SPC58EHx/SPC58NHx OctalSPI HyperBus

Introduction

Nowadays the HyperBus interface technology offers many advantages supporting both legacy and many new features available inside the memory devices. These are for enhancing the overall performance e.g. Single/Double Data Rate (SDR/DDR), simplifying the design (due to low-pin usage) and reducing the global system cost.

To meet the growing requests in the automotive context about the integration of high-speed memories, STMicroelectronics offers the SPC58EHx/SPC58NHx micro-controller family that embeds two OctalSPI interfaces.

These interfaces enable the connection of the OctalSPI and HyperBus high-speed volatile and non-volatile memories available in the market and automotive graduated.

The OctoSPI’s memory-mapped mode also allows to access to the external memories as if they were internal ones.

This application note documents how to setup the OctalSPI controllers inside the SPC58EHx/SPC58NHx micro-controller. It provides generic information about the protocols and literature to enter in the topic.

This document aims at explaining how to configure both OctalSPI instances in order to write and read HyperBus memories (HyperRAM and HyperFLASH).

It also describes some typical use cases to use the OctoSPI interface and providing some practical examples on how to use the interface depending on the type of the targeted memory.

This application is written by using SPC5Studio (tool is available at the following URL: https://www.st.com/en/development-tools/spc5-studio.html).

1 OctalSPI introduction

The OctoSPI is a serial interface which allows the communication on 8 data lines between the micro-controller and an external memory.

The OctoSPI provides a flexible hardware interface, which enables the support of multiple hardware configurations. It supports the Single-SPI (traditional SPI), Dual-SPI, Quad-SPI, Dual Quad-SPI and Octal-SPI.

The SPC58EHx/SPC58NHx OctalSPI has several features:

- Three functional modes: indirect, status-polling, and memory-mapped.
- Read and write support in the memory-mapped mode.
- Supports for single, dual, quad and octal communications.
- SDR and DTR support. Latter is supported and validated on this product.
- Data strobe support.
- Fully programmable opcode.
- Fully programmable frame format.
- HyperBus support.
- Integrated FIFO for reception and transmission.
- 8, 16, and 32-bit data accesses are allowed.
- DMA channel for indirect mode operations.
- Interrupt generation on FIFO threshold, timeout, operation complete, and access error.

This application note is focused on HyperBUS in DDR mode according to the Microcontroller requirements and on Cypress memories.

Note: for additional information refer to the “Octo-SPI” chapter inside the Microcontroller reference manual (see Section Appendix C Reference documents).

1.1 OctalSPI protocol and operating modes

The OctoSPI interface can operate in two different low-level protocols:

- Regular command mode.
- HyperBus frame format.

Each protocol supports three operating modes:

- Indirect mode.
- Memory-mapped mode.
- Status-polling mode.

1.1.1 Regular command mode

The regular-command mode is the classical frame format where the OctoSPI communicates with the external memory device by using commands; each command can include up to five phases and, at least, one of the five phases must be present. The interface provides a fully programmable frame support.

1.1.2 HyperBUS mode

The OctoSPI supports the HyperBus protocol which enables the communication with HyperRAM and HyperFlash memories.

The HyperBus has a Double Data Rate (DDR) interface where two data-bytes per clock cycle are transferred over the DQ input/output (I/O) signals, leading to high read and write throughputs.

Note: for additional information on HyperBus interface operation, refer to the HyperBus specification (see Section Appendix C Reference documents).

The HyperBus frame is composed of two phases:

- Command/address phase (CA).
- Data phase.

On the command/address phase, the HyperRAM memory will use the read-recovery time (RWDS) to understand if an additional initial access latency has to be inserted.

The OctoSPI peripheral can be configured in the following HyperBus modes:

- HyperBus memory mode: to allow the read/write access from/to the HyperBus memory.
- HyperBus register mode: to access to the memory device register space.

See Section Appendix A HyperBus timing for timing considerations.

1.2 OctalSPI operating modes

1.2.1 Indirect mode

The indirect mode is used either for both HyperBus and regular-command protocols.

The commands are started by writing to the OCTOSPI registers and the data is transferred by writing or reading the data registers.

Typically, this mode is adopted to configure the external memory or resetting/erasing a HyperFLASH device.

In the next chapters, this mode is implemented, and related examples are provided on both HyperRAM and HyperFLASH cases but only on the supported HyperBus mode.

1.2.2 Memory-mapped mode

The external memory device is seen as an internal one, but no more than 256 MB can be addressed in this architecture. The MCU can also execute code from the external memory. The application shows how to configure and debug this mode.

The memory mapped regions for both OctalSPI instances are reported in Table 1. OctalSPI memory map.

1.2.3 Status polling mode

This feature is used to avoid the polling made by the software when, for example, the memory has been configured or when the ongoing operation (like FLASH programming/erasing) is completed. In automatic-polling mode, in fact, the OctoSPI hardware can periodically read the status and data generating an interrupt on a matching flag.

2 SPC58EHx/SPC58NHx OctalSPI mapping

Each instance has its own chip select. The below table indicates the memory map layout:

Table 1. **OctalSPI memory map**

Instance	Description	Address space
OctalSPI 1	Memory mapped	0x70000000 - 0x7FFF_FFFF
	Controller Registers	0x80000000 - 0x8000_03FF
	Delay Block	0x80000400 - 0x8000_0407
OctalSPI 2	Memory mapped	0x60000000 - 0x6FFF_FFFF
	Controller Registers	0x80001000 - 0x800013FF
	Delay Block	0x80001400 - 0x80001407

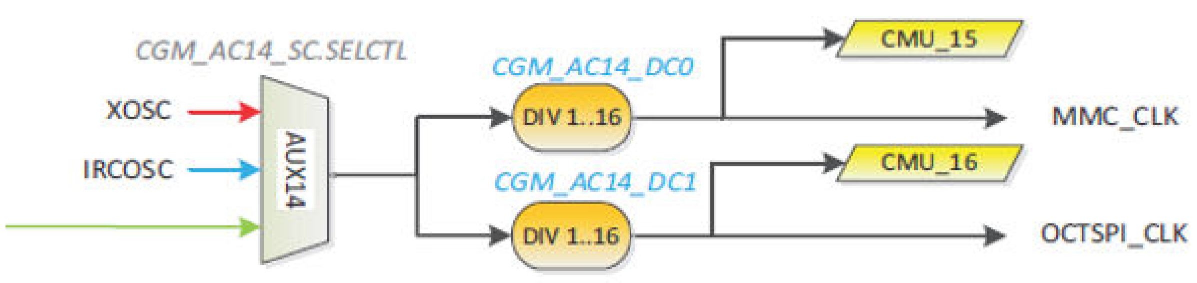
The two OctalSPI share the same data lines to reduce the number of signals and a dedicated IO manager is used to mux them.

The IO manager is mapped at 0x80002000 - 0x8000200B. For full description point to the micro-controller reference manual.

3 OctalSPI clock selection

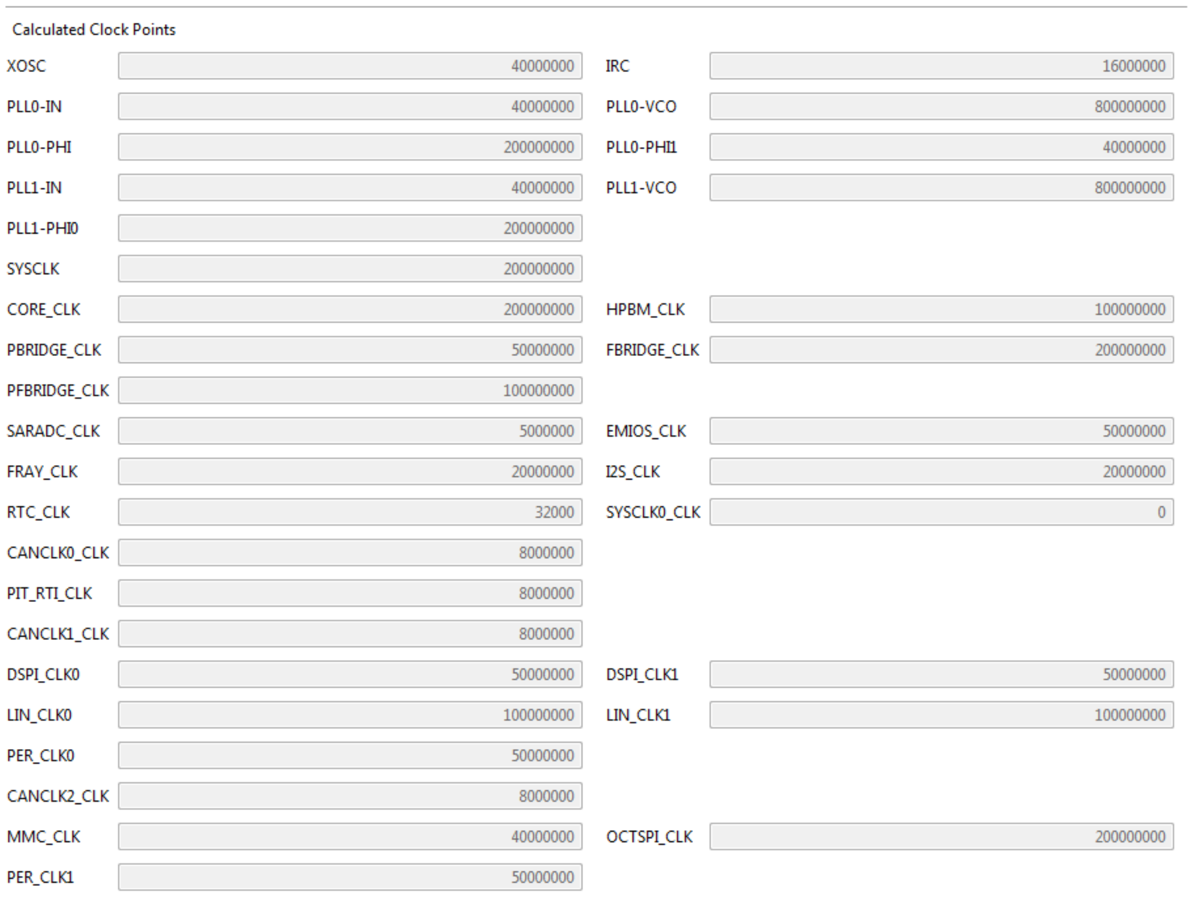
The CGM registers `CGM_AC14_DC1` (Aux Clock 14 Divider Configuration 1) and `CGM_AC14_SC` (Aux Clock 14 Select Control) have to be configured to provide the right clock to the controllers.

Figure 1. OCTSPI_CLK GCM selecion



The *OCTSPI_CLK* is configured to 200 MHz (from PLL); a pre-scaler is adopted later by programming the controller registers (see Figure 3. OctalSPI prescaler selection (200 / 2 = 100 MHz)) to obtain 100 MHz, this is the default clock in this application for both instances.

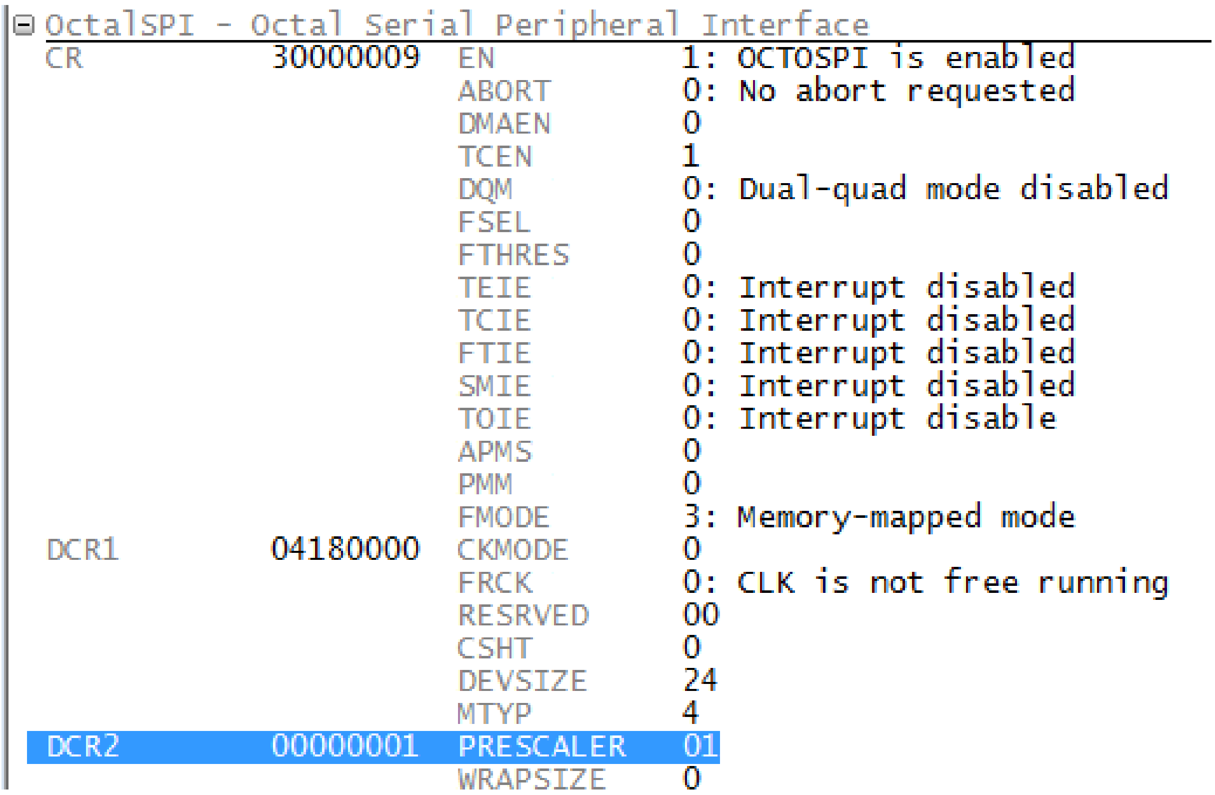
Figure 2. Application clock details (SPC5Studio wizard)



