EMBEDDED ETHERNET SYSTEM DESIGN GUIDE

Relevant Devices

This application note applies to the following devices:
CP2200, CP2201

1. Introduction

Embedded systems today are small, fast, and very powerful. Embedded connectivity stands at the forefront of harnessing the power of today's embedded systems. The CP220x is an Ethernet Controller that integrates Ethernet functionality into a single 5 x 5 mm package. When placed in an embedded system with an MCU, it provides the system with Embedded Ethernet Connectivity as shown in Figure 1.

This design guide discusses the benefits of Embedded Ethernet and walks you through three easy steps to add Ethernet connectivity to your embedded system: System Definition, Hardware Design, and Software Development. It also provides timesaving tips and suggestions to simplify the implementation of Embedded Ethernet. No previous knowledge of Ethernet or TCP/IP is required to use this guide for Embedded Ethernet Development.

2. Embedded Ethernet Connectivity

Imagine if you could remotely monitor the status of your embedded system using a web browser, or if the vending machine could send out an e-mail alert when it needs service or is sold out of specific items. These things are all made possible with Embedded Ethernet.

The key benefits of Embedded Ethernet Connectivity are described in the following paragraphs.

- **Remote Monitoring and Control** - Once an embedded system is on a network, it can be accessed from any PC on the same network. The user does not have to be in the same room or building in order to access and control the embedded system.
- **No PC Software Development Required** - In most systems, the user interface on the PC will use a Web Browser, HyperTerminal, or the embedded system will directly send the user an e-mail. Some systems will implement custom PC applications to perform specific tasks not easily implemented using commonly available software.
- **Utilizes Existing Infrastructure** - Ethernet is the most widely implemented networking standard. Most commercial offices and industrial factory workfloors are already wired for Ethernet connectivity. When wireless LAN is used across a factory floor, low cost bridges are available to connect wired ethernet devices to a wireless network.
- **Low Cost and Easy to Implement** - With MCUs such as the C8051F34x and Ethernet controllers such as the CP220x, Embedded Ethernet can now be integrated into cost sensitive applications. Using the TCP/IP Configuration Wizard and other Silicon Laboratories development tools, Embedded Ethernet is now easy to implement.

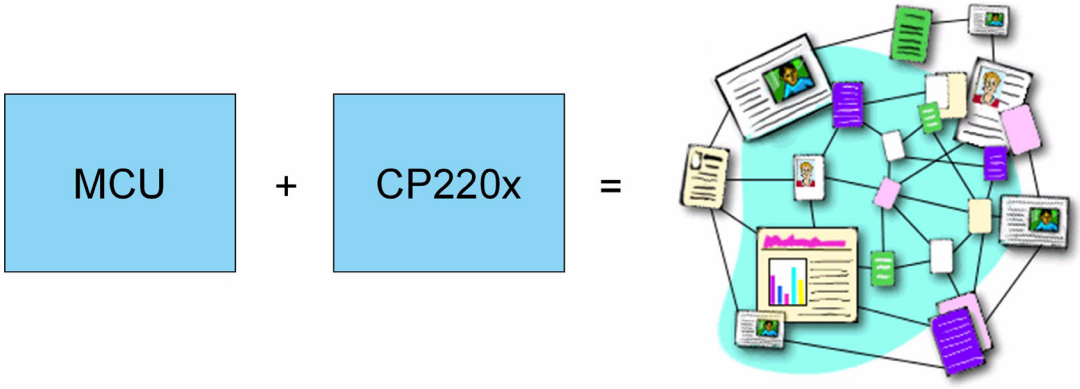

Figure 1. Embedded Ethernet Connectivity

3. How to Use This Design Guide

This design guide was developed for both beginners and experienced Embedded Ethernet system designers. Figure 2 shows the typical system design flow for Embedded Ethernet development. The structure of this document is based on Figure 2.

Each of the boxes in Figure 2 is discussed in detail and additional information about the basics of TCP/IP is included in the appendix on page 26. Since the content is modularized, advanced users may skip directly to the sections that will provide them with the most added benefit.

In addition to this design guide, the following design resources are essential for developing Embedded Ethernet with the CP220x:

- **"AN237: TCP/IP Library Programmer's Guide"** - contains the API reference for the TCP/IP Library based on the CMX Micronet (TM) TCP/IP Stack.
- **Embedded Ethernet Development Kit User's Guide** - contains information on how to setup the development kit and gives a hands-on tutorial of using the TCP/IP Configuration Wizard.
- **CP2201 Evaluation Kit User's Guide** - contains information on how to setup the evaluation kit and gives walks through the CP2201EK demo. The CP2201 Evaluation Kit allows you to quickly test drive the most common network interfaces.
- **CP220x Datasheet and the Target MCU Family Datasheet** - Contains the pinout, electrical characteristics, and specifications of the selected MCU and Ethernet controller.

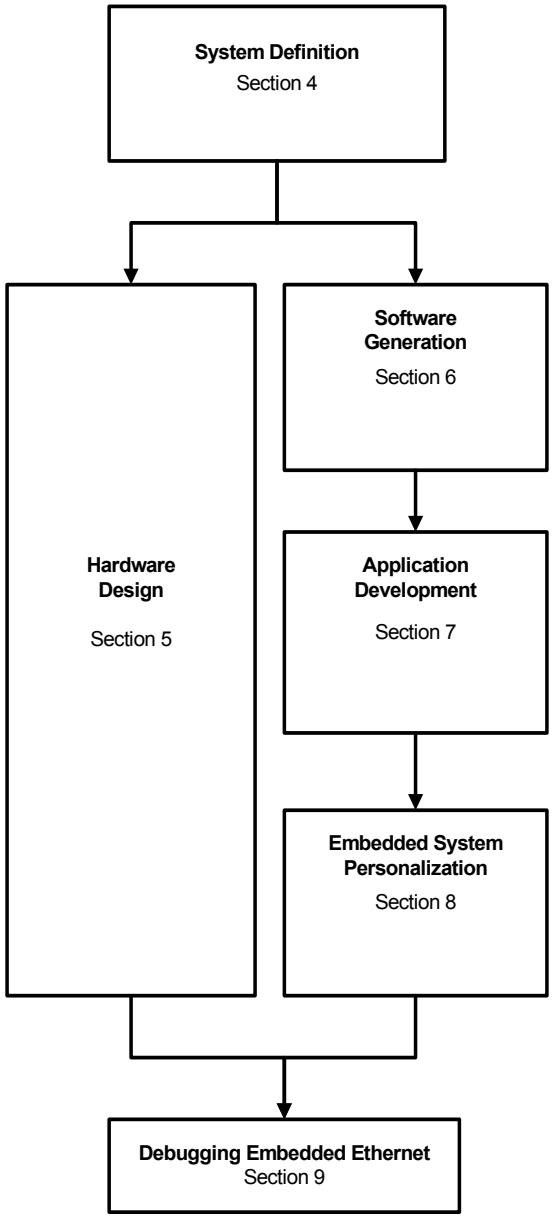

Figure 2. Embedded Ethernet System Design Flow Diagram

4. System Definition

There are four steps to defining a system with Embedded Ethernet Connectivity. Throughout the four stages, we will consider the embedded system to be a black box. Having a clear definition of the properties and characteristics of the embedded system prior to starting hardware and software development is essential to achieving a final result that matches its target specifications.

4.1. Specifying Required Functionality

The first question that should be answered when creating the definition is: *What do I want my embedded system to do?* The answer to this question varies by application and can encompass virtually anything that can be done with a high-speed MCU.

Figure 3 shows the black box representation of the embedded system. Based on the diagram, an example answer to question 1 is: *Monitor progress of the milling machine, control a motor, and keep an average of the number of boxes passing per minute.*

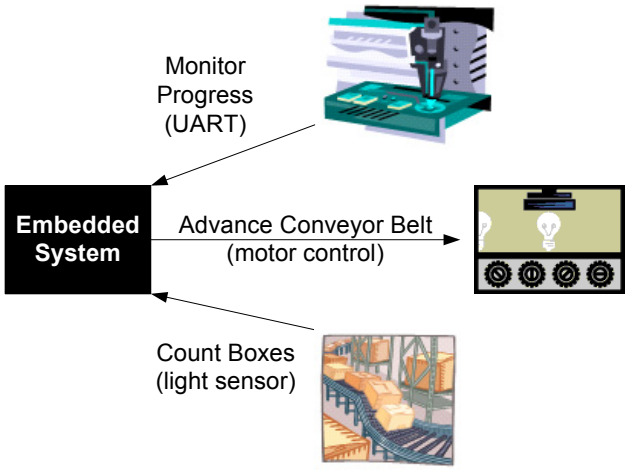

Figure 3. System Functionality Example

Please write down your answer to the first question in the system definition.

Q1: What do I want my embedded system to do?

4.2. Specifying Access Method

The second system definition question is: *How do I want to access my embedded system?* The answer to this question can be one or more of the most commonly used access methods:

- Using a web browser.
- Using HyperTerminal.
- Having the embedded system send e-mail.
- Using a custom application.

Figure 4 shows the black box representation of the embedded system and how the various access methods can be used to monitor and control the embedded system. See the CP2201 Evaluation Kit for a demonstration of each interface.

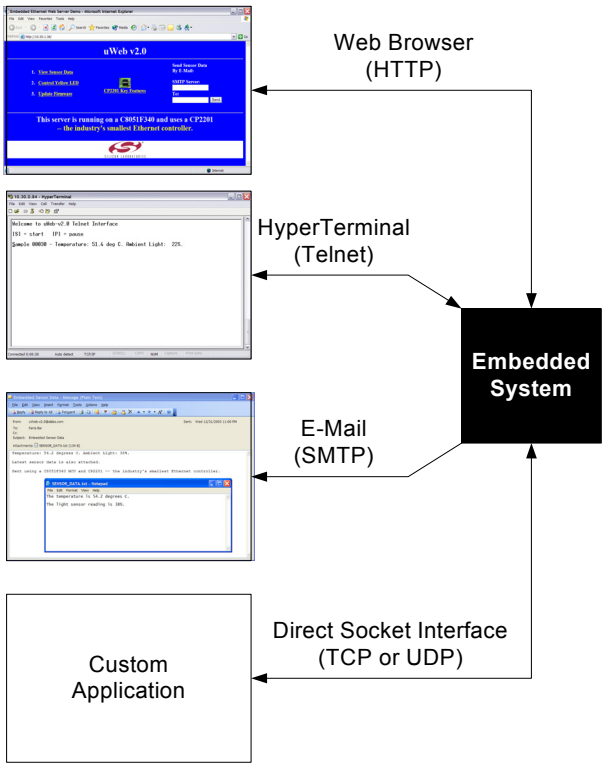

Figure 4. User Interface Options

Please write down your answer to the second question in the system definition.

Q2: How do I want to access my embedded system?

4.3. Specifying Configuration Method

Every device connected to a network requires both a MAC address and an IP address to communicate with other devices. Embedded systems using the CP220x only need to obtain an IP address because the MAC address is factory pre-programmed in Flash memory.

The third system definition question is: *How do I want to configure my embedded system?* There are four common configuration methods:

- Automatic Network Configuration.
- Automatic Network Configuration with Netfinder.
- Static Network Configuration.
- Static Network Configuration with Netfinder.

4.3.1. Automatic Network Configuration

Automatic Network Configuration allows a device to use the Dynamic Host Configuration Protocol (DHCP) to acquire an address from the network. This assumes that the network has a DHCP server that can assign an IP address. DHCP servers are typically found inside routers or other network equipment.

Embedded systems using automatic network configuration can access the network (e.g. to send an e-mail message or log in to a server with a known address); however, the user will not be able to directly access the device from a web browser or a telnet client without knowing the IP address assigned to it. This limitation can be overcome by adding additional hardware such as an LCD screen or using Netfinder.

4.3.2. Searching for Automatically Configured Embedded System Using Netfinder

Enabling Netfinder capability on an embedded system that uses DHCP allows the user to easily find out the IP address assigned to the embedded system. This is done by searching for the embedded system using the Netfinder PC utility.

When a new search is started, the Netfinder utility broadcasts an “Identity Request” message to all nodes on the network. Each embedded system that supports Netfinder replies with information that identifies and differentiates itself from similar embedded systems. This information can include: IP address, Elapsed time from an event (e.g. Time Powered, Time on Network), MAC Address, and a text description of the device. This information can be customized for each application.

4.3.3. Static Network Configuration

For networks that do not have a DHCP server, each network node including embedded systems must be assigned a static IP address. To prevent multiple devices from using the same IP address, the network administrator keeps a database of each device on the network and the IP address assigned to it.

