SPI Ethernet Library

The ENC28J60 is a stand-alone Ethernet controller with an industry standard Serial Peripheral Interface (SPI). It is designed to serve as an Ethernet network interface for any controller equipped with SPI.

The ENC28J60 meets all of the IEEE 802.3 specifications. It incorporates a number of packet filtering schemes to limit incoming packets. It also provides an internal DMA module for fast data throughput and hardware assisted IP checksum calculations. Communication with the host controller is implemented via two interrupt pins and the SPI, with data rates of up to 10 Mb/s. Two dedicated pins are used for LED link and network activity indication.

This library is designed to simplify handling of the underlying hardware (ENC28J60). It works with any dsPIC30/33 and PIC24 with integrated SPI and more than 4 Kb ROM memory. 38 to 40 MHz clock is recommended to get from 8 to 10 Mhz SPI clock, otherwise dsPIC30/33 and PIC24 should be clocked by ENC28J60 clock output due to its silicon bug in SPI hardware. If you try lower dsPIC30/33 and PIC24 clock speed, there might be board hang or miss some requests.

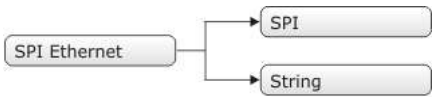
SPI Ethernet library supports:

- IPv4 protocol.
- ARP requests.
- ICMP echo requests.
- UDP requests.
- TCP requests (no stack, no packet reconstruction).
- ARP client with cache.
- DNS client.
- UDP client.
- DHCP client.
- packet fragmentation is **NOT** supported.

Important :

- Global library variable `SPI_Ethernet_userTimerSec` is used to keep track of time for all client implementations (ARP, DNS, UDP and DHCP). It is user responsibility to increment this variable each second in it's code if any of the clients is used.
- For advanced users there is `__EthEnc28j60Private.mpas` unit in Uses folder of the compiler with description of all routines and global variables, relevant to the user, implemented in the SPI Ethernet Library.
- The appropriate hardware SPI module must be initialized before using any of the SPI Ethernet library routines. Refer to SPI Library.
- For MCUs with multiple SPI modules it is possible to initialize them and then switch by using the `SPI_Set_Active()` routine.

Library Dependency Tree



External dependencies of SPI Ethernet Library

The following variables must be defined in all projects using SPI Ethernet Library:	Description:	Examples :
`var SPI_Ethernet_CS : sbit; sfr; external;`	ENC28J60 chip select pin.	`var SPI_Ethernet_CS : sbit at LATF1_bit;`
`var SPI_Ethernet_RST : sbit; sfr; external;`	ENC28J60 reset pin.	`var SPI_Ethernet_RST : sbit at LATF0_bit;`
`var SPI_Ethernet_CS_Direction : sbit; sfr; external;`	Direction of the ENC28J60 chip select pin.	`var SPI_Ethernet_CS_Direction : sbit at TRISF1_bit;`
`var SPI_Ethernet_RST_Direction : sbit; sfr; external;`	Direction of the ENC28J60 reset pin.	`var SPI_Ethernet_RST_Direction : sbit at TRISF0_bit;`

The following routines must be defined in all project using SPI Ethernet Library:	Description:	Examples :
`function SPI_Ethernet_UserTCP(var remoteHost : array[4] of byte, remotePort : word, localPort : word, reqLength : word) var flags: TEthPktFlags) : word;`	TCP request handler.	Refer to the library example at the bottom of this page for code implementation.
`function SPI_Ethernet_UserUDP(var remoteHost : array[4] of byte, remotePort : word, destPort : word, reqLength : word, var flags: TEthPktFlags) : word;`	UDP request handler.	Refer to the library example at the bottom of this page for code implementation.

Library Routines

SPI_Ethernet_Init

Prototype

```
procedure SPI_Ethernet_Init(mac: ^byte; ip: ^byte; fullDuplex: byte);
```

Description

This is MAC module routine. It initializes ENC28J60 controller. This function is internaly splited into 2 parts to help linker when coming short of memory.

ENC28J60 controller settings (parameters not mentioned here are set to default):

- receive buffer start address : 0x0000.
- receive buffer end address : 0x19AD.
- transmit buffer start address: 0x19AE.
- transmit buffer end address : 0x1FFF.
- RAM buffer read/write pointers in auto-increment mode.
- receive filters set to default: CRC + MAC Unicast + MAC Broadcast in OR mode.
- flow control with TX and RX pause frames in full duplex mode.
- frames are padded to 60 bytes + CRC.
- maximum packet size is set to 1518.
- Back-to-Back Inter-Packet Gap: 0x15 in full duplex mode; 0x12 in half duplex mode.
- Non-Back-to-Back Inter-Packet Gap: 0x0012 in full duplex mode; 0x0C12 in half duplex mode.
- Collision window is set to 63 in half duplex mode to accomodate some ENC28J60 revisions silicon bugs.
- CLKOUT output is disabled to reduce EMI generation.
- half duplex loopback disabled.
- LED configuration: default (LEDA-link status, LEDB-link activity).

Parameters

- mac: RAM buffer containing valid MAC address.
- ip: RAM buffer containing valid IP address.
- fullDuplex: ethernet duplex mode switch. Valid values: 0 (half duplex mode) and 1 (full duplex mode).

Returns

Nothing.

Requires

Global variables :

- SPI_Ethernet_CS: Chip Select line
- SPI_Ethernet_CS_Direction: Direction of the Chip Select pin
- SPI_Ethernet_RST: Reset line
- SPI_Ethernet_RST_Direction: Direction of the Reset pin

must be defined before using this function.

The SPI module needs to be initialized. See the SPIx_Init and SPIx_Init_Advanced routines.

