Iosoft Ltd.
ChipWeb Wireless Development Kit
Software Manual

1. Introduction

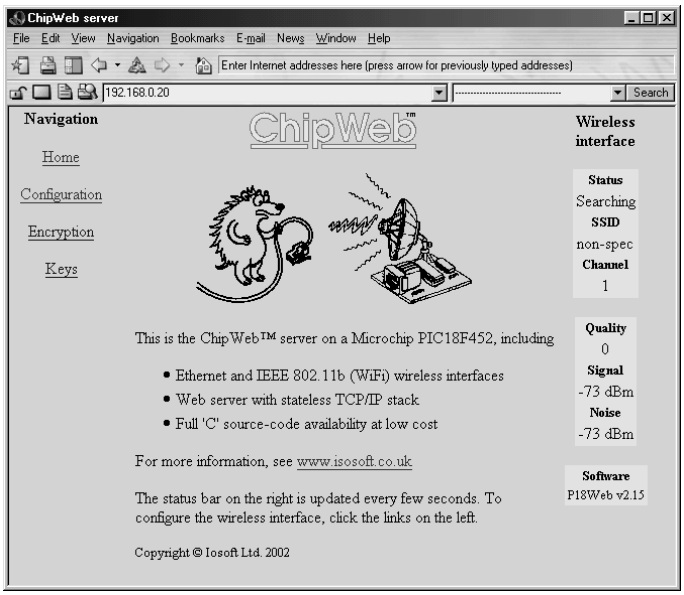
The Iosoft Ltd. *ChipWeb Wireless* kit supports the development of embedded 802.11b hardware and software, with particular reference to Microchip PIC18xxx microcontroller family, and the 'C' programming language. It is supplied in two parts:

- The ER21 development board
- The ChipWeb Wireless source-code package.

This manual describes the source-code package; it is assumed that you already have an ER21 development board operating in a wireless environment, configured as described in the hardware manual.

The basis for the source-code package is the Iosoft ChipWeb TCP/IP software for PICmicro® microcontrollers as described in 'TCP/IP Lean: Web Servers for Embedded Systems' by Jeremy Bentham (2nd edition ISBN 1-57820-108-X). You will also need a copy of the free Intersil 'PRISM Driver Programmer's Manual' AN9900, only available under Non-Disclosure Agreement directly from Intersil (www.intersil.com).

For sales and support information on ChipWeb products, refer to the Iosoft Ltd. Web site, www.iosoft.co.uk



2. Software development

Development environment

To use the source-code package, you will need a compiler for the Microchip PIC18xxx family of microcontrollers. The Iosoft software is broadly compatible with both the CCS (www.ccsinfo.com) and Hi-Tech (www.htsoft.com) compilers, though the initial release of the package has only been tested with the CCS compiler – see the README.TXT file for details.

Installation

The package is provided in a single zip file, which also indicates the version number, e.g. P18WEB_123.ZIP for version 1.23. The new files are only compatible with PIC18xxx-series microcontrollers, so it is recommended that a separate directory be created to house them, e.g. \ChipWeb\P18Web, and the source files unzipped into this. The unzip utility should preserve the original directory structure, so that a sub-directory ROMDOCS is automatically created as well. A README.TXT file has been included with release notes – please read this before using the files.

Software structure

The structure of the software follows closely the existing ChipWeb model, as described in 'TCP/IP Lean' 2nd edition, so this document will concentrate on the changes made to that code to support a wireless interface. These are

- File names and data types
- The wireless device driver
- Network device selector
- Non-volatile configuration
- Web interface

3. File names and data types

File names

It is impossible for the wireless code to run on a PIC16xxx processor, due to the additional memory requirements. For this reason, the new PIC18xxx-only versions of the ChipWeb files have dropped the 'P16' prefix, for example P16WEB.C has now become P18WEB.C.

New files

The following new files have been created:

```
P18_DEFS.C   Function prototypes & definitions
P18_NET.C    Network interface
P18_WLAN.C   Wireless driver
WLAN.H       Wireless definitions
```

All the functions prototypes are now collected in P18_DEFS.C, rather than being scattered throughout the files.

The new software supports multiple network interfaces, with run-time switching between them, so P18_NET.C provides this functionality.

