CODE TIME TECHNOLOGIES

Abassi RTOS

EMAC Support

Disclaimer

Table of Contents

List of Figures

List of Tables

1 Introduction

This document describes the EMAC driver used by Abassi[1] [R1] (including mAbassi [R2] and µAbassi [R3]). The EMAC driver, although primarily targeted for use with Abassi, is a standalone driver as it does not use any RTOS components.

1.1 Distribution Contents

The set of files supplied with this distribution are listed in Table 1-1 below:

Table 1-1 Distribution

File Name	Description
`???_ethernet.h`	Include file for the EMAC driver (??? is target dependent)
`???_ethernet.c`	"C" file for the EMAC driver (??? is target dependent)
`ethernet.c`	LwIP Ethernet interface supplied as a working example on how to use the EMAC driver.
`ethernet.h`	LwIP Ethernet interface supplied as a working example on how to use the EMAC driver.

1.2 Limitations

LwIP's `pbuf` direct support by the driver is not available on some target platform due to limitations on the EMAC capabilities. The driver does not support virtual memory remapping by the MMU; i.e. when processor memory areas are at different addresses than the physical memory is.

IMPORTANT: In the current implementation of the EMAC driver, the accesses to the EMAC registers through the macro `EMACreg()` are not protected against concurrent accesses (no mutexes, spinlocks, or interrupt disabling). This is by design because it is necessary to read and / or write EMACs registers in the EMAC interrupt handler. Isolated single read or write don't present an issue but grouped operations or read-modify-write could be problematic.

PHY registers accesses are mutex are protected. Therefore operations performing accesses to PHY registers cannot be used in an interrupt.

1.3 Features

The EMAC driver API is kept the same across all target platforms. Target specific extra functionality is not described in this document; refer to the code itself (either in the `??_ethernet.h` and / or `??_ethernet.c` files). There may also be some do-nothing functions when a platform specific EMAC does not need or does not support a feature.

[1] When Abassi is mentioned in this document, unless explicitly stated, it always means Abassi, mAbassi and µAbassi.

2 Target Set-up

All there is to do to configure and enable the use of the EMAC driver in an application based on Abassi is to include the following file in the build:

- `???_ethernet.c`

and set-up the include search directory order making sure the file `???_ethernet.h` is found.

The EMAC driver may or may not, depending on the target platform, be independent from other include files.

2.1 Build Options

There are a few build options that allow the EMAC driver to be configured for the needs of the target application. The following table lists all of them:

Table 2-1 Build Options

File Name	Default	Description
`OS_PLATFORM`	`0xAAC5`	Number indicating the target platform. The default value is Altera Arria V / Cyclone V. Refer to `???_ethernet.h` to see the list of supported platforms
`ETH_MAX_DEVICES`	`Target dependent`	Maximum number of EMACs module(s) supported by the platform. The default value is dependent on the build option `OS_PLATFORM`
`ETH_LIST_DEVICE`	`Target dependent`	Bit field selecting the EMACs module(s) to use. The default value is dependent on the build option `OS_PLATFORM`
`ETH_BUFFER_TYPE`	`ETH_BUFFER_UNCACHED`	The type of memory the EMACs DMA descriptors and buffers are located in
`ETH_RX_BUFSIZE`	`1536`	Size of the DMA payload buffers for reception
`ETH_TX_BUFSIZE`	`1536`	Size of the DMA payload buffers for transmission

`ETH_N_RXBUF`	`4: when uncached`
	`64: when cached`
		Number of DMA buffers in the reception chain

`ETH_N_TXBUF`	`4: when uncached`
	`64: when cached`
		Number of DMA buffers in the transmission chain

`ETH_MULTI_PHY`	`-1`	Control the use of single or multiple PHYs.
`ETH_ALT_PHY_IF`	`0`	Boolean enabling the use of an alternate PHY I/F in extra to the MDIO
`ETH_ALT_PHY_MTX`	`0`	Boolean selecting a global mutex to protect the access to the alternate PHY I/F instead of a per device mutex.
`ETH_OFF_DIR_RX`	`-1`	When applicable, sets-up the DMA direct cache accesses from the network to the cache
`ETH_OFF_DIR_TX`	`-1`	When applicable, sets-up the DMA direct cache accesses from the cache to the network

`ETH_DEBUG`	`0`	Boolean controlling the sending of progress / debug messages to `stdout`.
`ETH_DEBUG`	`0`	Boolean controlling the sending of progress / debug messages to `stdout`.
`ETH_#_SKEW_???`	`See Section 4`	Skews to apply on signals between the EMAC and the PHY. Symbol # is the EMAC device number and ??? specified the signal(s) to skew

2.1.1 OS_PLATFORM

The build option `OS_PLATFROM` informs the EMAC driver of which platform it is operating on. There are two benefits ensuing from the presence of this build option:

- The EMAC driver is able to configure and reset the EMAC devices without application intervention.
- Provides default setting for the `ETH_LIST_DEVICE` build option (see section 2.1.3).

2.1.2 ETH_MAX_DEVICES

The build option `ETH_MAX_DEVICES` informs the EMAC driver of how many EMAC devices are on the target platform. If this build option is not set, then the EMAC driver will rely on the build option `ETH_LIST_DEVICE` (Section 2.1.3). If the build option `ETH_LIST_DEVICE` is also not set, then the EMACS driver will rely on the `OS_PLATFORM` value (Section 2.1.1). When `ETH_MAX_DEVICES` is defined, `ETH_LIST_DEVICE` cannot be defined.

2.1.3 ETH_LIST_DEVICE

The build option `ETH_LIST_DEVICE` informs the EMAC driver of which EMAC modules are in use. When the target platform has multiple EMAC modules, enabling only the modules used by the application offers two benefits:

- Minimize the data memory used by the driver, as there is no need to reserve memory for the DMA descriptors and buffers of unused modules.
- When a single EMAC module is used, the EMAC registers are accessed through direct memory access, possibly making the driver more real-time efficient.

The build option is a bit field, where the bit position represents the EMAC module. When the corresponding bit is cleared to 0 it specifies the module is not used, when the bit is set to 1 then the module is used. The following table shows the combinations for a 3 module target platform:

Table 2-2 Build Options

ETH_LIST_DEVICE	EMAC #0	EMAC #1	EMAC #2
1	In use	Not used	Not used
2	Not used	In use	Not used
3	In use	In use	Not used
4	Not used	Not used	In use
5	In use	Not used	In use
6	Not used	In use	In use
7	In use	In use	In use

If the build option `ETH_LIST_DEVICE` is not externally defined, the default value will be set according to the build option `OS_PLATFORM` and will make all the EMAC modules on the target platform available. When `ETH_LIST_DEVICE` is defined, `ETH_MAX_DEVICES` (Section 2.1.2) cannot be defined.

2.1.4 ETH_BUFFER_TYPE

The EMAC driver is designed to operate with the controller DMA descriptors and DMA buffers in both regular non-cached memory and in cached memory. When used with cached memory, the driver performs all the required cache-invalidate and cache-flush operations to keep the DMA and the CPU in-sync when memory is updated by one or the other. The following table lists the values `ETH_BUFFER_TYPE` can be set to:

Table 2-3 Build Options

ETH_BUFFER_TYPE	Description
`ETH_BUFFER_UNCACHED`	The DMA descriptors and the DMA buffer are located in non-cached memory. The linker section `.uncached` MUST exist, be large enough to hold all DMA descriptors and DMA buffers, and be mapped into non-cached memory. This is the default value if `ETH_BUFFER_TYPE` is not externally defined.
`ETH_BUFFER_CACHED`	The DMA descriptors and the DMA buffers are located in cached memory. Depending on the target platform, it may not be possible to locate the DMA descriptors in cached memory if the DMA descriptors are not and cannot be set a size that is a multiple of the target platform cache line size. When this occurs, the DMA descriptors are located in the `.uncached` section (See `ETH_BUFFER_UNCACHED` for further information).
`ETH_BUFFER_PBUF`	This is alike `ETH_BUFFER_CACHED`, except for in the RX direction, the DMA buffers are not created, nor attached to the DMA descriptors by the EMAC driver. The attachment must be performed outside the EMAC driver. Only the RX direction can use supplied payload buffers and the size of the supplied payload buffers must be at least `ETH_RX_BUSIZE` (plus the padding size if needed) bytes.
`ETH_BUFFER_CACHED_DIRECT`	Same as `ETH_BUFFER_CACHED` with the difference no cache flushing and/or invalidation are done. This type of buffering is only available on target platforms with a mechanism for the EMAC DMA to directly access the cache contents.
`ETH_BUFFER_PBUF_DIRECT`	Same as `ETH_BUFFER_PBUF` with the difference no cache flushing and/or invalidation are done. This type of buffering is only available on target platforms with a mechanism for the EMAC DMA to directly access the cache contents.

