AN EMBEDDED SYSTEM FOR

THE INFRARED CLOUD IMAGER

by

Kristie Danielle Simpson

A professional paper submitted in partial fulfillment
of the requirements for the degree

of

Master of Science

in

Electrical Engineering

MONTANA STATE UNIVERSITY
Bozeman, Montana

November, 2008

APPROVAL

of a professional paper submitted by

Kristie Danielle Simpson

This professional paper has been read by each member of the professional paper committee and has been found to be satisfactory regarding content, English usage, format, citation, bibliographic style, and consistency, and is ready for submission to the Division of Graduate Education.

Dr. Joseph A. Shaw

Approved for the Department of Electrical and Computer Engineering

Dr. Robert C. Maher

Approved for the Division of Graduate Education

Dr. Carl A. Fox

STATEMENT OF PERMISSION TO USE

TABLE OF CONTENTS

TABLE OF CONTENTS - CONTINUED

TABLE OF CONTENTS - CONTINUED

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

The Infrared Cloud Imager (ICI) is a ground-based thermal infrared imaging instrument, with a new version currently in development at Montana State University, to measure cloud cover statistics. The next-generation ICI design incorporates an embedded system for consolidating the controls and sensors for the system. Instead of having dedicated lines to each component of the system, only one connection is needed to the internal microcontroller, and then the microcontroller connects to all other components. The embedded system is capable of supporting digital sensors and controls, analog sensors, and peripherals that communicate serially. In addition, an Ethernet connection is used to communicate with the embedded system inside the ICI system enclosure. The MC9S12NE64 is a 16-bit microcontroller from the Freescale Microcontroller HCS12 family that is capable of supporting the ICI system peripheral devices and was selected for the embedded system. Firmware was developed for the embedded ICI control system that supports TCP/IP Ethernet communications, analog channels, digital channels, and serial communications, and a command set was developed for interfacing with the system peripheral devices. A software application was created to establish an Ethernet connection with the embedded system and test the system commands. A second software application was implemented to act as a serial device connected to the embedded system. The development board for the microcontroller was used to test the analog and digital channels. The software and development board were used together to demonstrate that the ICI embedded system is capable of providing access to all of the ICI system peripheral devices through a single Ethernet connection. The test results verify that the next-generation ICI design is able to successfully incorporate an embedded system for consolidating the controls and sensors for the system.

INTRODUCTION

The Infrared Cloud Imager is a thermal infrared imaging instrument currently in development at Montana State University to measure cloud cover statistics. Communications with previous versions of the ICI system required a bundle of cables with dedicated lines that connected to each component of the system. The third-generation ICI design (ICI-3) has just been implemented and incorporates an embedded system for consolidating the controls and sensors for the system. Only one connection is needed to the internal microcontroller, and then the microcontroller connects to all other components. The embedded system is capable of supporting digital sensors and controls, analog sensors, and peripherals that communicate serially. In addition, an Ethernet connection is used to communicate with the embedded system inside the ICI optics enclosure.

The Infrared Cloud Imager

Over a time period between 1999 and 2007, the Optical Remote Sensor Laboratory (ORSL) at Montana State University (MSU) and the National Oceanic and Atmospheric Administration, in collaboration with the Communication Research Lab of Japan (now the National Institute of Information and Communication Technology), have worked on developing the Infrared Cloud Imager (ICI). The ICI instrument is a ground-based thermal imaging system that can provide high-spatial-resolution cloud cover statistics [1] [2].

Motivations for Cloud Cover Studies

The motivation behind the development of an instrument for studying cloud cover statistics has been twofold. The first of these motivations has been the need to understand the impact of global cloud cover on climate. The impact on climate, in particular the role clouds play in feedback processes in the Earth's radiation budget, is not fully understood. This facilitates a need for an increase in the understanding of the relationship between cloud cover and local or global climate [1].

The second motivation, which has arisen more recently, has been the establishment of Earth-space optical communication links. These are dependent on a free-space laser link between Earth-side telescopes and on-orbit or deep-space communication platforms. The availability of a free-space laser link has a fundamental dependence on the cloud cover occurring at the Earth based telescope receiver site [1].

Cloud Detection

The ICI system is composed of a software interface and peripheral hardware. The ICI software connects to the peripheral hardware, controls the data collection, and reads back the system measurements. The software analyzes the data and displays images of thermal sky emission and products derived from these images, including cloud-presence and cloud-type maps and cloud cover statistics. The core of the ICI hardware is a thermal infrared camera that captures thermal images of the sky. The ICI software controls the routines for calibrating the camera and capturing images.

Motivations for an Embedded System

An embedded system is a device that contains a computer dedicated to a specific task or series of related tasks [3]. In this case, the computer is a microcontroller. A microcontroller is a single chip that includes a microprocessor, memory, communication ports, and the additional hardware necessary to make it a complete computer. An embedded system is usually “embedded” as part of a larger system of electronics and hardware. For this project, the microcontroller is embedded in the ICI optics enclosure with the thermal infrared camera and other ICI hardware. The microcontroller communicates with a primary control computer that is typically located several tens of meters from the optics enclosure.

Generic Control

The primary motivation for incorporating an embedded system into the ICI-3 design is to consolidate the controls and sensors for the system. The embedded system acts as a generic hub that is capable of relaying information to and from the peripheral hardware. Only one connection is needed to the internal microcontroller, which connects to all other components. The embedded system therefore eliminates the need for dedicated lines to each component of the system.

Advanced Control

In addition to generic control, the embedded system also introduces the capability of integrating advanced ICI-specific control. The ICI system’s primary optical components are enclosed in a box with an opening in the top for viewing the sky and capturing data. The opening is covered with a hatch that is connected to a motor for automatic opening and closing. If only generic digital input and outputs were available, the ICI software running on the main system computer, typically located several tens of meters from the optics enclosure, would be responsible for telling the motor to move, polling the digital hatch position sensors until the desired position is reached, and then telling the motor to stop moving.

With a smart embedded system, all of the actions necessary to move the hatch are built into the ICI optical system. In this way, it is possible for the ICI software to simply send the desired hatch position to the embedded system and let it handle the rest. Even further, the embedded system is able to monitor the weather conditions and the length of time the hatch is open and automatically close the hatch. It also can detect when a sensor has failed and move the hatch to the best approximation of the closed position.

Beyond ICI

Although designed specifically for the ICI system, the embedded system could also be used with other projects. As a hub, the embedded system does not require any knowledge of the hardware it controls. It treats the devices as generic analog or digital sensors. The same applies for the serial communications. The embedded system only needs to know how to send and receive serial data. The hatch control system is the only portion of the embedded system implementation that is specific to ICI, but the hatch controls do not interfere with the general-purpose digital input and output lines. The generic nature of the embedded system makes it easily portable to other projects at MSU, and could even be used with projects outside the university.

This paper describes the embedded system developed for the ICI-3 system between May and October 2008. This work is presented here in this introduction plus the following four chapters. Chapter 2 presents the project specifications and development environment; Chapter 3 presents a detailed description of the embedded system implementation; Chapter 4 presents the hardware and software used for testing the implementation and describes the results; Chapter 5 concludes the work conducted for this project and provides an overview of future work that would expand on and improve the embedded system implementation.

PROJECT SPECIFICATIONS AND ENVIRONMENT

The first steps taken to implement the ICI embedded system were to generate detailed project specifications, select the microcontroller, and determine the development environment. Detailed project specifications for the ICI design were used to generate a list of requirements for the microcontroller for the embedded system. These requirements were used to select the microcontroller, and then a development environment and development tools were selected to fit that specific device. Additional development tools were selected for implementing the software for testing the system.

Project Specifications

The ICI-3 design incorporates an embedded system for consolidating the controls and sensors for the system, replacing the dedicated lines from the primary computer to each component of the system with only one connection to the internal microcontroller. The ICI-3 system includes digital sensors and controls, analog sensors, and peripherals that communicate serially (Figure 1).

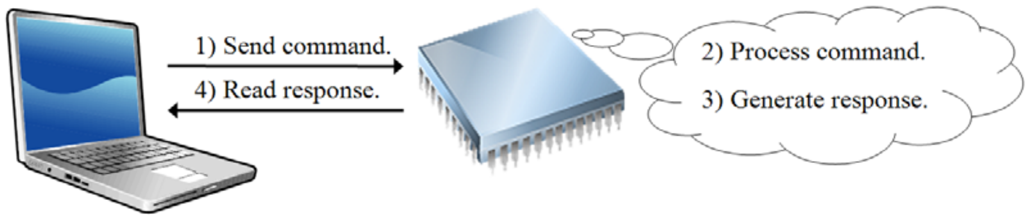


Figure 1: ICI System Functionality

ICI System Enclosure

The ICI-3 optical system is enclosed in a box with an opening in the top for viewing the sky and capturing data (Figure 2). The opening is covered with a hatch, and the hatch is opened and closed with a computer-controlled motor. The camera is fixed to a motor-driven mount that allows it to rotate to the calibration equipment for calibration, and also to rotate up to the enclosure opening to capture an image. The ICI-3 system rotates the camera to the calibration source, acquires a calibration image, rotates the camera to the hatch opening, opens the hatch, captures a sky image from the camera, closes the hatch, and then rotates the camera back to the calibration source.

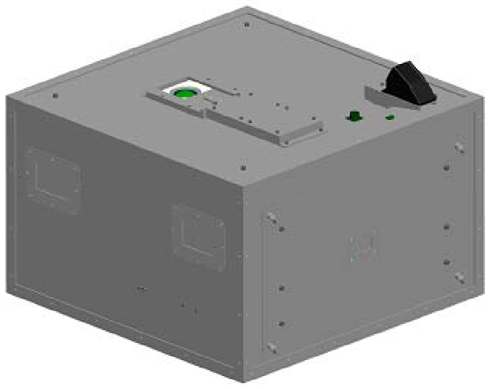


Figure 2: ICI System Enclosure with Hatch Open

In addition to the camera, calibration source, and hatch equipment, the ICI-3 optics enclosure also includes the embedded system, analog sensors, a thermoelectric air conditioner (A/C), and digital sensors (Figure 3). All peripheral devices except for the camera are connected to the microcontroller. The analog sensors measure the temperature and humidity in the areas around the camera and around the microcontroller. There is also an analog rain sensor to detect unfavorable weather conditions and prevent the hatch from opening. The thermoelectric A/C is used to control the temperature inside the system enclosure. Both the thermoelectric A/C and the camera motor communicate serially. The digital sensors control the hatch motor, detect the hatch position, and detect the camera position.

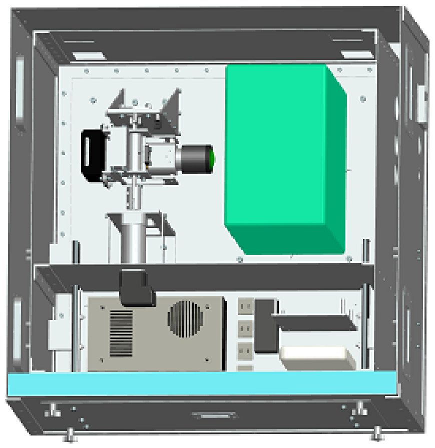


Figure 3: ICI System Enclosure with Top Removed (Top View)

ICI System Network

An Ethernet connection is used to communicate with the embedded system inside the ICI optics system enclosure. The network includes a personal computer running the ICI software interface, the microcontroller, and the ICI camera. The ICI system contains an internal Ethernet hub to which the microcontroller and the ICI camera connect. The computer has one physical Ethernet connection to the Ethernet hub. The ICI software interface uses the Ethernet link to establish a connection with the microcontroller, and separately establishes a connection with the ICI camera (Figure 4).

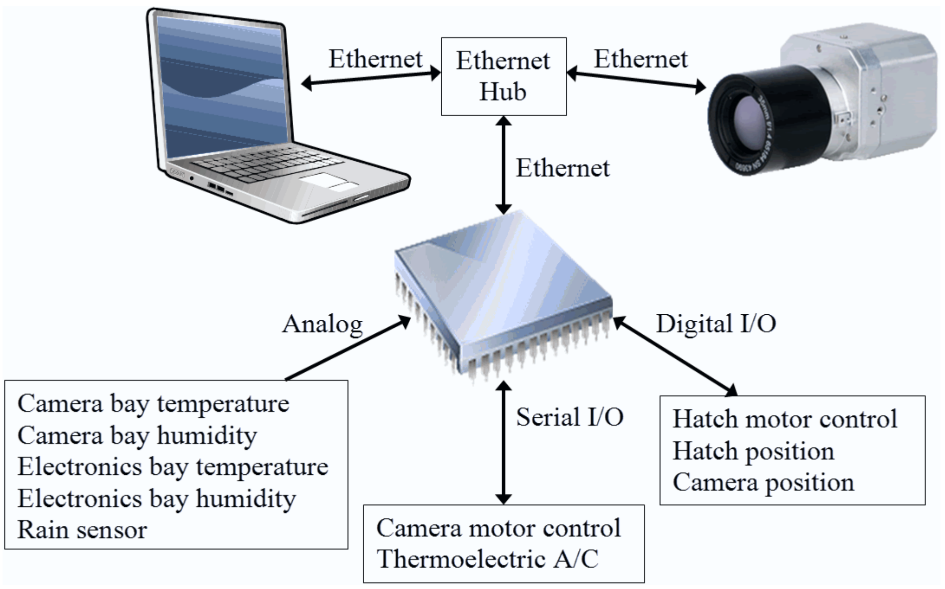


Figure 4: ICI System Network

<u>Microcontroller Specifications</u>

The microcontroller selected for the embedded system was required to be capable of supporting digital, analog, and serial components. ICI system requirements dictated that the microcontroller must have a minimum of four digital inputs, two digital outputs, six analog inputs, two serial ports, and Ethernet support. In addition, the hatch control system requires two digital inputs, one digital output, and one analog input.

<u>Microcontroller Selection</u>

The ICI project specifications were used to generate the microcontroller specifications, and then the microcontroller specifications were used to begin the microcontroller selection process. There are many companies that manufacture microcontrollers, each with different features and design styles, and selecting the right microcontroller for an application can determine the success or failure of a project. The main goal is to select the least expensive microcontroller that minimizes the overall cost of the system while still fulfilling the system specification [4].

Selection Process

The first step in the selection process was to determine what the microcontroller needs to do in the system. This dictated the required microcontroller features and controlled the selection process. The second step was to search for a microcontroller that fit all the system requirements. The primary goal was to find a microcontroller with on-chip resources that included support for all features in the specification. Otherwise, the goal was to find a microcontroller that best fit the requirements with a minimum of extra circuitry [4].

The system requirements were used to narrow the field of choices and eliminate infeasible solutions. For the ICI embedded system, the microcontroller is required to support digital, analog, serial, and Ethernet components. Most of these features are common on the majority of microcontrollers. Digital and analog components and support for a single serial port are standard features. Support for two serial ports is less common, but Ethernet is by far the least common component to find on-chip. The serial and Ethernet requirements greatly reduced the number of single-chip solutions for the ICI embedded system.

Writing code for any microcontroller requires a detailed knowledge of the microcontroller being programmed [5]. The single largest driving factor in the microcontroller selection for the ICI project was compatibility with the type of device currently used for teaching at MSU. One or more students will be selected to continue development on the ICI microcontroller implementation, and students in the electrical engineering program will be most familiar with the electronics and tools used in their classes. A microcontroller from the same family of devices used at MSU was the ideal solution for the ICI embedded system.

ICI Embedded System

The MSU electrical engineering department is currently using the MC9S12C32 microcontroller, a 16-bit device from the HCS12 family of microcontrollers at Freescale Semiconductor. This microcontroller has a single serial port and does not have Ethernet support. Thus, it does not meet the requirements for the ICI embedded system. The MC9S12C32 could not be used for this project, but a different microcontroller was selected from the same product family. The MC9S12NE64 is a 16-bit microcontroller from the HCS12 family with Ethernet, two serial ports, an eight channel Analog to Digital converter, and 70 digital input and output pins [6]. This microcontroller perfectly fits every aspect of the microcontroller specification and provides a single-chip solution for the ICI embedded system.

Development Environment

After selecting the microcontroller for the ICI embedded system, the development environment and development tools for the firmware and software implementations were selected. The program code in an embedded system is usually referred to as firmware [3]. For this project, “firmware” will be used to refer to the ICI embedded system implementation, and “software” will be used to refer to the test applications.

Microcontroller Development Tools

The EVB9S12NE64 is the evaluation board for the MC9S12E64 microcontroller (Figure 5). The development kit purchased for this project included the evaluation board, a power supply, a DB9 serial cable, an Ethernet cable, an evaluation version of Freescale CodeWarrior, firmware examples, and the serial debug monitor. The evaluation board is equipped with two serial ports with RS232 DB9-S connectors, a 10/100T Ethernet port, ports for all digital input and output, and a breadboard area for prototyping. The evaluation board is approximately 5.0 x 7.0 inches, which makes it small enough to include in the enclosure for the ICI system. This eliminated the need to design a custom board for the MC9S12NE64.

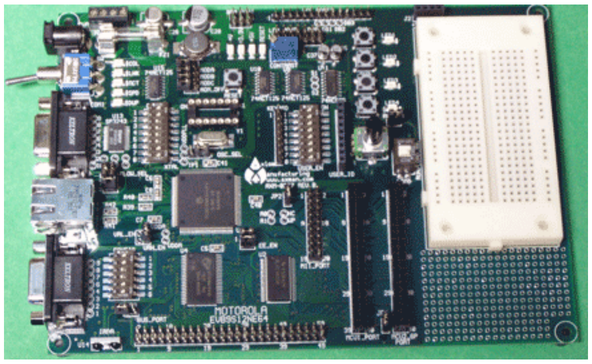


Figure 5: EVB8S12NE64 [Freescale.com]

The microcontroller evaluation board comes equipped with an internal serial monitor. This program is located in the microcontroller memory, and it supports 23 primitive debug commands that allow FLASH/EEPROM programming and debugging through an RS-232 serial interface to a personal computer [7]. The serial monitor uses one of the microcontroller serial ports for communication, and this prevents general use of this port in the ICI system. Since this project requires two serial ports, an alternative development system was necessary for programming and debugging the system.

A background debug mode (BDM) system is an electronic interface for in-circuit programming and debugging of an embedded system. The BDM system consists of a host computer, a BDM interface, and the microcontroller. The computer connects to the BDM interface via a USB cable, and the BDM interface connects to the microcontroller via a custom 6-pin connector and cable [8]. The BDM interface used for this project (Figure 6) was the P&E USB HC08/HC12 Multilink from P&E Micro.

Figure 6: P&E USB HC08/HC12 Multilink [pemicro.com]

All development and testing was performed on a Dell Inspiron 6000d laptop computer. This computer was loaded with Microsoft Windows XP, and was equipped with an Intel Pentium M 2[GHz] processor and 1[GB] of RAM. To test all the features of the ICI system, the laptop needed an Ethernet port, a serial port, and a USB port. The Ethernet port was for testing the TCP/IP implementation, the serial port was for testing the serial implementation, and the USB port was for programming the microcontroller.

Microcontroller Development Environment

All of the firmware for the microcontroller was implemented in the Freescale CodeWarrior IDE for the HC12. The firmware was developed in the C programming language. The basic version of CodeWarrior is free, but it limits the number and sizes of files that can be included in the project. The TCP/IP implementation pushed the project past these limits, and it was necessary to get a full license to compile and develop this project. An extended license was acquired for free through the university.

Software Development Environment

Two software applications were developed for testing the microcontroller functionality. The first application was a TCP/IP client used for testing the TCP/IP implementation. The second application was a terminal program used for testing the serial implementation. This software was implemented in Borland C++ Builder 6. C++ Builder is an integrated development environment used for designing graphical user interfaces, and it supports development in the programming languages C and C++. The test software was implemented in C++.

PROJECT IMPLEMENTATION

After determining the project specifications, the microcontroller, and the development environment, the firmware for the embedded system was designed and implemented. The firmware for this project was structured to match the project specifications and divided into separate files for each component of the system. Firmware was developed for the Ethernet communications, analog channels, digital channels, and serial communications, and a command set was developed for interfacing with the system peripheral devices.

Project Overview

The firmware for the microcontroller was implemented in the Freescale CodeWarrior IDE for the HC12. The CodeWarrior project is named ProjectICI, and the project files are divided into two main groups: the Ethernet implementation and the ICI system implementation (Figure 7). The Ethernet implementation uses OpenTCP, an open source implementation of the Ethernet TCP/IP stack. The ICI system implementation was created for this project and includes support for the analog, digital, and serial components.

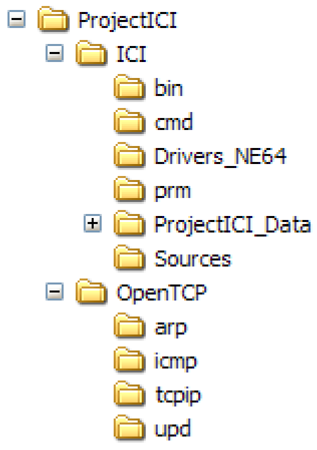


Figure 7: ProjectICI Directory Structure

The ICI directory contains the CodeWarrior project files and application source code. The OpenTCP directory contains the files required for implementing the Ethernet TCP/IP stack. The following is a description of the ProjectICI directory structure shown in Figure 7:

Table 1: ProjectICI Directory Description

{Project Directory}\ICI	Contains the CodeWarrior project files for the ICI system implementation including ProjectICI.mcp.
{Project Directory}\ICI\bin	Contains the CodeWarrior project binaries for programming.
{Project Directory}\ICI\cmd	Contains the target command files.
{Project Directory}\ICI\Drivers_NE64	Contains the MC9S12NE64 drivers.
{Project Directory}\ICI\prm	Contains the CodeWarrior project parameter files for each target.
{Project Directory}\ICI\ProjectICI_Data	Contains the CodeWarrior project object files for each target.
{Project Directory}\ICI\Sources	Contains the applications source code including main().
{Project Directory}\OpenTCP	Contains the OpenTCP source code.
{Project Directory}\OpenTCP\arp	Contains the ARP source code.
{Project Directory}\OpenTCP\icmp	Contains the ICMP source code.
{Project Directory}\OpenTCP\tcpip	Contains the TCP/IP source code.
{Project Directory}\OpenTCP\udp	Contains the UDP source code.

Ethernet Files

The Ethernet files from the OpenTCP implementation include support for the Address Resolution Protocol (ARP), the Internet Control Message Protocol (ICMP), the Transmission Control Protocol (TCP), the Internet Protocol (IP), and the User Datagram Protocol (UDP). Support for these standards is required to implement the Ethernet TCP/IP stack. The following is a description of the main OpenTCP files:

Table 2: OpenTCP File Description.

arp.c	OpenTCP ARP implemenation.
Icmp.c	OpenTCP ICMP implementation.
ip.c	OpenTCP IP implementation.
tcp.c	OpenTCP TCP implementation.
udp.c	OpenTCP UDP implemenation.

ICI System Files

The ICI system files include support for the system components, the microcontroller, and the CodeWarrior project. There are files for the analog, digital, and serial implementations, and there are files specifically for supporting the MC9S12NE64. The following is a description of the main ICI system files:

Table 3: ICI System File Description

MC9S12NE64	address.c	MAC address and IP address.
	mBug.c	Ethernet buffer descriptions.
	MC9S12NE64.c	Registery implementation.
	ne64api.c	TCP/IP stack interface function.
	ne64driver.c	Low-level initialization code.
ICI System	atd.c	ATD implementation.
	digital_io.c	Digital I/O implemenation.
	ici.c	ICI system implemenation.
	Main.c	Main program loop.
	rti.c	RTI implemenation.
	serial_io.c	Serial implemenation.
	tcp_server.c	ICI system TCP implemenation.
	vectors.c	Interrupt vectors.

System Clock

Many features of the ICI control system depend on the internal Clocks and Reset Generator (CRG). The serial baud rate, timed interrupts, and the Analog to Digital converter all depend on the internal clock. The ICI hatch control system, serial communications, and TCP/IP stack use timers derived from the CRG. The CRG block includes the phase-locked loop (PLL) frequency multiplier, the system clock generator, and the real-time interrupt (RTI).

System Clock Design

The PLL is used to run the microcontroller from a different time base than the incoming oscillator output clock signal (OSCCLK). The oscillator clock is a 25[MHz] crystal. This signal can be divided in a range of 1 to 16 to generate the reference frequency, and then the PLL can multiply this reference by a multiple of 2 up to 126,128. The system clock can be configured to use either the oscillator clock or the PLL clock, and the bus clock will run at half the speed of the system clock. The source clock for the RTI is the oscillator clock, and the RTI can be used to generate a hardware interrupt at a fixed periodic rate.

Several registers are used for the system clock initialization. The CRG clock select register is used to select whether the oscillator clock or PLL clock is used for the system clock. The synthesizer register (SYNR) and reference divider register (REFDV) control the multiplier and divider for configuring the PLL clock. The PLL control register enables and disables the PLL, and the CRG flags register is used to determine if the system has finished its transition to the PLL clock. The following formula is used to calculate the frequency of the PLL clock [6]:

$$\mathrm{PLLCLK} = 2 \times \mathrm{OSCCLK} \times \frac{(\mathrm{SYNR}+1)}{(\mathrm{REFDV}+1)}. \tag{1}$$

System Clock Functionality

In the ProjectICI CodeWarrior project, the system clock initialization is located in the file `main.c`. This file handles the initialization for the entire ICI system and includes the implementation for the main program loop. For this project, the system is configured to use the PLL clock. The PLL clock is set to twice the oscillator clock. This produces a PLL clock frequency equal to 50[MHz]. With this configuration, the bus clock is 25[MHz].

System Clock Implementation

The system clock initialization begins by setting the CRG clock select register to 0. The system clock is set to use the oscillator clock, and the PLL circuitry is disabled.