Calculated Clock Points

XOSC	40000000	IRC	16000000
PLL0-IN	40000000	PLL0-VCO	800000000
PLL0-PHI	200000000	PLL0-PHI1	40000000
PLL1-IN	40000000	PLL1-VCO	800000000
PLL1-PHI0	200000000		
SYSCLK	200000000		
CORE_CLK	200000000	HPBM_CLK	100000000
PBRIDGE_CLK	50000000	FBRIDGE_CLK	200000000
PFBRIDGE_CLK	100000000		
SARADC_CLK	5000000	EMIOS_CLK	50000000
FRAY_CLK	20000000	I2S_CLK	20000000
RTC_CLK	32000	SYSCLK0_CLK	0
CANCLK0_CLK	8000000		
PIT_RTI_CLK	8000000		
CANCLK1_CLK	8000000		
DSPI_CLK0	50000000	DSPI_CLK1	50000000
LIN_CLK0	100000000	LIN_CLK1	100000000
PER_CLK0	50000000		
CANCLK2_CLK	8000000		
MMC_CLK	40000000	OCTSPI_CLK	200000000
PER_CLK1	50000000		

Figure 3. OctalSPI prescaler selection (200 / 2 = 100 MHz)

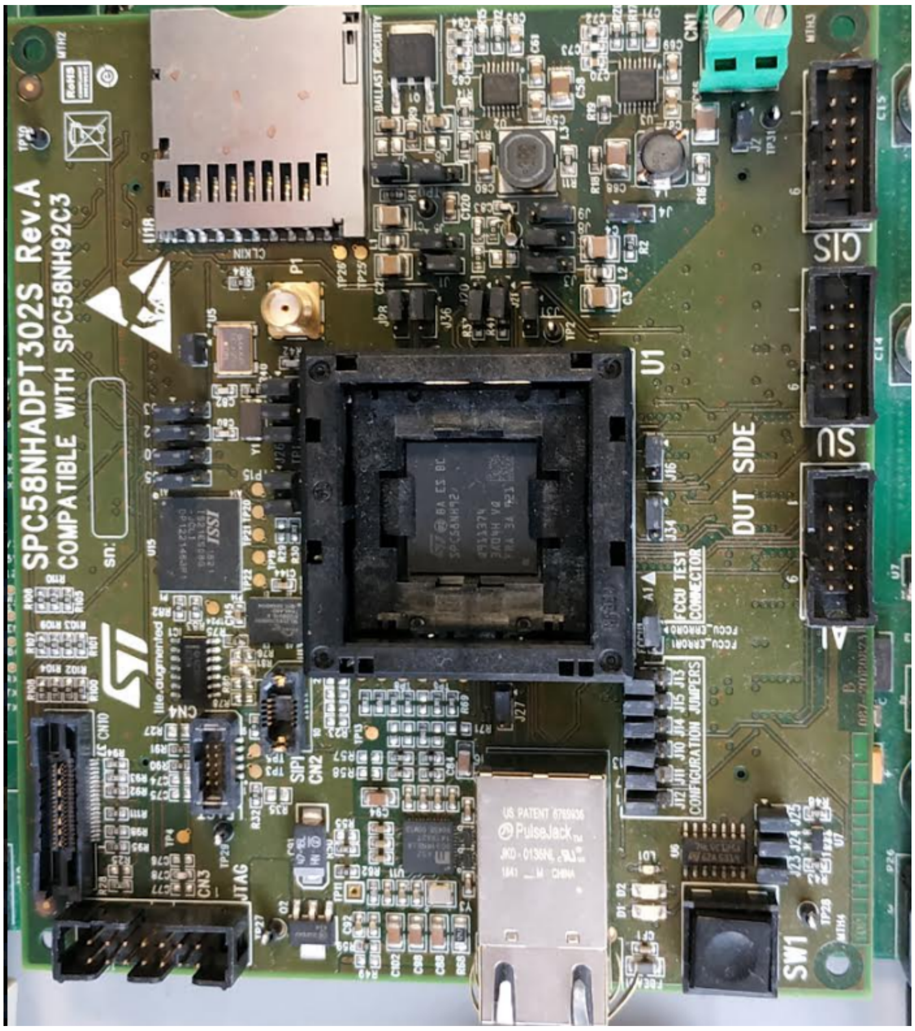


```
⊟ OctalSPI - Octal Serial Peripheral Interface
  CR            30000009  EN          1: OCTOSPI is enabled
                          ABORT       0: No abort requested
                          DMAEN       0
                          TCEN        1
                          DQM         0: Dual-quad mode disabled
                          FSEL        0
                          FTHRES      0
                          TEIE        0: Interrupt disabled
                          TCIE        0: Interrupt disabled
                          FTIE        0: Interrupt disabled
                          SMIE        0: Interrupt disabled
                          TOIE        0: Interrupt disable
                          APMS        0
                          PMM         0
                          FMODE       3: Memory-mapped mode
  DCR1          04180000  CKMODE      0
                          FRCK        0: CLK is not free running
                          RESRVED     00
                          CSHT        0
                          DEVSIZE     24
                          MTYP        4
  DCR2          00000001  PRESCALER   01
                          WRAPSIZE    0
```

4 Hardware setup

The OctalSPI instances can be tested on different platforms designed for this MCU family. In this application, the SPC58NHADPT302S board is the reference PCB.

Figure 4. SPC58NHADPT302S board



5 HyperRAM multi-chip package

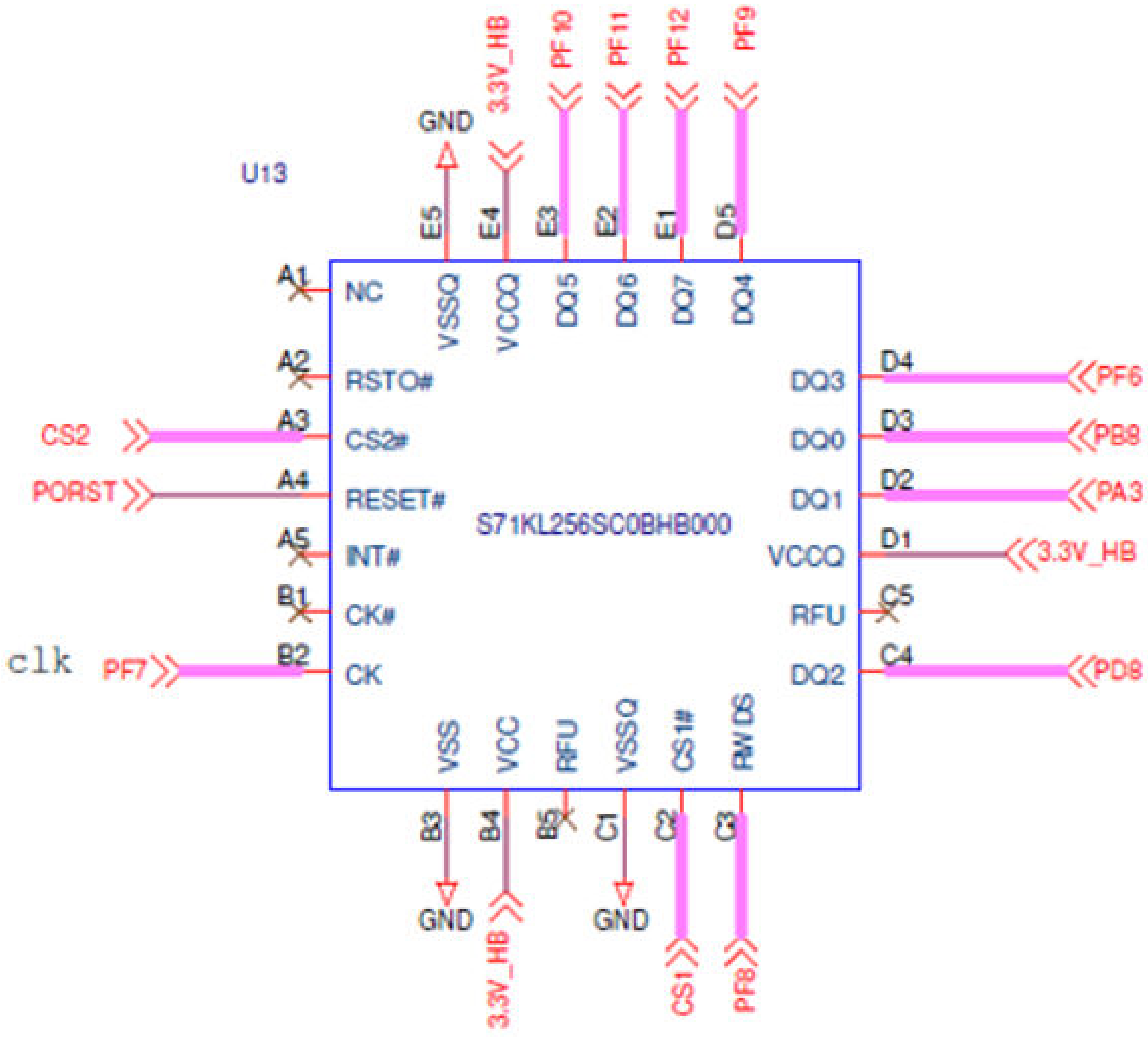
The SPC58NHADPT302S embeds the S71KL256SC0BHB000 by Cypress that's an HyperFlash and HyperRAM multi-chip package composed by:
- S26K HyperFlash device.
- S27K HyperRAM device.

Software can dialog with the two devices by using the OctalSPI instances **but not simultaneously.**

For HyperFlash and HyperRAM operating conditions refer to cypress related datasheets.

Note: the OCTOSPI_1 chip select (CSN0) addresses the HyperRAM (on CS2#) while the OCTOSPI2 (CSN1) addresses the HyperFLASH (on CS1#).

Figure 5. S71KL256SC0BHB000 package



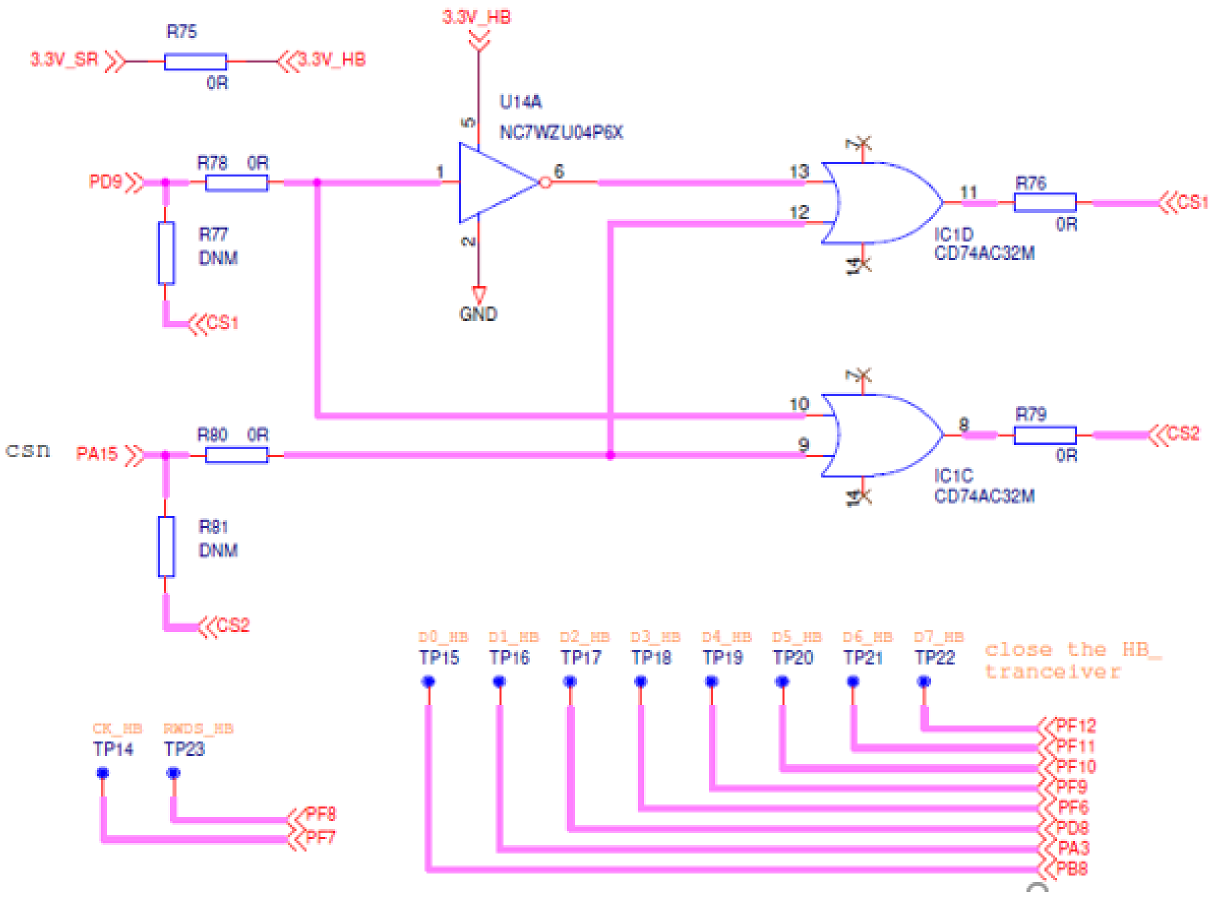


Figure 6. **S71KL256SC0BHB000 wiring**

Note: Ensure that on this PCB the following resistors R78, R80, R76 and R79 are removed and 0 Ohm fits to R81 and R77 instead of 0 Ohm fits to both R81 and R77.

