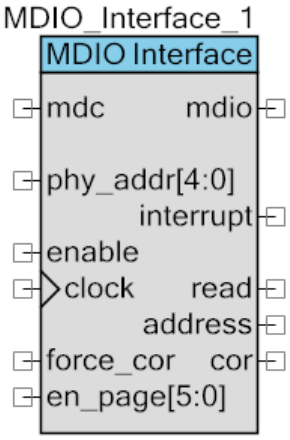
MDIO Interface

1.0

Features



- MDIO Interface component to be used in conjunction with Ethernet products
- Configurable physical address
- Supports up to 4.4 MHz in the clock bus (mdc)
- Compliant with IEEE 802.3 Clause 45
- Automatically allocates memory for the register spaces that can be configured through an intuitive, easy-to-use graphical configuration GUI

General Description

The MDIO_Interface component is an interface component that supports the Management Data Input/Output, which is a serial bus defined for the Ethernet family of IEEE 802.3 standards for the Media Independent Interface (MII). The MII connects Media Access Control (MAC) devices with Ethernet physical later (PHY) circuits. The component is compliant with IEEE 802.3 Clause 45.

When to Use an MDIO Interface

Use the MDIO_Interface component in a PHY management interface to read and write the PHY control and status registers. They configure each PHY before operation and monitor link status during operation.

The component can be configured to generate an interrupt for any frame received from the MDIO bus and lets the user implement their own data handling algorithm in firmware (Basic Mode); or it can be configured to automatically handle all the registers in hardware without CPU intervention (Advanced Mode).

Input/Output Connections

This section describes the various input and output connections for the MDIO Interface component. An asterisk (*) in the list of I/Os indicates that the I/O may be hidden on the symbol under the conditions listed in the description of that I/O.

mdc – Input

MDC is the bus clock provided by the MDIO Host. It is directly connected to the physical MDC input pin.

clock – Input

This clock is used for internal logic timing. When the component operates in the **Basic mode**, the clock rate must be set at least 8x the MDC bus clock. In the **Advanced mode** it is recommended to clock the component from BUS_CLK configured to at least 40 MHz

phy_addr[4:0] – Input *

Physical port address. Displays if a **Hardware** option is set for the **Physical address** parameter.

mdio – Inout *

A bidirectional pin that transmits or receives data. The pin connected to mdio should be configured as Open Drain Low and the Input Sync Mode should be configured as Double-Sync. It displays if the **Enable external OE** parameter is unchecked.

mdio_in – Input *

MDIO signal driven by the Host. Displays when the **Enable External OE** option is checked.

mdio_out – Output *

MDIO signal driven by the MDIO Interface. Displays when the **Enable External OE** option is checked.

enable – Input

Synchronous active high enable signal.

interrupt – Output

When configured in the **Basic mode**, this output generates a pulse in the end of the frame only if the physical address and device address match the pre-configured values.

In **Advanced mode**, a pulse is generated when the MDIO Host finishes a writing operation and the associated register is configured to trigger interrupt on write.

force_cor – Input *

Force a clear on read for the current MDIO address. Displays if an **Advanced** option is set for the **Configuration** parameter.

en_page[x] – Input *

Enable/disable the corresponding register space. When the page is disabled, the component drives low during the data portion of a read frame and ignores the data of a write frame. The width of this signal depends on the number of register spaces managed by the component. Displays if an **Advanced** option is set for the **Configuration** parameter.

read – Output *

This output generates a pulse when the MDIO Host finishes a reading operation and the associated register is configured to trigger interrupt on read. Displays if an **Advanced** option is set for the **Configuration** parameter.

address – Output *

This output generates a pulse in the end of every address frame sent by the MDIO Host. Displays if an **Advanced** option is set for the **Configuration** parameter.

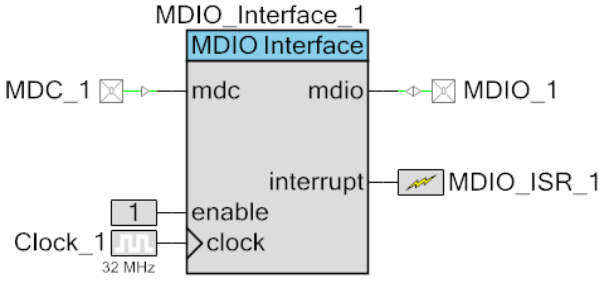
cor – Output *

This output generates a pulse when a read frame is received and the register is configured to be cleared on read. Displays if an **Advanced** option is set for the **Configuration** parameter.

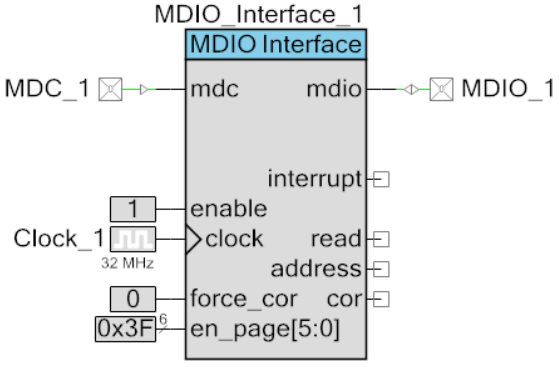
Schematic Macro Information

The PSoC Creator Component Catalog provides two macros for the Basic and Advanced mode operation as shown in the following diagrams. These macros contain the MDIO Interface component already connected to digital pin, clock and interrupt components. The generation of APIs for all of the pins is turned off.

