DWIFIcK chipKIT™ WiFi Library

Revision: May 24, 2012
Note: This document applies to the DWIFIcK Library.

www.digilentinc.com
1300 NE Henley Court, Suite 3
Pullman, WA 99163
(509) 334 6306 Voice | (509) 334 6300 Fax

Introduction

This document describes the Digilent® chipKIT™ compatible WiFi Internet library. The focus of this library is to enable WiFi support for the DNETcK internet library. DWIFIcK is designed to be used on chipKIT™ compatible boards using the MPIDE development platform and the Microchip MRF24WB0MA WiFi transceiver.

The model for this library is tailored to the Initialize/Loop type program structure where system tasks are typically executed once per each pass through the loop. As a result, all DWIFIcK methods are designed to return control to the loop quickly so other, potentially more critical, system tasks can be serviced. Also the methods are designed not to allocate memory from the heap. This will eliminate any chance that the DWIFcK library will consume, leak, or fragment the heap to the point of failure.

The presence of the DWIFIcK library included in a sketch modifies the DNETcK library to use the MRF24WB0MA WiFi transceiver instead of using a hardwired Ethernet NIC. Make sure to include DWIFIcK.h after you include DNETcK.h in your sketch. Before you can use any of the DNETcK methods a WiFi connection to a WiFi network must first be established. DWIFIcK provides methods to enumerate and connect to a WiFi network. While not strictly required, it is generally best to scan and connect to the WiFi network before calling the DNETcK::begin() method. Likewise, while not strictly required, it is best to call DNETcK::end() before disconnecting from the WiFi network. It is permissible to call DNETcK::begin() and DNETcK::end() repeatedly without disconnecting from the WiFi network. Sometimes it is useful to first call DNETcK::begin() using DHCP to get your network addresses from the gateway, and then call DNETcK::end() and DNETcK::begin() a second time specifying a static IP, but now knowing all of your other network parameters. While doing this, it is not necessary to call DWIFIcK:disconnect() and DWIFIcK::connect() to restart DNETcK using the static IP.

DWIFIcK is designed to do passive scans of all SSIDs and channels currently being broadcasted by APs / Wireless Routers (from now called APs) within range. In order for DWIFIcK to see a WiFi network, the AP must be doing active broadcasts, and must also be actively broadcasting the SSID. At this time DWIFIcK does not support querying APs for network information. Since the MRF24WB0MA operates in the 1&2Mbps range, it is required that your AP supports a basic rate of 1&2Mbps. This is usually the default for most APs, but not always. Some APs have high speed proprietary capabilities and these APs may not be seen by the MRF24WB0MA. If this is the case, turn off all proprietary features in the AP, select a basic rate of 1&2Mbps, or try selecting Wireless-B as one of the supported networks. Also make sure that your AP is broadcasting the SSID. A quick and easy way to determine what is being seen by the MRF24WB0MA is to run the WiFiScan example sketch; this sketch will enumerate all of the WiFi networks that can be seen. However, before a connection can be made, make sure that the SSID name is visible, that is, not blank in the WiFiScan printout.

DWIFIcK supports WPA2, WPA-PSK, WEP104, WEP40 and open networks. However, there are some difficulties with WPA2 and WPA. WPA was created as a stop gap solution to partially fix a security hole discovered in WEP. However, the ultimate fix to the security hole required, in most cases, new hardware. Once the hardware became available WPA2 was implemented. In reality, WPA is closer to WEP than WPA2, however from the user's perspective they look very similar using a SSID and passphrase to connect. Many APs support a mixed WPA/WPA2 mode as does the MRF24WB0MA. Unfortunately, there is no standard of negotiation to determine which security mode to pick should both sides be in mixed mode. Typically WPA2 is selected however this is not always the case and causes some connection failures. To reduce the number of connection failures, DWIFIcK::connect(const char * szSsid, const char * szPassPhrase) does not use mixed mode by default but specifically specifies WPA2. If you wish to use WPA, you must call the long form of DWIFIcK::connect() and explicitly specify a security of DWIFIcK ::WF_SECURITY_WPA_WITH_PASS_PHRASE.

