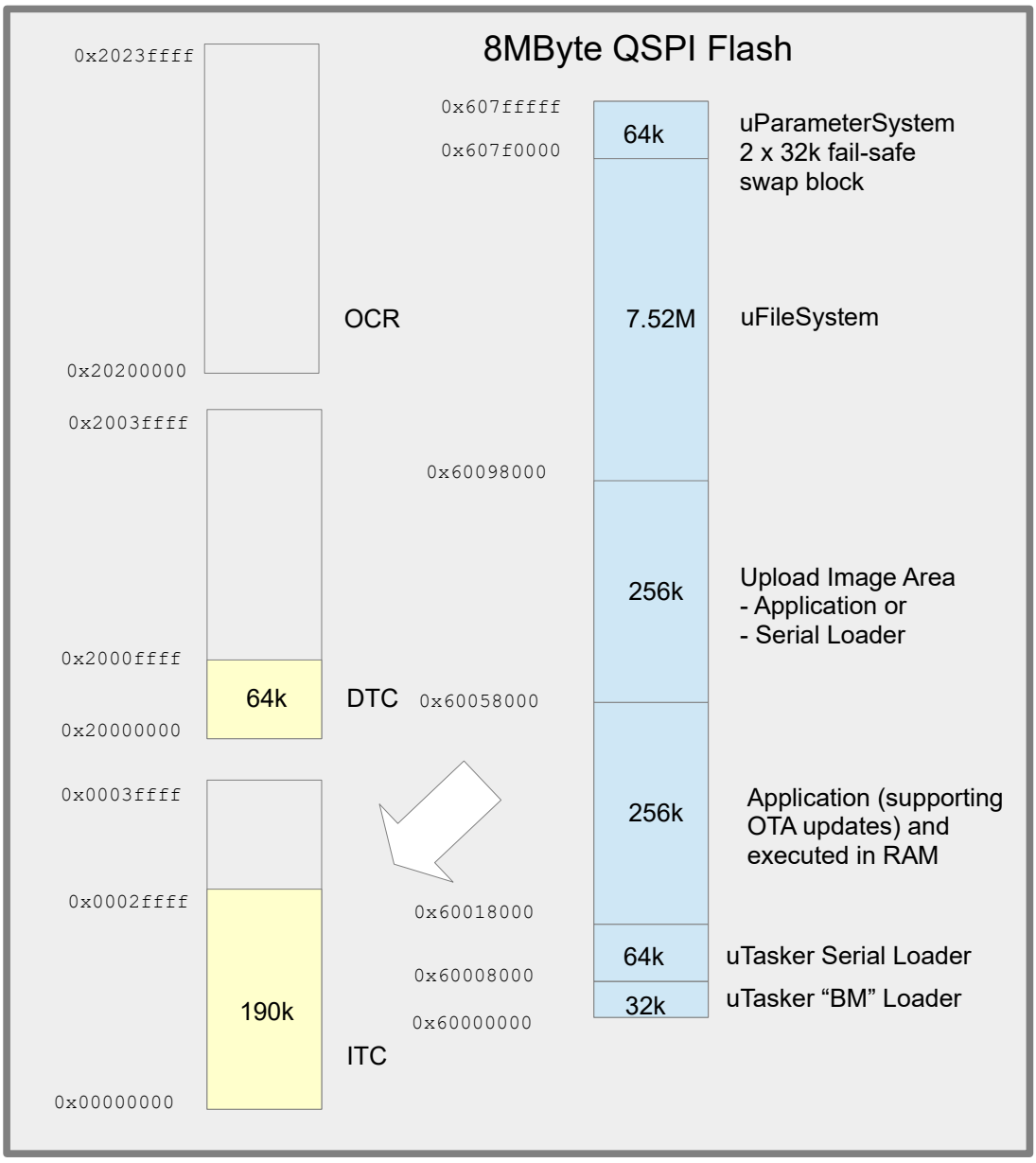
SW8

DIP-1 SRC_BT_CFG00 /GPIO2-IO18/GPIO_EMC_18	**DIP-2** SRC_BT_CFG05 /GPIO2-IO23/GPIO_EMC_23	**DIP-3** SRC_BOOT_MODE01/GPIO2-IO17/GPIO_EMC_17	**DIP-4** SRC_BOOT_MODE00/GPIO2-IO16/GPIO_EMC_16	
OFF	**OFF**	**ON**	**OFF**	Internal boot from FlexSPI1 (serial NOR)
ON	***OFF***	***ON***	***OFF***	*Internal boot from FlexSPI1 (serial NOR)* ***Encrypted XIP***

The eFUSES in a new part are set to supply the following configuration options in NOR Flash boot mode:
BOOT_CFG1[0] = 0 = XIP is not encrypted
BOOT_CFG2[2] = b00 = 500us hold time before read from device
BOOT_CFG1[7..4] = b0000 = serial NOR device
BOOT_CFG1[3..1] = b000 = NOR device supports 0x3b read by default (on primary pin-mux option)

The result is that when the processor starts the very first thing that it does (its internal ROM loader controlling it) is read a block of data from the start of SPI Flash (address 0x60000000 in the memory map) in 2 line mode at 30MHz. This block is expected to contain details about the NOR Flash that is connected (how many lines are connected, what speed is to be used to communicate with it – including setup times - what instruction sequences does it need, plus various other such information.
When the data read is valid (there is also a header to help identify valid content) the ROM loader sets up the FlexSPI interface accordingly, configures the SPI Flash further and begins communicating at the final speed.