6 OctalSPI signals

The OctoSPI interface uses up to eleven lines:

- CNS line for chip select.
- CLK line for clock.
- DQS line for data strobe.
- DQ[0...7] eight lines for data.

Below the configuration of the pins used in this application for this micro-controller:

Table 2. **OctalSPI IO mapping**

Port	SIUL MSCR#	MSCR SSS	Function	Description	Dir
OctalSPI 1 and OctalSPI 2 - Data signals					
PN[8]	216	00000010	DATA0	OctalSPI Data 0	i/o
PN[8]	912	00000001	DATA0	OctalSPI Data 0	i
PN[9]	217	00000010	DATA1	OctalSPI Data 1	i/o
PN[9]	913	00000001	DATA1	OctalSPI Data 1	i
PN[10]	218	00000010	DATA2	OctalSPI Data 2	i/o
PN[10]	914	00000001	DATA2	OctalSPI Data 2	i
PN[11]	219	00000010	DATA3	OctalSPI Data 3	i/o
PN[11]	915	00000001	DATA3	OctalSPI Data 3	i
PN[13]	221	00000010	DATA4	OctalSPI Data 4	i/o
PN[13]	916	00000001	DATA4	OctalSPI Data 4	i
PN[14]	222	00000010	DATA5	OctalSPI Data 5	i/o
PN[14]	917	00000001	DATA5	OctalSPI Data 5	i
PN[15]	223	00000010	DATA6	OctalSPI Data 6	i/o
PN[15]	918	00000001	DATA6	OctalSPI Data 6	i
PO[0]	224	00000010	DATA7	OctalSPI Data 7	i/o
PO[0]	919	00000001	DATA7	OctalSPI Data 7	i
OctalSPI 1 and OctalSPI 2 – Clock and DQS					
PP[14]	254	00000010	CLK	OctalSPI Clock	i/o
PP[14]	920	00000001	CLK	OctalSPI Clock	i
PQ[1]	257	00000010	DQS	OctalSPI Data strobe	i/o
PQ[1]	911	00000010	DQS	OctalSPI Data strobe	i
OctalSPI 1 and OctalSPI 2 – Chip select					
PD[9]	57	00000111	CSN1	OctalSPI 2 Chip select	o
PN[12]	220	00000010	CSN0	OctalSPI 1 Chip select	o

In the SIUL2 configuration the Output Edge Rate Control (OERC) bit for output pins should be set as very strong / ultra strong drive.

7 Delay block overview

The delay block module is integrated with the OctalSPI to generate a delay on DQS signal. This block is designed to be used for both SDR and DDR modes.Only the DDR mode and 100 MHz as frequency are supported.

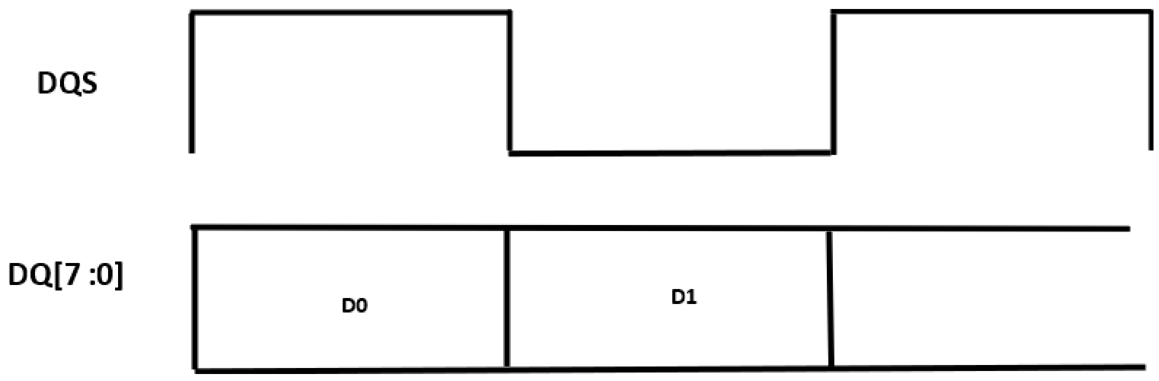
The usage of delay block allows to align the DQS edge to start the valid data window eye and try to give the maximum hold margin.

Generally, data and DQS are edge aligned so there is no extra delay necessary, in any case, the application can tune at runtime the block according to the concurrency of some events (e.g. temperature variation).

7.1 Delay block timing example

The DQS delay range can vary according to the memory type and constructor. This paragraph shows an example of the delay block timing adopted on this reference platform. According to Cypress HyperRAM timing parameter specification, for 3.0 Volt, in case of 100 MHz the RDWS, that is mapped on DQS, can be -0.8 or +0.8 ns.

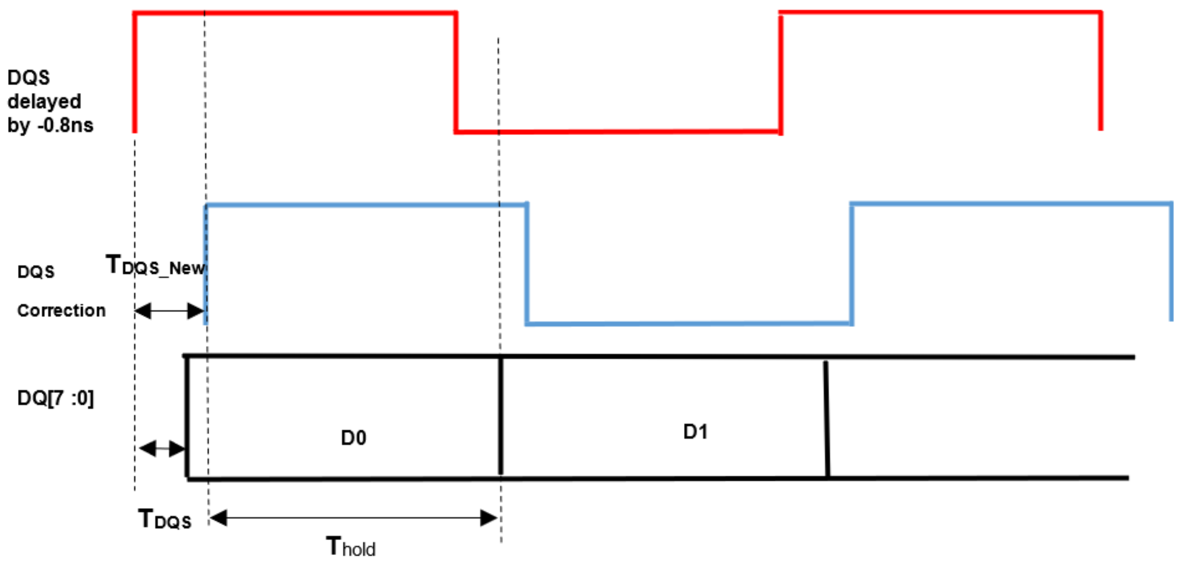
Figure 7. **DQS and data signals aligned**



So considering the case where the DQS introduces a delay of -0.8 ns, it is before the DQ [7:0] (issue on properly toggling the data).

Note that the delay introduced by the internal logic (T_{PAD}) on DQS w.r.t. DQ [7:0] lines must be also considered. This delay is ranging from 50 ps to 2.1 ns. The following figure shows the correction and hold margin when the delay block applies to a positive shift of 1/12.

Figure 8. **DQS is delayed by -0.8 ns and DLYBLK adds 1/12 as correction**



The following formulas can be considered to calculate the correction:

$T_{correct} = T_{DLY} + T_{PAD}$

$T_{DQS\_New} = T_{correct} + T_{DQS}$

T_{hold} is the time to toggle data valid after the RWDS edge.

$T_{hold} = T_{half\_period} - T_{DQS\_New}$

Where the T_{half_period} is 4.5 ns, the minimum half period according to the Cypress Hyperbus specification at 100 MHz.

Table 3. **Hold margin according to a DQS delayed by - 0.8 ns**

T_{DLY}		T_{DQS}	$T_{PAD(min)}$	$T_{correct}$	TDQS_New	T_{hold}
Sel/Unit	[ns]	[ns]	[ps]	[ps]	[ns]	[ns]
1/10	1	-0.8	50	1050	0.25	4.25
1/12	0.8		50	850	0.05	4.45
2/10	2		50	2050	1.25	3.25
2/12	1.6		50	1650	0.85	3.65

Table 4. **Hold margin according to a DQS delayed by + 0.8 ns**

T_{DLY}		T_{DQS}	$T_{PAD(max)}$	$T_{correct}$	TDQS_New	T_{hold}
Sel/Unit	[ns]	[ns]	[ps]	[ps]	[ns]	[ns]
1/10	1	+0.8	2100	3100	3.9	0.6
1/12	0.8		2100	2900	3.7	0.8
2/10	2		2100	4100	4.9	-0.4
2/12	1.6		2100	3700	4.5	0

When the DQS anticipates the data (T_{DQS} = - 0.8 ns), as mentioned in Table 4. Hold margin according to a DQS delayed by + 0.8 ns, applying a T_{DLY} of 1/12 or 1/10 there is a margin to toggle the data and T_{hold} is quite high.

In the second case, when the DQS is after than the data (T_{DQS} = + 0.8), selecting a T_{DLY} equals to 2/10 or 2/12 the T_{hold} is reduced so this T_{DLY} selection should be avoided and a delay of 1/12 or 1/10 remains a reasonable choice.

Concluding, in this specific case for this platform and test condition, applying a T_{DLY} from 1/12 to 1/10 can guarantee acceptable margins to toggle the right data in time.

7.2 Delay block enable

The `DelayBlock_Enable` function is the entry point to enable the module and execute the tuning procedure.