There are a number of ways to assign a static IP address to a device. First, the IP address can be hard coded in firmware. This method is not user friendly since the device must be reprogrammed in order to change its IP address.

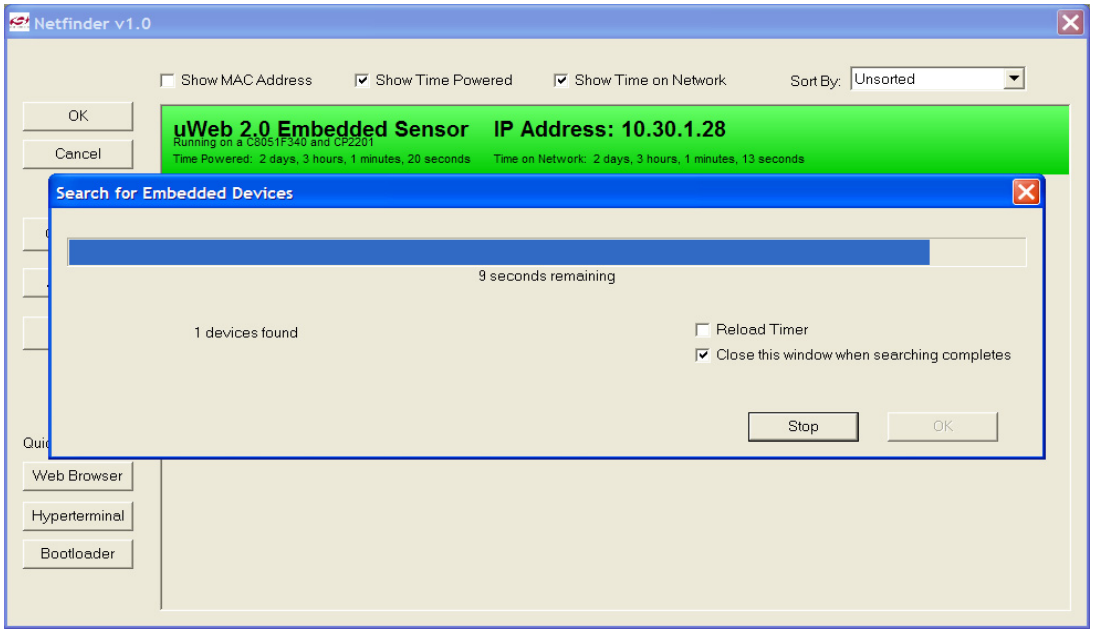

Figure 5. Searching for an Automatically Configured Embedded System Using Netfinder

Second, it can be assigned through the serial port and stored in Flash memory as demonstrated in the Embedded Ethernet Development Kit User's Guide. This method provides the flexibility to change the IP address after the system is in the field but requires the embedded system to implement a UART interface. Unless the UART interface is required by the application, there are smaller and more cost effective ways of programming the IP address.

4.3.4. Assigning an IP Address Using Netfinder

Enabling Netfinder capability on an embedded system that uses static network configuration allows the Netfinder utility to both search for the embedded system and assign it an IP address. If the embedded system does not have an IP address, it will default to the invalid address "0.0.0.1" until the user assigns it an IP address using the Netfinder utility.

Design Suggestion: If maximum compatibility with different networks is desired, the embedded system can be designed to use multiple configurations. For example, the CP2201 Evaluation Board first attempts to acquire an IP address through DHCP. If it fails to acquire an IP address after 4 attempts, it will go into static IP address mode and wait for the user to assign it an address using the Netfinder utility. As a result of including Netfinder capability in firmware, the embedded system supporting both static and dynamic network configuration measures only 1.25" x 1.5".

Please write down your answer to the third question in the system definition. If static network configuration is chosen, then please provide an answer to Part B.

Q3: How do I want to configure my embedded system?

_____
_____
_____
_____
_____

Q3B: If a static IP address is assigned, do I want the embedded system to permanently store the address or attempt to refresh it each time I plug it into a network?

_____
_____
_____
_____
_____

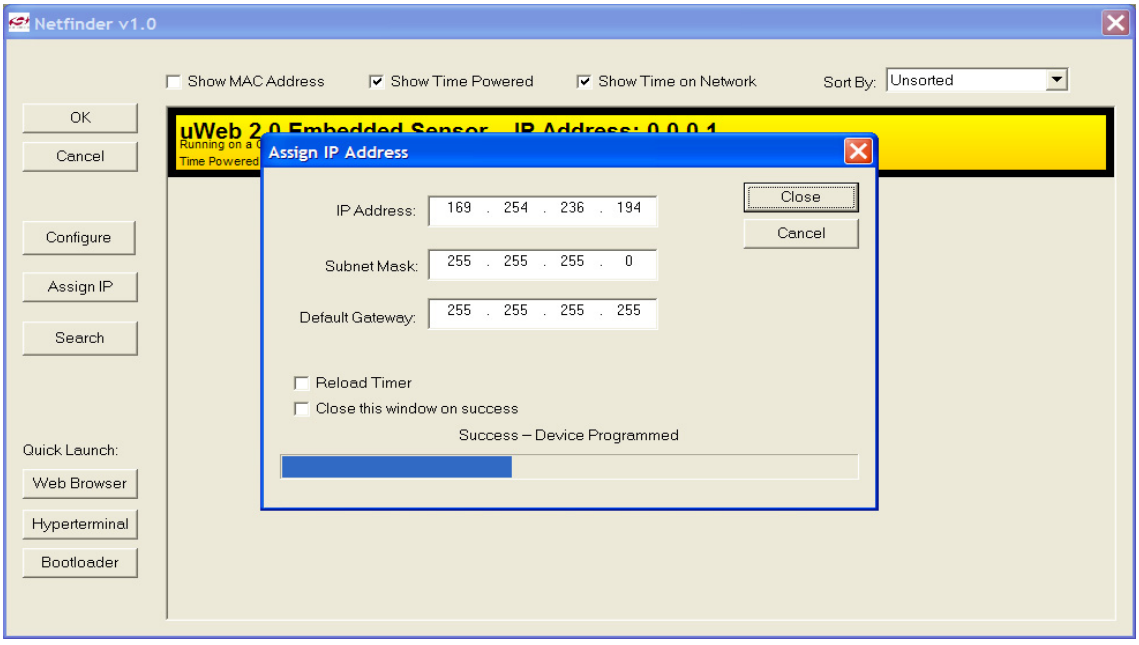

Figure 6. Assigning a Static IP Address Using Netfinder

4.4. Specifying Field Re-Programmability Requirements

The final part of the system definition is determining the field re-programmability requirements of the embedded system. The options for field reprogrammabilty are:

- No support for field re-programmability.
- Re-programmability using a 3 or 5 pin header.
- Re-programmability using a 10-pin header.
- Re-programmability using a bootloader.

Figure 7 shows the black box representation of the embedded system and the available field re-programmability options.

4.4.1. Updating Firmware using a Header

Firmware on the device may be updated by placing a programming header on the board. If a 10-pin header is used, the USB Debug Adapter can plug directly into the embedded system for debugging or updating firmware. If a custom header is used, then C2 devices will need at least 3 pins (C2CK, C2D, and GND) and JTAG devices will need at least 5 pins (TCK, TMS, TDI, TDO, and GND).

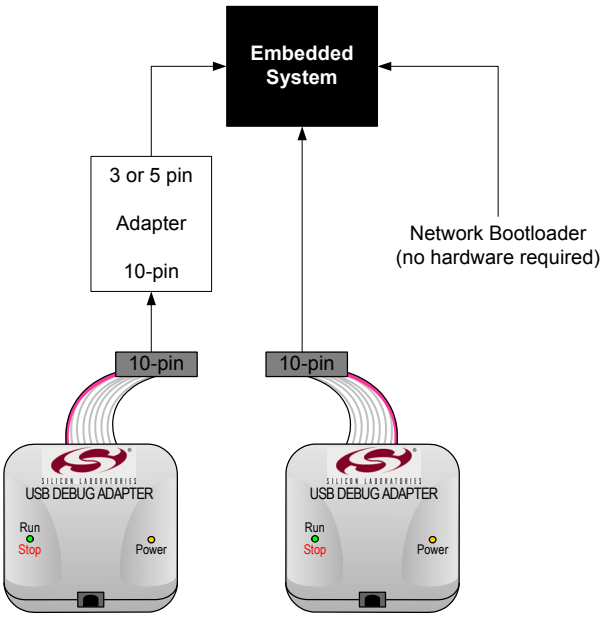

Figure 7. Field Re-programmability Options

When designing the embedded system for field reprogrammability, one criterion that may be used to select the best field re-programmabily option is the requirement of the USB Debug Adapter. Depending on the application and the cost of the embedded system, end customers may not have access to a USB Debug Adapter.

For end customers who do not have a USB Debug Adapter, the only option will be to purchase one or to send the entire system back to the factory for re-programming. This may be feasible in low quantities, but low cost, high volume applications will require an easier way of updating firmware in the field.

4.4.2. Updating Firmware over the Network

Using a TFTP Bootloader, firmware may be remotely updated over the network using any PC with a TFTP client. A TFTP client comes as a core utility in most operating systems including Windows 2000/XP.

A TFTP bootloader does not have any hardware requirements and is easy to use by the end customer. The drawback to using a TFTP bootloader is that it requires approximately 10 kB of code space. Other types of bootloaders, such as UART, SMBus, etc. may also be used to update firmware.

Timesaving Tip: The CP2201EK uses a TFTP bootloader to allow remote firmware updates over the network. If you are using the ‘F340, the source code for the bootloader is available for use in your design. If you are using a different device, then the TFTP bootloader may be easily ported.

Please write down your answer to the fourth question in the system definition.

Q4: How do I want to update firmware in my embedded system?

4.5. Translating the System Definition to Protocol Requirements

One of the benefits of having a formal definition is that a list of required protocols can be made and used during hardware and software development.

Table 1 shows the mapping of protocol acronyms to requirements in the system definition. Please make a list of the protocols needed by your embedded system because they will be used for MCU selection and software generation.

Note: For simplicity, some advanced protocols have been omitted from Table 1. They are described in the appendix on page 26.

Table 1. System Definition to Protocol Mapping

If the System Definition Specifies...	You need the following protocol(s):
Automatic Network Configuration	DHCP
Netfinder Search or Assign Capability	NETFINDER
Web Browser Interface	HTTP
HyperTerminal/Telnet Interface	TCP
E-mail Interface	SMTP
Custom Application Interface	TCP or UDP depending on application

5. Hardware Design

With a system definition in place, it is now time to start designing the hardware. The hardware design flow consists of 5 steps corresponding to the 5 sections of a schematic for an embedded system with Ethernet connectivity. The hardware design flow is shown in Figure 9. Each schematic section is described below:

- Custom application circuitry - sensors, indicators, and other application-specific circuitry.
- MCU - the main system controller.
- Ethernet Controller - provides the MCU with the capability to send and receive data over a network.
- Ethernet Connector - the RJ-45 connector, magnetics, and link/activity LEDs.
- Power circuit - provides the embedded system with regulated 3.3 V power.

Timesaving Tip: The CP2201 Evaluation Kit schematic found in the CP2201EK User's Guide can be used as a starting point for most designs. Figure 8 shows the 5 blocks of the CP2201EK schematic. Notice that they match the 5 sections outlined above.

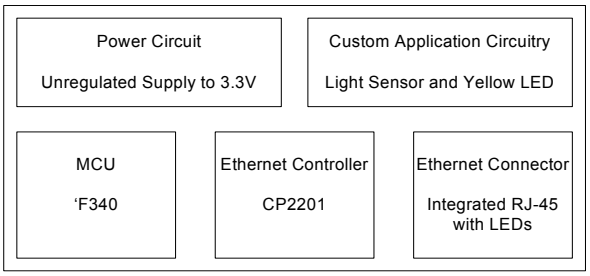

Figure 8. CP2201EK Schematic Blocks

5.1. Custom Application Circuitry

The custom application circuitry includes any application-specific sensors, control circuitry, interface headers, etc. that are required to perform the required system functions specified in the system definition. As the circuitry is being designed, the system designer should estimate the power requirements of this section for use when designing the power circuit.

5.2. Designing the MCU Section

Designing the MCU section involves determining the required analog peripherals (such as ADCs, DACs, Comparators, etc.) and estimating the memory and speed requirements. Based on these requirements, the most appropriate MCU can be selected and the circuitry surrounding the MCU can be designed.

