# Steps required for upgrade from TCP/IP stack v5 to v6.

**Draft**

Information contained in this publication regarding device applications and the like is provided only for your convenience and may be superseded by updates. It is your responsibility to ensure that your application meets with your specifications.

MICROCHIP MAKES NO REPRESENTATIONS OR WARRANTIES OF ANY KIND WHETHER EXPRESS OR IMPLIED, WRITTEN OR ORAL, STATUTORY OR

OTHERWISE, RELATED TO THE INFORMATION, INCLUDING BUT NOT LIMITED TO ITS CONDITION, QUALITY, PERFORMANCE, MERCHANTABILITY OR FITNESS FOR PURPOSE**.**

Microchip disclaims all liability arising from this information and its use. Use of Microchip devices in life support and/or safety applications is entirely at the buyer’s risk, and the buyer agrees to defend, indemnify and hold harmless Microchip from any and all damages, claims, suits, or expenses resulting from such use. No licenses are conveyed, implicitly or otherwise, under any Microchip intellectual property rights.

## TCP/IP Stack v6 distribution

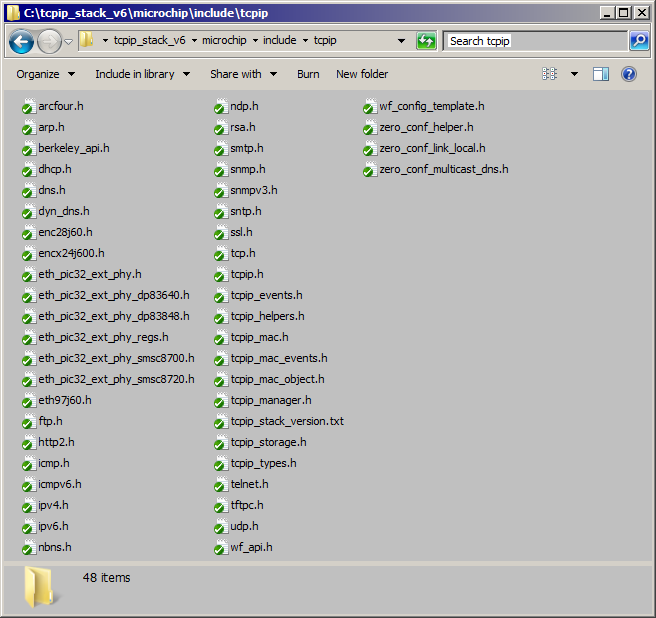
The TCP/IP stack v6 is a completely new distribution. No files need to be maintained from the old v5 installation. It is preferable to install the v6 stack into a new folder.

## TCP/IP Stack v6 key features

* **Multiple interfaces**
* **IPv6 support**
* **Fully dynamic:**
  + stack initialization/de-initialization
  + interface up/down
  + resource management
  + module configuration
* **Improved modularity and stack layout**
* **Run-time configuration - TCP/IP console**
* **Interrupt driven operation**
* **RTOS friendly - easy RTOS integration**
* **PIC18 – no longer supported**

## TCP/IP Stack v6 structure

* **Lower case file names**
* **Underscore is used**
  + tcpip\_manager.h
* **Private headers no longer exposed**
  + Moved to the source folder
* **System services removed from the stack**
  + Interrupts, drivers, timer services, FS, etc
  + BSP



## 

## TCP/IP Stack v6 API changes

For most of the modules the API should be backward compatible to the v5 TCP/IP stack. The changes in the stack API have been minimized so that the porting process is a straightforward one. However, new functionality has been added because of new features like IPv6 support, multiple network interfaces and dynamic configuration of the stack. Other than that the only changes to the API were done when the existing v5 functions did not support the required parameters/flexibility or they were confusing.

The API changes are at TCP, UDP, ARP, HTTP, SMTP and stack access and initialization level:

* TCP changes (“microchip/include/tcpip/tcp.h”).