Basic Mode



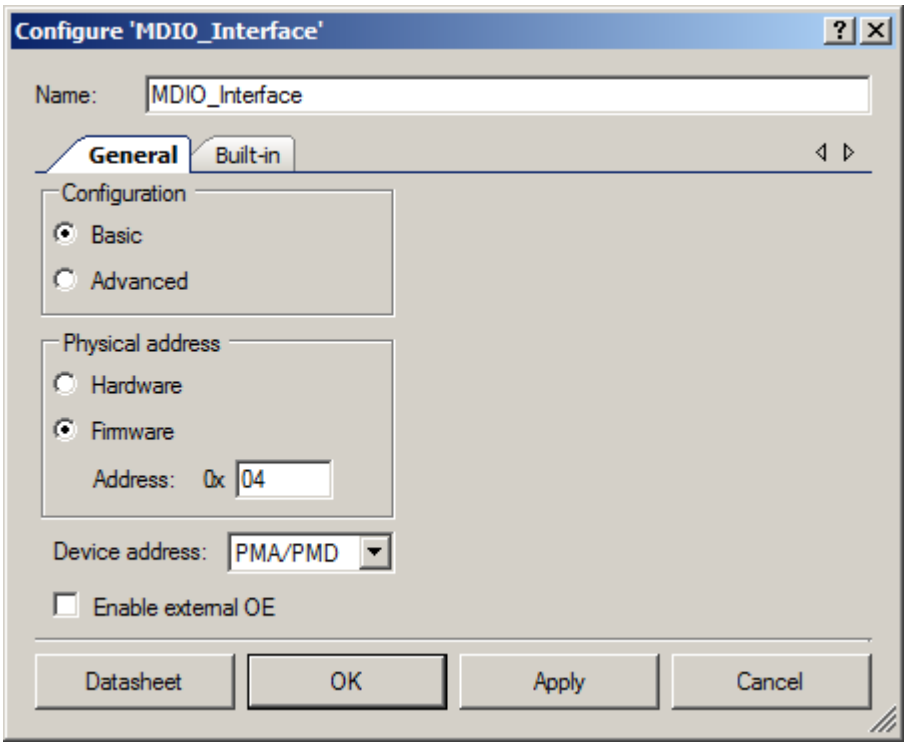
Advanced Mode



Component Parameters

Drag an MDIO Interface component onto your design and double-click it to open the **Configure** dialog.

Figure 1. General Tab



Configuration

Determines which mode the component operates – **Basic** (default) or **Advanced**.

Physical Address

Specifies whether the physical address is updated using **phy_addr[4:0]** port address bus or firmware API, and the default physical address if a **Firmware** option is set. Options for the address mode are **Firmware** (default) or **Hardware**. The address value can be set between 0 and 0x1F. The default setting is **0x04**.

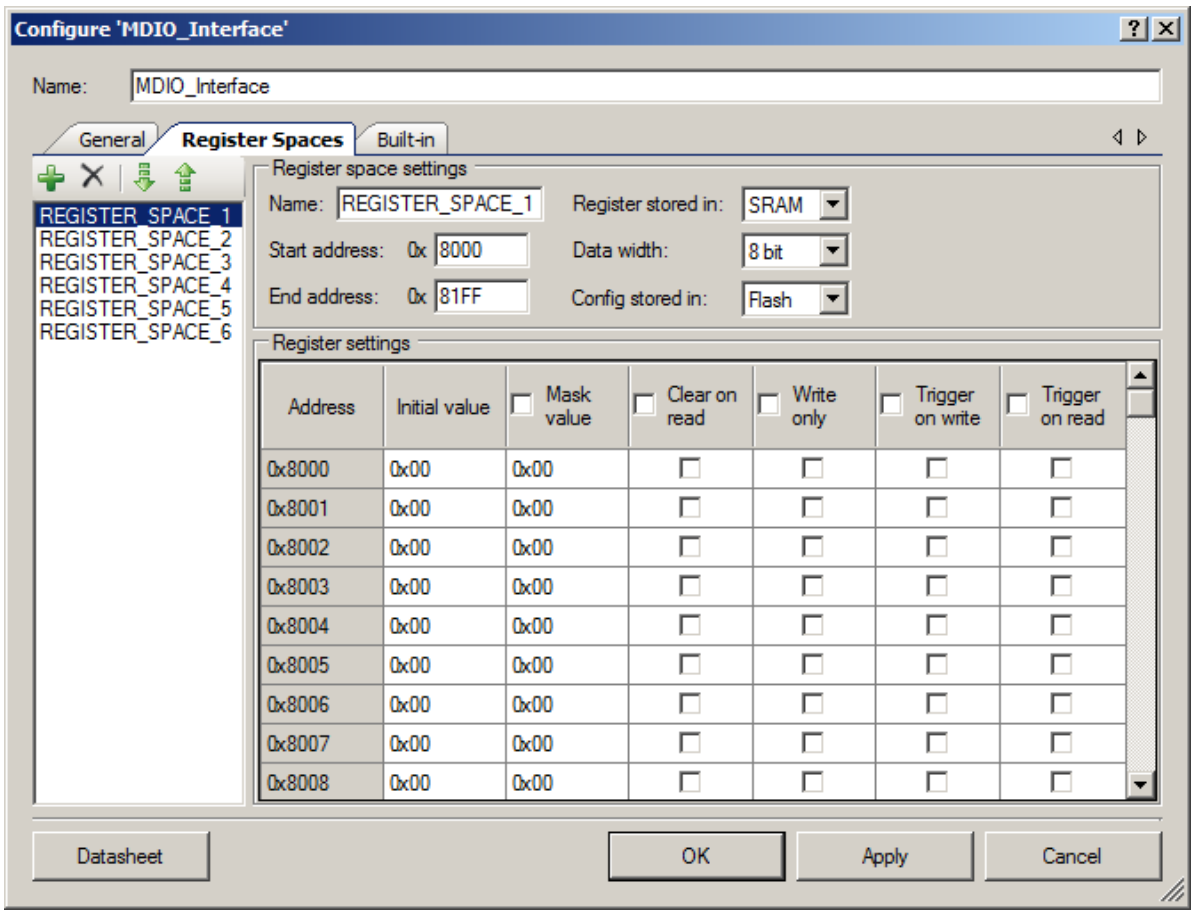
Device Address

Specifies the device type of the MDIO Interface. The value can be set to **PMA/PMD**, **WIS**, **PCS**, **PHY XS** or **DTE XS**. The default setting is **PMA/PMD**.