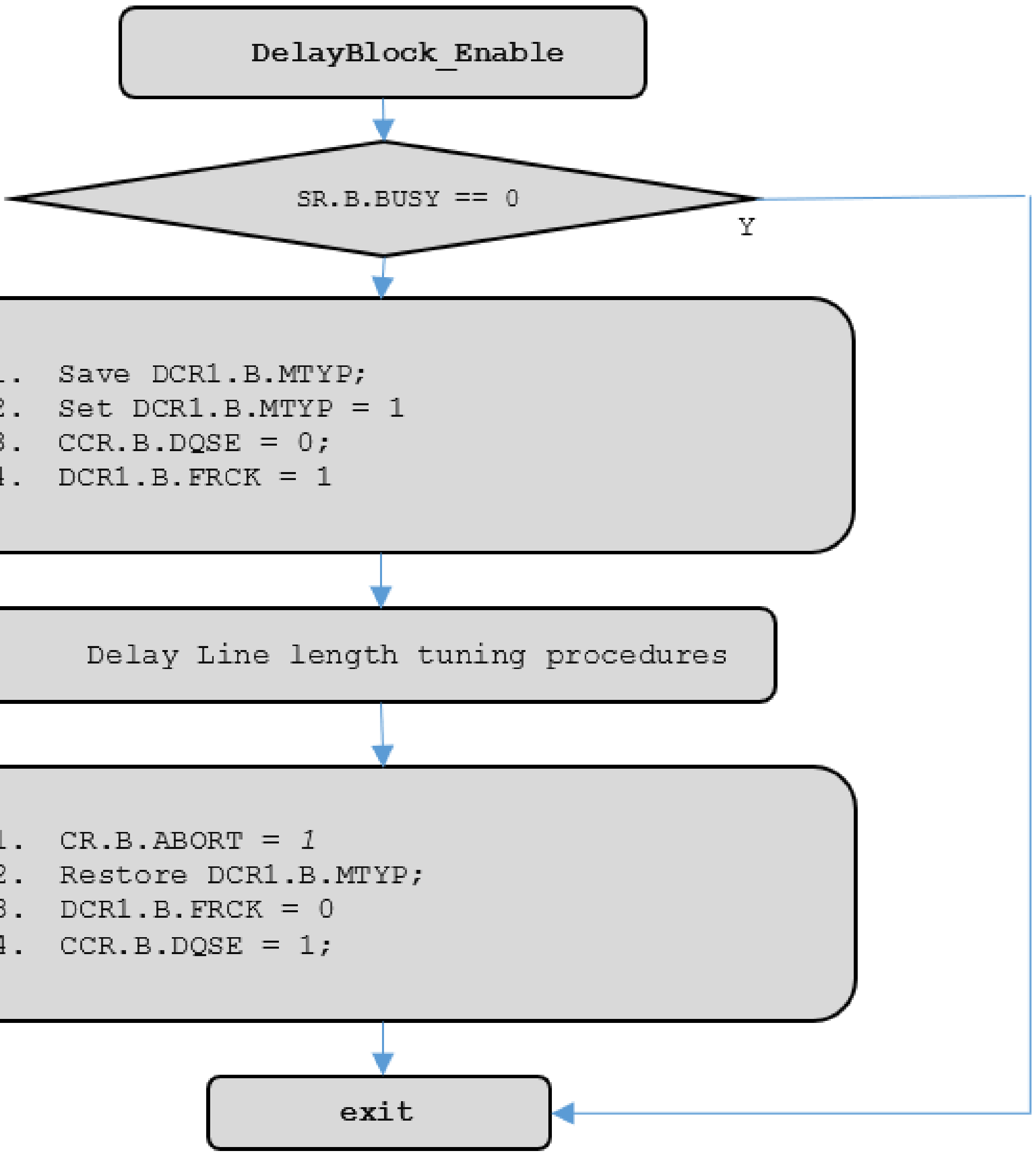
This function can be called when the OctalSPI instance is initialized.

Before re-calibrating the delay block, If `SR.B.BUSY` is set to 1, the `DelayBlock_Enable` has to save some parts of the configuration and after raising an `ABORT` it can start tuning the block as described in the next paragraph.

The initial status will be restored after completing the whole procedure.

Any error condition will be reported to the caller function.

Figure 9. Delay block enable procedure



7.3 Delay block length tuning

When the delay block is enabled the following algorithm must be adopted to tune the lanes and select the delay desired on this platform.

Below an extract of the delay line length tuning procedures implemented inside the `DelayBlock_Enable` function and related to the full period measurement.

```
uint32_t unit = 0;
  uint32_t tuning_done = 0;
  uint32_t length_value;

  //Enable Sampler and Delay blocks
  OCTALSPI.DELAY_CTRL.B.SE = 1U;
  OCTALSPI.DELAY_CTRL.B.DE = 1U;

  //Enable all 12 delay cells
  OCTALSPI.DELAY_CFG.B.DELAYSEL = 12U;

  for(unit = 0; DLYB_MAX_UNIT >= unit; unit++)
  {
      //apply new DELAYUNIT
      OCTALSPI.DELAY_CFG.B.DELAYUNIT = unit;
      //Wait for DELAYF
      while (0U == OCTALSPI.DELAY_CFG.B.DELAYF)
      {
         NOP();
      }

      length_value = OCTALSPI.DELAY_CFG.B.DELAYLENGTH;

      //0111_11x0_0000 >>> FULL_PERIOD we detect 2nd falling edge
      if((0U < length_value) && (0U == (length_value & DLYB_CFGR_LNG_11)))
      {
          //tuning is finished properly
          tuning_done = 1;
          break;
      }
   }

   if(1U == tuning_done)
   {
      //set the delay to 1/12 of CLK period
      OCTALSPI.DELAY_CFG.B.DELAYSEL = 1;
      //Sample length disabled
      OCTALSPI.DELAY_CTRL.B.SE = 0;
      //Delay Block enabled (output clock    generated with the selected phase)
      OCTALSPI.DELAY_CTRL.B.DE = 1;
   }
   else
   {
       //The algo did not find the right DELAYUNIT. The CLK period is probably too slow.
       //Issue may be reported to upper SW layer end/or Delay block may be disabled

       //only for debugging !
       OCTALSPI.DELAY_CFG.B.DELAYSEL = 1;
       OCTALSPI.DELAY_CTRL.B.DE = 0U;
       OCTALSPI.DELAY_CTRL.B.SE = 0U;
       OCTALSPI.CR.B.EN = 0U;
       OCTALSPI.DCR1.B.FRCK = 0U;
       while(1);
   }
```

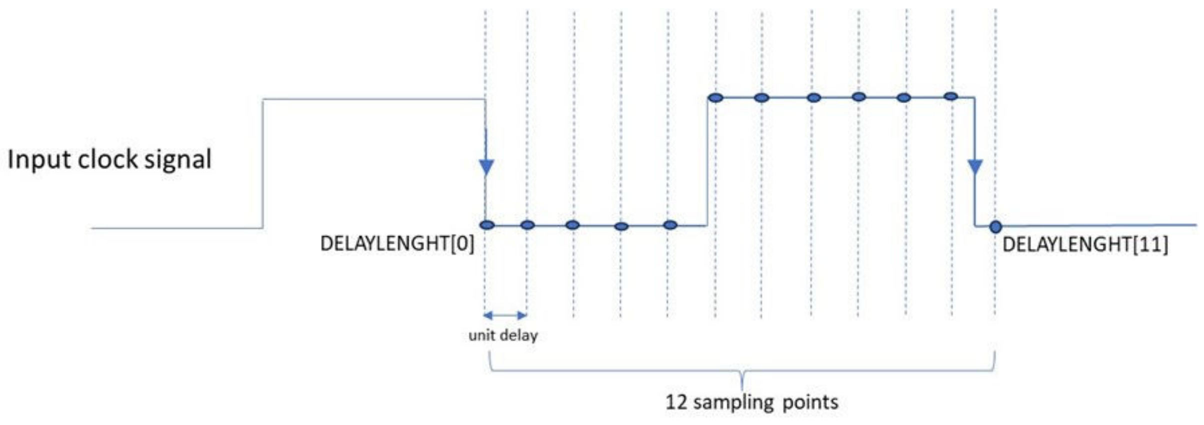
Note: Used values `#define DLYB_MAX_UNIT 127U #define DLYB_CFGR_LNG_11 0x800U`.

7.3.1 Code/algorithm description

The algorithm proposed consists of tuning the delay block on a full period.

DELAYLENGHT field can be considered as a shift register containing the sampling of the input clock signal done every delay unit (defined by the correspondent DELAYUNIT field) once the falling edge of the input clock is triggered, as described in the picture below.

Figure 10. Delay block tuning



The algo increases the duration of delay units (from 0 to 127) until the entire period is covered, passing through following phases:

1. As long as 12*delay units is lower than half a period, DELAYLENGHT field will remain at 0x0.
2. When "12*delay units" is between half a period and a period, the first bits of DELAYLENGHT will be at 0, and the last bits will be at 1.
3. When "12*delay units" is just greater than a full period, then the first bits of DELAYLENGHT will be at 0, then we will see some bits at 1, then the last bit (DELAYLENGHT[11]) will be at 0.

The condition 3) determines the completion of the tuning.

There is also the possibility to tune the delay block on a half period: in this case, it can be considered completed when the "12 *delay units" is just greater than a half period with the last bit of DELAYLENGHT at 1 and all the others at 0.

Such kind of algorithm can be optimized and modified according to the hardware constraints and not to typical temperature conditions. In some cases, further corrections need to be adopted for TDLY to guarantee a better alignment of the RWDS with the data signals. In fact, the delay selection in this example, is in-line with the timing calculated on this specific platform on normal temperature conditions. The application could retune the delay block when the voltage/temperature forces the DQS/RDWS to go out of the data valid window. This case is not covered in this demo application.

8 Application overview

This application after configuring the whole micro-controller, setup clocks and signals, then in a loop sequentially tests both OctalSPI_1 and OctalSPI_2. The focus is on the HyperBUS protocol implementation.

Below is the simple main function:

```
int main(void)
{
   int i, j;
   int fail = 0;

   componentsInit();
   irqIsrEnable();

   for (i=0; i < test_loops; i++) {
        for (j=0; j <=1 ; j++) {
          /* Arguments: instance number - delay block enabled */
          fail += HYPERRAM_Test(1, j);
          fail += HYPEFLASH_Test(2, j);
          }
   }
   printf("\nOCTALSPI result: %d failures in %d loops\r\n", fail, test_loops);
}
```

The first instance is mapped on the HyperRAM while the second one is mapped on HyperFLASH.

Both devices will be tested with a set of user case to check some features and robustness.

The test output is visible over serial console, a LINFLEX instance is properly configured as UART for this purpose. The LIN_RX PIN PD[14] is configured as UART TX signal.

Before accessing the OctalSPI, the following platform setup must be applied to enable the controller accesses.

```
static void octalspi_pltfrm_setup(void)
{
    PMCDIG.VSIO.B.VSIO_EMMC = 0;
    PMCDIG.VSIO.B.VSIO_ETH1 = 0;

    SPCSetPeripheralClockMode(SPC5_OCTALSPI_PCTL, SPC5_ME_PCTL_RUN(1) |
    SPC5_ME_PCTL_LP(2));

    /* Reset of peripheral to clear all registers */
    MC_RGM.PRST2.B.OCTALSPI_RST = 1;
    MC_RGM.PRST2.B.OCTALSPI_RST = 0;

    /* This enables the multiplexing of the two OctoSPI.
     * Note: this can be done before init the devices. */
    OCTALSPI_IOM.IOM_CR.R = 1U;
}
```

Where:

```
#define SPC5_OCTALSPI_PCTL 77 /* -> Reference Manual - PCTL */
```

Note: *the two OSPI instances are embedded along with an IO manager. This is controlled by a single PCTL.*

8.1 HyperRAM application

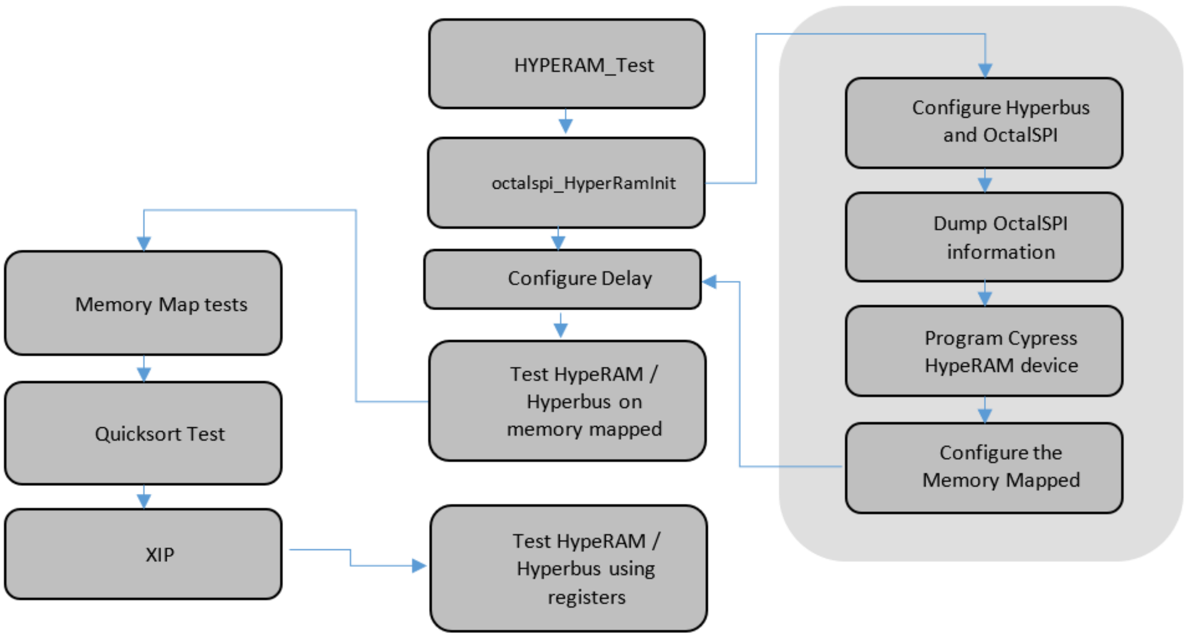
After configuring the OctalSPI_1, the following tests are executed on the device:

- The memory map space is filled with a fixed pattern and a memory copy is tested. The data moved from/to memory mapped space (e.g. 0x70000000) must be coherent.
- A quick sort algorithm is implemented. The data sorted in SRAM and on a memory mapped region must be equal.

- The indirect-mode transfer via registers.
- XIP mode (see the related paragraph).