Example

```
// SPI Ethernet module connections
var SPI_Ethernet_RST : sbit at RF0_bit;
var SPI_Ethernet_CS : sbit at RF1_bit;
var SPI_Ethernet_RST_Direction : sbit at TRISF0_bit;
var SPI_Ethernet_CS_Direction : sbit at TRISF1_bit;

const SPI_Ethernet_HALFDUPLEX = 0;
const SPI_Ethernet_FULLDUPLEX = 1;

var
  myMacAddr : array[6] of byte; // my MAC address
  myIpAddr  : array[4] of byte; // my IP addr
  ...
  myMacAddr[0] := 0x00;
  myMacAddr[1] := 0x14;
  myMacAddr[2] := 0xA5;
  myMacAddr[3] := 0x76;
  myMacAddr[4] := 0x19;
  myMacAddr[5] := 0x3F;

  myIpAddr[0]  := 192;
  myIpAddr[1]  := 168;
```

```
myIpAddr[2]   := 1;
myIpAddr[3]   := 60;

SPI1_Init();
SPI_Ethernet_Init(myMacAddr, myIpAddr, SPI_Ethernet_FULLDUPLEX);
```

Notes None.

SPI_Ethernet_Enable

Prototype

```
procedure SPI_Ethernet_Enable(enFlt : byte);
```

Description

This is MAC module routine. This routine enables appropriate network traffic on the ENC28J60 module by the means of it's receive filters (unicast, multicast, broadcast, crc). Specific type of network traffic will be enabled if a corresponding bit of this routine's input parameter is set. Therefore, more than one type of network traffic can be enabled at the same time. For this purpose, predefined library constants (see the table below) can be ORed to form appropriate input value.

Advanced filtering available in the ENC28J60 module such as Pattern Match, Magic Packet and Hash Table can not be enabled by this routine. Additionaly, all filters, except CRC, enabled with this routine will work in OR mode, which means that packet will be received if any of the enabled filters accepts it.

This routine will change receive filter configuration on-the-fly. It will not, in any way, mess with enabling/disabling receive/transmit logic or any other part of the ENC28J60 module. The ENC28J60 module should be properly cofigured by the means of SPI_Ethernet_Init routine.

Parameters

- enFlt: network traffic/receive filter flags. Each bit corresponds to the appropriate network traffic/receive filter:

Bit	Mask	Description	Predefined library const
0	0x01	MAC Broadcast traffic/receive filter flag. When set, MAC broadcast traffic will be enabled.	_SPI_Ethernet_BROADCAST
1	0x02	MAC Multicast traffic/receive filter flag. When set, MAC multicast traffic will be enabled.	_SPI_Ethernet_MULTICAST
2	0x04	not used	none
3	0x08	not used	none
4	0x10	not used	none
5	0x20	CRC check flag. When set, packets with invalid CRC field will be discarded.	_SPI_Ethernet_CRC
6	0x40	not used	none
7	0x80	MAC Unicast traffic/receive filter flag. When set, MAC unicast traffic will be enabled.	_SPI_Ethernet_UNICAST

Returns Nothing.

Requires Ethernet module has to be initialized. See SPI_Ethernet_Init.

Example

```
SPI_Ethernet_Enable(_SPI_Ethernet_CRC or _SPI_Ethernet_UNICAST); // enable CRC checking and Unicast traffic
```

Notes

Advanced filtering available in the ENC28J60 module such as Pattern Match, Magic Packet and Hash Table can not be enabled by this routine. Additionaly, all filters, except CRC, enabled with this routine will work in OR mode, which means that packet will be received if any of the enabled filters accepts it.

This routine will change receive filter configuration on-the-fly. It will not, in any way, mess with enabling/disabling receive/transmit logic or any other part of the ENC28J60 module. The ENC28J60 module should be properly cofigured by the means of SPI_Ethernet_Init routine.

SPI_Ethernet_Disable

Prototype

```
procedure SPI_Ethernet_Disable(disFlt : byte);
```

Description

This is MAC module routine. This routine disables appropriate network traffic on the ENC28J60 module by the means of it's receive filters (unicast, multicast, broadcast, crc). Specific type of network traffic will be disabled if a corresponding bit of this routine's input parameter is set. Therefore, more than one type of network traffic can be disabled at the same time. For this purpose, predefined library constants (see the table below) can be ORed to form appropriate input value.

Parameters

- disFlt: network traffic/receive filter flags. Each bit corresponds to the appropriate network traffic/receive filter:

Bit	Mask	Description	Predefined library const
0	0x01	MAC Broadcast traffic/receive filter flag. When set, MAC broadcast traffic will be disabled.	_SPI_Ethernet_BROADCAST
1	0x02	MAC Multicast traffic/receive filter flag. When set, MAC multicast traffic will be	_SPI_Ethernet_MULTICAST

		disabled.	
2	0x04	not used	none
3	0x08	not used	none
4	0x10	not used	none
5	0x20	CRC check flag. When set, CRC check will be disabled and packets with invalid CRC field will be accepted.	_SPI_Ethernet_CRC
6	0x40	not used	none
7	0x80	MAC Unicast traffic/receive filter flag. When set, MAC unicast traffic will be disabled.	_SPI_Ethernet_UNICAST

Returns	Nothing.
Requires	Ethernet module has to be initialized. See SPI_Ethernet_Init.
Example	`SPI_Ethernet_Disable(_SPI_Ethernet_CRC or _SPI_Ethernet_UNICAST); // disable CRC checking and Unicast traffic`
Notes	Advanced filtering available in the ENC28J60 module such as Pattern Match, Magic Packet and Hash Table can not be disabled by this routine. This routine will change receive filter configuration on-the-fly. It will not, in any way, mess with enabling/disabling receive/transmit logic or any other part of the ENC28J60 module. The ENC28J60 module should be properly cofigured by the means of SPI_Ethernet_Init routine.