NOTE: setting `ETH_BUFFER_TYPE` does not configure the cache. It simply informs the driver to use cache flushing and invalidation when the buffer type is set to "cached". When the buffer type is set to "cached", the memory section used by the DMA buffers is the same one used by all variables (e.g. for GCC it's `.bss`). For "un-cached" buffer, the buffers are assigned to a memory section named "`.uncached`". Therefore, the linker script file must include the section "`.uncached`" and that section must be located in a memory region set by the cache configuration as an "un-cached" memory. Not doing so will make the Ethernet driver fails. As the operations of cache flushing and invalidation are "do-nothing" when used on "un-cached" memory, it is safe to set `ETH_BUFFER_TYPE` set to `ETH_BUFFER_CACHED` or `ETH_BUFFER_PBUF` on "un-cached" buffers.

Depending on the target platform, even if `ETH_BUFFER_TYPE` is set to "cached", it is possible the DMA descriptors themselves still need to be located in "un-cached" memory. This happens when the individual DMA descriptors can't uniquely fit into an exact number of cache line size. For safety, the "`.uncached`" memory section should always be defined in the linker script and the cache configuration set-up accordingly.

NOTE: On Atera/Intel Arri V and Cyclone V platform the use of the `???_DIRECT` buffer types involve the DMA accessing the cache through the ACP mapper. As such, by default the EMAC driver sets-up the AVP mapper through the `acp_enable()` function. If the application itself uses and sets-up the ACP Mapper then refer to sections 2.1.9 and 2.1.10.

2.1.5 Buffer size and number of buffers

The build options `ETH_RX_BUFSIZE` and `ETH_TX_BUFSIZE` specify the size of the payload buffers the DMA uses. The build options `ETH_N_RXBUF` and `ETH_N_TXBUF` specify how many DMA payload buffers to use to create the circular chain of DMA buffers. The total amount of memory used for the payload in each direction, and per EMAC module is:

```
ETH_?X_BUFSIZE * ETH_N_?XBUF
```

2.1.6 ETH_MULTI_PHY

`ETH_MULTI_PHY` controls if the EMAC is connected to a single PHY or multiple ones. An example of multiple PHYs is a switch-PHY chip with a RGMII I/F to communicate with the EMAC and multiple PHYs connected to the switch. When `ETH_MULTI_PHY` is set to a negative (default value), or not defined, a single PHY is to be accessed by the EMAC. When set to a value between 0 and 31, the EMAC is informed it has to access multiple PHYs and the value assigned to `ETH_MULTI_PHY` is the address of the "master PHY". When `ETH_MULTI_PHY` is set to a value equal or larger than 32, the EMAC is informed it has to access multiple PHYs and the "master PHY" is set as the PHY with the lowest address. Refer to section for a detailed description on how the EMAC deal with multiple PHYs and what is the "master PHY".

2.1.7 ETH_ALT_PHY_IF

`ETH_ALT_PHY_IF` is a build option enabling the hook to access PHY communication with a different medium than MDIO. By default the hook is not called. To enable the EMAC driver to access PHYS with an alternate medium than MDIO, define the build option `ETH_ALT_PHY_IF` and set ti to a non-zero value. The interfacing with the hook is described in details in section 3.2.

2.1.8 ETH_ALT_PHY_MTX

The build option `ETH_ALT_PHY_MTX` is only recognized when the build option ETH_ALT_PHY_IF is defined and set to a value greater or equal to 0. `ETH_ALT_PHY_MTX` when not defined uses the same mutex (one per EMAC device) to protect the access to the `AltPhyIF()` attached function. If no mutexes protection is requited for `AltPhyIF(),` then define `ETH_ALT_PHY_MTX` and set it to a negative value. If a single mutex is needed to protect the access to `AltPhyIF()`, then define and set the build option to a positive value (See section 3.2 for why this may be needed). If the build option is defined and set to a value of 0 it is the same as if it hadn't been defined and one mutex per EMAC device is used for the protection.

2.1.9 ETH_OFF_DIR_RX

On the Altera/Intel Arria V and Cyclone V the EMAC DMA can access the processor cache contents when the DMA performs the data transfer through the ACP mapper. The ACP mapper needs to be set-up and the EMAC driver does it by default. If the application uses and sets-up the ACP mapper then it becomes necessary for the application to provide the EMAC driver with the address offset used during the address mapping. For the network to cache data transfers this is controlled through the build option `ETH_OFF_DIR_RX`. By default it is set to -1 informing the driver to set-up the ACP mapper itself using the ACP driver API `acp_enable().`

To inform the EMAC driver to not set-up the ACP then one way is to provide the address offset through the imported global variable `G_ETHoffDirRX`; this is achieved when the build option is set to a value of -2. The other way is to specify the address offset directly with `ETH_OFF_DIR_RX`; this happens when `ETH_OFF_DIR_RX` has a value other than -1 and -2.

2.1.10 ETH_OFF_DIR_TX

On the Altera/Intel Arria V and Cyclone V the EMAC DMA can access the processor cache contents when the DMA performs the data transfer through the ACP mapper. The ACP mapper needs to be set-up and the EMAC driver does it by default. If the application uses and sets-up the ACP mapper then it becomes necessary for the application to provide the EMAC driver with the address offset used during the address mapping. For the cache to networks data transfers this is controlled through the build option `ETH_OFF_DIR_TX`. By default it is set to -1 informing the driver to set-up the ACP mapper itself using the ACP driver API `acp_enable().`

To inform the EMAC driver to not set-up the ACP then one way is to provide the address offset through the imported global variable `G_ETHoffDirTX`; this is achieved when the build option is set to a value of -2. The other way is to specify the address offset directly with `ETH_OFF_DIR_TX`; this happens when `ETH_OFF_DIR_RX` has a value other than -1 and -2.

2.1.11 ETH_DEBUG

The build options `ETH_DEBUG` controls the printout of progress and error messages to `stdout`. This build option can have three set-ups; when set to a value of zero or less, no messages are sent to `stdout`. When set 1, it sends over `stdout` the set-up information used during initialization and causes of error during the operation. When set to a value greater than 1, it prints on `stdout` all operations and causes of errors.

2.1.12 ETH_#_SKEW_???

Multiple build options are available to adjust the time skew on each ones of the RGMII signals. Depending on the PHY, it's likely that only a subset of the RGMII signals can be skewed. Default values depends on the target PHY. Refer to section 4 for a detailed description.

3 Details

This section provides details on how the driver operates in a multiple PHY set-up and how to communicate with PHY interface other than the RGMII interface of the EMAC.

3.1 Multiple PHY

The MDIO interface of the EMAC supports the access to multiple PHYs through a simple addressing. When the build option `ETH_MULTI_PHY` is not defined or if defined and set to a negative value, the EMAC is informed it has to handle a single PHY. In that case, when `PHY_init()` (Section 6.1.2) is called it makes the EMAC to simply scans the addresses for the presence of a PHY where the addresses scanned start at 0. The first PHY discovered becomes the sole PHY used by the EMAC. The discovered PHY is set-up and auto-negotiation is started. If no PHYs are discovered or if the auto-negotiation fails, the initialization is aborted.

Alternatively, the EMAC driver can be informed it has to handle multiple PHY; one example of such a situation is the EMAC is connected to switch with multiple PHYs. In that case the EMAC, when `PHY_init()` (Section 6.1.2) is called, scans all PHY addresses and sets-up all the PHYs discovered. During the set-up and auto-negotiation, if the auto negotiation fails the error is ignored except in the case of the "master PHY". That "master PHY" may or may not be a real PHY but it is needed to set-up the speed of the RGMII interface. If the PHY set-up and auto-negotiation fails for the "master PHY", the initialization is aborted. The failure of the auto-negotiation fails for other PHYs than the master PHY isn't an issue, as proper designed application should periodically monitor the link status of all PHY and deal with a link down condition.

