# API Reference

12/13/2014

## Content

# 1 DIO library

## 1.1.DI_Read

1.1   int DI_Read(unsigned char *output)

    1.1.1 output[out]: Digital Input value in byte:

1.1.1.1 output[0]: MSB: DI#8, LSB: DI#1

1.1.1.2 output[1]: MSB: DI#16, LSB: DI#9

1.1.2 return:

1.1.2.1 number of DI channel: 16

## 1.2. DO_Read

1.2  int DO_Read(unsigned char *output)

1.2.1 output[out]: DO (Relay) value in byte:

1.2.1.1 output[0]: MSB: DO#8, LSB: DO#1

1.2.2 return:

1.2.2.1 number of DO channel: 8

## 1.3. DO_Write

1.3  int DO_Write(unsigned char do_value)

1.3.1 do_value[in]: DO(Relay) valuein byte:

1.3.1.1 MSB: DO#8, LSB: DO#1

1.3.2 return:

1.3.2.1 number of DO channel: 8

## 1.4. DO_WriteBit

1.4  int DO_WriteBit(int no, unsigned char do_value)

1.4.1 no[in]: DO channel number

1.4.2 do_value[in]: 0 or 1

1.4.3 return:

1.4.3.1 0:OK

1.4.3.2 -1:channel number error

1.4.3.3 -2: error in value

# 2 Temperature library

## 2.1. Sensor_GetID

2.1   int Sensor_GetID(unsigned char *output)
 2.1.1 output[out]: sensor ID (8 bytes long per sensor)：
  2.1.1.1 output[0]~output[7]: Sensor#1
  2.1.1.2 output[8]~output[15]: Sensor#2
  2.1.1.3 output[16]~output[23]: Sensor#3
 2.1.2 return:
  2.1.2.1 sensor number:

## 2.2. Sensor_ ReadByID

2.2   int Sensor_ReadByID(unsigned char *id, int unit, float *output)
 2.2.1 id[in]: Sensor ID，8 bytes: id[0] ~id[7]
 2.2.2 unit[in]: 0: degree C, 1: degree F
 2.2.3 output[out]: temperature value
 2.2.4 return:
  2.2.4.1 0: OK
  2.2.4.2 -1: ID error

## 2.3. Sensor_ ReadAll

2.3   int Sensor_ReadAll(int unit, Sensor_struct *resp)
 2.3.1 unit[in]: 0: degree C, 1: degree F
 2.3.2 resp[out]: Sensor_struct: unsigned ID[8], float value
 2.3.3 return:
  2.3.3.1 number of sensor：0~3

# 3  Network library

## 2.4.  BSD socket library

2.4.1. Reference link: http://en.wikipedia.org/wiki/Berkeley_sockets

We support:

2.4.1.1.  int socket(int domain, int type, int protocol)

2.4.1.2.  int bind(int s, const struct sockaddr *name, socklen_t namelen)

2.4.1.3.  int listen(int s, int backlog)

2.4.1.4.  int accept(int s, struct sockaddr *addr, socklen_t *addrlen)

2.4.1.5.  int recv(int s, void *mem, size_t len, int flags)

2.4.1.6.  int send(int s, void *dataptr, size_t size, int flags)

2.4.1.7.  int closesokcet(int s)

## 2.5.  Network connection functions

### 2.5.1.  netconn_new

2.5.1.1.  struct netconn * netconn_new(enum netconn type type)

2.5.1.2.  Creates a new connection abstraction structure. The argument can be one of NETCONN_TCP or NETCONN_UDP, yielding either a TCP or a UDP connection. No connection is established by the call to this function and no data is sent over the network.

### 2.5.2.  netconn_bind

2.5.2.1.  err_t netconn_bind(struct netconn *conn, ip_addr_t *addr, u16_t port)

2.5.2.2.  Binds the connection conn to the local IP address addr and TCP or UDP port port. If addr is NULL the local IP address is determined by the networking system.

## 2.5.3.  netconn_listen

2.5.3.1.  err_t netconn_ listen(struct netconn *conn)

2.5.3.2.  Puts the TCP connection conn into the TCP LISTEN state.

## 2.5.4.  netconn_accept

2.5.4.1.  err_t netconn_accept(struct netconn *conn, struct netconn **new_conn)

2.5.4.2.  Blocks the process until a connection request from a remote host arrives on the TCP connection conn. The connection must be in the LISTEN state so netconn_listen() must be called prior to netconn_accept(). When a connection is established with the remote host, a new connection structure is returned.

