API Reference

Content

1 Network library

1.1. BSD socket library

1.1.1. socket

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

1.1.1.2. The socket() call allocates a BSD socket. The parameters to 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.

1.1.2. bind

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

1.1.2.2. The bind() call binds the BSD socket to a local address. In the call to bind() the local IP address and port number are specified.

1.1.3. listen

1.1.3.1. int listen(int s, int backlog)

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

1.1.4. accept

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

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

1.1.5. recv

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

1.1.5.2. In the BSD socket API, the recv() and read() calls are used on a connected socket to receive data. They can be used for both TCP and UDP connections.

1.1.6. send

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

1.1.6.2. In the BSD socket API, the send() call is used in both UDP and TCP connection for sending data. Before a call to send() the receiver of the data must have been set up using connect(). For UDP sessions, the 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 send() call. Therefore, a buffer is allocated and the data is copied into the allocated buffer.

1.1.7. closesocket

1.1.7.1. int closesokcet(int s)
1.1.7.2. Closes the connection conn

1.2. Network connection functions

1.2.1. netconn_new

1.2.1.1. struct netconn * netconn_new(enum netconn type type)
1.2.1.2. 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.

1.2.2. netconn_bind

1.2.2.1. err_t netconn_bind(struct netconn *conn, ip_addr_t *addr, u16_t port)
1.2.2.2. 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.

1.2.3. netconn_listen

1.2.3.1. err_t netconn_ listen(struct netconn *conn)
1.2.3.2. Puts the TCP connection conn into the TCP LISTEN state.

1.2.4. netconn_accept

1.2.4.1. err_t netconn_accept(struct netconn *conn, struct netconn **new_conn)
1.2.4.2. 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.

1.2.5. netconn_recv

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

1.2.5.2. 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.

1.2.6. netconn_write

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

1.2.6.2. 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.

1.2.6.2.1. #define NETCONN_NOCOPY 0x00
#define NETCONN_COPY 0x01

When passed the flag NETCONN_COPY the data is copied into internal buffers 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.

1.2.7. netconn_close

1.2.7.1. err_t netconn_close(struct netconn *conn)

1.2.7.2. Closes the connection conn.

1.2.8. netconn_delete

1.2.8.1. err_t netconn_delete(struct netconn *conn)

1.2.8.2. Deallocates the netconn conn. If the connection is open, it is closed as a result of this call.

1.3. lwip socket library

1.3.1. lwip_socket

1.3.1.1. int lwip_socket(int domain, int type, int protocol)

1.3.1.2. 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.

1.3.2. lwip_bind

1.3.2.1. int lwip_bind(int s, const struct sockaddr *name, socklen_t namelen)

1.3.2.2. 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.

1.3.3. lwip_listen

1.3.3.1. int lwip_listen(int s, int backlog)

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

1.3.4. lwip_accept

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

1.3.4.2. 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.

1.3.5. lwip_recv

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

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

1.3.6. lwip_send

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

1.3.6.2. 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.

1.3.7. lwip_close

1.3.7.1. int lwip_close(int s)

1.3.7.2. Closes the connection conn

2 UART library

2.1. prg_uart_Open

2.1.1. int prg_uart_Open(int port_no, int type);

2.2. prg_uart_Close

2.2.1. int prg_uart_Close(int port_no);

2.3. prg_uart_Config

2.3.1. int prg_uart_Config(int port_no, unsigned long baud, char *serial_mode);

2.4. prg_uart_FlowControl

2.4.1. int prg_uart_FlowControl(int port_no , char *flowmode);

2.5. pgr_uart_RxTimeoutThreshold

2.5.1. int pgr_uart_RxTimeoutThreshold(int port_no, int vtime, int vmin);

2.6. prg_uart_puts

2.6.1. int prg_uart_puts(int port_no , void *buf , size_t size);

2.7. prg_uart_putc

2.8. int prg_uart_putc(int port_no , int c);

2.9. prg_uart_gets

2.9.1. int prg_uart_gets(int port_no, void *buf , size_t size);

2.10. prg_uart_getc

2.10.1. int prg_uart_getc(int port_no);

2.11. prg_uart_Available

2.11.1. int prg_uart_Available(int port_no);

2.12. prg_uart_SendBreak

2.12.1. int prg_uart_SendBreak(int port_no,int duration);

2.13. prg_uart_GetCTS

2.13.1. int prg_uart_GetCTS(int port_no);

2.14. prg_uart_GetDCD

2.14.1. int prg_uart_GetDCD(int port_no);

2.15. prg_uart_SetRTS

2.15.1. int prg_uart_SetRTS(int port_no,int state);

2.16. prg_uart_GetRTS

2.16.1. int prg_uart_GetRTS(int port_no);

2.17. prg_uart_SetDTR

2.17.1. int prg_uart_SetDTR(int port_no,int state);

2.18. prg_uart_GetDTR

2.18.1. int prg_uart_GetDTR(int port_no);

2.19. prg_uart_SetDSR

2.19.1. int prg_uart_SetDSR(int port_no,int state);

2.20. prg_uart_GetDSR

2.20.1. int prg_uart_GetDSR(int port_no);

3 EEPROM library

3.1. prg_eeprom_Open

3.1.1. int prg_eeprom_Open(void)

3.2. prg_eeprom_Close

3.2.1. int prg_eeprom_Close(void)

3.3. prg_eeprom_Erase

3.3.1. int prg_eeprom_Erase(void)

3.4. prg_eeprom_WriteBlock

3.4.1. size_t prg_eeprom_WriteBlock(unsigned long addr, unsigned char *data_buf, unsigned short tot_len)

3.5. prg_eeprom_ReadBlock

3.5.1. size_t prg_eeprom_ReadBlock(unsigned int addr, char *buffer, size_t size)

3.6. get_network_settings

3.6.1. int get_network_settings(struct net_settings *ptr_net_cfg)

3.7. get_uart_settings

3.7.1. int get_uart_settings(int port_no, struct serial_settings *ptr_serial_cfg)

3.8. set_network_settings

3.8.1. int set_network_settings(struct net_settings *ptr_net_cfg)

3.9. set_uart_settings

3.9.1. int set_uart_settings(int port_no, struct serial_settings *ptr_serial_cfg)

Note: set network and uart settings won't be effective until next boot up

4 Web library

4.1. user_web_init

4.1.1. void user_web_init(unsigned short (* user_web)(char *name, char *ajax_buf_ptr, char method, char *post_data, unsigned short post_len))

4.1.2. Configure the callback function while file open

4.1.3. Refer to E05_web&settings and web_ajax.c