WPA/WPA2 passphrases must be converted to a PSK key (Pre-Shared Key) before use. DWIFIcK will automatically pass to the MRF24WB0MA the passphrase where the MRF24WB0MA will apply the PBKDF2 key derivation function to generate the key. However this is a very slow and involved process that takes about 30 seconds to calculate. What this looks like to the sketch is that the WiFi connection is waiting to connect for about 30 seconds. However, if you pre-calculate the key first you can eliminate this 30 second delay. You can find a tool to pre-calculate your PSK key at http://www.wireshark.org/tools/wpa-psk.html[1]. Once calculated you can use the DWIFIcK::connect(const char * szSsid, WPA2KEY& key) method to connect to the network without the 30 second key calculation delay.

If the WiFi network goes down for a short period of time, the underlying code will attempt to automatically reconnect. If the WiFi network is attempting a reconnect, the non-fatal DNETcK:: WaitingReConnect status will be returned by the isConnected() and isListening() methods. If the reconnection is successful, they will automatically start returning DNETcK:: Connected or DNETcK::Listening respectively.

Currently DWIFIcK only supports 2 connection IDs, 0 and 1. Zero is typically used to indicate an invalid or unspecified connection and one is the only currently supported connection ID. However, just because you have a connection ID does not mean you are actively connected to a WiFi network. A connection ID is really just a collection of data that specifies the WiFi's connection parameters. The DWIFIcK methods are designed to allow for future support of multiple connection IDs where connection parameters can be store in eeprom, each representing a different WiFi network. The connection ID returned from DWIFIcK::connect() is the actively connected WiFi connection and the sketch should maintain which connection ID is active. In this release of DWIFIcK, if you call DWIFIcK::connect() with the invalid connection ID (0) this will tell DWIFIcK::connect() to use the connection information specified in WF_Config.x in your sketch directory and then return to you the active connection ID (in this release always 1). If you do not specify a WF_Config.x in your sketch directory, the connection information provided by the default WF_Config.x file found in …/libraries/DWIFIcK/WF_Config.x is used; and will typically not work for your network. Note in this release you may not call DWIFIcK::connect() with a connection ID of 1; that is reserved for potential future support.[2]

The WiFi stack requires the MAC stack to be running and will automatically initialize the MAC on any DWIFIcK call. If you need to specify your MAC address, you must do so by calling DNETcK::setMyMac() before calling any DWIFIcK method or DNETcK::begin(). If DNETcK::setMyMac() is not called, the hardware assigned MAC is used. Once a MAC address is assigned, or the stack is initialized, the MAC cannot be changed. Typically you would just use the hardware assigned MAC and not use DNETcK::setMyMac().

[1] Digilent makes no warranty for this site. This site is not affiliated with Digilent in any way. This site is provided for user convenience only; use at your own risk.
[2] Connection IDs are partially supported by the MRF24WB0MA and is why this concept is exposed. While the concept is not fully specified there potentially will be 3 types of connection IDs, 0: invalid, 1-X: transient (non-persistent) and X-Y: persistent (stored somewhere like eeprom).

Network Hardware

At this time the only WiFi hardware that is supported is the MRF24WB0MA. To specify this hardware the WiFiShieldOrPmodWiFi hardware library must be included in your sketch before both DNETcK.h and DWIFIcK.h library header files. There are no chipKIT™ compatible boards that have preinstalled WiFi hardware on them, so no default hardware exists for WiFi.

DWIFIcK Static Class

DWIFIcK is a static class that provides methods to scan and connect to a WiFi network. Connecting to a WiFi network should be done before calling DNETcK::begin(); and DNETcK::end() should be called before disconnecting from the WiFi network. Once the WiFi is connected, DNETcK should be fully functional.

Constants

int DWIFIcK:: INVALID_CONNECTION_ID

Has a value of 0 and specifies an invalid or unspecified connection.

int DWIFIcK:: WF_MAX_SSID_LENGTH

Has a value of 32 and is the maximum length of an SSID.

int DWIFIcK:: WF_BSSID_LENGTH

Has a value of 6 and represents the MAC of the SSID WiFi network.

int DWIFIcK:: WF_MAX_NUM_RATES

Has a value of 8 and is the maximum number of supported speeds (Mbps) that an AP can report to the MRF24WB0MA.

int DWIFIcK:: WF_SCAN_ALL

Has a value of 0xFF and specifies that all broadcasting WF networks should be scanned.