Note: The "master PHY" address is specified through the build option `ETH_ALT_PHY_IF`. If two or more EMAC devices are attached to non-MDIO interfaces, it is the responsibility of the application to make sure the multiple "master PHYs" are all at the same address. This could require device / address remapping to fulfill this restriction.

3.2 Alternate PHY I/F

Some PHY uses another communication mechanism the MDIO, e.g. I2C or SPI. The EMAC driver itself does not handle directly anther type of medium than MDIO but it supports access to PHYs with an alternate custom communication interface though calls to the function `AltPhyIF()` supplied by the application. The function `AltPhyIF()` prototype is:

```
int AltPhyIF(int Dev, int Cmd, int Addr, int P1, int P2);
```

The argument Dev specifies the EMAC device number for which the request applies and the argument Addr specifies the PHY address or PHY number on that EMAC device; both arguments numbering starts at 0. There are multiple case for which `AltPhyIF()` is called and it's specified by the argument `Cmd` which can be any of these defines:

```
ETH_ALT_PHY_IF_INIT
ETH_ALT_PHY_IF_CFG
ETH_ALT_PHY_IF_REG_RD
ETH_ALT_PHY_IF_REG_WRT
ETH_ALT_PHY_IF_GET_RATE
```

The calls to `AltPhyIF()` are protected by default with a mutex, where each EMAC device uses its own mutex. A share mutex among multiple EMAC device may be required for proper protection. An example for this is a switch with two or more RGMII interfaces and two or more of these interfaces are connected to individual EMACS. Typically such a switch still uses a single alternate PHY interface, alike I2C or SPI, so the multiple RGMII interfaces, i.e. multiple EMAC devices, must be protected with the same mutex. Defining the build option ETH_ALT_PHY_MTX and setting it to a positive value will make the EMAC driver use a single mutex for all alternate PHY accesses.

NOTE: unless the build option ETH_ALT_PHY_MTX is defined and set to a negative value, there are no needs to use a mutex to provide exclusive access to the alternate communication interface because the EMAC driver is already using a mutex to protect all PHY accesses.

The following sub-sections described in details what must be performed and meaning of the return value for each commands.

3.2.1 ETH_ALT_PHY_IF_INIT

The command `ATH_ALT_PHY_IF_INIT` is a request to initialize the alternate PHY communication mechanism to use by the EMAC device `Dev`. For example, it could be using the I2C driver and calling `i2c_init()` plus performing any required global set-up needed. As this is a "device" initialization, `AltPhyIF()` is called once per device and the value in the argument `Addr` must be ignored. If there are no alternate PHY interfaces supported by the Device `Dev`, then the return value must be 0. If there are one or more alternate PHY supported attached the device `Dev` the return value must be non-zero. The arguments `P1` and `P2` are always set to 0 and should be ignored.

3.2.2 ETH_ALT_PHY_IF_CFG

`AltPhyIF()` is called with the command `ETH_ALT_PHY_IF_CFG` only for valid Device and PHY addresses (specified by the arguments `Dev` and `Addr`). This command is used for PHY specific configuration alike clock vs data skew. The argument P2 is always 0 and the argument P1 is exactly the same as the argument Rate passed to `PHY_init()` by the application. `AltPhyIF()` will be called with every time `PHY_init()` for that device is called. The caller always ignores the return.

3.2.3 ETH_ALT_PHY_IF_REG_RD

The command `ETH_ALT_PHY_IF_REG_RD` is a request to read an IEEE standard register from the PHY specified by the arguments `Dev` and `Addr`. The register number to read is specified by the argument `P1` and the argument P2 is to be ignored and the return value is the 16 bits value read from the register specified by `P1`. Except for the register numbers 2 and 3, this command is only used for existing PHYs. In the case of register number 2 and 3, these are the PHY identification registers and they are read by the EMAC driver to determine is a PHY at the address `Addr` exists or not. When a PHY exists, the contents of the register 2 and 3 must be returned. When a PHY does not exist, the return value of both register reads MUST be set to `0xFFFF`. If a non-existent "master PHY" is used (See section 3.1), instead of returning `0xFFFF`, the return value must be `0x0001` for both register reads.

3.2.4 ETH_ALT_PHY_IF_REG_WRT

The command `ETH_ALT_PHY_IF_REG_WRT` is a request to write to an IEEE standard register in the PHY specified by the arguments `Dev` and `Addr`. The register number to write is specified by the argument `P1` and the argument P2 is the 16 bits value to write to the register. The return value must be 0 upon successful write and non-zero for a failed write operation. This command is only used for existing PHYs.

3.2.5 ETH_ALT_PHY_IF_GET_RATE

`ETH_ALT_PHY_IF_GET_RATE` is used to query the "master PHY" (See section 3.1) about the rate the RGMII interface is set to. The returned value must be the same numbering as used in the return value of the function `PHY_init()` (See section 6.1.2)

4 Skews

Multiple build options constructed in the form `ETH_#_SKEW_???` are available to adjust the skew on the RGMII signals between the EMAC and the PHY. In the build option, the character # must be replaced by the EMAC device number, starting from 0 and going upward. The characters `???` specify the signal to skew and these characters can take the following: `RXCTL`, `RXCLK`, `RX0`, `RX1`, `RX2`, `RX3`, `TXCTL`, `TXCLK`, `TX0`, `TX1`, `TX2`, `TX3`. `RXCLK` and `TXCLK` adjust the skew on the clock line, and `RXCTL` and `TXCTL` are for the skew on the control line. The `RX0`, `RX1`, `RX2`, `XR3`, `TX0`, `TX1`, `TX2`, and `TX3` are the skew for the data line where the numbering from 0 to 3 specify the pair numbering at the Ethernet interface. Unless specified otherwise the skew is an absolute delay / advance irrelevant from the other signals on the RGMII.

All skews are specified in picosecond (ps) units. Although no PHY have the capability to fine tune the skew at the picosecond level, the choice for such a granularity was necessary because PHYs typically use different skew stepping. The default value is used if the build option is not defined or if it is defined and set to -1 (the value MUST be -1 otherwise the value is literally used). The default values are values that work on the development boards used by Core Time.

A positive skew value is the signal(s) specified by the build option being delayed by the specified number in picoseconds. Alternatively, a negative skew is the signal(s) specified by the build option being advanced in time by the negative of the specified number in picoseconds.

The PHY labeled *Marvel 88E1x* covers the 4 following parts: 88E10, 88E12, 88E14 and 88E18.

The value specified by the build option should match the specified minimum, maximum and step values indicated in the table. If this is not respected then the following will occur

- If the value specified is not an exact multiple of the step value, the resulting value used is alike the mathematical integer operation `((Value/Step)*Step)` applied at the specified value.
- Build options values are not checked against the boundaries, nor clipped. If the value specified is less than the minimum supported or larger than the maximum supported then the skew used will be the result of some modulo operation.

Only one of build options for the skews is supported per EMAC device, i.e. when an EMAC device is connected to multiple PHYs then the skew to apply on the individual PHYs must be implemented in the EMAC driver itself (or in `AltPhyIF()`).

4.1 ETH_#_SKEW_RXCTL

The build option `ETH_#_SKEW_RXCTL` sets the skew on the Receiver Control signal of the RGMII interface. The character # must be set to the EMAC device number starting with a value of 0 going upward.

Table 4-1 ETH_#_SKEW_RXCTL values

PHY	Min	Max	Step	Default
Micrel KSZ9021	-840	960	120	-840
Micrel KSZ9031	-480	420	60	-480
Micrel KSZ9071	-480	420	60	-480
Lantiq PEF7071	Ignored – Not supported			
Marvel 88E1x	Ignored – Not Supported			
TI TIDP83847	Ignored – Not Supported			

4.2 ETH_#_SKEW_RXCLK

The build option `ETH_#_SKEW_RXCLK` sets the skew on the Receiver Clock signal of the RGMII interface. The character # must be set to the EMAC device number starting with a value of 0 going upward.