**There is no more DNS resolution done automatically when opening a TCP socket**. **The TCP layer takes only IP addresses as an input parameter.** If you need a DNS resolution you have to perform it before opening a socket (see “microchip/include/tcpip/dns.h”).

**Support for IPV6 added.**

To open TCP sockets, the required functions are now:

* + **TCPOpenServer**(IP\_ADDRESS\_TYPE addType, TCP\_PORT localPort, IP\_MULTI\_ADDRESS\* localAddress);
  + **TCPOpenClient**(IP\_ADDRESS\_TYPE addType, TCP\_PORT remotePort, IP\_MULTI\_ADDRESS\* remoteAddress);

These new functions replaced the old TCPOpen.

**The application code has to replace the calls to TCPOpen with TCPOpenClient or TCPOpenServer as appropriate.**

The new calls add the possibility to use IPv6 addresses and make clear choice of parameters for server or client sockets.

The parameters:

* typedef union

{

IPV4\_ADDR v4Add;

IPV6\_ADDR v6Add;

}IP\_MULTI\_ADDRESS;

* typedef enum

{

IP\_ADDRESS\_TYPE\_ANY = 0, // either IPv4 or IPv6; unspecified;

IP\_ADDRESS\_TYPE\_IPV4 = 1, // IPv4 address type

IP\_ADDRESS\_TYPE\_IPV6 // IPv6 address type

}IP\_ADDRESS\_TYPE;

Note: IP\_ADDRESS\_TYPE\_ANY is currently supported only for server sockets!

The functions to disconnect a socket have changed and have been added functionality:

* + **TCPDisconnect(TCP\_SOCKET hTCP)** 
    - This function closes the TX side of a connection by sending a FIN (if currently connected) to the remote node of the connection. It no longer sends a RST signal to the remote node so that a sequence of 2 TCPDisconnect() calls in a row is no longer needed.
    - If the socket has the linger option set (default), the queued TX data transmission will be attempted before sending the FIN. If the linger option is off, the queued TX data will be discarded.
    - No more data can be sent by this socket but more data can be received (the socket will be eventually closed when FIN received from the remote node or by a timeout dictated by the TCP\_FIN\_WAIT\_2\_TIMEOUT value in tcp\_config.h)
    - Please note that this call may fail in which case it can be re-issued.
    - Setting the socket options is done using the call TCPSetOptions();
  + **TCPAbort(TCP\_SOCKET hTCP, bool killSocket )**
    - This function aborts a connection to a remote node by sending a RST (if currently connected). Any pending TX/RX data is discarded.
    - A client socket will always be closed and the associated resources released. The socket cannot be used again after this call.
    - A server socket will abort the current connection:
      * if killSocket == false the socket will remain listening
      * if killSocket == true the socket will be closed and all associated resources released. The socket cannot be used again after this call.
  + **TCPClose(TCP\_SOCKET hTCP)**
    - Disconnects an open socket and destroys the socket handle, releasing the associated resources.
      * If the graceful option is set for the socket (default) a TCPDisconnect will be tried (FIN will be sent).
        + If the linger option is set (default) the TCPDisconnect will try to send any queued TX data before issuing FIN.
        + If the FIN send operation fails or the socket is not connected the abort is generated.
      * If the graceful option is not set, or the previous step could not send the FIN:
        + A TCPAbort() is called, sending a RST to the remote node. Communication is closed. The socket is no longer valid and the associated resources are freed.
    - Setting the socket options is done using the call TCPSetOptions();

The function to adjust the size of a socket’s RX and/or TX buffers has changed and has been added functionality:

* + **TCPAdjustFIFOSize(TCP\_SOCKET hTCP, uint16\_t wMinRXSize, uint16\_t wMinTXSize, TCP\_ADJUST\_FLAGS vFlags)**
    - The TX and RX FIFOs (buffers) associated with a socket are now completely separate and independent.
    - New flags have been added: **TCP\_ADJUST\_TX\_ONLY** and **TCP\_ADJUST\_RX\_ONLY that** allow changing the size of TX and RX buffers independently. This is the preferred option.
    - However, when **TCP\_ADJUST\_TX\_ONLY** or **TCP\_ADJUST\_RX\_ONLY** are not set, for the purpose of this function, the TX and RX FIFOs are considered to be contiguous so that the total FIFO space is divided between the TX and RX FIFOs. This provides backward compatibility with previous versions of this function.
    - Note that the TX or RX associated buffer sizes can be independently changed too using the socket options. See **TCPSetOptions** help**.**
* UDP changes (“microchip/include/tcpip/udp.h”).

**There is no more DNS resolution done automatically when opening a UDP socket. The UDP layer takes only IP addresses as an input parameter.** If you need a DNS resolution you have to perform it before opening a socket (see “microchip/include/tcpip/dns.h”).

**Support for IPV6 added.**

To openUDP sockets, the required functions are now:

* + **UDPOpenServer**(IP\_ADDRESS\_TYPE addType, UDP\_PORT localPort, IP\_MULTI\_ADDRESS\* localAddress);
  + **UDPOpenClient**(IP\_ADDRESS\_TYPE addType, UDP\_PORT remotePort, IP\_MULTI\_ADDRESS\* remoteAddress);

These new functions replaced the old UDPOpen.

**The application code has to replace the calls to UDPOpen with UDPOpenClient or UDPOpenServer ass appropriate.**

Note: IP\_ADDRESS\_TYPE\_ANY is currently supported only for server sockets!

* ARP changes (“microchip/include/tcpip/arp.h”).

The ARP module has been redesigned to support multiple network interfaces and to implement internal storage (caches) per interface for eliminating the need for frequent access to the network.

Some of the most important changes:

* + - Manipulation/control of the cache entries (ARPEntryGet, ARPEntrySet, ARPEntryRemove) with permanent entries support
    - Dynamic notification mechanism (ARPRegisterHandler, ARPDeRegisterHandler) for signaling of the ARP events
    - Resolution calls (ARPResolve, ARPProbe, ARPIsResolved).

Porting notes:

* + - The resolution calls haven’t changed (ARPResolve, ARPIsResolved).
    - Normally the application doesn’t need access to the ARP module. The address resolution is performed internally by the stack.
    - The ARP module cache manipulation access is meant for TCP/IP stack control and configuration depending on the actual network topology.
* HTTP API changes

(“microchip/include/tcpip/http2.h”):

The HTTP API has been improved. All the functions that the HTTP exposes as its API now take a first parameter, a HTTP connection handle (**HTTP\_CONN\_HANDLE**). This allows a cleaner and better behavior in both multi-threaded environments and in the situation where we may want to run multiple instances of the HTTP process itself, allowing even for serving multiple independent connections.

* + Basically the HTTP functions now look like:
    - **HTTPIncFile**(HTTP\_CONN\_HANDLE **connHandle**, const uint8\_t\* cFile) ; **HTTPExecuteGet**(HTTP\_CONN\_HANDLE **connHandle**); **HTTPExecutePost**(HTTP\_CONN\_HANDLE **connHandle**); **HTTPCurConnectionFileGet** (HTTP\_CONN\_HANDLE **connHandle**); **HTTPCurConnectionCallbackPosGet**(HTTP\_CONN\_HANDLE **connHandle**); etc, etc.
  + Also the functions exposed in **http\_print.h** take the connection handle parameter:
    - **HTTPPrint**(HTTP\_CONN\_HANDLE **connHandle**, uint32\_t callbackID); **HTTPPrint\_hellomsg**(HTTP\_CONN\_HANDLE **connHandle**);
    - etc.; etc.
  + Support for the HTTP module client to store/retrieve its own connection related data:
    - **HTTPCurConnectionUserDataSet**(HTTP\_CONN\_HANDLE connHandle, const void\* uData);
    - **HTTPCurConnectionUserDataGet**(HTTP\_CONN\_HANDLE connHandle);
  + These changes affect the functions calls in the **custom\_http\_app.c** . To port an existent application the extra parameter will have to be added. The connection handle is passed to the application as part of the HTTP callback. The modifications should be minimal and doable with an editor without any other impact.
  + Please note that the mpfs2 generator tool (mpfs2.jar) has been updated to support the new HTTP API.
  + See the list of the complete API changes in “microchip/include/tcpip/http2.h” and “..\tcpip\tcpip\_web\_server\_demo\_app\http\_print.h”
* SMTP API changes