In case an error occurs, while configuring the device, the application will exit with an error.
Any test failure will be logged and reported at the end of the execution. The following block diagram aims at summarizing the test execution.

Figure 11. **HyperRAM test simplified block schema**


HYPERAM_Test
octalspi_HyperRamInit
Configure Delay
Test HypeRAM / Hyperbus on memory mapped
Test HypeRAM / Hyperbus using registers
Memory Map tests
Quicksort Test
XIP
Configure Hyperbus and OctalSPI
Dump OctalSPI information
Program Cypress HypeRAM device
Configure the Memory Mapped

8.1.1 HyperRAM configuration

The `HYPERRAM_Test` function configures the OctalSPI from scratch by invoking the following API:

```
static void octalspi_init(const ospi_cfg_t *config)
{
    octalspi_pltfrm_setup(); /* Platform configuration */

    octalspi->DCR1.B.MTYP = config->memoryType;
    octalspi->DCR1.B.DEVSIZE = config->deviceSize - 1;
    octalspi->DCR1.B.CSHT = config->chipSelectHighTime - 1;
    octalspi->DCR1.B.FRCK = config->freeRunningClock;
    octalspi->DCR1.B.CKMODE = config->clockMode;
    octalspi->DCR2.B.WRAPSIZE = config->wrapSize;
    octalspi->DCR2.B.PRESCALER = config->clockPrescaler;
    octalspi->DCR3.R = 0;
    octalspi->DCR3.B.CSBOUND = config->chipSelectBoundary;
    octalspi->CR.B.FTHRES = config->fifoThreshold - 1;
    /* Wait until the device is not BUSY */
    while (octalspi->SR.B.BUSY)
        osalThreadDelayMilliseconds(1UL);

    octalspi->CR.B.DQM = config->dualQuad;
    octalspi->TCR.B.SSHIFT = config->sampleShifting;
    octalspi->TCR.B.DHQC = config->delayHoldQuarterCycle;
    octalspi_ccr(); /* setup the communication configuration register */
    octalspi->CR.B.EN = 1; /* Enable the OctalSPI */
}
```

The **`const`** `ospi_cfg_t` is a structure to configure the OctalSPI device that is fixed in the project configuration. All the parameters used to fill this structure must be selected according to the memory device adopted.

When accessing to the HyperRAM device, the following configuration is used to access the device and get the information from registers, the fixed latency mode is adopted:

```
const ospi_hyperbus_cfg_t octalspi_HyperRAM_cypress_bus_conf = {
    4,   /* Read write recovery time */
    6,   /* Device access time */
    1,    /* 1: no latency on Write to access registers */
    1    /* Latency mode => 0: variable, 1: fixed */
};
```

After configuring the device, the variable latency mode can be set up.

```
const ospi_hyperbus_cfg_t octalspi_HyperRAM_bus_conf = {
      4,   /* Read write recovery time */
      6,   /* Device access time */
      0,       /* 0 latency on write to access HyperRAM (device dependent) */
      0    /* Latency mode => 0: variable, 1: fixed */
};
```

If a fixed latency is selected, the memory will always indicate a refresh latency and delay the read data transfer accordingly. If a variable latency is selected, a latency is only added when a refresh is required at the same time a new transaction is starting. A variable latency is preferred for performance reasons on these kind of memories. For variable latency, the device access time has to be up to 6. For the fixed latency it might decrease according to the frequency used.

See also Section Appendix A HyperBus timing for timing considerations.

The configuration is propagated to the cypress memory device by using another low-level driver. This will be described later.

In the HyperBus mode, the device timing (tACC and tRWR) and the latency mode must be configured in the OCTOSPI HyperBus latency configuration register as shown below:

```
void octalspi_HyperbusCfg(const ospi_hyperbus_cfg_t *config)
{
    octalspi->HLCR.R = 0;
    octalspi->HLCR.B.TRWR = config->rwRecoveryTime;
    octalspi->HLCR.B.TACC = config->accessTime;
    octalspi->HLCR.B.WZL = config->writeZeroLatency;
    octalspi->HLCR.B.LM = config->latencyMode;
}
```

The following function is used to select the memory map mode.

```
void octalspi_MemoryMapped(const ospi_cfg_t *config)
{
    octalspi->LPTR.R = 0U;
    octalspi->LPTR.B.TIMEOUT = config->timeoutPeriod;
    octalspi->CR.B.TCEN = config->timeoutActivation;
    octalspi->CR.B.FMODE = OSPI_FMODE_MEMORY_MAPPED;
    octalspi->DCR1.B.MTYP = config->memoryType;
}
```

The `memoryType`, according to the reference manual, can be programmed with the following values, considering that it should be set in HyperBus memory mode on this platform.

0 : Micron mode, D0/D1 ordering in DTR 8-data-bit mode.

1 : Macronix mode, D1/D0 ordering in DTR 8-data-bit mode.

3 : Macronix RAM mode, D1/D0 ordering in DTR 8-data-bit mode.

4 : HyperBus memory mode, the protocol follows the Cypress

5 : HyperBus register mode, addressing register space.

Switching to memory map mode, `memoryType` is equal to 4.

Concerning the functional modes, the application defines the following:

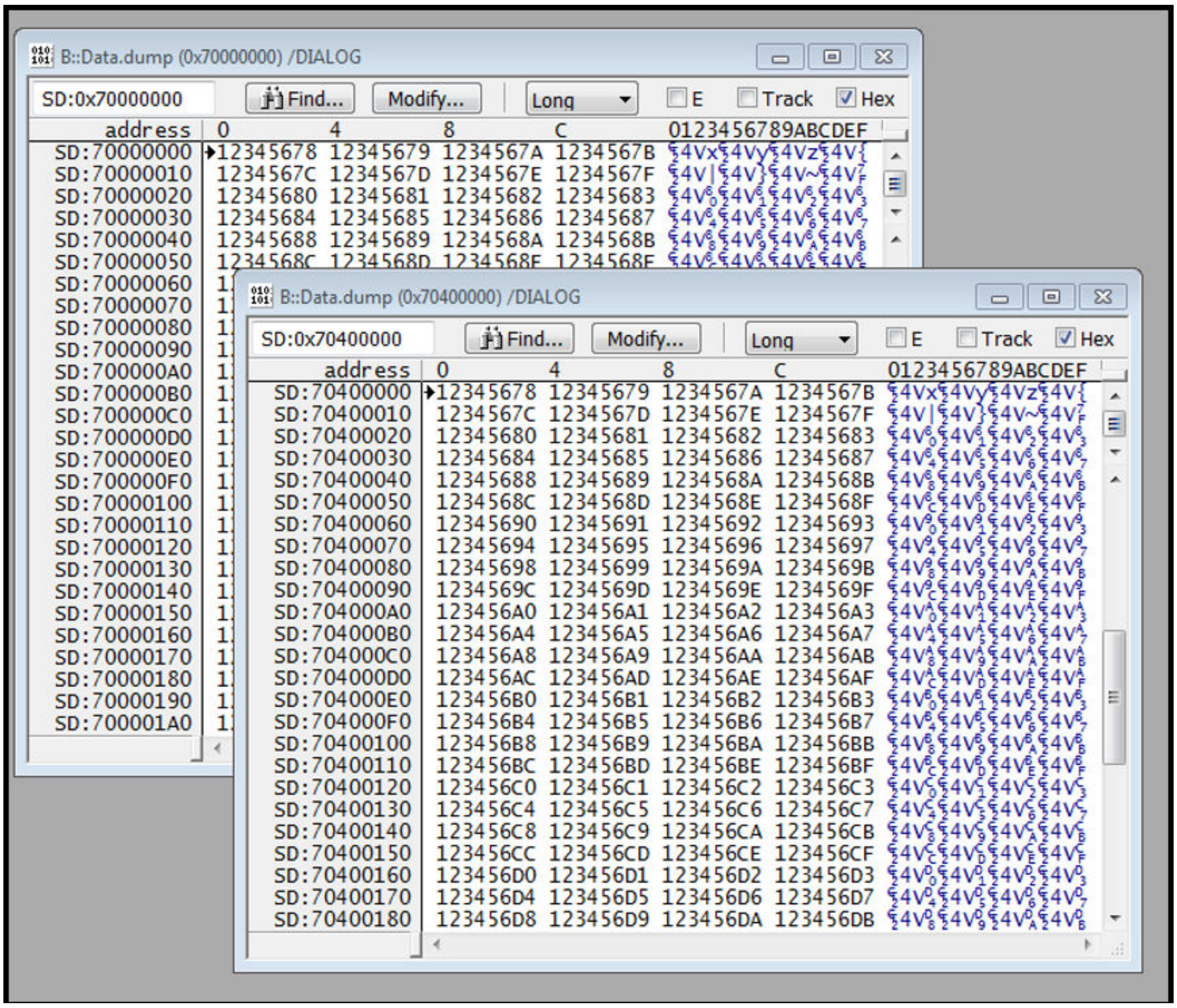
```
#define OSPI_FMODE_INDIRECT_WRITE    0
#define OSPI_FMODE_INDIRECT_READ              1
#define OSPI_FMODE_AUTO_POLLING               2
#define OSPI_FMODE_MEMORY_MAPPED              3
```

Completing the whole setup the application executes the battery of tests, mentioned above, to check the transfer through the memory map space. Completing those, a simple test using the indirect-mode is run as well.

8.1.2 Memory map example

The following figure shows a basic test where a block of the memory mapped space, e.g. 0x70000000, is filled with a pattern and then is copied to another memory. The test fails in case a bit differs.

Figure 12. Memory dump



```
B::Data.dump (0x70000000) /DIALOG
SD:0x70000000   Find...   Modify...   Long   [ ] E   [ ] Track   [✓] Hex
     address  0        4        8        C        0123456789ABCDEF
 SD:70000000 ▸12345678 12345679 1234567A 1234567B §4Vx§4Vy§4Vz§4V{
 SD:70000010  1234567C 1234567D 1234567E 1234567F §4V|§4V}§4V~§4V
 SD:70000020  12345680 12345681 12345682 12345683 §4V§4V§4V§4V
 SD:70000030  12345684 12345685 12345686 12345687 §4V§4V§4V§4V
 SD:70000040  12345688 12345689 1234568A 1234568B §4V§4V§4V§4V
 SD:70000050  1234568C 1234568D 1234568E 1234568F §4V§4V§4V§4V
 SD:70000060  1
 SD:70000070  1
 SD:70000080  1
 SD:70000090  1
 SD:700000A0  1
 SD:700000B0  1
 SD:700000C0  1
 SD:700000D0  1
 SD:700000E0  1
 SD:700000F0  1
 SD:70000100  1
 SD:70000110  1
 SD:70000120  1
 SD:70000130  1
 SD:70000140  1
 SD:70000150  1
 SD:70000160  1
 SD:70000170  1
 SD:70000180  1
 SD:70000190  1
 SD:700001A0  1
```

```
B::Data.dump (0x70400000) /DIALOG
SD:0x70400000   Find...   Modify...   Long   [ ] E   [ ] Track   [✓] Hex
     address  0        4        8        C        0123456789ABCDEF
 SD:70400000 ▸12345678 12345679 1234567A 1234567B §4Vx§4Vy§4Vz§4V{
 SD:70400010  1234567C 1234567D 1234567E 1234567F §4V|§4V}§4V~§4V
 SD:70400020  12345680 12345681 12345682 12345683 §4V§4V§4V§4V
 SD:70400030  12345684 12345685 12345686 12345687 §4V§4V§4V§4V
 SD:70400040  12345688 12345689 1234568A 1234568B §4V§4V§4V§4V
 SD:70400050  1234568C 1234568D 1234568E 1234568F §4V§4V§4V§4V
 SD:70400060  12345690 12345691 12345692 12345693 §4V§4V§4V§4V
 SD:70400070  12345694 12345695 12345696 12345697 §4V§4V§4V§4V
 SD:70400080  12345698 12345699 1234569A 1234569B §4V§4V§4V§4V
 SD:70400090  1234569C 1234569D 1234569E 1234569F §4V§4V§4V§4V
 SD:704000A0  123456A0 123456A1 123456A2 123456A3 §4V§4V§4V§4V
 SD:704000B0  123456A4 123456A5 123456A6 123456A7 §4V§4V§4V§4V
 SD:704000C0  123456A8 123456A9 123456AA 123456AB §4V§4V§4V§4V
 SD:704000D0  123456AC 123456AD 123456AE 123456AF §4V§4V§4V§4V
 SD:704000E0  123456B0 123456B1 123456B2 123456B3 §4V§4V§4V§4V
 SD:704000F0  123456B4 123456B5 123456B6 123456B7 §4V§4V§4V§4V
 SD:70400100  123456B8 123456B9 123456BA 123456BB §4V§4V§4V§4V
 SD:70400110  123456BC 123456BD 123456BE 123456BF §4V§4V§4V§4V
 SD:70400120  123456C0 123456C1 123456C2 123456C3 §4V§4V§4V§4V
 SD:70400130  123456C4 123456C5 123456C6 123456C7 §4V§4V§4V§4V
 SD:70400140  123456C8 123456C9 123456CA 123456CB §4V§4V§4V§4V
 SD:70400150  123456CC 123456CD 123456CE 123456CF §4V§4V§4V§4V
 SD:70400160  123456D0 123456D1 123456D2 123456D3 §4V§4V§4V§4V
 SD:70400170  123456D4 123456D5 123456D6 123456D7 §4V§4V§4V§4V
 SD:70400180  123456D8 123456D9 123456DA 123456DB §4V§4V§4V§4V
```

