Embedding it better...

µTasker Document

i.MX RT 1021 – the µTasker way

Table of Contents

1. Introduction....3
2. Clocking....4
1.1.ARM Core Clock....5
1.2.IPG Clock – used for the access of most peripherals....13
1.3.PERCLK – used by PIT and GPT....14
1.4.UART_CLK_ROOT – used by all LPUARTs....16
1.5.USDHC1_CLK_ROOT/USDHC2_CLK_ROOT....18
1.6.SEMC_CLK_ROOT – external memory controller clock....20
1.7.FLEXSPI_CLK_ROOT....22
1.8.LPSPI_CLK_ROOT – used by all LPSPIs....24
1.9.TRACE_CLK_ROOT....26
1.10.SAI1_CLK_ROOT/SAI2_CLK_ROOT/SAI2_CLK_ROOT....26
1.11.LPI2C_CLK_ROOT – used by all LPI2C controllers....27
1.12.CAN_CLK_ROOT....29
1.13.SPDIF0_CLK_ROOT....32
1.14.FLEXIO1_CLK_ROOT....32
1.15.USB1 Clock....32
3. Real Time Clock....33
4. Internal Clock Monitoring....34
5. LPUART....35
6. LPI2C....36
7. LPSPI....37
8. FLEXCAN....38
9. USDHC....39
10. PIT....40
11. DMA....41
12. GPIO....42
13. RAM and Cache....43
14. Boot Mode....47
15. Total Security and Simplicity....51
16. Ethernet....52
17. XBAR and AOI....53
18. Conclusion....56
Appendix A – Hardware Dependencies....57
a)Space for first Appendix....57

1. Introduction

The i.MX RT 1021 can be considered as the original basic embedded processor in NXP's i.MX RT range. It is fast (up to 500MHz Cortex-M7 with double-precision FPU), available in user friendly LQFP housing and contains a large and powerful range of internal peripherals including HS USB and Ethernet controller. It has 256kByte 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 relatively cheap, meaning that it competes with NXP Kinetis parts as economic solution - even with Cortex M0+ based KL parts.

The µTasker project has embraced this part as initial focus for its i.MX RT port, allowing Kinetis users to move Kinetis projects to more powerful i.MX RT parts with virtually no effort thanks to the highly compatible µTasker HAL (Hardware Application Layer). The main interest of the project is to allow applications that may originally have been designed to run on Kinetis parts to run on i.MX RT parts completely in internal RAM so that the full speed of the Cortex M7 can be utilised, whereby an SPI Flash based program loading device is shared with parameter and file systems to ensure feature compatibility without additional memory devices being needed. *i.MX RT 1021 projects can equally simply be moved to Kinetis parts when this makes sense*.

This document discusses technical details of the i.MX RT 1021 of interest to users of the device and especially for users of the µTasker project with the device so that projects can be quickly and accurately configured to achieve the required level of performance and power efficiency.

The **MIMXRT1020-EVK** is used as test vehicle throughout the document but any board using the part can be easily configured based on the contained details.

It is expected that many details concerning the i.MX RT 1021 are valid across the i.MX RT range or can be easily interpolated to other parts where needed.

2. Clocking

The clocking the i.MX RT 1021 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.

The first detail of interest is that the i.MX RT 1021 is designed to run from a fixed frequency reference called **OSC_CLK**. This is either an external 24.0MHz oscillator connected to its XTALI pin or a 24.0MHz crystal across XTALI and XTALO. No other frequency should be used because various internal PLLs are designed to derive their required output frequencies from this single frequency source.

The next thing to be aware of is that if no specific configuration is performed the processor will run at this 24MHz OSC_CLK frequency and various internal clocks reference to – or from divided versions of it. To illustrate the various clocking possibilities the default state is the first one discussed in the following clock initialisations, followed by a complete list of possibilities that can be configured as alternates. The first clock to be discussed is the one used by the ARM core itself – which is of course indispensable for the processors program operation - followed by additional (sometimes optional) internal clocks that are used for individual peripherals or groups of peripherals.

Once the core clock has been configured from one of a set of possibilities it will be seen that all further peripheral clocks are very simple to then define for their defined usage.

1.1. ARM Core Clock

The ARM core is clocked by an internal clock signal called **AHB_CLK_ROOT**, which can be up to 500MHz. It may be automatically gated off in certain low power mode but these modes are of no concern to the discussion of the clocking capabilities and configuration. The following diagrams show the complete set of possibilities together with the project defines that control them – as well as divider settings and their ranges. The AHB_CLK_ROOT speed is probably the most important clock setting and the one that one will define as first frequency setting.

It is seen that there are essentially 7 possible settings. *There are in fact some addition permutations but they don't make any sense to use in the i.MX RT 1021 because they are just further duplications of what is possible, with no added advantage; for example there are further paths that could select OSC_CLK as source which are not shown to avoid unnecessary complications.*

The 7 possibilities are now shown, whereby in each case the user can define an optional output divider between 1 and 8. It will be seen that three PLL sources are possible as source for this clock whereby it is useful to understand that these PLLs are all powered down by default and bypassed so that their input clock (OSC_CLK in every case) is available at their outputs. If they are used they are powered up, their lock waited for and then the bypass removed. Each of these PLLs is a fixed frequency PLLs, meaning that the VCO output is defined to generate a fixed frequency from the fixed 24MHz OSC_CLK input. However, some of the PLLs have PFDs (Phase Fractional Dividers) which can be tapped as output too – for example the System PLL (PLL2), which has its main output at 528MHz also has 4 PFD outputs (PFD0..PFD3) with frequencies of 352MHz, 594MHz, 396MHz and 594MHz respectively by default. Each of the PFD output frequencies can however be individually programmed or disabled, whereby the formula for the respective PDF frequency is PLL fixed ((frequency * 18) / fraction) where fraction can be any integer value between 12 and 35. The following table shows the complete list of frequencies that can be selected for PLL2 and PLL3, whereby the PFD3 or each are optional core clock references as detailed further below.

Fraction	System PLL (528MHz PLL2) ((528MHz * 18) / fraction)	USB1 PLL (480MHz PLL3) ((480MHz * 18) / fraction)
12	792MHz	720MHz - default PFD0
13	731.0769231MHz	664.6153846MHz - default PFD1
14	678.8571429MHz	617.1428571MHz
15	633.6MHz	576MHz
16	594MHz - default PFD1 and PDF3	540MHz
17	559.0588235MHz	508.2352941MHz - default PFD2
18	528MHz	480MHz
19	500.2105263MHz	454.7368421MHz - default PFD3
20	475.2MHz	432MHz
21	452.5714285MHz	411.4285714MHz
22	432MHz	392.7272727MHz
23	413.2173913MHz	375.6521739MHz
24	396MHz - default PFD2	360MHz
25	380.16MHz	345.6MHz
26	365.5384615MHz	332.3076923MHz

27	352MHz - default PFD0	320MHz
28	339.4285714MHz	308.5714286MHz
29	327.7241379MHz	297.9310345MHz
30	316.8MHz	288MHz
31	306.58064524MHz	278.7096774MHz
32	297MHz	270MHz
33	288MHz	261.8181818MHz
34	279.5294118MHz	254.1176471MHz
35	271.5428571MHz	246.8571429MHz

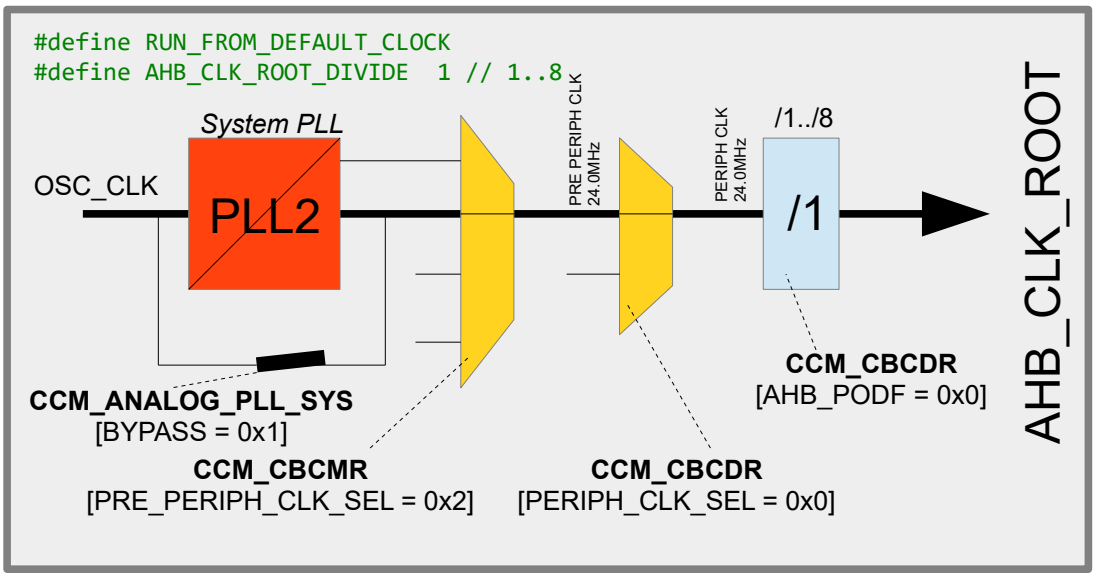
Although the system PLL2 is intended to be used as a fixed 528MHz oscillator it can in fact be adjusted within the range 528MHz .. 552MHz be modifying its PLL loop settings. If the frequency is adjusted the values in the previous table are of course no longer exact and need to also be adjusted accordingly!

Very fine adjustments can be made and the optional define

```
#define PLL2_FREQUENCY                          528123456
```

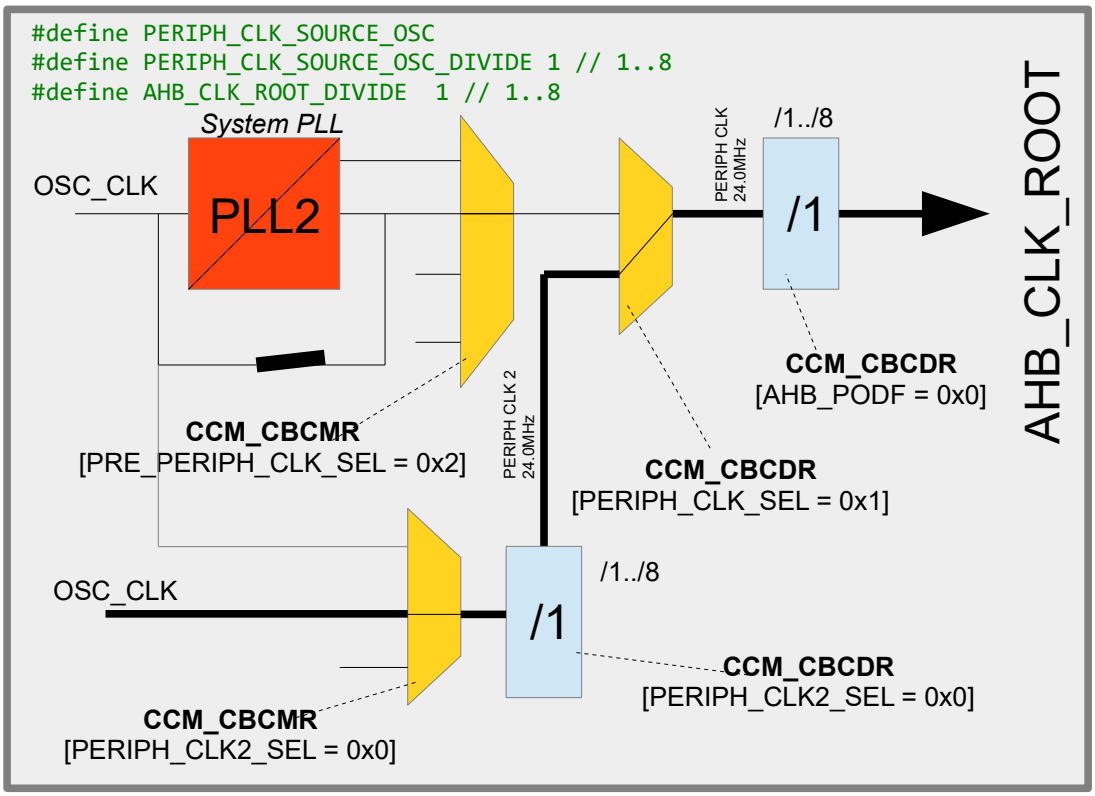
causes the clock configuration to be performed to give the required frequency.

```
#define RUN_FROM_DEFAULT_CLOCK
```



Choose this setting to use the default clock configuration, which is in effect the OSC_CLK (bypassed at the system PLL and switched through as shown to give 24MHz). The user can optionally reduce this frequency by a factor or /1 to /8 with the define `#define AHB_CLK_ROOT_DIVIDE` set to the value desired (defaults to 1 if not defined).

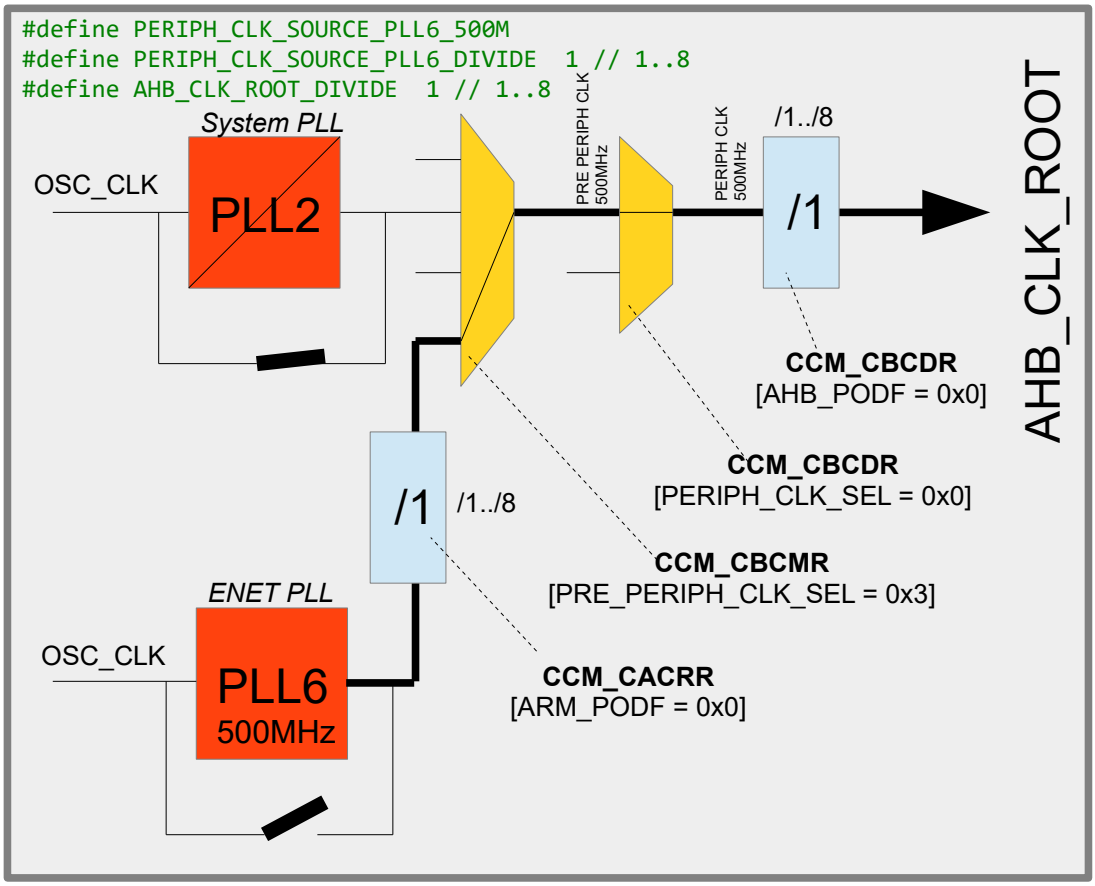
```
#define PERIPH_CLK_SOURCE_OSC
```



This is a variation of the same theme as the default clock setting but instead of routing the OSC_CLK from the bypassed system PLL it is routed via a different path. In this configuration there is a second optional pre-scaler, `#define`