Enable external OE

Allows to split the **mdio** bidirectional in the **mdio_in** input and **mdio_out** output signals. Options = Checked or unchecked. Enabling this option exposes the **mdio** inout to **mdio_in** input and **mdio_out** output on the symbol. (Default = unchecked).

Figure 2. Register Spaces Tab



This tab is available only in **Advanced mode** and follows the CFP Register Allocation specified in the CFP MSA specification. The tab allows allocating CFP register spaces and configuring parameters for each single register that belongs to the allocated register spaces.

Name

Text field, 24 characters, used to create MACROs for the quick access register space start address and end address. By default this field is labeled **REGISTER_SPACE_N** (where N=1..**Number of register spaces**).

Start address

Defines the start address in hex for each of the allocated register spaces. The value can be set from 0x8000 to 0xFFFF.

End address

Defines the end address in hex for each of the allocated register spaces. The value can be set from 0x8000 to 0xFFFF and must be larger than the start address of the associated register space.

Register stored in

Determines the memory location of the register space. The register space data can be stored in either **Flash** or **SRAM** (default).

Data width

Specifies if the data is 8 bit or 16 bit width.

Config stored in

Determines the memory location of the configuration data associated with the register space. The register space configuration can be stored in either **Flash** (default) or **SRAM**.

Note that the configuration of each register space contains four bytes of parameters for each register in this register space.

Initial value

Defines the register initial value in hex. Can be 8-bit or 16-bit value depending on the **Data width** parameter for the associated register space. (Default = 0).

Mask value

Defines the register writable mask bit in hex. Can be 8-bit or 16-bit value depending on the **Data width** parameter for the associated register space. Ignored if the associated register space is located in Flash. (Default = 0).

Clear on read

Defines the register to be Clear-on-Read (COR). Options = Checked or unchecked. If checked, the register is cleared to 0 upon MDIO Host read. Ignored if the associated register space is located in Flash. (Default = Unchecked).

Note that Clear-on-Read is 16-bit wide operation. If an 8-bit register is configured to be COR, reading of this register will clear its value and the value stored in subsequent memory address.

Write only

Defines the register to be Write Only (WO). Options = Checked or unchecked. If the MDIO Host reads a WO register, 0xFFFF is returned for the data portion of a MDIO frame. Ignored if the associated register space is located in Flash. (Default = Unchecked).

Trigger on write

Defines whether the **interrupt** output generates a pulse at the end of a write operation by the MDIO Host to the corresponding register. Options = Checked or unchecked. Ignored if the associated register space is located in Flash. (Default = Unchecked).

Trigger on read

Defines whether the **read** output generates a pulse at the end of a read operation by the MDIO Host to the corresponding register. Options = Checked or unchecked. Ignored if the associated register space is located in Flash. (Default = Unchecked).

Clock Selection

There is no internal clock in this component. You must attach a clock source. When the component operates in **Basic mode**, the clock rate provided must be at least 8x the desired MDC clock rate. In **Advanced mode** it is recommended to clock the component from BUS_CLK and configure BUS_CLK to be at least 40 MHz

Application Programming Interface

Application Programming Interface (API) routines allow you to configure the component using software. The following table lists and describes the interface to each function. The subsequent sections discuss each function in more detail.

By default, PSoC Creator assigns the instance name “MDIO_Interface_1” to the first instance of a component in a given design. You can rename the instance to any unique value that follows the syntactic rules for identifiers. The instance name becomes the prefix of every global function name, variable, and constant symbol. For readability, the instance name used in the following table is “MDIO_Interface.”

Function	Description
MDIO_Interface_Start()	Initializes and enables the MDIO Interface.
MDIO_Interface_Stop()	Disables the MDIO Interface.
MDIO_Interface_Init()	Initializes default configuration provided with customizer.
MDIO_Interface_Enable()	Enables the MDIO Interface.
MDIO_Interface_EnableInt()	Enables the interrupt output terminal.
MDIO_Interface_DisableInt()	Disables the interrupt output terminal.
MDIO_Interface_SetPhyAddress()	Sets the 5-bit physical address for the MDIO Interface.
MDIO_Interface_UpdatePhyAddress()	Updates the physical address of the MDIO Interface.
MDIO_Interface_SetDevAddress()	Sets the 5-bit device address for the MDIO Interface.

Function	Description
MDIO_Interface_GetData()	Returns the value stored in the given address.
MDIO_Interface_SetData()	Sets the argument value in the given address.
MDIO_Interface_SetBits()	Sets the specified bits at the specified address.
MDIO_Interface_GetAddress()	Returns the last address written by the MDIO Host.
MDIO_Interface_GetConfiguration()	Returns a pointer to the configuration array of the given register space.
MDIO_Interface_PutData()	Sets the data to be transmitted to the MDIO Host.
MDIO_Interface_ProcessFrame()	Processes the last frame received from the MDIO Host.
MDIO_Interface_Sleep()	Stops the MDIO Interface and saves the user configuration.
MDIO_Interface_Wakeup()	Restores the user configuration and enables the MDIO Interface.
MDIO_Interface_SaveConfig()	Saves the current user configuration.
MDIO_Interface_RestoreConfig()	Restores the user configuration.