8.1.3 Indirect mode example

In this mode, the following two APIs are implemented to write and read on the device by using the registers. This test just performs the write operation reading back the value. It fails in case just one bit differs.

```
void octalspi_HyperbusMemWrite(uint32_t *buffer, int size, uint32_t addr)
{
    int i;
    octalspi_HyperbusCommand(SPC5_OSPI_MEMTYPE_HYPERBUS_RAM, size, addr);
    octalspi->CR.B.FMODE = OSPI_FMODE_INDIRECT_WRITE;
    while (!(octalspi->SR.B.FTF)) {};

    for (i=0; i< size / (sizeof(uint32_t)); i++) {
        *((uint32_t *) &octalspi->DR.R) = buffer[i];
    }
   /* In polling mode checks the transfer status */
   octalspi_xfer_completed();
}

void octalspi_HyperbusMemRead(uint32_t *buffer, int size, uint32_t addr)
{
    int i;
    octalspi_HyperbusCommand(SPC5_OSPI_MEMTYPE_HYPERBUS_RAM, size, addr);

    octalspi->CR.B.FMODE = OSPI_FMODE_INDIRECT_READ;
    addr = octalspi->AR.R;
    octalspi->AR.R = addr;

    while (!(octalspi->SR.B.FTF)) {};

    for (i=0; i< size / (sizeof(uint32_t)); i++) {
        buffer[i] = *((uint32_t *) &octalspi->DR.R);
    }
    octalspi_xfer_completed();
}
```

The following code is to check if the transfer has been completed (no interrupt is used in this case):

```
static inline void octalspi_xfer_completed(void)
{
while (!octalspi->SR.B.TCF) {};
octalspi->FCR.B.CTCF = 1;
}
```

The `octalspi_HyperbusCommand` is invoked to prepare the transfer of data from/to the memory by programming the MTYP field in the device configuration register 1.

```
static void octalspi_HyperbusCommand(uint8_t memtype, uint32_t data_length, uint32_t address)
{
     while (octalspi->SR.B.BUSY){};
    /* Functional mode */
    octalspi->CR.B.FMODE = OSPI_FMODE_INDIRECT_WRITE;
    /* Memory Type to access on register or memory address */
    octalspi->DCR1.B.MTYP = memtype;
    /* Specify a number of data bytes to read or write in the OCTOSPI_DLR */
    octalspi_ccr();
    octalspi->DLR.R = data_length-1;
    /* Specify the targeted address in the OCTOSPI_AR */
    octalspi->AR.R = address;
}
```

8.2 Cypress low level driver

A low level driver is necessary to program the S71KL256SC0BHB000 device.

The designed driver provides a set of basic APIs to access the registers and implement a part of the protocol.

8.2.1 Cypress S27K HyperRAM

There are two identification read only, non-volatile, word registers, that provide information on the device plus two registers used for configuring power mode and access protocol operating conditions.

The following code is used to access these registers:

```
static void cypress_read_config_reg(void)
{
    uint16_t cfg0;
    uint16_t cfg1;

    cfg0 = octalspi_HyperbusRegRead(CYPRESS_CFG0_REG);
    cfg1 = octalspi_HyperbusRegRead(CYPRESS_CFG1_REG);
…
```

While the identification information can be read as shown below:

```
id0 = octalspi_HyperbusRegRead(CYPRESS_ID0_REG);
       id1 = octalspi_HyperbusRegRead(CYPRESS_ID1_REG);
```

The offset of the mentioned registers is:

```
#define CYPRESS_ID0_REG 0x0
#define CYPRESS_ID1_REG 0x2
#define CYPRESS_CFG0_REG 0x1000
#define CYPRESS_CFG1_REG 0x1002
```

Through the configuration register 0 it is possible to define the operating conditions for the HyperRAM device and setting latency and initial latency modes or entering in power down.

8.2.2 Cypress S26K HyperFlash

To test the S26K HyperFLASH, the application invokes the cypress low level APIs for example to reset, erase and program the device.

The S26K does not use a traditional serial flash command protocol so the regular command protocol is not implemented.

To perform each operation the software must implement the device protocol to send the commands to the device in HyperBUS.

The cypress Programming Guide documents all supported commands, so this paragraph reports how the application implements one of them (for reference).

The code below shows the Cypress Macro Erase (Chip Erase) Command Sequence List. According to the protocol, this is completed in 6 cycles with a sequence of ADDRESS/DATA. The octalspi_HyperbusMemWrite8 is invoked to access the device according to the APIs mentioned in the previous sections.

```
/*
 *___________________________________________________________________
 *        1st         2nd       3rd         4th         5th        6th
 *___________________________________________________________________
 *     555h|00AAh 2AAh|0055h 555h|0080h 555|00AAh 2AAh|0055h 2AAh|0010
 */
void cypress_flash_erase(void)
{
    int i;
    uint8_t *status;
    uint32_t erase_address[6] = { ADDR_PATTERN_1, ADDR_PATTERN_2,
           ADDR_PATTERN_1, ADDR_PATTERN_1,
          ADDR_PATTERN_2, ADDR_PATTERN_1};

    uint8_t  erase_data[(2*6)] = {DATA_PATTERN_1, 0x00, DATA_PATTERN_2, 0x00,
             ERASE_CMD, 0x00, DATA_PATTERN_1, 0x00,
                     DATA_PATTERN_2, 0x00,
                     CHIP_ERASE_CMD, 0x00};

     for (i=0; i < 6; i++) {
           octalspi_HyperbusMemWrite8(&erase_data[2*i], 2, erase_address[i]);
     }
     do {
        status = cypress_read_status();
     } while(!(status[0] & 0x80));    /* READY BIT SR[7] */

     printf("S26K HyperFlash erase phase completed\r\n");
}
```

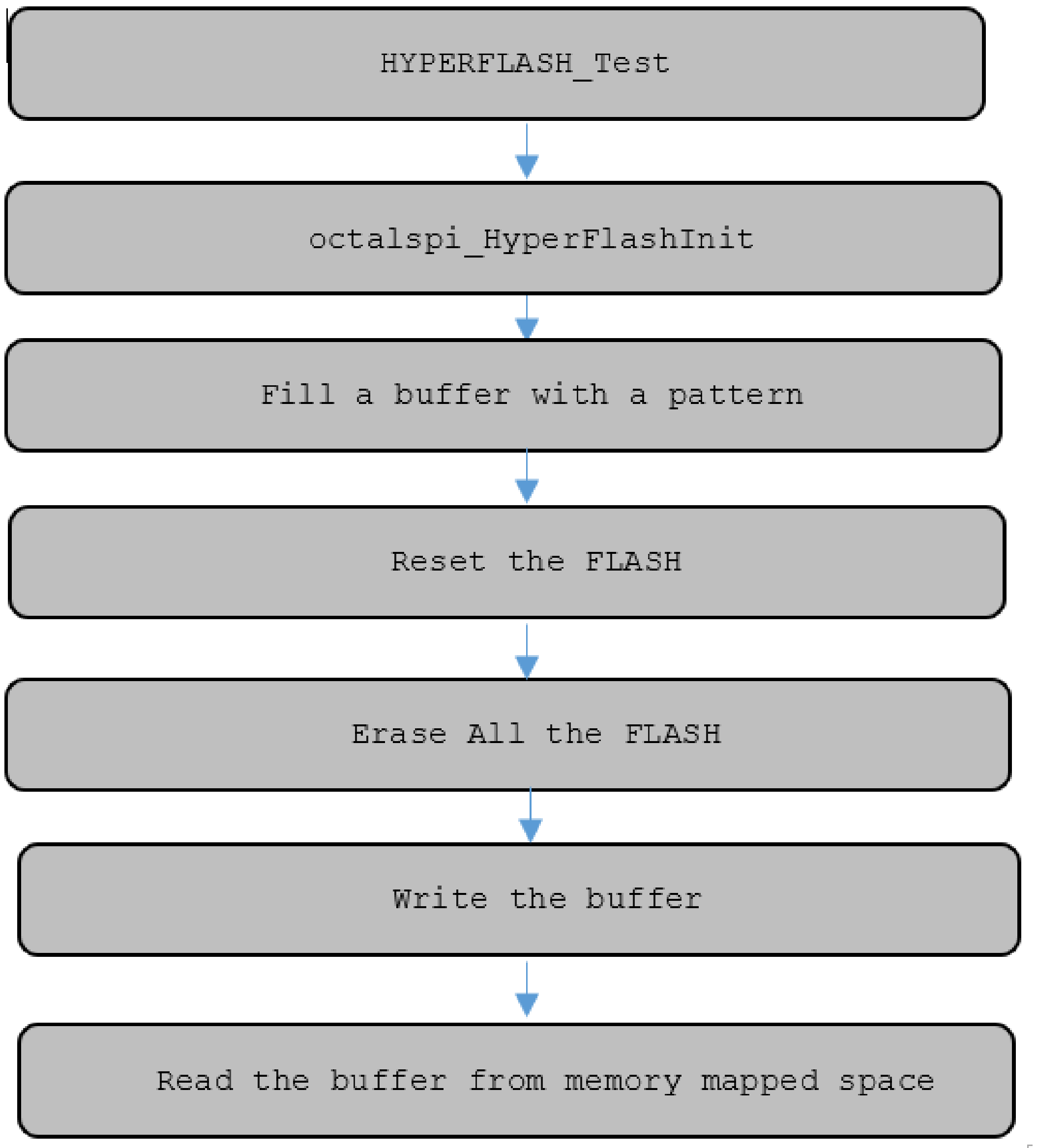
Where:

```
#define ADDR_PATTERN_1 (0x555 << 1)
          #define ADDR_PATTERN_2 (0x2AA << 1)
          #define DATA_PATTERN_1  0xAA
          #define DATA_PATTERN_2  0x55
```

8.3 HyperFLASH application

After configuring the OctalSPI to use the HyperFLASH, the applications have to reset and erase the whole flash. Then a user buffer is filled and written to the device. The application reads back the buffer from memory map and fails if any bits differ.

Figure 13. HyperFLASH block schema



8.4 HyperRAM XIP test

The execute in place (XIP) operation is implemented in this architecture. The XIP feature allows to execute programs directly from the external device.

This paragraph aims at showing how to test this kind of feature where a program is placed in HyperRAM.

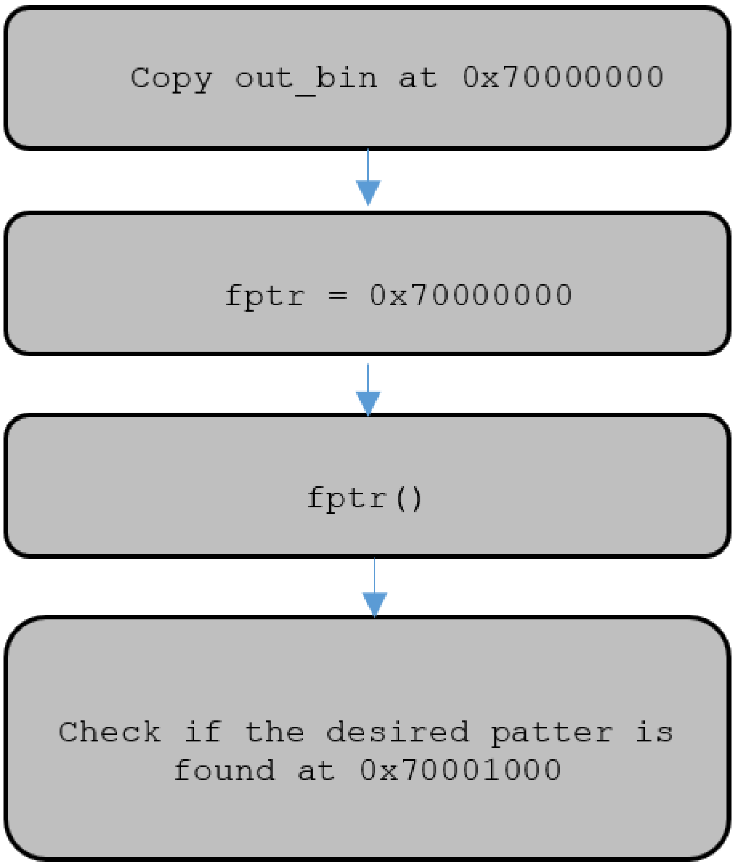
The first step is to prepare an application that will be copied in the external memory. For this purpose, a very simple main has been developed: it just writes a pattern to a known address, e.g.: 0x70001000.