Table 4-2 ETH_#_SKEW_RXCLK values

PHY	Min	Max	Step	Default
Micrel KSZ9021	-840	960	120	360
Micrel KSZ9031	-900	960	60	0
Micrel KSZ9071	-900	960	60	0
Lantiq PEF7071	0	1500	500	1500
Marvel 88E1x	Ignored – Not Supported			
TI TIDP83847	0	4000	250	2250

PEF7071 if `ETH_#_SKEW_RXCLK` is not defined (or defined and set to -1), the default value specified in the table is used. To not overload and use the strap pin skew setting instead, define and set `ETH_#_SKEW_RXCLK` to a value of -2.

TIDP83847 if the strap option are enabled the skew used is configured from the strap and not from S/W.

4.3 ETH_#_SKEW_RXn

The build options `ETH_#_SKEW_RXn` covers, with `n` having the possible values of 0, 1, 2, and 3, the skew to apply on each one of the 4 pairs of data signals (at the Ethernet I/F). Some PHYs support individual pair setting when others apply the same skew on all 4 pairs. In the following tables, when the column labeled `n` only lists 0 it means the skew specified is applied on all 4 pairs and the build option `ETH_#_SKEW_RX0` is the one used to set the skew. In that case all other build options for the Receiver Data skews, i.e. `ETH_#_SKEW_RX1`, `ETH_#_SKEW_RX2` and, `ETH_#_SKEW_RX3`, are ignored.

Table 4-3 ETH_#_SKEW_RXn values

PHY	Min	Max	Step	Default	n
Micrel KSZ9021	-840	960	120	-840	0, 1, 2, 3
Micrel KSZ9031	-480	420	60	0	0, 1, 2, 3
Micrel KSZ9071	-480	420	60	0	0, 1, 2, 3
Lantiq PEF7071	Ignored – Not Supported				
Marvel 88E1x	0	120000	8000	0	0, 1, 2, 3
TI TIDP83847	Ignored – Not Supported				

4.4 ETH_#_SKEW_TXCTL

The build option `ETH_#_SKEW_TXCTL` sets the skew on the Transmitter Control signal of the RGMII interface. The character # must be set to the EMAC device number starting with a value of 0 going upward.

Table 4-4 ETH_#_SKEW_TXCTL values

PHY	Min	Max	Step	Default
Micrel KSZ9021	-840	960	120	-840
Micrel KSZ9031	-480	420	60	-480
Micrel KSZ9071	-480	420	60	-480
Lantiq PEF7071	Ignored – Not Supported			
Marvel 88E1x	Ignored – Not Supported			
TI TIDP83847	Ignored – Not Supported			

4.5 ETH_#_SKEW_TXCLK

The build option `ETH_#_SKEW_TXCLK` sets the skew on the Transmitter Clock signal of the RGMII interface. The character # must be set to the EMAC device number starting with a value of 0 going upward.

Table 4-5 ETH_#_SKEW_TXCLK values

PHY	Min	Max	Step	Default
Micrel KSZ9021	-840	960	120	720
Micrel KSZ9031	-900	960	60	0
Micrel KSZ9071	-900	960	60	0
Lantiq PEF7071	0	1500	500	1500
Marvel 88E1x	Set to 0 or non-0 (See below)			
TI TIDP83847	0	4000	250	2750

PEF7071	if `ETH_#_SKEW_RXCLK` is not defined (or defined and set to -1), the default value specified in the table is used. To not overload and use the strap pin skew setting instead, define and set `ETH_#_SKEW_RXCLK` to a value of -2.
88E1x	The 88E1x only supports an enable and disable of an internal delay of around ¼ clock period. Setting `ETH_#_SKEW_TXCLK` to a zero value disable the TX clock internal delay and when set to a non-zero value it enables the delay. If `ETH_#_SKEW_TXCLK` is not defined or if defined and set to a value of -1 then the delay is enable.
TIDP83847	if the strap option are enabled then the skew to use is configured from the straps and not from the S/W.

4.6 ETH_#_SKEW_TXn

The build options `ETH_#_SKEW_TXn` covers, with `n` having the possible values of 0, 1, 2, and 3, the skew to apply on each one of the 4 pairs of data signals (at the Ethernet I/F). Some PHYs support individual pair setting when others apply the same skew on all 4 pairs. In the following table, when the column labeled `n` only lists 0 it means the skew specified is applied on all 4 pairs and the build option `ETH_#_SKEW_TX0` is the one used to set the skew. In that case all other build options for the Transmitter Data skews, i.e. `ETH_#_SKEW_TX1`, `ETH_#_SKEW_TX2`, and `ETH_#_SKEW_TX3`, are ignored.

Table 4-6 ETH_#_SKEW_TXn values

PHY	Min	Max	Step	Default	n
Micrel KSZ9021	-840	960	120	-840	0, 1, 2, 3
Micrel KSZ9031	-480	420	60	0	0, 1, 2, 3
Micrel KSZ9071	-480	420	60	0	0, 1, 2, 3
Lantiq PEF7071	Ignored – Not Supported				
Marvel 88E1x	Ignored – Not Supported				
TI TIDP83847	Ignored – Not Supported				

5 EMAC API

In this section, the EMAC API of all common EMAC driver functions is provided. For the PHY API, refer to the next section. Section 7 gives examples on how to use the EMAC.

5.1.1 ETH_MACAddressConfig

Synopsis

```
#include “???_ethernet.h”

void ETH_MACAddressConfig(int Dev, unint32_t MacAddr, uint8_t *Addr);
```

Description

`ETH_MACAddressConfig()` is the component used to set the MAC address in the EMAC module. The module’s device number is indicated by the argument `Dev` and the MAC address register to program is indicated by the argument `MacAddr`. The 6 byte MAC address is supplied through the `Addr` array of bytes, where the first byte (index 0) is the LSByte of the MAC address.

Arguments

`Dev`	Module’s device number (Number starting at 0)
`MacAddr`	MAC address register to set-up (These tokens must be used: `ETH_MAC_Address0` up to `ETH_MAC_AddressN`, where *N* is target specific)
`Addr`	48 bit MAC address to set-up. The LSByte of the MAC address is in `Addr[0]` and the MSByte of the MAC address is in `Addr[5]`.

Returns

`void`

Component type

Function

Options

Notes

See Also

`ETH_MACAddressGet`

5.1.2 ETH_MACAddressGet

Synopsis

```
#include “???_ethernet.h”

void ETH_MACAddressGet(int Dev, unint32_t MacAddr, uint8_t *Addr);
```

Description

`ETH_MACAddressGet()` is the component used to retrieve the MAC address that was set-up in the EMAC module. The module’s device number is indicated by the argument `Dev` and the MAC address register to extract the MAC address from is indicated by the argument `MacAddr`. The 6 byte MAC address is reported through the `Addr` array of bytes, where the first byte (index 0) is the LSByte of the MAC address.

Arguments

`Dev`	Module’s device number (Number starting at 0)
`MacAdrr`	MAC address register to extract the MAC address (These tokens must be used: `ETH_MAC_Address0` up to `ETH_MAC_Address`*`N`*, where *`N`* is target specific)
`Addr`	The 48 bit MAC address extracted is deposited in this buffer. The LSByte of the MAC address is in `Addr[0]` and the MSByte of the MAC address is in `Addr[5]`. Must be at least dimensioned to 6 bytes

Returns

`void`

Component type

Function

Options

Notes

See Also

`ETH_MACAddressConfig`

5.1.3 ETH_DMARxDescChainInit

Synopsis

```
#include “???_ethernet.h”

void ETH_DMARxDescChainInit(int Dev);
```

Description

`ETH_DMARxDescChainInit()` creates the circular linked list of descriptors used by the DMA in the receiving direction, rendering the reception ready to be started. The EMAC module number is specified through the argument `Dev`. `ETH_DMARxDescChainInit()` does not enable the received interrupts, nor does it starts the transfers. For these, refer to the components `ETH_DMARxDescReceiveITConfig()` and `ETH_Start()`.