SPI_Ethernet_doPacket

Prototype	`function SPI_Ethernet_doPacket() : byte;`
Description	This is MAC module routine. It processes next received packet if such exists. Packets are processed in the following manner: ▪ ARP & ICMP requests are replied automatically. ▪ upon TCP request the SPI_Ethernet_UserTCP function is called for further processing. ▪ upon UDP request the SPI_Ethernet_UserUDP function is called for further processing.
Parameters	None.
Returns	▪ 0 - upon successful packet processing (zero packets received or received packet processed successfully). ▪ 1 - upon reception error or receive buffer corruption. ENC28J60 controller needs to be restarted. ▪ 2 - received packet was not sent to us (not our IP, nor IP broadcast address). ▪ 3 - received IP packet was not IPv4. ▪ 4 - received packet was of type unknown to the library.
Requires	Ethernet module has to be initialized. See SPI_Ethernet_Init.

Example	`while true do`
	`  begin`
	`  ...`
	`  SPI_Ethernet_doPacket(); // process received packets`
	`  ...`
	`  end;`

Notes	SPI_Ethernet_doPacket must be called as often as possible in user's code.

SPI_Ethernet_putByte

Prototype	`procedure SPI_Ethernet_putByte(v : byte);`
Description	This is MAC module routine. It stores one byte to address pointed by the current ENC28J60 write pointer (EWRPT).
Parameters	▪ v: value to store
Returns	Nothing.
Requires	Ethernet module has to be initialized. See SPI_Ethernet_Init.
Example	`var` `  data : byte;`

	... SPI_Ethernet_putByte(data); // put an byte into ENC28J60 buffer
Notes	None.

SPI_Ethernet_putBytes

Prototype	`procedure SPI_Ethernet_putBytes(ptr : ^byte; n : word);`
Description	This is MAC module routine. It stores requested number of bytes into ENC28J60 RAM starting from current ENC28J60 write pointer (EWRPT) location.
Parameters	▪ ptr: RAM buffer containing bytes to be written into ENC28J60 RAM. ▪ n: number of bytes to be written.
Returns	Nothing.
Requires	Ethernet module has to be initialized. See SPI_Ethernet_Init.
Example	var buffer : array[17] of byte; ... buffer := 'mikroElektronika'; ... SPI_Ethernet_putBytes(buffer, 16); // put an RAM array into ENC28J60 buffer
Notes	None.

SPI_Ethernet_putConstBytes

Prototype	`procedure SPI_Ethernet_putConstBytes(const ptr : ^byte; n : word);`
Description	This is MAC module routine. It stores requested number of const bytes into ENC28J60 RAM starting from current ENC28J60 write pointer (EWRPT) location.
Parameters	▪ ptr: const buffer containing bytes to be written into ENC28J60 RAM. ▪ n: number of bytes to be written.
Returns	Nothing.
Requires	Ethernet module has to be initialized. See SPI_Ethernet_Init.
Example	const buffer : array[17] of byte; ... buffer := 'mikroElektronika'; ... SPI_Ethernet_putConstBytes(buffer, 16); // put a const array into ENC28J60 buffer
Notes	None.

SPI_Ethernet_putString

Prototype	`function SPI_Ethernet_putString(ptr : ^byte) : word;`
Description	This is MAC module routine. It stores whole string (excluding null termination) into ENC28J60 RAM starting from current ENC28J60 write pointer (EWRPT) location.
Parameters	▪ ptr: string to be written into ENC28J60 RAM.
Returns	Number of bytes written into ENC28J60 RAM.
Requires	Ethernet module has to be initialized. See SPI_Ethernet_Init.
Example	var

	`buffer : string[16];`
	`...`
	`buffer := 'mikroElektronika';`
	`...`
	`SPI_Ethernet_putString(buffer); // put a RAM string into ENC28J60 buffer`

Notes	None.

SPI_Ethernet_putConstString

Prototype	`function SPI_Ethernet_putConstString(const ptr : ^byte) : word;`
Description	This is MAC module routine. It stores whole const string (excluding null termination) into ENC28J60 RAM starting from current ENC28J60 write pointer (EWRPT) location.
Parameters	▪ ptr: const string to be written into ENC28J60 RAM.
Returns	Number of bytes written into ENC28J60 RAM.
Requires	Ethernet module has to be initialized. See SPI_Ethernet_Init.

Example	`const`
	`buffer : string[16];`
	`...`
	`buffer := 'mikroElektronika';`
	`...`
	`SPI_Ethernet_putConstString(buffer); // put a const string into ENC28J60 buffer`

Notes	None.

SPI_Ethernet_getByte

Prototype	`function SPI_Ethernet_getByte() : byte;`
Description	This is MAC module routine. It fetches a byte from address pointed to by current ENC28J60 read pointer (ERDPT).
Parameters	None.
Returns	Byte read from ENC28J60 RAM.
Requires	Ethernet module has to be initialized. See SPI_Ethernet_Init.

Example	`var`
	`buffer : byte;`
	`...`
	`buffer := SPI_Ethernet_getByte(); // read a byte from ENC28J60 buffer`

Notes	None.

SPI_Ethernet_getBytes

Prototype	`procedure SPI_Ethernet_getBytes(ptr : ^byte; addr : word; n : word);`
Description	This is MAC module routine. It fetches equested number of bytes from ENC28J60 RAM starting from given address. If value of 0xFFFF is passed as the address parameter, the reading will start from current ENC28J60 read pointer (ERDPT) location.
Parameters	▪ ptr: buffer for storing bytes read from ENC28J60 RAM. ▪ addr: ENC28J60 RAM start address. Valid values: 0..8192. ▪ n: number of bytes to be read.
Returns	Nothing.
Requires	Ethernet module has to be initialized. See SPI_Ethernet_Init.
Example	`var` `buffer: array[16] of byte;`

```
...
SPI_Ethernet_getBytes(buffer, 0x100, 16); // read 16 bytes, starting from address 0x100
```

Notes	None.