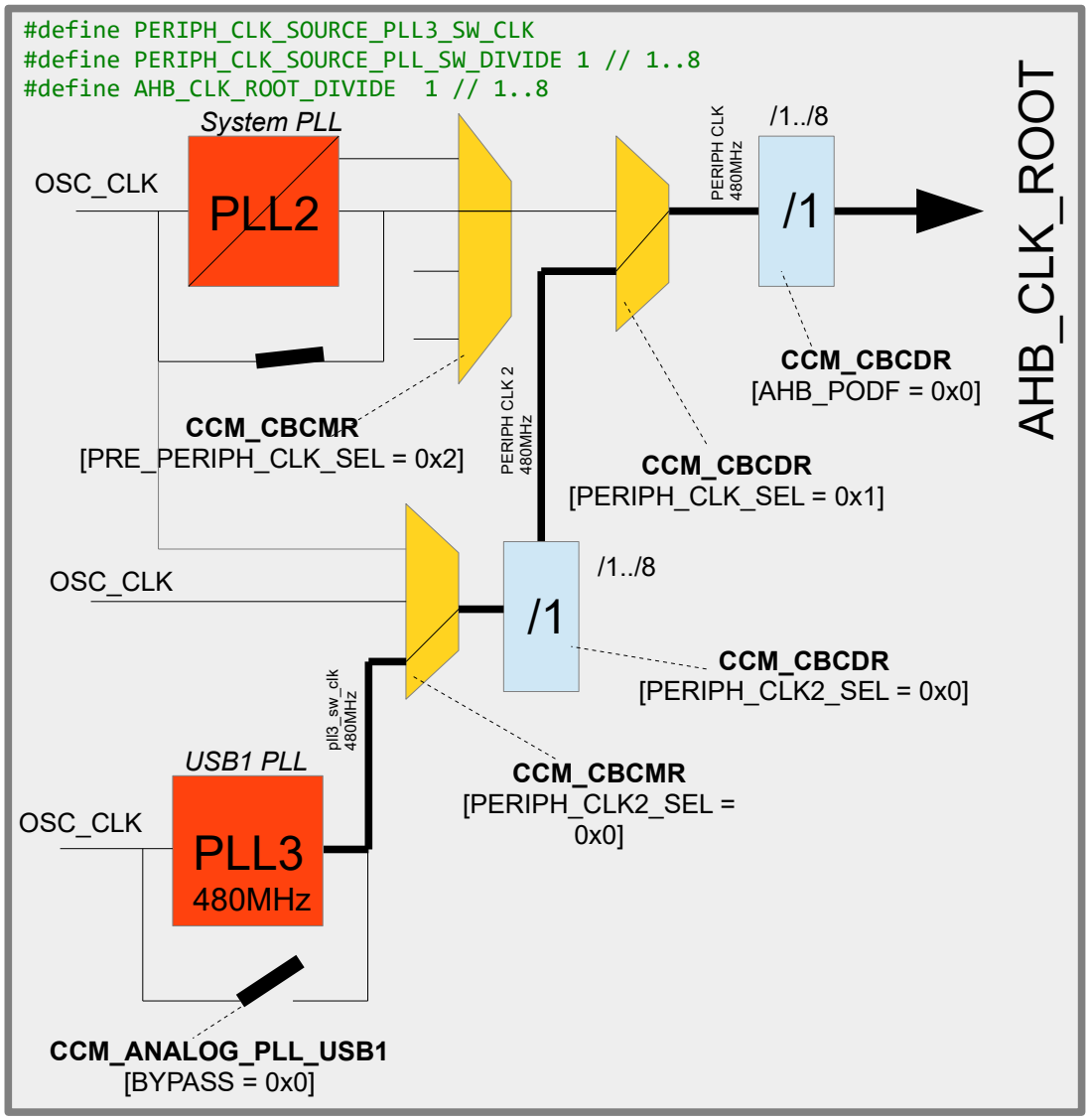
PERIPH_CLK_SOURCE_OSC_DIVIDE, which allows further reduction of the core frequency if needed. If not defined the divider defaults to 1.

```
#define PERIPH_CLK_SOURCE_PLL6_500M
```



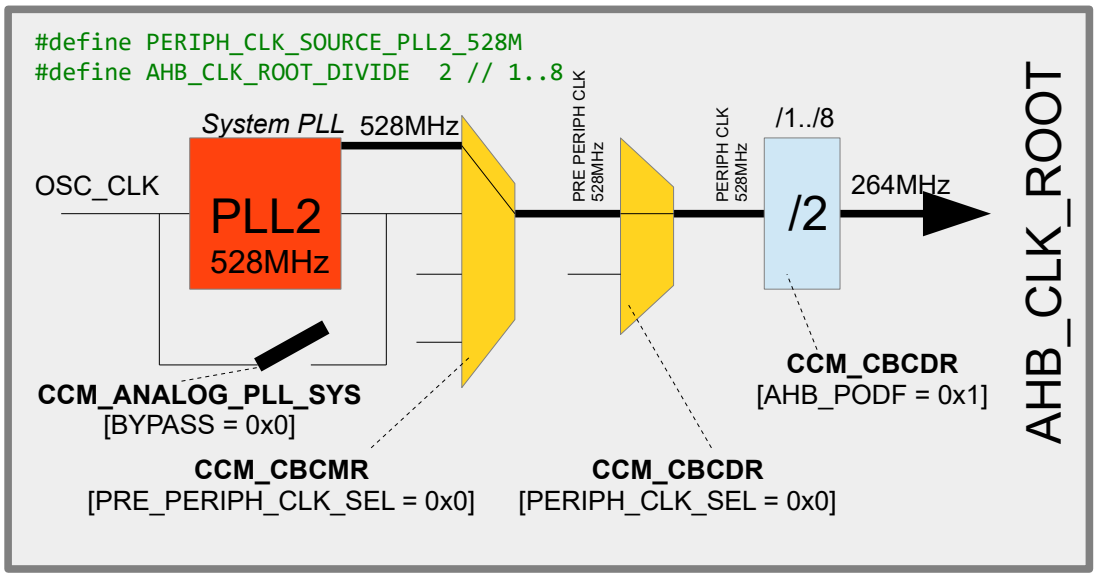
This configuration is useful for simply obtaining the maximum operating frequency of the i.MX RT 1021 by using the output of the fixed 500MHz ENET PLL. When used, the ENET PLL is powered up, lock is waited for and then its bypass is removed so that the clock can be routed to the core *[when Ethernet is used this PLL is also required for its operation]*. In addition to the AHB_CLK_ROOT divider PERIPH_CLK_SOURCE_PLL6_DIVIDE can optionally be defined to pre-scale the PLL output by 1..8 (when not defined the default is 1).

```
#define PERIPH_CLK_SOURCE_PLL3_SW_CLK
```



This configuration uses the USB1 PLL (PLL3) as reference. This is a 480MHz PLL that can be additionally pre-scaled by 1..8 by using the define `PERIPH_CLK_SOURCE_PLL6_DIVIDE`. If not defined the pre-scaler defaults to 1.
As in the case of the other PLLs this configuration causes the PLL to be powered, lock waited for and then its bypass removed so that the signal can be routed to the core.

```
#define PERIPH_CLK_SOURCE_PLL2_528M
```



This configuration uses the System PLL (PLL2) as reference. This is a fixed 528MHz PLL and, as in the case of the other PLLs, this configuration causes the PLL to be powered, lock waited for and then its bypass removed so that the signal can be routed to the core.

It is to be noted that 528MHz is beyond the specification of AHB_CLK_ROOT and so an AHB_CLK_ROOT pre-scaler divide of at least 2 is needed!

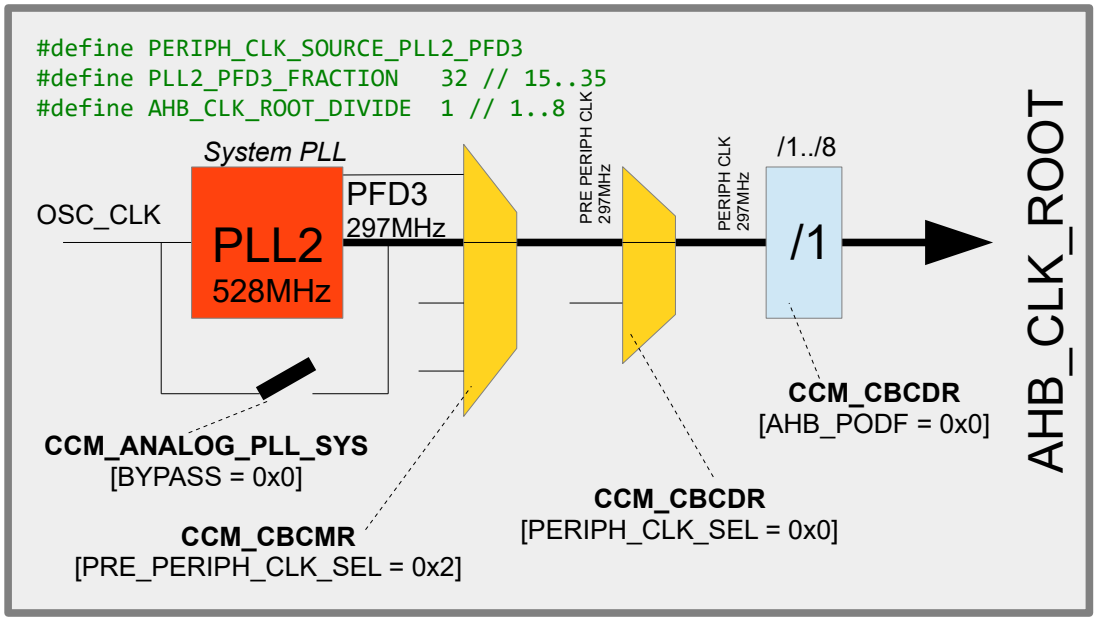
Note further that, although the system PLL is usually used at 528MHz it can in fact be tuned over the range 528MHz..552MHz

Very fine adjustments can be made and the optional define

```
#define PLL2_FREQUENCY                          528123456
```

causes the clock configuration to be performed to give the required frequency rather than the default one.

```
#define PERIPH_CLK_SOURCE_PLL2_PFD3
```

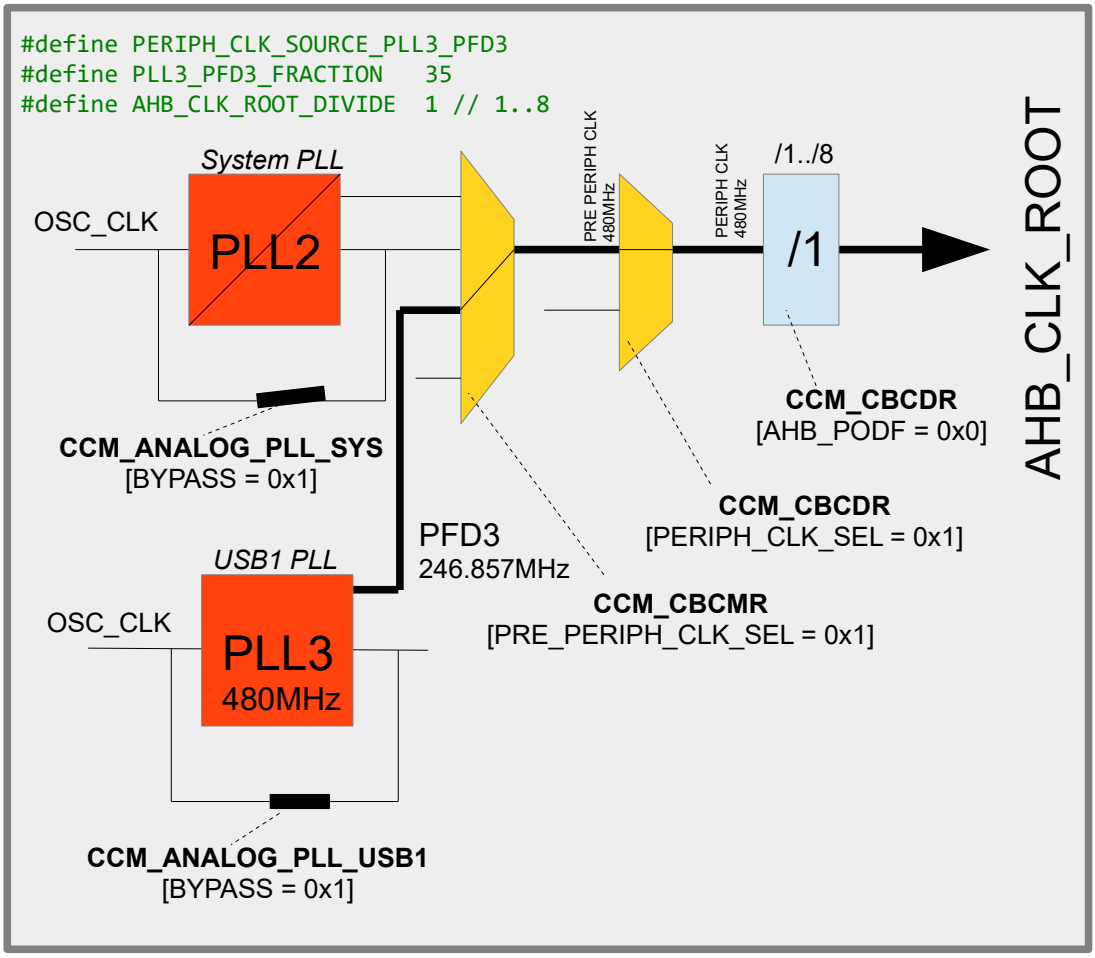


This configuration uses the System PLL (PLL2) as reference. This is a fixed 528MHz PLL but its PFD3 output is sourced instead.

The PFD3 frequency is calculated by ((528MHz * 18) / PLL2_PFD3_FRACTION) and so the illustrated value of 32 results in 297MHz. The list of possible PLL2-PFD3 frequencies can be found in the introduction to this chapter.

As in the case of the other PLLs, this configuration causes the PLL to be powered, lock waited for and then its bypass removed so that the signal can be routed to the core.

```
#define PERIPH_CLK_SOURCE_PLL3_PFD3
```



This configuration uses the USB1 PLL (PLL3) as reference. This is a fixed 480MHz PLL but its PFD3 output is sourced instead.