Enums

```
typedef enum
 {
    WF_ACTIVE_SCAN      = 1,
    WF_PASSIVE_SCAN     = 2,
 } WFSCAN;
```

When requesting a scan of available WiFi networks either a passive or active scan can be done. By default passive scans are done as most APs broadcast their information. While you may specify an active scan for the scan, active scanning is not supported when attempting a connect in this release of DWIFIcK.

```
typedef enum
 {
    WF_SECURITY_OPEN                                = 0,
    WF_SECURITY_WEP_40                              = 1,
    WF_SECURITY_WEP_104                             = 2,
    WF_SECURITY_WPA_WITH_KEY                        = 3,
    WF_SECURITY_WPA_WITH_PASS_PHRASE                = 4,
    WF_SECURITY_WPA2_WITH_KEY                       = 5,
    WF_SECURITY_WPA2_WITH_PASS_PHRASE               = 6,
    WF_SECURITY_WPA_AUTO_WITH_KEY                   = 7,
    WF_SECURITY_WPA_AUTO_WITH_PASS_PHRASE           = 8
 } SECURITY;
```

This represents the type of security.

Structures

```
typedef struct
{
   byte                    rgbKey[32];
} WPA2KEY;
```

The WPA2KEY data structure represents a 32 byte PSK key for both WPA and WPA2.

```
typedef struct
{
   union
   {
      byte                 rgbKey[5];
      char                 asciiKey[5];
   } key[4];
} WEP40KEY;
```

The WEP40KEY data type represents a WEP40 keyset. There is a possibility of 4 indexed keys, 0 – 3. Only the key index that is to be used must be filled in.

```
typedef struct
{
   union
   {
      byte                 rgbKey[13];
      char a               sciiKey[13];
   } key[4];
} WEP104KEY;
```

The WEP104KEY data type represents a WEP104 keyset. There is a possibility of 4 indexed keys, 0 – 3. Only the key index that is to be used must be filled in.

```
typedef struct
{
   char                    szSsid[WF_MAX_SSID_LENGTH];
   SECURITY                securityType;
   byte                    channel;
   byte                    signalStrength;
   byte                    cBasicRates;
   byte                    dtimPeriod;
   unsigned short          atimWindow;
   unsigned short          beconPeriod;
   unsigned int            basicRates[WF_MAX_NUM_RATES];
   byte                    ssidMAC[WF_BSSID_LENGTH];
} SCANINFO;
```

The SCANINFO data type represents the information transmitted by a specific AP describing the AP's capabilities.

```
   typedef struct
   {
      SECURITY                 securityType;
      struct
      {
         Int                    Index;
         int                   cbKey;
         union
         {
            WPA2KEY            wpa;
            WEP40KEY           wep40;
            WEP104KEY          wep104;
            char               szPassPhrase[WF_MAX_PASS_PHRASE];
            byte               rgbKey[1];
         };
      } key;
   } SECINFO;
```

The SECINFO data type represents the security information needed to connect to an AP. If a passphrase is specified, the SECINFO will contain the passphrase until the PSK is calculated. Once the PSK key is calculated the WPA2KEY if filled in with the calculated key.

```
typedef struct
{
   unsigned int        pollingInterval;
   unsigned int        minChannelTime;
   unsigned int        maxChannelTime;
   unsigned int        probeDelay;
   WFSCAN              scanType;
   byte                minSignalStrength;
   byte                connectRetryCount;
   byte                beaconTimeout;
   byte                scanCount;
} CONFIGINFO;
```

The CONFIGINFO data type represents the current configuration parameters used by the MRF24WB0MA.