Global Variables

Variable	Description
MDIO_Interface_initVar	MDIO_Interface_initVar indicates whether the MDIO Interface has been initialized. The variable is initialized to 0 and set to 1 the first time MDIO_Interface_Start() is called. This allows the component to restart without reinitialization after the first call to the MDIO_Interface_Start() routine. If reinitialization of the component is required, then the MDIO_Interface_Init() function can be called before the MDIO_Interface_Start() or MDIO_Interface_Enable() function.

MDIO_Interface_registerConfig_N[]
(Advanced mode only)
	Stores register configuration associated with the register space N. N=1..8 – the number of register spaces allocated by the component. None of these arrays are accessible directly by user firmware.
	Note that the configuration of each register space contains four bytes of parameters for each register in this register space. These parameters are stored in the C structure declared as follows:
	typedef struct
	{
	uint16 mask; /* 16 bit writable mask */
	uint8 ctrlReg; /* Register configuration */
	uint8 reserved; /* Reserved */
	}
	For each allocated register space, the MDIO_Interface_registerConfig_N[] is initialized based on the register settings in customizer. Usually there is no need to access or modify these arrays.

Variable	Description
MDIO_Interface_registerSpace_N[] (Advanced mode only)	Stores register values for the register space N. N=1..8 – the number of register spaces allocated by the component. None of these arrays are accessible directly by user firmware. The MDIO_GetData() function can be used to obtain the register values, and MDIO_SetData() function to modify the register values within the MDIO_Interface_registerSpace_N[] array.

void MDIO_Interface_Start(void)

Description: This is the preferred method to begin component operation. This function sets the initVar variable, calls the MDIO_Interface_Init() function, and then calls the MDIO_Interface_Enable() function.

Parameters: None

Return Value: None

Side Effects: If the initVar variable is already set, this function only calls the MDIO_Interface_Enable() function.

void MDIO_Interface_Stop(void)

Description: Disables the MDIO Interface. If the component is configured to operate in Advanced mode, this function disables all internal DMA channels.

Parameters: None

Return Value: None

Side Effects: None

void MDIO_Interface_Init(void)

Description: Initializes or restores default MDIO Interface configuration provided with customizer. Initializes internal DMA channels if the component is configured to operate in Advanced mode. It is not necessary to call MDIO_Interface_Init() because the MDIO_Interface_Start() routine calls this function, which is the preferred method to begin component operation.

Parameters: None

Return Value: None

Side Effects: None

void MDIO_Interface_Enable(void)

Description: Activates the hardware and begins component operation. It is not necessary to call MDIO_Interface_Enable() because the MDIO_Interface_Start() routine calls this function, which is the preferred method to begin component operation.

Parameters: None

Return Value: None

Side Effects: Enables the terminal output interrupt if it was disabled by MDIO_Interface_DisableInt() API.

void MDIO_Interface_EnableInt(void)

Description: Enables the terminal output interrupt.

Parameters: None

Return Value: None

Side Effects: None

void MDIO_Interface_DisableInt(void)

Description: Disables the terminal output interrupt.

Parameters: None

Return Value: None

Side Effects: None

void MDIO_Interface_SetPhyAddress(uint8 phyAddr)

Description: Sets the 5-bit physical address for the MDIO Interface.

Parameters: uint8 phyAddr: The physical address value.

Return Value: None

Side Effects: Overwrites the default physical address defined in the customizer.

void MDIO_Interface_UpdatePhyAddress(void)

Description: Updates the physical address based on the current phy_addr[4:0] from the component. If a Firmware option is set for the Physical address mode parameter, the address is set equal to the default value from the customizer.

Parameters: None

Return Value: None

Side Effects: None

void MDIO_Interface_SetDevAddress(uint8 devAddr)

Description: Sets the 5-bit device address for the MDIO Interface.

Parameters: uint8 devAddr: The device address value.

Return Value: None

Side Effects: Overwrites the default device address defined in the customizer.

uint8 MDIO_Interface_GetData(uint16 address, uint16 *regData, uint16 numWords)

Description: Returns N values starting from the given address. If any address does not belong to the allocated register space, it returns an error. This API is only available in Advanced mode.

Parameters: uint16 address: address to be accessed.
uint16 *regData: array with the read data.
uint16 numWords: number of words to be read.

Return Value: uint8: 0 on success
1 on failure

Side Effects: None

uint8 MDIO_Interface_SetData(uint16 address, const uint16 *regData, uint16 numWords)

Description: Writes N values starting from the given address. If any address does not belong to the allocated register space or the register space is located in Flash, it returns an error. This API is only available in Advanced mode and if at least one register space is located in SRAM.

Parameters: uint16 address: address to be accessed.
const uint16 *regData: array with the written data.
uint16 numWords: number of words to be written.

Return Value: uint8: 0 on success
1 on failure

Side Effects: None

uint8 MDIO_Interface_SetBits(uint16 address, uint16 regBits)

Description: Sets the bits at the given address. If the address does not belong to the allocated register space or the register space is located in Flash, it returns an error. This API is only available in Advanced mode and if at least one register space is located in SRAM.

Parameters: uint16 address: address to be accessed.
uint16 regBits: bits to be set.

Return Value: uint8: 0 on success
1 on failure

Side Effects: None

uint16 MDIO_Interface_GetAddress(void)

Description: Returns the last address written by the MDIO Host.

Parameters: None

Return Value: uint16: the address.

Side Effects: None

uint8 MDIO_Interface_GetConfiguration(uint8 regSpace)

Description: Returns a pointer to the configuration array of the given register space.

Parameters: uint8 regSpace: the register space index.

Return Value: uint8*: pointer to the configuration array.

Side Effects: None

uint8 MDIO_Interface_PutData(uint16 regData)

Description: Writes the given data in the internal FIFO, which will be transmitted to the host in the next frame. Only available in Basic mode.