The wireless code is concentrated in P18_WLAN.C, which is structurally similar to the existing Ethernet driver, P18_ETH.C.

Changed files

Most of the files have has minor changes, due to the data type change described below; the most significant changes are to P18_HTTP.C, where major improvements have been made to the dynamic variable substitution code; these are discussed in detail later.

Data types

Early versions of the CCS compiler didn't support 32-bit long integers, so a LWORD structure was defined to allow them to be handled in 16-bit chunks.

```
typedef union
{
    BYTE b[4];
    WORD w[2];
    unsigned INT32 l;
} LWORD_;
```

This is no longer necessary now that the compiler provides full 32-bit support, so a new data type has been defined,

```
#define DWORD unsigned int32
```

The type DWORD is a compiler-independent 32-bit unsigned integer, and allows the use of more consistent handling of function arguments, for example the original software used the following definitions:

```
void put_word(WORD w);
void put_lword(LWORD *lwp);
```

The use of a longword pointer is inconsistent with other put_xxx functions, so the function has been changed to

```
void put_dword(DWORD dw);
```

Unfortunately, the CCS type checking seemingly can't differentiate between the pointer and the longword, so no error messages are generated if the wrong type is used. For this reason, the new definition DWORD has been created, rather than just re-defining LWORD – at least an error message is generated every time the old definition is used.

4. Wireless device driver

Intersil PRISM

The driver currently works with only one wireless chipset, the Intersil PRISM. The earliest version, PRISM 1, is now obsolete so the development has concentrated on the PRISM 2 variant. Later versions (PRISM 2.5 and 3) are starting to appear on the market, and the software should (according to the Intersil documentation) be compatible with these; refer to the software release notes for details.

Due to the restrictions of the Intersil non-disclosure agreement (NDA), no information can be provided on the internal architecture and operation of the PRISM chipset; you will have to sign the Intersil NDA and get hold of this information from them, which comes in the form of an excellent free 250-page document, the 'PRISM Driver Programmer's Manual', part number AN9900.

PCMCIA card interface

The wireless interface is provided in the form of a PCMCIA (PC) card, for which the ER21 board provides an appropriate interface to the PICmicro. Once initialised, the PCMCIA interface is transparent, so the I/O cycles to the PRISM closely resemble those to the Ethernet controller. There is also a memory interface on the card (called *attribute* memory), which gives information about the chipset in standardised (*tuple*) format. A 10-bit address and 8-bit data bus is connected to the PICmicro, together with 4 strobe lines; two to read/write the attribute memory, and two to read/write the I/O devices.

The current software does not use the tuple information, but does show how it can be displayed; after setting the data direction and an even-value address, the Output Enable strobe is asserted, and the data appears after a short delay.

```
#define DATA_TO_NIC   set_tris_d(ALL_OUT)
#define DATA_FROM_NIC set_tris_d(ALL_IN)
DEFBIT_0(PORTA, WC_OE_)
. . .
for (n=0; n<16; n++)  // Dump first 16 CIS bytes for debug
    {
        b = wcfg_rd(n+n);
        printf("%02X ", b);
    }
. . .
/* Read a byte from PC card configuration */
BYTE wcfg_rd(BYTE reg)
{
    BYTE b;

    DATA_FROM_NIC;
    NIC_ADDR = reg;
    WC_OE_ = 0;
    DELAY_ONE_CYCLE;
    DELAY_ONE_CYCLE;
    b = NIC_DATA;
    WC_OE_ = 1;
    return(b);
}
```

Before making any I/O cycles to the chipset, it is necessary to enable it via a write cycle to the PCMCIA *configuration option register*, which is usually at address 3E0 hex.

```
    WC_A8 = 1;         // Set hi address bits
    WC_A9 = 1;
    wcfg_wr(0xe0, 1); // Write Config Option Reg at 3E0h
    WC_A8 = 0;         // (to enable I/O mode)
    WC_A9 = 0;
```