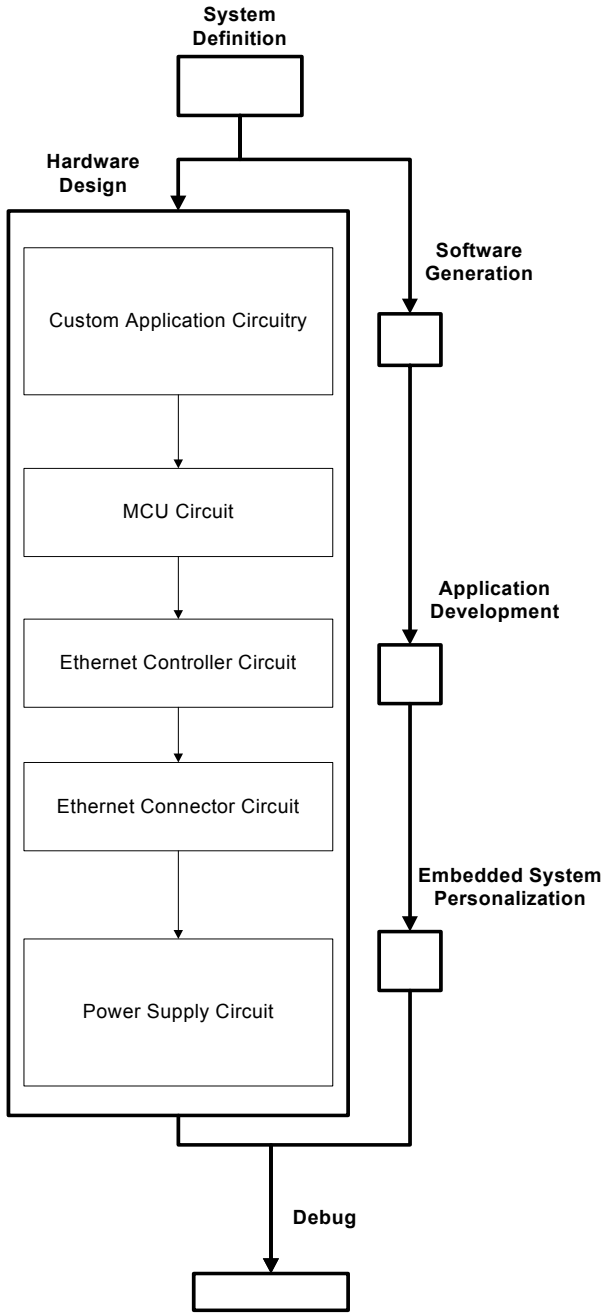

Figure 9. Hardware Design Flow

5.2.1. Determining MCU Peripheral Requirements

The analog peripheral requirements of the MCU will come directly from the system definition. If the MCU needs to sense an analog voltage, then an ADC will be required. If the MCU needs to drive an analog output such as a speaker, then a DAC will be required.

5.2.2. Determining Flash Memory Requirements

The TCP/IP Library requires 16 to 50 kB Flash depending on the interfaces selected. Figure 11 shows the Flash requirements of the TCP/IP Library for various common configurations. The configuration needed for your embedded system comes directly directly from question 2 and 3 in the system definition.

5.2.3. Determining RAM Requirements

The TCP/IP Library also requires 3–5 kB of RAM depending on the protocols enabled. Since buffer sizes are configurable by the user, RAM requirements will depend on the user's desired performance. All configurations shown in Figure 11 can be implemented with less than 4 kB RAM except for the largest configuration on the far right. The largest configuration requires 4.5 to 5 kB RAM.

5.2.4. Determining MIPS requirements

Any Silicon Laboratories 25 MIPS or greater MCU will have more than enough CPU bandwidth to run both the TCP/IP Library and application code. If the application being developed will benefit from increased CPU bandwidth, the TCP/IP Library supports high performance MCUs with up to 100 MIPS.

5.2.5. Selecting an MCU

Any Silicon Laboratories MCU with 32 kB Flash or higher can be interfaced with the CP220x from a hardware standpoint. In order to use the TCP/IP Configuration Wizard and TCP/IP Library, the MCU must be in one of the following device families: 'F12x–'F13x, 'F02x, and 'F34x.

Figure 10 shows a comparison of the three supported device families. The most full-featured device from each family was selected for the comparison. Reduced functionality devices are also available in each family. See the MCU family datasheet for a description of the reduced functionality devices.

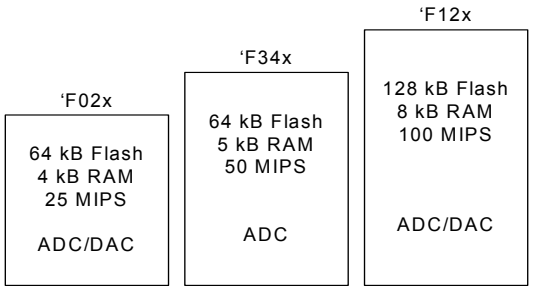

Figure 10. MCU Family Comparison

The TCP/IP Configuration Wizard is always being updated to add features and support for additional devices. Check the Silicon Laboratories Website at www.silabs.com/ethernet for the latest updates.

5.2.6. Adding Additional Memory

Some applications require more volatile or nonvolatile memory than available in the MCU. For such applications, external memory may be added to the system. The CP220x has 8 K of on-chip Flash that may be used for storing web server content or as general purpose nonvolatile memory. Note that nonvolatile memory added external to the MCU can be used for data storage, but cannot be used for code execution.

5.2.7. Adding the MCU to the Embedded System

Once the MCU has been selected, it is time to integrate it into the embedded system. The MCU integration guidelines on page 12 list the key points that should be followed when designing the MCU section of the schematic.

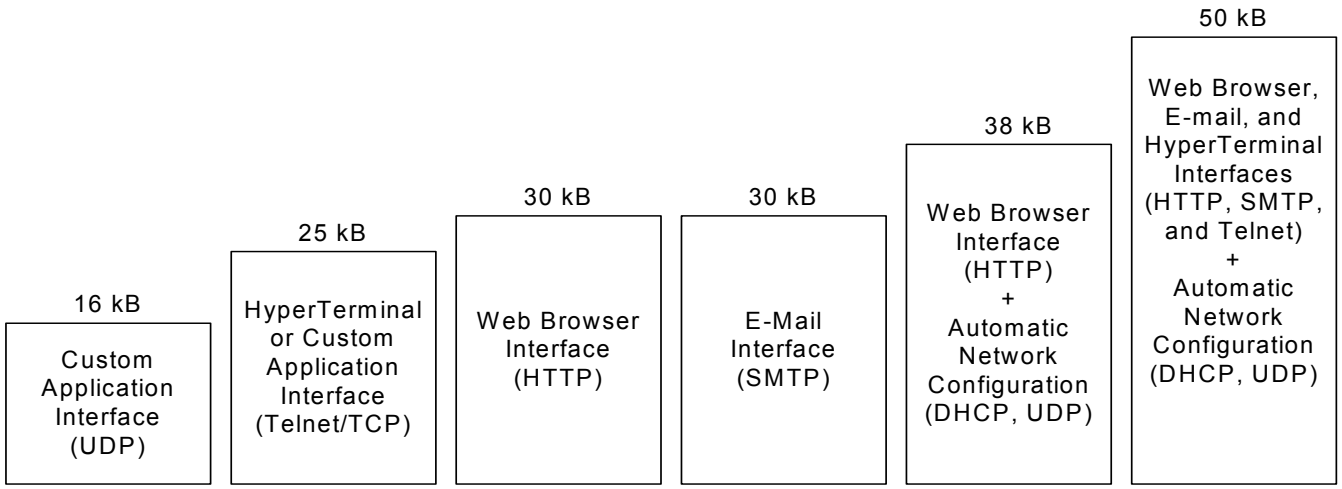

Figure 11. Flash Requirements for Various Interfaces

5.3. Ethernet Controller Section

Designing the Ethernet controller section involves selecting an Ethernet controller and integrating it into the embedded system.

5.4. System Level Benefits of the CP220x

The CP220x provides many system level benefits for embedded systems with Ethernet connectivity. Below are some of the key features of the CP220x:

- **Small (5 x 5 mm) package** - minimizes system size.
- **High speed parallel interface** - minimizes CPU bandwidth spent on transferring data.
- **Autonegotiation** - allows the use of full-duplex communication without manually configuring routers and switches.
- **8 kB Flash with Pre-Programmed MAC Address** - provides additional non-volatile memory in the system and simplifies product serialization.
- **TCP/IP Configuration Wizard** - auto generates a TCP/IP Library and framework code based on the CMX Micronet(TM) TCP/IP stack.

Currently, two Ethernet controllers are available in the CP220x family. Table 2 outlines the differences between the two devices.

Table 2. CP220x Comparison

Feature	CP2200	CP2201
Package	TQFP-48	QFN-28
Footprint	9 x 9 mm	5 x 5 mm
Parallel Interface Mode	Multiplexed or Non-Multiplexed	Multiplexed Only
Parallel Interface Speed	30 Mbps	25 Mbps
LED Controls	2	1
Link/Activity Indicators	Separate LEDs	Combined LED

5.4.1. Adding the Ethernet Controller to the Embedded System

After an Ethernet controller has been selected, it is time to integrate it into the system. The CP220x integration guidelines on page 13 list the key points to follow when designing the CP220x section of the schematic.

5.5. Ethernet Connector

The CP220x interfaces to an Ethernet cable through an RJ-45 connector and isolation transformers as shown in Figure 12. The connector, transformers, and optional link/activity LEDs can be discrete components or can be part of an integrated connector.

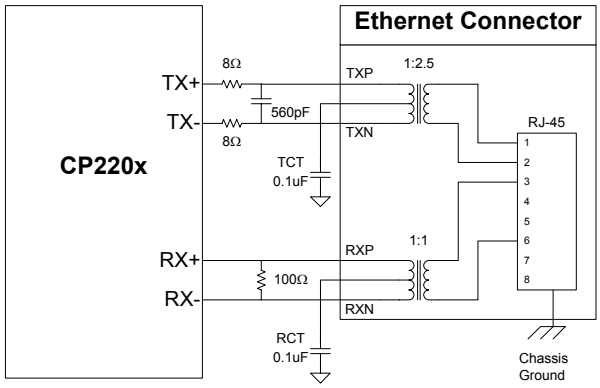

Figure 12. CP220x Connector Interface

To achieve the smallest board area, an RJ-45 connector with integrated magnetics and LEDs can be used. The following should be checked when selecting a connector:

- **Transformer Turns Ratio:** The transformer turns ratio must be 1:2.5 for the transmit side and 1:1 for the receive side.
- **Availability:** Connectors with integrated magnetics have typical lead times from 4 to 16 weeks depending on the supplier. Please check connector availability and lead time before designing the hardware. Note that connectors from different vendors typically do not share the same footprint.

Table 2 shows a partial list of Ethernet connectors that are compatible with the CP220x. It also shows the part numbers for discrete magnetics in case an integrated connector is not the best choice for the application.

Table 3. Example Part Numbers for Integrated Connectors and Discrete Magnetics

Manufacturer	Website	Part Number (Integrated)	Part Number (Discrete)
Halo	www.haloelectronics.com	HFJ11-1041[E]-[L12RL]	TG41-2006N
Tyco	www.tycoelectronics.com	[1-]6605752-1	HB724
Pulse	www.pulseeng.com	J00-0063	E2023
Bel Fuse	www.belfuse.com	SI-40047	LM01509

5.6. Power Circuit

The power supply circuit should be designed to provide a regulated 3.3 V DC output capable of delivering enough current to meet the demands of the entire system at peak loads. Since the CP220x requires a 3.1 to 3.6 V supply voltage, a 3.0 V regulator cannot be used. To provide adequate power to the system, the power supply should be capable of providing the MCU with at least 1 mA/Mhz and the CP220x with 150 mA (peak current). The maximum current capacity of the regulator should always exceed the peak current requirements of the system.

There are three options for powering the system: 9 V wall adapter and 3.3 V LDO, power over ethernet, and battery power. These options are described in the following paragraphs.

5.6.1. 9 V Wall Adapter and 3.3 V LDO

This method is the simplest to implement and is often the lowest cost. However, because the efficiency of linear regulators is low, a large amount of heat may be generated. To dissipate the heat, a multi-layer board with solid supply and ground planes may be used. An alternative is to use a switching regulator instead of a linear regulator.

5.6.2. Power over Ethernet

With the introduction of VoIP phones, powered ethernet switches are becoming mainstream and are falling in price. A powered ethernet plug delivers power to the embedded system through the 4 unused wires in the CAT5 ethernet cable. To design the power supply circuit to accept power directly from the ethernet cable, two functions are needed:

- An IEEE 802.3af compliant powered device (PD) interface - this interface provides a signature to the power sourcing equipment during PD detection and programs the correct classification mode according to the 802.3af specification.
- A 48–3.3 V switching regulator to convert the 48 VDC power on the ethernet cable to 3.3 VDC.