Parameters: uint16 regData: data to be transmitted.

Return Value: None

Side Effects: None

void MDIO_Interface_ProcessFrame(uint8* opCode, uint16* regData)

Description: Processes and parses the last frame received from the host. Only available in Basic mode.

Parameters: uint8* opCode: operational code.

uint16* regData: register data received.

Return Value: None

Side Effects: None

void MDIO_Interface_Sleep(void)

Description: This is the preferred routine to prepare the component for sleep. The MDIO_Interface_Sleep() routine saves the current component state. Then it calls the MDIO_Interface_Stop() function and calls MDIO_Interface_SaveConfig() to save the hardware configuration.

Call the MDIO_Interface_Sleep() function before calling the CyPmSleep() or the CyPmHibernate() function. See the PSoC Creator *System Reference Guide* for more information about power-management functions.

Parameters: None

Return Value: None

Side Effects: None

void MDIO_Interface_Wakeup(void)

Description: This is the preferred routine to restore the component to the state when MDIO_Interface_Sleep() was called. The MDIO_Interface_Wakeup() function calls the MDIO_Interface_RestoreConfig() function to restore the configuration. If the component was enabled before the MDIO_Interface_Sleep() function was called, the MDIO_Interface_Wakeup() function also re-enables the component.

Parameters: None

Return Value: None

Side Effects: Calling the MDIO_Interface_Wakeup() function without first calling the MDIO_Interface_Sleep() or MDIO_Interface_SaveConfig() function can produce unexpected behavior.

void MDIO_Interface_SaveConfig(void)

Description: This function saves the component configuration and nonretention registers. It also saves the current component parameter values, as defined in the Configure dialog or as modified by appropriate APIs. This function is called by the MDIO_Interface_Sleep() function.

Parameters: None

Return Value: None

Side Effects: None

void MDIO_Interface_RestoreConfig(void)

Description: This function restores the component configuration and nonretention registers. It also restores the component parameter values to what they were before calling the MDIO_Interface_Sleep() function.

Parameters: None

Return Value: None

Side Effects: If this API is called before MDIO_Interface_SaveConfig(), the component configurations will be restored to their default settings.

API Constants

Variable[1]	Description
MDIO_Interface_ADDRESS	Operation code for an address frame
MDIO_Interface_WRITE	Operation code for a write frame
MDIO_Interface_READ	Operation code for a read frame

1. RegSpaceName – the name entered in customizer for a given register space.

Variable[1]	Description
MDIO_Interface_POS_READ	Operation code for a post read frame
MDIO_Interface_NUMBER_OF_PAGES	Number of allocated register spaces
MDIO_Interface_RegSpaceName_IDX	Index of the Register Space
MDIO_Interface_RegSpaceName_START	Starting address of the Register Space
MDIO_Interface_RegSpaceName_END	Ending address of the Register Space

MISRA Compliance

This section describes the MISRA-C:2004 compliance and deviations for the component. There are two types of deviations defined:

- project deviations – deviations that are applicable for all PSoC Creator components
- specific deviations – deviations that are applicable only for this component

This section provides information on component-specific deviations. Project deviations are described in the MISRA Compliance section of the *System Reference Guide* along with information on the MISRA compliance verification environment.

The MDIO Interface component has the following specific deviations:

MISRA-C:2004 Rule	Rule Class (Required/ Advisory)	Rule Description	Description of Deviation(s)
10.1	R	The value of an expression of integer type shall not be implicitly converted to a different underlying type if: a) it is not a conversion to a wider integer type of the same signedness, or b) the expression is complex, or c) the expression is not constant and is a function argument, or d) the expression is not constant and is a return expression.	The DMA component provides general integer type definitions.
11.4	A	A cast should not be performed between a pointer to object type and a different pointer to object type.	The casts between a pointer to object type and a different pointer to object type is used to map a CFP register space to the component internal memory when configuring DMA data movement.

MISRA-C:2004 Rule	Rule Class (Required/ Advisory)	Rule Description	Description of Deviation(s)
13.2	A	Tests of a value against zero should be made explicit, unless the operand is effectively Boolean.	The DMA component provides general integer type definitions which are ORed together to provide correct function argument. The definitions are positive numbers and operation result does not cause any side effects.
16.7	A	A pointer parameter in a function prototype should be declared as pointer to const if the pointer is not used to modify the addressed object.	The pointer addresses the object that is modified by a DMA transfer configured in this API.
17.4	R	Array indexing shall be the only allowed form of pointer arithmetic.	To access DMA registers the pointer arithmetic with array indexing is used. This type of access is provided by DMA component.
19.7	A	A function should be used in preference to a function-like macro.	Deviations with function-like macros to allow for more efficient code. The component allocates 8 or 16-bit registers in Flash or SRAM. Macros with arguments are used to quickly determine register width and memory location.

This component has the following embedded components: DMA. Refer to the corresponding component datasheet for information on their MISRA compliance and specific deviations.

Sample Firmware Source Code

PSoC Creator provides many example projects that include schematics and example code in the Find Example Project dialog. For component-specific examples, open the dialog from the Component Catalog or an instance of the component in a schematic. For general examples, open the dialog from the Start Page or **File** menu. As needed, use the **Filter Options** in the dialog to narrow the list of projects available to select.

Refer to the “Find Example Project” topic in the PSoC Creator Help for more information.