The PLL multiplication factor and the PLL reference divider are set to 0, resulting in a PLL clock of 50[MHz]. Once the PLL frequency is configured, the PLL circuitry is enabled. The system waits for the PLL to reach the desired frequency, and then the system clock is switched from the oscillator clock to the PLL clock.

```
CLKSEL=0;            //Reset the CRG clock select register.
CLKSEL_PLLSEL = 0;   //De-select the PLLCLK (select the OSCCLK).
PLLCTL_PLLON = 0;    //Turn off the PLL circuitry.
SYNR = 0;            //Set PLL the multiplication factor to 0.
REFDV = 0;           //Set PLL the reference divider to 0.
PLLCTL = 192;        //Turn on clock monitor and PLL circuitry.
PLLCTL_PLLON = 1;    //Turn on the PLL circuitry.
while(!CRGFLG_LOCK); //Wait for PLL to reach desired frequency.
CLKSEL_PLLSEL = 1;   //Select the PLLCLK.
```

Real-Time Interrupt

The ICI control system requires updating components of the system in real-time. Normal program flow in the embedded system is through a main program loop. The main loop is infinite and loops continuously through system tasks while the system is running. The ICI system needs the ability to interrupt this loop and take immediate action on time-critical tasks. For this purpose, the system uses the real-time interrupt (RTI). The real-time interrupt is an interrupt that is generated after a specified time interval has expired. Each time the time expires, the main program loop is interrupted, the system processes time-critical tasks, and then the main program loop continues executing.

Real-Time Interrupt Design

The source clock for the RTI is the oscillator clock. The real-time interrupt is generated after a time interval has expired, and the rate at which this interrupt is generated is configurable. Several registers are used for RTI initialization. The CRG interrupt enable register is used to enable the real-time interrupt, and the CRG flags register is used to determine if an interrupt has occurred. The RTI control register is used to configure the interrupt time interval. This register includes a pre-scale rate and a modulus counter, and these values together divide the oscillator clock frequency and determine the RTI frequency [6].

$$\text{RTI frequency} = \frac{\text{OSCCLK}}{\text{PRESCALE} \times \text{MODULUS}} \tag{2}$$

Real-Time Interrupt Functionality

In the ProjectICI CodeWarrior project, the RTI implementation is located in the files `RTI.h` and `RTI.c`. These files handle the initialization and control of the RTI. For this project, the RTI is set to 10[ms]. This configuration was necessary for the open source TCP/IP stack implementation. The stack implementation assumes the RTI is set to 10[ms]. The following are the functions used to implement the RTI.

- void rti_init (void): The `rti_init` function is called during the ICI system initialization in the file `main.c` to initialize the real-time interrupt. The initialization sets the interrupt interval to 10[ms] and disables the RTI.
- void rti_enable (void): The `rti_enable` function enables the RTI. This function is called during the ICI system initialization in the file `main.c` after the RTI is initialized.
- void rti_disable (void): The `rti_disable` function disables the RTI. This function is not currently used.

- interrupt void realtime_interrupt (void): The `realtime_interrupt` is the interrupt service routine for the RTI and is called each time the RTI interval expires. The interrupt updates the timers for the hatch control system, the serial I/O, and the TCP/IP stack.

Real-Time Interrupt Implementation

The RTI initialization begins by clearing the RTI enable flag in the CRG interrupt enable register and disabling the interrupt. The RTI pre-scale rate select bits are set to 2^{16}, and the modulus counter is set to 4. This configuration sets the frequency divider to $4*2^{16}$, and this divider is used to scale the oscillator clock frequency to the RTI frequency. With this configuration, the interrupt frequency is 95.367[Hz], or 10.486[ms]. After configuring the interrupt, the RTI is enabled using the RTI enable flag in the CRG interrupt enable register.

```
CRGINT_RTIE = 0;   //Disable the RTI.
RTICTL_RTR = 0x73; //Set the RTI frequency to 10[ms].
CRGINT_RTIE = 1;   //Enable the RTI.
```

The real-time interrupt requires an interrupt service routine (ISR). The ISR for the RTI is vector number 7 in the interrupt vector table, and the ISR is located at the addresses 0xFFF0 and 0xFFF1. The interrupts for this project are implemented as an array of functions that make up an interrupt table. To add the ISR for the RTI, the name of the function is added to the location in the array corresponding to the correct address.

```
const tIsrFunc _vect[] @0xFF80 =
{ /* Interrupt table */
  …,                  //other interrupts.
  …,
  realtime_interrupt, //The ISR for the RTI.
  …,
  …                   //other interrupts.
};
```

The ISR for the RTI is called each time the RTI interval expires, approximately every 10[ms]. The ISR is declared using the `interrupt` qualifier, and this tells the compiler to generate a return from interrupt at the end of the service routine. The ISR is used to update the ICI system timers for the hatch control system, serial communications, and the TCP/IP stack. This routine is also responsible for clearing the RTI interrupt flag in the CRG flags register. This flag is cleared by writing a 1 to the bit corresponding to the RTI flag.

```
interrupt void realtime_interrupt (void)
{
  CRGFLG = CRGINT_RTIE_MASK; //Clear the RTI flag.
  …; //Update the ICI system timers.
}
```

Digital Inputs and Outputs

The ICI design requires several digital inputs and outputs for reading digital sensors and setting digital controls. The project specifications include a minimum of four digital inputs and two digital outputs, but this was considered an absolute minimum. Additional lines were requested to allow room for more digital I/O if necessary as the ICI project evolves.

Digital I/O Design

The MC9S12NE64 microcontroller comes with up to 70 I/O lines. The microcontroller is equipped with a port integration module that establishes the interface between the microcontroller's peripheral modules and the I/O pins for all ports. Not all ports share the same features, but a standard port is capable of input and output selection, reduced drive selection, configuring pull resistors, interrupts, and status flags.

The I/O pins have a nominal level of 3.3[V]. The internal structure of each pin is identical, and the same absolute maximum ratings apply to all pins. The supply voltage is typically 3.3[V], but can range from 3.135[V] to 3.465[V]. The input voltage is rated from -0.3[V] to 6.5[V], and the input high voltage has a minimum value of 65% of the supply voltage. The output high voltage has a minimum value of the supply voltage minus 0.4[V], and the output low voltage has a maximum value of 0.4[V]. An output pin at full strength can source 4.5[mA].

Not all pins in the port integration module are available for general-purpose digital I/O. For example, port S is associated for the serial communications interfaces and serial peripheral interface. Since this project requires the serial communications interface, port S will be used for serial I/O and can not be used for digital I/O. Port H, port G, port J, and port L are also being used for this project and are associated with the Ethernet media access controller and the Ethernet physical transceiver. This leaves port A, port B, port E, and port K. These ports are associated with the multiplexed external bus interface. Access to external memory or peripheral devices is not currently a requirement for this project, and these ports are available for general purpose digital I/O.

Several registers are associated with each port and are used for initializing and controlling the port pins. The data direction register is used to configure the port pins as digital inputs or digital outputs. If a pin is configured for output, the reduced drive register is used to select the drive strength as either full or reduced. If reduced, the pin drives the output at about 1/3 of the full drive strength. If a pin is configured for input, the pull device enable register specifies if a pull resistor is enabled. If enabled, the polarity select register selects whether a pull-up or pull-down resistor is connected to the pin. The interrupt enable register and interrupt flag registers are used to enable and read pin interrupts. The data register returns the port status [6].

Digital I/O Functionality

In the ProjectICI CodeWarrior project, the digital I/O implementation is located in the files `digital_io.h` and `digital_io.c`. These files handle the initialization and control of the digital input and output channels. For this project, port A was selected for the ICI digital inputs, and port B for the digital outputs. Port A and port B each have eight pins, and this implementation allocates eight digital inputs and eight digital outputs. This meets the minimum requirement of four digital inputs and two digital outputs, and it leaves room for later expansion. The following are the functions used to implement the digital I/O.

- void digital_io_init (void): The `digital_io_init` function is called during the ICI system initialization in the file `main.c` to initialize the digital input and output channels. The initialization sets port A to digital input channels and port B to digital output channels.
- UINT8 get_digital_in(UINT8 Index, UINT8 *Data): The `get_digital_in` function gets the state of the specified digital input channel. The first parameter `Index` is an 8-bit unsigned integer that specifies the location of the digital input channel. The second parameter `Data` is a pointer to an 8-bit unsigned integer that will hold the current state of the digital input channel. The function return value is

an 8-bit unsigned integer that will be zero if the function is successful and an error code if the function fails.

- UINT8 get_digital_out(UINT8 Index, UINT8 *Data): The `get_digital_out` function gets the state of the specified digital output channel. The first parameter `Index` is an 8-bit unsigned integer that specifies the location of the digital output channel. The second parameter `Data` is a pointer to an 8-bit unsigned integer that will hold the current state of the digital output channel. The function return value is an 8-bit unsigned integer that will be zero if the function is successful and an error code if the function fails.
- UINT8 set_digital_out(UINT8 Index, UINT8 Data): The `set_digital_out` function sets the state of the specified digital output channel. The first parameter `Index` is an 8-bit unsigned integer that specifies the location of the digital output channel. The second parameter `Data` is an 8-bit unsigned integer that specifies the new state of the digital output channel. The function return value is an 8-bit unsigned integer that will be zero if the function is successful and an error code if the function fails.

Digital I/O Implementation

The digital I/O initialization begins by setting the data direction register for port A to 0x00. This configures all the pins on port A as digital inputs. The data direction register for port B is set to 0xFF, which configures all the pins on port B as digital outputs. The other port configuration registers are left at 0. The output pins are used at full drive strength, and pull resistors are not used with the input pins.

```
DDRA = 0x00; //Set Port A to all input.
DDRB = 0xFF; //Set Port B to all output.
```

The data register for port A is used to determine the state of each input channel. The size of the data register is one byte, or eight bits. Each input channel is associated with a bit. To read the state of an input channel, the corresponding bit is read from the data register. The index of the channel is used to find the correct bit. The data register is shifted to the right until the desired channel is in the first bit position, and then the resulting value is masked with a 0x01 to read the channel value from of the first position. The same process is used to read the state of each output channel on port B.

```
//Read the state of the input pin located at Index (0-7).
State = (PORTA >> Index) & 0x01;
//Read the state of the output pin located at Index (0-7).
State = (PORTB >> Index) & 0x01;
```

The data register for port B is also used to set and clear the state of each output channel. To set the state of an output channel, the corresponding bit is set to 1. To clear the state, the bit is set to 0. Again, the index of the channel is used to find the correct bit. To set the output channel, the bit-mask 0x01 is shifted to the left until the mask is at the desired channel's bit position, and then a bit-wise `OR` operation is used with the mask to set the bit in the data register. To clear the output channel, the same bit-mask is generated, but the bits are flipped to generate a 0 in the desired channel's bit position. Then a bit-wise `AND` operation is used with the mask to clear the bit in the data register.

```
//Set the state of the output pin located at Index (0-7).
PORTB |= 0x01 << Index;
//Clear the state of the output pin located at Index(0-7).
PORTB &= ~(0x01 << Index);
```

Hatch Control System

The ICI-3 system design includes a motorized hatch that opens and closes under direction of the microcontroller. The hatch opens to allow the camera to collect data, and then closes when the camera is finished. The microcontroller operates the hatch and provides failsafe routines in the event of an error. The hatch control system has two position sensors that indicate if the hatch is open or closed, and one control line that tells the hatch motor to start moving or stop moving. When the microcontroller receives an open or close command, the microcontroller starts moving the hatch, waits until the hatch is in the correct position, and then stops moving the hatch. The hatch control system is also equipped with an analog rain sensor to detect when the weather conditions are unfavorable for camera operation [9].

Hatch Control System Design

The hatch control system failsafe routines are necessary to minimize the possibility that the hatch is unintentionally left open or opened while it is raining. There are several scenarios that may result in an improperly opened hatch. For example, a user may tell the microcontroller to open the hatch and then fail to send the command to close the hatch. Or, if a sensor fails the microcontroller may not be able to reach the commanded hatch state, and the hatch will continue to open and close. In the worst-case scenario, both sensors may fail, and the microcontroller will have no knowledge of the hatch position. In any case, the failsafe routines are designed to move the hatch to the best estimate of the closed position.

The hatch control system is implemented using digital input and output channels. The hatch control is simply an extension of the general-purpose digital I/O already required for this project. The ICI project specifications include a minimum of four digital inputs and two digital outputs. During the digital I/O implementation, it was determined that the microcontroller could easily support eight digital inputs and eight digital outputs. Some of these inputs and outputs were originally intended to be used with the hatch control system, but it was later requested that, if possible, additional digital lines be included for this system. The hatch control system requires two digital inputs and one digital output.

The MC9S12NE64 microcontroller comes with up to 70 I/O lines that are accessed through the port integration module. The configuration and electrical characteristics of these ports are discussed in the previous section on general purpose digital I/O. Port A, port B, port E, and port K are associated with the multiplexed external bus interface. Access to external memory or peripheral devices is not currently a requirement for this project, and these ports are available for general purpose digital I/O. Port A and port B were selected for supporting the ICI digital I/O requirements. This leaves port E and port K for use with the hatch control system.

The hatch control system also requires one analog input for the analog rain sensor. The configuration and electrical characteristics of the analog channels are discussed in the next section on the Analog to Digital (ATD) converter. The MC9S12NE64 microcontroller comes with one ATD converter that has eight analog channels. The ICI project specifications include a minimum of six analog channels for general purpose analog inputs. This specification leaves room for reserving an analog input for the hatch control system. For this project, the analog channel used for the rain sensor is configurable, and the user can select any of the eight channels for the rain sensor.

Hatch Control System Functionality

In the ProjectICI CodeWarrior project, the hatch control system implementation is located in the files `digital_io.h` and `digital_io.c`. These files handle the initialization and control of the digital input and output channels, and the initialization and control for the hatch control system. For this project, port K was selected for use with the hatch control system because of its position on the microcontroller evaluation board. The pins for port K are located right after the pins for port A and port B, the digital input and output ports. On port K, pin 1 and pin 2 are used for the digital input lines, and pin 0 is used for the digital output line. The following are the functions used to implement the hatch control system.

- void digital_io_init(void): The `digital_io_init` function is called during the ICI system initialization in the file `main.c` to initialize the hatch control system. The initialization sets pin 1 and pin 2 on port K to digital input channels and pin 0 on port K to a digital output channel. This function initializes the global structures `hatch_state` and `rain_sensor`, and clears the timer `hatch_open_timer`. The structure `hatch_state` contains information on the hatch position and movement, and the structure `rain_sensor` is used to monitor the state of the analog rain sensor. The timer `hatch_open_timer` controls the time that the hatch is allowed to remain open.

- void update_hatch_timers(void): The `update_hatch_timers` function is called approximately every 10[ms] during the interrupt service routine for the real-time interrupt. The update function decrements the value of the `hatch_open_timer`. When the open time expires, it closes the hatch. It updates the value of the `rain_sensor`, closes the hatch if rain is detected, and does not allow the hatch to open until the conditions are considered dry. It also decrements the value of the move time in `hatch_state`. If the move time has not expired, the hatch control system is updated to keep the hatch moving until the desired hatch position is reached.
- void process_hatch_status(void): The `process_hatch_status` function is called approximately every 10[ms] during the `update_hatch_timers` function. This function updates the hatch control system. It determines if the hatch is moving and reads the current hatch position. Based on this information and the values in the structure `hatch_state`, it determines the next action necessary to reach the desired hatch state.
- UINT8 get_hatch_control(UINT8* Data): The `get_hatch_control` function gets the state of the hatch control system's digital output line. If the state is 0, the hatch is not moving. If the state is 1, the hatch is moving. The parameter `Data` is a pointer to an 8-bit unsigned integer that will hold the current state of the digital output line. The function return value is an 8-bit unsigned integer that will be zero if the function is successful and an error code if the function fails.

- UINT8 set_hatch_control(UINT8 Data): The `set_hatch_control` function sets the state of the hatch control system's digital output line. The parameter `Data` is an 8-bit unsigned integer that specifies the new state of the digital output line. The function return value is an 8-bit unsigned integer that will be zero if the function is successful and an error code if the function fails.
- UINT8 get_hatch_state(UINT8* Data): The `get_hatch_state` function gets the state of the hatch control system's digital input lines. The hatch control system uses two digital input lines that are connected to hatch position sensors, and the two lines correspond to the two bits that determine the current position of the hatch. A state of 0 indicates the hatch is between sensors, 1 indicates the hatch is open, 2 indicates the hatch is closed, and 3 indicates an error has occurred. The parameter `Data` is a pointer to an 8-bit unsigned integer that will hold the current state of the digital input lines. The function return value is an 8-bit unsigned integer that will be zero if the function is successful and an error code if the function fails.
- UINT8 set_hatch_state(UINT8 Data): The `set_hatch_state` function sets the position of the hatch. The new position corresponds to the desired state of the digital input lines. A position of 1 indicates the hatch should move until the open sensor is active, or the hatch is open, and a position of 2 indicates the hatch should close. This function calls `set_hatch_control` to start moving the hatch and initializes the structure `hatch_state` with the information necessary to reach the desired hatch position. The parameter `Data` is an 8-bit unsigned integer that

specifies the new position of the hatch. The function return value is an 8-bit unsigned integer that will be zero if the function is successful and an error code if the function fails.

- UINT8 get_hatch_error(UINT8* Data, UINT8 Clear): The `get_hatch_error` function gets the current value of the hatch control system error. The error is stored in the structure `hatch_state` and indicates if an error has occurred while the system was moving the motor. The first parameter `Data` is a pointer to an 8-bit unsigned integer that will hold the current error value. The second parameter `Clear` is an 8-bit unsigned integer that specifies if the system should clear the error. The function return value is an 8-bit unsigned integer that will be zero if the function is successful and an error code if the function fails.
- UINT8 get_rain_index(UINT8 *Data): The `get_rain_index` function gets the index of the ATD channel associated with the hatch control system's analog rain sensor. The parameter `Data` is a pointer to an 8-bit unsigned integer that will hold the index of the rain sensor. The function return value is an 8-bit unsigned integer that will be zero if the function is successful and an error code if the function fails.
- UINT8 set_rain_index(UINT8 Data): The `set_rain_index` function sets the index of the ATD channel associated with the hatch control system's analog rain sensor. The parameter `Data` is an 8-bit unsigned integer that specifies the new index of the rain sensor. The function return value is an 8-bit unsigned integer that will be zero if the function is successful and an error code if the function fails.

- UINT8 get_rain_wet(UINT8 *Data): The `get_rain_wet` function gets the wet threshold of the hatch control system’s analog rain sensor. The wet threshold determines when the system assumes that it is raining and closes the hatch. The parameter `Data` is a pointer to an 8-bit unsigned integer that will hold the value of the wet threshold. The wet threshold is given in ticks and can range from 0 to 1023 ticks. The function return value is an 8-bit unsigned integer that will be zero if the function is successful and an error code if the function fails.
- UINT8 set_rain_wet(UINT8 Data): The `set_rain_wet` function sets the wet threshold of the hatch control system’s analog rain sensor. The parameter `Data` is an 8-bit unsigned integer that specifies the new value of the wet threshold. The function return value is an 8-bit unsigned integer that will be zero if the function is successful and an error code if the function fails.
- UINT8 get_rain_dry(UINT8 *Data): The `get_rain_dry` function gets the dry threshold of the hatch control system’s analog rain sensor. The dry threshold determines when the system assumes that it has stopped raining and allows the hatch to open. The parameter `Data` is a pointer to an 8-bit unsigned integer that will hold the value of the dry threshold. The dry threshold is given in ticks and can range from 0 to 1023 ticks. The function return value is an 8-bit unsigned integer that will be zero if the function is successful and an error code if the function fails.
- UINT8 set_rain_dry(UINT8 Data): The `set_rain_dry` function sets the dry threshold of the hatch control system’s analog rain sensor. The parameter `Data` is

an 8-bit unsigned integer that specifies the new value of the dry threshold. The function return value is an 8-bit unsigned integer that will be zero if the function is successful and an error code if the function fails.

Hatch Control System Implementation

The hatch control system uses two digital input lines that are connected to hatch position sensors. If the state of a digital input line is 0, the sensor is not active and the hatch is not touching that sensor. If the state is 1, the sensor is active and the hatch is touching that sensor. The sensors are positioned on opposite sides of the hatch opening. The hatch will activate a sensor when it is completely open or completely closed. It is not possible for the hatch to be in both positions concurrently. Therefore, the sensors should never be active at the same time. The two digital inputs are used together to form a 2-bit state variable that indicates the current position of the hatch. A state of 0 indicates the hatch is between sensors, 1 indicates the hatch is open, 2 indicates the hatch is closed, and 3 indicates both sensors are active and an error has occurred.

The hatch control system uses one digital output line that is connected to the hatch motor. The hatch motor is used to move the system hatch to the open and closed positions. If the state of the digital output line is 0, the hatch motor is not moving. If the state is 1, the hatch motor is moving. The hatch is designed to move in a circular motion. While the hatch motor is active, that hatch will move in a circular motion from the open position to the closed position and back to the open position. Without any additional logic, the hatch would keep moving in a circle indefinitely. The hatch control system is designed to detect when the hatch has reached the desired position and stop the hatch motor.

The hatch control system uses one analog input that is connected to an analog rain sensor. The rain sensor is used to detect when it is raining. The sensor input ranges from 0[V] to 3.3[V]. The low voltage corresponds to wet conditions and the high voltage corresponds to dry conditions. The hatch control system uses configurable wet and dry voltage thresholds to divide the sensor voltage range into sections for wet and dry conditions, and the system wet and dry thresholds are configurable. If the voltage is below a wet threshold, it is raining. If it is above the dry threshold, it is not raining. The hatch control system is designed to detect when it is raining and automatically close the hatch. If rain is detected, the hatch control system prevents the hatch from opening until the rain has stopped.

Several global structures and variables are used to manage the hatch control system. The first structure is `hatch_state`. This structure contains information on the hatch position and movement. The `Start` variable is set to the initial position of the hatch, and the `Target` variable is set to the desired position of the hatch. For example, if the hatch is currently closed and the system instructs the hatch to open, `Start` will be set to 2 (closed) and `Target` will be set to 1 (open). The `MoveTime` variable is set to the time elapsed since the last hatch state. This variable is used to record the time it takes to move from one hatch position to another. The `Timeout` is set to the maximum amount of time the hatch is allowed to move, and `Error` is set to an error code if the hatch control system encounters an error.

```
typedef struct
{
  UINT8 Start;     //Start hatch state.
  UINT8 Target;    //Desired hatch state.
  UINT16 MoveTime; //Time elapsed since last hatch state.
  UINT16 Timeout;  //Total time hatch is allowed to move.
  UINT8 Error;     //Hatch error.
} hatch_state_info;
hatch_state_info volatile hatch_state;
```

The second hatch control system structure is `rain_sensor`. This structure is used to monitor the state of the analog rain sensor. The `Index` variable is the index of the analog channel associated with the rain sensor. The `Wet` and `Dry` variables hold the wet and dry thresholds of the rain sensor. These values are given in ticks and can range from 0 to 1023 ticks. The `Value` variable is used to hold the current value of the analog rain sensor.

```
typedef struct
{
  UINT8 Index;  //Index of rain sensor ATD.
  UINT16 Wet;   //Wet value.
  UINT16 Dry;   //Dry value.
  UINT16 Value; //Current value.
} rain_sensor_info;
rain_sensor_info volatile rain_sensor;
```

The last global variable for the hatch control system is the `hatch_open_timer`. This timer determines the time the hatch is allowed to remain open. For this project, the open time is set to 60[s]. After this time has expired, the hatch control system is designed to automatically close the hatch.

The hatch control system initialization begins by setting the data direction register for port K to 0x01. This configures all the pins on port K as digital inputs except for bit 0. Bit 0 is configured as an output and is used for the hatch motor control line. Bits 1 and 2 are configured as inputs and are used for the hatch position sensors. Bit 1 corresponds to the open sensor, and bit 2 corresponds to the closed sensor.

```
DDRK = 0x01; //Set Port K to inputs, and bit 0 as output.
```

The `hatch_state` structure and `hatch_open_timer` are cleared, and the `rain_sensor` structure is set to the default sensor configuration. The default configuration sets the rain sensor analog channel to the first analog channel, or the channel located at index 0. The wet threshold is set to 1.5[V], and the hatch control system will wait for the sensor to read below 1.5[V] before it assumes it is raining and the conditions are wet. The dry threshold is set to 2[V], and the system will wait for the rain sensor to read above 2[V] before it assumes it is not raining and the conditions are dry.

```
//Clear the hatch_state.
(void)memset((void*)(&hatch_state), 0, sizeof(hatch_state_info));
//Clear hatch open timer.
hatch_open_timer = 0;
//Configure the rain sensor.
rain_sensor.Index = HATCH_RAIN_INDEX; //Default index to ATD 0.
rain_sensor.Wet = HATCH_RAIN_WET;     //Default wet to 1.5[V].
rain_sensor.Dry = HATCH_RAIN_DRY;     //Default dry to 2[V].
```

The data register for port K is used to determine the state of the hatch control system’s digital input and output channels. The input channels are the hatch position sensors and the output channel is the hatch motor control. The size of the data register is one byte, or eight bits. Each channel is associated with a bit. To read the state of a channel, the corresponding bit is read from the data register. The input channels associated with the hatch position sensors are located at bits 1 and 2. A bit-wise `AND` operation is used with the data register and the mask 0x06 to read bits 1 and 2, and the result is shifted to the right 1 bit position. The output channel associated with the hatch motor is located at bit 0. The data register is masked with a 0x01 to read bit 0.

```
//Read the hatch position sensors (bits 1 and 2).
State = (PORTK & 0x06) >> 0x01;
//Read the hatch motor control (bit 0).
State = PORTK & 0x01;
```

The data register for port K is also used to set and clear the state of the hatch control system's digital output channel. The output channel is the hatch motor control. To set the state of an output channel, the corresponding bit is set to 1. To clear the state, the bit is set to 0. To set the hatch motor control, a bit-wise `OR` operation is used with the mask 0x01 to set bit 0 in the data register. To clear the motor control, the same bit-mask is used, but the bits are flipped to generate a 0 in the first bit position. Then a bit-wise `AND` operation is used with the mask to clear the bit in the data register.

```
//Set the state of the hatch motor control (bit 0).
PORTK |= 0x01;
//Clear the state of the hatch motor control (bit 0).
PORTK &= ~0x01;
```

The hatch control system uses the digital input and output channels to control the position of the hatch. The system uses the digital output line to control the hatch motor and move the hatch until it reaches a new position. The new position corresponds to the desired state of the digital input lines. A position of 1 indicates the hatch should move until the open sensor is active, or the hatch is open, and a position of 2 indicates the hatch should close.