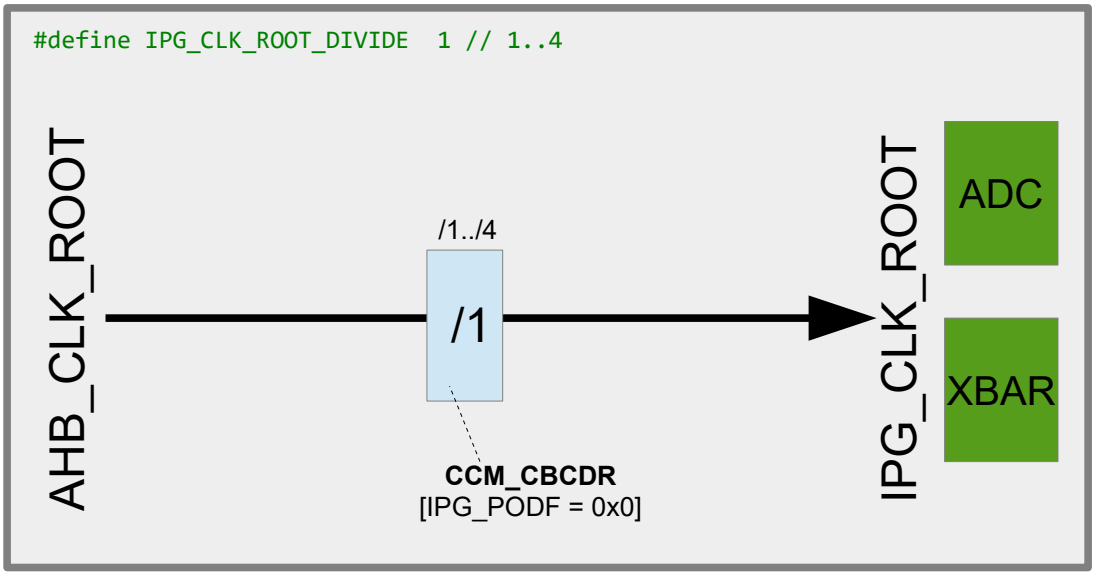
The PFD3 frequency is calculated by ((480MHz * 18) / PLL3_PFD3_FRACTION) and so the illustrated value of 35 results in 246.857MHz. The list of possible PLL3-PFD3 frequencies can be found in the introduction to this chapter.

As in the case of the other PLLs, this configuration causes the PLL to be powered and lock waited for so that the stable signal can be routed to the core.

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 control code.*

1.2. IPG Clock – used for the access of most peripherals

IPG_CLK_ROOT is an internal clock that is required to access most peripherals. It is derived exclusively from the AHB_CLK_ROOT, which was configured in the initial step.
IPG_CLK_ROOT is equal to AHB_CLK_ROOT divided by 1 to 4.




```
#define IPG_CLK_ROOT_DIVIDE 1                    // 1..4
```

is used to configure this ratio, whereby if the define is not used it defaults to 1 (that is, IPG_CLK_ROOT is equal to AHB_CLK_ROOT).

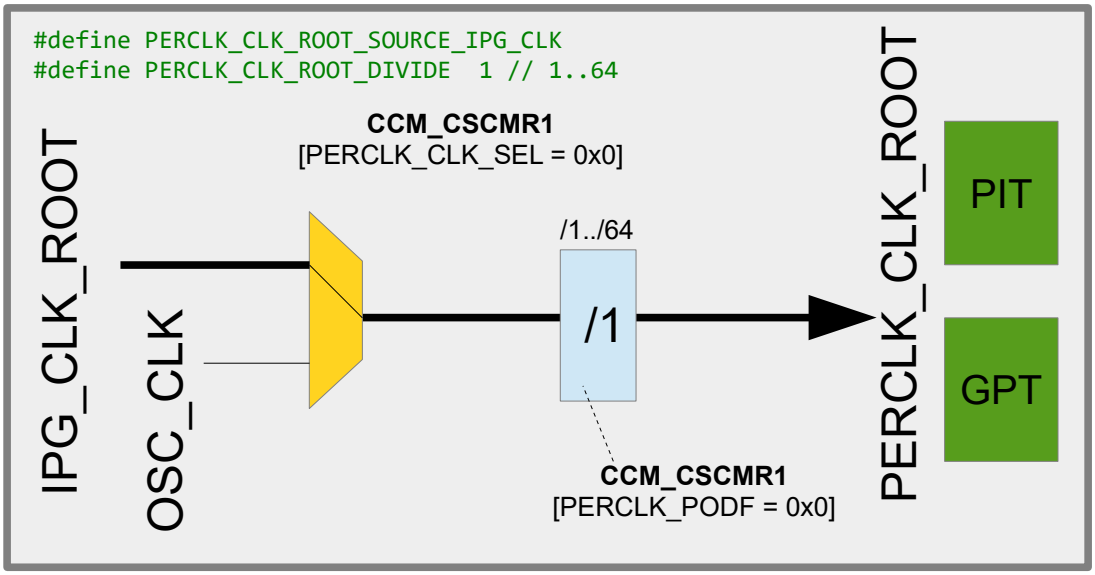
The maximum IPG_CLK_ROOT frequency for the i.MX RT 1021 is 150MHz (max. for AHB_CLK_ROOT is 500MHz) and so it is important to ensure that the divider is set to ensure this speed is not exceeded.

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.

1.3. PERCLK – used by PIT and GPT

PERCLK_CLK_ROOT is the internal clock that feeds the PIT and GPT. It is derived either from the IPG_CLK_ROOT frequency, which was configured in the previous step, or from the 24.0MHz OSC_CLK. In each case it has an optional pre-scaler of divide by 1..64.

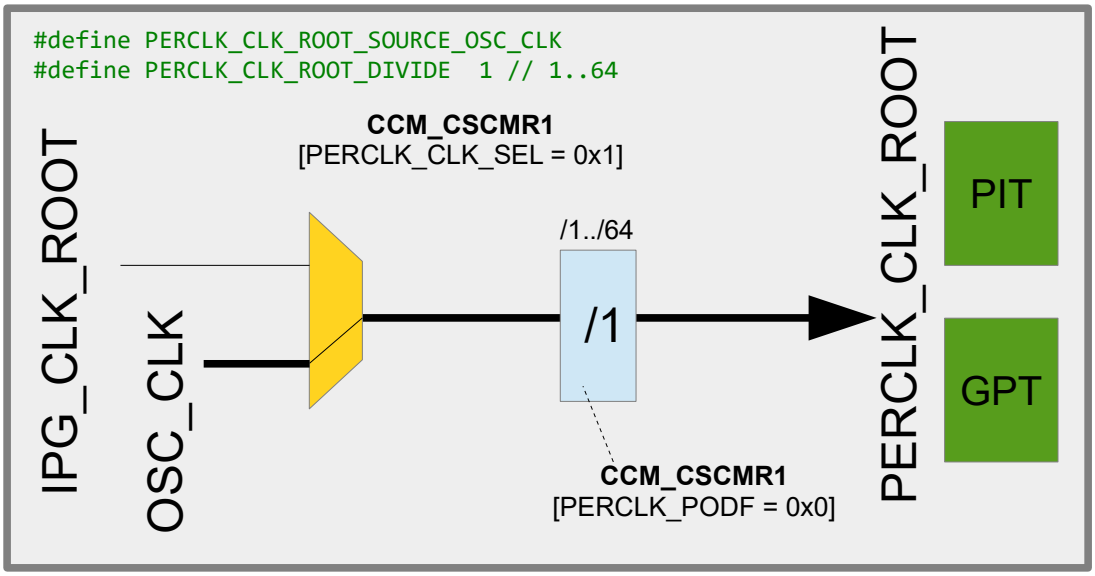
```
#define PERCLK_CLK_ROOT_SOURCE_IPG_CLK
```




```
#define PERCLK_CLK_ROOT_DIVIDE 1 // 1..64
```

is used to configure this ratio, whereby if the define is not used it defaults to 1.

```
#define PERCLK_CLK_ROOT_SOURCE_OSC_CLK
```



Should neither the PIT nor the GPT be used in the project this clock will automatically be disabled by the control code by disabling its clock gate.

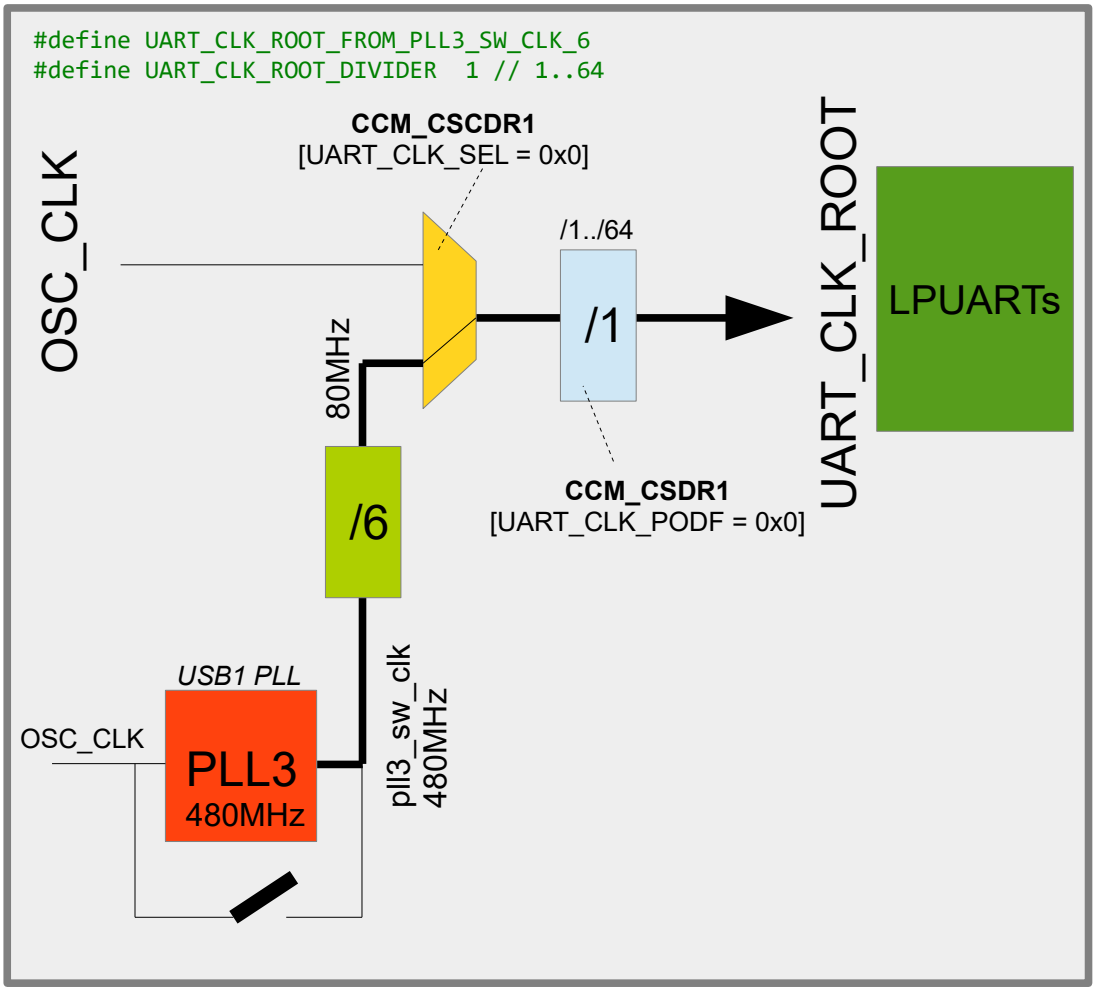
The maximum `PERCLK_CLK_ROOT` frequency for the i.MX RT 1021 is 75MHz and so it is important to ensure that the divider is set to ensure this speed is not exceeded.

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

1.4. UART_CLK_ROOT – used by all LPUARTs

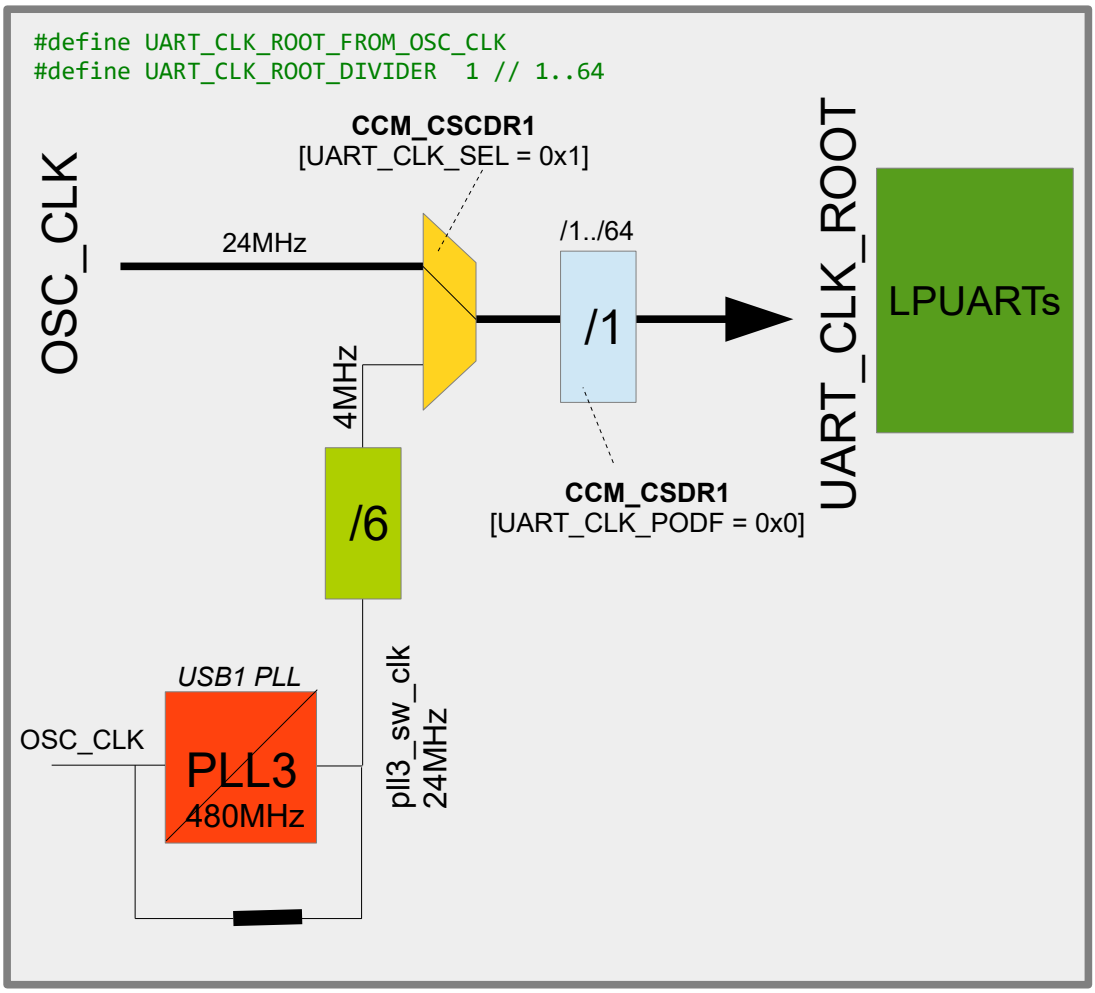
The LPUARTs in the i.MX RT 1021have a common clock that can be derived from either `OSC_CLK` or from `pll_sw_clk/6`. A common pre-scaler allows the frequency to be divided by 1..64.

```
#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
```



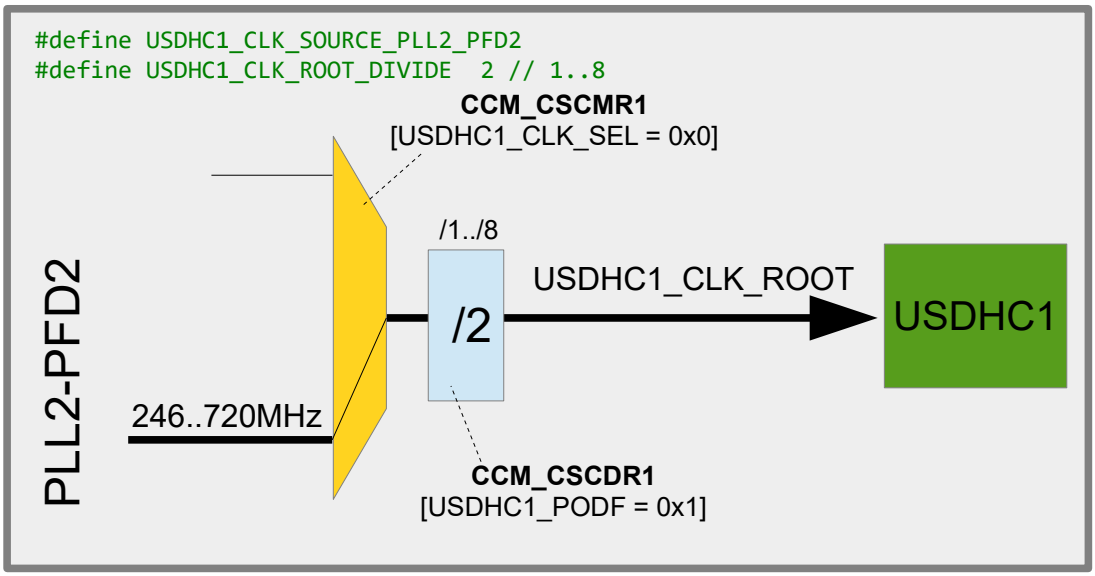
In this configuration the OSC_CLK (24MHz) is used as reference, with the optional pre-scaler of 1 to 64.