Methods

int connect(int connectionID);
int connect(int connectionID, DNETcK::STATUS * pStatus);
int connect(const char * szSsid);
int connect(const char * szSsid, DNETcK::STATUS * pStatus);
int connect(const char * szSsid, WEP40KEY& keySet, int iKey);
int connect(const char * szSsid, WEP40KEY& keySet, int iKey, DNETcK::STATUS * pStatus);
int connect(const char * szSsid, WEP104KEY& keySet, int iKey);
int connect(const char * szSsid, WEP104KEY& keySet, int iKey, DNETcK::STATUS * pStatus);
int connect(const char * szSsid, const char * szPassPhrase);
int connect(const char * szSsid, const char * szPassPhrase, DNETcK::STATUS * pStatus);
int connect(const char * szSsid, WPA2KEY& key);
int connect(const char * szSsid, WPA2KEY& key, DNETcK::STATUS * pStatus);
int connect(SECURITY security, const char * szSsid, const char * szPassPhrase);
int connect(SECURITY security, const char * szSsid, const char * szPassPhrase, DNETcK::STATUS * pStatus);
int connect(SECURITY security, const char * szSsid, const byte * rgbKey, int cbKey, int iKey);
int connect(SECURITY security, const char * szSsid, const byte * rgbKey, int cbKey, int iKey, DNETcK::STATUS * pStatus);

Parameters:

szSsid	Is a zero terminated string of the SSID of the wireless network to connect to.
szPassPhrase	Is a zero terminated string of the WPA or WPA2 passphrase to use.
connectionID	Currently only a connection ID of zero is supported and instructs connect() to read the AP connection parameters from WF_Config.x.
keySet	Either a WEP40KEY or WEP104KEY keyset. There can be up to 4 keys, but only the key specified by iKey must be filled in.
iKey	The index of the key to use out of the WEP keyset, valid values are from 0 - 3. Only the key that is specified by the iKey needs to be filled in. If WEP is not being used this parameter is ignored.
pStatus	An optional parameter to receive the status of the connect.

Returns:

The connection ID that the MRF24WB0MA is connecting to. In this release, on success the connection ID will always be 1. If the connection process could not be started then DWIFIcK:: INVALID_CONNECTION_ID is returned.

Just because a connection ID is returned does not mean you are connected, call DWIFIcK::isConnected() to see if you are connected. Connecting can take a long time, up to 30 seconds if a passphrase is used. Keep calling DWIFIcK::isConnected() until it passes, or until the returned status is a hard failure as returned by DNETcK:: isStatusAnError(). Optionally, if you know you are going to connect, you can call DNETcK:begin() and then call DNETcK::isInitialized(), TcpClient.isConnected(), or Tcp/UdpServer.isListening() until they pass or a hard failure occurs. All of these methods will return “false” with a non-fatal status until the WiFi connection is made.

bool isConnected(int connectionID);
bool isConnected(int connectionID, unsigned long msBlockMax);
bool isConnected(int connectionID, DNETcK::STATUS * pStatus);
bool isConnected(int connectionID, unsigned long msBlockMax, DNETcK::STATUS * pStatus);

Parameters:

connectionID	This must be the connection ID as returned from DWIFIcK::connect().
msBlockMax	The maximum number of milliseconds to wait for the task to complete before returning. If omitted the default block time as specified by DNETcK::setDefaultBlockTime() is used.
pStatus	An optional parameter to receive the current state of the connection.

Returns:

True if the WiFi connection is made, false if it is still pending or a hard error has occurred.

To determine if a hard error has occurred, check the status with DNETcK:: isStatusAnError(). If the connection has failed, call DWIFIcK::disconnect() to free the active connection ID.

void disconnect(int connectionID);

Parameters:

connectionID	This must be the connection ID as returned from DWIFIcK::connect().

Returns:

None

This terminates the WiFi connection. If DNETcK::begin() was called, DNETcK::end() must be call before DNETcK::begin() can be called again. While not strictly required, it is highly recommended that DNETcK::end() be called before DWIFIcK:disconnect().

This frees the connection ID as the actively connected connection. If the WiFi connection fails to connect, or if the connection is dropped, DWIFIcK::disconnect() must be called to free the connection ID.

bool getSecurityInfo(int connectionID, SECINFO * psecInfo);

Parameters:

connectionID	If zero is passed, the security info for the actively connected connection is returned. Otherwise this is the "stored" security info for the connection ID. In this release, only connection IDs of 0 and 1 are supported.
psecInfo	This is a pointer to a SECINFO data structure to receive the security information.

Returns:

True if the security information was returned, false if the security information was not returned, the SECINFO data will be indeterminate on failure.