5.6.3. Battery Power

Due to the nature of most network enabled monitoring and control applications, the embedded system must be continuously powered. This does not lend itself well for batteries because batteries will need to be frequently replaced. If an application only requires ethernet connectivity for a few hours at a time, then battery power may be used. A typical 9 V Alkaline battery can provide 625 mAH @ 9 V leading to a typical battery life of 4–10 hours depending on the application and the amount of power used by the embedded system.

MCU Integration Guidelines

- **Pinout** - Each MCU's pinout is determined by a Crossbar, which is configured from software. It is important to verify that the desired device pinout is possible before finalizing the hardware design.
- **Port Input/Output Configuration** - When assigning I/O pins to specific functions, it is important to check if the selected I/O pins are capable of being configured to the desired mode. Some pins are digital only, some are analog only, and some can be used for both digital or analog signals.
  Timesaving Tip: On C8051F02x devices port 4 through port 7, each set of 4 adjacent bits must be configured to the same output mode. See the P47MDOUT register description in the C8051F02x datasheet for more details.
- **Special Signals** - The port pin selection for special signals should be chosen carefully to ensure that the desired functionality is achievable. These special signals include:
  - **CP220x Reset Pin (/RST)** - This signal should be connected to a pin configured as an open-drain output because it may be driven low either by the MCU or by the CP220x. The port pin selected for this signal should have the capability of generating a interrupt. This allows software to detect if the CP220x ever goes into reset due to a brownout or oscillator-fail condition.
  - **CP220x Interrupt Pin (/INT)** - This signal should be connected to a digital input pin capable of becoming External Interrupt 0. This is required by the TCP/IP Stack Library as outlined in the important notes section of "*AN237: TCP/IP Library Programmer's Guide*".
  - **/RD, /WR, ALE, and Address/Data Pins** - These signals should all be configured as push-pull outputs. During a read operation, the external memory interface automatically turns off the output drivers (making the pins high impedance) for the duration of the read operation.
- **Voltage Reference** - If the device has an on-chip voltage reference and the analog peripherals are used, then VREF decoupling capacitors should be added to the VREF pin.
- **Power and Ground Pins** - All power and ground pins on the device must be connected to power or ground. Also, be sure to provide adequate power supply decoupling.
- **V_{DD} Monitor** - If the MCU has a MONEN pin, then it should be tied directly to V_{DD} to enable the V_{DD} Monitor. Some devices also require input from software to turn on the V_{DD} Monitor. For example, the 'F12x and 'F13x family have a MONEN pin and require input from software. See the datasheet for the selected device to determine how to enable the V_{DD} Monitor.
- **Reset and Debug Pins** - We recommend placing a 1–5 K pull-up resistor on the reset pin and a 1 K pull-up on TCK (for JTAG devices). Do not add a capacitor directly on the debug pins as this may limit the ability to perform in-system debugging.
- **Testpoints** - At a minimum, testpoints should be provided for the debug interface signals C2CK, C2D, GND and the UART signals TX and RX.
  Design Example: The CP2201 Evaluation Board does not have a programming header, however, it supports in-system debugging by providing testpoint access to the C2 Interface debug signals. The testpoints are located close to the edge of the board to allow large sized clips to easily connect to the debug signals without installing actual testpoints. A custom cable was built that converts the 10-pin USB Debug Adapter cable to 3 large clips that connect to C2CK, C2D, and GND on the CP2201EB.
  Design Example: The CP2201 Evaluation Board gives testpoint access to the UART pins to allow debugging while the MCU is running. Often, if the MCU is halted to examine registers and memory, then devices waiting to receive packets from the MCU will time out. UART based debugging can allow the MCU to continue processing packets while printing system status to a UART terminal. Since packets are transmitted every 100 to 200 ms on average, the debug messages do not significantly impact the performance of the MCU.

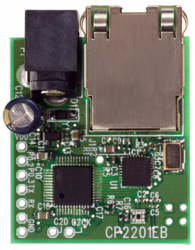

Figure 13. CP2201 Evaluation Board with Testpoint Access to C2 and UART Signals

CP220x Integration Guidelines

- **Typical Connection Diagram** - The CP220x datasheet has a typical connection diagram for both multiplexed and a non-multiplexed configurations. Please follow the typical connection diagram when designing the CP220x section of the schematic.
- **Clock** - The CP220x requires a 20 MHz ± 50 ppm clock with a 50% duty cycle. This can be derived from a crystal or from a CMOS clock source.
- **Power and Ground Pins** - All power and ground pins on the device must be connected to power or ground. The power supply should be capable of sourcing 150 mA at a supply voltage of 3.1–3.6 V. Also, be sure to provide adequate power supply decoupling.
- **Unused Pins** - If using the CP2201, there should not be any unused pins and no input pins should be left floating. If using the CP2200, only pins marked N.C. can be left floating.
- **Chip Select (/CS)** - The chip select input should be driven low when the CP220x is being accessed by the MCU. Tieing /CS to the most significant bit of the address bus A15 makes the CP220x occupy off-chip external memory addresses up to 0x7FFF. Addresses 0x8000 to 0xFFFF may be used for adding other devices (such as an external RAM or Flash) to the external memory bus.
- **ALE** - The ALE output of the MCU must be connected to the ALE input of the CP220x if using the multiplexed bus mode. It is not required if using the non-multiplexed bus mode.
- **Reset Pin** - We recommend placing a 4.75 K pull-up resistor on the reset pin. This is both an input and an output for the CP220x.
- **Interrupt Pin** - The interrupt pin is optional if writing a polled mode driver. The TCP/IP Library uses an interrupt driven driver, therefore, this signal is required if the TCP/IP Library is used.
- **MOTEN/MUXEN** - The MOTEN pin enables the Motorola bus format when tied high. This pin should be tied to ground when using a Silicon Laboratories MCU. The MUXEN pin enables multiplexed mode on the CP2200 and is not available on the CP2201.
- **Link, Activity, Link/Act LED Drivers** - The CP2200 has two LED drivers: Link and Activity. The CP2201 has a single LED driver that turns the LED on when there is a link and blinks it when there is both link and activity. The LED drivers have push-pull outputs. See the CP220x datasheet for an LED control example.

Layout Considerations:

The following guidelines should be followed when laying out the hardware. Figure 14 shows an example PCB layout for the CP2201.

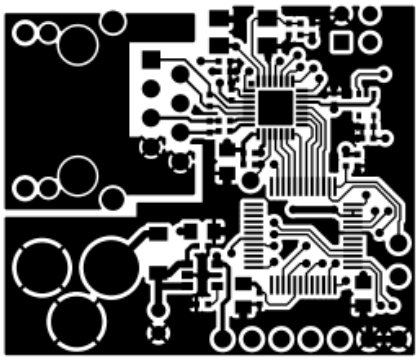

Figure 14. CP2201 Evaluation Board Layout

- The traces used for the parallel memory interface should be matched. The difference in propagation delay through the address/data, /RD, /WR, /CS, and ALE signals must not vary by more than 5 ns.
- The CP220x should be located close to the Ethernet connector.
- The crystal should be located within 1 inch of the CP220x.
- The traces used for TX+/TX– should be short, thick, matched and run on the same side of the PCB. These traces carry current, therefore, using thick traces minimizes signal loss.
- The traces used for RX+/RX– should be short, thick, matched, and should run on the same side of the PCB (if possible).
- The TX+/TX– and the RX+/RX– traces should not have 90 degree corners and should be shielded by the ground plane (if possible).
- The Ethernet connector and the rest of the system should have separate ground planes. The Ethernet connector's ground plane can be connected to the connector chassis.

6. Software Generation

In this step of the system design process, we will be generating the software that interacts with the CP220x to provide the embedded system with Ethernet connectivity. Figure 15 shows the software generation flow. Using the TCP/IP Configuration Wizard, this step is one of the easiest steps in the entire system design process. The TCP/IP Configuration Wizard is available for download from www.silabs.com/ethernet and is also included in each MCU development kit.

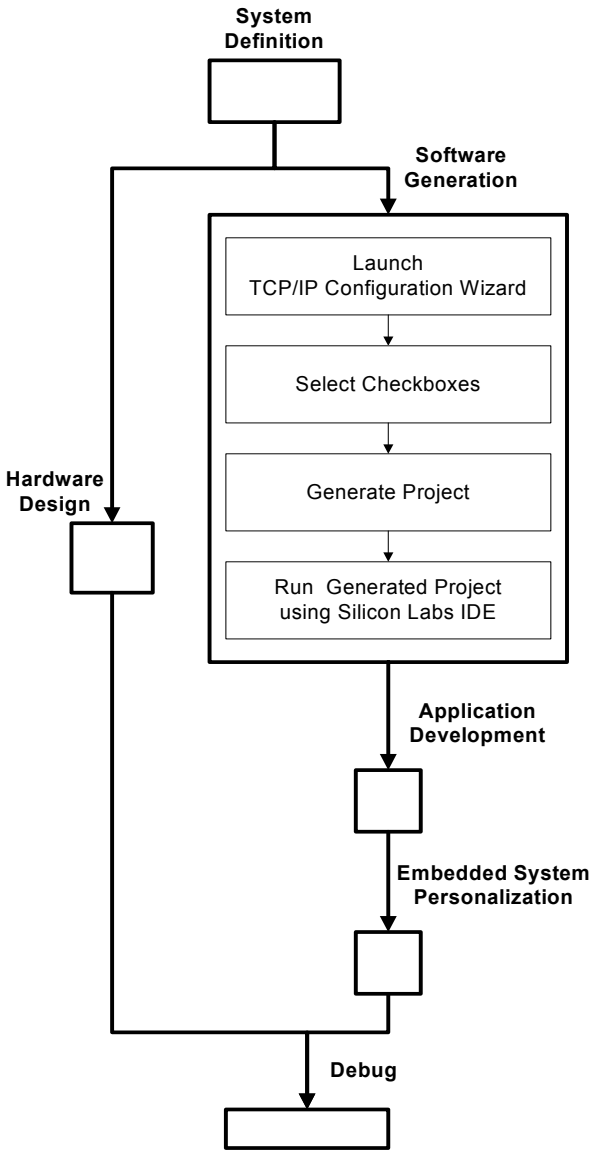

Figure 15. Software Generation Flow

6.1. TCP/IP Configuration Wizard

When the TCP/IP Configuration Wizard is first launched, the user is presented with an option tree as shown in Figure 16. Clicking on an option will show additional settings in the right-hand pane. In Figure 16 the TCP option is highlighted; therefore, the settings pane shows the protocol settings for TCP. Enabling the checkbox next to an option adds support for that option to the custom library that will be generated.

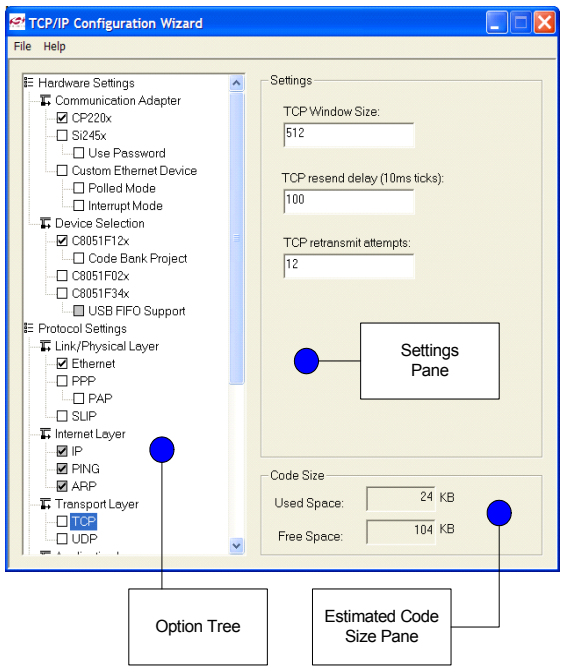

Figure 16. TCP/IP Configuration Wizard

The checkboxes needed for your application come directly from the protocol list generated from the system definition. Recall Table 1 on page 7 which maps required protocols to items in the system definition. Please have your protocol list handy as we describe the TCP/IP Configuration Wizard option tree.

6.2. Selecting Checkboxes

The TCP/IP Configuration Wizard has three main categories of checkboxes: *Hardware Settings*, *Protocol Settings*, and *System Settings*. A full listing of all available options is shown in Figure 17.

6.2.1. Hardware Settings

The first group of checkboxes labeled *Communication Adapter* select the device driver that will be added to your custom library. The default selection is the CP220x. The TCP/IP Configuration Wizard also allows you to generate a library that supports the Si245x dial-up modem or provide your own custom driver.