(“microchip/include/tcpip/smtp.h”):

The SMTP API has been updated so that the access to internal variables was removed. This avoids synchronization issues and complications when the stack has to run in multi-threaded environments and there are multiple SMTP clients.

* + The SMTP function that has changed is SMTPSendMail:

**void SMTPSendMail(SMTP\_POINTERS\* smtpClientMessage);**

**It takes an extra parameter that carries all the information needed for sending a new SMTP message.**

* + Please note that the data fields pointed to by the **smtpClientMessage** parameter have to be non-volatile until the SMTP message is sent. There is no copy of the user data in the internally maintained SMTP storage area.
* TCP/IP Stack v6 initialization changes

(“microchip/include/tcpip/tcpip\_manager.h”):

To initialize the stack the call is now

**TCPIP\_STACK\_Init**(const TCPIP\_NETWORK\_CONFIG\* pNetConf, int nNets, const TCPIP\_STACK\_MODULE\_CONFIG\* pModConfig, int nModules);

* pNetConf- is a pointer to an array of configurations for all the initialized network interfaces
* nNets – is the number of the network interfaces that the stack is to support
* pModConfig – is a pointer to an table storing configuration data for the TCP/IP stack modules
* nModules – is the number of entries in this table, i.e. the number of initialized modules

The default TCP/IP stack configuration is provided with the stack distribution and consists of:

**“project/configs/platform/bsp\_profile/network\_config.h”:: TCPIP\_HOSTS\_CONFIGURATION[]** table: an array of **TCPIP\_NETWORK\_CONFIG** entries describing the default configuration of each network interface.

**“project/configs/platform/tcpip\_profile/tcpip\_module\_config.h”:: TCPIP\_STACK\_MODULE\_CONFIG\_TBL[]** table: an array of **TCPIP\_STACK\_MODULE\_CONFIG** entries storing the default configuration data for each module of the TCP/IP stack. This is just an aggregate of all the default configurations per module found in “**project/configs/platform/tcpip\_profile/**module\_config.h” headers, like: tcp\_config.h, udp\_config.h, arp\_config.h, etc. etc.

These tables are defined by default for you in the configuration files.

See the stack configuration topic below.

You can change the parameters of these structures as appropriate.

Example of the TCP/IP initialization is part of the distributed **main\_demo.c.**

**Note: The stack initialization may fail! The application code has to check the result of this call to detect if the stack failed to initialize properly.**

* TCP/IP Stack v6 access changes

(“microchip/include/tcpip/tcpip\_manager.h”):

Direct access to the internally maintained stack structures has been removed.

In order to access the information for a specific network interface a handle has to be obtained using the interface name. For example:

TCPIP\_NET\_HANDLE netH = TCPIP\_STACK\_NetHandle(“PIC32INT”);

Or

TCPIP\_NET\_HANDLE netH = TCPIP\_STACK\_NetHandle(“MRF24W”);

Once you have a handle to the interface you can query/set different parameters of that network interface:

TCPIP\_STACK\_NetAddress(netH) – returns the network interface address

TCPIP\_STACK\_NetGatewayAddress(netH) – returns the interface gateway address

Etc., etc.

Please see the “microchip/include/tcpip/tcpip\_manager.h” for a complete list of functions.

Note that the TCPIP\_NET\_HANDLE is an opaque data type.