If DWIFIcK:: getSecurityInfo() is called before the connection is actually made (as determined by DWIFIcK:isConnected()), then the security information will reflect the information stored for that connection ID. After the WiFi connection is made, all parameters will be updated to those used in the connection. In particular, if a passphrase was used, the PSK key will be returned.

In this release stored connection IDs are not supported and this call can only be made for the active connection. Because it is possible to interrupt the MRF24WB0MA from calculating a key, this call should not be made while DWIFIcK::connect() is in the process of connecting to a WiFi network. However, DWIFIcK:: getSecurityInfo() may be called after the connection is successfully made to retrieve the generated PSK, or after DWIFIcK::connect() fails (but before DWIFIcK::disconnect() is called) to query the security info used.

bool beginScan(void);
bool beginScan(int connectionID);
bool beginScan(WFSCAN scanType);
bool beginScan(int connectionID, WFSCAN scanType);

Parameters:

connectionID	Methods taking this parameter are defined for future use. Currently only DWIFIcK:: WF_SCAN_ALL is supported; and this is the default if this parameter is omitted.
scanType	Specifies if you want to actively probe the AP for its SSID/BSSID or passively listen to for it on its next broadcast. If this parameter is omitted, the scan type will be passive and is the most common and reliable way to get the APs SSID.

Returns:

True if the scan was successfully started, false otherwise.

For most applications use beginScan(void) as this is usually what is desired. Active probing is typically for more advanced usage. Scanning can only be done before a WiFi connection is made. Once DWIFIcK::connect() is called, scanning will be disabled until DWIFIcK:disconnect() is called.

bool isScanDone(int * pcNetworks);
bool isScanDone(int * pcNetworks, unsigned long msBlockMax);
bool isScanDone(int * pcNetworks, DNETcK::STATUS * pStatus);

bool isScanDone(int * pcNetworks, unsigned long msBlockMax, DNETcK::STATUS * pStatus);

Parameters:

pcNetworks	A pointer to an integer to receive the number of networks detected, this value will only have meaning if true was returned from isScanDone().
msBlockMax	The maximum number of milliseconds to wait for the task to complete before returning. If omitted the default block time as specified by setDefaultBlockTime() is used.
pStatus	An optional parameter to receive the current state of the scan.

Returns:

True if the scan is done, false if it is still scanning or if there was a hard error. Call DNETcK:: isStatusAnError() to determine if a hard error occurred.

bool getScanInfo(int iNetwork, SCANINFO * pscanInfo);

Parameters:

iNetwork	A zero based index that must be less than cNetwork returned by isScanDone(). This allows enumeration to retrieve information for each network scanned.
pscanInfo	A pointer to a SCANINFO structure to receive the network information.

Returns:

True if the requested scan information was returned, false if not. The contents of the scanInfo structure will be indeterminate if false is returned. The most common cause of a failure is that iNetwork was out of bounds. This call can only be made immediately after isScanDone() or a preceeding getScanInfo().

bool getConfigInfo(CONFIGINFO * pConfigInfo);

Parameters:

pConfigInfo	A pointer to a CONFIGINFO structure to receive the MRF24WB0MA WiFi configuration.

Returns:

True if the configuration was returned, false otherwise..

These are the global configuration parameters applied to the MRF24WB0MA. In this release common defaults are automatically applied. In future releases some parameters may be programmatically set. For now, this call is provided for informational purposes only.

Advanced Override Features

It is possible to override the MRF24WB0MA SPI interface, TCPIP socket configuration, or WiFi security parameters in your sketch. Overriding the SPI interface for the MRF24WB0MA and modifying the TCPIP socket configuration is exactly like overriding the SPI interface for an Ethernet NIC and TCPIP configuration as described in the DNETcK documentation under “Advanced Override Features”. WiFi only introduces the additional WF_Config.x security configuration file.

WF_Config.x

WF_Config.x contains your default WiFi security mode and keys. You can find this file at “…/libraries/DWIFIcK/WF_Config.x” and you can copy it to your sketch directory to override the default settings. The file is well commented and you should be able to easily select the security mode you want and provide any keys or passphrases you need for your particular network. In your sketch, you should call DWIFIcK::connect(0) to indicate that you want to read the security information from WF_Config.x. As an example you can look at the WiFiConfigOverride example sketch.