Once the PCMCIA interface is initialised, an input cycle from the PRISM requires only that the data direction and address is set, and the 'I/O read' line asserted; the data appears on an 8-bit data bus after a short delay.

```
DEFBIT_2(PORTA, WC_IOR_)
. . .
/* Input a byte from a WLAN Controller register */
BYTE wc_in(BYTE reg)
{
    BYTE b;

    DATA_FROM_NIC;
    NIC_ADDR = reg;
    WC_IOR_ = 0;
    DELAY_ONE_CYCLE;
    DELAY_ONE_CYCLE;
    b = NIC_DATA;
    WC_IOR_ = 1;
    DELAY_ONE_CYCLE;
    DATA_TO_NIC;         // Drive data bus high
    DELAY_ONE_CYCLE;
    NIC_DATA = 0xff;
    DATA_FROM_NIC;
    return(b);
}
```

As a precautionary measure, the data bus is driven high after the cycle, rather than being left to float – some CMOS devices can oscillate when their inputs float around half their supply voltage.

All cycles to the PRISM must be in 16-bit words, so two byte-wide cycles are aggregated:

```
/* Input a word from a WLAN Controller register */
WORD wc_inw(BYTE reg)
{
    BYTE hi, lo;

    lo = wc_in(reg);
    hi = wc_in(reg+1);
    return(((WORD)hi<<8) | (WORD)lo);
}
```

For more information on PCMCIA, see 'PCMCIA System Architecture' by Don Anderson, ISBN 0-201-40991-7.

5. Network device selector

Previous versions of the ChipWeb software could only drive one network device at a time; switching between network interfaces involved re-compilation. The new version has abstracted all the device-independent network code into a new file P18_NET.C, and has the facility to dynamically switch between devices. This allows the same Web pages to appear simultaneously on both Ethernet and wireless networks.

The switch-over is achieved by a function select_device()

```
// Net device definitions
#define DEVICE_ETH  0   // Ethernet device
#define DEVICE_WLAN 1   // Wireless LAN device
#define NUM_DEVICES 2   // Total number of network devices
BYTE my_mac[MACLEN];    // My MAC addr (Ether or WLAN)
. . .
/* Select the Tx and Rx network device */
void select_device(BYTE device)
{
    if (device < NUM_DEVICES)
    {
        rx_device = device;
        tx_device = device;
        if (device == DEVICE_WLAN)
            memcpy(my_mac, wlan_mac, MACLEN);
        else
            memcpy(my_mac, myeth, MACLEN);
    }
}
```

It is assumed that switch-over occurs before the device is polled for incoming traffic, so there isn't any partially-analysed data to be saved. It is important that any high-level function requiring a local MAC address refers to the new 'my_mac' variable, rather than the old Ethernet-specific 'myeth'.

On each pass of the main polling loop, the 'next' device is selected, using the following function:

```
/* Select the next Rx device */
void next_device(void)
{
    BYTE device;

    device = DEVICE_ETH;
#if INCLUDE_ETH && INCLUDE_WLAN
    device = rx_device + 1;
    if (device >= NUM_DEVICES)
        device = 0;
#else
#if INCLUDE_WLAN
    device = DEVICE_WLAN;
#endif
#endif
    select_device(device);
}
```

6. Non-volatile configuration

In contrast to the Ethernet controller, the wireless controller requires a significant amount of configuration information that has to be stored in non-volatile memory. A structure is used to hold this

```
#define SSID_LEN        32    // Max length of SSID
#define WEP128_KEYLEN   13    // Length of 128-bit WEP key
. . .
typedef struct {
    BYTE type;
    BYTE chan;                  // Channel number (1-13)
    char ssid[SSID_LEN+2];      // Desired network name;
null=any
    BYTE weptype;               // WEP enable
    BYTE authtype;              // Authentication enable
    BYTE defkey;                // Default key number (0-3)
    BYTE keylen1;               // Length of key
    BYTE key1[WEP128_KEYLEN];   // Only 1 key in this
release
} WLAN_CFG;
WLAN_CFG wlancfg;
```

The structure is saved to the non-volatile EEPROM memory on a PIC18F452 using the existing write_eeprom() function:

```
#define WLANCFG_ADDR    0x10    // Address of WLAN config
. . .
/* Save the WLAN configuration data in EEPROM */
BOOL save_wlancfg(void)
{
    BYTE *bp, n;

    bp = (BYTE *)&wlancfg;
    for (n=0; n<sizeof(WLAN_CFG); n++)
        write_eeprom(WLANCFG_ADDR+n, *bp++);
    return(1);
}
```

The corresponding load_wlancfg() function performs a simple sanity check on the data, and loads default values if an error is detected.