If the buffer type build option `ETH_BUFFER_TYPE` is set to `ETH_BUFFER_PBUF`, then the application must attach the DMA payload buffers to the descriptors after `ETH_DMARxDescChainInit()` has been applied. Refer to the initialization example (Sect 7.1.2).

Arguments

`Dev`	Module’s device number to set-up (Number starting at 0)

Returns

`void`

Component type

Function

Options

Notes

See Also

`ETH_DMATxDescChainInit`
`ETH_DMARxDescReceiveITConfig`
`ETH_Start`

5.1.4 ETH_DMARxDescReceiveITConfig

Synopsis

```
#include “???_ethernet.h”

void ETH_DMARxDescReceiveITConfig(int Dev, int NewState);
```

Description

`ETH_DMARxDescReceiveITConfig()` controls if the reception of packets generates an interrupt. The EMAC module number is specified through the argument `Dev`. To enable the reception interrupts, the argument `NewState` must be set to a non-zero value. To disable the reception interrupts, the argument `NewState` must be set to a value of 0.

Arguments

`Dev`	Module’s device number to set-up (Number starting at 0)
`NewState`	`== 0` : disable the reception interrupts
	`!= 0` : enable the reception interrupts

Returns

`void`

Component type

Function

Options

Notes

See Also

`ETH_DMARxDescChainInit`
`ETH_Start`

5.1.5 ETH_DMATxDescChainInit

Synopsis

```
#include "???_ethernet.h"

void ETH_DMATxDescChainInit(int Dev);
```

Description

`ETH_DMATxDescChainInit()` creates the circular linked list of descriptors used by the DMA in the transmitting direction, rendering the transmission ready to be started. The EMAC module number is specified through the argument `Dev`. `ETH_DMATxDescChainInit()` does not start, nor does enable the checksum insertion by the EMAC module (when applicable). For these, please refer to the components `ETH_Start()` and `ETH_DMATxDescChecksumInsertionConfig()`.

Arguments

`Dev`	Module's device number to set-up (Number starting at 0)

Returns

`void`

Component type

Function

Options

Notes

.

See Also

```
ETH_DMARxDescChainInit
ETH_DMARxDescReceiveITConfig
ETH_DMATxDescChecksumInsertionConfig
ETH_Start
```

5.1.6 ETH_DMATxDescTransmitITConfig

Synopsis

```
#include "???_ethernet.h"

void ETH_DMATxDescTransmitITConfig(int Dev, int NewState);
```

Description

`ETH_DMATxDescTransmitITConfig()` controls if the transmission of packets generates an interrupt. The EMAC module number is specified through the argument `Dev`. To enable the transmission interrupts, the argument `NewState` must be set to a non-zero value. To disable the transmission interrupts, the argument `NewState` must be set to a value of 0.

Arguments

`Dev`	Module's device number to set-up (Number starting at 0)
`NewState`	`== 0` : disable the transmission interrupts
	`!= 0` : enable the transmission interrupts

Returns

`void`

Component type

Function

Options

Notes

See Also

`ETH_DMATxDescChainInit`
`ETH_Start`

5.1.7 ETH_DMATxDescChecksumInsertionConfig

Synopsis

```
#include "???_ethernet.h"

void ETH_DMATxDescChecksumInsertionConfig(int Dev, int ChkSum);
```

Description

`ETH_DMATxDescChecksumInsertionConfig()` controls the insertion by the EMAC module of the TCP / UDP / ICMP checksum, including the pseudo header. The module device to set-up is specified with the argument `Dev`. The argument `ChkSum`, a Boolean, requests the module to insert the checksum (`ChkSum` != 0) or to not insert the checksum (`ChkSum` == 0). `ETH_DMATxDescChecksumInsertionConfig()` should only be used before enabling the transmission with `ETH_Start()` (Section 5.1.17).

Arguments

`Dev`	Module's device number to set-up (Number starting at 0)
`ChkSum`	`== 0` : request the module to not insert the checksum.
	`!= 0` : request the module to insert the checksum

Returns

`void`

Component type

Function

Options

Notes

5.1.8 ETH_Get_Received_Frame_Length

Synopsis

```
#include "???_ethernet.h"

int ETH_Get_Received_Frame_Length(int Dev);
```

Description

`ETH_Get_Received_Frame_Length()` is used to know if a packet is available in the reception direction and, when one is available, what is its size. The module's device number is specified through the argument `Dev` and the information is delivered in the retuned value. When the return value is negative, then no new packets are available. Non-negative values indicate the number of bytes in the new packet (including 0 for empty payload packet).

Arguments

`Dev`	EMAC device number to retrieve the packet info from

Returns

`int`	`< 0:` no new packet available `>=0:` size in bytes of the new packet

Component type

Function

Options

Notes

If `ETH_Get_Received_Frame_Length()` is used after reading one or more segments through `ETH_Get_Received_Multi()`, the returned value will not report the size of the whole packet, but instead will report the size of the unread section of the packet.

See Also

`ETH_Get_Received_Frame_Length`
`ETH_Get_Received_Multi`

5.1.9 ETH_Get_Received_Multi

Synopsis

```
#include "???_ethernet.h"

FrameTypeDef ETH_Get_Received_Multi(int Dev)
```

Description

`ETH_Get_Received_Multi()` is used to retrieve the information on the oldest non-extracted segment received by the DMA. The module's device number is specified through the argument `Dev` and the retrieved information is inserted in the retuned data structure of type `FrameTypeDef`. The number of bytes held in the segment is indicated in `FrameTypeDef.length`. If the value is negative then there are no more segments available. The base address of the payload is indicated by `FrameTypeDef.buffer`, and if the base address of the payload buffer is `NULL` then there are no more segments available.

Arguments

`Dev`	EMAC device number to retrieve the packet segment from

Returns

`FrameTypeDef`

Component type

Function

Options

Notes

Once all the segments of a packet have been extracted, the component `ETH_Release_Received()` must be called to return all the DMA descriptors that were holding the segment(s) of the packet back to the DMA.

See Also

```
ETH_Get_Received_Frame_Length
ETH_Release_Received
```

5.1.10 ETH_Get_Transmit_Buffer

Synopsis

```
#include "???_ethernet.h"

void *ETH_Get_Transmit_Buffer(int Dev);
```

Description

`ETH_Get_Transmit_Buffer()` is used by the application to know the base address of the next transmit buffer available to send a packet, or segment of a packet. The module's device number is specified through the argument `Dev`. The component returns the base address of the payload buffer, and NULL if there are no more DMA transmit descriptors available. The non-availability of a transmit descriptor is due to the DMA owning all transmit descriptors; in other words, all the transmit descriptors have been given to the DMA and transmission is still pending.

Arguments

`Dev`	EMAC device number to obtain the payload buffer from

Returns

`Void *`	`== NULL`	: no transmit payload buffers available
	`!= NULL`	: base address of the payload buffer

Component type

Function

Options

Notes

The non-availability of a transmit descriptor is due to the DMA owning all transmit descriptors; in other words, all the transmit descriptors have been given to the DMA and transmission is still pending. This could also be provoked if the application requires a number of segments for a packet where the number of segments exceeds the number of transmission DMA descriptors (`ETH_N_TXBUF`).

`ETH_Get_Transmit_Buffer()` must always be used and its return value checked before calling the component `ETH_Prepare_Transmit_Descriptors()` or the component `ETH_Prepare_Transmit_Descriptors()`.

See Also

`ETH_Prepare_Transmit_Descriptors`
`ETH_Prepare_Multi_Transmit`

5.1.11 ETH_MacConfigDMA

Synopsis

```
#include "???_ethernet.h"

int ETH_MacConfigDMA(int Dev, int Rate);
```

Description

`ETH_MacConfigDMA()` is the component used to start-up the Ethernet link (this does not start the RX & TX of the packets). The device to start-up is specified with the argument `Dev`. The link rate can be set to auto-negotiate or set to a desired speed / duplexing with the value assigned to the argument `Rate`. The returned value reports the established speed and duplex when successful or an error if the link did not established connection.