Then its binary data are converted into a source header file:

```
static const unsigned char out_bin[] = {
    0x70, 0x1e, 0xe0, 0xcb, 0x71, 0x82, 0x00, 0x00, 0x1c, 0x00, 0xcc, 0x1a,
    0x71, 0x60, 0x00, 0x64, 0x19, 0x8c, 0x83, 0x70, 0x7d, 0x69, 0x03, 0xa6,
    0x19, 0x8c, 0x84, 0xfc, 0x18, 0x0c, 0x06, 0x04, 0x20, 0x00, 0x7a, 0x20,
    0xff, 0xfa, 0x48, 0x03, 0x00, 0x04, 0x00, 0xb3, 0x00, 0x07, 0x18, 0x00,
    0xd0, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x03, 0xe8
};
```

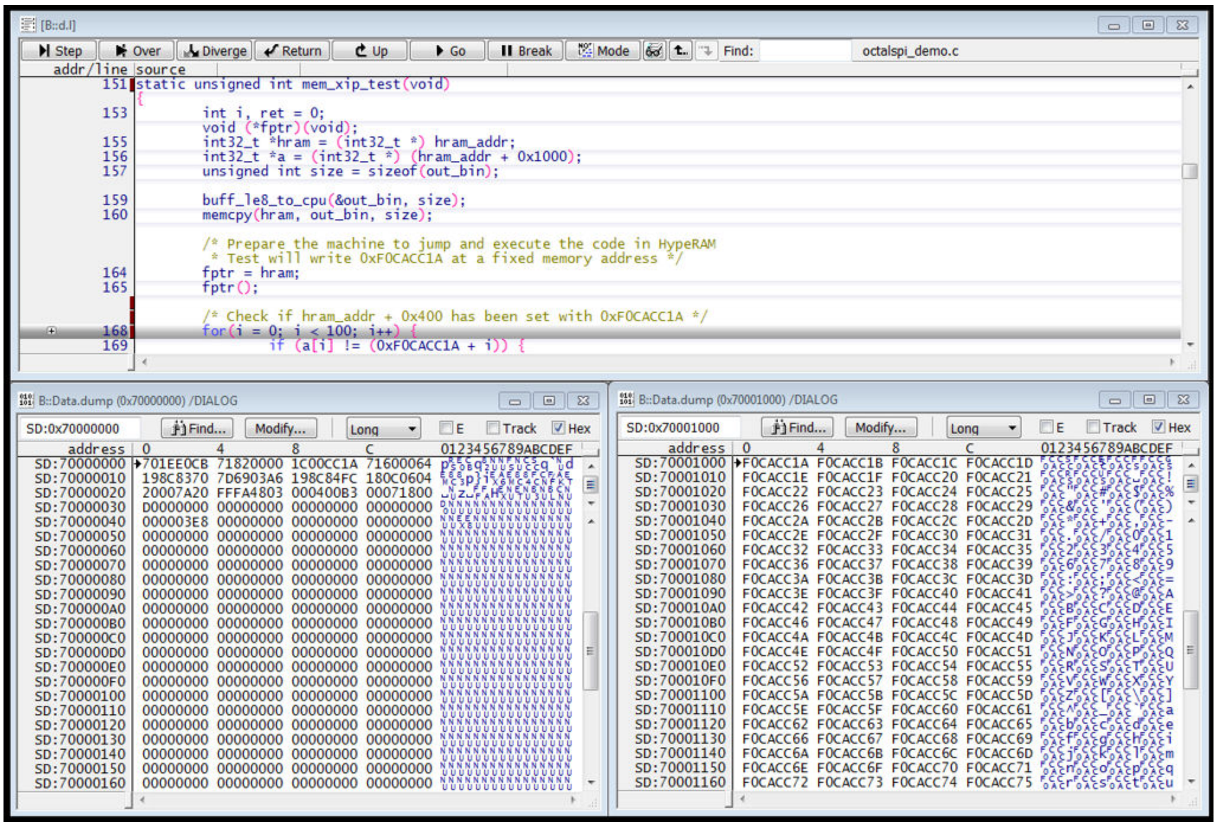
This header file can be included in the OctalSPI application and, at runtime, the application copies it inside the HyperRAM at the following address: 0x70000000. It is necessary to take care about the endianness.

Figure 14. XIP block schema


Copy out_bin at 0x70000000
fptr = 0x70000000
fptr()
Check if the desired patter is found at 0x70001000

After copying the binary in the selected portion of the external device, the code can jump to it (see block schema above). As soon as the `out_bin` code is executed, the program counter will keep on the caller. The following debug session shows the result of the execution of the `out_bin`.

Figure 15. XIP test result



```
static unsigned int mem_xip_test(void)
{
        int i, ret = 0;
        void (*fptr)(void);
        int32_t *hram = (int32_t *) hram_addr;
        int32_t *a = (int32_t *) (hram_addr + 0x1000);
        unsigned int size = sizeof(out_bin);

        buff_le8_to_cpu(&out_bin, size);
        memcpy(hram, out_bin, size);

        /* Prepare the machine to jump and execute the code in HypeRAM
         * Test will write 0xF0CACC1A at a fixed memory address */
        fptr = hram;
        fptr();

        /* Check if hram_addr + 0x400 has been set with 0xF0CACC1A */
        for(i = 0; i < 100; i++) {
                if (a[i] != (0xF0CACC1A + i)) {
```

```
B::Data.dump (0x70000000) /DIALOG
SD:0x70000000
     address  0        4        8        C
SD:70000000  701EE0CB 71820000 1C00CC1A 71600064
SD:70000010  198C8370 7D6903A6 198C84FC 180C0604
SD:70000020  20007A20 FFFA4803 000400B3 00071800
SD:70000030  D0000000 00000000 00000000 00000000
SD:70000040  000003E8 00000000 00000000 00000000
SD:70000050  00000000 00000000 00000000 00000000
SD:70000060  00000000 00000000 00000000 00000000
SD:70000070  00000000 00000000 00000000 00000000
SD:70000080  00000000 00000000 00000000 00000000
SD:70000090  00000000 00000000 00000000 00000000
SD:700000A0  00000000 00000000 00000000 00000000
SD:700000B0  00000000 00000000 00000000 00000000
SD:700000C0  00000000 00000000 00000000 00000000
SD:700000D0  00000000 00000000 00000000 00000000
SD:700000E0  00000000 00000000 00000000 00000000
SD:700000F0  00000000 00000000 00000000 00000000
SD:70000100  00000000 00000000 00000000 00000000
SD:70000110  00000000 00000000 00000000 00000000
SD:70000120  00000000 00000000 00000000 00000000
SD:70000130  00000000 00000000 00000000 00000000
SD:70000140  00000000 00000000 00000000 00000000
SD:70000150  00000000 00000000 00000000 00000000
SD:70000160  00000000 00000000 00000000 00000000
```

```
B::Data.dump (0x70001000) /DIALOG
SD:0x70001000
     address  0        4        8        C
SD:70001000  F0CACC1A F0CACC1B F0CACC1C F0CACC1D
SD:70001010  F0CACC1E F0CACC1F F0CACC20 F0CACC21
SD:70001020  F0CACC22 F0CACC23 F0CACC24 F0CACC25
SD:70001030  F0CACC26 F0CACC27 F0CACC28 F0CACC29
SD:70001040  F0CACC2A F0CACC2B F0CACC2C F0CACC2D
SD:70001050  F0CACC2E F0CACC2F F0CACC30 F0CACC31
SD:70001060  F0CACC32 F0CACC33 F0CACC34 F0CACC35
SD:70001070  F0CACC36 F0CACC37 F0CACC38 F0CACC39
SD:70001080  F0CACC3A F0CACC3B F0CACC3C F0CACC3D
SD:70001090  F0CACC3E F0CACC3F F0CACC40 F0CACC41
SD:700010A0  F0CACC42 F0CACC43 F0CACC44 F0CACC45
SD:700010B0  F0CACC46 F0CACC47 F0CACC48 F0CACC49
SD:700010C0  F0CACC4A F0CACC4B F0CACC4C F0CACC4D
SD:700010D0  F0CACC4E F0CACC4F F0CACC50 F0CACC51
SD:700010E0  F0CACC52 F0CACC53 F0CACC54 F0CACC55
SD:700010F0  F0CACC56 F0CACC57 F0CACC58 F0CACC59
SD:70001100  F0CACC5A F0CACC5B F0CACC5C F0CACC5D
SD:70001110  F0CACC5E F0CACC5F F0CACC60 F0CACC61
SD:70001120  F0CACC62 F0CACC63 F0CACC64 F0CACC65
SD:70001130  F0CACC66 F0CACC67 F0CACC68 F0CACC69
SD:70001140  F0CACC6A F0CACC6B F0CACC6C F0CACC6D
SD:70001150  F0CACC6E F0CACC6F F0CACC70 F0CACC71
SD:70001160  F0CACC72 F0CACC73 F0CACC74 F0CACC75
```

8.5 Interrupts handling

Table 5. OctalSPI interrupt shows the interrupt mapping for the two controllers.

Table 5. OctalSPI interrupt

	OctalSPI1	OctalSPI2
IRQ Name	**Number**	
Global Interrupt	186	192
Timeout	187	193
Status match	188	194
FIFO threshold	189	195
Transfer complete	190	196
Transfer error	191	197

The global interrupt vector handles the other five interrupts. The following code is to install the global interrupt inside the SPC5Studio tool.

```
#define    SPC5_OCTALSPI_1_HANDLER        vector186
#define    SPC5_OCTALSPI_2_HANDLER        vector192
#define    SPC5_OCTALSPI_1_NUMBER        186
#define    SPC5_OCTALSPI_2_NUMBER        192
#define    SPC5_OCTALSPI_PRIORITY              INTC_PSR_ENABLE(INTC_PSR_CORE2, 10)

IRQ_HANDLER(SPC5_OCTALSPI_1_HANDLER) {
    IRQ_PROLOGUE();
    osalEnterCriticalFromISR();
    octalspi_main_isr();        /* Interrupt Service Routine */
    osalExitCriticalFromISR();
    IRQ_EPILOGUE();
}
…
void octalspi_irq_install(unsigned int instance)
{
    if (instance == 1)
        INTC_PSR ( SPC5_OCTALSPI_1_NUMBER ) = SPC5_OCTALSPI_PRIORITY;
    else
        INTC_PSR ( SPC5_OCTALSPI_2_NUMBER ) = SPC5_OCTALSPI_PRIORITY;

    /* Enable the IRQ events */
    octalspi_EnableIRQ();
}
```

Where the `octalspi_main_isr` is the low-level APIs that just enables, inside the control register (CR) the related IRQ, i.e.:

- Timeout interrupt enable (`TOIE`)
- Status match interrupt enable (`SMIE`)
- FIFO threshold interrupt enable (`FTIE`)
- Transfer complete interrupt enable (`TCIE`)
- Transfer error interrupt enable (`TEIE`)

Note: refer to OctalSPI related guide inside the Reference manual for a complete description of the bits mentioned above (see Section Appendix C Reference documents).

Note: in the examples reported in the previous chapters the transfer status is checked in polling mode to better show the flow of each single operation.
The application configuration adopts a specific defines macro to enable / disable interrupt support.