## 2.5.5.  netconn_recv

2.5.5.1.  err_t netconn_recv(struct netconn *conn, struct netbuf **new_buf)

2.5.5.2.  Blocks the process while waiting for data to arrive on the connection conn. If the connection has been closed by the remote host, NULL is returned, otherwise a netbuf containing the recevied data is returned.

## 2.5.6.  netconn_write

2.5.6.1.  err_t netconn_write(struct netconn *conn, const void *dataptr, size_t size, u8_t apiflags)

2.5.6.2.  This function is only used for TCP connections. It puts the data pointed to by data on the output queue for the TCP connection conn. The length of the data is given by len. There is no restriction on the length of the data. This function does not require the application to explicitly allocate buffers, as this is taken care of by the stack. The flags parameter has two possible states, as shown below.

2.5.6.2.1. #define NETCONN_NOCOPY 0x00

#define NETCONN_COPY 0x01

When passed the flag NETCONN_COPY the data is copied into internal buffer which is allocated for the data. This allows the data to be modified directly after the call, but is inefficient both in terms of execution time and memory usage. If the flag NETCONN_NOCOPY is used, the data is not copied but rather referenced.

The data must not be modified after the call, since the data can be put on the retransmission queue for the connection, and stay there for an indeterminate amount of time. This is useful when sending data that is located in ROM and therefore is immutable. If greater control over the modifiability of the data is needed, a combination of copied and non-copied data can be used, as seen in the example below.

### 2.5.7. netconn_close

2.5.7.1. err_t netconn_close(struct netconn *conn)
2.5.7.2. Closes the connection conn.

### 2.5.8. netconn_delete

2.5.8.1. err_t netconn_delete(struct netconn *conn)
2.5.8.2. Deallocates the netconn conn. If the connection is open, it is closed as a result of this call.

## 2.6. lwip socket library

### 2.6.1. lwip_socket

2.6.1.1. int lwip_socket(int domain, int type, int protocol)
2.6.1.2. The lwip_socket() call allocates a socket. The parameters to lwip_socket() are used to specify what type of socket that is requested. Since this socket API implementation is concerned only with network sockets, these are the only socket type that is supported. Also, only UDP (SOCK DGRAM) or TCP (SOCK STREAM) sockets can be used.

### 2.6.2. lwip_bind

2.6.2.1. int lwip_bind(int s, const struct sockaddr *name, socklen_t namelen)
2.6.2.2. The lwip_bind() call binds the socket to a local address. In the call to lwip_bind() the local IP address and port number are specified.

### 2.6.3.  lwip_listen

2.6.3.1.  int lwip_listen(int s, int backlog)

2.6.3.2.  Puts the TCP connection conn into the TCP LISTEN state. The backlog parameter is ignored in this version.

### 2.6.4.  lwip_accept

2.6.4.1.  int lwip_accept(int s, struct sockaddr *addr, socklen_t *addrlen)

2.6.4.2.  If no pending connections are present on the queue, and the socket is not marked as nonblocking, lwip_accept() blocks the caller until a connection is present. If the socket is marked nonblocking and no pending connections are present on the queue, lwip_accept() fails with the error **EAGAIN** or **EWOULDBLOCK**.

### 2.6.5.  lwip_recv

2.6.5.1.  int lwip_recv(int s, void *mem, size_t len, int flags)

2.6.5.2.  The lwip_recv() call is used on a connected socket to receive data. They can be used for both TCP and UDP connections.

### 2.6.6.  lwip_send

2.6.6.1.  int lwip_send(int s, void *dataptr, size_t size, int flags)

2.6.6.2.  The lwip_send() call is used in both UDP and TCP connection for sending data. Before a call to lwip_send() the receiver of the data must have been set up using lwip_connect(). For UDP sessions, the lwip_send() call resembles the netconn_send() function from the lwIP API, but since the lwIP API require the application to explicitly allocate buffers, a buffer must be allocated and deallocated within the lwip_send() call. Therefore, a buffer is allocated and the data is copied into the allocated buffer.

### 2.6.7.  lwip_close

2.6.7.1.  int lwip_close(int s)

2.6.7.2.  Closes the connection conn

# 4 UART library

# 5  EEPROM library

# 6  Web library

5.1.3. web_ajax.c : please refer to example program in E05_web&settings