Arguments

`Dev`	EMAC device number to start-up the Ethernet link
`Rate`	Specifies if using auto-negotiation or using a fixed rate and duplexing. The accepted values for Rate are:

`10`	10 Mbps / full duplex
`11`	10 Mbps / half duplex
`100`	100 Mbps / full duplex
`101`	100 Mbps / half duplex
`1000`	1000 Mbps / full duplex
`1001`	1000 Mbps / half duplex
`Negative`	auto-negotiation

Returns

`int`	Resulting connection speed and duplexing or error

`10`	10 Mbps / full duplex
`11`	10 Mbps / half duplex
`100`	100 Mbps / full duplex
`101`	100 Mbps / half duplex
`1000`	1000 Mbps / full duplex
`1001`	1000 Mbps / half duplex
`Negative`	the connection was not established

Component type

Function

Options

Notes

5.1.12 ETH_Prepare_Transmit_Descriptors

Synopsis

```
#include "???_ethernet.h"

int ETH_Prepare_Transmit_Descriptors(int Dev, int Len);
```

Description

`ETH_Prepare_Transmit_Descriptors()` is the component to use to give to the DMA a transmit descriptor that holds a non-segmented packet. The payload that will be transmitted is the most recent one returned by the component `ETH_Get_Transmit_Buffer()`. The module's device number is specified through the argument `Dev`, and the size of the packet (in bytes) is specified by the argument `Len`.

Arguments

`Dev`	EMAC device number to retrieve the packet segment from
`Len`	Number of bytes in the packet

Returns

`int`	`== 0`	Success
	`== 1`	The DMA owns all the descriptors This can only happen if `ETH_Get_Transmit_Buffer` was not called before or if it was called, the returned value not checked against `NULL`.
	`== 2`	The packet length specified by `Len` exceeds the size of the DMA payload

Component type

Function

Options

Notes

The component `ETH_Get_Transmit_Buffer()` must always be used and its return value checked before calling `ETH_Prepare_Transmit_Descriptors()`.

If a packet is too large to fit in the allocated DMA payload (build option `ETH_RX_BUFSIZE`), then the component `ETH_Prepare_Multi_Transmit` must be used instead of `ETH_Prepare_Transmit_Descriptors()`.

See Also

`ETH_Get_Transmit_Buffer`
`ETH_Prepare_Multi_Transmit`

5.1.13 ETH_Prepare_Multi_Transmit

Synopsis

```
#include "???_ethernet.h"

int ETH_Prepare_Multi_Transmit(int Dev, int Len, int IsFirst,
                               int IsLast);
```

Description

`ETH_Prepare_Multi_Transmit()` is the component to use to give to the DMA a transmit descriptor that holds a segment of a packet. The payload of the segment is the most recent one returned by the component `ETH_Get_Transmit_Buffer()`. The module's device number is specified through the argument `Dev` and the size of the packet (in bytes) is specified by the argument `Len`. The argument `IsFirst`, a Boolean, indicates the segment as being the first segment of a packet when set to a non-zero value. When set to a value of 0, then the segment is not the first one. The argument `IsLast`, a Boolean, indicates the segment as being the last segment of a packet when set to a non-zero value. When set to a value of 0, then the segment is not the last one. When a single segment is the whole packet, both argument `IsFirst` and `IsLast` must be set to a non-zero value.

Arguments

`Dev`	EMAC device number to retrieve the packet segment from
`Len`	Number of bytes in the segment
`IsFirst`	When non-zero, indicates this is the first segment of the packet
`IsLast`	When non-zero, indicates this is the last segment of the packet

Returns

`int`	`== 0`	Success
	`== 1`	The DMA owns all the descriptors This can only happen if `ETH_Get_Transmit_Buffer` was not called before or if it was called, the returned value not checked against `NULL`.
	`== 2`	The packet length specified by `Len` exceeds the size of the DMA payload
	`== 3`	The packet was never given its first segment

Component type

Function

Options

Notes

The packet is not transmitted as long as the argument `IsLast` is zero. The internal state machine is self-correcting upon incorrect usage of `IsFirst` and/or `IsLast`.

The component `ETH_Get_Transmit_Buffer()` must always be used and its return value checked before calling `ETH_Prepare_Multi_Transmit()`.

If a packet is too large to fit in the allocated DMA payload (build option `ETH_TX_BUFSIZE`), then the component `ETH_Prepare_Transmit_Descriptors()` must be used instead of `ETH_Prepare_Multi_Transmit()`.

See Also

`ETH_Get_Transmit_Buffer`
`ETH_Prepare_Transmit_Descriptors`

5.1.14 ETH_ReleaseMulti

Synopsis

```
#include "???_ethernet.h"

void ETH_ReleaseMulti(int Dev);
```

Description

Once a packet all the segments of a packet have been extracted through the component `ETH_Get_Received_Multi()`, the component `ETH_ReleaseMulti()` must be called to return to the DMA the descriptor the extracted packet was using. This component also sets-up the EMAC driver internal mechanism to handle the next packet received or to be retrieved.

Arguments

`Dev`	EMAC device number to wrap-up the extraction of a packet

Returns

`void`

Component type

Function

Options

Notes

If `ETH_ReleaseMulti()` is used before having read all the segments of a packet (through `ETH_Get_Received_Multi()`), then the internal EMAC driver mechanism will consider all the data that was extracted was one packet and the remainder of the un-read packet becomes now considered as a newly received packet. Therefore, `ETH_ReleaseMulti()` should only be used once all the segments of a packet have been extracted.

See Also

```
ETH_Get_Received_Frame_Length
ETH_Get_Received_Multi
```

5.1.15 ETH_ResetEMAC

Synopsis

```
#include "???_ethernet.h"

void ETH_ResetEMACs(int Dev);
```

Description

`ETH_ResetEMAC()` is used to reset and pre-initialize one EMACs device on the platform. It should be called only once and this should be done before using any other EMAC driver components of that EMAC device.

Arguments

`Dev`	EMAC device number to reset & initialize

Returns

`void`

Component type

Function

Options

Notes

5.1.16 ETH_ResetEMACs

Synopsis

```
#include “???_ethernet.h”

void ETH_ResetEMACs(void);
```

Description

`ETH_ResetEMACs()` is used to reset and pre-initialize all EMACs devices on the platform. It should be called only once and this should be done before using any other EMAC driver components.

Arguments

`void`

Returns

`void`

Component type

Function

Options

Notes

5.1.17 ETH_Start

Synopsis

```
#include "???_ethernet.h"

void ETH_Start(int Dev);
```

Description

`ETH_Start()` start the transmission and reception of the EMAC module. The module device to start is specified with the argument `Dev`.

Arguments

`Dev`	Module's device number to start (Number starting at 0)

Returns

`void`

Component type

Function

Options

Notes

Before using `ETH_Start()`, each of these steps must be performed:

- RX DMA chain initialization
- If the buffer type is set to `ETH_BUFFER_PBUF`, attach the payload buffers to the reception descriptors
- Enable / disable the reception interrupt
- TX DMA chain initialization
- Enable / disable the hardware insertion of the checksum.

See Also

```
ETH_DMARxDescChainInit
ETH_DMARxDescReceiveITConfig
ETH_DMATxDescChainInit
ETH_DMATxDescChecksumInsertionConfig
ETH_Stop
```

5.1.18 ETH_Stop

Synopsis

```
#include "???_ethernet.h"

void ETH_Stop(int Dev);
```

Description

`ETH_Stop()` stops the transmission and reception of the EMAC module. The module device to stop is specified with the argument `Dev`.

Arguments

`Dev`	Module's device number to stop (Number starting at 0)

Returns

`void`

Component type

Function

Options

Notes

After using `ETH_Stop()`, it may be possible or may not be as this depends on the target platform, to use `ETH_start()` without having go through the whole initialization sequence.

See Also

`ETH_Start`

6 PHY API

In this section, the PHY API of all common EMAC driver functions is provided. For the EMAC API, refer to the previous section. Section 7 gives examples on how to use the EMAC.

6.1.1 AltPhyIF

Synopsis

```
#include "???_ethernet.h"

int AltPhyIF(int Dev, int Cmd, int Addr, int P1, int P2);
```

Description

`AltPhyIF()` is a function supplied by the application for custom communications with one or multiple PHYs. This function is called only f the build option `ETH_ALT_PHY_IF` is defined and set to a non-zero value. The EMAC device calling `AltPhyIF()` is indicated by the argument `Dev` and the PHY address by the argument `Addr`. There are 5 commands that can be specified with the argument `Cmd` and depending on the command, the arguments `P1` and `P2` could be arguments specific to the command.