The well known names of the network interfaces are currently in the file “microchip/include/tcpip/tcpip.h”.

Please note that the exact header file that exposes the network interfaces names may change in the future (but the names of the supported network interfaces will be retained).

Use the “**project/configs/platform/bsp\_profile/network\_config.h”** as an example.

Please see main\_demo.c for an example of how to use the interface handle functions.

* TCP/IP Stack v6 storage changes

(“microchip/include/tcpip/tcpip\_storage.h”)

**The TCP/IP storage has become obsolete** **and is no longer maintained**.

This is mainly because of the way the dynamic initialization of the stack is done now:

* + Each module has its own initialization data:
  + There are many parameters that could be relevant for an application and that may require storage besides the IP address, IP Mask, or SSID, etc.
  + The data is passed dynamically at the stack initialization. There is no more “build time” set of parameters against which to check at stack configuration (although support for a default configuration set at build time is present). The actual data contents are the application responsibility.
  + A system database service will be added that will use the File System. The database service will maintain configurations for all the modules in the system, including the TCP/IP. The system database service API will be available to the applications as well for storing/retrieving proprietary information.
  + The symbol “tcpip\_config.h::TCPIP\_STACK\_USE\_STORAGE” should NOT be enabled. **The service will be eventually removed from the distribution**.

## TCP/IP Stack v6 new API functions

There are lots of new functions available that have been introduced to take advantage of new features like IPv6 and multiple interfaces support.

However, there is no concern for the porting process regarding the new API calls because the previous stack implementation did not support this kind of services.

The new API functions should be added to the application only when the access to the new features is needed.

Existent applications should port easily without using the new API.

For reference following are few of the most important new API functions that could provide useful in the porting of your application:

* Initialization of the stack network interfaces:
  + TCPIP\_STACK\_NetUp(), TCPIP\_STACK\_NetDown()
* Default interface selection:
  + TCPIP\_STACK\_GetDefaultNet(), TCPIP\_STACK\_SetDefaultNet
* TCP/UDP Multiple interface support:
* TCPBind(), TCPRemoteBind()
* TCPSocketSetNet(), TCPSocketGetNet()
* UDPBind(), UDPRemoteBind()
* UDPSocketSetNet(), UDPSocketGetNet()

Please check “microchip/include/tcpip/tcpip\_manager.h” header file for a complete list to all new network interfaces API.

## Changes to the main program.

There are few changes that the main program loop has to take care of. A full working example is given with the main\_demo.c that comes with the stack distribution.

* The main program should call

SYS\_Initialize().

This is what initializes the system and runs the BSP (Board Support Package) specific code.

* The main program loop should call:

SYS\_Tasks();

This is the function that takes care of the system tasks and has to be called periodically.

* The main program loop should call:

TCPIP\_STACK\_Task ();

This is the TCP/IP stack tasks function that runs all the state machines that are part of the TCP/IP stack. It has to be called periodically. Note the name change.

* To get access to a network interface use:

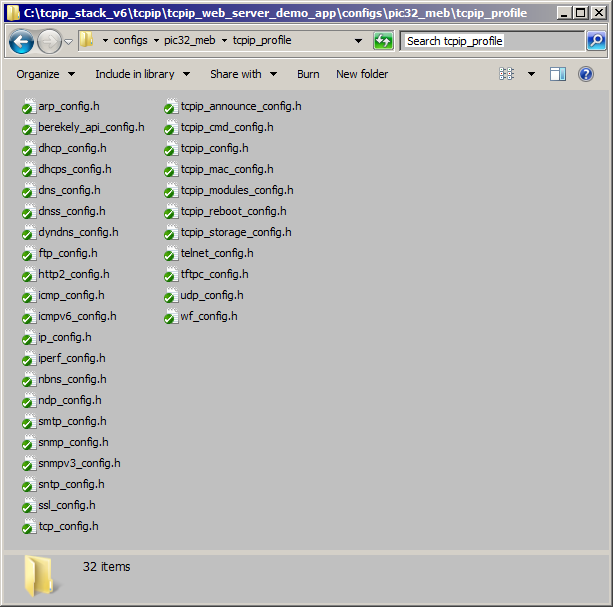
TCPIP\_STACK\_NetHandle(“iname”);

See examples in the previous section.