The process for moving the hatch begins with the initialization of the global `hatch_state` structure. The `Start` variable is set to the current hatch position, and the `Target` variable is set to the desired hatch position. The `MoveTime` is set to 0, and the `Timeout` is set to 60[s]. After the `hatch_state` is initialized, the hatch motor control line is set high to start moving the motor. If the hatch is already open, the `hatch_open_timer` is set back to the full open timeout of 60[s].

```
//Set hatch state info.
hatch_state.Start = state; //Set start to the current position.
hatch_state.Target = Data; //Set target to the new position.
hatch_state.MoveTime = 0; //Clear the movetime.
hatch_state.Timeout = HATCH_TIMEOUT; //Set the timeout to 60[s].
//Start moving the hatch.
set_hatch_control(HATCH_MOVE_START);
//Reset the hatch open timer.
hatch_open_timer = HATCH_TIMEOUT;
```

The hatch control system only allows the hatch to move if there is no error. If the `Error` variable is set, the error must be cleared by reading the `Error` variable. Once the error is cleared, the hatch will be allowed to move. In the special case that the `Error` variable indicates rain was detected, the hatch is allowed to close.

The hatch control system uses the RTI to update the hatch controls and variables. The RTI updates the system approximately every 10[ms]. The update begins by updating the `hatch_open_timer`. If the timer has not expired, the value of the `hatch_open_timer` is decremented. When the open time expires, the hatch control system instructs the hatch to close.

```
//Update hatch open timer.
if (hatch_open_timer > 0) //Check if the timer has not expired.
{
  hatch_open_timer--; //Decrement the timer.
  if (hatch_open_timer == 0) //Check if the timer has expired.
  {
    //Hatch open time expired. Close the hatch.
    set_hatch_state(HATCH_STATE_CLOSED);
  }
}
```

The next structure updated is the global `rain_sensor` structure. First, the system reads the current value of the analog rain sensor. The current value of the sensor is compared the wet threshold. If the value is less than the threshold, the system assumes it is raining and the conditions are considered wet. If rain was not previously detected, the system's `Error` variable is set to rain and the system closes the hatch. If rain is not detected, the system compares the current value of the rain sensor to the dry threshold. If the value is greater than the threshold, the system assumes it is not raining and the conditions are considered dry. If rain was previously detected and the `Error` variable is already set to rain and the system clears the `Error` variable.

```
//Update rain sensor.
get_atd_in(rain_sensor.Index, &(rain_sensor.Value));
if (rain_sensor.Value < HATCH_RAIN_WET) //Check if wet.
{
  if (hatch_state.Error != ERROR_RAIN) //Check if previously wet.
  {
    hatch_state.Error = ERROR_RAIN; //Set hatch error to rain.
    set_hatch_state(HATCH_STATE_CLOSED); //Close the hatch.
  }
}
else if (rain_sensor.Value > HATCH_RAIN_DRY) //Check if dry.
{
  if (hatch_state.Error == ERROR_RAIN) //Check if previously wet.
  {
    hatch_state.Error = OK; //Clear the hatch error.
  }
}
```

The update function also updates the `Timeout` variable in the global `hatch_state` structure. This variable corresponds to the total time the hatch has been moving. When the hatch is instructed to move to a new position, the `Timeout` variable is set to 60[s]. The system assumes that the hatch can reach any position in less than 60[s]. If a full 60[s] has passed and the hatch has not reached the desired position, the system assumes the position can not be reached and stops moving the hatch. If the timer has not expired, the `Timeout` is decremented and the hatch control system is updated. If the time has expired, the last system update will stop the hatch motor and the system will stop updating.

```
//Update hatch control timer.
if (hatch_state.Timeout > 0) //Check if timer has not expired.
{
  hatch_state.Timeout--; //Decrement the timer.
  process_hatch_status(); //Update the hatch control system.
}
```

After updating all the system variables, the update function updates the hatch control system. It determines if the hatch is moving and reads the current hatch position. Based on this information and the values in the `hatch_state` structure, it determines the next action necessary to reach the desired hatch position. If the hatch is moving, the system checks if an error has occurred, the hatch has reached the desired state, or the move time has expired. In any of these cases, the hatch control system stops moving the hatch. If the hatch just opened, the `hatch_open_timer` is set to 60[s].

```
if (control == HATCH_MOVE_START) //Hatch is moving.
{
  if (hatch_state.Error != OK &&
      hatch_state.Error != ERROR_RAIN) //Check for error.
  {
    set_hatch_control(HATCH_MOVE_STOP); //Stop moving.
  }
  else if (hatch_state.Target == state) //Check if Target.
  {
    set_hatch_control(HATCH_MOVE_STOP); //Stop moving.
    if (hatch_state.Target == HATCH_STATE_OPEN) //Check if open.
    {
      hatch_open_timer = HATCH_TIMEOUT; //Set timeout to 60[s].
    }
  }
  else if (hatch_state.Timeout == 0) //Check for timeout.
  {
    set_hatch_control(HATCH_MOVE_STOP); //Stop moving.
    hatch_state.Error = ERROR_TIMEOUT;
  }
  else //No error. Target not reached. Process current info.
  {
    …;
  }
}
```

If the system does not encounter a case that stops moving the hatch, the next action taken is based on the current hatch position. The hatch control system uses two position sensors that correspond to the open and closed positions. If neither sensor is active, the system assumes the hatch is between sensors and it should keep moving the hatch. It’s possible for the hatch to be between sensors when the system starts moving the hatch. In this case, when the system detects the sensor has reached the open or closed positions, it sets the `Start` variable to the new position and resets the `MoveTime` variable. This is necessary to ensure the system timing starts at a valid hatch position.

```
//If the hatch state is HATCH_STATE_NONE,
//the hatch is not open or closed. Increment the move time.
hatch_state.MoveTime++;
//If the hatch state is HATCH_STATE_OPEN,
//the hatch is currently open. Check the start hatch state.
if (hatch_state.Start == HATCH_STATE_NONE)
{
  //Started in between sensors. Set start state to open.
  hatch_state.Start = HATCH_STATE_OPEN;
  //Reset move time.
  hatch_state.MoveTime = 0;
}
//If the hatch state is HATCH_STATE_CLOSED,
//the hatch is currently closed. Check the start hatch state.
if (hatch_state.Start == HATCH_STATE_NONE)
{
  //Started in between sensors. Set start state to closed.
  hatch_state.Start = HATCH_STATE_CLOSED;
  //Reset move time.
  hatch_state.MoveTime = 0;
}
```

If the open sensor is active, the system checks if the `Start` variable is equal to the open position and the `MoveTime` is not equal to 0. In this case, the hatch started in the open position, moved for a while, and reached the open position again. This means that the system never detected the closed sensor, and the system assumes the closed sensor has failed. If the closed sensor fails, the system timing must be used to close the hatch. At this point, the `MoveTime` is equal to the time it takes for the hatch to move through a complete revolution (open, closed, and then open again). If the hatch moves for half the time it takes to make a complete revolution, it will only move for half of a revolution. During this time, the hatch will move from the open position and stop at the closed position. To get the hatch to move half of a revolution, the `Timeout` variable is set to half of the `MoveTime`, and the `MoveTime` variable is cleared.

```
//If the hatch state is HATCH_STATE_OPEN,
//the hatch is currently open. Check the start hatch state and
//time the hatch has been moving.
if (hatch_state.Start == HATCH_STATE_OPEN &&
    hatch_state.MoveTime != 0)
{
  //Hatch started open, moved, and opened again.
  //Closed sensor never found.
  //Try to close. Keep moving for half of move time.
  hatch_state.Timeout = hatch_state.MoveTime / 2;
  //reset move time.
  hatch_state.MoveTime = 0;
}
```

If the closed sensor is active, the system checks if the `Start` variable is equal to the closed position and the `MoveTime` is not equal to 0. In this case, the hatch started in the closed position, moved for a while, and reached the closed position again. This means that the system never detected the open sensor, and the system assumes the open sensor has failed. If the open sensor fails, the system can not open the hatch and should stop trying to open the hatch. The hatch control system stops the hatch motor and sets the `Error` variable to indicate an error has occurred.

```
//If the hatch state is HATCH_STATE_CLOSED,
//the hatch is currently closed. Check the start hatch state and
//time the hatch has been moving.
if (hatch_state.Start == HATCH_STATE_CLOSED &&
    hatch_state.MoveTime != 0)
{
  //Hatch started closed, moved, and closed again.
  //Open sensor never found.
  //Stop moving with hatch closed.
  set_hatch_control(HATCH_MOVE_STOP);
  //Set hatch error.
  hatch_state.Error = ERROR_TIMEOUT;
}
```

If both sensors are active, an error has occurred. It is not possible for the hatch to be in the open and closed positions at the same time. In this case, the system does not have enough information to determine the current hatch position and move to a new position, and the system should stop trying to move the hatch. The hatch control system stops the hatch motor and sets the `Error` variable to indicate an error has occurred.

```
//If the hatch state is HATCH_STATE_INVALID,
//the hatch sensors are both active. Stop moving and set error.
set_hatch_control(HATCH_MOVE_STOP);
hatch_state.Error = ERROR_STATE;
```

ATD Inputs

The ICI design requires several 10-bit Analog to Digital (ATD) inputs for reading analog sensors. The project specifications include a minimum of six ATDs, but this was considered an absolute minimum. Additional lines were requested to allow room for more analog inputs if necessary as the ICI project evolved.

ATD Design

The MC9S12NE64 microcontroller comes with one ATD converter. The converter has eight analog channels and is capable of 8-bit or 10-bit resolution. The microcontroller is equipped with a port integration module that establishes the interface between the ATD converter and the input pins for the channels. The ATD pins are on Port AD. The converter is highly configurable and has many controls and features. Of interest to this project, the converter has power controls, a configurable clock frequency, left or right justified result data, a configurable conversion length, a conversion complete flag, and is capable of continuous or single conversions.

The ATD I/O pins can be used for general purpose I/O, but for this project they are used for analog inputs. The internal structure of each pin is identical, and the same absolute maximum ratings apply to all pins. The supply voltage is typically 3.3[V], but can range from 3.135[V] to 3.465[V]. The analog reference voltage is rated from -0.3[V] to 6.5[V], and the low reference has a maximum value of half the supply voltage, and the high reference has a minimum value of half the supply voltage. The ATD converter has a 20[us] recovery time after the power is cycled.

The ATD converter has a series of control registers for configuring the ATD functionality, but only a few of these controls were needed for this project. The zero and first control registers are not used, but the second ATD control register provides a bit for turning the ATD power on and off. The ATD requires a recovery period after it is turned on. The third ATD control register has bits for configuring the conversion sequence length and controls the number of conversions per sequence. Up to eight conversions are allowed in a sequence. The forth control register configures the ATD results as either 8-bit or 10-bit, and it selects the conversion clock frequency. The maximum ATD conversion clock frequency is 2[MHz]. The following formula is used to calculate the frequency of the ATD clock:

$$\mathrm{ATDclock} = \frac{\mathrm{BusClock}}{\mathrm{PRS}+1} \times 0.5 . \tag{3}$$

The fifth ATD control register determines the data justification, the data representation, and the type of conversion sequence. The result data can be left or right justified, depending on how the user would like to read out the data, and the data can be signed or unsigned. The ATD converter can sample channels continuously or only once, and it can loop through multiple channels or just scan a single channel. The ATD status register contains the sequence complete flag and is used to determine when the channel results are ready. The ATD conversion result registers are used to read out the ATD channel values [6].

ATD Functionality

In the ProjectICI CodeWarrior project, the ATD implementation is located in the files `atd.h` and `atd.c`. These files handle the initialization and control of the ATD converter. For this project, the ATD clock is set to 1.79[MHz], the ATD is configured to do one conversion per sequence, and 10-bit resolution is used for storing the ATD channel results. The 1.79[MHz] ATD clock is the maximum frequency allowed with the bus clock configured to 25[MHz], and only one conversion per sequence is necessary because only one analog result is collected during each sample request. The following are the functions used to implement the ATD converter.

- void atd_init (void): The `atd_init` function is called during the ICI system initialization in the file `main.c` to initialize the ATD converter. The initialization sets the ATD clock to 1.79[MHz], the conversion length to one conversion per sequence, and 10-bit resolution.
- UINT8 get_atd_in(UINT8 Index, UINT8 *Data): The `get_atd_in` function gets the state of the specified ATD channel. The first parameter `Index` is an 8-bit unsigned integer that specifies the location of the ATD channel. The second parameter `Data` is a pointer to an 8-bit unsigned integer that will hold the current state of the ATD channel. The function return value is an 8-bit unsigned integer that will be zero if the function is successful and an error code if the function fails.

- UINT8 get_atd_resolution(UINT8 *Data): The `get_atd_resolution` function gets the resolution of the ATD converter. The parameter `Data` is a pointer to an 8-bit unsigned integer that will hold the ATD resolution. The function return value is an 8-bit unsigned integer that will be zero if the function is successful and an error code if the function fails.

ATD Implementation

The ATD initialization begins by setting the power up bit in the second ATD control register. This powers the ATD circuitry. The third ATD control register is used to configure the conversion length to one conversion per sequence. The forth ATD control register is used to set the resolution to 10-bit and to set the clock pre-scale rate. The pre-scale rate is set to 6. This configuration sets the divisor to (6+1)*2, which gives a total divisor value of 14. This value is used to scale the bus clock frequency to the ATD clock frequency. With this configuration, the ATD frequency is 1.79[Hz], which is the maximum frequency allowed for a 25[MHz] bus clock.

```
ATDCTL2 = 0x80; //Power up the ATD.
ATDCTL3 = 0x08; //Set to 1 conversion per sequence.
ATDCTL4 = 0x06; //Set to 10-bit resolution and 1.79[MHz].
```

A combination of registers is used to read the ATD result. First, the fifth control register is used to set the result register data justification to right justified. The fifth register is also used to select the correct ATD channel. After the desired channel is selected, the ATD status register is polled until the conversion complete flag is set. Once ready, the channel result is read from the ATD result registers. Since the ATD is configured to 10-bit resolution and right justification, the lower eight bits of the result are stored in the register for the low byte, and the upper two bits are stored in the register for the high byte. Last, the sequence complete flag in the ATD status register is cleared by writing a 1 to the corresponding bit.

```
//Set to right justified and select the channel at index (0-7).
ATDCTL5 = 0x80 | Index;
//Wait for the ATD to finish converting the channel result.
while (ATDSTAT0_SCF == 0);
//Read the channel result.
Result = (ATDDR0H << 8) | ATDDR0L; //10-bit result.
//Reset the conversion sequence complete flag.
ATDSTAT0_SCF = 1;
```

Serial I/O

In addition to manipulating digital and analog information, the microcontroller is intended to act as a “pass through” device. Data is sent to the microcontroller over Ethernet, and the microcontroller takes that data and sends it out serially. The microcontroller also receives data serially and then sends that data over Ethernet. The microcontroller has no knowledge of the devices connected to it. It simply “passes” information from one interface to another.

Serial I/O Design

The MC9S12NE64 microcontroller comes with two serial communications interface (SCI) modules. The SCI allows full duplex or single-wire operation. The full duplex implementation uses separate lines for transmitting and receiving and it can transmit and receive at the same time. A single-wire system transmits and receives on the same line and requires handshaking signals. The SCI has a 13-bit baud rate selection. The baud rate is the number of times a second the state of the signal on a line is changed, and it is given in bits per second [10].

The SCI allows 8-bit or 9-bit data frames and supports even or odd parity. A frame with eight data bits has a total of ten bits. With no parity, the frame has one start bit, eight data bits, and one stop bit. With parity, the frame has one start bit, seven data bits, one parity bit, and one stop bit. A frame with nine data bits has a total of eleven bits and follows the same format. The SCI is also capable of interrupt-driven operation and is equipped with interrupts for transmits, receives, and errors.

The SCI modules are configured independently and each has a series of control registers for configuring the SCI functionality. The first SCI control register is used to configure the data format and parity bit, and the second control register is used to enable the SCI interrupts. The SCI status register contains the state of each interrupt and is used to determine which interrupt has occurred. The SCI data registers are used to transmit and receive serial data, and the SCI baud rate registers are used to configure the baud rate. The baud rate is 13-bits and is split over two registers. The following formula is used to calculate the SCI baud rate [6]:

$$\text{SCI baud rate} = \frac{\text{SCI module clock}}{16 \times \text{SBR}[12:0]}. \tag{4}$$

<u>Serial I/O Functionality</u>

In the ProjectICI CodeWarrior project, the serial I/O implementation is located in the files `serial_io.h` and `serial_io.c`. These files handle the initialization and control of the SCI modules. For this project, the SCI baud rate is set to 9600, the data format is set to one start bit, eight data bits, and one stop bit, and no parity is used. The same settings are used for both SCI modules. The following are the functions used to implement the serial I/O.

- void serial_io_init(void): The `serial_io_init` function is called during the ICI system initialization in the file `main.c` to initialize the SCI modules. The initialization sets the SCI baud rate to 9600, the data format to one start bit, eight data bits, and one stop bit, and the parity pit is disabled. The same settings are used for both SCI modules. The transmitter and receiver for each module are enabled, and the receiver full interrupt is enabled. This function initializes the global structures `RxBuffer` and `TxBuffer`. The structure `RxBuffer` contains information on the data received from each SCI module, and the structure `TxBuffer` contains information used to transmit data with each SCI module.
- void update_serial_timers(void): The `update_serial_timers` function is called approximately every 10[ms] during the interrupt service routine for the real-time interrupt. The update function decrements the value of the timer variables in the global structures `RxBuffer` and `TxBuffer`.
- interrupt void sci0_interrupt (void): The `sci0_interrupt` is the interrupt service routine for the SCI0 module and is called for each serial interrupt. The ISR determines which serial interrupt has occurred and then processes the interrupt. If the interrupt indicates the receive data register is full, the function pulls the data from the receive register and stores it in the global structure `RxBuffer`. If the

interrupt indicates the transmit data register is empty, the function pulls the next byte to transmit from the global structure `TxBuffer` and sends the byte.

- interrupt void sci1_interrupt (void): The `sci1_interrupt` is the interrupt service routine for the SCI1 module and is called for each serial interrupt. The ISR determines which serial interrupt has occurred and then processes the interrupt. If the interrupt indicates the receive data register is full, the function pulls the data from the receive register and stores it in the global structure `RxBuffer`. If the interrupt indicates the transmit data register is empty, the function pulls the next byte to transmit from the global structure `TxBuffer` and sends the byte.
- UINT8 send_serial(UINT8 Index, UINT8* Data, UINT16 NumData): The `send_serial` function transmits data serially using the specified SCI module. This function does not actually transmit the data. It sets up the structures necessary to allow the SCI interrupt service routine to transmit the data. This function loads the global structure `TxBuffer` with the data to transmit and initializes the transmit timer. It clears the global structure `RxBuffer` in anticipation of receiving a response to the transmitted data and enables the transmit interrupt to allow the data to transmit. The first parameter `Index` is an 8-bit unsigned integer that specifies the desired SCI module. The second parameter `Data` is a pointer to an array of 8-bit unsigned integers that contain the data to transmit. The third parameter `NumData` is the number bytes in the `Data` array. The function return value is an 8-bit unsigned integer that will be zero if the function is successful and an error code if the function fails.

- UINT8 send_serial_check(UINT8 Index): The `send_serial_check` function checks the state of the serially transmitted data. The process for transmitting data is started with the `send_serial` function, but the data is actually transmitted during the SCI interrupt service routine. The `send_serial_check` function is used to check if the serial transmit time has expired, which indicates there are bytes that have not been transmitted and the send has failed. The parameter `Index` is an 8-bit unsigned integer that specifies the desired SCI module. The function return value is an 8-bit unsigned integer that will be zero if the function is successful and an error code if the function fails.
- UINT8 receive_serial (UINT8 Index, UINT8* Data, UINT16* NumData, UINT16 MaxNumData): The `receive_serial` function gets one byte of received data from the specified SCI module. This function is intended to be called repeatedly until all the desired bytes are received. Each function call reads one byte from the global structure `RxBuffer`. The received bytes are stored in `RxBuffer` during the interrupt service routine that is triggered when the SCI modules received data register is full. The first parameter `Index` is an 8-bit unsigned integer that specifies the desired SCI module. The second parameter `Data` is a pointer to an array of 8-bit unsigned integers that will hold the received data. The third parameter `NumData` is the number bytes in the `Data` array. The forth parameter `MaxNumData` is the maximum number of byes allowed in the `Data` array. The function return value is an 8-bit unsigned integer that will be zero if the function is successful and an error code if the function fails.

- UINT8 receive_serial_all (UINT8 Index, UINT8* Data, UINT16* NumData, UINT16 MaxNumData): The `receive_serial_all` function gets every byte of received data from the specified SCI module. This function reads the bytes from the global structure `RxBuffer`. The received bytes are stored in `RxBuffer` during the interrupt service routine that is triggered when the SCI modules received data register is full. The first parameter `Index` is an 8-bit unsigned integer that specifies the desired SCI module. The second parameter `Data` is a pointer to an array of 8-bit unsigned integers that will hold the received data. The third parameter `NumData` is the number bytes in the `Data` array. The forth parameter `MaxNumData` is the maximum number of byes allowed in the `Data` array. The function return value is an 8-bit unsigned integer that will be zero if the function is successful and an error code if the function fails.

Serial I/O Implementation

Several global variables are used to manage the SCI. The first variable is `RxBuffer`. This variable is an array of two structures, one for each SCI module, and each structure contains information on the data received from the SCI module. The `Data` variable is an array of bytes that holds the received data. The `ReadOffset` variable is set to the index of the first received byte in the `Data` array, and the `WriteOffset` variable is set to the index after the last received byte. The `ReadOffset` is used to read received data from the buffer and is incremented after each read. The `WriteOffset` is used to write data to the buffer and is incremented after each write. The `Timeout` variable is set to the maximum amount of time the system will wait to receive serial data.

```
typedef struct
{
  UINT8 Data[SERIAL_DATA_SIZE]; //Transmit/receive buffers.
  UINT8 ReadOffset;  //Start of transmit/receive data.
  UINT8 WriteOffset; //End of transmit/receive data.
  UINT16 Timeout;    //Total time for transmit/receive.
} serial_buffer;
serial_buffer volatile RxBuffer[SERIAL_NUM_PORTS];
serial_buffer volatile TxBuffer[SERIAL_NUM_PORTS];
```

The second global variable is `TxBuffer`. This variable is an array of two structures, one for each SCI module, and uses exactly the same variables as the `RxBuffer`. The `Data` variable is an array of bytes that holds the data to transmit. The `ReadOffset` variable is set to the index of the first transmit byte in the `Data` array, and the `WriteOffset` variable is set to the index after the last transmit byte. The `ReadOffset` is used to read the data to transmit from the buffer and is incremented after each read. The `WriteOffset` is used to write data to the buffer and is incremented after each write. The `Timeout` variable is set to the maximum amount of time the system will wait to transmit serial data.