Arguments

`Dev`	Module's device number to stop (Number starting at 0)
`Cmd`	Type of operation to perform.
`Addr`	PHY address or number (Starts at 0)
`P1`	1st argument to the command
`P2`	2nd argument to the command

Returns

`int`	The return value is command specific

Component type

Function

Options

There are 5 commands that can be specified

```
ETH_ALT_PHY_IF_INIT
ETH_ALT_PHY_IF_CFG
ETH_ALT_PHY_IF_REG_RD
ETH_ALT_PHY_IF_REG_WRT
ETH_ALT_PHY_IF_GET_RATE
```

A detailed description is provided in section 0

Notes

See Also

Section 0.

6.1.2 PHY_init

Synopsis

```
#include "???_ethernet.h"

int PHY_init(int Dev, int Rate);
```

Description

`ETH_MACAddressConfig()` is the component used to set the MAC address in the EMAC module. The module's device number is indicated by the argument `Dev` and the MAC address register to program is indicated by the argument `MacAddr`. The 6 byte MAC address is supplied through the `Addr` array of bytes, where the first byte (index 0) is the LSByte of the MAC address.

Arguments

`Dev`	Module's device number (Number starting at 0)
`Rate`	Desired Ethernet rate. The recognized values for `Rate` are: 10 : 10 Mbps full duplex 11 : 10 Mbps half duplex 100 : 100 Mbps full duplex 101 : 100 Mbps half duplex 1000 : 1000 Mbps full duplex 1001 : 1000 Mbps half duplex -ve : auto-negotiation

Returns

`int`	
	10 : 10 Mbps full duplex 11 : 10 Mbps half duplex 100 : 100 Mbps full duplex 101 : 100 Mbps half duplex 1000 : 1000 Mbps full duplex 1001 : 1000 Mbps half duplex -ve : error

Component type

Function

Options

Notes

See Also

7 Examples

7.1 Initialization

The first step required when using the EMAC driver is to reset and do the basic initialization of all EMAC modules on the chip. This needs to be done only once and it must performed before using any EMAC driver components. Then the IP stack and the driver/buffer can be configured and interrupts enabled:

Table 7-1 Global Initialization

```
ETH_ResetEMACs();

- Install EMAC interrupt handler
- Init the IP Stack & Driver/Buffers
- Enable EMAC interrupts
```

Once the EMAC modules have been reset and are ready to be programmed and used, the sequence of initialization is different if regular buffers are used or if supplied buffers are used. The supplied buffer examples use the LwIP `pbufs`. For sake of generality, the EMAC device used in the followings example is the token `EMAC_DEVICE` as the value is dependent on the EMAC module to handle.

When using Abassi, a custom interrupt handler, which is IP stack dependent, should be used instead of relying on polling. Such an interrupt handler is provided for the lwIP IP stack in the file `ethernetif.c` located in `lwip-?-?-?/ports/Target/RTOS` for the lwIP IP stack.

7.1.1 Initialization for regular buffers

The initialization when the EMAC driver is using regular buffers (`ETH_BUFFER_TYPE` is set to either `ETH_BUFFER_CACHED` or `ETH_BUFFER_UNCACHED`) is straightforward. The first step is to bring the link up using the component `ETH_MacConfigDMA()`, specifying which EMAC module and desired link rate. Here, the link rate is left to be determined through auto-negotiation as the `Rate` argument (2nd argument) is set to -1. Once the link is up, the MAC address (not the IP address) must be provided to the EMAC module through the component `ETH_MACAddressConfig()`. Then the sequence becomes:

- Initialize the receiver DMA descriptor & buffer chain through `ETH_DMARxDescChainInit()`
- Initialize the transmission DMA descriptor & buffer chain through `ETH_DMATxDescChainInit()`
- Enable the Receive interrupts if interrupts are used in the reception, through `ETH_DMARxDescReceiveITConfig()`
- Enable the Transmit interrupts if interrupts are used in the transmission, through `ETH_DMATxDescTransmitITConfig()`
- If the EMAC module is capable of inserting the IP header checksum and if the IP stack is configure to offload the checksum calculation, then set-up the module to compute and insert the checksum itself through `ETH_DMATxDescChecksumInsertionConfig()`
- Finally, the last step in the initialization is to start the DMA through `ETH_Start()`

Table 7-2 Regular Buffer Initialization

```
  ii = ETH_MacConfigDMA(EMAC_DEVICE, -1);
  if (ii >= 0) {
      printf("The link #%d is up at %d Mbps %s duplex\n",
                         EMAC_DEVICE, ii&~1, (ii&1)?"half":"full");
  }
  else {
      printf("Ethernet link #%d failed to connect\n", EMAC_DEVICE);
  }

  ETH_MACAddressConfig(EMAC_DEVICE, ETH_MAC_Address0, netif->hwaddr);

  ETH_DMARxDescChainInit(EMAC_DEVICE);
  ETH_DMARxDescReceiveITConfig(EMAC_DEVICE, 1);

  ETH_DMATxDescChainInit(EMAC_DEVICE);
  ETH_DMARxDescTransmitITConfig(EMAC_DEVICE, 1);
#ifdef CHECKSUM_BY_HARDWARE
  ETH_DMATxDescChecksumInsertionConfig(EMAC_DEVICE, 1);
#else
  ETH_DMATxDescChecksumInsertionConfig(EMAC_DEVICE, 0);
#endif

  ETH_Start(EMAC_DEVICE);
```

7.1.2 Initialization for pbuf

The initialization when the EMAC driver is using supplied buffers (`ETH_BUFFER_TYPE` is set to `ETH_BUFFER_PBUF`) must attach the supplied buffers to the DMA chain used in the reception. The internal global data structure `G_DMArxDescTbl[][]` must be set-up to supply the buffer information to the DMA. Two entries of interest in `G_DMArxDescTbl[][]` when attaching external buffer:

➢ `Buff`	pointer to the payload buffer
➢ `PbufPtr`	general purpose pointer (of type `void *`) that can be used to hold any higher level information related to the payload buffer attached to the DMA descriptor.

As for the regular buffer, `ETH_DMARxDescChainInit()` is called first, then, before going further (calling `ETH_DMATxDescChainInit()`), the buffers are attached to the RX DMA descriptors. Step by step this involves, one iteration for each of the `ETH_N_RXBUF` in the reception DMA chain:

- ➢ Get a new buffer or buffer description element. In the case of LwIP, this involves using the `pbuf_alloc()` component.
- ➢ Optionally, if buffer description elements are used, memorize the buffer description element for later use by the stack in the field `PbufPtr` in `G_DMArxDescTbl[MAP_EMAC(EMAC_DEVICE)][]`.
- ➢ If IP padding is required due to integer alignment on the target platform, then perform the dropping of the padding to obtain a new payload buffer address
- ➢ Memorize the payload buffer address for use by the DMA and for later use by the IP stack in the field `Buff` in `G_DMArxDescTbl[MAP_EMAC(EMAC_DEVICE)][]`
- ➢ If the payload buffers are located in cached memory, then the whole payload buffer memory must be invalidated. The invalidation must be performed everytime after the buffer is used to extract newly received data.

Finally, if the DMA descriptors are located in cached memory, the memory of all `ETH_N_RXBUF` descriptors must be flushed to move the DMA descriptors from the data cache into the external physical memory such that the DMA can access them.