## TCP/IP Stack v6 configuration changes

The stack configuration has a completely new structure. Each project has its own “configs” folder that stores multiple configurations based on CPU platforms and hardware boards.

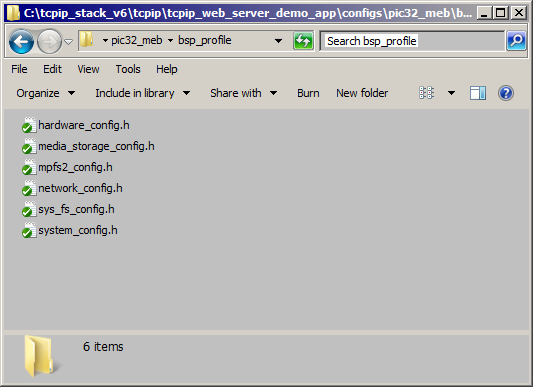
Example TCP/IP stack modules configuration layout:



The following is a list of the most important features:

* **“Config” folder for projects -** Stores all the “profiles”
* **Multiple profiles per project -** Separated by CPU platform. The TCP/IP profile is usually common.
* “**project/configs/platform/tcpip\_profile/**tcpip\_config.h” – selects modules that are part of the build
* Each module has its own profile: “**project/configs/platform/tcpip\_profile/**arp\_config.h”, tcp\_config.h, ssl\_config.h, etc.
* **BSP profiles per board containing:**
  + “**project/configs/platform/bsp\_profile/**hardware\_config.h” – the BSP specific hardware settings
  + “**project/configs/platform/bsp\_profile/**media\_storage\_config.h” – storage media and partitioning settings
  + “**project/configs/platform/bsp\_profile/**sys\_fs\_config.h” – file system configuration and settings
  + “**project/configs/platform/bsp\_profile/**mpfs\_config.h” – MPFS file system settings
  + “**project/configs/platform/bsp\_profile/**network\_config.h” – configuration for every network interface: NetBIOS names, default IP addresses, etc
  + “**project/configs/platform/bsp\_profile/**system\_config.h” – system configuration: the system tick, the system console selection, system debug services, etc.

Example BSP configuration layout:



**TCP/IP stack v6 configuration note**:

The presented structure is just a model and a different layout can be chosen.

The only requirement is to use the proper path in your project so that the necessary configuration files are found at build time.

For example you can add something like this to your project include path:

../configs/pic32\_eth\_sk/mrf24w/tcpip\_profile

or

../configs/pic32\_eth\_sk/mrf24w/bsp\_profile/meb

depending on your exact hardware platform/board.

### TCP/IP Stack v6 Heap configuration:

The TCP/IP stack v6 uses dynamic memory for both initialization and run time buffers.

Therefore there are 2 requirements for a project containing the stack to work properly:

* The TCP/IP stack should be configured with enough heap space. The amount of heap used by the stack is specified by the “**project/configs/platform/tcpip\_profile/tcpip\_**config.h”:: TCPIP\_STACK\_DRAM\_SIZE parameter.

The value of the required TCP/IP heap for a project is application dependent. Some of the most important factors that impact the heap size are:

* + Number of TCP sockets. By default each TCP sockets requires 512 bytes for a RX buffer and 512 bytes for a TX buffer. These parameters can be adjusted by modifying the values specified in tcp\_config.h or dynamically through TCPSetOptions().
  + Number of UDP sockets:
    - For each UDP socket that needs to initiate a transmission the IP layer will have to allocate the required space (suggested by the UDPv4IsTxPutReady or UDPv4IsTxPutReady functions.
    - Once the UDP buffering will be added, each socket will have its own RX and TX buffers. These parameters can be adjusted by modifying the values specified in udp\_config.h or dynamically through UDPSetOptions().
  + The type of Ethernet MAC that is used. The PIC32MX6-7 family with built in 100 Mbps Ethernet controller uses system memory for buffering incoming RX traffic. For sustained 100 Mbps operation enough RX buffer space has to be provided. This parameter can be adjusted by modifying the values specified in “**project/configs/platform/bsp\_profile/**network\_config.h”:: TCPIP\_HOSTS\_CONFIGURATION[] table.
  + If SSL is enabled, it will require additional buffering for encryption/decryption at run time. TBD: the exact RAM requirements per SSL connection.
* **The project that includes the TCP/IP stack has to be built with sufficient heap size** to accommodate the stack (obviously the project needs at least the TCPIP\_STACK\_DRAM\_SIZE bytes of heap). This parameter is adjusted on the linker tab in the project properties.

As a general rule, for a TCP/IP project running on a PIC32MX with embedded Ethernet controller at least 8 KB heap space is needed by the stack. However, this implies:

* Not sustained 100 Mbps traffic
* No SSL
* Relatively few sockets

A typical value for handling comfortably 100 Mbps Ethernet traffic would be 40 KB of TCP/IP heap space.

The amount of the required heap is less for an external Ethernet MAC.

Notes:

1. If there is not enough heap space the stack initialization may fail.
2. If there is not enough heap space some run time buffer allocation may fail and some packets transmission will have to be deferred until a later time impacting thus the stack performance.
3. It is always a good idea to have a reserve, at least extra 2 KB of heap space above the total amount of space that’s used.
4. A very useful tool in understanding the heap allocation and how the space is distributed among the stack modules is the tcpip\_console. By enabling the symbol “**project/configs/platform/tcpip\_profile/tcpip\_**config.h”:: TCPIP\_STACK\_DRAM\_DEBUG\_ENABLE the stack will output debug messages when it runs out of memory. Then, using the command “heapinfo” at the tcpip\_console (system console) prompt will return a snapshot of the current TCP/IP heap status and can help in early detection of problems.

Furthermore, enabling the symbol “**project/configs/platform/tcpip\_profile/tcpip\_**config.h”:: TCPIP\_STACK\_DRAM\_TRACE\_ENABLE will instruct the TCP/IP heap allocation module to store trace information that will be displayed with the “heapinfo” command.

## TCP/IP Stack v6 utilities

1. The MPFS2 Java utility that assists the application in generating the MPFS image of the files needed by the Web server has been updated. It supports both versions of the TCP/IP stacks with improved functionality.
   1. The old tool can still be used but some minor adjustments may need to be done manually.
2. TCPIP Discoverer. This Java tool has been updated to support not only TCP/IP Stack v6 but also IPv6. It is up and running and it is backward compatible with TCP/IP stack v5.
3. TCPIP Configuration tool. This tool is not supported for now. The need of individually configuring TCP and UDP sockets is gone: all the sockets behave similar and they are allocated dynamically as needed. Probably the tool will be updated eventually to support some other features still needed in the TCP/IP stack v6.

## TCP/IP Stack v6 SSL/RSA usage

The SSL design has been updated to support multiple interfaces. The same is true for the RSA engine. However, the SSL module doesn’t yet use the Microchip Crypto Library. Until the proper crypto library support is added to SSL the stack is distributed with the RSA and RC4 code embedded into the stack.

Eventually the crypto libraries will be completely removed from the TCPIP stack. The stack will be just a client to the crypto library. This is work in progress.

## TCP/IP Stack v6 design

The stack is designed as being a part of a system running some other applications, middleware, etc.

Therefore it makes use of the system services that are available to other modules in the system or to the application, like:

* File System
* System interrupts, driver services
* System timers
* System device drivers
* System command processor
* System console, debug services, etc.

Please see “microchip/include/system” for the header files that expose the system wide available API.