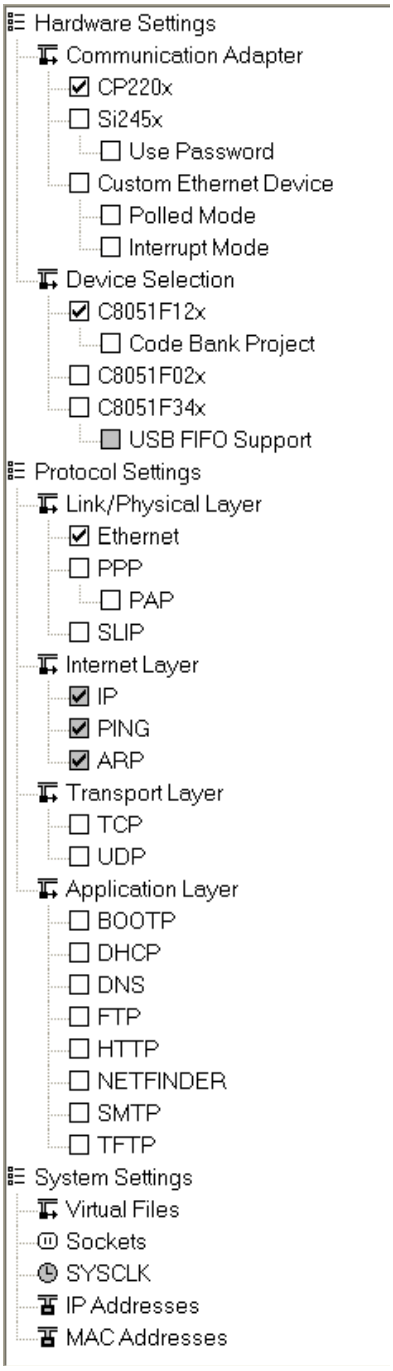

Figure 17. TCP/IP Configuration Wizard Options

The second group of checkboxes labeled *Device Selection* is used to generate a TCP/IP project and framework code. The framework code includes initialization routines that configure the custom TCP/IP Library and initialize the MCU for communication with the CP220x. The checkbox selection for this group must match the MCU being used to ensure that the proper initialization code is generated.

6.2.2. Protocol Settings

The first three checkbox groups under *Protocol Settings* are automatically filled in by the TCP/IP Configuration wizard. The user may choose to select advanced settings for these protocols or leave them at their default values. Let us now skip down to the *Application Layer* checkboxes, where we will use our required protocol list to specify our requirements.

You may notice that some protocols in the option tree are not listed in Table 1 on page 7. These protocols are more advanced and are described in detail in the appendix on page 26. Application Note “AN237: TCP/IP Library Programmer’s Guide” also describes how these protocols can be used in your embedded system.

6.2.3. System Settings

All customization options under *System Settings* are optional. If the system will be using static network configuration, a default IP address and subnet mask may be assigned to the embedded system now, or may be assigned in *mn_userconst.h* after the project is generated.

6.3. Generating a Project

After the protocols required to meet your system definition have been selected, it is now time to generate the custom TCP/IP Library and supporting project files. In the TCP/IP Configuration Wizard, press the File→Generate Project command as shown in Figure 18.

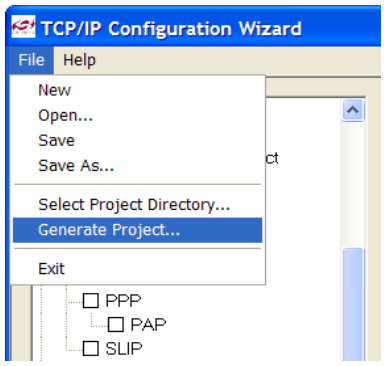

Figure 18. Generate Project Command

6.4. Running the Generated Code

The TCP/IP Configuration Wizard generates a TCP/IP Library and supporting code for execution on an MCU target board with an attached CP2200 Ethernet Development Board (AB4). This allows immediate evaluation of the wizard's output and provides a stable platform for software development. The AB4 board is compatible with the 'F12x, 'F34x, and 'F02x Target Boards.

Once the project has been generated, it may be managed in the Silicon Laboratories IDE as shown in. A step-by-step tutorial which shows how to manage the generated project in the Silicon Laboratories IDE can be found in the Embedded Ethernet Development Kit User's Guide. A detailed description of the generated project files can be found in "AN237: TCP/IP Library Programmer's Guide."

After testing the code on a development board, the code may be ported to your hardware by changing the initialization routines. The following items need to be modified to run the software on new hardware:

- **External Memory Interface Initialization** - The EMIF_Init() routine should be modified to configure the MCU into the correct duplex mode.
- **Port Input/Output Initialization** - The Port_Init() routine should be modified to specify the location of the interrupt pin and configure the input and output mode of each pin.
- **Reset Pin Control Routines** - The ether_reset_high() and ether_reset_low() routines should be modified such that the reset pin of the CP220x can be controlled by the library.

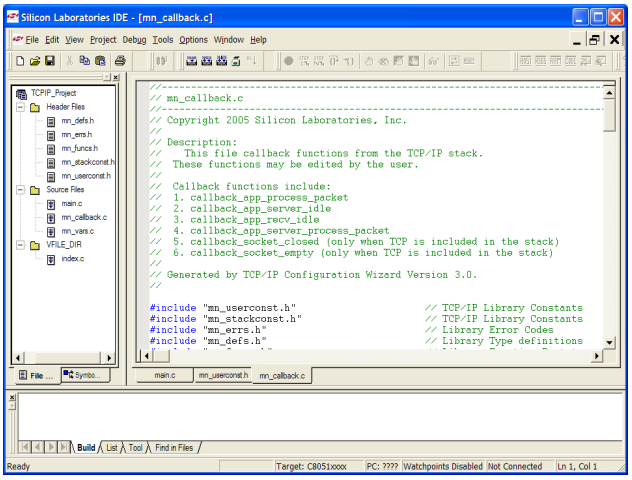

Figure 19. Project Management Using the Silicon Laboratories IDE

7. Application Development

The TCP/IP Configuration Wizard generates the framework code for basic network functionality. We will now start developing the application code that gives the embedded system its required functionality. Figure 20 shows the application development flow.

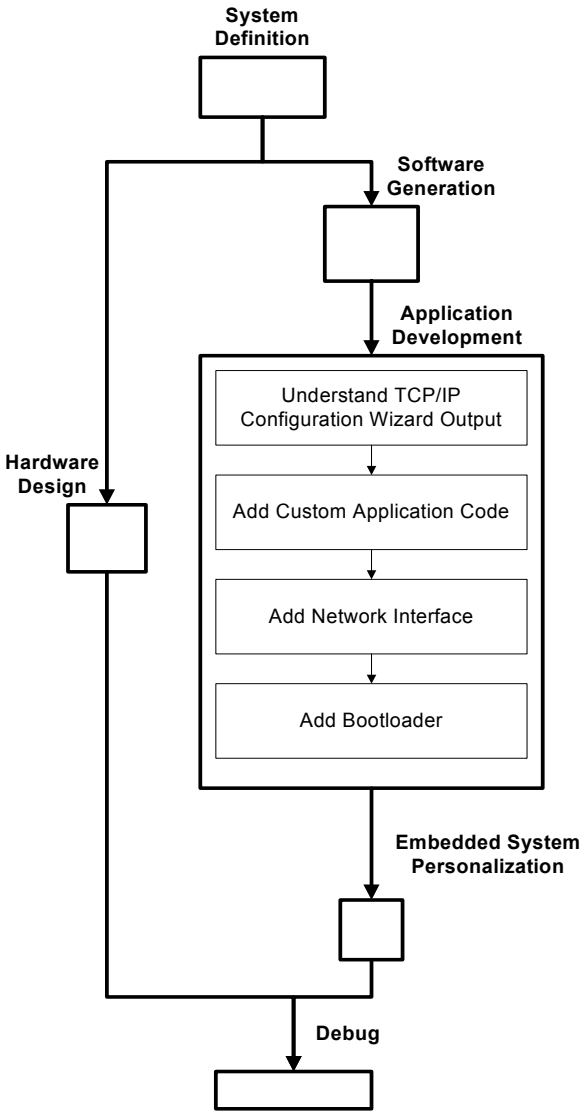

Figure 20. Application Development Flow

7.1. Application Structure

The application code that will implement the required system functionality specified in the first question of the system definition must co-exist and share resources with the TCP/IP Library. To develop the code, we will need a good understanding of how the TCP/IP Stack operates.

Figure 21 shows the main application loop for projects generated using the TCP/IP Configuration Wizard. On reset, the MCU is initialized then an attempt is made to establish a network connection. Once the embedded system is on a network, the mn_server() library routine is called and typically does not exit unless the device is disconnected from the network.

The mn_server() routine handles many network tasks as shown in Figure 22. This includes automatically responding to web server, ping, and virtual file system requests.

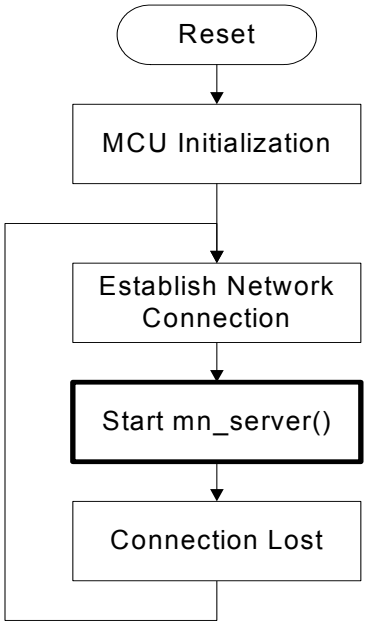

Figure 21. Main Application Loop

7.2. Adding Application Code

Application code can be inserted in three places as shown in Figure 22. Each of these 'application code holders' are described in the paragraphs below.

7.2.1. Interrupt Service Routines

The MCU has multiple interrupt sources including external pin, timer overflow, ADC end-of-conversion, etc. Application code that requires accurate timing, such as an ADC sampling engine, should be placed inside a high or low priority interrupt service routine. Note that the TCP/IP library uses a low priority interrupt for communication with the CP220x.

7.2.2. Callback Functions

The TCP/IP Library uses callback functions to notify application code of certain events such as packet received, server idle, waiting to receive packet, etc. Application code that interfaces with the TCP/IP Library or does not require accurate timing should be placed inside a callback function. Application Note "AN237: TCP/IP Library Programmer's Guide" has a complete description of all available callback functions.

7.2.3. Common Gateway Interface (CGI) scripts.

The Common Gateway Interface (CGI) is a standard protocol for communication between a web browser and a web server. A CGI script is a firmware routine residing on the embedded web server that may be executed by clicking on a link or button inside a web browser.

CGI scripts are capable of accepting data from a web browser and returning dynamically created web pages for display in the web browser.

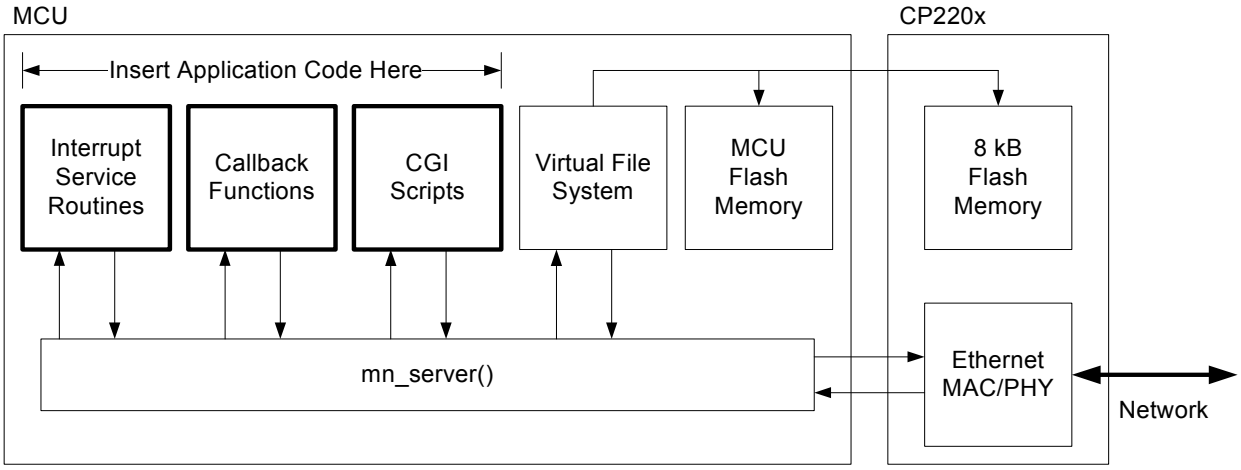

Figure 22. Application Code Model For MCU Firmware After mn_server() is Started

7.3. Developing a Web Browser Interface

The application code added to the embedded system should allow it to meet the required system functionality specified in question 1 of the system definition. We will now develop an example system and demonstrate how to build a network interface.