SPI_Ethernet_UserTCP

Prototype	`function SPI_Ethernet_UserTCP(var remoteHost : array[4] of byte; remotePort, localPort, reqLength : word; var flags: TEthPktFlags) : word;`
Description	This is TCP module routine. It is internally called by the library. The user accesses to the TCP request by using some of the SPI_Ethernet_get routines. The user puts data in the transmit buffer by using some of the SPI_Ethernet_put routines. The function must return the length in bytes of the TCP reply, or 0 if there is nothing to transmit. If there is no need to reply to the TCP requests, just define this function with return(0) as a single statement.

Parameters	▪ `remoteHost`: client's IP address.
	▪ `remotePort`: client's TCP port.
	▪ `localPort`: port to which the request is sent.
	▪ `reqLength`: TCP request data field length.
	▪ `flags`: record consisted of two fields :
	Copy Code To Clipboard
	`type TEthPktFlags = record`
	`  canCloseTCP: boolean;  // flag which closes socket`
	`  isBroadcast: boolean;  // flag which denotes that the IP package has been received via subnet broadcast address`
	`end;`

Returns	▪ `0` - there should not be a reply to the request. ▪ Length of TCP reply data field - otherwise.
Requires	Ethernet module has to be initialized. See SPI_Ethernet_Init.
Example	This function is internally called by the library and should not be called by the user's code.
Notes	The function source code is provided with appropriate example projects. The code should be adjusted by the user to achieve desired reply.

SPI_Ethernet_UserUDP

Prototype	`function SPI_Ethernet_UserUDP(var remoteHost : array[4] of byte; remotePort, destPort, reqLength : word; var flags: TEthPktFlags) : word;`
Description	This is UDP module routine. It is internally called by the library. The user accesses to the UDP request by using some of the SPI_Ethernet_get routines. The user puts data in the transmit buffer by using some of the SPI_Ethernet_put routines. The function must return the length in bytes of the UDP reply, or 0 if nothing to transmit. If you don't need to reply to the UDP requests, just define this function with a return(0) as single statement.

Parameters	▪ `remoteHost`: client's IP address.
	▪ `remotePort`: client's port.
	▪ `destPort`: port to which the request is sent.
	▪ `reqLength`: UDP request data field length.
	▪ `flags`: record consisted of two fields :
	Copy Code To Clipboard
	`type TEthPktFlags = record`
	`  canCloseTCP: boolean;  // flag which closes socket (not relevant to UDP)`
	`  isBroadcast: boolean;  // flag which denotes that the IP package has been received via subnet broadcast address`
	`end;`

Returns	▪ `0` - there should not be a reply to the request. ▪ Length of UDP reply data field - otherwise.
Requires	Ethernet module has to be initialized. See SPI_Ethernet_Init.
Example	This function is internally called by the library and should not be called by the user's code.
Notes	The function source code is provided with appropriate example projects. The code should be adjusted by the user to achieve desired reply.

SPI_Ethernet_setUserHandlers

Prototype	`procedure SPI_Ethernet_setUserHandlers(TCPHandler : ^TSPI_Ethernet_UserTCP; UDPHandler : ^TSPI_Ethernet_UserUDP);`
Description	Sets pointers to User TCP and UDP handler function implementations, which are automatically called by SPI Ethernet library.
Parameters	▪ `TCPHandler`: TCP request handler ▪ `UDPHandler`: UDP request handler.
Returns	Nothing.
Requires	SPI_Ethernet_UserTCP and SPI_Ethernet_UserUDP have to be previously defined.
Example	`SPI_Ethernet_setUserHandlers(@SPI_Ethernet_UserTCP, @SPI_Ethernet_UserUDP);`
Notes	Since all libraries are built for SSA, SSA restrictions regarding function pointers dictate that modules that use SPI_Ethernet_setUserHandlers must also be built for SSA.

SPI_Ethernet_getIpAddress

Prototype	`function SPI_Ethernet_getIpAddress() : ^byte;`
Description	This routine should be used when DHCP server is present on the network to fetch assigned IP address.
Parameters	None.
Returns	Pointer to the global variable holding IP address.
Requires	Ethernet module has to be initialized. See SPI_Ethernet_Init.

Example	`var`
	`  ipAddr : array[4] of byte;  // user IP address buffer`
	`  ...`
	`  memcpy(ipAddr, SPI_Ethernet_getIpAddress(), 4); // fetch IP address`

Notes	User should always copy the IP address from the RAM location returned by this routine into it's own IP address buffer. These locations should not be altered by the user in any case!

SPI_Ethernet_getGwIpAddress

Prototype	`function SPI_Ethernet_getGwIpAddress() : ^byte;`
Description	This routine should be used when DHCP server is present on the network to fetch assigned gateway IP address.
Parameters	None.
Returns	Pointer to the global variable holding gateway IP address.
Requires	Ethernet module has to be initialized. See SPI_Ethernet_Init.

Example	`var`
	`  gwIpAddr : array[4] of byte;  // user gateway IP address buffer`
	`  ...`
	`  memcpy(gwIpAddr, SPI_Ethernet_getGwIpAddress(), 4); // fetch gateway IP address`

Notes	User should always copy the IP address from the RAM location returned by this routine into it's own gateway IP address buffer. These locations should not be altered by the user in any case!

SPI_Ethernet_getDnsIpAddress

Prototype	`function SPI_Ethernet_getDnsIpAddress() : ^byte;`
Description	This routine should be used when DHCP server is present on the network to fetch assigned DNS IP address.
Parameters	None.

Returns	Pointer to the global variable holding DNS IP address.
Requires	Ethernet module has to be initialized. See SPI_Ethernet_Init.

Example	`var`
	`  dnsIpAddr : array[4] of byte;  // user DNS IP address buffer`
	`  ...`
	`  memcpy(dnsIpAddr, SPI_Ethernet_getDnsIpAddress(), 4); // fetch DNS server address`

Notes	User should always copy the IP address from the RAM location returned by this routine into it's own DNS IP address buffer. These locations should not be altered by the user in any case!