Table 7-3 Regular Buffer Initialization

```
…

ETH_DMARxDescChainInit(EMAC_DEVICE);

for (ii=0 ; ii<ETH_N_RXBUF ; ii++) {
    NewPbuf = pbuf_alloc(PBUF_RAW, ETH_RX_BUFSIZE+ETH_PAD_SIZE, PBUF_POOL);
    G_DMArxDescTbl[ETH_MAP_DEV(EMAC_DEVICE)][ii].PbufPtr = NewPbuf
    SKIP_PAD(NewPbuf);
    G_DMArxDescTbl[ETH_MAP_DEV(EMAC_DEVICE)][ii].Buf = (uint32_t)NewPbuf->payload;
    DCacheInvalRange(NewPbuf->payload, ETH_RX_BUFSIZE);
}
DCacheFlushRange(&G_DMArxDescTbl[0], ETH_N_RXBUF*sizeof(G_DMArxDescTbl[0]));

ETH_DMARxDescReceiveITConfig(EMAC_DEVICE, 1);

…
```

7.2 Packet transmission

7.2.1 Sending Segmented Packets

This example for sending a packet through the EMAC driver covers all possible ways an IP stack could deliver a packet to send out. Mainly, the IP stack could have broken the packet to send it into multiple smaller segments. Plus, the whole packet, or the packet segments, could be larger than size of the DMA payload buffer in the transmit direction.

Basically, to send something through the EMAC driver requires 3 basic steps:

- Obtain the address of the payload buffer the DMA will use for the next transmission. The buffer address is obtained through the component `ETH_Get_Transmit_Buffer()`
- Copy the payload to send into the DMA payload buffer, up to the maximum size of the payload buffer, which is `ETH_TX_BUFSIZE`.
- Set-up the DMA descriptor to prepare it to be used by the DMA through the component `ETH_Prepare_Multi_Transmit()`

If there are no available DMA descriptors for transmission, then the application can pause and wait until one becomes available, or it can abort. The EMAC driver has been designed to be self-recovering. If a packet was partially set-up to be transmitted and the construction was aborted, the EMAC driver will self-recover from this error. Recovery will also be performed if the first segment of a packet is missing.

Table 7-4 Segmented packet transmission

```
SKIP_PAD(p);

IsFirst = 1;
for(Pbuf=p ; Pbuf!=NULL ; Pbuf=Pbuf->next) {
    LeftOver = Pbuf->len;
    BufPtr   = Pbuf->payload;
    while(LeftOver > 0) {
        for (ii=0 ; ii<10 ; ii++) {
            DMAbuf = ETH_Get_Transmit_Buffer(EMAC_DEVICE);
            if (DMAbuf != NULL) {
                break;
            }
            TSKsleep(OS_MS_TO_TICK(20));
        }
        if (DMAbuf == NULL) {
            CLAIM_PAD(p);
            Return(ERR_MEM);
        }
        Nbytes = (LeftOver < ETH_TX_BUFSIZE )
               ? LeftOver
               : ETH_TX_BUFSIZE;
        LeftOver -= Nbytes;
        IsLast =  (Pbuf->len == Pbuf->tot_len)
               && (LeftOver == 0);
        memmove((void *)&DMAbuf[0], BufPtr, (size_t)Nbytes);
        BufPtr += Nbytes;

        ETH_Prepare_Multi_Transmit(EMAC_DEVICE, Nbytes, IsFirst, IsLast);
        IsFirst = 0;
    }
    if (IsLast != 0) {
        IsFirst = 1;
    }
}

CLAIM_PAD(p);
```

In the previous example, based on LwIP, the outer loop deals with possibly multiple packets where each packet could have been broken into smaller segments by the IP stack:

```
for(Pbuf=p ; Pbuf!=NULL ; Pbuf=Pbuf->next) {
```

The last segment of a packet is indicated when `Pbuf->len == Pbuf->tot_len` and the last packet is indicated when `Pbuf->next == NULL`.

The inner loop deals with the possible breaking down of the individual segments into sizes that don't exceed the DMA payload buffer. It uses the variable `LeftOver` to keep track of how many bytes have still not been sent out and the pointer `BufPtr` to keep track of the address of the start of the next sub-segment to transmit. All there is to do is to get a DMA payload buffer, copy up to `ETH_TX_BUFSIZE` byte and give the payload to the EMAC driver. Then the left over number of bytes to transmit and the address of the next byte to copy is updated. All along, the first ever segment is reported to `ETH_Prepare_Multi_Transmit()` as being the first segment of the packet. Only when the updated number of bytes left to transmit is zero will `ETH_Prepare_Multi_Transmit()` be informed of it.

7.2.2 Sending Non-Segmented Packets

This example, much simpler than the previous, shows the operations required to send full packets. The 3 step sequence is the same except the component `ETH_Prepare_Transmit_Descriptor()` is used instead of the component `ETH_Prepare_Multi_Transmit()`. The example does not include the verification that the packets are indeed non-segmented, nor that the packet size does not exceed the size of the DMA payload buffer. A semaphore that would be posted by the TX done interrupt is used in this example.

Table 7-5 Non-segmented packet transmission

```
SKIP_PAD(p);

for(Pbuf=p ; Pbuf!=NULL ; Pbuf=Pbuf->next) {
    for (ii=0 ; ii<10 ; ii++) {
        DMAbuf = ETH_Get_Transmit_Buffer(EMAC_DEVICE);
        if (DMAbuf != NULL) {
            break;
        }
        SEMwait(EmacTXsema, OS_MS_TO_TICK(20));
    }
    if (DMAbuf == NULL) {
        CLAIM_PAD(p);
        return(ERR_MEM);
    }
    memmove((void *)&DMAbuf[0], Pbuf->payload, (size_t)Pbuf->len);
    ETH_Prepare_Transmit_Desriptor(EMAC_DEVICE, (size_t)Pbuf->len);
}

CLAIM_PAD(p);
```

7.3 Receiving a Packet

7.3.1 Receiving a Packet (internal payload buffers)

Packet reception using the internal payload buffers of the EMAC driver is simple. Basically, extracting the payload of packets that are received through the EMAC driver requires 3 steps:

- Check if a packet is available, and if available how many bytes in the packet. This is done through the use of the component `ETH_Get_Received_Frame_Length()`.
- Copy the payload from the DMA payload buffer into the stack payload buffers. If the packet that was received is segmented, a simple loop handles the segment reconstruction.
- Set-up the DMA descriptor to get it ready to be used by the DMA through the component `ETH_Release_Received()`

Table 7-6 Packet reception (internal buffers)

```
Len = ETH_Get_Received_Frame_Length(EMAC_DEVICE);
if (Len >= 0) {
    Len   += ETH_PAD_SIZE;
    BufDst = pbuf_alloc(PBUF_RAW, Len+ETH_PAS_SIZE, PBUF_POOL);
    SKIP_PAD(BufDst);
    Buf8   = BufDst->payload;
    do {
        Frame = ETH_Get_Received_Multi(EMAC_DEVICE);
        if (Frame.length >= 0) {
            memmove(Buf8, (void *)Frame.buffer, Frame.length);
            Buf8 += Frame.length;
            Len  -= Frame.length;
        }
        else {
            Len = 0;
        }
    } while (Len > 0);

    ETH_Release_Received(EMAC_DEVICE);

    CLAIM_PAD(BufDst);

    STATS_INC(lwip_stats.link.recv);
}

return(DstBuf);
```

7.3.2 Receiving a Packet (external payload buffers)

The proper general code example for receiving a packet using external buffer is more complex than the one shown here. This example is simplified because it requires the received packet to not be segmented.

Table 7-7 Packet reception (external buffers)

```
Len = ETH_Get_Received_Frame_Length(EMAC_DEVICE);
if (Len >= 0) {
    Frame = ETH_Get_Received_Multi(EMAC_DEVICE);
    if (Frame.length == Len) {
        BufDst = Frame.descriptor->PbufPtr;
        pbuf_realloc(BufDst, Frame.length);
        NewPbuf = pbuf_alloc(PBUF_RAW, ETH_RX_BUFSIZE+ETH_PAD_SIZE, PBUF_POOL);
        Frame.descriptor->PbufPtr = NewPbuf;
        if (NewPbuf = NULL) {
            return(NULL);
        }
        SKIP_PAD(NewPbuf);
        Frame.descriptor->Buff = (uint32_t)NewPbuf->payload;
    }

    ETH_Release_Received(EMAC_DEVICE);

    CLAIM_PAD(BufDst);

    STATS_INC(lwip_stats.link.recv);
}

return(NewPbuf);
```

8 References

[R1] Abassi RTOS – User Guide, available at http://www.code-time.com
[R2] mAbassi RTOS – User Guide, available at http://www.code-time.com
[R3] µAbassi RTOS – User Guide, available at http://www.code-time.com