References

CFP MSA Management Interface Specification v2.0

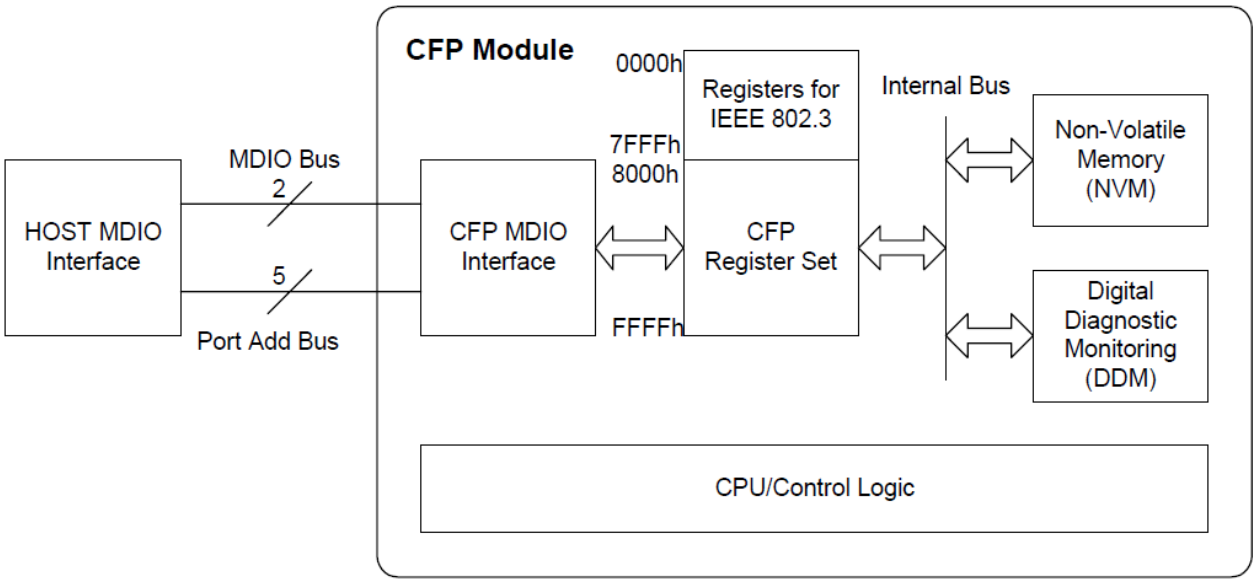
CFP MSA Hardware Specification v1.4

Functional Description

CFP Module overview

The MDIO_Slave component supports the IEEE 802.3 standards for the Media Independent (MII) clause 45 specification. It is used in agreement with the CFP Management Interface, which specifies the use of MDIO bus as the management interface between a host and a CFP module. The block diagram of the CFP module is shown in the following figure.

Figure 3. CFP Module Block Diagram



The MDIO Host uses this interface to control and monitor the startup, shutdown and normal operation of the CFP modules it manages. At the same bus, multiple slaves can be connected to the same host with different bus addresses. The address of the module can be configured through the Port Address bus.

The component interfaces with the host through two lines: a data line designated as MDIO bus, and a clock line designated as MDC. The MDIO bus is bidirectional and is transmitted at the rate specified in MDC. The MDC clock then is driven by the host to control the communication rate.

Once enabled, the MDIO Interface component continuously monitors and communicates with the MDIO Host device. In basic mode, the end of each MDIO frame triggers an interrupt that allows the user to transfer this captured data in the FIFO to the system memory by using the necessary API. The advanced mode examines the packet information and allows the packet transfers to take place to and from the system memory.

When running in advance mode, the MDIO component allocates CFP register table automatically based on the customizer settings. The component supports the following register access types:

- Support Read-Only (RO) registers;
- Support Read-Write (R/W) registers;

- Support RO & R/W mixed within a 16-bit word environment;
- Support Clear on Read (COR);
- Support Write-Only (WO) registers;

All these features are handled in hardware when the MDIO Host sends a read/write frame.

If the MDIO Host reads a Write only register, 0xFFFF is returned to the Host. If the MDIO Host writes in a read only register or in a read only bits, the whole frame or the read only bits are ignored be the MDIO Interface component.

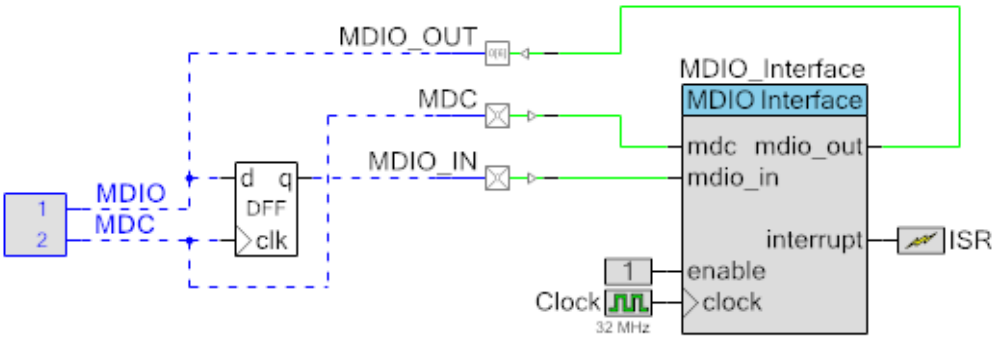
Any access to addresses between 0x0000 and 0x7FFF is ignored (no activity in the MDIO bus).

Exposing bidirectional mdio pin in to mdio_in and mdio_out terminals

The component can be configured to expose the **mdio** bidirectional terminal in to two terminals – **mdio_in** and **mdio_out**. This might be useful when an external DFF is used in order to reduce the overall hold time of the MDIO bus and meet the timing requirements from the MDIO specification. Another useful situation is when the MDIO bus should be muxed with multiple MDIO slaves. To do that, select an **Enable external OE** check box on the **General** tab of **Configure** dialog.