The SCI initialization begins by setting the serial baud rate. The baud rate is 13-bits and is split over two registers. The low register is set to 163, and the high register is set to 0. This configuration sets the divisor to 16*163, which gives a total divisor value of 2608. This divisor is used to scale the bus clock frequency to the SCI baud rate. With this configuration, the baud rate is 9585[bps], which is the configuration closest to the desired 9600[bps] with a 25[MHz] bus clock. The first SCI control register is used to set the data format to one start bit, eight data bits, and one stop bit, and to disable the parity bit. The second control register is used to enable the transmitter and receiver and to enable the receive data register full interrupt. This initialization is applied to both SCI ports.

```
//Configure SCI0.
SCI0CR1_M = 0;   //Set the data format (1 start,8 data,1 stop).
SCI0CR1_PE = 0;  //Set to no parity.
SCI0CR2_TE = 1;  //Enable the transmitter.
SCI0CR2_RE = 1;  //Enable the receiver.
SCI0BDH = 0;     //Set the baud rate to ~9600[bps].
SCI0BDL = 163;
SCI0CR2_RIE = 1; //Enable receive data register full interrupt.
//Configure SCI1.
SCI1CR1_M = 0;   //Set the data format (1 start,8 data,1 stop).
SCI1CR1_PE = 0;  //Set to no parity.
SCI1CR2_TE = 1;  //Enable the transmitter.
SCI1CR2_RE = 1;  //Enable the receiver.
SCI1BDH = 0;     //Set the baud rate to ~9600[bps].
SCI1BDL = 163;
SCI1CR2_RIE = 1; //Enable receive data register full interrupt.
```

An interrupt is used for transmitting and receiving data on each SCI port. Each SCI interrupt requires an interrupt service routine (ISR). The ISR for the first SCI is vector number 20 in the interrupt vector table, and the ISR is located at the addresses 0xFFD6 and 0xFFD7. The ISR for the second SCI is vector number 21, and is located at the addresses 0xFFD4 and 0xFFD5. The interrupts for this project are implemented as an array of functions that make up an interrupt table. To add the ISRs for the SCI, the name of each function is added to the location in the array corresponding to the correct address.

```
const tIsrFunc _vect[] @0xFF80 =
{ /* Interrupt table */
  …,              //other interrupts.
  …,
  sci1_interrupt, //The ISR for SCI1.
  sci0_interrupt, //The ISR for SCI0.
  …,
  …               //other interrupts.
};
```

The ISR for the SCI is called each time an SCI interrupt occurs. The ISR is declared using the `interrupt` qualifier, and this tells the compiler to generate a return from interrupt at the end of the service routine. The ISR is used to update the SCI transmit and receive buffers. For each interrupt, the ISR uses the SCI status register to determine which serial interrupt has occurred and then processes the interrupt. This function watches for the receive data register full interrupt and the transmit data register empty interrupt.

```
interrupt void sci0_interrupt (void)
{
  if (SCI0SR1_RDRF)
  {
    //The receive data register is full. Receive data.
  }
  if (SCI0SR1_TDRE)
  {
    //The transmit data register is empty. Transmit data.
  }
}
```

When the receive data register is full, there is received data is in the SCI data registers. The data is split over the SCI data high register and the SCI data low register. For this project, the SCI modules were configured for eight data bits. Thus, the received data is eight bits and fits in the SCI data low register. The SCI data high register is not used. The data is read from the low register and stored in a local variable. Reading the data resets the receive data register full flag.

```
//Get the data and reset Receive Data Register Full flag.
Data = SCI0DRL;
```

After the ISR reads the data, the next position in the global `RxBuffer` is calculated. The `RxBuffer` contains a data array that acts as a ring buffer. The variable `ReadOffset` is the index of the first received byte in the data array, and the variable `WriteOffset` is the index after the last received byte in the array. The `WriteOffset` points to the next available location for storing the received data. The received byte is stored at the index `WriteOffset`, and then `WriteOffset` is moved to the next position. The next position for storing a received byte is calculated by adding 1 to the

`WriteOffset` and performing a modulus division with the maximum number of bytes in the buffer.

```
//Find next position in the receive buffer (WriteOffset).
Offset = (UINT8)((RxBuffer[0].WriteOffset+1) % SERIAL_DATA_SIZE);
```

If the result is equal to the `ReadOffset`, then the receive buffer is full and the received byte can not be added. If they are not equal, then a position is available in the receive buffer for the received byte. After calculating the next position in the global `RxBuffer`, the received serial byte is added to the buffer at the current `WriteOffset`. Then the `WriteOffset` is moved to the next position for the next received byte. The `Timeout` variable is reset to the maximum serial timeout value. When the system is receiving serial data, the `Timeout` is reset and never reaches zero. When not receiving serial data, the `Timeout` is decremented during the real-time interrupt.

```
//Check if the receive buffer is full.
if (Offset != RxBuffer[0].ReadOffset)
{
  //Receive buffer is not full. Add data to the receive buffer.
  RxBuffer[0].Data[RxBuffer[0].WriteOffset] = Data;
  //Move WriteOffset to the next position.
  RxBuffer[0].WriteOffset = Offset;
  //Reset the receive timer.
  RxBuffer[0].Timeout = SERIAL_TIMEOUT;
}
```

A similar process is used when the transmit data register is empty. When the transmit data register is empty, the next serial byte to be written can be transmitted. The next transmit byte is located at the `ReadOffset` in the global `TxBuffer`. If the `ReadOffset` is equal to the `WriteOffset`, then there is no data in the transmit buffer. Otherwise, the byte located at `ReadOffset` is loaded in the SCI data register, and the `ReadOffset` is moved to the next position. The `Timeout` variable is reset to the maximum serial timeout value. When the system is transmitting serial data, the `Timeout` is reset and never reaches zero. When not transmitting data, the `Timeout` is decremented during the real-time interrupt.

```
//Check if serial transmit data is present.
if (TxBuffer[0].ReadOffset != TxBuffer[0].WriteOffset)
{
  //Write the data in the transmit buffer (ReadOffset).
  SCI0DRL = TxBuffer[0].Data[TxBuffer[0].ReadOffset];
  //Move ReadOffset to the next position.
  TxBuffer[0].ReadOffset =
    (UINT8)((TxBuffer[0].ReadOffset+1) % SERIAL_DATA_SIZE);
  //Reset the transmit timer.
  TxBuffer[0].Timeout = SERIAL_TIMEOUT;
}
```

The receive data register full interrupt is enabled during the serial initialization and remains enabled throughout program execution. However, the transmit data register empty interrupt is only enabled when there is data to send. The transmit interrupt is enabled after the `TxBuffer` is loaded with transmit data, and the interrupt is disabled after all of the bytes have been transmitted. During the transmit data register empty interrupt, the ISR determines if there is any data to send. If the `ReadOffset` is equal to the `WriteOffset`, the buffer is empty and the interrupt is disabled.

```
//Nothing to transmit. Disable the transmit interrupt.
SCI0CR2_SCTIE = 0;
```

The interrupt service routines for the SCI modules are responsible for writing the received serial data to the `RxBuffer` and reading the transmit data from the `TxBuffer`. Other methods outside the interrupt service routines are responsible for reading the received data from the `RxBuffer` and writing the transmit data to the `TxBuffer`. These methods are used by the system to send and receive serial data.

The process for sending serial data begins by clearing the `TxBuffer`. The `TxBuffer` is cleared by setting the `ReadOffset` to the `WriteOffset`. This indicates the buffer is empty. After clearing the buffer, the data to transmit is loaded in the `TxBuffer`. One byte is added at the index `WriteOffset`, and then the `WriteOffset` is moved to the next position. This process is repeated for every byte. The `RxBuffer` is also cleared to make room for the data received in response to the transmitted data. After configuring the buffers, the transmit interrupt is enabled to allow the interrupt service routine to process the data.

```
//Clear the TxBuffer.
TxBuffer[0].ReadOffset = TxBuffer[0].WriteOffset;
//Set the transmit timeout.
TxBuffer[0].Timeout = SERIAL_TIMEOUT;
//Load the TxBuffer.
for (i = 0; i < NumData && retval == OK; i++)
{
  //Find next position in transmit buffer (WriteOffset).
  Offset = (UINT8)((TxBuffer[0].WriteOffset+1) %
             SERIAL_DATA_SIZE);
  if (Offset != TxBuffer[0].ReadOffset)
  {
    //Add the data byte to the transmit buffer.
    TxBuffer[0].Data[TxBuffer[0].WriteOffset] = Data[i];
    //Move WriteOffset to the next position.
    TxBuffer[0].WriteOffset = Offset;
  }
}
//Clear the RxBuffer.
RxBuffer[0].ReadOffset = RxBuffer[0].WriteOffset;
//Set the receive timeout.
RxBuffer[0].Timeout = SERIAL_TIMEOUT;
//Enable transmitter interrupt.
SCI0CR2_SCTIE = 1;
```

To determine if the data has finished transmitting, the `ReadOffset` and `WriteOffset` are compared and the `Timeout` variable is checked. If the `ReadOffset` is not equal to the `WriteOffset`, there is data in the `TxBuffer` and the transmission is not complete. If the `Timeout` is equal to zero, the transmit time has expired. When there is data remaining and the time has expired, an error has occurred and the data was not successfully transmitted. Otherwise, the system is still processing the transmission.

```
//Check if serial data is present and check transmit time.
if (TxBuffer[0].ReadOffset != TxBuffer[0].WriteOffset &&
    TxBuffer[0].Timeout == 0)
{
  //Transmission is not complete and transmit time has expired.
  retval = ERROR_TIMEOUT;
}
```

The process for receiving serial data begins by checking the state of the `RxBuffer`. The `ReadOffset` to the `WriteOffset` are compared. If the `ReadOffset` is not equal to the `WriteOffset`, there is data in the `RxBuffer` and data has been received. The `Timeout` variable is also checked. If the receive time has not expired, the received data is read from the `RxBuffer`. One byte is read at the index `ReadOffset`, and then the `ReadOffset` is moved to the next position. This process is repeated until all bytes are read.

```
while (RxBuffer[0].ReadOffset != RxBuffer[0].WriteOffset &&
       RxBuffer[0].Timeout != 0 &&
       NumData < MaxNumData)
{
  //Add a byte to the Data buffer.
  Data[NumData] = RxBuffer[0].Data[RxBuffer[0].ReadOffset];
  //Move ReadOffset to the next position.
  RxBuffer[0].ReadOffset =
    (UINT8)((RxBuffer[0].ReadOffset+1) % SERIAL_DATA_SIZE);
  //Increment the number of bytes in the Data buffer.
  NumData++;
}
```

Ethernet

An Ethernet connection is used to communicate with the embedded system inside the ICI-3 optics system enclosure. The network includes a personal computer running the ICI software interface, the microcontroller, and the ICI camera. The ICI software interface uses the Ethernet link to establish a connection with the microcontroller, and separately establishes a connection with the ICI camera.

Ethernet Standard

Ethernet is a networking standard that defines the physical network interface and specifies the rules and structures for transmitting data. A network consists of two or more computers that need to communicate, and one of those computers may be an embedded system. The computers are physically connected using hardware defined by the Ethernet standard. The network design and implementation is divided into modules that work together to handle network communications. Each module is responsible for a single task or a small set of related tasks, and they are visualized as being stacked in layers, one above another. These modules form the network protocol stack (Figure 8) [3].

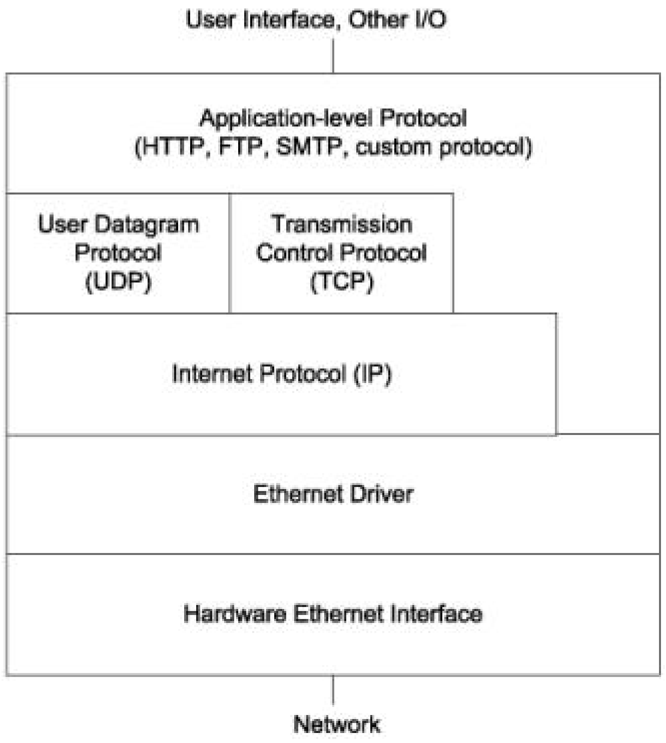

User Interface, Other I/O
Application-level Protocol (HTTP, FTP, SMTP, custom protocol)
User Datagram Protocol (UDP)
Transmission Control Protocol (TCP)
Internet Protocol (IP)
Ethernet Driver
Hardware Ethernet Interface
Network

Figure 8: Network Protocol Stack [3]

At the top of the stack is the application layer. The application is a module or modules that provide data to send on the network and use the data received from the network. The data sent by the application follows a protocol, or set of rules, that enables the application at the receiving end to understand the received data. The application may use a standard protocol, such as the hypertext transfer protocol (HTTP), or use a custom protocol [3].

The next layer in the Ethernet stack is the TCP and UDP layer. The transfer control protocol (TCP) adds information for error checking, flow control, and for identifying an application-level process. A simpler alternative to TCP is the user datagram protocol (UDP). UDP has optional error checking, but no flow control. Every UDP and TCP communication is between two endpoints, or sockets, and each socket has a port number and an IP address. A socket is one end of a logical connection between computers, and a socket's port number identifies the process sending or receiving the data [3].

The Internet protocol (IP) layer enables computers on the Internet to communicate with each other. Local networks that use TCP and UDP also use IP. The term TCP/IP refers to communications that use TCP, IP, and related protocols such as UDP. The IP protocol adds information for error checking, routing, and protocol identification. Hosts that support IP must also support the Internet control message protocol (ICMP), which is basic a protocol for sending messages [3].

To send an Ethernet frame on the network, the Ethernet standard uses a system of addresses. The media access control (MAC) physical address is a 48-bit number. Each network device must have a unique MAC hardware address. This address is used by the datalink layer. In a local network, a random MAC hardware address can be used as long as it is not connected on a network that has a device with the same 48-bit MAC hardware address [11]. The address resolution protocol (ARP) is used to learn the physical address that corresponds to an IP address in a local network [3].

The IP addresses are used by the IP layer to identify the sending and receiving computers. In addition, sending a message using IP may require a subnet mask, the IP address of a gateway, and the IP address of a domain-name server. Subnetting is the process of dividing a network into groups called subnetworks, or subnets. A subnet mask is used to determine the bits in the host address are the subnet ID. A gateway, or router, enables an Ethernet network to communicate with computers in other networks, including computers on the Internet. Domain name servers enable computers to match a domain name with the IP address required to access the domain's resources [3].

Ethernet Design

The microcontroller selected for the ICI-3 embedded system is equipped with all the hardware necessary to implement an Ethernet network. The MC9S12NE64 is equipped with an Ethernet media access controller (EMAC) and an Ethernet physical transceiver (EPHY). The EMAC provides a 10/100[Mbps] Ethernet media access control function and is designed to connect to a physical layer device. The EMAC and the EPHY together implement an interface to a 10/100[Mbps] Ethernet network.

The EMAC and the EPHY form the bottom layer of the Ethernet stack, or the hardware interface. All other layers in the Ethernet standard must be implemented in the microcontroller firmware to achieve a fully functional Ethernet device. Firmware must be implemented to support the Ethernet drivers, the IP layer, the TCP and UDP layer, and the application layer.

Ethernet Implementation

Implementing the firmware for the Ethernet stack would be a monumental task. Fortunately, many products are available for free or for purchase with full Ethernet TCP/IP support. The Ethernet implementation selected for this project is OpenTCP, an open source implementation of the Ethernet TCP/IP stack. Originally developed by Viola systems, OpenTCP was released under an open source license, which allows for modification and redistribution without having to pay the original author. It includes support for the protocols ARP, ICMP, IP, UDP, and TCP. A port of OpenTCP is provided for the MC9S12NE64 at SourceForge. OpenTCP is a CodeWarrior compatible implementation that is tailored for 8-bit and 16-bit embedded processors. [11].

OpenTCP for the MC9S12NE64 is released with a fully functional sample project. This project was used as the foundation for the firmware for the ICI-3 embedded system. OpenTCP includes files to support many of the Ethernet protocols, but not every protocol was required for this project, and the extra files were removed from the project before beginning the ICI implementation. The protocol files remaining in the project are those that support ARP, ICMP, IP, UDP, and TCP.

The only change to the OpenTCP implementation was to the Ethernet IP address. The address information was changed to match the subnet used by the Ethernet based ICI-3 camera. All address information for the embedded system is located in the file *address.c*. The default IP address was 192.168.2.3. The IP address was changed to 192.168.250.3.

TCP Control System

The TCP control system is a custom implementation of the application layer of the Ethernet stack. The TCP control system provides data to send on the network and uses the data received from the network. The TCP control system sends the received data to the ICI control system for processing, and the ICI control system sends data to the TCP control system for transmission.

TCP Control System Design

The TCP control system is based on the HTTP implementation provided with the OpenTCP project. The HTTP implementation is located in the OpenTCP files `http_server.h`, `http_server.c`, and `https_callbacks.c`. This design uses an array of structures to manage multiple HTTP connections. The TCP control system closely follows this design and is capable managing multiple ICI connections.

TCP Control System Functionality

In the ProjectICI CodeWarrior project, the TCP control system implementation is located in the files `tcp_server.h` and `tcp_server.c`. These files handle the initialization and control of the TCP control system. The TCP control system is capable of handling multiple ICI connections. For this project, only one connection is needed and the system was modified to restrict the number of connections.

- INT8 tcp_server_init(void): The `tcp_server_init` function is called during the ICI system initialization in the file `main.c` to initialize the TCP control system. The initialization clears the global variable `tcps`. This variable is an array of structures that contain information on the state of the associated TCP socket. This function gets a TCP socket for each structure and then configures the sockets to start listening for TCP traffic. The function return value is an 8-bit signed integer that specifies the index of the last TCP socket. For this project, only one connection is needed and there is only one element in the array of structures.
- void tcp_server_run(void): The `tcp_server_run` function is called once during each pass through the main program loop in the file `main.c`. This function updates the TCP control system. It uses the global array of structures `tcps` to check the state of each TCP socket. If the socket has data to send, this function sends the data.
- INT32 tcp_server_eventlistener (INT8 cbhandle, UINT8 event, UINT32 par1, UINT32 par2): The `tcp_server_eventlistener` function is the event listener for the TCP control system. Events are generated by the OpenTCP stack implementation, and this function is called for each event. This function determines which event was generated and for which TCP socket. It finds the global structure associated with this socket in the `tcps` array, and then processes the event. The first parameter `cbhandle` is an 8-bit signed integer that specifies the handle of the socket that generated the event. The second parameter `event` is an 8-bit unsigned integer that contains the type of event. The third and forth

variables `par1` and `par2` are 32-bit unsigned integers. These variables are event dependent and their use is determined by the type of event. The function return value is a 32-bit signed integer that will be a positive value if the function is successful and a negative value if the function fails.

- void tcp_server_clearsession (UINT8 ses): The `tcp_server_clearsession` function clears the information in the global structure associated with a TCP socket in the `tcps` array. The function zeros the information associated with the socket, but it leaves the socket active. The parameter `ses` is an 8-bit unsigned integer that specifies the index of the socket in the `tcps` array.
- void tcp_server_deletesession (UINT8 ses): The `tcp_server_deletesession` function frees the global structure associated with a TCP socket in the `tcps` array. The function zeros the information associated with the socket, and it frees the socket. The parameter `ses` is an 8-bit unsigned integer that specifies the index of the socket in the `tcps` array.
- void tcp_server_searchsession (UINT8 soch): The `tcp_server_searchsession` function finds a TCP socket in the global `tcps` array. The function searches through the structures until it finds the correct TCP socket. The parameter `soch` is an 8-bit unsigned integer that specifies the handle of the socket in the `tcps` array.
- INT16 tcp_server_bindsession (UINT8 soch): The `tcp_server_bindsession` function reserves a TCP socket in the global `tcps` array. The function searches through the structures until it finds the correct TCP socket. The parameter `soch` is an 8-bit unsigned integer that specifies the handle of the socket in the `tcps` array.

The function return value is a 16-bit signed integer that specifies the index of the TCP socket.

- void tcp_server_activatesession (UINT8 ses): The `tcp_server_activatesession` function activates the global structure associated with a TCP socket in the `tcps` array. The function sets the socket's state to active. The parameter `ses` is an 8-bit unsigned integer that specifies the index of the socket in the `tcps` array.
- void tcp_server_send (UINT8 ses, UINT8* Data, UINT16 NumData): The `tcp_server_send` function sends data with a TCP socket in the `tcps` array. This function does not actually transmit the data. It sets up the structures necessary to allow the `tcp_server_run` function to transmit the data. This function loads the global structure `tcps` with the data to transmit. The first parameter `ses` is an 8-bit unsigned integer that specifies the index of the socket in the `tcps` array. The second parameter `Data` is a pointer to an array of 8-bit unsigned integers that contain the data to transmit. The third parameter `NumData` is the number bytes in the `Data` array.

TCP Control System Implementation

The TCP control system uses the global array of structures `tcps` to manage the TCP system. These structures contain information on each TCP socket. The `State` variable is the current state of the TCP socket. The socket can be free, reserved, or active. The state free means the socket has not been reserved and is available for use, the state reserved means the socket is in use but is not currently active, and the state active means the socket is reserved and active. The `OwnerSocket` variable is the handle to the TCP socket. This variable is used to access the socket. The `Data` array is the data transmitted or received over the socket, and `NumData` is the number of bytes in the array. The `Offset` variable is the number of bytes transmitted from the `Data` array, and `Unacked` is the number of bytes that have not been acknowledged after a transmission.

```
typedef struct
{
  UINT8 State;                //Server state.
  INT8 OwnerSocket;           //TCP socket.
  UINT8 Data[TCP_DATA_SIZE]; //Transmit/receive buffer.
  UINT16 NumData;             //Number of bytes in data buffer.
  UINT16 Offset;              //Number of transmitted bytes.
  UINT16 Unacked;             //Number of unacknowledged bytes
} tcps_server_state;
tcps_server_state volatile tcps[NO_OF_TCP_SESSIONS];
```

The TCP control system initialization begins by clearing the global variable `tcps`. This variable is an array of structures that contain information on the state of the associated TCP socket. For this project, there is only one element in the array. The state of the socket is set to free to indicate the socket has not been reserved and is available for use. The socket handle is set to -1, which is an invalid handle value, and the rest of the information is set to 0.

```
tcps[i].State = TCPS_STATE_FREE; //Set state to free.
tcps[i].OwnerSocket = -1;        //Set handle to invalid.
(void)memset(tcps[i].Data, 0, TCP_DATA_SIZE); //Set data to 0.
tcps[i].NumData = 0;             //Set number of bytes to 0.
tcps[i].Offset = 0;              //Set data offset to 0.
tcps[i].Unacked = 0;             //Set unacked bytes to 0.
```

After initializing the socket, a valid TCP socket is requested. If the request is successful, the socket is assigned to the `tcps` element, and then the initialization configures the socket to start listening for TCP traffic. The socket request specifies an event listener for the socket, and the listen request specifies the socket’s port. For this project, the port is set to 1024.

```
//Get a TCP socket.
soch = tcp_getsocket(TCP_TYPE_SERVER, TCP_TOS_NORMAL,
       TCP_DEF_TOUT, tcp_server_eventlistener);
//Assign the socket to the server.
tcps[i].OwnerSocket = soch;
//Set the TCP socket to listening.
soch = tcp_listen(tcps[i].OwnerSocket, TCP_SERVER_PORT);
```

The TCP sockets are updated every pass through the main program loop. During each update, the update function checks if the socket is listening. If it is not listening, the update sets it to listening. Next, the update checks if the socket is active. The socket will be active if a valid TCP connection has been established. If the socket is listening and active, the update checks if the socket is waiting for an acknowledgement. The system must wait for all transmissions to be acknowledged before processing any additional commands. If the acknowledge was received, the update checks if the server has data to send. If there is data, the data is loaded into the system buffer and the data is sent.

```
if(tcp_getstate(tcps[ses].OwnerSocket) < TCP_STATE_LISTENING)
{ //Socket is not listening. Set socket to listening.
  (void)tcp_listen(tcps[ses].OwnerSocket, TCP_SERVER_PORT);
}
if(tcps[ses].State != TCPS_STATE_ACTIVE)
{ //Socket is not active. Do nothing.
}
if(tcps[ses].Unacked != 0)
{ //Socket has unacked data. Waiting for ack. Do nothing.
}
if(tcps[ses].NumData == 0)
{ //Socket has no data to send. Do nothing.
}
//Load the network buffer with the data.
len = tcps[ses].NumData - tcps[ses].Offset;
(void)memcpy(&net_buf[TCP_APP_OFFSET],
             tcps[ses].Data+tcps[ses].Offset, len);
//Send the data.
len = tcp_send(tcps[ses].OwnerSocket, &net_buf[TCP_APP_OFFSET],
               NETWORK_TX_BUFFER_SIZE - TCP_APP_OFFSET, len);
```

The data sent by the TCP control system is generated by the ICI control system. When the ICI control system is ready to send data, it passes the data and the number of bytes to the TCP control system. The TCP control system loads the data in the data buffer for the TCP socket and sets the number of bytes. The data is transmitted the next time the main program loop updates the TCP control system.

```
//Load the TCP server buffer.
(void)memcpy(tcps[ses].Data, Data, NumData);
//Set the number of bytes.
tcps[ses].NumData = NumData;
```

The rest of the TCP control system is driven by events. An event listener is registered when the socket is requested, and the system events are managed and generated by the OpenTCP implementation. For each event, the event listener determines the type of event and then processes the event. The following are the actions taken for each event:

- TCP_EVENT_CONREQ: This is the TCP socket connect request event. A connect request is generated when another device asks to connect to the embedded device. For a connect request, the event listener reserves a TCP socket. The socket is not active until the connection is complete.

  ```
  //Reserve a socket.
  session = tcp_server_bindsession(cbhandle);
  ```

- TCP_EVENT_ABORT: This is the TCP socket abort event. An abort occurs when the OpenTCP implementation detects an error. An abort is requested when the system times out, invalid data is detected, or the system needs to reset. In any case, the event listener clears the information associated with the socket to reset the socket.

```
//Clear the TCP server.
tcp_server_clearsession((UINT8)session);
```

- TCP_EVENT_CONNECTED: This is the TCP socket connected event. The connected event is generated after a valid connection is established. This event occurs after the initial connect request event. On connect, the TCP socket is activated indicating it is ready to send and receive data.

```
//Activate the TCP server.
tcp_server_activatesession((UINT8)session);
```

- TCP_EVENT_CLOSE: This is the TCP socket close event. The close event occurs when the socket is closed. The socket is closed on disconnect or after a period of no activity. The event listener clears the information associated with the socket and frees the socket.

```
//Delete the TCP server.
tcp_server_deletesession((UINT8)session);
```

- TCP_EVENT_ACK: This is the TCP socket acknowledge event. An acknowledgment is part of the OpenTCP handshaking implementation and is received in response to a transmission. For this project, the acknowledge event is used to detect when a data transmission is complete. After sending data, an acknowledge event is generated, and the event listener processes the data transmission. The event listener adds the number of bytes sent to the total bytes sent and zeros the number of unacknowledged bytes. If the total number of bytes sent is equal to the total number of bytes in the original data, the transmission is complete and the information associated with the socket is cleared.

```
//Move the send offset forward. Clear the unacked bytes.
tcps[session].Offset += tcps[session].Unacked;
tcps[session].Unacked = 0;
//Check if the send is complete.
if( tcps[session].Offset >= tcps[session].NumData)
{
  //Finished sending the data. Clear the server.
  tcp_server_clearsession((UINT8)session);
}
```

- TCP_EVENT_DATA: This is the TCP socket data event. The data event is generated when the system receives data. The data received is processed by the ICI control system. The event listener reads the received data from the network buffer and passes the data to the ICI control system.

```
//Process the data.
//par1 is the length of the data.
if (par1 != 0)
{
  //Read in the data.
  for (i = 0; i < par1 && i < TCP_DATA_SIZE; i++)
  {
    Data[i] = RECEIVE_NETWORK_B();
  }
  //Process the data.
  (void)ici_process_message(ICI_TYPE_TCP, (UINT8)session, Data,
                            (UINT16)par1);
}
```

- TCP_EVENT_REGENERATE: This is the TCP socket regenerate event. A regenerate event occurs when the system needs to resend the previously transmitted data. Data is normally transmitted when the TCP control system is updated in the main program loop. In the case of an error, the event listener handles the transmission. The last section of data transmitted is reloaded into the network buffer and sent again.

```
//Load the network buffer with the data.
i = tcps[session].NumData - tcps[session].Offset;
(void)memcpy(&net_buf[TCP_APP_OFFSET],
             tcps[session].Data+tcps[session].Offset, i);
//Send the data.
i = tcp_send(tcps[session].OwnerSocket, &net_buf[TCP_APP_OFFSET],
             NETWORK_TX_BUFFER_SIZE - TCP_APP_OFFSET, i);
```

ICI Control System

The ICI control system is the message interface for the ICI-3 embedded system. The ICI control system processes incoming messages and generates system responses. These messages are used to read, configure, and control the digital, analog, and serial components of the ICI-3 optics system.