The following diagram shows the µTasker boot loader strategy which allows for running applications in internal SRAM (for maximum efficiency), stand-alone code updating (via USB, UART, Ethernet, and with existing protocols such as KBOOT), as well as OTA (Over The Air) loading by the working application via an Upload Image Area in QSPI with the support of the µTasker Bare-Minimum loader for fail safe re-flashing (of new applications or of updated stand-alone Serial Loader).



The operation is as follows:

1. Each time the processor starts, the µTasker “BM” Loader is the first code to be run in QSPI-Flash. It checks to see whether there is a valid new Application or Serial loader in the Upload Image Area.
2. If there is no valid image found it jumps to the Serial Loader (the serial loader is always present)
3. If the serial loader doesn't detect that it should operate (usually via an input) and also detects a valid application in the application code area it calls the µTasker “BM” Loader again, signalling that it should now start this valid application.
4. The µTasker “BM” Loader copies the application code to internal RAM and allows this to start.

2a. If the µTasker “BM” Loader finds either a valid new Application or a valid new Serial loader in the Upload Image Area it deletes the original code and updates it with the new source before resetting so that the new versions can operate.

3a. If the serial loader operates (due to detecting that it should remain in its mode or because there is no valid application code available) it allows new code to be loaded directly to the application area (using the chosen µTaster Serial Loader strategy, such as USB-MSD) before resetting so that the new code can subsequently be executed.

Typically the remaining QSPI-Flash space is used by the application for storing parameters (µParameterSystem) and files (µFileSystem).

The application is responsible for allowing OTA uploads to the Upload Image Area by whatever means is appropriate for the project/product. It can support both OTA loading of new applications (to replace itself) or new serial loader, after which the physical swap of the code is performed by simply commanding a reset so that the µTasker “BM” Loader can complete the work.

Initially there is always a combined firmware consisting of the µTasker “BM” Loader and the µTasker Serial Loader programmed to the board so that the first application can be loaded via the serial loader. Alternatively an initial application can also be combined to immediately have a complete firmware set.

All internal operations performed by the µTasker “BM” Loader are fail safe to ensure that updating code from the Upload Image Area cannot fail and cause the product to subsequently not be operational. *The application's OTA procedure must however ensure that non-operational images are ever loaded as valid new images to ensure that these cannot fail after such an upgrade!*

The exact dimensions of the areas shown are configurable, as are the sizes of the OCR/ITC and DTC areas in RAM. Generally it is the “BM” loader that configures the RAM banks to optimally suit the application that is to be loaded and initialises the application's initial stack pointer suitably to the top of DTC memory (see the section on RAM and Cache for more details).

13. Ethernet

The i.MX RT 1021 has an internal 10/100M Ethernet controller but requires an external PHY for the connection to the cable.

There is a dedicated 500MHz fixed frequency PLL (PLL6) which supplies the Ethernet controller clocks. It also has a fixed 25MHz output referred to as ref_enet_pll2 and a configurable output called ref_enet_pll1, programmable to 25MHz, 50MHz, 100MHz or 125MHz.

The MIMXRT1020-EVK uses a Micrel KSZ8081RNB as PHY, connected in RMII mode. This PHY accepts a 25MHz reference clock from the processor (Ethernet controller) which is supplied on `GPIO_AD_B0_08 (GPIO1-08)` – peripheral function `ENET_REF_CLK1`

14. Conclusion

The present state of development is that the i.MX RT 1021 (MIMXRT1020-EVK) can be operated from SRAM (with debugger) or from QSPI Flash (stand-alone) in various clocking configurations up to maximum speed. GPIO, port interrupts, LPUARTs (interrupt and DMA driven), Ethernet, PIT, dynamic low power (WAIT) mode and DMA operations (memory-to-memory) can be demonstrated. Instruction and dat cache can be optionally enabled/disabledAll such operation is simulated in visual studio.

Next immediate steps:
- Add QSPI loader to run exclusively in SRAM (for optimal speed efficiency)
- Demonstrate full Modbus serial and TCP operation.
- Verify dynamic low power operation

Subsequent steps:
- Add (new) USB device driver
- Add FreeRTOS configurations
- Verify on cheapest i.MX RT 1010 system (without ENET)

Present problems/investigation:

- memory to memory DMA transfer is not giving any speed benefits over CPU copy: https://community.nxp.com/thread/518925
- Free-running LPUART DMA reception works correctly when cache is disabled but has disturbances when cache is enabled
- Ethernet works reliably after a power cycle but not always after a push-button or a commanded reset
- Push button resets are understood as power cycle resets by the reset monitor

Modifications:
V0.07 : Initial version in development

Appendix A – Hardware Dependencies

a) Space for first Appendix