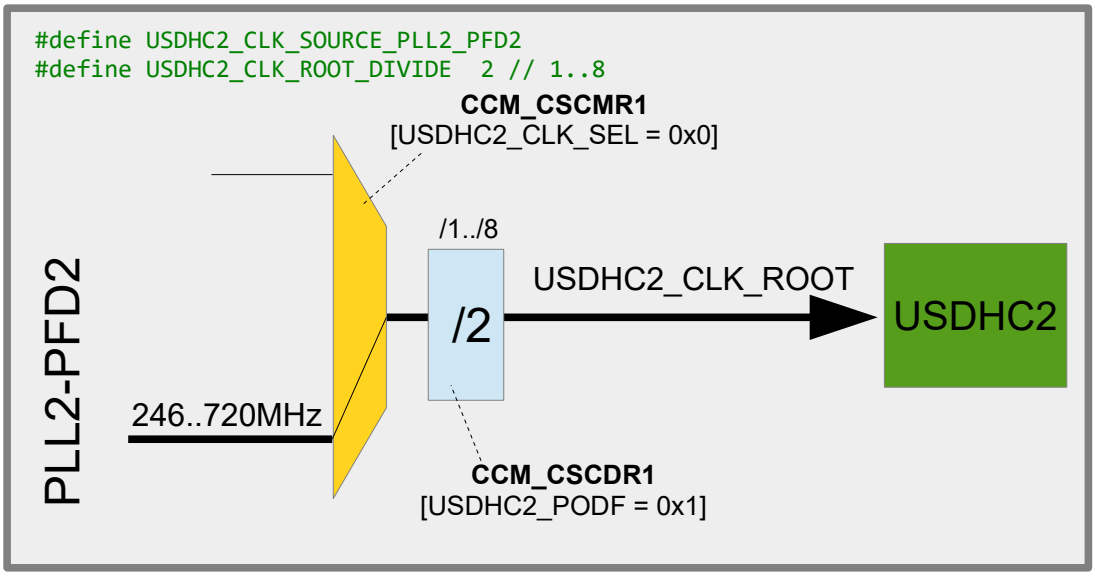
Note that when PLL3 is not enabled it is left in its powered down, bypassed mode and the alternative clock would be 4MHz instead of 80MHz. This does in fact correspond to the default for the LPUART clock out or reset but this is not used as configuration option since it has no advantage.

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

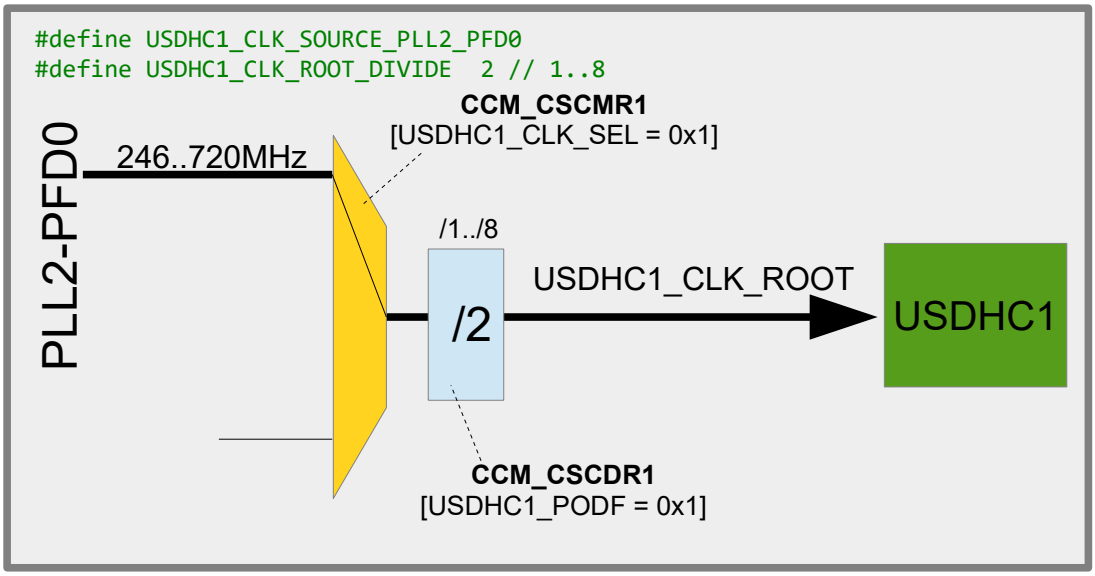
1.5. USDHC1_CLK_ROOT/USDHC2_CLK_ROOT

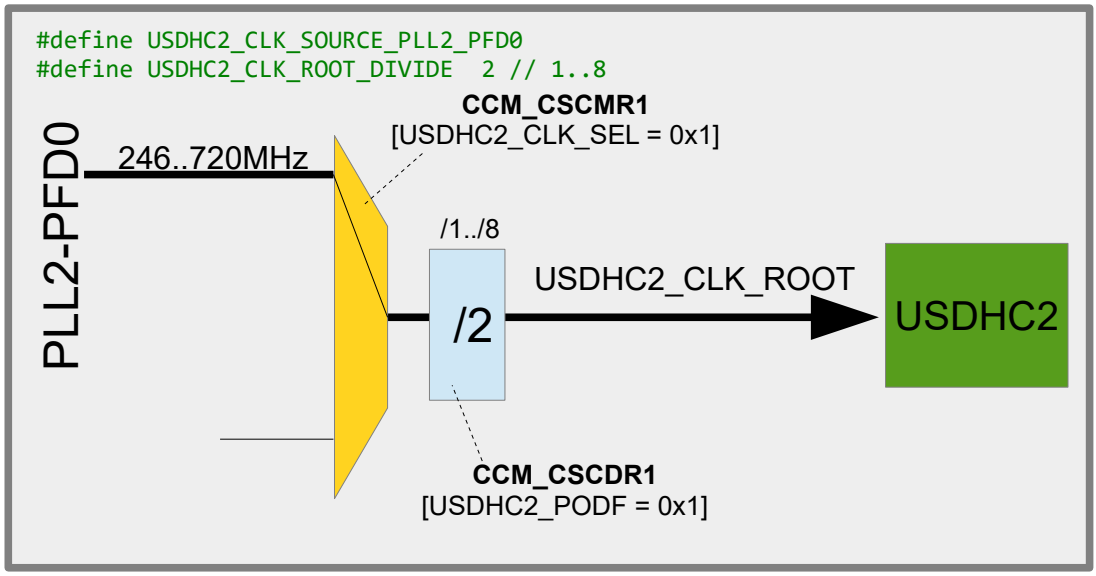
By default the `USDHC1_CLK_ROOT` and `USDHC2_CLK_ROOT` are both sourced from PLL2-PFD2, which has a default frequency of 396MHz when PLL2 is enabled. A 3-bit pre-scale divider reduces this to half the value.





The alternative clock root source is PLL2-PFD0.



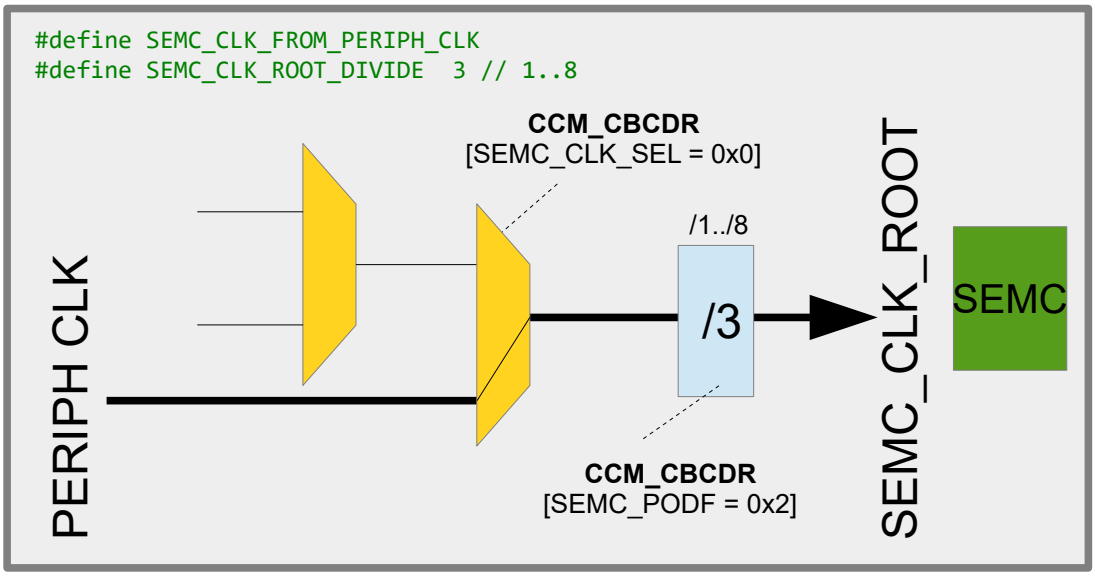


When an USDHC controller is used PLL2 is unconditionally used because it needs to be in operation.

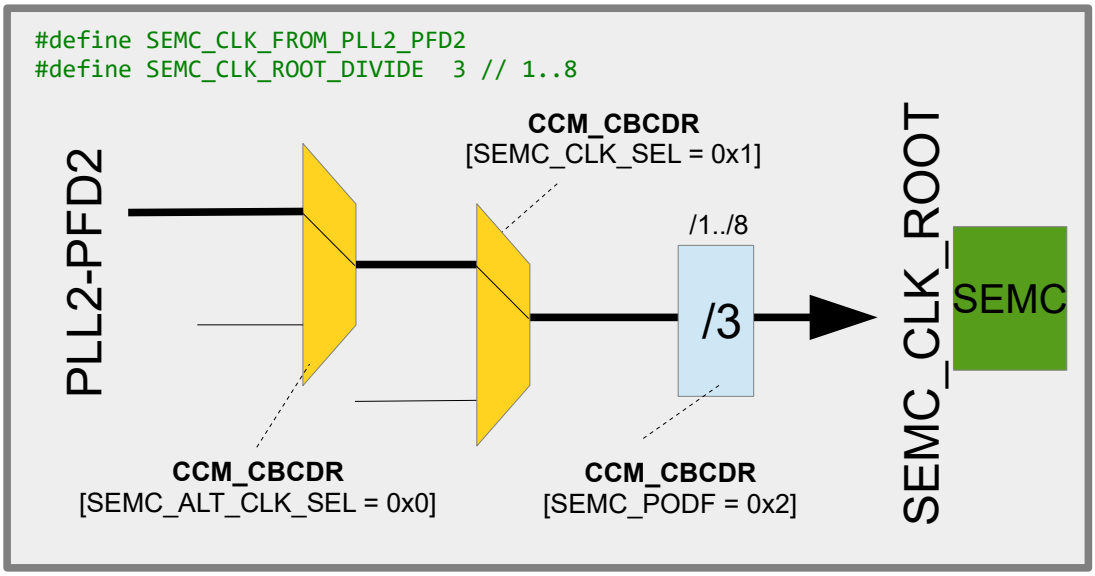
The `USDHCx_CLK_ROOT` frequencies must not exceed 198MHz.

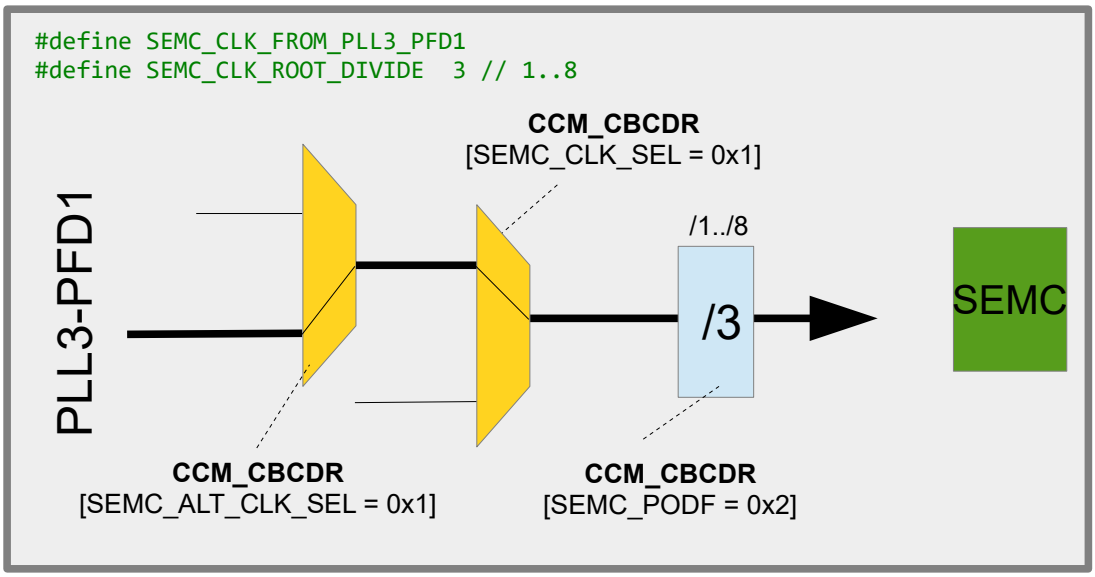
1.6. SEMC_CLK_ROOT – external memory controller clock

By default the SEMC_CLK_ROOT is sourced from PERIPH_CLK, which is the clock that supplies the core clock's pre-scaler. Out of reset it is divided by 3. This configuration is shown in the first image:



Other configuration possibilities are for PLL2-PFD2 or PLL3-PFD1 as shown in the further images.