ICI Control System Design

The ICI control system defines a specific format for the system messages. This format specifies a received message consists of a command, length, and data. The command is the system command and corresponds to a particular action in the ICI control system. The length is the number of bytes in the data, and the data is the additional information needed to support the command. The command and length are 16-bit unsigned integers, and up to 128 bytes of data may be associated with each command.

Table 4: Expected Format of a Received Message

Command		Length		Data		

The ICI control system also defines the message format for all message responses. This format specifies a message response consists of a command, length, acknowledge field, and data. The command corresponds to the command processed in the received message. The length is the number of bytes in the acknowledge field and data. The acknowledge field specifies if the system was successful in processing the received message, and the data is the response to the received message. The command and length are 16-bit unsigned integers, the acknowledge field is an 8-bit unsigned integer, and up to 128 bytes of data may be used in the message response.

Table 5: Expected Format for a Message Response

Command		Length		ACK	Data		

The ICI control system assumes little-endian format for the transmitted and received data. Endianness is the byte ordering used to store data. Big-endian byte ordering stores the most significant byte first and significance decreases with increasing memory addresses. Little-endian byte ordering stores the least significant byte first and significance increases with increasing memory addresses. For example, the hex value 0x1234 is stored in memory as `12 34` in big-endian format, and `34 12` in little-endian format.

The MC9S12NE64 uses big-endian byte ordering. The Intel processor in the computer used to develop the software interface uses little-endian byte ordering. With opposite byte orders, the bytes must be swapped by either the embedded system or the computer to get the byte orders to match. For this project, the bytes are swapped from big-endian to little-endian in the embedded system.

The ICI control system is generically designed to receive data from any interface. The system remembers the interface that sent the message and uses that interface to send the message response. In the current implementation, there is support for sending and receiving messages with the TCP control system or the serial interface. For this project, the serial interface was not used for messaging, and the ICI control system only receives messages from the TCP control system.

When the TCP control system receives data, it passes the data to the ICI control system. The ICI control system loads the data into a message structure. During each pass through the main program loop, the ICI control system is updated and the message is processed. During each update, the update function checks the state of the current message. When the ICI control system is finished processing the message, it generates a message response with the results and passes the response to the TCP control system.

ICI Control System Functionality

In the ProjectICI CodeWarrior project, the ICI control system implementation is located in the files `ici.h` and `ici.c`. These files handle the initialization and control of the ICI control system. The ICI control system is generically designed to receive data from any interface. For this project, only the TCP control system is used.

- void ici_init(void): The `ici_init` function is called during the ICI system initialization in the file `main.c` to initialize the ICI control system. The initialization configures the global structures `message_info` and `message`. The structure `message_info` contains information on the current system message's state and interface, and `message` contains information on the message data.

- void ici_run(void): The `ici_run` function is called once during each pass through the main program loop in the file `main.c`. This function updates the ICI control system and processes the current system message.
- void ici_clearmessage (void): The `ici_clearmessage` function clears the information in the global structures `message_info` and `message`.
- UINT8 ici_process_message(UINT8 From, UINT8 Index, UINT8* Data, UINT16 NumData): The `ici_process_message` function processes the incoming data and parses the information into the `message_info` and `message` structure. If a message is already in progress, the function generates a message indicating the system is busy. The first parameter `From` is an 8-bit unsigned integer that indicates which type of interface, the TCP control system or the serial interface, is sending the data. The second parameter `Index` is an 8-bit unsigned integer that specifies which interface sent the data. The third parameter `Data` is a pointer to an array of 8-bit unsigned integers that contain the incoming data. The forth parameter `NumData` is the number bytes in the `Data` array. The function return value is an 8-bit unsigned integer that will be zero if the function is successful and an error code if the function fails.
- UINT8 get_device_info(ici_device_info* device_info): The `get_device_info` function retrieves the ICI system information. This information includes the firmware version, the ATD converter resolution, the hatch control system's open timeout, and the serial system's serial timeout. The parameter `device_info` is a pointer to an `ici_device_info` structure that will hold the ICI system

information. The function return value is an 8-bit unsigned integer that will be zero if the function is successful and an error code if the function fails.

- UINT8 swap_device_info(ici_device_info* device_info): The `swap_device_info` function converts the ICI system information to little-endian byte order. The processor uses big-endian byte order and swaps the bytes to little-endian before transmitting the data. The parameter `device_info` is a pointer to an `ici_device_info` structure that contains the ICI system information. The function return value is an 8-bit unsigned integer that will be zero if the function is successful and an error code if the function fails.
- UINT8 swap_byte_order(UINT8* Address, int Size): The `swap_byte_order` function swaps the byte order of the given data. The first parameter `Address` is a pointer to an array of 8-bit unsigned integers that contain the data to swap. The second parameter `Size` is an integer that specifies the number of bytes in the data array. The function return value is an 8-bit unsigned integer that will be zero if the function is successful and an error code if the function fails.

ICI Control System Implementation

Several global structures and variables are used to manage the ICI control system. The first structure is `message_info`. This structure contains information on the current system message state and interface. The `State` variable holds the current state of the message. The message state can be none, new, waiting, or finished. The state none indicates there is no message, the state new indicates there is a new message, the state waiting indicates a message is waiting for information, and the state finished indicates the message is finished. The `From` variable is set to the type of interface that sent the message. The `Index` variable is the specific interface within that type that sent the message.

```
typedef struct
{
  UINT8 State; //Message state.
  UINT8 From;  //Message interface.
  UINT8 Index; //Index of message interface.
} ici_message_info;
ici_message_info message_info;
```

The second ICI control system structure is `message`. This structure contains the actual message information. The `Command` variable is the system command and corresponds to a particular action in the ICI control system. The `NumData` variable is the number of bytes in the data, and the `Data` variable is an array of bytes with the additional information needed to support the command. The `Data` array is limited to 128 bytes.

```
typedef struct
{
  UINT16 Command;             //Command.
  UINT16 NumData;             //Number of bytes in Data.
  UINT8 Data[ICI_DATA_SIZE]; //Data.
} ici_message;
ici_message message;
```

The third ICI control system structure is `device_info`. This structure contains the embedded system information. The `FirmwareVersion` is the current version of the embedded system firmware. The `AtdResolution` is the current resolution of the ATD converter. The `HatchOpenTimeout` is the maximum time in milliseconds the hatch is allowed to be open, and the `SerialTimeout` is the maximum time in milliseconds the system waits to transmit or receive serial data.

```
typedef struct
{
  UINT32 FirmwareVersion;  //Firmware version.
  UINT16 AtdResolution;    //ATD resolution.
  UINT16 HatchOpenTimeout; //Hatch open timeout [ms].
  UINT16 SerialTimeout;    //Serial timeout [ms].
  UINT8 ExtraB[54];        //Extra bytes.
} ici_device_info;
```

The firmware version is 4 bytes, and the bytes indicate the major version, the minor version, the release version, and the build version. The build version is incremented for every test build of the firmware. Once testing is complete and the firmware is verified, the build number is set to 0 and the release version is incremented. The minor version is only incremented for significant firmware changes, and the major version stays the same for the life of the ICI-3 embedded system.

The ICI control system initialization begins by clearing the system message. The ICI control system is designed to process one message at a time. The initialization clears the message state and interface, and then clears the message command, length, and data.

```
(void)memset(&message_info, 0, sizeof(ici_message_info));
(void)memset(&message, 0, sizeof(ici_message));
```

When the TCP control system receives data, it passes the data to the ICI control system. If a message is already in progress, the ICI control system does not process the new message and responds with a busy message. A busy message consists of the original message command and an acknowledge field with the busy error. The original message is copied into a message header, and the length in the message header is set to the number of bytes in the busy response. A busy response only includes a single acknowledge byte, and the length is always 1. The endian order of the length is swapped to little-endian, and the busy header is copied into the final message response. The acknowledge field is set to the busy error to finish the message, and the ICI control system sends the busy response.

```
//Copy raw message command into busy response header.
(void)memcpy(&busy_header, Data, sizeof(busy_header.Command));
//Set number of bytes to the size of the ACK/NAK byte.
busy_header.NumData = 1;
//Convert the number of bytes to little endian.
swap_byte_order((UINT8*)&busy_header.NumData,
                sizeof(busy_header.NumData));
//Copy the comand and number of bytes into the busy response.
(void)memcpy(&busy_response, &busy_header,
             sizeof(ici_message_header));
//Fill in the ACK/NAK byte in the busy response.
busy_response[sizeof(ici_message_header)] = ERROR_BUSY;
//Send response.
tcp_server_send(Index, busy_response,
                sizeof(ici_message_header)+1);
```

If the system is not busy, the ICI control system loads the data into a message structure. The message state is set to new to indicate a new message has arrived. The interface information is stored with the message and is used later during message processing when the system sends the message response. The original message is copied into the system message structure, and the command and length are swapped to big-endian format for processing.

```
//Set the message state to NEW.
message_info.State = ICI_MESSAGE_NEW;
//Set the message interface.
message_info.From = From;
message_info.Index = Index;
//Set the index of the message interface.
(void)memcpy(&message, Data, NumData);
//Convert the command and numdata to big endian.
swap_byte_order((UINT8*)&message.Command,
                sizeof(message.Command));
swap_byte_order((UINT8*)&message.NumData,
                sizeof(message.NumData));
```

During each pass through the main program loop, the ICI control system is updated and the message is processed. During each update, the update function checks the state of the current message. If the message state is new, the system assumes a new message has arrived and begins processing the message. The actions taken depend on the type of command in the system message. The following are the actions taken for each type of new message:

- ICI_COMMAND_GET_ATD_IN: This command instructs the ICI control system to retrieve the current value of the specified ATD input channel. The first byte of data in the received message is the index of the ATD input channel. The update function gets the current value of the ATD input channel and stores the result in the message data buffer. The ATD value is two bytes and the bytes are swapped to little-endian format. No further processing is necessary, and the message is moved to the finished state.

```
//Get the ATD input channel.
retval = get_atd_in(message.Data[0], (UINT16*)(message.Data+1));
if (retval == OK)
{
  //Set the number of bytes in the message response to the size
  //of the ATD input channel result (2 bytes).
  message.NumData = 2;
  //Convert the result to little endian.
  swap_byte_order(message.Data+1, message.NumData);
}
//Finished processing message.
message_info.State = ICI_MESSAGE_FINISHED;
```

- ICI_COMMAND_GET_DIGITAL_IN: This command instructs the ICI control system to retrieve the current value of the specified digital input channel. The first byte of data in the received message is the index of the digital input channel. The update function gets the current value of the digital input channel and stores the result in the message data buffer. The digital input is one byte and no byte swapping is necessary. No further processing is necessary, and the message is moved to the finished state.

```
//Get digital input channel.
retval = get_digital_in(message.Data[0], message.Data+1);
if (retval == OK)
{
  //Set the number of bytes in the message response to the size
  //of the digital input channel result (1 byte).
   message.NumData = 1;
}
//Finished processing message.
message_info.State = ICI_MESSAGE_FINISHED;
```

- ICI_COMMAND_GET_DIGITAL_OUT: This command instructs the ICI control system to retrieve the current value of the specified digital output channel. The first byte of data in the received message is the index of the digital output channel. The update function gets the current value of the digital output channel and stores the result in the message data buffer. The digital output is one byte and no byte swapping is necessary. No further processing is necessary, and the message is moved to the finished state.

```
//Get digital output channel.
retval = get_digital_out(message.Data[0], message.Data+1);
if (retval == OK)
{
  //Set the number of bytes in the message response to the size
  //of the digital output channel result (1 byte).
  message.NumData = 1;
}
//Finished processing message.
message_info.State = ICI_MESSAGE_FINISHED;
```

- ICI_COMMAND_SET_DIGITAL_OUT: This command instructs the ICI control system to set the current value of the specified digital output channel. The first byte of data in the received message is the index of the digital output channel, and the second byte is the output channel value. The update function sets the current value of the digital output channel. No further processing is necessary, and the message is moved to the finished state.

```
//Set digital output channel.
retval = set_digital_out(message.Data[0], message.Data[1]);
//Finished processing message.
message_info.State = ICI_MESSAGE_FINISHED;
```

- ICI_COMMAND_PASS_SCI0: This command instructs the ICI control system to send the specified data using the first serial interface and then wait for the serial response with the specified terminator. The first byte of data in the received message is the expected value of the last byte in the serial response to this message, and the remaining bytes are the bytes to be transmitted serially. The update function sends the data bytes. Further processing is necessary to wait for the serial response, and the message is moved to the waiting state.

```
//Pass data to SCI0 and wait for response.
retval = send_serial(0, message.Data+1, message.NumData-1);
if (retval == OK)
{
  //Wait for the serial response to this message.
  message_info.State = ICI_MESSAGE_WAITING;
}
```

- ICI_COMMAND_PASS_SCI1: This command instructs the ICI control system to send the specified data using the second serial interface and then wait for the serial response with the specified terminator. The first byte of data in the received message is the expected value of the last byte in the serial response to this message, and the remaining bytes are the bytes to be transmitted serially. The update function sends the data bytes. Further processing is necessary to wait for the serial response, and the message is moved to the waiting state.

```
//Pass data to SCI1 and wait for response.
retval = send_serial(1, message.Data+1, message.NumData-1);
if (retval == OK)
{
  //Wait for the serial response to this message.
  message_info.State = ICI_MESSAGE_WAITING;
}
```

- ICI_COMMAND_GET_HATCH_CONTROL: This command instructs the ICI control system to retrieve the state of the hatch motor control. The update function gets the current value of the hatch control and stores the result in the message data buffer. The hatch control is one byte and no byte swapping is necessary. No further processing is necessary, and the message is moved to the finished state.

```
//Get hatch control (START, STOP).
retval = get_hatch_control(message.Data+1);
if (retval == OK)
{
  //Set the number of bytes in the message response to the size
  //of the hatch control result (1 byte).
  message.NumData = 1;
}
//Finished processing message.
message_info.State = ICI_MESSAGE_FINISHED;
```

- ICI_COMMAND_SET_HATCH_CONTROL: This command instructs the ICI control system to set the state of the hatch motor control. The first byte of data in the received message is the hatch control value. The update function sets the current value of the hatch control. No further processing is necessary, and the message is moved to the finished state.

```
//Set hatch control (START, STOP).
retval = set_hatch_control(message.Data[0]);
//Finished processing message.
message_info.State = ICI_MESSAGE_FINISHED;
```

- ICI_COMMAND_GET_HATCH_STATE: This command instructs the ICI control system to get the state of the enclosure’s hatch. The update function gets the current value of the hatch state and stores the result in the message data buffer. The hatch state is one byte and no byte swapping is necessary. No further processing is necessary, and the message is moved to the finished state.

```
//Get hatch state (NONE, OPEN, CLOSED, INVALID).
retval = get_hatch_state(message.Data+1);
if (retval == OK)
{
  //Set the number of bytes in the message response to the size
  //of the hatch state result (1 byte).
  message.NumData = 1;
}
//Finished processing message.
message_info.State = ICI_MESSAGE_FINISHED;
```

- ICI_COMMAND_SET_HATCH_STATE: This command instructs the ICI control system to set the state of the enclosure's hatch and wait for the hatch to reach the specified state. The first byte of data in the received message is the hatch state value. The update function sets the current value of the hatch state. Further processing is necessary to wait for the hatch to reach the desired state, and the message is moved to the waiting state.

```
//Set hatch state (OPEN, CLOSED).
retval = set_hatch_state(message.Data[0]);
//Wait for the hatch to reach the new state.
message_info.State = ICI_MESSAGE_WAITING;
```

- ICI_COMMAND_GET_HATCH_ERROR: This command instructs the ICI control system to retrieve the current hatch error and clear the value. The update function gets the current value of the hatch error, clears the error, and stores the result in the message data buffer. The hatch error is one byte and no byte swapping is necessary. No further processing is necessary, and the message is moved to the finished state.

```
//Get hatch control error (NONE, OPEN, CLOSED, INVALID) and
//clear error.
 retval = get_hatch_error(message.Data+1, 1);
if (retval == OK)
{
  //Set the number of bytes in the message response to the size
  //of the hatch error result (1 byte).
  message.NumData = 1;
}
//Finished processing message.
message_info.State = ICI_MESSAGE_FINISHED;
```

- ICI_COMMAND_GET_ATD_IN_ALL: This command instructs the ICI control system to retrieve the current value of all the ATD input channels. The update function gets the current value of all the ATD input channels and stores the results in the message data buffer. The ATD values are each two bytes and the bytes are swapped to little-endian format. No further processing is necessary, and the message is moved to the finished state.

```
//Get all ATD input channels.
for (i = 0; i < 8 && retval == OK; i++)
{
  retval = get_atd_in(i, (UINT16*)(message.Data+1+i*2));
  if (retval == OK)
  {
    //Set the number of bytes in the message response to the size
    //of the ATD input channel result (2 bytes).
    message.NumData += 2;
    //Convert the result to little endian.
     swap_byte_order(message.Data+1+i*2, 2);
  }
}
//Finished processing message.
message_info.State = ICI_MESSAGE_FINISHED;
```

- ICI_COMMAND_GET_SCI0: This command instructs the ICI control system to retrieve the received serial data on the first serial interface. The update function gets the serial data and stores the results in the message data buffer. No further processing is necessary, and the message is moved to the finished state.

```
//Read data from the SCI0 receive buffer.
retval = receive_serial_all(0, message.Data+1, &message.NumData,
                            ICI_DATA_SIZE-1);
//Finished processing message.
message_info.State = ICI_MESSAGE_FINISHED;
```

- ICI_COMMAND_SET_SCI0: This command instructs the ICI control system to send the specified data on the first serial interface. The update function sends the data bytes. No further processing is necessary, and the message is moved to the finished state.

```
//Write data to the SCI0 transmit buffer.
retval = send_serial(0, message.Data, message.NumData);
//Finished processing message.
message_info.State = ICI_MESSAGE_FINISHED;
```

- ICI_COMMAND_GET_SCI1: This command instructs the ICI control system to retrieve the received serial data on the second serial interface. The update function gets the serial data and stores the results in the message data buffer. No further processing is necessary, and the message is moved to the finished state.

```
//Read data from the SCI1 receive buffer.
retval = receive_serial_all(1, message.Data+1, &message.NumData,
                            ICI_DATA_SIZE-1);
//Finished processing message.
message_info.State = ICI_MESSAGE_FINISHED;
```

- ICI_COMMAND_SET_SCI1: This command instructs the ICI control system to send the specified data on the second serial interface. The update function sends the data bytes. No further processing is necessary, and the message is moved to the finished state.

```
//Write data to the SCI1 transmit buffer.
retval = send_serial(1, message.Data, message.NumData);
//Finished processing message.
message_info.State = ICI_MESSAGE_FINISHED;
```

- ICI_COMMAND_GET_DEVICE_INFO: This command instructs the ICI control system to retrieve the current value of all the system device information. The update function gets the current value of all the device information and stores the result in the message data buffer. The device information contains several multi-byte values, and the bytes are swapped to little-endian format. No further processing is necessary, and the message is moved to the finished state.

```
//Get device information.
retval = get_device_info((ici_device_info*)(message.Data+1));
if (retval == OK)
{
  //Set the number of bytes in the message response to the size
  //of the ICI device information structure (64 bytes).
  message.NumData = sizeof(ici_device_info);
  //Convert the result to little endian.
  retval = swap_device_info((ici_device_info*)(message.Data+1));
}
//Finished processing message.
message_info.State = ICI_MESSAGE_FINISHED;
```

- ICI_COMMAND_GET_RAIN_INDEX: This command instructs the ICI control system to retrieve the current value of the hatch rain sensor index. The update function gets the current value of rain sensor index and stores the result in the message data buffer. The rain sensor index is one byte and no byte swapping is necessary. No further processing is necessary, and the message is moved to the finished state.

```
//Get the rain sensor ATD index.
retval = get_rain_index(message.Data+1);
if (retval == OK)
{
  //Set the number of bytes in the message response to the size
  //of the rain sensor index (1 byte).
  message.NumData = 1;
}
//Finished processing message.
message_info.State = ICI_MESSAGE_FINISHED;
```

- ICI_COMMAND_SET_RAIN_INDEX: This command instructs the ICI control system to set the current value of the hatch rain sensor index. The first byte of data in the received message is the rain sensor index. The update function sets the current value of the rain sensor index. No further processing is necessary, and the message is moved to the finished state.

```
//Set the rain sensor ATD index.
retval = set_rain_index(message.Data[0]);
//Finished processing message.
message_info.State = ICI_MESSAGE_FINISHED;
```

- ICI_COMMAND_GET_RAIN_WET: This command instructs the ICI control system to retrieve the current value of the hatch rain sensor wet threshold. The update function gets the current value of wet threshold and stores the result in the message data buffer. The wet threshold is two bytes and the bytes are swapped to little-endian format. No further processing is necessary, and the message is moved to the finished state.

```
//Get the rain sensor wet threshold.
retval = get_rain_wet((UINT16*)(message.Data+1));
if (retval == OK)
{
  //Set the number of bytes in the message response to the size
  //of the sensor wet threshold (2 bytes).
  message.NumData = 2;
  //Convert the result to little endian.
  swap_byte_order(message.Data+1, message.NumData);
}
//Finished processing message.
message_info.State = ICI_MESSAGE_FINISHED;
```

- ICI_COMMAND_SET_RAIN_WET: This command instructs the ICI control system to set the current value of the hatch rain sensor wet threshold. The first two bytes of data in the received message are the wet threshold value and the bytes are swapped to big-endian format. The update function sets the current value of the wet threshold. No further processing is necessary, and the message is moved to the finished state.

```
//Convert the threshold to big endian.
swap_byte_order(message.Data, message.NumData);
//Set the rain sensor wet threshold.
retval = set_rain_wet(*(UINT16*)message.Data);
//Finished processing message.
message_info.State = ICI_MESSAGE_FINISHED;
```

- ICI_COMMAND_GET_RAIN_DRY: This command instructs the ICI control system to retrieve the current value of the hatch rain sensor dry threshold. The update function gets the current value of dry threshold and stores the result in the message data buffer. The dry threshold is two bytes and the bytes are swapped to little-endian format. No further processing is necessary, and the message is moved to the finished state.

```
//Get the rain sensor dry threshold.
 retval = get_rain_dry((UINT16*)(message.Data+1));
if (retval == OK)
{
  //Set the number of bytes in the message response to the size
  //of the sensor dry threshold (2 bytes).
  message.NumData = 2;
  //Convert the result to little endian.
  swap_byte_order(message.Data+1, message.NumData);
}
//Finished processing message.
message_info.State = ICI_MESSAGE_FINISHED;
```

- ICI_COMMAND_SET_RAIN_DRY: This command instructs the ICI control system to set the current value of the hatch rain sensor dry threshold. The first two bytes of data in the received message are the dry threshold value and the bytes are swapped to big-endian format. The update function sets the current value of the dry threshold. No further processing is necessary, and the message is moved to the finished state.

```
//Convert the threshold to big endian.
swap_byte_order(message.Data, message.NumData);
//Set the rain sensor dry threshold.
retval = set_rain_dry(*(UINT16*)message.Data);
//Finished processing message.
message_info.State = ICI_MESSAGE_FINISHED;
```

In most cases, a new system message can be processed immediately and is moved to the finished state to finalize the response. Some messages require more information and may take many passes through the main program loop to finish processing. Messages that need additional processing are moved to the waiting state. If the message state is waiting, the system assumes it needs to continue processing the message. The actions taken depend on the type of command in the system message. The following are the actions taken for each type of waiting message:

- ICI_COMMAND_PASS_SCI0: This is a continuation of the command that instructed the ICI control system to send the specified data using the first serial interface and then wait for the serial response with the specified terminator. The first byte of data in the received message is the expected value of the last byte in the serial response to this message. The update function checks for a transmit error and then gets the next serial byte. The most recent byte is compared to the serial terminator. If the terminator is found, no further processing is necessary and the message is moved to the finished state.

```
//Check for a transmit error.
retval = send_serial_check(0);
if (retval == OK)
{
  //Get the serial response.
  retval = receive_serial(0, message.Data+1, &message.NumData,
                          ICI_DATA_SIZE-1);
}
//Check if the serial terminator was found.
if (message.NumData > 0 &&
    (message.Data[message.NumData] == message.Data[0] ||
     message.Data[message.NumData] == '?'))
{
  //Finished processing message.
  message_info.State = ICI_MESSAGE_FINISHED;
}
```

- ICI_COMMAND_PASS_SCI1: This is a continuation of the command that instructed the ICI control system to send the specified data using the second serial interface and then wait for the serial response with the specified terminator. The first byte of data in the received message is the expected value of the last byte in the serial response to this message. The update function checks for a transmit error and then gets the next serial byte. The most recent byte is compared to the serial terminator. If the terminator is found, no further processing is necessary and the message is moved to the finished state.

```
//Check for a transmit error.
retval = send_serial_check(1);
if (retval == OK)
{
  //Get the serial response.
  retval = receive_serial(1, message.Data+1, &message.NumData,
                          ICI_DATA_SIZE-1);
}
//Check if the serial terminator was found.
if (message.NumData > 0 &&
    (message.Data[message.NumData] == message.Data[0] ||
    message.Data[message.NumData] == '?'))
{
  //Finished processing message.
  message_info.State = ICI_MESSAGE_FINISHED;
}
```

- ICI_COMMAND_SET_HATCH_STATE: This is a continuation of the command that instructed the ICI control system to set the state of the enclosure's hatch and wait for the hatch to reach the specified state. The first byte of data in the received message is the hatch state value. The update function checks the hatch control system error and gets the current hatch state. The hatch state is compared to the specified state. If the state has been reached, no further processing is necessary and the message is moved to the finished state.

```
//Check for a hatch control system error.
retval = get_hatch_error(message.Data+1, 0);
if (retval == OK)
{
  //Get the current hatch state.
  retval = get_hatch_state(message.Data+1);
}
//Check if the hatch state has reached the desired value.
if (message.Data[0] == message.Data[1])
{
  //Finished processing message.
   message_info.State = ICI_MESSAGE_FINISHED;
}
```

When a message is finished, the ICI control system finalizes the message response and sends the message. To finalize the message, the system fills the acknowledge field with a success or error code. The number of bytes is incremented to account for the extra byte added with the acknowledge field. The system calculates the total size of the message response, which includes the size of the command, the size of the length, and the number of bytes in the response data. The command and length are converted back to little-endian. The ICI control system passes the final message response to the TCP control system and then clears the system message.

```
//Fill in the ACK/NAK byte in message response.
message.Data[0] = retval;
//Increment the number of bytes to include the size of the
//ACK/NAK byte.
message.NumData++;
//Calculate total size of the message response (header and data).
NumData = message.NumData + sizeof(ici_message_header);
//Convert the command and numdata to little endian.
swap_byte_order((UINT8*)&message.Command,
                sizeof(message.Command));
swap_byte_order((UINT8*)&message.NumData,
                sizeof(message.NumData));
//Send the message response.
tcp_server_send(message_info.Index, (UINT8*)&message, NumData);
//Clear the message.
ici_clearmessage();
```

Main Program

The firmware for this project was structured to match the project specifications and divided into separate files for each component of the system. The system components are consolidated and controlled by the main program for the embedded system. The main program includes the ICI system initialization and the main program loop.