Our example system will provide remote monitoring capability for a light sensor. A block diagram of the example system is shown in Figure 23.

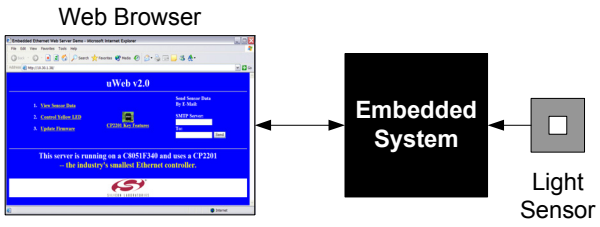

Figure 23. Example System Block Diagram

The software for this system can be generated by the TCP/IP Configuration Wizard. The only required application layer protocol is HTTP. After generating the basic project, we have added an interrupt-based ADC sampling engine. From this point, we will assume that we always have access to a global variable called *ambient_light*.

7.3.1. Creating Common Gateway Interface (CGI) Scripts

Our goal in this section is to view the *ambient_light* variable from a web browser. To exchange data with a web browser, we will need to use the common gateway interface. Thanks to the TCP/IP library, this task is as simple as placing application code inside an empty function stub.

The first step to creating a CGI script is to create a new function stub as follows:

```
// Prototype:
void get_data(PSOCKET_INFO socket_ptr);

// Definition:
void get_data(PSOCKET_INFO socket_ptr)
{
   // Insert application code here.
}
```

To make our new CGI script visible to a web browser, we must add it to the virtual file system. The following code adds the CGI script to the virtual file system:

```
void main(void)
{
   // Initialization Code
   ...

   // Add CGI Script to Virtual File System
   mn_pf_set_entry(
       (byte*)"get_data",
       get_data
   );

   // Start mn_server()
   ...
}
```

```
Note: (byte*) "get_data" is the string which
the web browser will use to call our CGI
script.

Note: <get_data> is a function pointer to our
new CGI script.
```

The last step to creating our new CGI script is making sure we have enough empty slots in the virtual file system to add our new CGI script. Open the *mn_userconst.h* header file and scroll down to the *num_post_funcs* constant. This value should be greater than or equal to the number of CGI scripts added to the file system.

7.3.2. Adding Application Code to a CGI Script

Our new CGI script should now be executable from a web browser. To test this functionality, we can place a breakpoint on the CGI script and call it from a web browser. Assuming our embedded system has an IP address of 10.10.10.163, we can call the CGI script from a web browser as follows:

```
http://10.10.10.163/get_data?
```

This should cause code execution to stop at the breakpoint. To pass data to the CGI script, we can call it from a web browser as follows:

```
http://10.10.10.163/
get_data?type=html&setbgcolor=yes
```

When text is passed after the question mark following the IP address and script name, it is automatically copied by the TCP/IP library to the global *BODYptr* buffer. The size of this buffer can be set by modifying the *body_buffer_len* constant in *mn_userconst.h*.

The TCP/IP library provides the mn_http_find_value() function to parse the incoming data. Application code can parse the information in the *BODYptr* as shown in the following example:

```
byte msg_buff1[52];
byte msg_buff2[52];

// Search for the "type" field and store the
// result in <msg_buff1>.
status1 = mn_http_find_value (BODYptr,
    (byte*)"type", msg_buff1);

// Search for the "setbgcolor" field
// and store the result in <msg_buff2>.
status2 = mn_http_find_value (BODYptr,
    (byte*)"setbgcolor",msg_buff2);

// Check status1 and status2 to determine if
// msg_buff1 and msg_buff2 are valid.
if(status1 && status2){
    ...
}
```

Using the data passed in *msg_buff1* and *msg_buff2*, application code can perform an application-specific task, then fill a memory buffer with data it wishes to return to the web browser.

7.3.3. Sending a Web Page to the Web Browser

Once we have received the browser's request, we can generate a web page containing the value of *ambient_light* and return it to the web browser. We do this with the following code:

```
static byte html_buffer[256];

// Write the HTML code to a buffer.
sprintf( html_buffer, "<HTML>%i</HTML>",
    ambient_light);

// Fill the socket with data to send.
socket_ptr->send_ptr = html_buffer;
socket_ptr->send_len = strlen(html_buffer);

// Return from the CGI script
return;
```

The above code makes a very simple HTML page containing the value of *ambient_light* and stores it in a buffer. The buffer is sent to the web browser using the socket pointer provided by the TCP/IP library as the first parameter in the CGI script. A socket is a data structure that allows application code to send and receive data using the TCP/IP library.

Data is sent using the socket by specifying its starting address and length. As soon as the CGI script returns, the TCP/IP library will check the socket's *send_len* field for a value greater than zero and send the html page to the web browser. Note that the temporary HTML buffer must be a global variable because it is accessed after the CGI script returns.

7.4. HyperTerminal (Telnet) Interface

We will now add a Telnet interface to our example embedded system to allow access from Telnet clients such a HyperTerminal, PuTTy, Microsoft Telnet, etc. The TCP/IP library provides callback functions which make implementing a Telnet interface very easy.

The TCP/IP library functions used for implementing a Telnet server interface are:

- *mn_open()*
- *callback_app_server_process_packet()*
- *callback_app_server_idle()*
- *mn_send()*
- *callback_socket_closed()*
- *mn_close()*

7.4.1. Starting the Embedded Telnet Server

The first step to creating a Telnet server is opening a passive TCP socket at port 23, the well known port number for Telnet. This can be done using the *mn_open()* routine. A passive socket means that it will wait for a client to connect, rather than actively trying to establish a connection. It is the most suitable type of socket for implementing a server application.

Once the user starts a Telnet session (e.g. by pressing the "connect" button in Hyperterminal), the Telnet client will attempt to establish a TCP connection with the embedded Telnet server. If the connection attempt is successful, the TCP/IP library will alert application code using the *callback_app_server_process_packet()* callback function. Below is an example of application code placed inside this callback function:

```
callback_app_server_process_packet
    (PSOCKET_INFO socket_ptr)
{
    // Check if the incoming packet
    // was addressed to Port 23
    if(socket_ptr->dest_port == 23){

        if(TELNET_STATE == WAITING){
            // Change Telnet State Variable
            TELNET_STATE = CONNECTED;

            // Display Welcome Message
            mn_send(telnet_socket_no,
                TELNET_WELCOME_STR,
                sizeof(TELNET_WELCOME_STR));
        }
    }
}
```

The mn_send() routine used in the above example can be used to send data back to the Telnet client that has just established a connection.

7.4.2. Communication During the Telnet Session

Now that the Telnet connection is fully working, either the server or the client may send data to the other device. The client sends data to the embedded system when the user types characters into the keyboard, and the embedded system may update the client's screen using the *mn_send()* routine. The two callback functions used in the "connected" state are:

- *callback_app_server_process_packet()* - called by the TCP/IP library when a packet is received from the client. Application code can access the received data by reading from *socket_ptr->recv_ptr* up to *socket_ptr->recv_len* bytes.
- *callback_app_server_idle()* - periodically called when the TCP/IP library is not sending or receiving packets. Application code, including calls to *mn_send()*, may be placed in this callback function.

7.4.3. Ending a Telnet Session

The Telnet session may be ended by either the client or the embedded server. The Telnet session is considered closed when either end of the TCP connection is lost. On the embedded server, the TCP connection may be closed by calling the *mn_close()* routine. The Telnet client closes the TCP connection when the user clicks the "disconnect" button or the window is closed.

If the Telnet client closes the connection, application code is notified using the *callback_socket_closed()* callback function. In case the client loses power and is unable to cleanly close the TCP connection, application code may not receive notification that the connection has been lost. To handle this case, application code may implement a timeout.

7.4.4. Data Rate Considerations

Since Telnet uses the TCP transport protocol, each packet sent must be acknowledged by the receiver before further packets may be sent. This causes the transfer rate to be dependant on how fast the receiving device can acknowledge a packet.

If the Telnet connection uses uni-directional data flow, then it may be affected by a congestion control algorithm called TCP delayed acknowledgement. This algorithm causes most PCs to withhold packet acknowledgement until it needs to send data back to the embedded system or a 200 ms timeout expires. This phenomenon limits the data rate of uni-directional TCP traffic to 5 packets per second. Most applications such as Web Server, E-mail, etc. that use bi-directional data flow are not affected by this phenomenon.

More information is available from Microsoft Knowlegebase Article 214397 available from http://support.microsoft.com/.

7.5. Transferring Data By Email

E-mail transmission across the Internet requires a network infrastructure. The network infrastructure for e-mail consists of outgoing mail servers (SMTP) and incoming mail servers (POP, IMAP, HTTP). The incoming and outgoing mail servers handle the routing and queuing of e-mails as they are sent across the Internet.

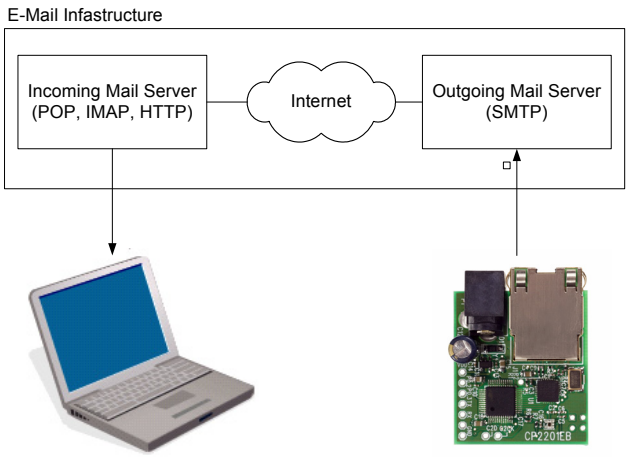

Figure 24. E-Mail Transmission

The TCP/IP stack implements the SMTP protocol which allows the MCU to communicate with an outgoing mail server. Several pieces of information are required for an MCU to send e-mail. These are:

- IP address of the SMTP server.
- Destination e-mail address (TO field).
- Return e-mail address (FROM field).
- Subject.
- Message Body.
- Attachment Name.
- Attachment Contents.

Each of the information fields above can be stored in dynamic RAM buffers or in static code constants. The only restriction on the information in the buffers is that it must be plain text (ASCII). Binary files such as images can only be sent if the MIME protocol is used to convert the binary attachment to ASCII.

If an SMTP server is not available on your network, then software SMTP servers, such as the PostCast server available from www.postcastserver.com allow any PC on the network to become an SMTP server.

A firmware example of sending e-mail is located in the Ethernet examples folder. For a typical IDE Installation, this folder's path is *"C:\Silabs\MCU\Examples\C8051F...\Ethernet.*

7.6. Custom Application Interface

The TCP/IP Stack allows opening TCP and UDP sockets for communication with custom applications. The firmware required to implement the custom interface is very similar to a Telnet interface. If the custom application chooses to use UDP, then the maximum data transfer rate will increase and the application may use broadcast packets; however, the reliability and connection-oriented nature of TCP will be lost. See the Appendix on page 26 for a detailed comparison between TCP and UDP.

Timesaving Tip: If a listening UDP socket receives a packet from a device, the socket is automatically bound to the IP address and port number of the sender. This means that the embedded system may only send packets to and receive packets from the device that has sent it a packet. To allow packet transmission and reception to/from other devices, the socket must be reset by closing and re-opening it. See Application Note "AN237: TCP/IP Library Programmer's Guide" for a complete description of the TCP/IP Library API.

7.7. Running without a Network

The TCP/IP Library can detect when the CP220x has been disconnected from a network and causes the *mn_server()* routine to exit. In most systems, the library will immediately be re-initialized using *mn_init()* then software will enter the *establish_network_connection()* routine. This routine does not exit until a network connection has been established.

The *establish_network_connection()* routine can be customized to perform specific system functionality while waiting for a network connection.

7.8. Managing RAM

On devices with 4 kB RAM, the RAM usage should be managed as the application is being developed to ensure that the RAM usage does not exceed the amount of physical RAM on the device. If using the 'F340 MCU and not utilizing the USB interface, the USB FIFO RAM may add up to 1 kB of additional RAM for use by the TCP/IP Library or application code.