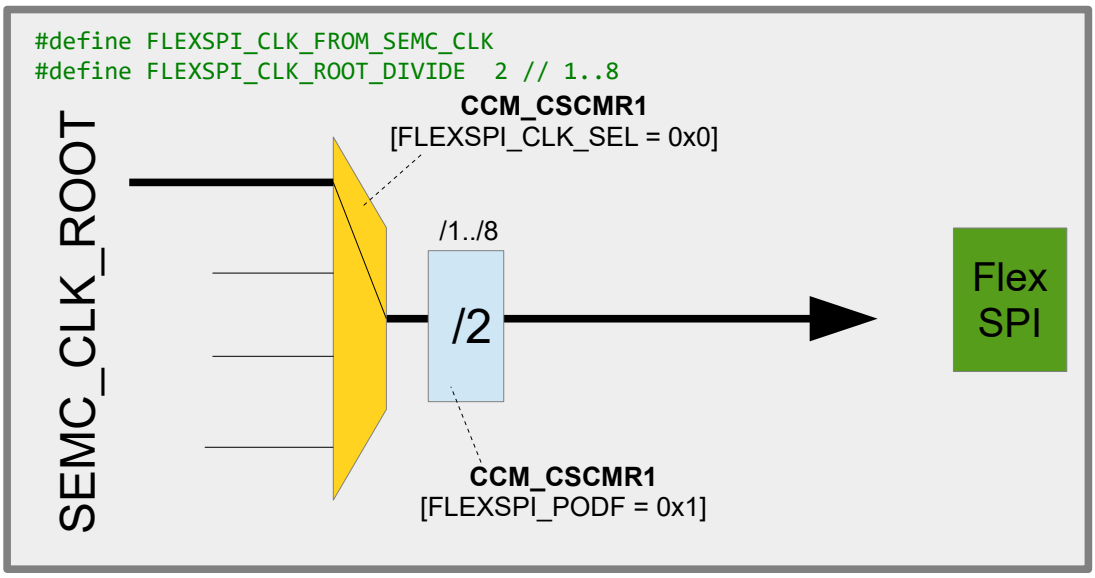


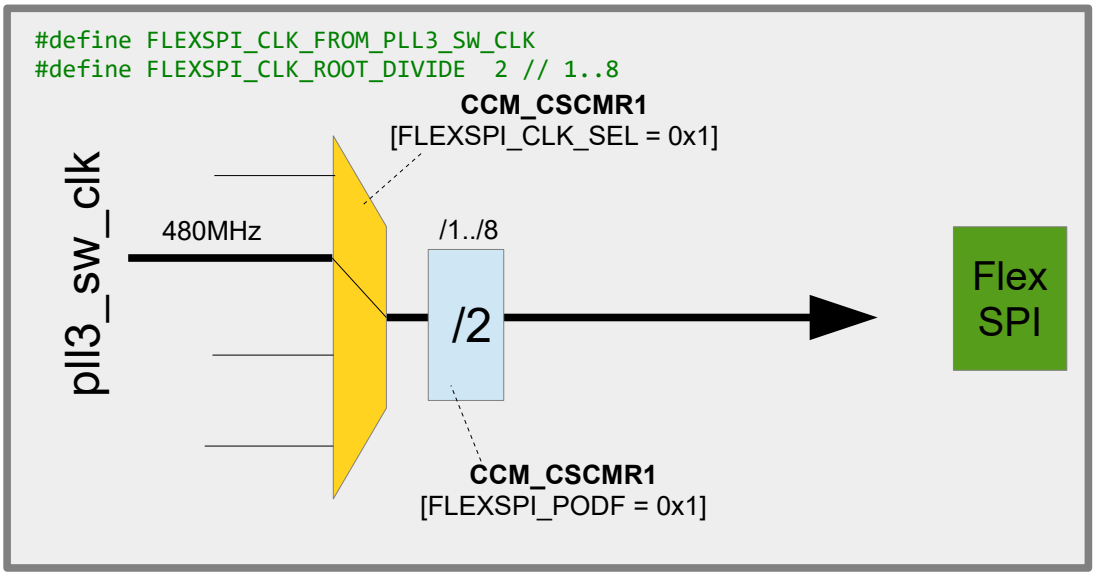


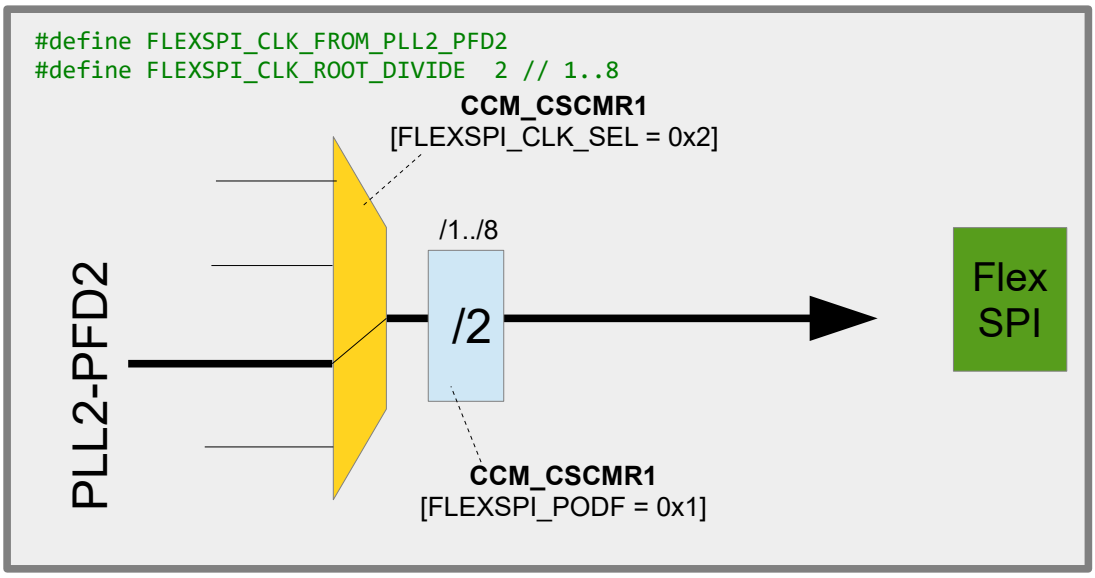
The SEMC_CLK_ROOT frequency must not exceed 166MHz.

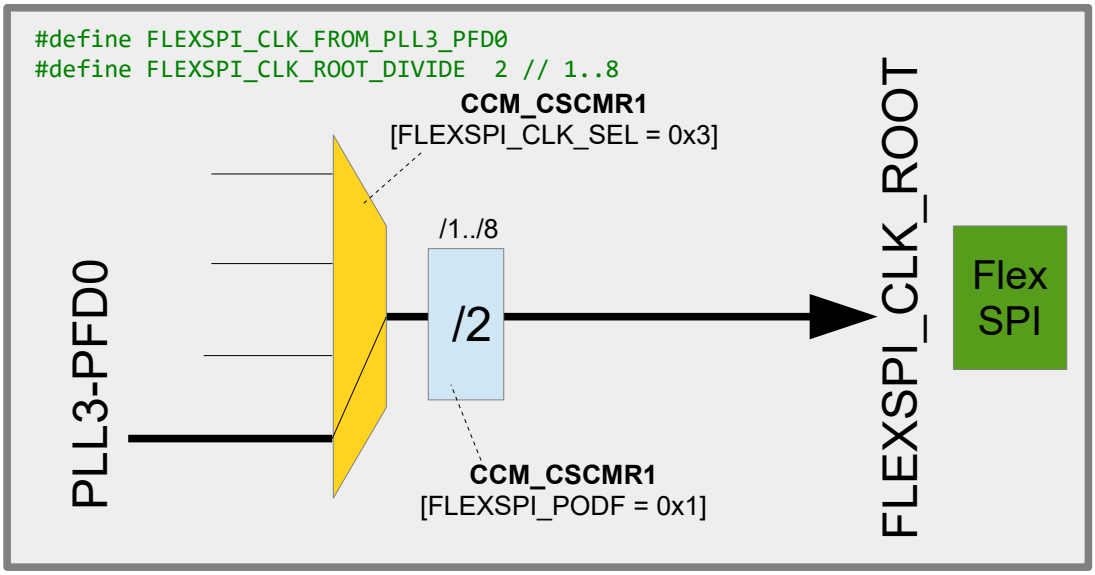
1.7. FLEXSPI_CLK_ROOT

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.





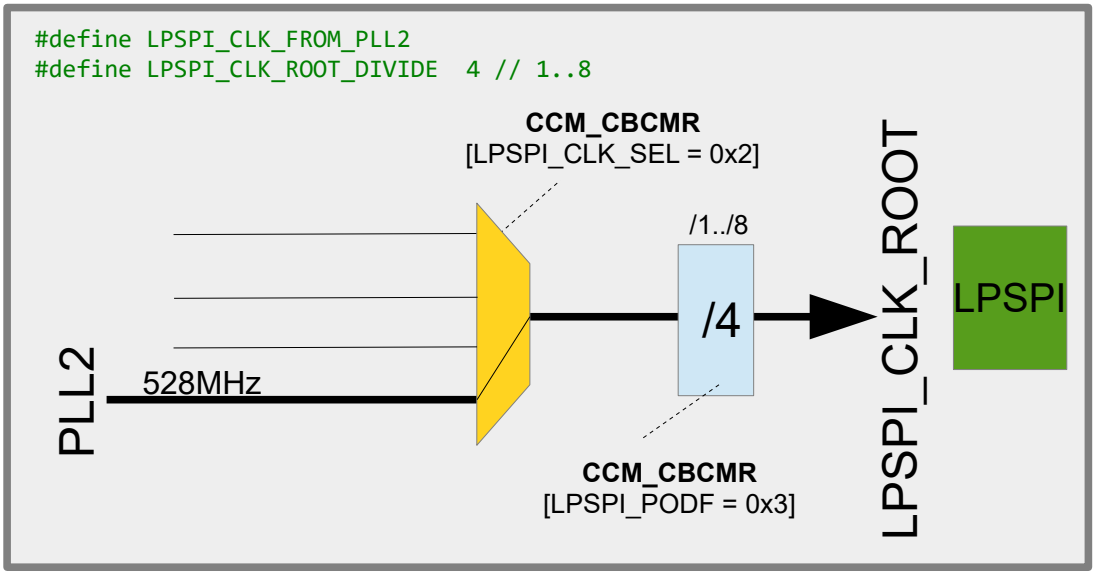

#define FLEXSPI_CLK_FROM_PLL2_PFD2
#define FLEXSPI_CLK_ROOT_DIVIDE 2 // 1..8
CCM_CSCMR1
[FLEXSPI_CLK_SEL = 0x2]
/1../8
/2
PLL2-PFD2
Flex SPI
CCM_CSCMR1
[FLEXSPI_PODF = 0x1]


#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]

1.8. LPSPI_CLK_ROOT – used by all LPSPIs

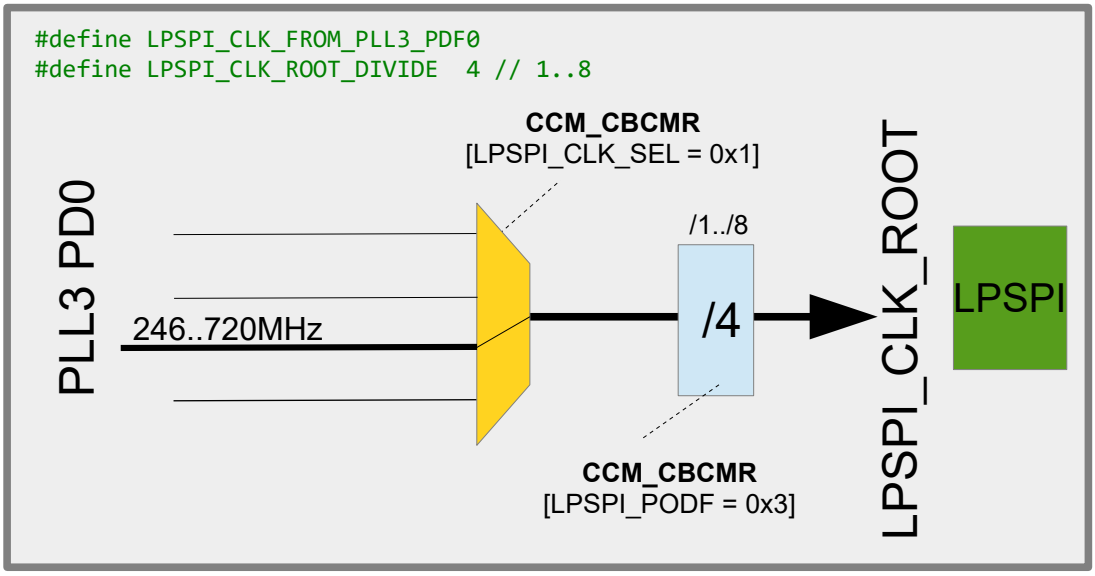
The LPSPIs in the i.MX RT 1021have a common clock that can be derived from either `PLL2` , `PLL2 PFD2`, `PLL3 PFD0` or `PLL3 PFD1`. A common pre-scaler allows the frequency to be divided by 1..8.

```
#define LPSPI_CLK_FROM_PLL2
```

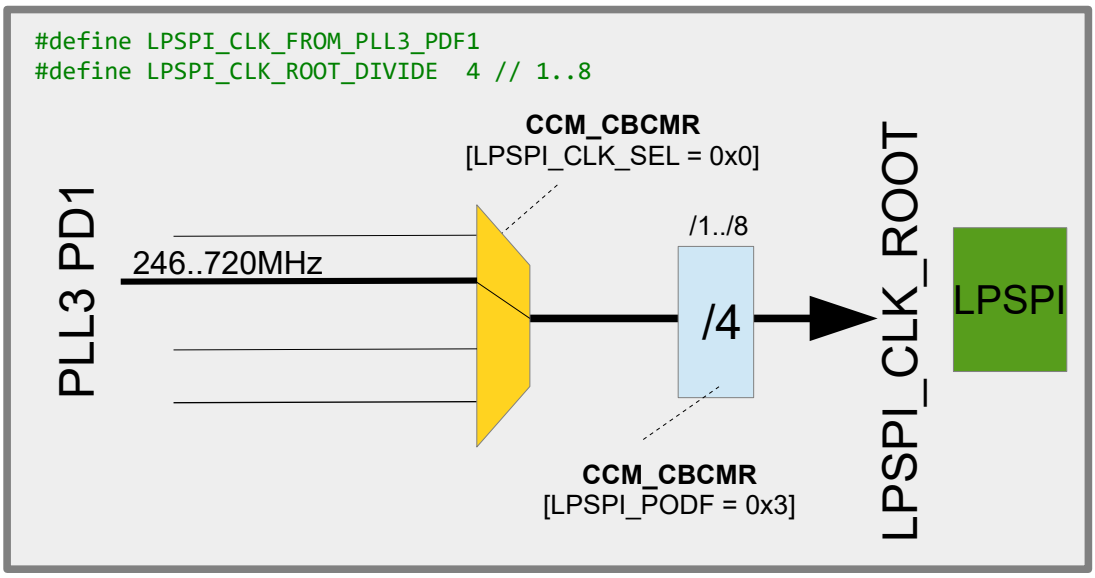


If PLL2 were to be bypassed the source would be 24MHz rather than 528MHz.