8.6 Automatic polling mode

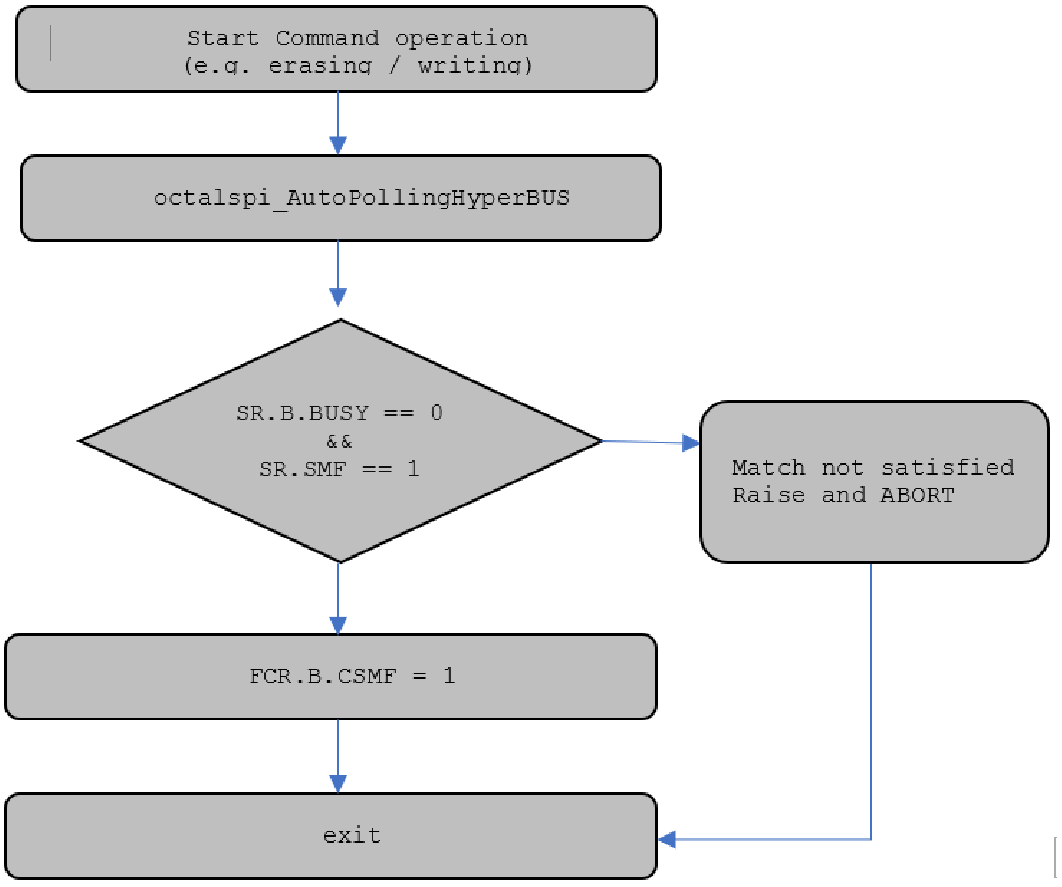
The status match flag is set in automatic-polling mode when the unmasked received data match the corresponding bits in the match register (OCTOSPI_PSMAR). This paragraph shows an example code on how to configure this mode (HyperBUS / Regular Command protocols) and also shows a diagram that aims at detailing the logic flow.

```
uint32_t octalspi_AutoPollingHyperBUS(ospi_polling_cfg_t *cfg, uint32_t poll_addr)
{
    /*
     *  If PSMAR.B.MASK[n] = 1 and the content of bit[n] is the same as
     *  PSMAR.B.MATCH[n] so the match is satisfied.
     *
     *  If PSMAR.B.MASK[n] = 0, so the bit n is masked and not considered.
     */
    octalspi->PSMAR.B.MATCH = cfg->Match;
    octalspi->PSMKR.B.MASK = cfg->Mask;
    octalspi->PIR.B.INTERVAL = cfg->Interval;

    octalspi->CR.B.FMODE = OSPI_FMODE_AUTO_POLLING;
    /* 1: stop after a match */
    octalspi->CR.B.APMS = cfg->AutomaticStop;
    /*
     * Match Mode: 0: AND, 1: OR
     *
     * In AND mode means that only when there is a match on all of the
     * unmasked bits, then the status match flag (SMF) is set.
     *
     * then “OR” match mode is activated, which means that the SMF gets set if
     * there is  a match on any of the unmasked bits.
     *
     * BUSY goes to 0 as soon as a match is detected
     *
     * OCTOSPI_DLR = 1, 8bit
     */
    octalspi->CR.B.PMM = cfg->MatchMode;

    /* The maximum amount of data read in each frame is 4 bytes. If more data is
       requested in OCTOSPI_DLR, it will be ignored and only 4 bytes will be
       read.
     */
    octalspi->DLR.R = 3;
    /* The command sequence starts as soon as the address is updated
     * with a write to OCTOSPI_AR */
    /* Targeted address is saved in the OCTOSPI_AR */
    octalspi->AR.R = poll_addr;

    /* Note: In automatic polling mode, OCTOSPI_DR contains the last data read
       from the external device (without masking).

       Same can be done by using IR register.
     */
}
```

Note: *When programming the match mask and read the DR register, the endianess format must be considered because the data could be swapped due to a different flash device endianess.*

Note: *Used values:* `#define DLYB_MAX_UNIT 127U; #define DLYB_CFGR_LNG_11 0x800U`

Figure 16. Automatic polling mode (generic flow diagram)



Note: *The reference manual gives a detailed description of the matching flag programming and the controller status on each state of the polling mode (see Section Appendix C Reference documents).*

8.7 Application output

On the serial console the following output messages are logged for both OctalSPI 1 and OctalSPI2 tests.

Note: *After the configuration of the peripheral and delay block, all the test can be executed in one or more loops.*

8.7.1 HyperRAM test output

```
======================
 OctalSPI Application
======================
OCTALSPI_1 (0x80000000 - 0x70000000)

OctalSPI version: 0x22, ID 0x140041, Magic ID 0xA3C5DD01
device configured as Master
Memory mapped write present
AHB interface
No memory mapped write 8 byte
DelayBlock_Enable (OCTSPI_1)
Delay Block Tuning completed
HyperBus Configuration
 - Read write recovery time = 4
 - Device access time = 6
 - no latency on Write
 - Fixed latency Mode
 Found Cypress Manufacturer ID
HyperRAM mode selected
 - 64 Mbit
 Card Configuration reg: 0x8F17, 0x2
HyperBus Configuration
 - Read write recovery time = 4
 - Device access time = 6
 - latency on Write
 - Variable latency Mode

 -- S27K HyperRAM Test --
 TEST 1: filled 131072 byte of memory (0x70000000)
 TEST 1: memcpy check PASSED
 TEST 2: Quick Sort Test
 TEST 2: QuickSort PASSED
 TEST 3: transfer through registers PASSED
DelayBlock_Disable (OCTSPI_1)

 Reconfiguring the Hyperbus ...
 OctalSPI version: 0x22, ID 0x140041, Magic ID 0xA3C5DD01
 device configured as Master
 Memory mapped write present
 AHB interface
 No memory mapped write 8 byte
DelayBlock_Enable (OCTSPI_1)
Delay Block Tuning completed
 TEST 4: XIP test: PASSED
DelayBlock_Disable (OCTSPI_1)
```

8.7.2 HypeFLASH test output

```
OCTALSPI_2 (0x80001000 - 0x60000000)
 -- S26K HyperFlash Test --
 OctalSPI version: 0x22, ID 0x140041, Magic ID 0xA3C5DD01
 device configured as Slave
 Memory mapped write present
 AHB interface
 No memory mapped write 8 byte
DelayBlock_Enable (OCTSPI_2)
Delay Block Tuning completed
HyperBus Configuration
 - Read write recovery time = 0
 - Device access time = 16
 - no latency on Write
 - Variable latency Mode
S26K HyperFlash reset done
S26K HyperFlash erase phase completed
 [Writing data 0x528003C0 (512 bytes)]
S26K HyperFlash write test done: 512 bytes
>>> Reconfiguring the Hyperbus for memory mapped addressing...
HyperBus Configuration
 - Read write recovery time = 0
 - Device access time = 16
 - no latency on Write
 - Variable latency Mode
DelayBlock_Disable (OCTSPI_2)
TEST: memory mapped data are coherent

Test Completed!
```

Appendix A HyperBus timing

The commands, the addresses and data are transferred over the eight DATA [7:0] signals. The clock is used for information capture by the device. The command or address values are aligned with clock transitions which start with the assertion of CS# and Command-Address (CA) signals. These are followed by the start of clock transitions to transfer six command bytes. Then there is the initial access latency and either read or write data transfers, until the chip select is de-asserted.

The read and write operations need to respect two timings: tRWR that is the minimal read/write recovery time for the device and tACC used as access time for the device. Both must be programmed in the OctalSPI registers as shown below:

```
octalspi->HLCR.B.TRWR = config->rwRecoveryTime;
octalspi->HLCR.B.TACC = config->accessTime;
```

The DRAM cells cannot be refreshed during a read or write transactions: a host is required to limit the duration of transactions and allow an additional initial access latency as well. An additional latency is before tACC and after RWR and it is driven by the RWDS signal.

If a variable latency is selected, a latency for a refresh is only added when a refresh is required at the same time a new transaction is starting.

Figure 17. Read operation with an initial latency

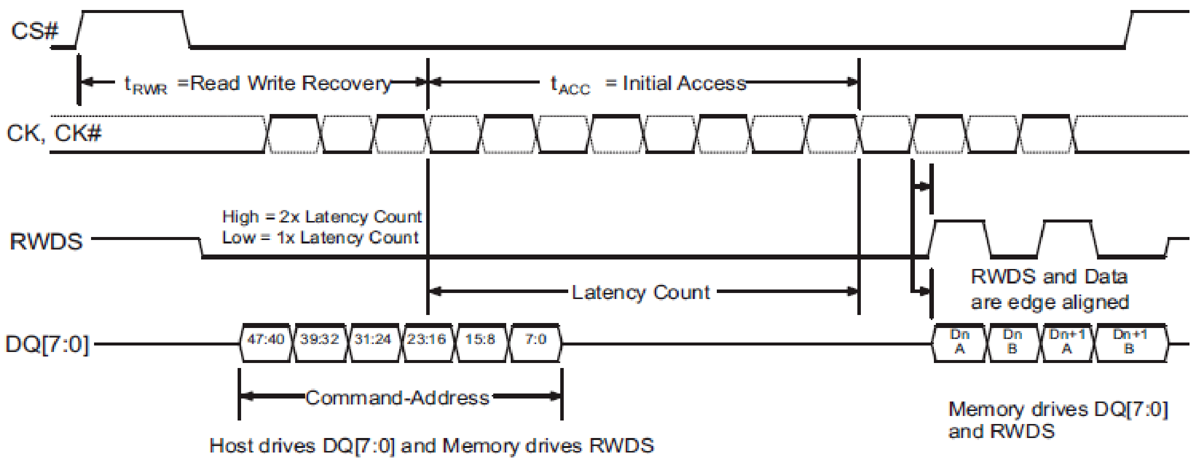
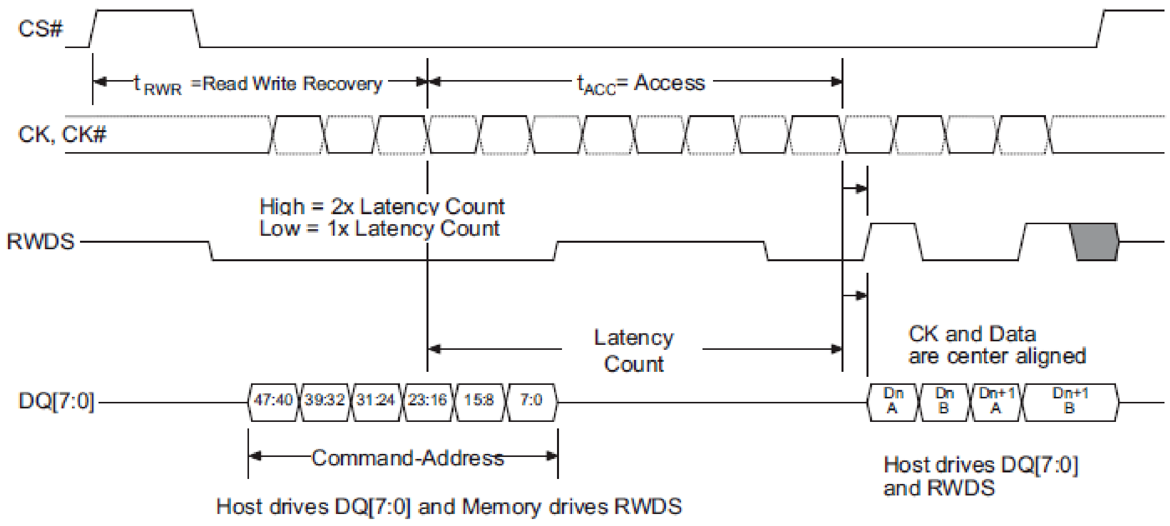


Figure 18. Write operation with an initial latency



It is possible to force the latency to be fixed to 2 x tACC by programming the following register:

```
octalspi->HLCR.B.LM = 1;
```

The state of the RWDS signal is not considered in this mode. If a fixed latency is selected the memory will always indicate a refresh latency and delay the read data transfer accordingly.

Appendix B Acronyms and abbreviations

Table 6. Acronyms

Abbreviation	Complete name
SIUL2	System integration unit lite 2
CGM	Clock Generation Module
XIP	Execution in place
SRAM	System RAM
PCM	Platform configuration module
PSR	Protocol Status Register
SDR/DDR	Single / Double Data Rate

Appendix C Reference documents

Table 7. Reference documents

Name	Title
RM0452	SPC58EHx/SPC58NHx 32-bit Power Architecture microcontroller
DS12304	SPC58EHx/SPC58NHx Errata sheet, Datasheet and IO map excel file
-	S27KL0641/S27KS0641 S70KL1281/S70KS1281 3.0 V/1.8 V, 64 Mbit (8 Mbyte)/128 Mbit (16 Mbyte), HyperRAM™ Self-Refresh DRAM
-	S71KL256SC0_S71KL512SC0, HYPERFLASH AND HYPERRAM MULTI CHIP PACKAGE 3 V

Revision history

Table 8. Document revision history

Date	Version	Changes
05-Nov-2020	1	Initial release.
05-Mar-2024	2	Updated Table 4. Hold margin according to a DQS delayed by + 0.8 ns; Section 7.3 Delay block length tuning. Added Section 7.3.1 . Minor text changes.

Contents

List of tables

List of figures