All buffers used by the TCP/IP Library are adjustable by the user. The buffer sizes can be configured in the *mn_userconst.h* header file.

Timesaving Tip: DHCP requires a large RAM buffer (548 bytes) for acquiring and renewing the embedded system's IP address. Since this buffer is used only during initialization and seldom otherwise, some of the RAM can be temporarily recovered. For example, in the CP2201EB, the last 504 bytes of the DHCP buffer (starting with the *sname* field of the *dhcp_info* structure) are used as a general purpose buffer for dynamically creating web page content using *sprintf()*.

7.9. Saving Data to Flash

If the system definition requires storing a static IP address in Flash, either the Flash on the MCU or the Flash on the CP220x may be used. Note that using the CP220x Flash to store an IP address or other data will allows more of the MCU's Flash to be used as executable program memory.

Application Note "AN201: Writing to Flash from Firmware" provides pre-written routines for writing the MCU's Flash. The TCP/IP library also provides routines for reading and writing the CP220x Flash. The TCP/IP Library API can be found in "AN237: TCP/IP Library Programmer's Guide."

7.10. Implementing a Network Bootloader

To update firmware over the network, a TFTP bootloader can be used to transfer a new firmware image to the embedded system. To use a bootloader to update firmware, a special build of the TCP/IP Library is required. This library is located at address 0x2400 in code space to allow room in Flash for a bootloader.

The special libraries can be found in a folder named "ExtraLibraries" typically found in *C:\SiLabs\MCU\TCP-IP Config\ExtraLibraries*.

An example project which uses the bootload libraries is located in *C:\SiLabs\MCU\Examples\C8051F34x\Ethernet\CP2201EK_SOURCE\CP2201EK_AB4_BL*.

For more information about the bootload libraries, please see the MCU Knowlegebase, available from www.silabs.com/support.

8. Personalizing the Ethernet Enabled Embedded System

Now that the application has been developed, it is time to add personalized content (e.g. web pages, images, etc.) to the embedded system. The two primary areas that require customization are network configuration and web server interface. Figure 25 shows the embedded system personalization flow.

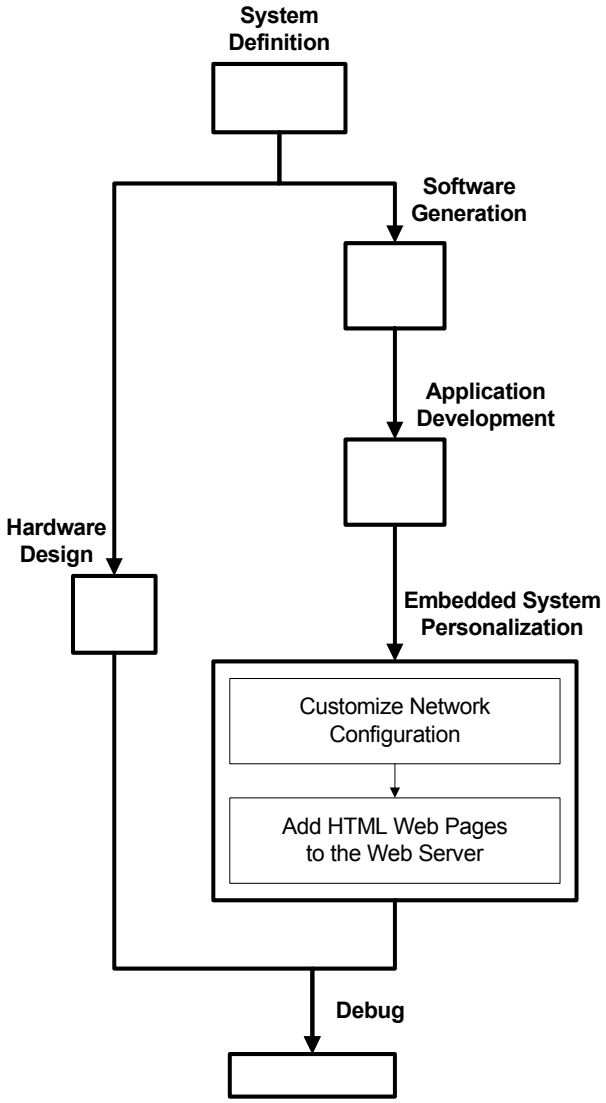

Figure 25. Embedded System Personalization Flow

8.1. Customizing Network Configuration

If the embedded system has Netfinder enabled, then the TCP/IP Configuration Wizard will generate two additional files in the project directory: *netfinder.c* and *netfinder.h*. These files may be used as-is or modified to suite the application requirements.

Figure 26 shows a screenshot of the Netfinder utility after finding an embedded system that has been loaded with the default *netfinder.c* and *netfinder.h* files. The following fields can be customized by modifying the customization strings in *netfinder.h*:

- Device Name.
- Text Description.
- Definition of Event 1 (e.g. Time Powered).
- Definition of Event 2 (e.g. Time on Network).

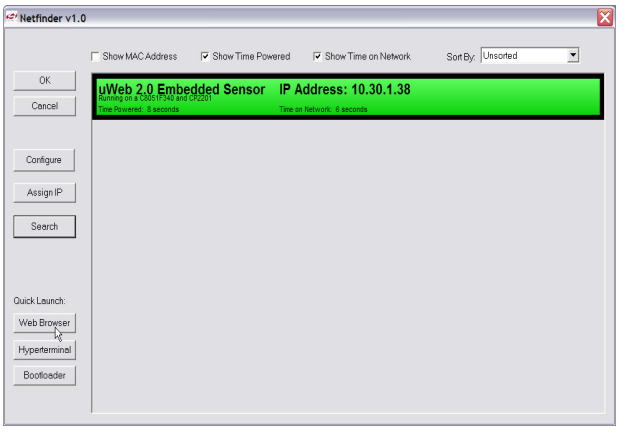

Figure 26. Netfinder Utility Screenshot

If Netfinder capability is needed in your system, you may use the dedicated Netfinder utility or integrate Netfinder functionality in your own custom application. “AN237: TCP/IP Library Programmer’s Guide” provides the information required to search for Netfinder-enabled devices on a local area network.

8.2. Customizing the Web Server Content

The TCP/IP Configuration Wizard generates a basic “Hello World” web page when web server functionality is included in the generated library. The single-page “Hello World” website can be modified as shown in the Embedded Ethernet Development Kit User’s Guide or a multi-page website with images and javascript can be developed and stored in the embedded system.

8.2.1. Adding Web Pages and Images

Recall our example embedded system described in Section 7.3 on page 18. We will now develop and add web pages to display our light sensor data inside a web browser.

We will be designing the HTML pages on a PC and adding them to the embedded system using the procedure illustrated in Figure 27 and summarized below:

1. Develop the HTML content and preview on a PC. Make a note of the file sizes as you are developing the content. An image that is 3 kB on the PC will consume 3 kB of Flash memory.

2. Use the HTML2C utility to convert the HTML content to file arrays. For each file array, the HTML2C utility generates a (.c) source file and a (.h) header file.
3. Include the header file at the beginning of *main.c*.
4. Add the source file to the project and to the project build.
5. Add each file array to the virtual file system using the mn_vf_set_entry() function. See “AN237: TCP/IP Library Programmers Guide” for more details.
6. Modify the *num_vf_pages* constant in *mn_userconst.h* such that the value is greater than or equal to the total number of files arrays added to the file system.

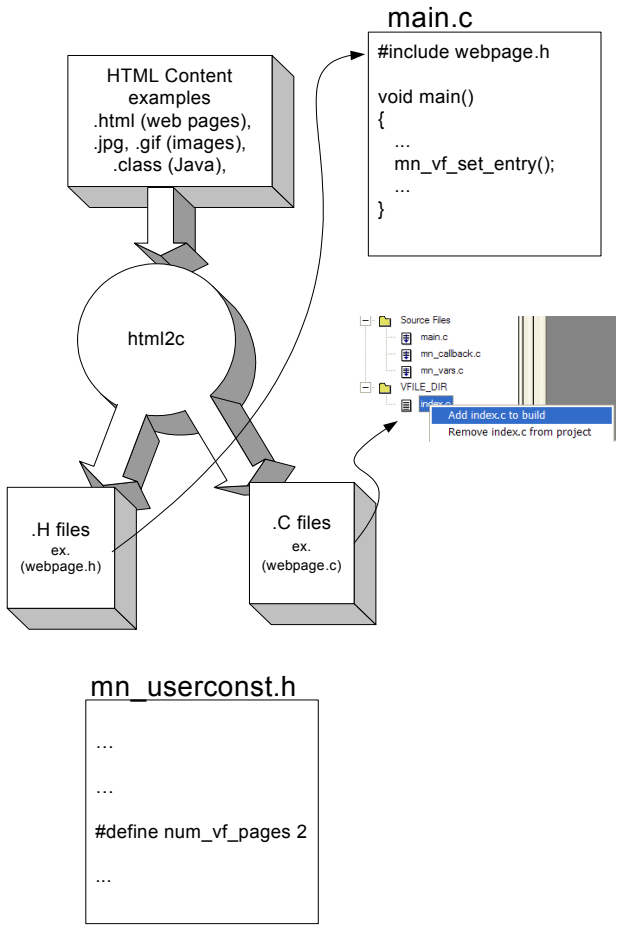

Figure 27. Adding HTML Content

For additional information, the Embedded Ethernet Development Kit User’s Guide has a step by step tutorial which shows how to use the HTML2C utility.

8.2.2. Creating Basic HTML Content

HTML stands for Hyper Text Markup Language and is a file format used to specify web page content. The HTML language uses tags tell the web browser how to display a web page. Below is an example of a simple web page:

```
<html>
    <!-- HTML comments take up code space -->
    <head>
       <title>Hello World</title>
    <head>

    <!-- Whitespace takes up code space -->
    <body bgcolor="green">
       <h1>Hello World!</h1>  <br><br>
       <B>This page is served from a
       C8051F12x and uses the Silicon
       Laboratories TCP/IP stack.</B>
    </body>
</html>
```

Designing HTML content will require some knowledge of how web pages are created. If you are not familiar with the HTML language used to compose web pages, you can use applications such as Frontpage or Microsoft Word to generate web pages. This method is not preferred because the output of such applications is not optimized and quickly fills up code space.

If you are willing to learn HTML, there are excellent tutorials on the Internet. Below are some links to websites that have HTML tutorials:

HTML Tutorials and Examples:
http://www.w3schools.com/html/default.asp*
http://www.htmlgoodies.com/primers/html/
http://www.pagetutor.com/

* Recommended as a starting point.

8.2.3. HTML Frames - A page within a page.

Now that we have a basic web page, we will add a display showing the light sensor data. Since our CGI script returns an HTML page, we will create a frame inside our main HTML page to display the page returned from our CGI script. Below is an example of adding an inline frame to a web page:

```
<html>
    ...

    <body bgcolor=”green”>

       <!-- Add an inline HTML frame -->
       <iframe src="get_data?type=html">
       </iframe>
    </body>

</html>
```

Notice that in the *<iframe>* tag, we have specified the relative path of the CGI script, which will provide the

content for the new frame. Example usage of inline frames can be found in the CP2201EK.

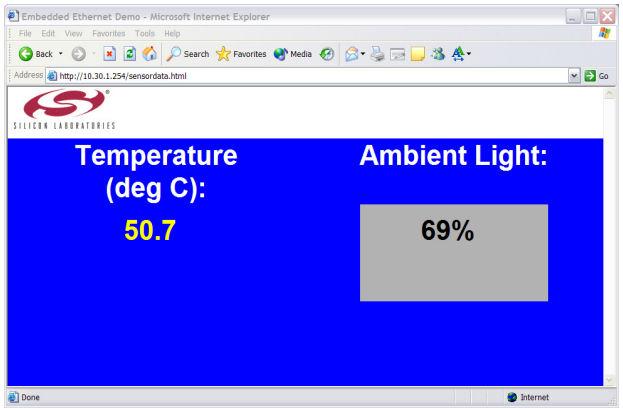

Figure 28. Displaying Data Inside a Frame

8.2.4. Using Javascript to Automate a Web Page

An inline frame is a good way to display sensor data; however, the user must refresh the web page to update the display. After several minutes, this can become very tedious. The solution... Javascript.