SPI_Ethernet_getIpMask

Prototype	`function SPI_Ethernet_getIpMask() : ^byte;`
Description	This routine should be used when DHCP server is present on the network to fetch assigned IP subnet mask.
Parameters	None.
Returns	Pointer to the global variable holding IP subnet mask.
Requires	Ethernet module has to be initialized. See SPI_Ethernet_Init.

Example	`var`
	`  IpMask : array[4] of byte;  // user IP subnet mask buffer`
	`  ...`
	`  memcpy(IpMask, SPI_Ethernet_getIpMask(), 4); // fetch IP subnet mask`

Notes	User should always copy the IP address from the RAM location returned by this routine into it's own IP subnet mask buffer. These locations should not be altered by the user in any case!

SPI_Ethernet_confNetwork

Prototype	`procedure SPI_Ethernet_confNetwork(var ipMask, gwIpAddr, dnsIpAddr : array[4] of byte);`
Description	Configures network parameters (IP subnet mask, gateway IP address, DNS IP address) when DHCP is not used.
Parameters	▪ `ipMask`: IP subnet mask. ▪ `gwIpAddr` gateway IP address. ▪ `dnsIpAddr`: DNS IP address.
Returns	Nothing.
Requires	Ethernet module has to be initialized. See SPI_Ethernet_Init.

Example	`var`
	`  ipMask    : array[4] of byte;  // network mask (for example : 255.255.255.0)`
	`  gwIpAddr  : array[4] of byte;  // gateway (router) IP address`
	`  dnsIpAddr : array[4] of byte;  // DNS server IP address`
	`  ...`
	`  gwIpAddr[0]  := 192;`
	`  gwIpAddr[1]  := 168;`
	`  gwIpAddr[2]  := 20;`
	`  gwIpAddr[3]  := 6;`
	
	`  dnsIpAddr[0] := 192;`
	`  dnsIpAddr[1] := 168;`
	`  dnsIpAddr[2] := 20;`
	`  dnsIpAddr[3] := 100;`
	
	`  ipMask[0]    := 255;`
	`  ipMask[1]    := 255;`
	`  ipMask[2]    := 255;`
	`  ipMask[3]    := 0;`
	`  ...`
	`  SPI_Ethernet_confNetwork(ipMask, gwIpAddr, dnsIpAddr); // set network configuration parameters`

Notes	The above mentioned network parameters should be set by this routine only if DHCP module is not used. Otherwise DHCP will override these settings.

SPI_Ethernet_arpResolve

Prototype	`function SPI_Ethernet_arpResolve(var ip : array[4] of byte; tmax : byte) : ^byte;`
Description	This is ARP module routine. It sends an ARP request for given IP address and waits for ARP reply. If the requested IP address was resolved, an ARP cash entry is used for storing the configuration. ARP cash can store up to 3 entries. For ARP cash structure refer to `"eth_enc28j60LibDef.mpas"` file in the compiler's Uses folder.
Parameters	• `ip`: IP address to be resolved. • `tmax`: time in seconds to wait for an reply.
Returns	• MAC address behind the IP address - the requested IP address was resolved. • `0` - otherwise.
Requires	Ethernet module has to be initialized. See SPI_Ethernet_Init.

Example	`var`
	`  IpAddr : array[4] of byte;  // IP address`
	`  ...`
	`  IpAddr[0] := 192;`
	`  IpAddr[0] := 168;`
	`  IpAddr[0] := 1;`
	`  IpAddr[0] := 1;`
	`  ...`
	`  SPI_Ethernet_arpResolve(IpAddr, 5); // get MAC address behind the above IP address, wait 5 secs for the response`

Notes	The Ethernet services are not stopped while this routine waits for ARP reply. The incoming packets will be processed normaly during this time.

SPI_Ethernet_sendUDP

Prototype	`function SPI_Ethernet_sendUDP(var destIP : array[4] of byte; sourcePort, destPort : word; pkt : ^byte; pktLen : word) :`
Description	This is UDP module routine. It sends an UDP packet on the network.
Parameters	• `destIP`: remote host IP address. • `sourcePort`: local UDP source port number. • `destPort`: destination UDP port number. • `pkt`: packet to transmit. • `pktLen`: length in bytes of packet to transmit.
Returns	• `1` - UDP packet was sent successfully. • `0` - otherwise.
Requires	Ethernet module has to be initialized. See SPI_Ethernet_Init.

Example	`var`
	`  IpAddr : array[4] of byte;  // remote IP address`
	`  ...`
	`  IpAddr[0] := 192;`
	`  IpAddr[0] := 168;`
	`  IpAddr[0] := 1;`
	`  IpAddr[0] := 1;`
	`  ...`
	`  SPI_Ethernet_sendUDP(IpAddr, 10001, 10001, 'Hello', 5); // send Hello message to the above IP address, from UDP port 10001 to`

Notes	None.

SPI_Ethernet_dnsResolve

Prototype	`function SPI_Ethernet_dnsResolve(var host : string; tmax : byte) : ^byte;`
Description	This is DNS module routine. It sends an DNS request for given host name and waits for DNS reply. If the requested host name was resolved, it's IP address is stored in library global variable and a pointer containing this address is returned by the routine. UDP port `53` is used as DNS port.
Parameters	• `host`: host name to be resolved. • `tmax`: time in seconds to wait for an reply.

Returns	▪ pointer to the location holding the IP address - the requested host name was resolved. ▪ 0 - otherwise.
Requires	Ethernet module has to be initialized. See SPI_Ethernet_Init.