7. Web interface

Dynamic variables

The Web interface is called upon to display a wide variety of status and configuration information, and the existing EGI variable-substitution technique proved too cumbersome, so a new system based on long variable names has been developed.

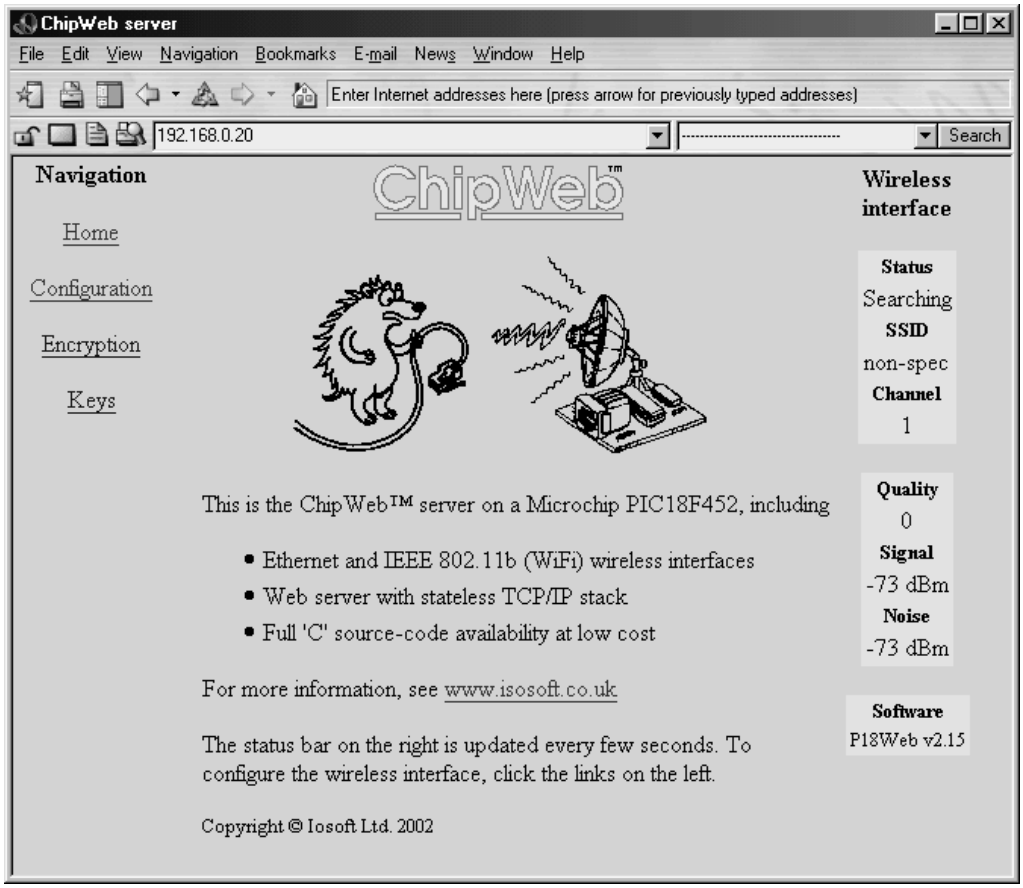


Fig. 1: Main Web frameset

Figure 1 shows the main frameset, which is similar to that used on previous ChipWeb projects; the left-hand frame is used for navigation, and the right-hand frame to display status information that is updated every few seconds.

The following HTML code fragment from the status frame shows the new long-variable-name scheme in action

```
<table bgcolor=#e0e0ff vspace=0>
<tr><th align=center
colspan=2><small>Quality</small></th></tr>
<tr><td align=center>$quality</td></tr>
<tr><th align=center
colspan=1><small>Signal</small></th></tr>
<tr><td align=center>$signal dBm</td></tr>
<tr><th align=center
colspan=1><small>Noise</small></th></tr>
<tr><td align=center>$noise dBm</td></tr></table>
```

The numeric quality, signal & noise values are represented by the dynamic variables `$quality`, `$signal` and `$noise` (bold in the text above). When the pages are fetched from EEPROM, the correct numeric values are substituted before the page is sent out of the network interface, so `$signal` becomes, say, `-73`.