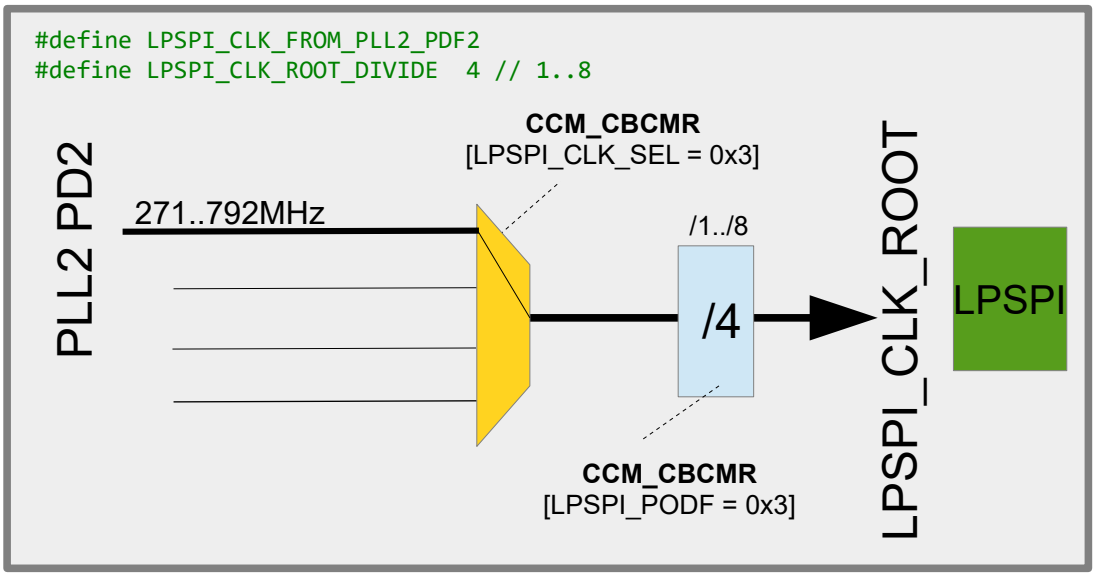
```
#define LPSPI_CLK_FROM_PLL3_PDF0
```



```
#define LPSPI_CLK_FROM_PLL3_PDF1
```




```
#define LPSPI_CLK_FROM_PLL2_PDF2
```



`LPSPI_CLK_ROOT` frequency must not exceed 132MHz.

When the use of a PLL or their phase fractional dividers is selected the PLL and necessary outputs are enabled as required, otherwise they are left in power down states when also not required for other clock roots.

`LPSPI_CLK_ROOT` supplies all LPSPIs but the clock is automatically disabled at each individual LPSPI input when the corresponding LPSPI is not used.

It is to be noted that the LPSPI timing is configured further in the LPSI module itself and in fact there is a minimum divide of 2, meaning that the actual LPSPI output clock speed is half of LPSPI_CLK_ROOT at least.

1.9. TRACE_CLK_ROOT

To add..

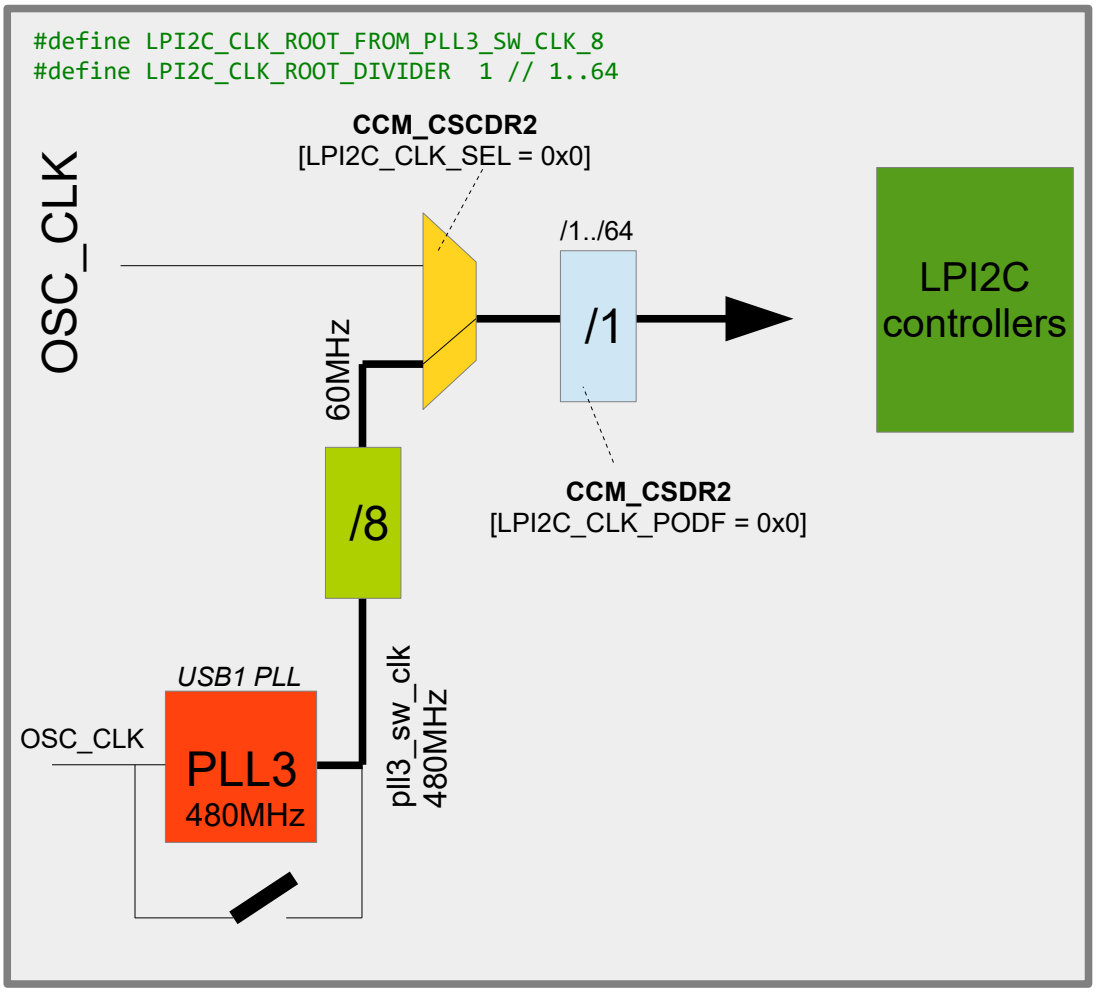
1.10. SAI1_CLK_ROOT/SAI2_CLK_ROOT/SAI2_CLK_ROOT

To add..

1.11. LPI2C_CLK_ROOT – used by all LPI2C controllers

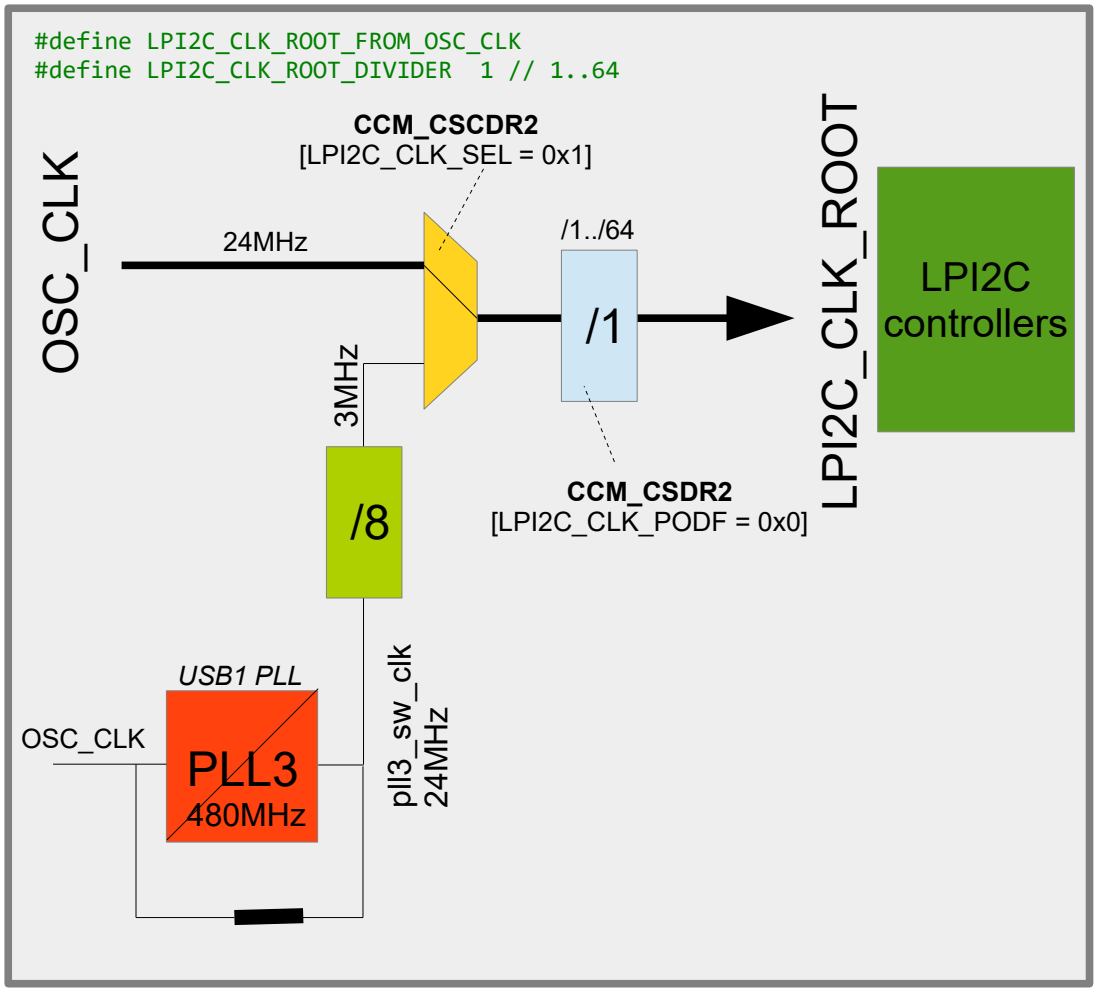
The LPI2C controllers in the i.MX RT 1021 have a common clock that can be derived from either `OSC_CLK` or from `pll_sw_clk/8`. A common pre-scaler allows the frequency to be divided by 1..64.

```
#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.

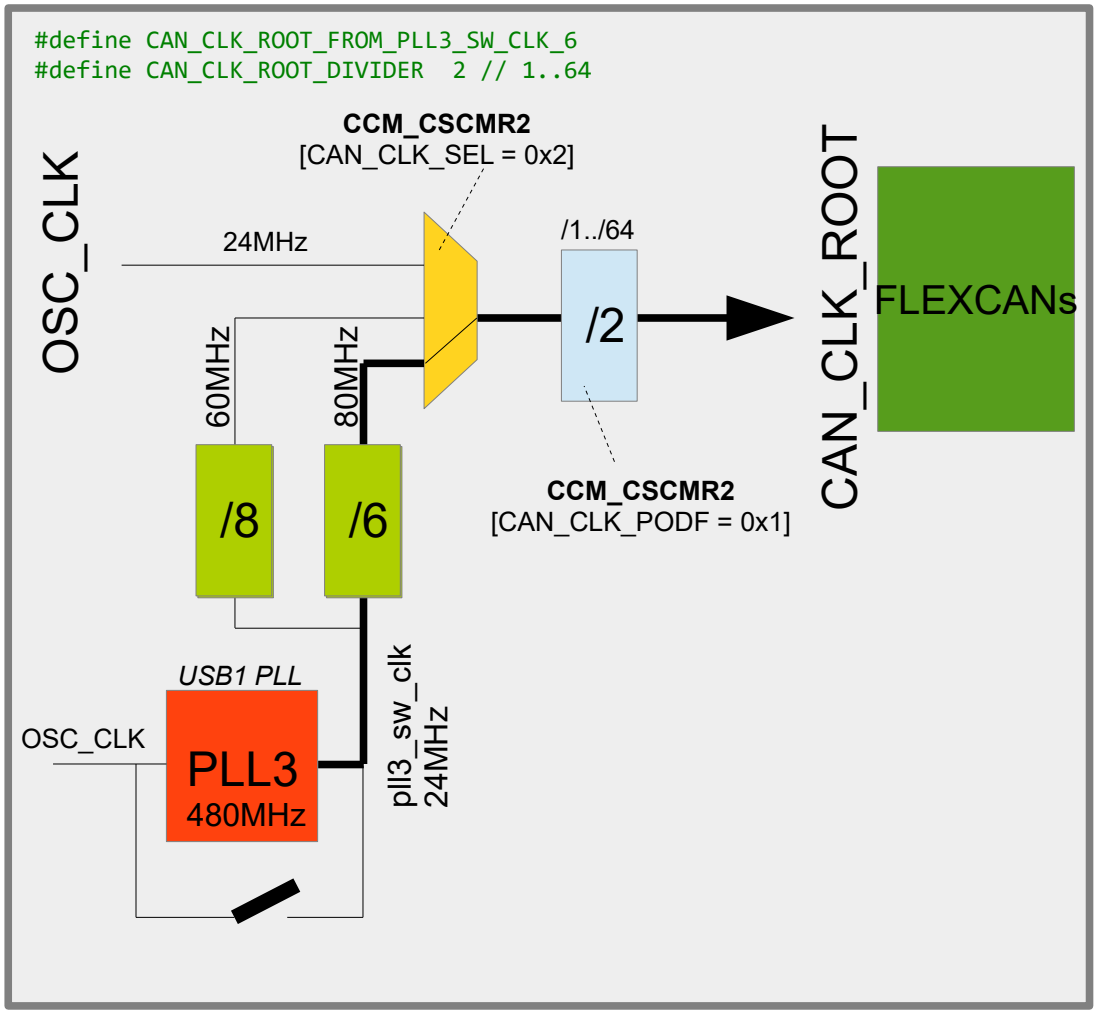
Note that when PLL3 is not enabled it is left in its powered down, bypassed mode and the alternative clock would be 3MHz instead of 60MHz. This does in fact correspond to the default for the LPI2C clock out or reset but this is not used as configuration option since it has no advantage.

`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.

1.12. CAN_CLK_ROOT

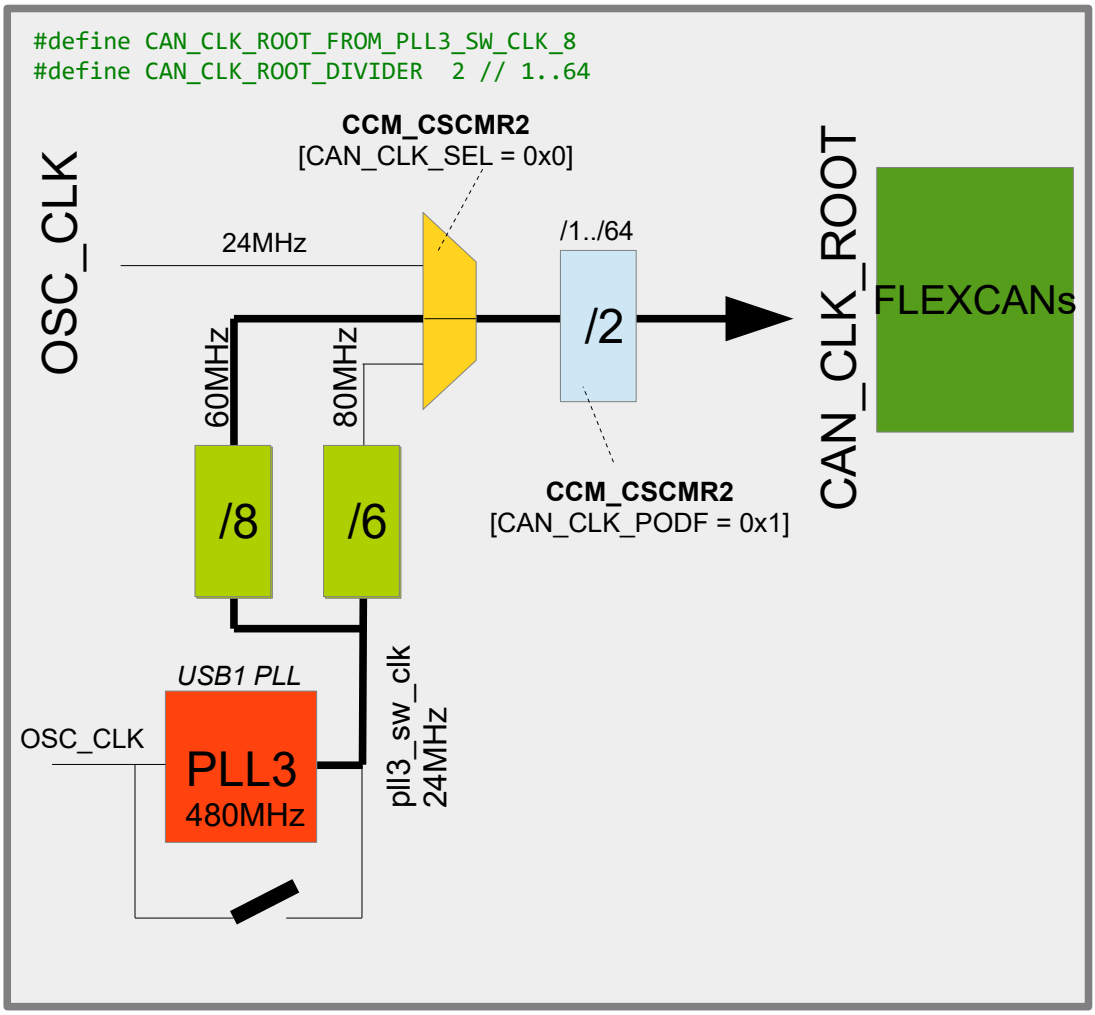
The FLEXCAN controllers in the i.MX RT 1021have a common clock that can be derived from either `OSC_CLK`, `pll_sw_clk/6` or from `pll_sw_clk/8`. A common pre-scaler allows the frequency to be divided by 1..64.