Main Program Design

The ICI embedded system Ethernet implementation uses OpenTCP, an open source implementation of the Ethernet TCP/IP stack. The main program for the embedded system is based on the original OpenTCP project. The main program for the OpenTCP project included example code for initializing the network interface and using the main program loop to receive and process Ethernet frames.

In the ProjectICI CodeWarrior project, the main program implementation is located in the file `main.c`. This file handles the initialization and control of the main program. The main program is broken into two parts, the initialization portion of the program and the applications portion of the program. The following is the function used to implement the main program:

- void main(void): The `main` function initializes the ICI control system. It watches for Ethernet frames, processes the frames, and then updates the TCP connection. After processing the incoming data, the function updates the TCP control system and the ICI control system.

Main Program Implementation

Several global variables are used to manage the OpenTCP implementation. The first is the global structure `localmachine`. The `localmachine` variable holds the information that defines the embedded network interface. The `localip` variable is the IP address of the embedded device, and the `localHW` variable is the MAC address. The `defgw` variable is the default gateway used to communicate with the Internet. The `netmask` variable is the network submask and is used to determine if the device is sending data on the local network or through the gateway. The main program only needs to initialize the network information. The rest is handled by the OpenTCP.

```
struct netif
{
  LWORD localip;    //IP address.
  BYTE  localHW[6]; //MAC address.
  LWORD defgw;      //Default gateway.
  LWORD netmask;    //Network submask.
};
struct netif localmachine;
```

The second global variable is `gotxflowc`. This variable is defined and managed by the NE64 driver and is used to determine if flow control packets are sent in full duplex. The third global variable is `gotlink`. This variable is used to determine if the

Ethernet link is active. The forth global variable is `LEDCounter`. This variable is used to drive the Ethernet status indicators. These variables are part of the OpenTCP implementation.

```
extern tU16 gotxflowc; // flow control packets in full duplex.
extern tU08 gotlink; //Determines if link is active.
tU16 LEDcounter=0; // EPHY status indicators count.
```

The main program initialization begins by configuring the system clock. The system clock initialization is described in an earlier section on the system clock. The clock is configured to 25[MHz]. The next step in the initialization disables the interrupt request (IRQ) signal. This is an external interrupt that interrupts the main program loop, and the IRQ is not used for this project.

```
INTCR_IRQEN = 0;     //Disable the IRQ (enabled after reset).
```

After configuring the system clock and disabling the IRQ, the main program initializes the system network information. This sets the IP address, the MAC address, the default gateway, and the network mask. All of the network information is defined in the file `address.c`. The IP address is set to 192.168.250.3, the MAC address is set to 01-23-45-56-78-9A, the default gateway is set to 192.168.250.1, and the network mask is set to 255.255.255.0.

```
localmachine.localip = *((UINT32 *)ip_address); //IP address.
localmachine.defgw = *((UINT32 *)ip_gateway); //Default gateway.
localmachine.netmask = *((UINT32 *)ip_netmask); //Subnet mask.
localmachine.localHW[0] = hard_addr[0]; //MAC address.
localmachine.localHW[1] = hard_addr[1];
localmachine.localHW[2] = hard_addr[2];
localmachine.localHW[3] = hard_addr[3];
localmachine.localHW[4] = hard_addr[4];
localmachine.localHW[5] = hard_addr[5];
```

The last part of the main program initialization walks through the initialization for each component of the system. The main program calls the initialization routines for the

OpenTCP timers, buffers, Ethernet, and ARP and TCP protocols. The main program also initializes the ATD, digital inputs and outputs, serial communications, TCP control system, the ICI control system, and the RTI.

```
timer_pool_init(); //OpenTCP timers.
atd_init();        //Atd converter.
digital_io_init(); //Digital I/O.
serial_io_init();  //Serial I/O.
mBufInit ();       //OpenTCP buffers.
EtherInit();       //OpenTCP Ethernet.
asm CLI;           //Enable interrupts.
arp_init();        //OpenTCP ARP.
(void)tcp_init();  //OpenTCP TCP.
void)tcp_server_init(); //TCP server.
ici_init();        //ICI control system.
RTI_Init();        //RTI.
RTI_Enable ();     //Enabled RTI.
```

The main program loop begins immediately after the system initialization. The main loop is infinite and is intended to loop forever through the system tasks. It watches for Ethernet frames, processes the frames, and then updates the TCP connection. After processing the incoming data, the function updates the TCP control system and the ICI control system.

```
for (;;) //Loop forever.
{
  …; //Update system.
}
```

The main loop checks for incoming Ethernet frames. When a frame is received, it determines which protocol, ARP or IP, needs to process the message. In the case of IP, the frame is processed further to determine which IP protocol, ICMP, UDP, or TCP, needs to process the message. In any case, the correct function is called to process the packet.

```
switch( received_frame.protocol)
{
  case PROTOCOL_ARP: //ARP frame.
    process_arp (&received_frame);
    break;
  case PROTOCOL_IP:      //IP frame.
    len = process_ip_in(&received_frame);
    if(len < 0)
      break;
    switch (received_ip_packet.protocol)
    {
      case IP_ICMP:  //ICMP/IP frame.
        process_icmp_in (&received_ip_packet, len);
        break;
      case IP_UDP:   //UDP/IP frame.
        process_udp_in (&received_ip_packet,len);
        break;
      case IP_TCP:   //TCP/IP frame.
        process_tcp_in (&received_ip_packet, len);
        break;
      default:
        break;
    }
    break;
  default:
    break;
}
```

After processing the incoming Ethernet frames, the main program loop updates the OpenTCP ARP and TCP components. Then the program updates the TCP control system and the ICI control system.

```
arp_manage();     //Update OpenTCP ARP.
tcp_poll();       //Update OpenTCP TCP.
tcp_server_run(); //Update TCP server.
ici_run();        //Update ICI control system.
```

PROJECT TESTING

After completing the ICI embedded system implementation, a test system was created for testing the microcontroller firmware. This test system was necessary to verify the functionality of the analog, digital, serial, and Ethernet components before integrating the microcontroller into the actual ICI system. The test system includes the hardware and software necessary to simulate a complete ICI system.

Test Hardware

The test system hardware includes the microcontroller development board and the system connections. The microcontroller development board is equipped with all the hardware necessary to test the analog, digital, serial, and Ethernet components of the ICI system. To complete the test system, cables and wires were necessary for connecting the hardware to the test software.

Development Board

The EVB9S12NE64 evaluation board for the MC9S12NE64 microcontroller is equipped with two port connectors that provide access to the analog channels and the digital input and output channels, two serial communications ports with RS232 DB9-S connectors, and a 10/100T Ethernet port. The board has an adjustable potentiometer that provides a linear voltage output to bit 0 on port AD and can be used to sweep the ATD from 0[V] to 3.3[V]. The board is also equipped with a power port terminal block. The power port provides access to 3.3[V]. This voltage is equivalent to logic high for the digital channels.

System Connections

To test the digital input channels, wires were used to connect the inputs to the appropriate voltage level on the power port. A standard 9-pin serial cable was used to connect the test software to the serial port on the development board, and a crossover Ethernet cable was used to connect the test software to the Ethernet port. The serial cable used for testing was male-to-female, but this cable was only for testing and does not work in the actual ICI system. The development board and the ICI serial peripherals both have female connectors and require a male-to-male cable for communication. Further, a null modem adapter is required in the ICI system to make the serial cable a crossover cable.

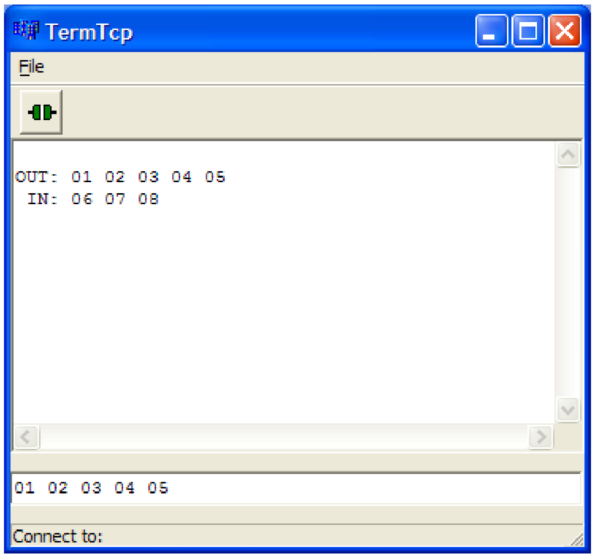
Test Software

The test system software was developed in Borland C++ Builder 6 and includes applications for testing the Ethernet and serial communications. The ICI-3 software interface uses an Ethernet link to establish a connection with the embedded system inside the system enclosure. To simulate the ICI-3 software and test the system communications, a test software application was created for sending and receiving data through an Ethernet connection. The ICI-3 optical system includes devices that communicate serially with the embedded system. The microcontroller passes data from the serial devices to the software interface and from the software interface back to the serial devices. A second test software application was created for sending and receiving data through a serial connection.

TermTCP

The project TermTCP is a TCP/IP client application developed for testing the Ethernet communications with the ICI-3 embedded system. This application is based on a sample program provided with the C++ Builder development environment. The sample application is a client/server program. Since only the client side was needed for testing, the server implementation was removed. The TCP/IP port was set to 1024 to match the port used by the embedded system.

The software interface (Figure 9) includes a connect option that is used to enter the IP address of the target device. For this project, the IP address of the embedded system is 192.168.250.3. After connecting, the interface is designed to accept hex characters as input. The user enters the data in the edit box at the bottom of the screen and presses enter. The software converts the input from characters to data and transmits the data. When it receives data, it converts the data to hex characters and displays the result on the screen. All communications are displayed in the memo in the center of the screen.



```
OUT: 01 02 03 04 05
 IN: 06 07 08
```

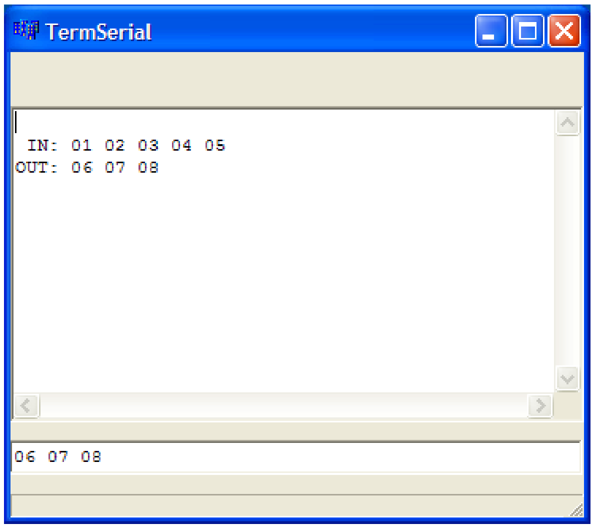
```
01 02 03 04 05
```

Figure 9: TermTcp

<u>TermSerial</u>

The project TermSerial is a serial terminal program developed for testing serial communications with the ICI-3 embedded system. This application is based on the TermTCP project. The TCP/IP support was removed from the application, and support was added for the serial communications.

The software interface (Figure 10) automatically opens the port for the embedded system. For this project, the software assumes the microcontroller is connected to the computer on com port 1. After connecting, the interface is designed to accept hex characters as input. The user enters the data in the edit box at the bottom of the screen and presses enter. The software converts the input from characters to data and transmits the data. When it receives data, it converts the data to hex characters and displays the result on the screen. All communications are displayed in the memo in the center of the screen.



```
 IN: 01 02 03 04 05
OUT: 06 07 08
```

```
06 07 08
```

Figure 10: TermSerial

Test Results

The test hardware in combination with the test software was use to create a test system capable of simulating the entire ICI-3 system. While testing the embedded system, the Ethernet cable was used to connect the computer to the development board, and the serial cable was used to connect the development board to the computer. The application TermTCP was launched and used to establish an Ethernet link with the board, and TermSerial was launched and connected to one of the board’s serial ports. Wires were used to connect the digital input channels to 3.3[V] on the power port.

ATD and Digital I/O Testing

The board has an adjustable potentiometer that provides a linear voltage output to bit 0 on port AD and can be used to sweep the ATD from 0[V] to 3.3[V]. The potentiometer was used to adjust the voltage, and then TermTCP was used to ask the embedded system for the associated analog channel. The analog channels are given in ticks. With the potentiometer turned to 0[V], the microcontroller returned 0x000 for the current analog channel value. With the potentiometer at 3.3[V], the microcontroller returned 0x3FF, which corresponds to the full 10-bit ATD resolution. This test verifies that the microcontroller returns the correct value for the ATD channel for the entire 10-bit range.

To test the digital inputs, the 3.3[V] power port on the development board was linked to the first digital input channel. TermTCP was used to read the value of each of the eight digital input channels, and then the high voltage was moved to the next digital input channel. The test was repeated for each channel. In each case, the microcontroller returned 0x01 for the channel connected to the voltage, and 0x00 for all other channels. This test verifies the digital input channels are implemented correctly.

To test the digital outputs, a multi-meter was used to measure the voltage for each digital output channel. TermTCP was used to set the value of an output channel to 0x01, which corresponds to digital logic high. The voltage on each channel was measured with the multi-meter, and the software was used to read the value of each channel. Then the software was used to clear the output channel. This process was repeated for each channel. In each case, only one output voltage measured high and all the measured values matched the values read from the microcontroller. This test verifies the digital output channels are implemented correctly.

<u>Serial I/O Testing</u>

The microcontroller is intended to act as a “pass through” device. Data will be sent to the microcontroller over Ethernet, and the microcontroller will take that data and send it out serially. The microcontroller will also receive data serially and then send that data over Ethernet. TermTCP was used in conjunction with TermSerial to simulate a complete “pass through” system.

The embedded system pass through command was used to test the serial pass through. The pass through command for the first microcontroller serial port is 0x0005. The following data is sent to pass ’01 02 03 04 05’ to the first serial port and wait for a response with the terminator ‘08’:

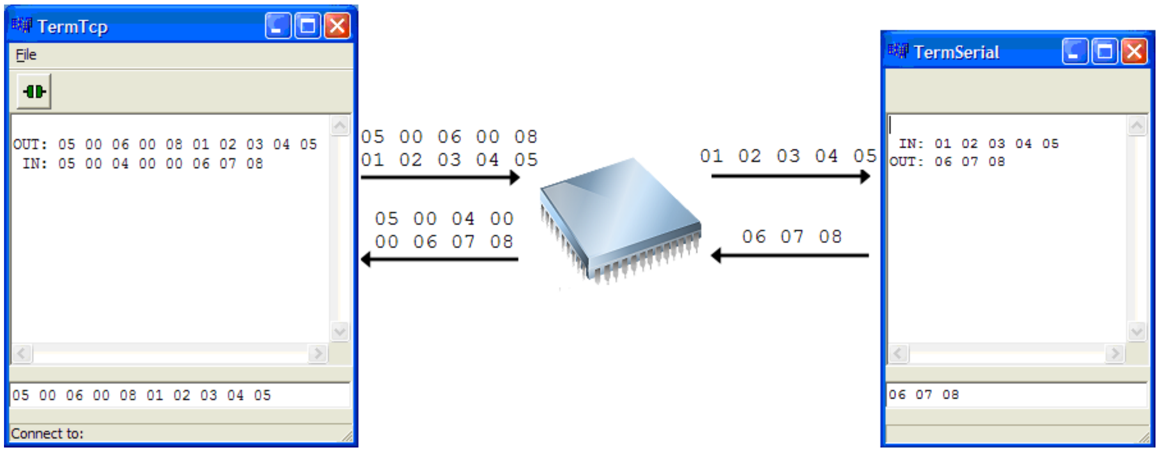
```
OUT: 05 00 06 00 08 01 02 03 04 05
     05 00                        –Command
           06 00                  –NumData
                 08               –Terminator
                    01 02 03 04 05 –Tx data
```

To test the serial communications, TermTCP was used to send the data in the previous example to the microcontroller. The microcontroller parsed the data and sent the bytes ’01 02 03 04 05’ to TermSerial. Once that data was received, TermSerial was used to generate the serial response ’06 07 08’. The microcontroller recognized the ‘08’ as the terminator and sent the serial data back to TermTCP. The following is the data received:

```
IN: 01 00 03 00 00 FF 03
    05 00                 -Command
          04 00           -NumData
                00        -ACK
                   06 07 08 -Value of atd_in_0
```

The pass through process is illustrated in Figure 10. The data transmitted by TermTCP matches the data received by TermSerial, and the data transmitted by

TermSerial matches the data received by TermTCP. This test verifies the serial communications and pass through logic are implemented correctly.



```
OUT: 05 00 06 00 08 01 02 03 04 05
 IN: 05 00 04 00 00 06 07 08
```

```
05 00 06 00 08 01 02 03 04 05
```

```
 IN: 01 02 03 04 05
OUT: 06 07 08
```

```
06 07 08
```

Figure 11: Test System for Serial I/O

Hatch Control System Testing

The ICI-3 system design includes a motorized hatch that opens and closes under direction of the microcontroller. The hatch control system is implemented using digital input and output channels. The hatch control is simply an extension of the general-purpose digital I/O, and the same concepts were used to test its functionality. The hatch control system also requires one analog input for the hatch rain sensor. By default, the rain sensor is configured to use the first analog channel, and the first channel is connected to the potentiometer on the development board. The same test process for the ATD channels was used to test the rain sensor.

The first hatch tests were designed to test the open and close commands. The hatch control system has two digital inputs that are connected to the hatch position sensors. When the digital input is logic high, the hatch is in contact with that position sensor. The 3.3[V] power port on the development board was switched between the position sensors to simulate hatch open and hatch closed. The hatch control system also uses a digital output that controls the hatch motor. A multi-meter was used to measure the state of the hatch digital output.

TermTCP was used to send the hatch open command. A multi-meter was used to verify the digital output line switched to high to start moving the hatch. The high voltage was connected to the open sensor to simulate the hatch reaching the open position, and the digital output was measured again to verify the line switched low to stop moving the hatch. This test was repeated with the hatch close command. These tests verify the hatch control system is capable of opening and closing the system hatch.

Additional tests were designed to test the hatch failsafe routines. The first failsafe is the open timeout. The hatch automatically closes after 60[s]. TermTCP was used to send the open command, but not the close command. The high voltage was connected to the open sensor position to simulate the hatch in the open position. After 60[s], the digital output line went high and the hatch automatically closed. This test verifies that the hatch open failsafe is working correctly.

The second failsafe accounts for a failed open sensor. TermTCP was used to send the open command, and the high voltage was connected to the closed sensor to simulate the hatch in the closed position. Next, the high voltage was disconnected from the closed sensor, and then reconnected to the closed sensor. This simulates the hatch moving away from the closed sensor, failing to activate the open sensor, and then reaching the closed sensor again. After a full revolution, the hatch control system switched the digital output line to low to stop moving the hatch. This test verifies that the open sensor failsafe is working correctly.

The third failsafe accounts for a failed closed sensor. TermTCP was used to send the open command, and the high voltage was connected to the open sensor to simulate the hatch in the open position. After opening, TermTCP was used to send the close command. In this case, the high voltage was disconnected from the open sensor, and then reconnected to the open sensor. This simulates the hatch moving away from the open sensor, failing to activate the closed sensor, and then reaching the open sensor again. After a full revolution, the hatch control system continued moving the hatch for half the revolution time and then switched the digital output line to low to stop moving the hatch. This test verifies that the closed sensor failsafe is working correctly.

The forth failsafe accounts for both sensors failing. TermTCP was used to send the open command, and the high voltage line was not connected to either position sensor. In this case, the hatch continued to open and close for 60[s], and then the hatch control system switched the digital output line to low to stop moving the hatch. This test verifies that the failed sensors failsafe is working correctly.

The last failsafe uses the hatch rain sensor to detect unfavorable weather conditions. TermTCP was used to send the open command, and the potentiometer on the development board was turned below the sensor wet threshold to signal rain. That hatch control system automatically closed the hatch. With the hatch closed, TermTCP was used to resend the open command. In this case, the hatch control system refused to open the hatch because of rain. The potentiometer was turned above the dry threshold and the open command was sent again. When dry conditions were detected, the hatch opened successfully. This verifies the rain failsafe is working correctly.

CONCLUSIONS

This project has focused on the development and testing of an embedded system for the third-generation ICI design. To accomplish this task, project specifications for the ICI design were used to generate a list of requirements for the embedded system, these requirements were used to select the microcontroller, and the system firmware was implemented and tested. Testing has shown that the microcontroller is functioning as expected and meets all the requirements of the original specification. The embedded system is capable of supporting digital sensors and controls, analog sensors, and peripherals that communicate serially. In addition, an Ethernet connection is used to communicate with the embedded system inside the ICI optics enclosure.

Areas for Improvement

Throughout project development, the specifications for the ICI-3 embedded system were modified to account for corrections, new concepts, and new feature requests. Most of the changes were integrated into the system as development progressed, but some issues were not addressed because of time constraints. The incomplete items do not prevent the system from working, but finishing the implementation would improve the overall functionality of the embedded system.

Access to Flash Memory

The variables used in the firmware implementation for the embedded system are currently stored in random-access memory (RAM). RAM is volatile memory and is only preserved while power is on. On reset, the variables lose their values. Early in this project, volatile memory was not an issue. There were no values that needed to keep their values when the system resets. However, a rain sensor was added to the hatch control system, and the sensor has several configurable parameters stored in volatile memory. These values must be restored after every system reset.

Instead, the rain sensor configuration should be stored in nonvolatile memory. The MC9S12NE64 is equipped with 64[KB] of nonvolatile Flash memory. As nonvolatile memory, Flash memory is ideal for program and data storage because it can retain the stored information even when the system is not powered. Flash support should be added to the embedded system to account for the nonvolatile memory requirements.

Additions to the ICI Command Set

Several additions were requested for the ICI control system command set. The first is a command for configuring the baud rate for each SCI module. The serial interface is designed to act as a generic “pass through” device, and the embedded system has no knowledge of the devices connected to it. In order to communicate, the baud rate of the SCI module must match the baud rate of the serial device. Currently, the baud rate for each SCI module is set to 9600 and is not configurable. Devices connected to the embedded system must support the same rate. Instead of forcing devices to match the system baud rate, the embedded system should include support for configuring its baud rate to match the device.

The second addition to the ICI command set is a command for reading and settings all digital channels. The embedded system supports eight digital inputs and eight digital outputs. In the current implementation, the system includes separate commands for reading the state of each digital input and output, and setting the state of each digital output. In addition to supporting individual channel manipulation, the embedded system should include a command for reading all digital inputs, a command for reading all digital outputs, and a command for setting all digital outputs.

The third addition to the ICI command set is a command for resetting the embedded system. The microcontroller development board is equipped with a reset button, and this button was used during development to reset the system. In the released system, the microcontroller is embedded in the ICI optics enclosure with the thermal infrared camera and other ICI hardware, and the reset button will not be easily accessible. To allow for a simple and remote reset, the embedded system should include a command for resetting the microcontroller.

Future Work

The work summarized in this paper completes the work necessary for a functional embedded system for the third-generation ICI system. ICI-3 is the first of the ICI systems to include an embedded system, and the implementation only includes the minimum features necessary to support the project specifications. As development continues, more could be done to fully utilize the power of having an intelligent system. The following section presents several concepts that might be explored as a continuation of this project.

Field-reprogrammable

A field-reprogrammable device is a device that is capable of rewriting its internal program memory without any external hardware. This allows the end-user to update the device firmware without replacing any components or returning the device to the manufacturer. The user can get a software interface for updating the firmware and the latest version of the firmware from the manufacturer, and then use the software to update the device.