Example	`var`
	`  remoteHostIpAddr : array[4] of byte;  // user host IP address buffer`
	`  ...`
	`  // SNTP server:`
	`  // Zurich, Switzerland: Integrated Systems Lab, Swiss Fed. Inst. of Technology`
	`  // 129.132.2.21: swisstime.ethz.ch`
	`  // Service Area: Switzerland and Europe`
	`  memcpy(remoteHostIpAddr, SPI_Ethernet_dnsResolve('swisstime.ethz.ch', 5), 4);`

Notes	The Ethernet services are not stopped while this routine waits for DNS reply. The incoming packets will be processed normaly during this time. User should always copy the IP address from the RAM location returned by this routine into it's own resolved host IP address buffer. These locations should not be altered by the user in any case!

SPI_Ethernet_initDHCP

Prototype	`function SPI_Ethernet_initDHCP(tmax : byte) : byte;`
Description	This is DHCP module routine. It sends an DHCP request for network parameters (IP, gateway, DNS addresses and IP subnet mask) and waits for DHCP reply. If the requested parameters were obtained successfully, their values are stored into the library global variables. These parameters can be fetched by using appropriate library IP get routines: ▪ SPI_Ethernet_getIpAddress - fetch IP address. ▪ SPI_Ethernet_getGwIpAddress - fetch gateway IP address. ▪ SPI_Ethernet_getDnsIpAddress - fetch DNS IP address. ▪ SPI_Ethernet_getIpMask - fetch IP subnet mask. UDP port 68 is used as DHCP client port and UDP port 67 is used as DHCP server port.
Parameters	▪ tmax: time in seconds to wait for an reply.
Returns	▪ 1 - network parameters were obtained successfully. ▪ 0 - otherwise.
Requires	Ethernet module has to be initialized. See SPI_Ethernet_Init.

Example	`...`
	`SPI_Ethernet_initDHCP(5); // get network configuration from DHCP server, wait 5 sec for the response`
	`...`

Notes	The Ethernet services are not stopped while this routine waits for DNS reply. The incoming packets will be processed normaly during this time. When DHCP module is used, global library variable SPI_Ethernet_userTimerSec is used to keep track of time. It is user responsibility to increment this variable each second in it's code.

SPI_Ethernet_doDHCPLeaseTime

Prototype	`function SPI_Ethernet_doDHCPLeaseTime() : byte;`
Description	This is DHCP module routine. It takes care of IP address lease time by decrementing the global lease time library counter. When this time expires, it's time to contact DHCP server and renew the lease.
Parameters	None.
Returns	▪ 0 - lease time has not expired yet. ▪ 1 - lease time has expired, it's time to renew it.
Requires	Ethernet module has to be initialized. See SPI_Ethernet_Init.

Example	`while true do`
	`  begin`
	`  ...`

```
    if (SPI_Ethernet_doDHCPLeaseTime() <> 0) then
      begin
        ... // it's time to renew the IP address lease
      end;
  end;
```

Notes	None.

SPI_Ethernet_renewDHCP

Prototype	`function SPI_Ethernet_renewDHCP(tmax : byte) : byte;`
Description	This is DHCP module routine. It sends IP address lease time renewal request to DHCP server.
Parameters	▪ `tmax`: time in seconds to wait for an reply.
Returns	▪ `1` - upon success (lease time was renewed). ▪ `0` - otherwise (renewal request timed out).
Requires	Ethernet module has to be initialized. See SPI_Ethernet_Init.

Example	`while true do`
	`  begin`
	`    ...`
	`    if (SPI_Ethernet_doDHCPLeaseTime() <> 0) then`
	`      begin`
	`        SPI_Ethernet_renewDHCP(5); // it's time to renew the IP address lease, with 5 secs for a reply`
	`      end;`
	`    ...`
	`  end;`

Notes	None.

Library Example

This code shows how to use the Ethernet mini library :

- the board will reply to ARP & ICMP echo requests
- the board will reply to UDP requests on any port :
  - returns the request in upper char with a header made of remote host IP & port number
- the board will reply to HTTP requests on port 80, GET method with pathnames :
  - / will return the HTML main page
  - /s will return board status as text string
  - /t0 ... /t7 will toggle RD0 to RD7 bit and return HTML main page
  - all other requests return also HTML main page.