The substitution is not limited to numeric values; the variable `$version` becomes `P18Web v2.15`, so that the current software version number is indicated on the page.

The software to perform the substitution is in P18_HTTP.C, and is based on a lookup table, which contains the variable names (minus the dollar character), and the corresponding index values:

```
BYTE const egivars[] =
    "version configup xchan xssid ess ibss wstatus ssid chan "
    "quality signal noise defkey authen keylen";
#define EGIVAR_VERSION  1   // Index number of each variable
. . .
#define EGIVAR_QUALITY  10
#define EGIVAR_SIGNAL   11
#define EGIVAR_NOISE    12
. . .
```

In the main software loop that outputs the HTML pages, there is code that detects a variable, translates that variable to an index number, and then takes appropriate action:

```
if ((idx = match_egivar()) > 0)
{
    switch (idx)
    {
    case EGIVAR_VERSION:   // Software version?
        putstr(SIGNON);
        break;
    . . .
    case EGIVAR_QUALITY:    // Quality value
        PRINTF2("%u", (BYTE)wlan_qsn[0]);
        break;
    case EGIVAR_SIGNAL:     // Signal value
        PRINTF2("-%u", (BYTE)(-wlan_qsn[1]));
        break;
    case EGIVAR_NOISE:      // Noise value
        PRINTF2("-%u", (BYTE)(-wlan_qsn[2]));
        break;
    . . .
```

The resulting run-time substitution is very versatile; for example, it is necessary to display a message when the user attempts to update the non-volatile configuration, in case there was an error:

```
    case EGIVAR_CONFIGUP:   // Config update OK?
        if (form_err == FORMERR_CANCEL)
            putstr("cancelled: not updated");
        else if (form_err == FORMERR_CHAN)
            putstr("error: invalid channel");
        else if (form_err == FORMERR_SSID)
```

```
        putstr("error: invalid SSID");
. . .
else
        putstr("updated OK");
form_err = 0;
break;
```

The variable `$configup` can be embedded in a simple HTML page to report the status after an update.

```
<HTML><HEAD><TITLE>Wireless configuration update</TITLE></HEAD>
<BODY bgcolor=#d0d0ff><FONT face=helvetica>
<H4>Wireless configuration $configup</H4>
<FORM action=wlan1.egi>
   <INPUT TYPE="submit" NAME="ok" VALUE="OK">
</FORM></BODY></HTML>
```

This produces the appropriate acknowledgement depending on whether the update was successful or not.

Form variables

The increasing complexity of the Web pages also results in more complex HTML forms responses that have to be parsed, such as shown in figure 2.