The ICI embedded system is not field-reprogrammable. A BDM interface was used to program and debug the firmware for the ICI embedded system. The BDM interface connects to the microcontroller via a custom 6-pin connector and cable. This method of reprogramming requires access to the microcontroller inside the ICI optics enclosure. Instead of requiring the BDM interface for updates, a better solution would be to reprogram the embedded system through the Ethernet interface.

Several application notes on reprogramming devices in the HCS12 family are available from Freescale Semiconductor. These implementations reprogram a device through a controller area network (CAN) or serial interface. However, little information is available for reprogramming the MC9S12NE64 through an Ethernet interface. More research needs to be conducted on the feasibility of reprogramming an Ethernet device.

Web Server

An Ethernet connection is used to communicate with the embedded system inside the ICI optics enclosure. With support for Ethernet and TCP/IP data transmission, the embedded device, with some modifications, could be connected to the Internet. This would allow access to the device from anywhere with an Internet connection.

The ICI embedded system Ethernet implementation uses OpenTCP, an open source implementation of the Ethernet TCP/IP stack. In its original form, OpenTCP provides support for a web server. This includes an HTTP server and C representations of web related HTML files. This implementation allows a web browser to navigate to the microcontroller and request a web page.

The web server files were removed from the ICI firmware because they were not used for this project, but this support could easily be added back. The default OpenTCP HTTP server is enough to get an ICI web server started, but more work would be required to make web access useful. Code would need to be implemented to support live updates on a web page.

Custom Board for the Microcontroller

The EVB9S12NE64 is the development board for the MC9S12NE64 microcontroller. The board is approximately 5.0 x 7.0 inches, which makes it small enough to include in the ICI optics enclosure. This eliminated the need to design a custom board for the MC9S12NE64. The development board was a great solution for this project because it allowed for quick and efficient development and was used to test the system implementation.

In the future, as more ICI-3 systems are produced, additional embedded systems will be required. With the current design, a development board with the MC9S12NE64 would need to be purchased for each system. While this is a feasible solution, it may not be the optimal solution. Designing a custom board may reduce the cost and size of the system. More research needs to be done to determine the benefits of a custom board design.

REFERENCES CITED

1. Nugent, P. W., “Wide-Angle Infrared Cloud Imaging for Cloud Cover Statistics,” Masters Thesis, Montana State University Electrical and Computer Engineering (2008).

2. Shaw J. A. Nugent P. W., Pust N. J., Thurairajah B., Mizutani K., “Radiometric Cloud Imaging with an Uncooled Microbolometer Thermal Infrared Camera,” Optics Express 13 (15), 5807-5817 (2005).

3. Axelson J., “Embedded Ethernet and Internet Complete: Designing and Programming Small Devices for Networking,” Lakeview Research LLC, Madison, WI (2003).

4. “Selecting the Right Microcontroller Unit,” Freescale Semiconductor (2004). Available Online: http://www.freescale.com/files/microcontrollers/doc/app_note/AN1057.pdf

5. Van Sickle T., “Programming Microcontrollers in C,” Newnes Publishing, USA (2003).

6. “MC9S12NE64 Data Sheet,” Freescale Semiconductor, rev 1.1 (2006). Available Online: http://www.freescale.com/files/microcontrollers/doc/data_sheet/MC9S12NE64V1.pdf

7. Williams J., “Serial Monitor Program for HCS12 MCUs,” Freescale Semiconductor (2003). Available Online: http://www.freescale.com/files/microcontrollers/doc/app_note/AN2548.pdf

8. Airaudi T. J., “Using Background Debug Mode for the M68HC12 Family,” Freescale Semiconductor (2004). Available Online: http://www.freescale.com/files/microcontrollers/doc/app_note/AN2104.pdf

9. Nugent P. W., personal communication: “Motor Control Capabilities Required of the ICI Microcontroller,” Optical Remote Sensor Laboratory, Montana State University (2008).

10. Cady, F. M., “Microcontrollers and Microcomputers: Principles of Software and Hardware Engineering,” Oxford University Press, New York, NY (1991).

11. Torres S., “Web Server Development with MC9S12NE64 and OpenTCP,” Freescale Semiconductor, rev 0 (2004). Available Online: http://www.freescale.com/files/microcontrollers/doc/app_note/AN2836.pdf

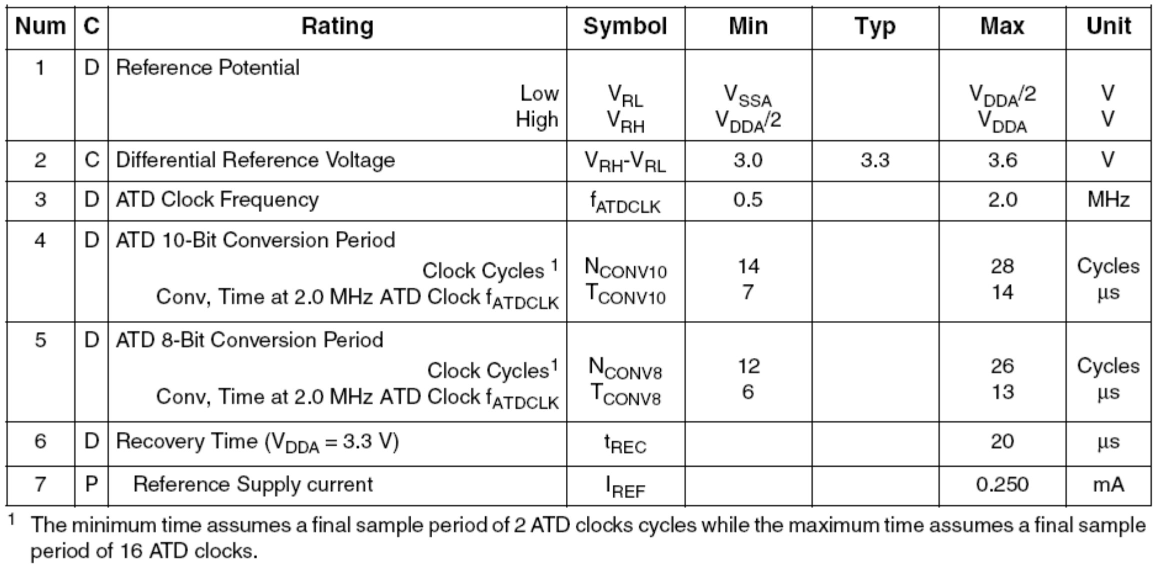
12. “EVB9S12NE64 Schematic,” Freescale Semiconductor, rev B (2004). Available Online: http://www.freescale.com/files/soft_dev_tools/hardware_tools/schematics/EVB9S12NE64SCH.pdf?fsrch=1

APPENDICES

APPENDIX A

MC9S12NE64 ELECTRICAL CHARACTERISTICS

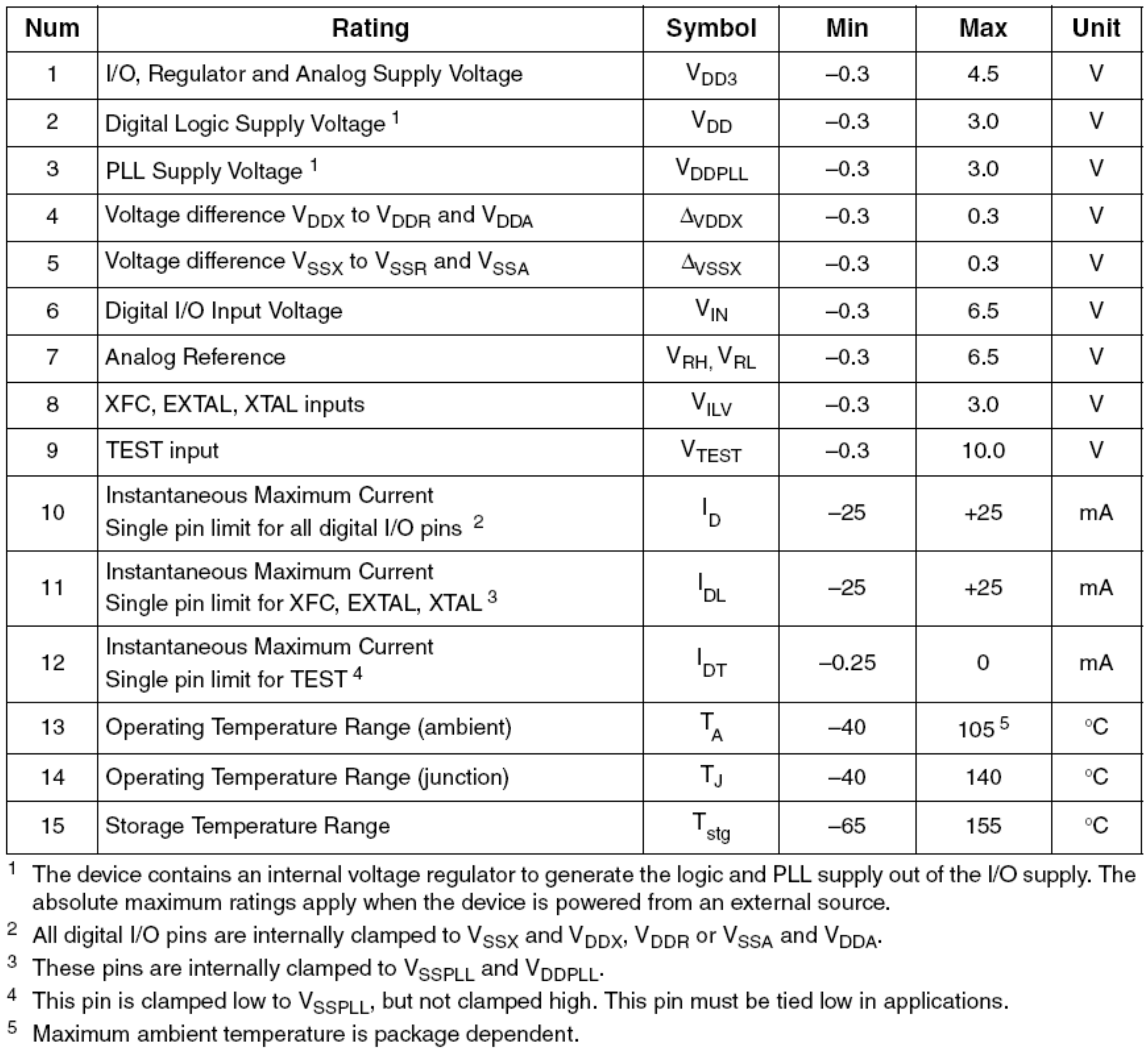
Table 6: 3.3[V] ATD Operating Characteristics [6]



Num	C	Rating	Symbol	Min	Typ	Max	Unit
1	D	Reference Potential Low High	 V_{RL} V_{RH}	 V_{SSA} $V_{DDA}/2$		 $V_{DDA}/2$ V_{DDA}	 V V
2	C	Differential Reference Voltage	V_{RH}-V_{RL}	3.0	3.3	3.6	V
3	D	ATD Clock Frequency	f_{ATDCLK}	0.5		2.0	MHz
4	D	ATD 10-Bit Conversion Period Clock Cycles [1] Conv, Time at 2.0 MHz ATD Clock f_{ATDCLK}	 N_{CONV10} T_{CONV10}	 14 7		 28 14	 Cycles µs
5	D	ATD 8-Bit Conversion Period Clock Cycles[1] Conv, Time at 2.0 MHz ATD Clock f_{ATDCLK}	 N_{CONV8} T_{CONV8}	 12 6		 26 13	 Cycles µs
6	D	Recovery Time (V_{DDA} = 3.3 V)	t_{REC}			20	µs
7	P	Reference Supply current	I_{REF}			0.250	mA

[1] The minimum time assumes a final sample period of 2 ATD clocks cycles while the maximum time assumes a final sample period of 16 ATD clocks.

Table 7: Absolute Maximum Ratings [6]



Num	Rating	Symbol	Min	Max	Unit
1	I/O, Regulator and Analog Supply Voltage	V_{DD3}	–0.3	4.5	V
2	Digital Logic Supply Voltage [1]	V_{DD}	–0.3	3.0	V
3	PLL Supply Voltage [1]	V_{DDPLL}	–0.3	3.0	V
4	Voltage difference V_{DDX} to V_{DDR} and V_{DDA}	Δ_{VDDX}	–0.3	0.3	V
5	Voltage difference V_{SSX} to V_{SSR} and V_{SSA}	Δ_{VSSX}	–0.3	0.3	V
6	Digital I/O Input Voltage	V_{IN}	–0.3	6.5	V
7	Analog Reference	$V_{RH,}$ V_{RL}	–0.3	6.5	V
8	XFC, EXTAL, XTAL inputs	V_{ILV}	–0.3	3.0	V
9	TEST input	V_{TEST}	–0.3	10.0	V
10	Instantaneous Maximum Current Single pin limit for all digital I/O pins [2]	I_D	–25	+25	mA
11	Instantaneous Maximum Current Single pin limit for XFC, EXTAL, XTAL [3]	I_{DL}	–25	+25	mA
12	Instantaneous Maximum Current Single pin limit for TEST [4]	I_{DT}	–0.25	0	mA
13	Operating Temperature Range (ambient)	T_A	–40	105 [5]	°C
14	Operating Temperature Range (junction)	T_J	–40	140	°C
15	Storage Temperature Range	T_{stg}	–65	155	°C

[1] The device contains an internal voltage regulator to generate the logic and PLL supply out of the I/O supply. The absolute maximum ratings apply when the device is powered from an external source.

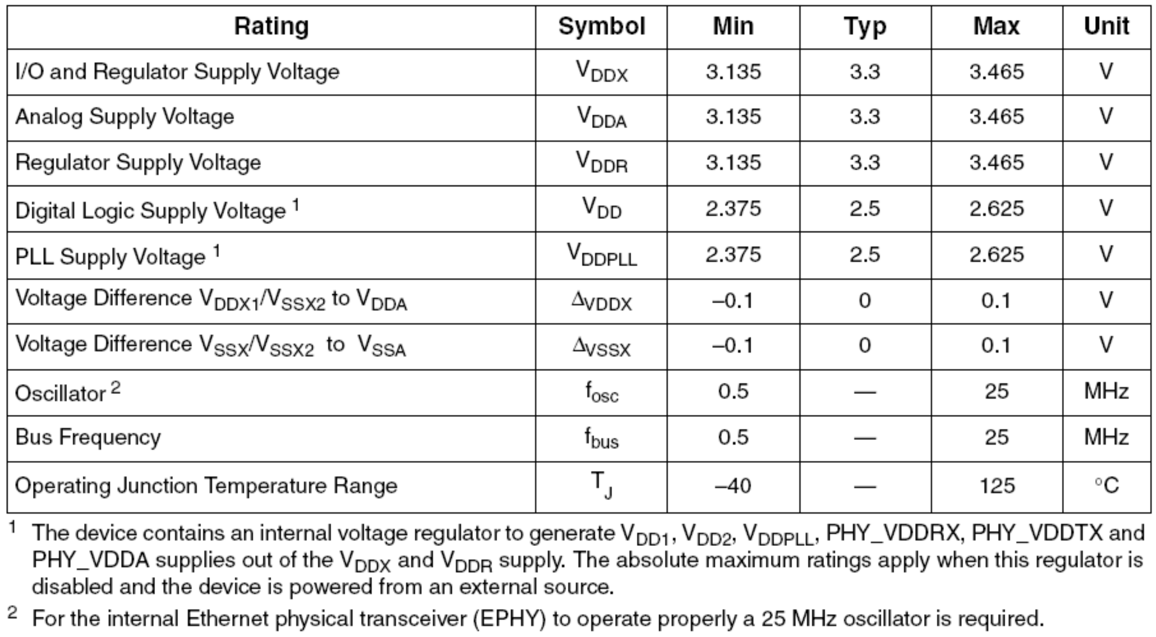
[2] All digital I/O pins are internally clamped to V_{SSX} and V_{DDX}, V_{DDR} or V_{SSA} and V_{DDA}.

[3] These pins are internally clamped to V_{SSPLL} and V_{DDPLL}.

[4] This pin is clamped low to V_{SSPLL}, but not clamped high. This pin must be tied low in applications.

[5] Maximum ambient temperature is package dependent.

Table 8: Operating Conditions [6]

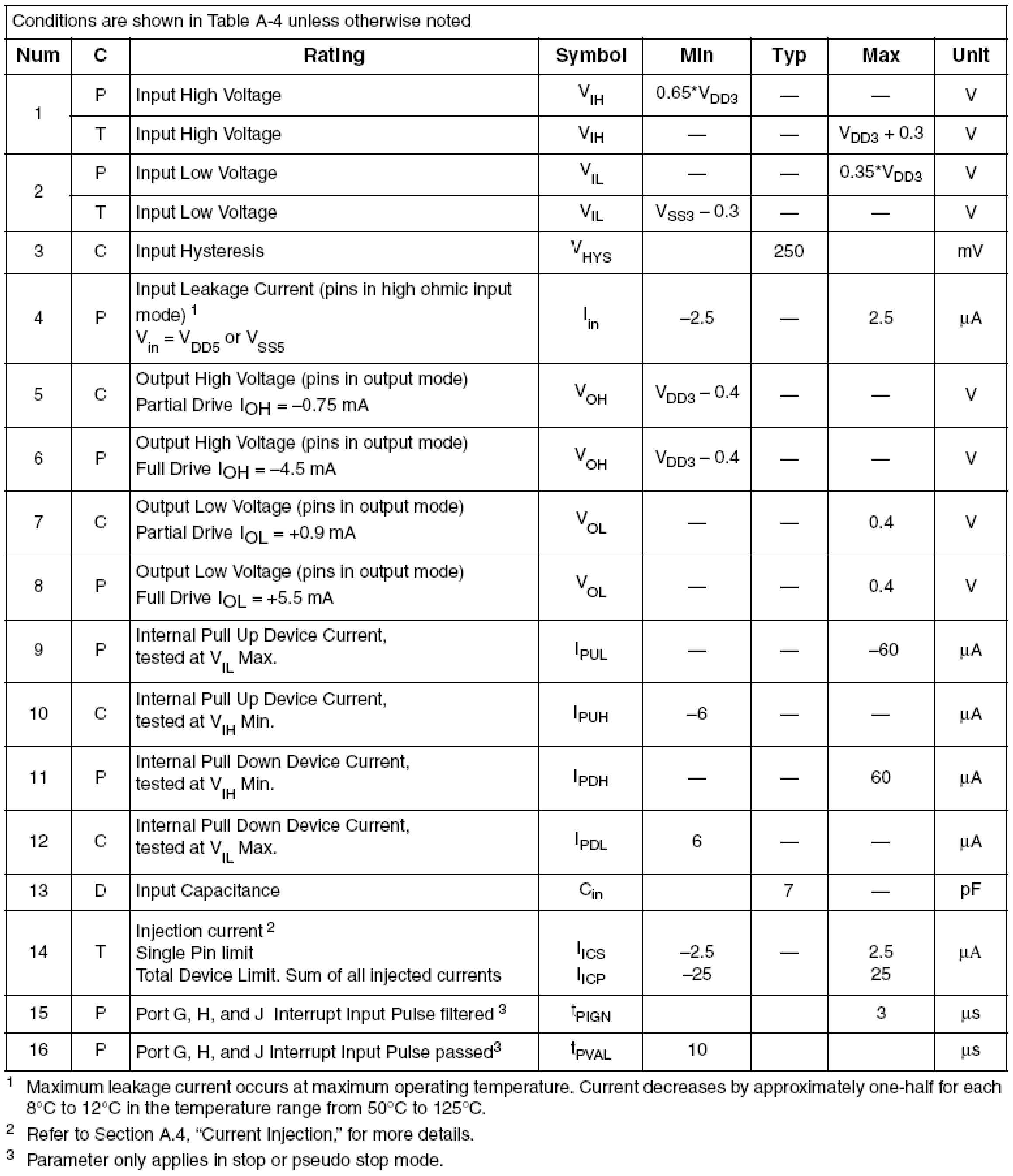


Rating	Symbol	Min	Typ	Max	Unit
I/O and Regulator Supply Voltage	V_{DDX}	3.135	3.3	3.465	V
Analog Supply Voltage	V_{DDA}	3.135	3.3	3.465	V
Regulator Supply Voltage	V_{DDR}	3.135	3.3	3.465	V
Digital Logic Supply Voltage [1]	V_{DD}	2.375	2.5	2.625	V
PLL Supply Voltage [1]	V_{DDPLL}	2.375	2.5	2.625	V
Voltage Difference V_{DDX1}/V_{SSX2} to V_{DDA}	Δ_{VDDX}	–0.1	0	0.1	V
Voltage Difference V_{SSX}/V_{SSX2} to V_{SSA}	Δ_{VSSX}	–0.1	0	0.1	V
Oscillator [2]	f_{osc}	0.5	—	25	MHz
Bus Frequency	f_{bus}	0.5	—	25	MHz
Operating Junction Temperature Range	T_J	–40	—	125	°C

[1] The device contains an internal voltage regulator to generate V_{DD1}, V_{DD2}, V_{DDPLL}, PHY_VDDRX, PHY_VDDTX and PHY_VDDA supplies out of the V_{DDX} and V_{DDR} supply. The absolute maximum ratings apply when this regulator is disabled and the device is powered from an external source.

[2] For the internal Ethernet physical transceiver (EPHY) to operate properly a 25 MHz oscillator is required.

Table 9: Preliminary 3.3[V] I/O Characteristics [6]



Conditions are shown in Table A-4 unless otherwise noted

Num	C	Rating	Symbol	Min	Typ	Max	Unit
1	P	Input High Voltage	V_{IH}	$0.65*V_{DD3}$	—	—	V
	T	Input High Voltage	V_{IH}	—	—	$V_{DD3} + 0.3$	V
2	P	Input Low Voltage	V_{IL}	—	—	$0.35*V_{DD3}$	V
	T	Input Low Voltage	V_{IL}	$V_{SS3} – 0.3$	—	—	V
3	C	Input Hysteresis	V_{HYS}		250		mV
4	P	Input Leakage Current (pins in high ohmic input mode) [1] $V_{in} = V_{DD5}$ or V_{SS5}	I_{in}	–2.5	—	2.5	μA
5	C	Output High Voltage (pins in output mode) Partial Drive $I_{OH} = –0.75$ mA	V_{OH}	$V_{DD3} – 0.4$	—	—	V
6	P	Output High Voltage (pins in output mode) Full Drive $I_{OH} = –4.5$ mA	V_{OH}	$V_{DD3} – 0.4$	—	—	V
7	C	Output Low Voltage (pins in output mode) Partial Drive $I_{OL} = +0.9$ mA	V_{OL}	—	—	0.4	V
8	P	Output Low Voltage (pins in output mode) Full Drive $I_{OL} = +5.5$ mA	V_{OL}	—	—	0.4	V
9	P	Internal Pull Up Device Current, tested at V_{IL} Max.	I_{PUL}	—	—	–60	μA
10	C	Internal Pull Up Device Current, tested at V_{IH} Min.	I_{PUH}	–6	—	—	μA
11	P	Internal Pull Down Device Current, tested at V_{IH} Min.	I_{PDH}	—	—	60	μA
12	C	Internal Pull Down Device Current, tested at V_{IL} Max.	I_{PDL}	6	—	—	μA
13	D	Input Capacitance	C_{in}		7	—	pF
14	T	Injection current [2] Single Pin limit Total Device Limit. Sum of all injected currents	I_{ICS} I_{ICP}	–2.5 –25	—	2.5 25	μA
15	P	Port G, H, and J Interrupt Input Pulse filtered [3]	t_{PIGN}			3	μs
16	P	Port G, H, and J Interrupt Input Pulse passed[3]	t_{PVAL}	10			μs

[1] Maximum leakage current occurs at maximum operating temperature. Current decreases by approximately one-half for each 8°C to 12°C in the temperature range from 50°C to 125°C.

[2] Refer to Section A.4, “Current Injection,” for more details.

[3] Parameter only applies in stop or pseudo stop mode.

APPENDIX B

EVB9S12NE64 BOARD CONFIGURATION

The EVB9S12NE64 evaluation board for the MC9S12NE64 microcontroller is equipped with two port connectors that provide access to all digital inputs and outputs. These ports are labeled MCU1_PORT and MCU2_GP PORT. The digital input channels, the digital output channels, and the inputs and outputs for the hatch control system are on MCU1_PORT, and the analog input channels are on MCU2_GP PORT.

The digital input channels are located on port A bits 0-7. There are eight digital input channels, and the first channel is on bit 0. These bits are located on pins 9-16 on the MCU1_PORT.

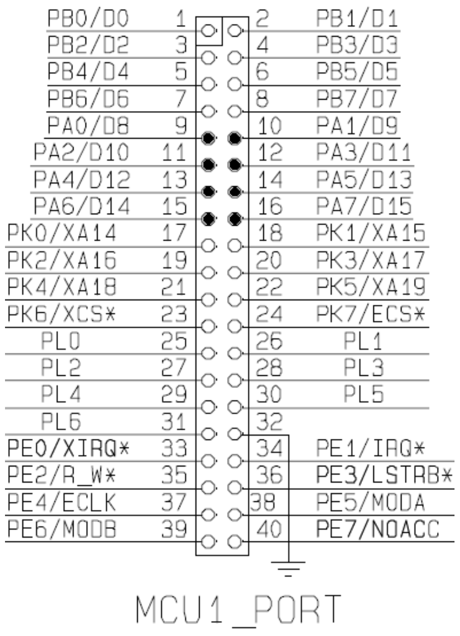


Figure 12: Digital Input Ports [12]

The digital output channels are located on port B bits 0-7. There are eight digital output channels, and the first channel is on bit 0. These bits located on pins 1-8 on the MCU1_PORT.