Javascript is a browser scripting language that is used to automate common browser tasks. The CP2201EK uses Javascript to refresh the sensor data displayed inside the inline frames. The javascript code used for performing these operations in the CP2201EK can be examined by selecting the "view source" command in the web browser. Below is a simple example of refreshing a frame using Javascript:

```
<html>

   <head>
      <script type="text/javascript">
      var delay = 2000;
      function refresh()
      {
         document.getElementById("frame1")
            .src="get_data?type=html";
         setTimeout("refresh()", delay);
      }
      </script>
   </head>

   <body bgcolor="green" onload="refresh()">

      <!-- Add an inline HTML frame -->
      <iframe id="frame1" src="about:blank">
      </iframe>
   </body>

</html>
```

The webpage containing this Javascript example has a single blank *<iframe>* and a single Javascript function named *refresh()*. Once the HTML page is loaded by the web browser, the *refresh()* function is executed because we have added the *onload = "refresh()"* statement to the <body> tag.

The *refresh()* function performs two tasks. First, it forces a reload of the frame contents by setting its *src* field to the relative path of the CGI script.

Second, it starts a timeout of *delay* milliseconds by calling *setTimeout(function, time)*. This Javascript function allows a specified *function* to be called after the specified *time*. In this example, each call to *refresh()* triggers a new call to *refresh()* 2 seconds later. This allows the sensor data to be continuously refreshed every 2 seconds.

Below are some links to websites that have additional Javascript tutorials and examples:

Javascript Tutorials and Examples:
http://www.w3schools.com/js/
http://www.htmlgoodies.com/beyond/javascript/
http://www.webteacher.com/javascript/

8.2.5. Collecting Data Using HTML Forms

HTML forms are another web page construct that may be used for sending data to a CGI script. An HTML form collects data in text fields on a web page and passes it to the embedded system by calling a CGI script. Below is an example of an HTML form:

```
<form action="get_data" target="_blank">
   SMTP Server:<br>
   <input type="text" name="server"><br>
   To:<br>
   <input type="text" name="to">
   <input type="submit" value="Send">
</form>
```

Figure 29 shows this form inside an HTML page. When the user presses the Send button (input type = submit), the *get_data* CGI script will be called as follows:

```
/get_data?server={text}&to={text}
```

The fields labeled *{text}* come from text entered by the user inside the HTML page. The *target = "_blank"* statement causes the data returned from the embedded system to be displayed on a new page.

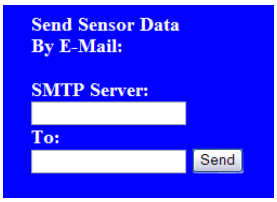

Figure 29. HTML Form

9. Debugging Embedded Ethernet

Occasionally when developing an Embedded Ethernet system, a problem is encountered. Debugging and finding the cause of the problem can be simplified if the problem can be isolated to specific part of the system. There are four conditions that may be checked that help isolate the problem to a specific piece of the system. They are illustrated in Figure 30 and described below:

- Is the PC sending the correct data?
- Is the embedded system receiving the correct data?
- Is the embedded system transmitting a response?
- Is the PC receiving the embedded system's response?

On the PC side, a packet capture utility such as Ethereal can be used to view network traffic being received and transmitted by the PC. Ethereal is a widely used open-source utility available for download from www.ethereal.com.

On the embedded system side, a UART terminal may be used to view debug messages printed by the debug version of the TCP/IP Library. Some of the messages printed by the library are "packet received", "packet transmitted", "packet skipped", "device reset", etc. An example of a problem that may be solved by these messages is a receive buffer that is sized too small. In this case, the user would continue to see "packet skipped" messages for every packet sent by the PC larger than the receive buffer size.

The debug libraries can be used by adding UART initialization code to the embedded system. This allows the library to print messages using the *printf()* library function. **The debug libraries can be found in a folder named "ExtraLibraries" typically found in *C:\SiLabs\MCU\TCP-IP Config\ExtraLibraries*.**

Example projects which uses the debug libraries are located in *C:\SiLabs\MCU\Examples\C8051F34x\Ethernet\CP2201EK_SOURCE\CP2201EK_AB4_DBG* and *C:\SiLabs\MCU\Examples\C8051F12x\Ethernet\DEBUG*.

If the embedded system does not have an RS-232 level translator, the level translator for any target board may be borrowed to debug the embedded system. Alternatively, the CP210x evaluation boards may be used to convert TTL level UART signaling to a virtual COM port on the PC.

An alternative method of debugging is halting the MCU and viewing the CP220x registers in the Silicon Laboratories IDE. The memory windows associated with the CP220x allow the user to view the direct and indirect registers, transmit buffer, receive buffer and Flash memory.

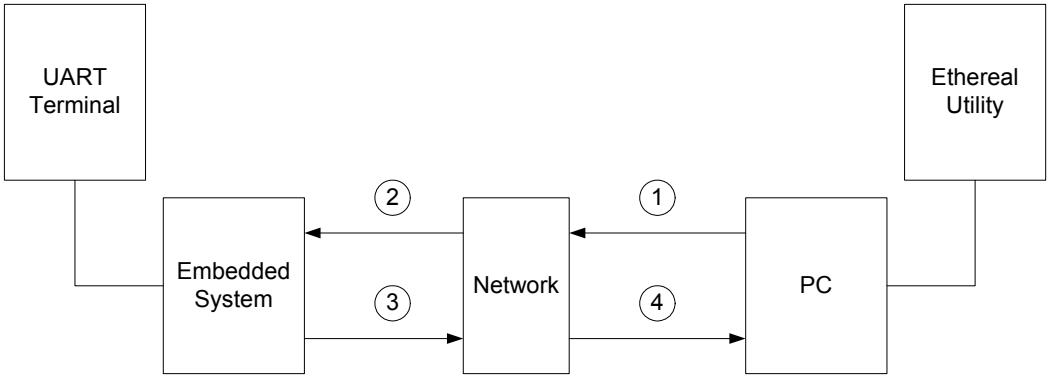

Figure 30. Embedded Ethernet Debug Setup

APPENDIX—THE BASICS OF TCP/IP

TCP/IP refers to a set of standard protocols used for communication over a network. The protocols are based on the Open Systems Interconnection (OSI) Model, a layered abstract description of network communication. The layers specified in the OSI model are structured such that each layer only depends on the layers below it. Implementations of the OSI Model, such as TCP/IP, result in a protocol stack as shown in Figure 31 which describes functionality both at the application layer and at the low level physical interface.

9.0.1. Physical Layer

The physical layer is typically implemented in hardware and can use a wide variety of interfaces. This allows networks to consist of Ethernet devices, wireless devices, dial-up modem devices, or a combination of all three.

9.0.2. Data Link Layer

The data link layer is implemented in both software and hardware. It contains the low level device driver and the Ethernet Media Access Controller (MAC). At this level, data is exchanged in IEEE 802.3 Ethernet frames using physical addressing.

The CP220x is a single-chip Ethernet controller containing the physical and data link layers, buffer space, and 8 kB Flash memory pre-programmed with a unique physical address.

9.0.3. Network Layer

The network layer builds on top of the data link layer by implementing logical addressing. Logical addressing allows devices with different data link and physical layers to communicate. It also allows devices to have a temporary address that can be easily changed or re-assigned as it moves between networks. The logical address is also called an IP Address.

Devices with different data link and physical layers can communicate using their logical addresses because of the Address Resolution Protocol (ARP). ARP is a protocol used to map the logical IP Address to a physical address. For Ethernet, the physical address is the MAC address, however, this can be different for other types of networks such as dial-up networking.

The PING protocol allows a user to check if a device assigned to a particular IP address is responding. A PING application sends a small packet to a device's logical address and measures the time it takes to receive a response. This round trip time can be used to estimate network latency.

9.0.4. Transport Layer

The transport layer builds on the network layer by dividing each IP address into 65536 ports. This allows multiple applications to run on a network node using a single IP address. A packet's full destination at the transport level includes both an IP address and a port number.

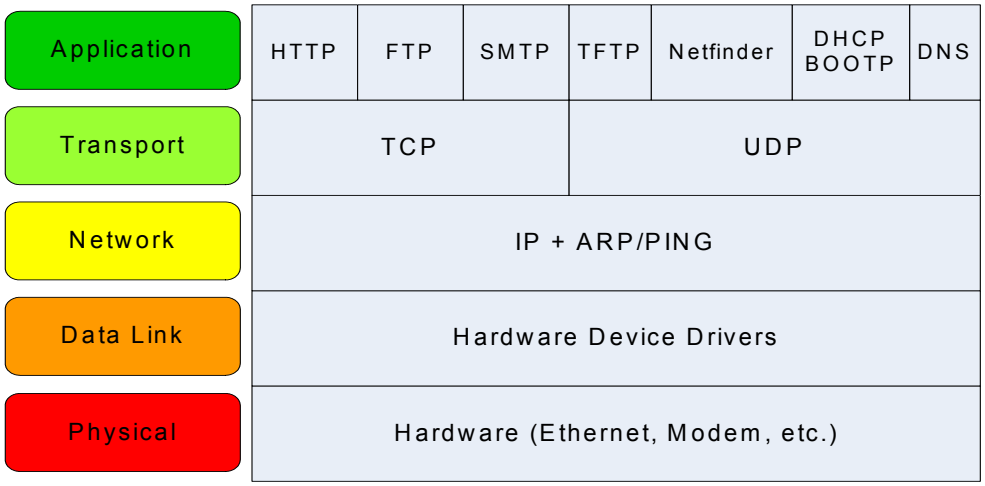

Figure 31. TCP/IP Protocol Stack

There are two types of ports (also called sockets) at the transport level: TCP and UDP. Each is described below:

- Transmission Control Protocol (TCP) - TCP is a connection-oriented protocol providing a reliable byte-stream between two devices. Data delivery is guaranteed and always arrives in-order.
- User Datagram Protocol (UDP) - UDP is a connectionless protocol providing fast, best effort datagram delivery. A single node may broadcast or multicast packets to multiple nodes.

Table 4 compares the TCP and UDP transport layer protocols. Most network nodes implement one or both of the transport layer protocols.

9.0.5. Application Layer

The application layer is the topmost protocol level and directly implements the user interface. Each user interface relies on either UDP or TCP at the transport layer. Based on this, the application layer protocols can be divided into two groups. The application layer protocols below rely on UDP:

- **Automatic Network Configuration (BOOTP/ DHCP)** - These protocols allow the embedded system to automatically acquire an IP address from the network. A DHCP/BOOTP server must exist on the network. BOOTP is an older and less efficient version of the DHCP specification but is provided for compatibility with older network hardware.
- **Netfinder** - The netfinder protocol allows a PC application to search for embedded systems on a network. When using DHCP, this saves space and hardware costs because the embedded system does not need to display its IP address on an LCD screen. Multiple embedded systems can be differentiated through an external event path. If DHCP is not used, the Netfinder protocol allows a PC application to assign a static IP address to an embedded system. This also saves space and hardware costs because static IP address assignment occurs over the network. A second interface (e.g. UART, keypad, etc.) is not required to program the IP Address.
- **Trivial File Transfer Protocol (TFTP)** - The TFTP protocol is a simple way to transfer files. It is typically used to update firmware or download configuration information from a TFTP server.

The application layer protocols below rely on TCP:

- **Hyperterminal/Telnet Interface (TCP)** - A Hyperterminal/Telnet interface is the simplest interface that can be implemented using TCP. Data is transmitted in both directions and is displayed on a terminal very similar to UART/RS-232.
- **Web Server Interface (HTTP)** - HTTP stands for Hyper Text Transfer Protocol and is used to transfer information (web pages, images, etc.) for display inside a web browser. This protocol allows an embedded system to be monitored and controlled from a web browser.
- **E-mail Interface (SMTP)** - SMTP stands for Simple Mail Transfer Protocol and is used to send e-mail messages. This interface allows the embedded system to send e-mail with or without attachments.
- **File Transfer Protocol (FTP) -** This protocol allows the embedded system to become an FTP server accessible from an FTP Client. Files may be uploaded or downloaded to the embedded system.
- **Domain Name Service (DNS)** - This protocol allows domain names such as www.silabs.com to be resolved into an IP address.

Table 4. TCP/UDP Protocol Comparison

Feature	TCP	UDP
Complexity	High	Low
Packet Delivery	Guaranteed. Uses acknowledgements and retransmits lost packets.	Best Effort. Lost packets are not retransmitted.
Speed	Slow. Retransmitted packets and overhead affect data rate.	Fast. Very low overhead since each packet is transmitted only once.
Data Stream	A TCP connection implements a byte stream between two devices very similar to a virtual RS-232 cable.	Each UDP datagram is self-contained. Data only arrives in-order if it fits inside a single datagram.
Broadcast Capability	TCP requires two devices to establish a connection.	Any network node may broadcast/multicast a datagram to any number of devices.

CONTACT INFORMATION

Silicon Laboratories Inc.
4635 Boston Lane
Austin, TX 78735
Email: MCUinfo@silabs.com
Internet: www.silabs.com

SILICON LABORATORIES