When using an external DFF, an extra pin should be used in order to separate the MDIO input and output signals. The figure below shows how the DFF should be connected to the PSoC pins. Note that the MDIO_OUT pin should be configured as **Open Drain, Drives Low** digital output. The MDIO_IN and MDC_IN should be configured as **High Impedance** digital inputs.

Figure 4. Example Using an External DFF



Block Diagram and Configuration

The MDIO Interface component is implemented as a set of configured UDBs. When configured in Advanced mode, the component uses DMAs to make the transfers to and from the system SRAM/Flash. The implementation is shown in the following block diagrams.

Figure 5. MDIO Interface (Basic mode)

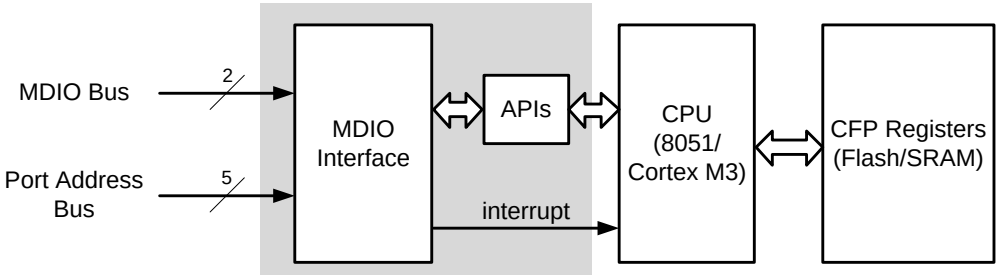
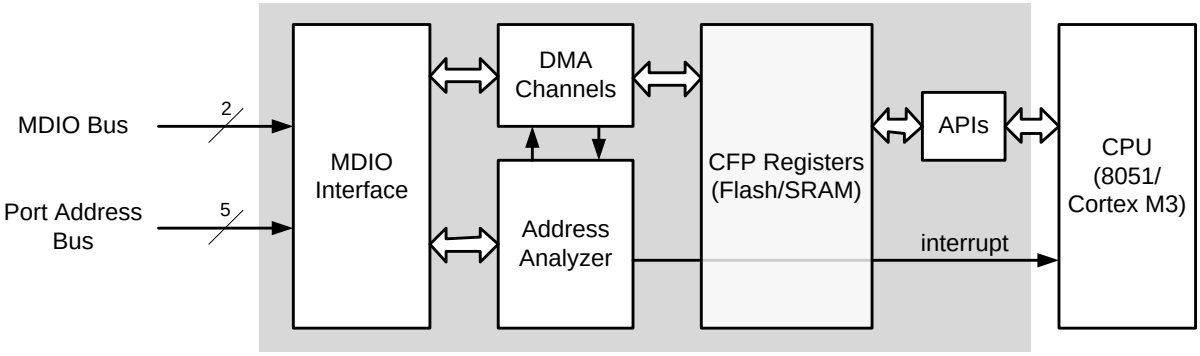


Figure 6. MDIO Interface (Advanced mode)



Registers

The MDIO Interface component has several control and status registers that are used by the firmware APIs to control operation and monitor status. None of these registers are accessible directly by user firmware.

Resources

The MDIO Interface component is placed throughout the UDB array. When configured in Advanced mode, the component uses DMAs to make the transfers to and from the system SRAM/Flash. The component utilizes the following resources.

Configuration	Resource Type					
	Datapath Cells	Macrocells	Status Cells	Control Cells	DMA Channels	Interrupts
Basic	2	14	1	2	–	–
Advanced	8	57	1	5	9 [2]	–

API Memory Usage

The component memory usage varies significantly, depending on the compiler, device, number of APIs used and component configuration. The following table provides the memory usage for all APIs available in the given component configuration.

The measurements have been done with associated compiler configured in Release mode with optimization set for Size. For a specific design, the map file generated by the compiler can be analyzed to determine the memory usage.

Configuration	PSoC 3 (Keil_PK51)		PSoC 5 (GCC)		PSoC 5LP (GCC)	
	Flash Bytes	SRAM Bytes	Flash Bytes	SRAM Bytes	Flash Bytes	SRAM Bytes
Basic	365	5	508	7	460	7
Advanced [3]	4.3K + R_{FLS}	53 + R_{SRAM}	3K + R_{FLS}	86 + R_{SRAM}	3K + R_{FLS}	86 + R_{SRAM}

2. PSoC 5/5LP implementation requires one additional DMA Channel. The total number of DMA channels is 10 for PSoC 5/5LP.

3. In advanced mode the component automatically allocates and manages CFP register table. R_{FLS} and R_{SRAM} are the Flash and SRAM memory usage to store the register data and configuration. They can be calculated using the following equations:
R_{SRAM} = 16*PAGE_NUM + REG_NUM_SRAM$_{8\text{-}BIT}$ + 2*REG_NUM_SRAM$_{16\text{-}BIT}$ + 4*REG_NUM_CFG_SRAM;
R_{FLS} = 8*PAGE_NUM + REG_NUM_FLS$_{8\text{-}BIT}$ + 2*REG_NUM_FLS$_{16\text{-}BIT}$ + 4*REG_NUM_CFG_FLS, where:
PAGE_NUM – Number of allocated register pages;
REG_NUM_SRAM$_{8\text{-}BIT}$ and REG_NUM_SRAM$_{16\text{-}BIT}$ – Number of 8 and 16-bit registers stored in SRAM;
REG_NUM_FLS$_{8\text{-}BIT}$ and REG_NUM_FLS$_{16\text{-}BIT}$ – Number of 8 and 16-bit registers stored in Flash;
REG_NUM_CFG_SRAM – Number of registers with configuration stored in SRAM;
REG_NUM_CFG_FLS – Number of registers with the configuration stored in Flash;

DC and AC Electrical Characteristics

Specifications are valid for –40 °C ≤ T_A ≤ 85 °C and T_J ≤ 100 °C, except where noted.
Specifications are valid for 1.71 V to 5.5 V, except where noted.