Copy Code To Clipboard

```
program HTTP_Demo;

{*********************************
 * RAM variables
 *}

// mE ehternet NIC pinout
var
  SPI_Ethernet_Rst : sbit at LATF0_bit;  // for writing to output pin always use latch
  SPI_Ethernet_CS  : sbit at LATF1_bit;  // for writing to output pin always use latch
  SPI_Ethernet_Rst_Direction : sbit at TRISF0_bit;
  SPI_Ethernet_CS_Direction  : sbit at TRISF1_bit;
// end ethernet NIC definitions

var myMacAddr   : array[6] of byte; // my MAC address
    myIpAddr    : array[4] of byte; // my IP address
    gwIpAddr    : array[4] of byte; // gateway (router) IP address
    ipMask      : array[4] of byte; // network mask (for example : 255.255.255.0)
    dnsIpAddr   : array[4] of byte; // DNS server IP address

{*********************************************************
 * ROM constant strings
 *}
const httpHeader : string[30] = 'HTTP/1.1 200 OK'+#10+'Content-type: ';  // HTTP header
const httpMimeTypeHTML : string[11]   = 'text/html'+#10+#10;             // HTML MIME type
const httpMimeTypeScript : string[12] = 'text/plain'+#10+#10;            // TEXT MIME type
const httpMethod : string[5] = 'GET /';
{*
 * web page, splited into 2 parts :
 * when coming short of ROM, fragmented data is handled more efficiently by linker
 *
```

```
 * this HTML page calls the boards to get its status, and builds itself with javascript
 *}
const indexPage : string[761] =
                    '<meta http-equiv="refresh" content="3;url=http://192.168.20.60">' +
                    '<HTML><HEAD></HEAD><BODY>'+
                    '<h1>dsPIC + ENC28J60 Mini Web Server</h1>'+
                    '<a href=/>Reload</a>'+
                    '<script src=/s></script>'+
                    '<table><tr><td valign=top><table border=1 style="font-size:20px ;font-family: terminal ;">'+
                    '<tr><th colspan=2>ADC</th></tr>'+
                    '<tr><td>AN0</td><td><script>document.write(AN0)</script></td></tr>'+
                    '<tr><td>AN1</td><td><script>document.write(AN1)</script></td></tr>'+
                    '</table></td><td><table border=1 style="font-size:20px ;font-family: terminal ;">'+
                    '<tr><th colspan=2>PORTB</th></tr>'+
                    '<script>'+
                    'var str,i;'+
                    'str="";'+
                    'for(i=2;i<10;i++)'+
                    '{str+="<tr><td bgcolor=pink>BUTTON #"+i+"</td>";'+
                    'if(PORTB&(1<<i)){str+="<td bgcolor=red>ON";}'+
                    'else {str+="<td bgcolor=#cccccc>OFF";}'+
                    'str+="</td></tr>";}'+
                    'document.write(str);'+
                    '</script>';

const indexPage2 : string[466] =
                    '</table></td><td>'+
                    '<table border=1 style="font-size:20px ;font-family: terminal ;">'+
                    '<tr><th colspan=3>PORTD</th></tr>'+
                    '<script>'+
                    'var str,i;'+
                    'str="";'+
                    'for(i=0;i<4;i++)'+
                    '{str+="<tr><td bgcolor=yellow>LED #"+i+"</td>";'+
                    'if(PORTD&(1<<i)){str+="<td bgcolor=red>ON";}'+
                    'else {str+="<td bgcolor=#cccccc>OFF";}'+
                    'str+="</td><td><a href=/t"+i+">Toggle</a></td></tr>";}'+
                    'document.write(str);'+
                    '</script>'+
                    '</table></td></tr></table>'+
                    'This is HTTP request #<script>document.write(REQ)</script></BODY></HTML>';

var    getRequest  : array[15] of byte;  // HTTP request buffer
       dyna        : array[30] of char;  // buffer for dynamic response
       httpCounter : word;               // counter of HTTP requests

{*****************************************
 * user defined functions
 *}

{*
 * this function is called by the library
 * the user accesses to the HTTP request by successive calls to SPI_Ethernet_getByte()
 * the user puts data in the transmit buffer by successive calls to SPI_Ethernet_putByte()
 * the function must return the length in bytes of the HTTP reply, or 0 if nothing to transmit
 *
 * if you don't need to reply to HTTP requests,
 * just define this function with a return(0) as single statement
 *
 *}
function SPI_Ethernet_UserTCP(var remoteHost : array[4] of byte;
                              remotePort, localPort, reqLength : word; var flags: TEthPktFlags) : word;
  var  i : word;       // my reply length
       bitMask : byte; // for bit mask
       tmp: string[11]; // to copy const array to ram for memcmp
  begin
    result := 0;

    // should we close tcp socket after response is sent?
    // library closes tcp socket by default if canCloseTCP flag is not reset here
    // flags.canCloseTCP := 0; // 0 - do not close socket
                               // otherwise - close socket

    if(localPort <> 80) then  // I listen only to web request on port 80
      begin
        result := 0;
        exit;
      end;

    // get 10 first bytes only of the request, the rest does not matter here
    for i := 0 to 9 do
      getRequest[i] := SPI_Ethernet_getByte();
    getRequest[i] := 0;

    // copy httpMethod to ram for use in memcmp routine
    for i := 0 to 4 do
      tmp[i] := httpMethod[i];

    if(memcmp(@getRequest, @tmp, 5) <> 0) then  // only GET method is supported here
      begin
        result := 0;
        exit;
```

```
      end;

    Inc(httpCounter);                          // one more request done

    if(getRequest[5] = 's') then               // if request path name starts with s, store dynamic data in transmit buffer
      begin
        // the text string replied by this request can be interpreted as javascript statements
        // by browsers
        result := SPI_Ethernet_putConstString(@httpHeader);                  // HTTP header
        result := result + SPI_Ethernet_putConstString(@httpMimeTypeScript);  // with text MIME type

        // add AN2 value to reply
        WordToStr(ADC1_Get_Sample(0), dyna);
        tmp := 'var AN0=';
        result := result + SPI_Ethernet_putString(@tmp);
        result := result + SPI_Ethernet_putString(@dyna);
        tmp := ';';
        result := result + SPI_Ethernet_putString(@tmp);

        // add AN3 value to reply
        WordToStr(ADC1_Get_Sample(1), dyna);
        tmp := 'var AN1=';