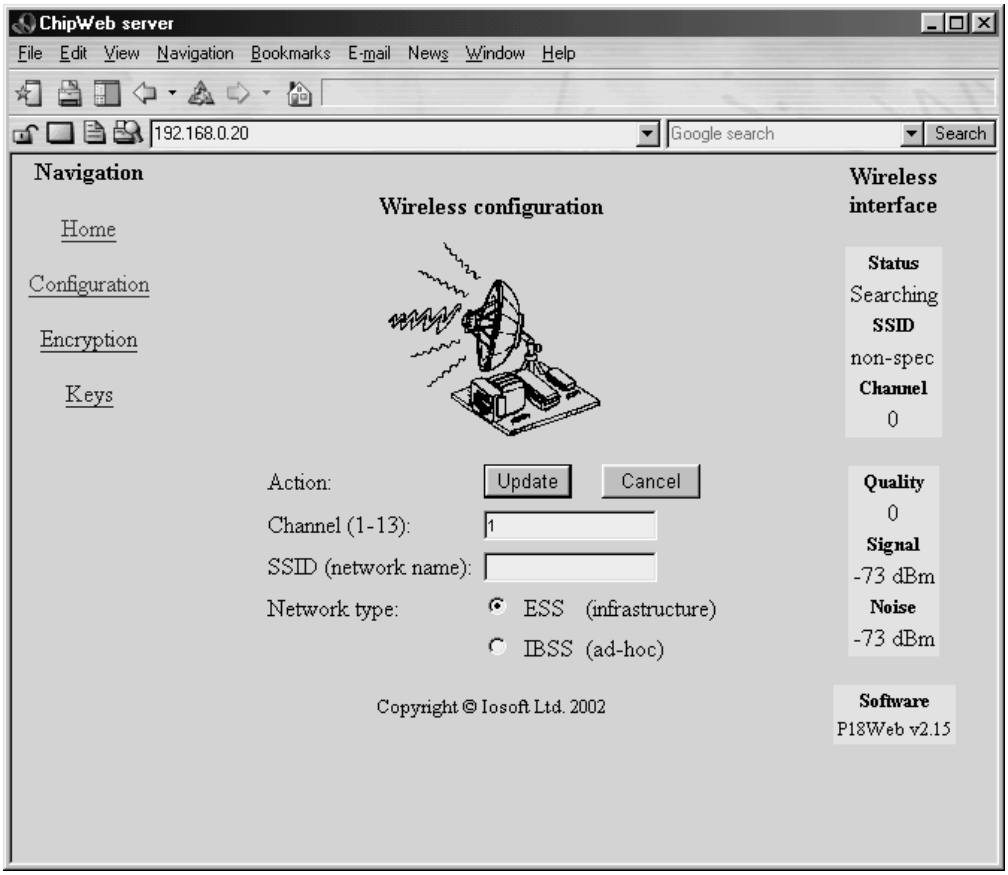


Figure 2: Wireless configuration form

If the user selects ESS network channel 1 with an SSID of 'Wireless', the browser sees the following response:

```
GET wlan1a.egi?submit=Update&chan=1&ssid=Wireless&type=ess
```

It is necessary to walk through the URL-encoded list of form variables and their values, in order to establish what the user has entered. A new helper function has been provided to simplify this process, and it too relies on a table of variable names and index values:

```
BYTE const formvars[] =
    "cancel chan ssid type defkey authen keynum keystr keyhex";
#define FORMVAR_CANCEL  1   // Index number of each variable
#define FORMVAR_CHAN    2
#define FORMVAR_SSID    3
#define FORMVAR_TYPE    4
. . .
```

The function check_formargs() processes the browser response, and takes appropriate action:

```
if (b=='=' && (idx=match_formvar())>0)
{
    switch (idx)
    {
    case FORMVAR_CANCEL:    // Cancel button
        form_err = FORMERR_CANCEL;
        break;
    case FORMVAR_CHAN:      // Channel number 1-13
        if (!get_num(&w) || w<MIN_WLAN_CHAN || w>MAX_WLAN_CHAN)
            form_err = FORMERR_CHAN;
        else
        {
            wlancfg.chan = (BYTE)w;
            updat++;
        }
        break;
    case FORMVAR_SSID:      // SSID string
        get_formval_str(wlancfg.ssid, SSID_LEN);
        updat++;
        break;
    . . .
```

The order in which the variables are processed matches the order they appear on the form. This is useful as it is necessary to perform an up-front check that the user hasn't pressed the 'cancel' button; if so, an error flag is set and the remaining form variables aren't processed.

Web pages

The pages and graphics are stored in the sub-directory ROMDOCS; HTML pages should have a .HTM extension, or .EGI if they include dynamic variables, as described in chapter 11 of 'TCP/IP Lean'. In this implementation, any one page or graphic is limited to approximately 1440 bytes in size.

To create a ROM image of the files, the WEBROM utility from the 'TCP/IP Lean' CD-ROM is required. It takes an arbitrary number of files from a single directory, adds the HTTP headers, and stores them as a single binary image, e.g.

```
CD \CHIPWEB\P18WEB
\TCPLEAN\WEBROM webpage.rom romdocs
```

This creates the file WEBPAGE.ROM using all the files in the sub-directory ROMDOCS. The size of each file is displayed, and the total image size, and it is worth checking that these do not exceed the limits for any one file, or the total EEPROM size (32 Kbytes).

To load the files into the serial EEPROM device on the ER21 a suitable device programmer may be used, or the image may be uploaded to the board using XMODEM over a serial link, as described in the ER21 hardware manual.

JPB 23/10/02

--ends--