```
#define CAN_CLK_ROOT_FROM_PLL3_SW_CLK_6
```



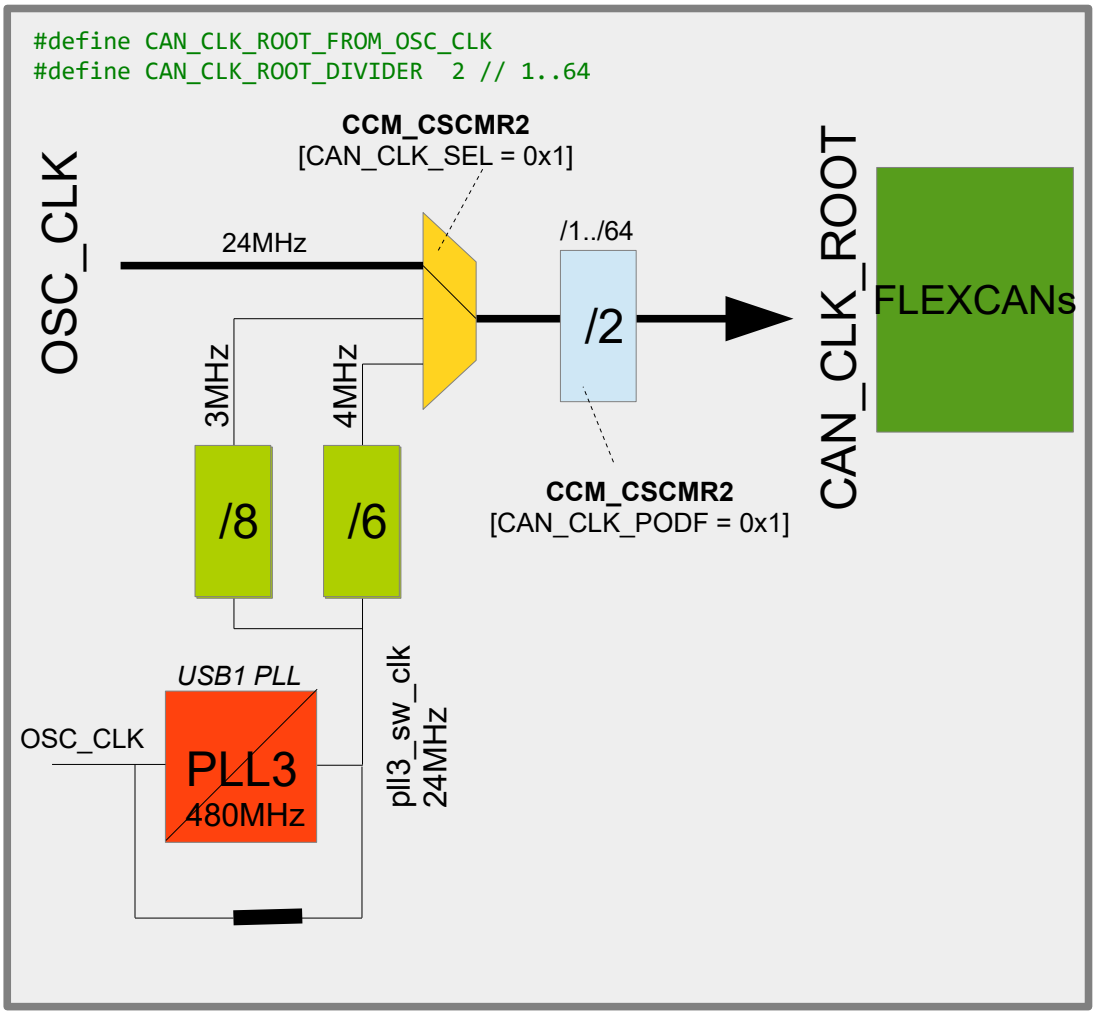
In this configuration the USB1 PLL is used as reference (called `pll_sw_clk`) and divided by a fixed value of 6, resulting in 80MHz. An optional pre-scaler defined by `CAN_CLK_ROOT_DIVIDER` can divide this by 1 to 64 (when not defined, the default is 2)

```
#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.

1.13. SPDIF0_CLK_ROOT

To add..

1.14. FLEXIO1_CLK_ROOT

To add..

1.15. USB1 Clock

USB1's PHY requires a 480MHz clock which is derived exclusively from PLL3's fixed 480MHz output – for this reason PLL3 is known also as USB1 PLL. This means that, logically, PLL3 must be operating at 480MHz for USB1 to be used.

When USB is enabled with

```
#define USB_INTERFACE
```

it is automatically configured as part of the clock initialisation so that it is subsequently available for use.

3. Real Time Clock

The i.MX RT 1021 has a low power RTC in its SNVS (Secure Non-Volatile Storage Module) which can be powered by a coin cell (2.4V .. 3.6V) on its VDD_SNVS_IN pin. Although an internal ring oscillator can be used to supply a rough 32kHz clock to the module higher accuracy time keeping is achieved by using a 32kHz crystal connected to the RTC_XTALI and RTC_XTALO pins.

4. 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 internal 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
```

5. 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.

6. 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.

7. LPSPI

The i.MX RT 1021 LPSPI driver is shared with the Kinetis LPSPI 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 LPSPI 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.

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_LPSPI_1
iMX_LPSPI_2
iMX_LPSPI_n
```

instead of channel numbers, whereby `iMX_LPSPI_1` is in fact 0.

SPI Flash drivers originally available for Kinetis can also be used by i.MX RT – these are located in `\Hardware\SPI_Memory` and include drivers for a number of popular parts.

8. 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.

9. USDHC

The i.MX RT 1021 USDHC driver is shared with the Kinetis SDHC 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 SDHC 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 utFAT user's manual for general details of FAT12/FAT16 and FAT32 on SD cards:
https://www.utasker.com/docs/uTasker/uTasker_utFAT.PDF

10. 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).

11. 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.

12. 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.

13. RAM and Cache

The i.MX RT 1021 contains 256k of internal RAM which is constructed of 8 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 16kBytes 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

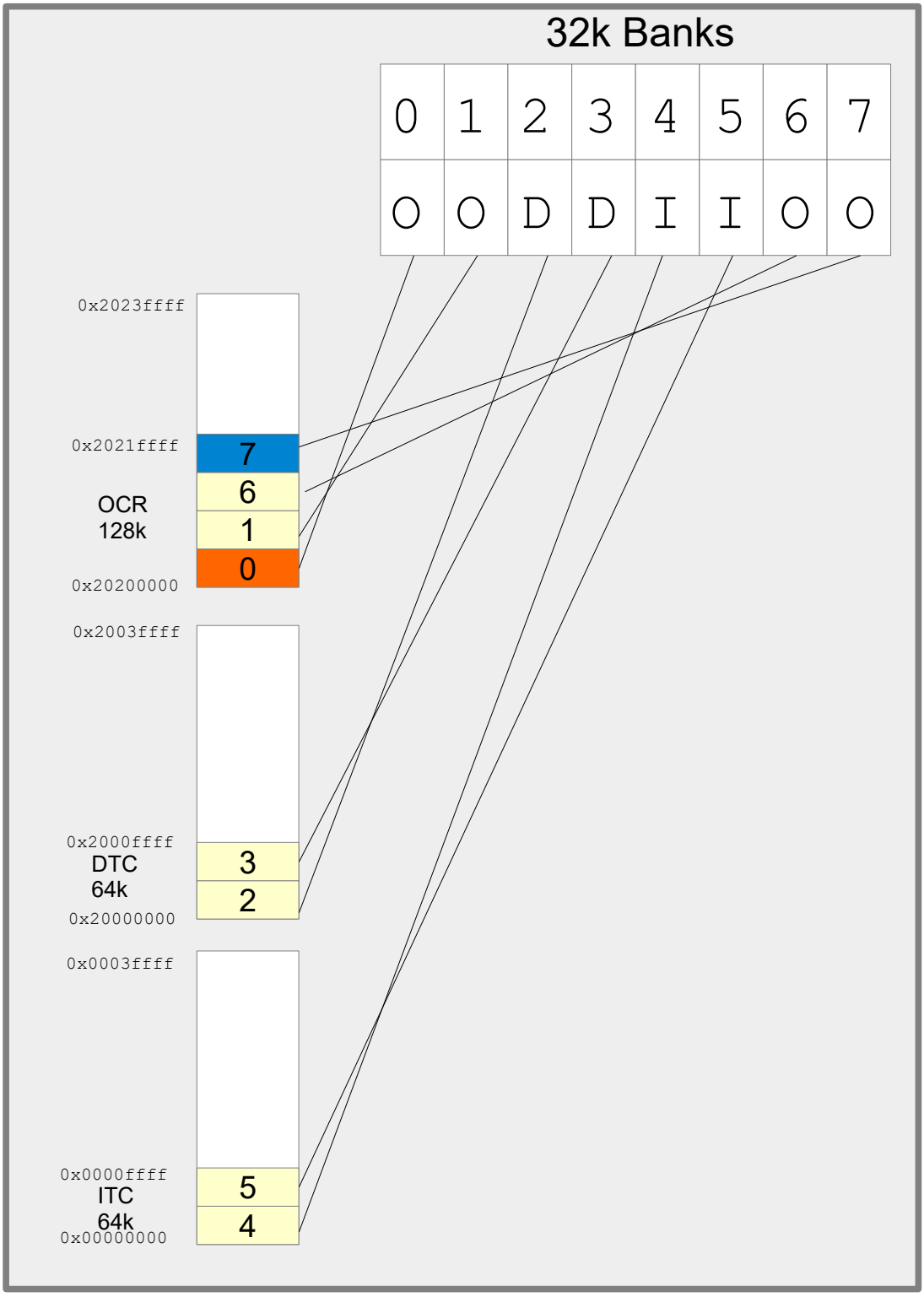
```
#define ENABLE_INSTRUCTION_CACHE
```

and

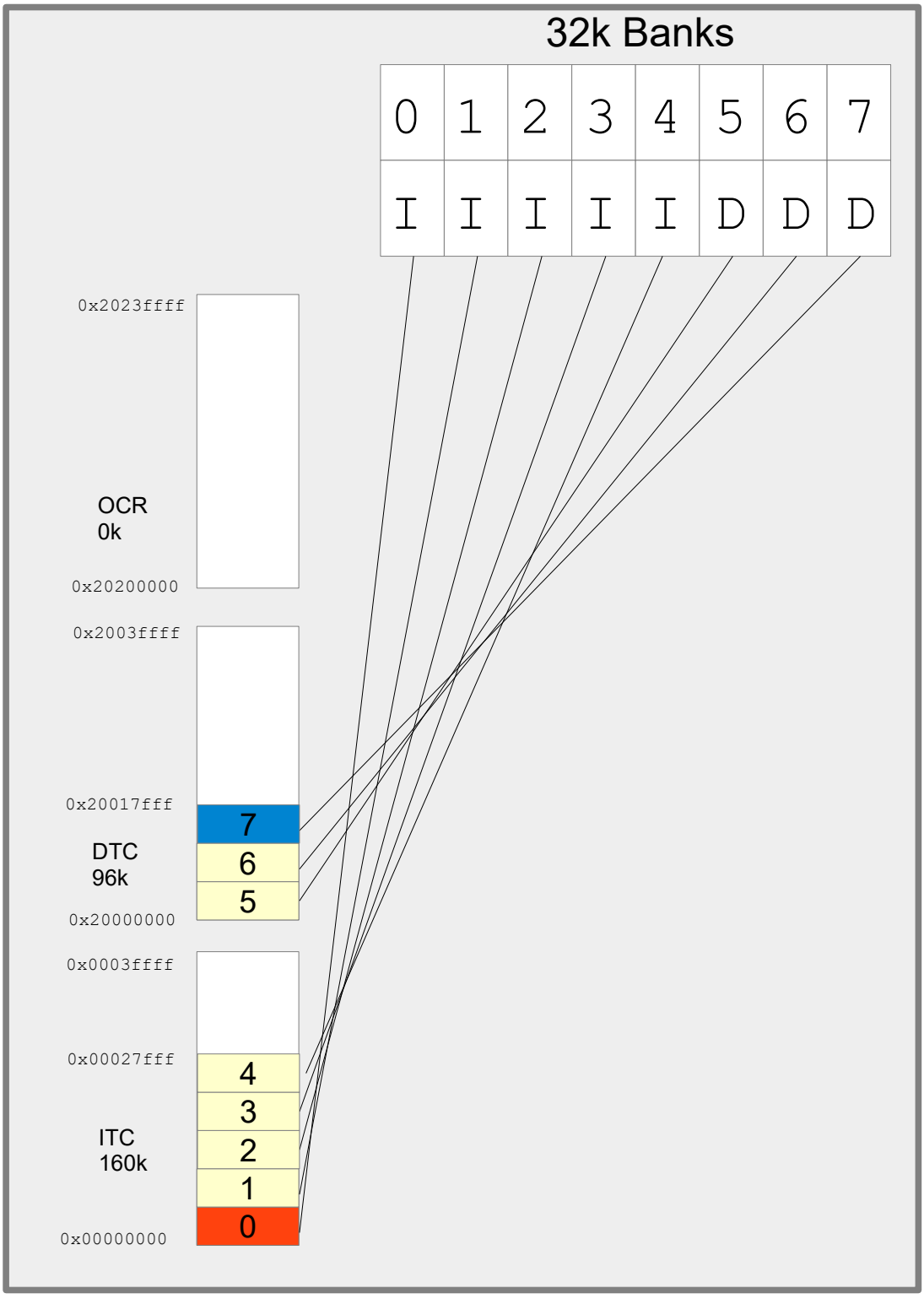
```
#define ENABLE_DATA_CACHE
```

The FlexRAM controller configures the RAM banks at reset based on eFuse settings. The standard setting (when nothing else has been programmed) is for 128k OCRAM, 64k DTCM and 64k 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 5 banks for ITCM (so that the code could be completely located there – 160k) and 3 banks for data (so that up to 96k of data could be accesses at optimal speed) and no OCRAM the 8 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 130k were encountered it would thus assign 160k ITC and 96k 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 7 is assigned to OCR but will not be used by the ROM loader (the ROM loader may use up to 64k 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 1021, 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).