DC Characteristics

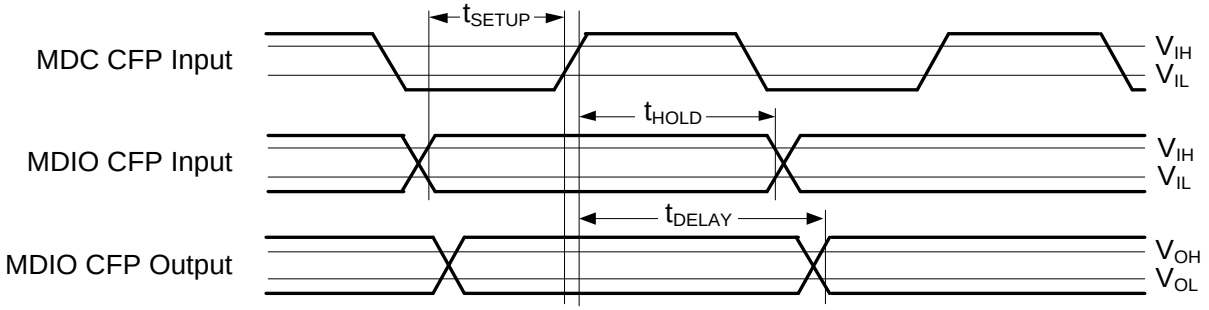
Parameter	Description	Min	Typ	Max	Unit
V_{IH}	Input high voltage	0.8	–	5.5	V
V_{IL}	Input low voltage	-0.5	–	0.42	V
V_{OH} [4]	Output high voltage (I_{OH} = -100 µA)	–	–	–	V
V_{OL}	Output low voltage (I_{OL} = 100 µA)	0.001	–	0.002	V
I_{OH} [5]	Output high current (V_{OH} = 1.0 V)	–	–	–	mA
I_{OL}	Output low current (V_{OL} = 0.2 V)	10	–	–	mA
C_i	Input Capacitance	–	–	7	pF

AC Characteristics

Parameter	Description	Min	Typ	Max	Units
f_{MDC}	MDC clock frequency	–	–	4.4	MHz
f_{CLOCK}	Component clock frequency	8 * f_{MDC}	42	–	MHz
t_{SETUP}	Host MDIO setup time	10	–	–	ns
t_{HOLD}	Host MDIO hold time				
	External OE unchecked	85	–	–	ns
	External OE checked [6]	2.4	–	–	ns
t_{DELAY}	CFP MDIO delay time	–	–	150	ns

4. PSoC output driver is open drain, so V_{OH} is determined by pullup resistor on the MDIO bus.
5. I_{OH} is not applicable to open drain drivers.
6. This parameter applies to an external D flip-flop to the MDIO bus. By default the component does not meet t_{HOLD} requirement as described in the CFP MSA Hardware specification. To meet the hold time specified in the CFP MSA Hardware specification, an external D flip-flop (DFF) such as the SN74AUP1G79 from Texas Instruments or the NC7SP74 from Fairchild Semiconductor is required to capture MDIO data from the bus and pass it on to PSoC. Refer to **Exposing bidirectional mdio pin in to mdio_in and mdio_out terminals** section for the description of proper component connection with an external DFF.

Figure 7. MDC/MDIO Timing Diagram



Component Changes

This section lists the major changes in the component from the previous version.

Version	Description of Changes	Reason for Changes/Impact
1.0	Version 1.0 is the first release of the MDIO Interface component	

© Cypress Semiconductor Corporation, 2013. The information contained herein is subject to change without notice. Cypress Semiconductor Corporation assumes no responsibility for the use of any circuitry other than circuitry embodied in a Cypress product. Nor does it convey or imply any license under patent or other rights. Cypress products are not warranted nor intended to be used for medical, life support, life saving, critical control or safety applications, unless pursuant to an express written agreement with Cypress. Furthermore, Cypress does not authorize its products for use as critical components in life-support systems where a malfunction or failure may reasonably be expected to result in significant injury to the user. The inclusion of Cypress products in life-support systems application implies that the manufacturer assumes all risk of such use and in doing so indemnifies Cypress against all charges.

PSoC® is a registered trademark, and PSoC Creator™ and Programmable System-on-Chip™ are trademarks of Cypress Semiconductor Corp. All other trademarks or registered trademarks referenced herein are property of the respective corporations.

Any Source Code (software and/or firmware) is owned by Cypress Semiconductor Corporation (Cypress) and is protected by and subject to worldwide patent protection (United States and foreign), United States copyright laws and international treaty provisions. Cypress hereby grants to licensee a personal, non-exclusive, non-transferable license to copy, use, modify, create derivative works of, and compile the Cypress Source Code and derivative works for the sole purpose of creating custom software and or firmware in support of licensee product to be used only in conjunction with a Cypress integrated circuit as specified in the applicable agreement. Any reproduction, modification, translation, compilation, or representation of this Source Code except as specified above is prohibited without the express written permission of Cypress.

Disclaimer: CYPRESS MAKES NO WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, WITH REGARD TO THIS MATERIAL, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. Cypress reserves the right to make changes without further notice to the materials described herein. Cypress does not assume any liability arising out of the application or use of any product or circuit described herein. Cypress does not authorize its products for use as critical components in life-support systems where a malfunction or failure may reasonably be expected to result in significant injury to the user. The inclusion of Cypress' product in a life-support systems application implies that the manufacturer assumes all risk of such use and in doing so indemnifies Cypress against all charges.

Use may be limited by and subject to the applicable Cypress software license agreement.