        result := result + SPI_Ethernet_putString(@tmp);
        result := result + SPI_Ethernet_putString(@dyna);
        tmp := ';';
        result := result + SPI_Ethernet_putString(@tmp);

        // add PORTB value (buttons) to reply
        tmp := 'var PORTB= ';
        result := result + SPI_Ethernet_putString(@tmp);
        WordToStr(PORTB, dyna);
        result := result + SPI_Ethernet_putString(@dyna);
        tmp := ';';
        result := result + SPI_Ethernet_putString(@tmp);

        // add PORTD value (LEDs) to reply
        tmp := 'var PORTD= ';
        result := result + SPI_Ethernet_putString(@tmp);
        WordToStr(PORTD, dyna);
        result := result + SPI_Ethernet_putString(@dyna);
        tmp := ';';
        result := result + SPI_Ethernet_putString(@tmp);

        // add HTTP requests counter to reply
        WordToStr(httpCounter, dyna);
        tmp := 'var REQ=   ';
        result := result + SPI_Ethernet_putString(@tmp);
        result := result + SPI_Ethernet_putString(@dyna);
        tmp := ';';
        result := result + SPI_Ethernet_putString(@tmp);
      end
    else
      if(getRequest[5] = 't') then                      // if request path name starts with t, toggle PORTD (LED) bit number that comes after
        begin
          bitMask := 0;
          if(isdigit(getRequest[6]) <> 0) then          // if 0 <= bit number <= 9, bits 8 & 9 does not exist but does not matter
            begin
              bitMask := getRequest[6] - '0';           // convert ASCII to integer
              bitMask := 1 shl bitMask;                 // create bit mask
              PORTD   := PORTD xor bitMask;             // toggle PORTD with xor operator
            end;
        end;

    if(result = 0) then // what do to by default
      begin
        result := SPI_Ethernet_putConstString(@httpHeader);                 // HTTP header
        result := result + SPI_Ethernet_putConstString(@httpMimeTypeHTML);  // with HTML MIME type
        result := result + SPI_Ethernet_putConstString(@indexPage);         // HTML page first part
        result := result + SPI_Ethernet_putConstString(@indexPage2);        // HTML page second part
      end;
    // return to the library with the number of bytes to transmit
end;

{*
 * this function is called by the library
 * the user accesses to the UDP request by successive calls to SPI_Ethernet_getByte()
 * the user puts data in the transmit buffer by successive calls to SPI_Ethernet_putByte()
 * the function must return the length in bytes of the UDP reply, or 0 if nothing to transmit
 *
 * if you don't need to reply to UDP requests,
 * just define this function with a return(0) as single statement
 *
 *}
function SPI_Ethernet_UserUDP(var remoteHost : array[4] of byte;
                              remotePort, destPort, reqLength : word; var flags: TEthPktFlags) : word;
  var tmp : string[5];
  begin
    result := 0;
    // reply is made of the remote host IP address in human readable format
    byteToStr(remoteHost[0], dyna);           // first IP address byte
    dyna[3] := '.';
    byteToStr(remoteHost[1], tmp);            // second
```

```
    dyna[4] := tmp[0];
    dyna[5] := tmp[1];
    dyna[6] := tmp[2];
    dyna[7] := '.';
    byteToStr(remoteHost[2], tmp);            // second
    dyna[8] := tmp[0];
    dyna[9] := tmp[1];
    dyna[10] := tmp[2];
    dyna[11] := '.';
    byteToStr(remoteHost[3], tmp);            // second
    dyna[12] := tmp[0];
    dyna[13] := tmp[1];
    dyna[14] := tmp[2];

    dyna[15] := ':';                          // add separator

    // then remote host port number
    WordToStr(remotePort, tmp);
    dyna[16] := tmp[0];
    dyna[17] := tmp[1];
    dyna[18] := tmp[2];
    dyna[19] := tmp[3];
    dyna[20] := tmp[4];
    dyna[21] := '[';
    WordToStr(destPort, tmp);
    dyna[22] := tmp[0];
    dyna[23] := tmp[1];
    dyna[24] := tmp[2];
    dyna[25] := tmp[3];
    dyna[26] := tmp[4];
    dyna[27] := ']';
    dyna[28] := 0;

    // the total length of the request is the length of the dynamic string plus the text of the request
    result := 28 + reqLength;

    // puts the dynamic string into the transmit buffer
    SPI_Ethernet_putBytes(@dyna, 28);

    // then puts the request string converted into upper char into the transmit buffer
    while(reqLength <> 0) do
      begin
        SPI_Ethernet_putByte(SPI_Ethernet_getByte());
        reqLength := reqLength - 1;
      end;
    // back to the library with the length of the UDP reply
  end;

begin
  ADPCFG := 0xFFFD;          // all digital but rb10(AN10)

  PORTB := 0;
  TRISB := 0xffff;           // set PORTB as input for buttons and adc

  PORTD := 0;
  TRISD := 0;                // set PORTD as output,

  ADC1_Init();

  httpCounter := 0;

  // set mac address
  myMacAddr[0] := 0x00;
  myMacAddr[1] := 0x14;
  myMacAddr[2] := 0xA5;
  myMacAddr[3] := 0x76;
  myMacAddr[4] := 0x19;
  myMacAddr[5] := 0x3F;

  // set IP address
  myIpAddr[0] := 192;
  myIpAddr[1] := 168;
  myIpAddr[2] := 20;
  myIpAddr[3] := 60;

  // set gateway address
  gwIpAddr[0]  := 192;
  gwIpAddr[1]  := 168;
  gwIpAddr[2]  := 20;
  gwIpAddr[3]  := 6;

  // set dns address
  dnsIpAddr[0] := 192;
  dnsIpAddr[1] := 168;
  dnsIpAddr[2] := 20;
  dnsIpAddr[3] := 1;

  // set subnet mask
  ipMask[0]    := 255;
  ipMask[1]    := 255;
  ipMask[2]    := 255;
  ipMask[3]    := 0;
```

```
  {*
   * starts ENC28J60 with :
   * reset bit on PORTC.B0
   * CS bit on PORTC.B1
   * my MAC & IP address
   * full duplex
   *}

  SPI1_Init_Advanced(_SPI_MASTER, _SPI_8_BIT, _SPI_PRESCALE_SEC_1, _SPI_PRESCALE_PRI_4,
                     _SPI_SS_DISABLE, _SPI_DATA_SAMPLE_MIDDLE, _SPI_CLK_IDLE_LOW, _SPI_IDLE_2_ACTIVE);
  SPI_Ethernet_Init(myMacAddr, myIpAddr, _SPI_Ethernet_FULLDUPLEX);               // init ethernet module
  SPI_Ethernet_setUserHandlers(@SPI_Ethernet_UserTCP, @SPI_Ethernet_UserUDP);     // set user handlers

  // dhcp will not be used here, so use preconfigured addresses
  SPI_Ethernet_confNetwork(ipMask, gwIpAddr, dnsIpAddr);

  while true do                   // do forever
    begin
      SPI_Ethernet_doPacket();    // process incoming Ethernet packets

      {*
       * add your stuff here if needed
       * SPI_Ethernet_doPacket() must be called as often as possible
       * otherwise packets could be lost
       *}
    end;
end.
```

HW Connection