14. 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/GPIO2-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 (QSPI) 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 µTasker boot loader strategy flow diagram is show at https://www.utasker.com/docs/iMX/Loader.pdf whereby it automatically supplies a complete secure and clone-protected solution for serial loading and fail-safe OTA (Over-the-Air) updates.
The serial loader can be configured to support one or more of a number of loading methods such a USB, UART, Ethernet, and with existing protocols such as KBOOT.
It allows for running applications in internal SRAM (for maximum efficiency) or directly form QSPI flash.

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. The “Fall-back” serial loader allows updates of the serial loader itself and provides recovery even when non-operational serial loaders were to be installed.

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

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).

The µTasker boot loader concept is discussed ad demonstrated in the video:
https://youtu.be/2XfgZq19XDw

15. Total Security and Simplicity

The µTasker concept allows an optimal combination of not just performance and power consumption but also code security. Rather than IP (intellectual property) protection being an after-thought it is already fully implemented so that the user hardly needs to be aware of it being in use.

The “Bare-Minimum” boot loader operates from QSPI flash but is stored in AES128 encrypted form. By using the chip's Bus Encryption Engine (BEE) its content is decrypted on-the-fly so that the “BM” loader - and the secret keys that it may store - are protected from being read in a plain text form directly from the QSPI flash memory.

The serial loader is (optionally) stored in an AES256 encrypted form in QSPI memory and can thus not be read in a plain text form. The “Bare-Minimum” boot loader performs AES256 decryption when in copies it to its RAM execution location. Since there are no subsequent accesses to the QSPI flash the BEE can be powered down as soon as the serial loader has been started. The serial loader then runs at full speed in tightly coupled instruction memory and has no power consumption overhead due to either QSPI access nor BEE operation.

The application is (optionally) saved in an AES256 encrypted form in QSPI memory, whereby the details and benefits are equivalent to those of the serial loader.

When updates are performed over-the-air there are no additional encryption/decryption steps needed (they are simply stored in the format as received since this is already protected both during the transmission and in the storage medium).

The “Bare-minimum” loader, which is responsible for storing the private decryption keys for the particular product and performing the decryption process, also recognises application code that is saved in plain text format and allows it to be executed if required. Developers thus have no addition complication if there is a reason - during development - to temporarily load non-encryped applications.

16. 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`

17. XBAR and AOI

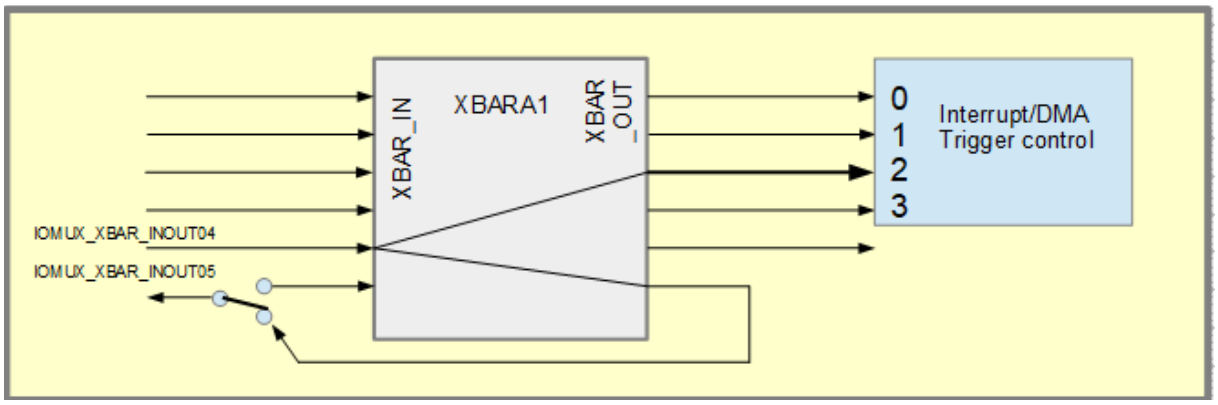
XBAR is a Crossbar multiplexer with a certain number of inputs that can be connected to a certain number of outputs. It allows some processor pins and a variety of peripherals to be connected between each other. For example, a timer peripheral could be connected via the XBAR to the hardware trigger input of an ADC.

AOI is an “And-Or-Inverter” logic array. This allows some XBAR inputs to be logically treated to generate further XBAR inputs that only trigger events when the defined logical states are as programmed, and allows complex control of XBAR outputs beyond that of simply connecting inputs to outputs.

The i.MX RT 1021 has 3 XBARs. XBARA1, XBARB2 and XBARB3. Each of the XBARs have its own array of possible inputs and outputs that can be connected – specifically, the XBARA1 has 88 inputs and 132 outputs; XBARB2 has 58 inputs and 16 outputs; XBARB3 has 58 inputs and 16 outputs, whereby the XBARB2 and 3 outputs are dedicted to use with the AOI.

XBARA1 is special in that its first 4 outputs can be configured to generate interrupts when the change state (rising edge, falling edge or both edges) or they can generate DMA triggers on the same state changes (rising edge, falling edge or both edges).

The following image shows a simple practical example where a pin input (XBAR_INOUT4 is connected to a pin output (XBAR_INOUT5) and also to XBAR_OUT02, which can also be used to generate interrupts or trigger DMA on the defined signal edge:



The AOI has 4 channels and each channel has 4 inputs which can be selected to be from certain XBAR outputs (see later for example).
These 4 inputs of the channel are fed in to 4 x 4-input AND gates, whereby each of the 16 inputs can be individually configured to be set to logical '0', logical '1', the selected input, or an inverted version fo the selected input (eg. If any of the inputs to an individual AND gate were to be set to '0' it effectively disables that AND gate for the logical array).

The outputs of the 4 x 4-input AND gates are then Ored together to generate the final output signal. The output signal can then be selected by the XBAR as an input which it then connects to one of its outputs to be output on a pin, connected to the input of a peripheral or to trigger an interrupt or DMA transfer.

For simplicity of configuration of a AOI channel logic the uTasker project supplies a macro:

```
CONFIGURE_LOGIC_AOI1_EVENT(0,                  // AOI1 instance, channel (0)
(AOI_A_LOW + AOI_B_HIGH + AOI_C_CONNECTED + AOI_D_INVERTED),   // AND 0 inputs
(AOI_A_HIGH + AOI_B_LOW + AOI_C_INVERTED + AOI_D_CONNECTED),   // AND 1 inputs
(AOI_A_CONNECTED + AOI_B_INVERTED + AOI_C_LOW + AOI_D_HIGH),   // AND 2 inputs
(AOI_A_INVERTED + AOI_B_CONNECTED + AOI_C_HIGH + AOI_D_LOW));  // AND 3 inputs
```

This configures AOI1 channe 0 logically as

```
('0' & '1' & InputC & NOT Input D) | ('1' & '0' & NOT InputC & Input
D) | (InputA & NOT InputB & '0' & '1') | (NOT InputA & InputB & '1'
& '0')
```

which makes no sense in a real case since the output would always be '0', but shows all of the input controls in use. It should also show that it is very simple to generate a complex logical function based on the 4 inputs (named A, B, C and D).

Control of the AOI inputs from the XBARB2 is performed by macros as follows:

```
XBAR2_OUT_AOI1_IN00(XBAR2_LOGIC_HIGH);   // connect a logic high to AOI1 A output
XBAR2_OUT_AOI1_IN01(XBAR2_LOGIC_HIGH);   // connect a logic high to AOI1 A output
XBAR2_OUT_AOI1_IN02(XBAR2_PIT_TRIGGER0); // connect the PIT trigger 0 to AOI1 C output
XBAR2_OUT_AOI1_IN03(XBAR2_ACMP1_OUT);    // connect the ACMP1 output to AOI1 D output
```

This resuts in the first channel of the AOI1 module to have its 4 inputs (A,B,C and D) connected to the signals defined by these multiplexer settings ('1', '1', PIT_TRIGGER0, ACMP1_OUT).

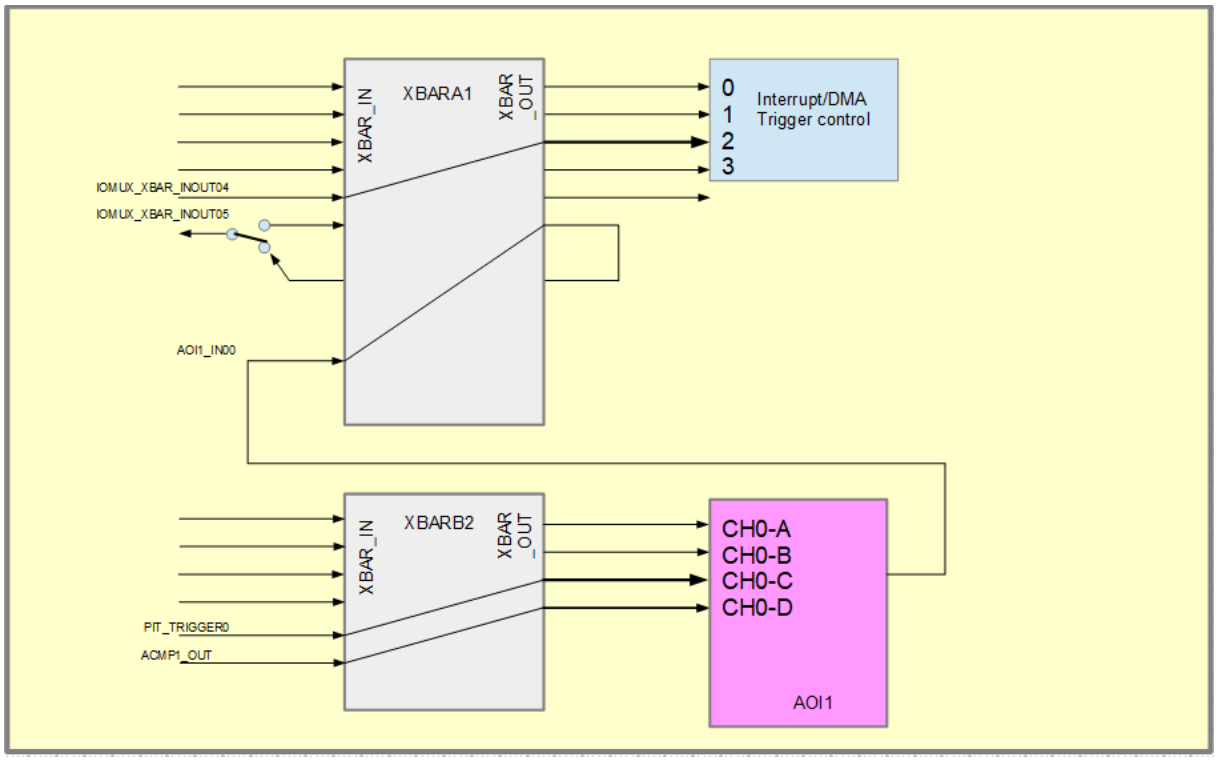
The second channel would be controlled by `XBAR2_OUT_AOI1_IN04()`, `XBAR2_OUT_AOI1_IN05()`, `XBAR2_OUT_AOI1_IN06()` and `XBAR2_OUT_AOI1_IN07()`

The third channel by `XBAR2_OUT_AOI1_IN08()`, `XBAR2_OUT_AOI1_IN09()`, `XBAR2_OUT_AOI1_IN10()` and `XBAR2_OUT_AOI1_IN11()`

And the fourth channel by `XBAR2_OUT_AOI1_IN12()`, `XBAR2_OUT_AOI1_IN13()`, `XBAR2_OUT_AOI1_IN14()` and `XBAR2_OUT_AOI1_IN15()`

The combination of XBARB2, AOI1 and XBARA1 - along with their inputs from pins and peripherals and outputs to pins and periperals - is known as the On-Chip Cross Trigger Network.

The following diagram shows how the IAO is 'inserted' into the path in the previous code example. The logical combination of the two inputs (PIT_TRIGGER0 and ACMP1_OUT) is conneted to the pin output XBAR_INOUT05, but could also be used to generate an input or DMA trigger, or be connected to other peripheral inputs that are connected on XBARA1.



For a list of the possible connections see the i.MX RT 1021 user's manual or the iMX.h header file in the uTasker project, which allows cnnection macros to be used with names rather than input/output numbers and also good compatibility between i.MX RT parts due to this abstraction.

See the following video for further details about this module: **TO ADD**

18. Conclusion

The present state of development is that the i.MX RT 1021 (MIMXRT1020-EVK) can be operated from SRAM (with debugger or together with the µTasker boot loader) or from QSPI Flash (stand-alone) in various clocking configurations up to maximum speed. The following operations can be demonstrated:
- GPIO, port interrupts
- LPUARTs (interrupt and DMA driven)
- Ethernet
- HS USB device
- USDHC
- Parameter system/File system in QSPI
- PIT
- Boot-loading with USB-MSD and application operation in dynamically configured FlexRAM
- dynamic low power (WAIT) mode
- DMA operations (memory-to-memory) can be demonstrated
- Instruction and data cache can be optionally enabled/disabled
- Operation of I.MX RT 1010, 1015, 1020, 1050, 1060, 1064
- Encrypted code storage and over-the air concept

All such operation is simulated in visual studio.

Next immediate steps:
- Demonstrate full Modbus serial and TCP operation.

Subsequent steps:
- Add FreeRTOS configurations

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 reset
- LPI2C, RTC integration requires validation
- General support for i.MX RT 1010, i.MX RT 1015, i.MX RT 1020, i.MX RT 1050, i.MX RT 1060 and i.MX RT 1064 integrated but 1064 operation for internal QSPI flash not yet functional
- Dynamic low power operation supported but not yet successful on all parts (SYSTICK seems to stop on I.MX Rt 1060/64 parts, for example)

Modifications:
V0.10 22.3.2020: Initial version in development
V0.13 01.9.2020: Added XBAR, AOI chapter

Appendix A – Hardware Dependencies

a) Space for first Appendix