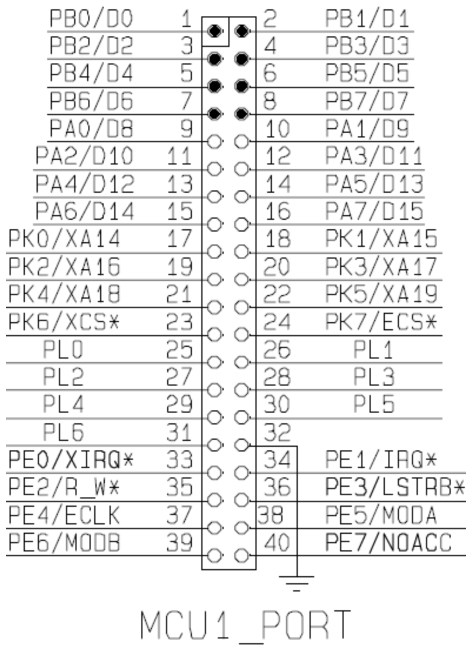


Figure 13: Digital Output Ports [12]

The ATD input channels are on port AD bits 0-7. There are eight analog input channels, and the first channel is on bit 0. These bits are located on pins 11-18 on the MCU2_GP PORT. The evaluation board has an adjustable potentiometer that provides a linear voltage output to port AD bit 0 located on the MCU2_GP PORT pin 17. This bit position corresponds to analog input channel 0, and the potentiometer can be used to sweep the channel value from 0[V] to 3.3[V]. The potentiometer is enabled and disabled through a jumper labeled RV1_EN. Remove this jumper to disable the potentiometer voltage input.

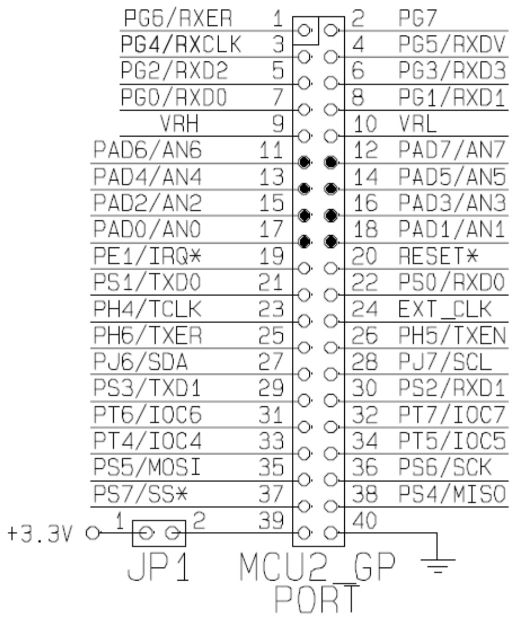


Figure 14: Analog Input Ports [12]

The digital inputs and outputs for the hatch control system are on port K bits 0-2. The hatch motor control is a digital output and is on port K bit 0. The hatch position sensors are digital inputs and are on port K bits 1 and 2. Bit 1 is the open sensor and bit 2 is the closed sensor. These bits are located on pins 17-19 on the MCU1_PORT.

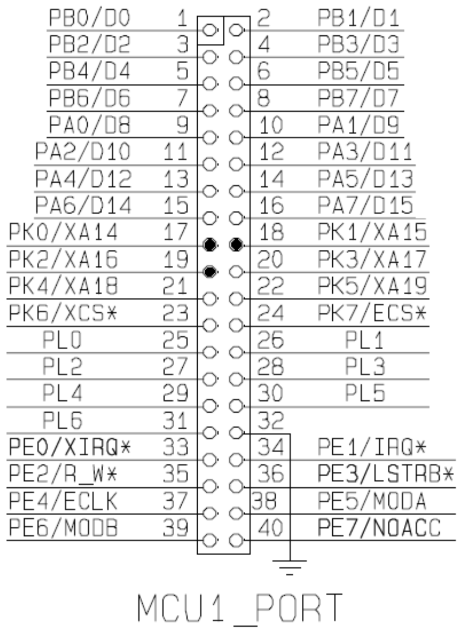


Figure 15: Hatch Control Ports [12]

The evaluation board is equipped with a power port terminal block and breadboard area. The power port is labeled POWER_PORT and provides access to the input voltage, +5[V], +3.3[V], and ground. The input voltage is directly connected to the board's power supply and is not switched by the board's ON/OFF switch. For this project, the power port was used to supply an input voltage and ground to the breadboard area for wiring some of the system peripherals.

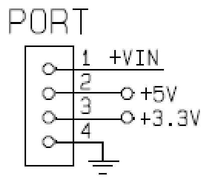


Figure 16: Power Port [12]

The evaluation board includes two serial communications ports with RS232 DB9-S connectors. These ports are configured with the COM_SW switch and the FLOW_SEL jumpers. The switch is used to enable the serial transmit and receive lines and configure flow control. For this project, data is transmitted and received on both serial ports, and no flow control is used. Switches 1-4 on the COM_SW switch are set to closed, and switches 5-8 are set to open. The FLOW_SEL jumpers are set to indicate no flow control.

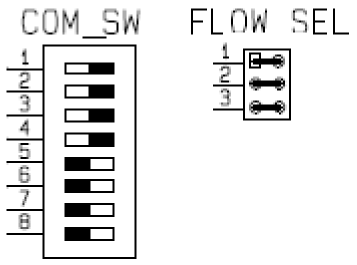


Figure 17: Com Switch and Flow Select Jumpers [12]

The evaluation board is also equipped with a reset button. This button is labeled RESET and is used to manually reset the microcontroller. There is a corresponding RESET indicator light that will be on for the duration of the reset signal.

APPENDIX C

COMMAND SET FOR THE ICI CONTROL SYSTEM

Notes:
All examples are given in hexadecimal.
All data is transmitted little-endian.
The channel indexing is 0-based (the first channel is at index 0).

Message structure:
```
ICI_DATA_SIZE 128
typedef struct (132 bytes)
{
  UINT16 Command; //2 bytes
  UINT16 NumData; //2 bytes, number of bytes in Data
  UINT8 Data[ICI_DATA_SIZE]; //128 bytes
} ici_message;
```
Example: Command - 0x0102, NumData - 0x0304, Data - 0xFF
```
Out: 02 01 04 03 FF
```

Maximum packet size:
```
sizeof(Command)+sizeof(NumData)+ICI_DATA_SIZE = 132
```

Commands (0x0000-0xFFFF):
```
GET_ATD_IN        0x0001
GET_DIGITAL_IN    0x0002
GET_DIGITAL_OUT   0x0003
SET_DIGITAL_OUT   0x0004
PASS_SCI0         0x0005
PASS_SCI1         0x0006
GET_HATCH_CONTROL 0x0007
SET_HATCH_CONTROL 0x0008
GET_HATCH_STATE   0x0009
SET_HATCH_STATE   0x000A
GET_HATCH_ERROR   0x000B
GET_ATD_IN_ALL    0x000C
GET_SCI0          0x000D
SET_SCI0          0x000E
GET_SCI1          0x000F
SET_SCI1          0x0010
GET_DEVICE_INFO   0x0011
GET_RAIN_INDEX    0x0012
SET_RAIN_INDEX    0x0013
GET_RAIN_WET      0x0014
SET_RAIN_WET      0x0015
GET_RAIN_DRY      0x0016
SET_RAIN_DRY      0x0017
```

Errors Codes (0x00-0xFF):
```
OK       0x00 //success
TIMEOUT  0x01 //time expired while processing command
COMMAND  0x02 //invalid command
DATA     0x03 //invalid data
INDEX    0x04 //invalid index for analog or digital channel
POINTER  0x05 //NULL pointer
```

```
OVERFLOW 0x06 //buffer overflow
BUSY     0x07 //processing command
STATE    0x08 //invalid hatch state
RAIN     0x09 //rain sensor detected rain
UNKNOWN  0xFF //unknown error
```

Get_Atd_In (0x0001)
Example: Get value of first atd input channel.

```
OUT: 01 00 01 00 00
     01 00              -Command
           01 00        -NumData
                 00     -Index of atd_in_0
 IN: 01 00 03 00 00 FF 03
     01 00              -Command
           03 00        -NumData
                 00     -ACK
                    FF 03 -Value of atd_in_0
```

Get_Digital_In (0x0002)
Example: Get value of first digital input channel.

```
OUT: 02 00 01 00 00
     02 00           –Command
           01 00     –NumData
                 00  –Index of digital_in
 IN: 02 00 02 00 00 00
     02 00           –Command
           01 00     –NumData
                 00  –ACK
                    00 -Value of digital_in_0
```

Get_Digital_Out (0x0003)
Example: Get value of first digital output channel.

```
OUT: 03 00 01 00 00
     03 00           –Command
           01 00     –NumData
                 00  –Index of digital_out
 IN: 03 00 02 00 00 00
     03 00           –Command
           02 00     –NumData
                 00  –ACK
                    00 –Value of digital_out_0
```

Set_Digital_Out (0x0004)
Example: Set value of first digital output channel to 1.

```
OUT: 04 00 02 00 00 01
     04 00           –Command
           02 00     –NumData
                 00  –Index of digital_out
                   01 –Value of digital_out_0
 IN: 04 00 01 00 00
     04 00           –Command
           01 00     –NumData
                 00  -ACK
```

Pass_Sci0 (0x0005)
Example: Pass ‘4D 52 20 39 30 30 30 30 0D’ to the first serial port and wait for a response with terminator ‘3E’.

```
OUT: 05 00 0A 00 3E 4D 52 20 39 30 30 30 30 0D
     05 00                                        –Command
           0A 00                                  –NumData
                 3E                               –Terminator
                    4D 52 50 39 30 30 30 30 0D    –Tx data
 IN: 05 00 0C 00 00 4D 52 20 39 30 30 30 30 0D 0A 3E
     05 00                                        –Command
           0C 00                                  –NumData
                 00                               –ACK
                    4D 52 20 39 30 30 30 30 0D 0A 3E –Rx data
```

Pass_Sci1 (0x0006)
Example: Pass ‘4D 52 20 39 30 30 30 30 0D’ to the second serial port and wait for a response with terminator ‘3E’.

```
OUT: 06 00 0A 00 3E 4D 52 20 39 30 30 30 30 0D
     06 00                                        –Command
           0A 00                                  –NumData
                 3E                               –Terminator
                    4D 52 50 39 30 30 30 30 0D    –Tx data
 IN: 06 00 0C 00 00 4D 52 20 39 30 30 30 30 0D 0A 3E
     06 00                                        –Command
           0C 00                                  –NumData
                 00                               –ACK
                    4D 52 20 39 30 30 30 30 0D 0A 3E –Rx data
```

Get_Hatch_Control (0x0007)
Example: Get the value of the hatch control.
```
OUT: 07 00 00 00
     07 00              –Command
           00 00        –NumData
 IN: 07 00 02 00 00 00
     07 00              –Command
           02 00        –NumData
                 00     –ACK
                    00 -Value of hatch_control
```
Hatch control values:
```
STOP  0x00 //not moving
START 0x01 //moving
```

Set_Hatch_Control (0x0008)
Example: Set the value of the hatch control to START (moves hatch).
```
OUT: 08 00 01 00 01
     08 00              –Command
           01 00        –NumData
                 01     –Value of hatch_control
 IN: 08 00 01 00 00
     08 00              –Command
           01 00        –NumData
                 00     -ACK
```
Hatch control values:
```
STOP  0x00 //not moving
START 0x01 //moving
```

Get_Hatch_State (0x0009)
Example: Get the value of the hatch state.
```
OUT: 09 00 00 00
     09 00              –Command
           00 00        –NumData
 IN: 09 00 02 00 00 01
     09 00              –Command
           02 00        –NumData
                 00     –ACK
                    01 -Value of hatch_state
```
Hatch state values:
```
NONE    0x00 //neither sensor active
OPEN     0x01 //open sensor active
CLOSED  0x02 //closed sensor active
INVALID 0x03 //both sensors active
```

Set_Hatch_State (0x000A)
Example: Set the value of the hatch state to OPEN (moves hatch to the open position).

```
OUT: 0A 00 01 00 01
     0A 00           –Command
           01 00     –NumData
                 01  –Value of hatch_state
 IN: 0A 00 01 00 00
     0A 00           –Command
           01 00     –NumData
                 00  -ACK
```

Hatch state values:

```
NONE    0x00 //neither sensor active
OPEN     0x01 //open sensor active
CLOSED  0x02 //closed sensor active
INVALID 0x03 //both sensors active
```

Get_Hatch_Error (0x000B)
Example: Get the value of the hatch error.

```
OUT: 0B 00 00 00
     0B 00           –Command
           00 00     –NumData
 IN: 0B 00 02 00 00 01
     0B 00           –Command
           02 00     –NumData
                 00  –ACK
                    01 -Value of hatch_error
```

Get_Atd_In_All (0x000C)
Example: Get value of all atd input channels.

```
OUT: 0C 00 00 00
     0C 00                            -Command
           00 00                      -NumData
 IN: 0C 00 11 00 00 FF 03 FF 03 … 03 FF 03
     0C 00                            -Command
           11 00                      -NumData
                 00                   -ACK
                    FF 03 FF 03 … FF 03 -Value of atd_in 0-7
```

Get_Sci0 (0x000D)
Example: Get data received from the first serial port.

```
OUT: 0D 00 00 00
     0D 00                                         –Command
           00 00                                   –NumData
 IN: 0D 00 0C 00 00 4D 52 20 39 30 30 30 30 0D 0A 3E
     0D 00                                         –Command
           0C 00                                   –NumData
                 00                                –ACK
                    4D 52 20 39 30 30 30 30 0D 0A 3E –Rx data
```

Set_Sci0 (0x000E)

Example: Set data to transmit to the first serial port.

```
OUT: 0E 00 0B 00 4D 52 20 39 30 30 30 30 0D 0A 3E
     0E 00                                        –Command
           0B 00                                  –NumData
                 4D 52 20 39 30 30 30 30 0D 0A 3E –Tx data
 IN: 0E 00 01 00 00
     0E 00                                        –Command
           01 00                                  –NumData
                 00                               –ACK
```

Get_Sci1 (0x000F)

Example: Get data received from the second serial port.

```
OUT: 0F 00 00 00
     0F 00                                           –Command
           00 00                                     –NumData
 IN: 0F 00 0C 00 00 4D 52 20 39 30 30 30 30 0D 0A 3E
     0F 00                                           –Command
           0C 00                                     –NumData
                 00                                  –ACK
                    4D 52 20 39 30 30 30 30 0D 0A 3E –Rx data
```

Set_Sci1 (0x0010)

Example: Set data to transmit to the first serial port.

```
OUT: 10 00 0B 00 4D 52 20 39 30 30 30 30 0D 0A 3E
     10 00                                        –Command
           0B 00                                  –NumData
                 4D 52 20 39 30 30 30 30 0D 0A 3E –Tx data
 IN: 10 00 01 00 00
     10 00                                        –Command
           01 00                                  –NumData
                 00                               –ACK
```

Get_Device_Info (0x0011)

Example: Get the device information.

```
OUT: 11 00 00 00
     11 00                                          –Command
           00 00                                    –NumData
 IN: 11 00 41 00 00 01 00 00 01 FF 3F … 00 00 00 00
     11 00                                          –Command
           41 00                                    –NumData
                 00                                 –ACK
                    01 00 00 01 FF 3F … 00 00 00 00 –Device info
```

ICI device information (64 bytes):

```
typedef struct
{
  UINT32 FirmwareVersion;  //Firmware version.
  UINT16 AtdResolution;    //ATD resolution.
  UINT16 HatchOpenTimeout; //Hatch open timeout [ms].
  UINT16 SerialTimeout;    //Serial timeout [ms].
  UINT8 ExtraB[54];        //Extra bytes.
} ici_device_info;
```

Get_Rain_Index (0x0012)

Example: Get the index of the rain sensor atd input.

```
OUT: 12 00 00 00
     12 00             –Command
           00 00       –NumData
 IN: 12 00 02 00 00 00
     12 00             –Command
           02 00       –NumData
                 00    –ACK
                    00 –Atd index
```

Set_Rain_Index (0x0013)

Example: Set the index of the rain sensor atd input.

```
OUT: 13 00 01 00 00
     13 00             –Command
           01 00       –NumData
                 00    -Atd index
 IN: 13 00 01 00 00
     13 00             –Command
           01 00       –NumData
                 00    –ACK
```

Get_Rain_Wet (0x0014)
Example: Get the rain sensor wet threshold.

```
OUT: 14 00 00 00
     14 00                 –Command
           00 00           –NumData
 IN: 14 00 03 00 00 D1 01
     14 00                 –Command
           03 00           –NumData
                 00        –ACK
                    D1 01 –Wet threshold
```

Set_Rain_Wet (0x0015)
Example: Set the rain sensor wet threshold.

```
OUT: 15 00 02 00 D1 01
     15 00              –Command
           02 00        –NumData
                 D1 01 –Wet threshold
 IN: 15 00 01 00 00
     15 00              –Command
           01 00        –NumData
                 00     –ACK
```

Get_Rain_Dry (0x0016)
Example: Get the rain sensor dry threshold.

```
OUT: 16 00 00 00
     16 00                 –Command
           00 00           –NumData
 IN: 16 00 03 00 00 6C 02
     16 00                 –Command
           03 00           –NumData
                 00        –ACK
                    6C 02 –Dry threshold
```

Set_Rain_Dry (0x0017)
Example: Set the rain sensor wet threshold.

```
OUT: 17 00 02 00 6C 02
     17 00              –Command
           02 00        –NumData
                 6C 02 –Wet threshold
 IN: 17 00 01 00 00
     17 00              –Command
           01 00        –NumData
                 00     –ACK
```

APPENDIX D

SOURCE CODE FOR THE ICI EMBEDDED SYSTEM

```
//-------------------------------------------------------------------
// atd.h - ATD header file for ProjectICI.
// Handles the initialization and control of the Analog-to-digital
// converter and the 8 analog input channels.
//-------------------------------------------------------------------

//-------------------------------------------------------------------
#ifndef INCLUDE_ATD_H
#define INCLUDE_ATD_H
//-------------------------------------------------------------------

//-------------------------------------------------------------------
#include "datatypes.h"
//-------------------------------------------------------------------

//-------------------------------------------------------------------
//ATD initialization.
void atd_init(void);
//Get the ATD channel result.
UINT8 get_atd_in(UINT8 Index, UINT16* Data);
//Get the ATD converter resolution.
UINT8 get_atd_resolution(UINT16* Data);
//-------------------------------------------------------------------

//-------------------------------------------------------------------
#endif
//-------------------------------------------------------------------

//-------------------------------------------------------------------
// atd.c - ATD source file for ProjectICI.
// Handles the initialization and control of the Analog-to-digital
// converter and the 8 analog input channels.
//-------------------------------------------------------------------

//-------------------------------------------------------------------
#include "atd.h"
#include "MC9S12NE64.h"
#include "error_defines.h"
//-------------------------------------------------------------------

//-------------------------------------------------------------------
// Function: atd_init - ATD initialization. Powers the ATD, configures
//           the ATD for 1 conversion per sequence, sets the resolution
//           to 10-bits, set the ATD clock to 1.79[Mz].
// Parameters: void
// Return Value: void
//-------------------------------------------------------------------
void atd_init(void)
{
  //ATD Power Up bit 7, 1 = power on.
  ATDCTL2 = 0x80;
  //Conversion Sequence Length bits 6-3
  //0001 = 1 conversion per sequence.
  ATDCTL3   = 0x08;
  //A/D Resolution Select bit 7, 0 = 10-bit resolution.
  //ATD Clock Prescaler bits 4-0, 00110 = 6 (divide by 14).
  //ATD Clock = 25MHz/[(6+1)*2] = 1.79MHz.
  ATDCTL4 = 0x06;
}
//-------------------------------------------------------------------

//-------------------------------------------------------------------
// Function: get_atd_in - Get the ATD channel result.
// Parameters: Index - Index of ATD channel (0 - 7).
//             Data - Pointer to 2-byte variable to hold the result.
// Return Value: Error code. 0 == success.
//-------------------------------------------------------------------
UINT8 get_atd_in(UINT8 Index, UINT16* Data)
{
  UINT8 retval = OK;

  if (!Data) //Invalid pointer.
  {
    retval = ERROR_POINTER;
  }
  else if (Index > 7) //Invalid Index.
  {
    retval = ERROR_INDEX;
  }
  else
  {
    //Disable all interrupts while converting ATD.
    //This is necessary to prevent another process from calling
    //this function during the conversion.
    __asm SEI;
    //Select the channel and configure the result.
    //ATD Result Register Data Justification bit 7
    //1 = right justified.
    //ATD Analog Input Channel Select bits 2-0.
    ATDCTL5 = 0x80 | Index;
    //Wait for the ATD to finish converting the channel result.
    //ATD Sequence Complete Flag bit 7
    // 1 = conversion sequence completed.
    while (ATDSTAT0_SCF == 0);
    //Read the channel result.
    *Data = (ATDDR0H << 8) | ATDDR0L; //10-bit result.
    //Reset the conversion sequence complete flag.
    //ATD Sequence Complete Flag bit 7, write 1 clears flag.
    ATDSTAT0_SCF = 1;
    //Enable all interrupts.
    __asm CLI;
  }

  return retval;
}
//-------------------------------------------------------------------

//-------------------------------------------------------------------
```

```
// Function: get_atd_resolution - Get the ATD converter resolution.
// Parameters: Data - Pointer to 2-byte variable to hold the result.
// Return Value: Error code. 0 == success.
//-------------------------------------------------------------------
UINT8 get_atd_resolution(UINT16* Data)
{
  UINT8 retval = OK;

  if (!Data) //Invalid pointer.
  {
    retval = ERROR_POINTER;
  }
  else
  {
    if (ATDCTL4_SRES8 == 0)
    {
      //10-bit.
      *Data = 0x03FF;
    }
    else
    {
      //8-bit.
      *Data = 0x00FF;
    }
  }

  return retval;
}
//-------------------------------------------------------------------

//-------------------------------------------------------------------
// digital_io.h - Digital I/O header file for ProjectICI.
// Handles the initialization and control of the digital input and
// output channels. Handles the hatch control system.
//-------------------------------------------------------------------

//-------------------------------------------------------------------
#ifndef INCLUDE_DIGITAL_IO_H
#define INCLUDE_DIGITAL_IO_H
//-------------------------------------------------------------------

//-------------------------------------------------------------------
#include "datatypes.h"
//-------------------------------------------------------------------

//-------------------------------------------------------------------
//Set hatch open timeout, number of real-time interrupts.
//Assumes the RTI is set to 10[ms].
#define HATCH_TIMEOUT 6000 //60[s].
//Hatch control values.
#define HATCH_MOVE_STOP  0x00 //Hatch is not moving.
#define HATCH_MOVE_START 0x01 //Hatch is moving.
//Hatch states.
#define HATCH_STATE_NONE    0x00 //Hatch is not open or closed.
```

```
#define HATCH_STATE_OPEN    0x01 //Hatch is open.
#define HATCH_STATE_CLOSED  0x02 //Hatch is closed.
#define HATCH_STATE_INVALID 0x03 //Hatch is open and closed.
//Rain sensor information.
#define HATCH_RAIN_INDEX 0x00  //Assume rain sensor is on ATD0.
#define HATCH_RAIN_WET   0x1D1 //465[TICKS] (1.5[V])
#define HATCH_RAIN_DRY   0x26C //620[TICKS] (2[V])

//Hatch state and information.
typedef struct
{
  UINT8 Start;     //Start hatch state.
  UINT8 Target;    //Desired hatch state.
  UINT16 MoveTime; //Time elapsed since last hatch state.
  UINT16 Timeout;  //Total time hatch is allowed to move.
  UINT8 Error;     //Hatch error.
} hatch_state_info;

//Rain sensor information.
typedef struct
{
  UINT8 Index;  //Index of rain sensor ATD.
  UINT16 Wet;   //Wet value.
  UINT16 Dry;   //Dry value.
  UINT16 Value; //Current value.
} rain_sensor_info;
//-----------------------------------------------------------------

//-----------------------------------------------------------------
//Digital I/O initialization.
void digital_io_init(void);
//Get the digital input channel value.
UINT8 get_digital_in(UINT8 Index, UINT8 *Data);
//Get the digital output channel value.
UINT8 get_digital_out(UINT8 Index, UINT8 *Data);
//Set the digital output channel value.
UINT8 set_digital_out(UINT8 Index, UINT8 Data);
//Get the hatch control value.
UINT8 get_hatch_control(UINT8* Data);
//Set the hatch control value.
UINT8 set_hatch_control(UINT8 Data);
//Get the hatch state.
UINT8 get_hatch_state(UINT8* Data);
//Set the hatch state.
UINT8 set_hatch_state(UINT8 Data);
//Get the hatch error.
UINT8 get_hatch_error(UINT8* Data, UINT8 Clear);
//Updates timers used for hatch control.
void update_hatch_timers(void);
//Move through hatch control system.
void process_hatch_status(void);
//Get the rain sensor ATD index.
UINT8 get_rain_index(UINT8* Data);
//Set the rain sensor ATD index.
```

```
UINT8 set_rain_index(UINT8 Data);
//Get the rain sensor wet threshold.
UINT8 get_rain_wet(UINT16* Data);
//Set the rain sensor wet threshold.
UINT8 set_rain_wet(UINT16 Data);
//Get the rain sensor dry threshold.
UINT8 get_rain_dry(UINT16* Data);
//Set the rain sensor dry threshold.
UINT8 set_rain_dry(UINT16 Data);
//-----------------------------------------------------------------

//-----------------------------------------------------------------
#endif
//-----------------------------------------------------------------

//-----------------------------------------------------------------
// digital_io.c - Digital I/O source file for ProjectICI.
// Handles the initialization and control of the digital input and
// output channels. Handles the hatch control system.
//-----------------------------------------------------------------

//-----------------------------------------------------------------
#include "digital_io.h"
#include "MC9S12NE64.h"
#include "error_defines.h"
#include "atd.h"
#include <string.h>
//-----------------------------------------------------------------

//-----------------------------------------------------------------
//Hatch state and information.
hatch_state_info volatile hatch_state;
//Timer to limit the time the hatch is open.
UINT16 volatile hatch_open_timer;
//Rain sensor information.
rain_sensor_info volatile rain_sensor;
//-----------------------------------------------------------------

//-----------------------------------------------------------------
// Function: digital_io_init - Digital I/O initialization. Sets Port A
//           to digital input channels and sets Port B to digital
//           output channels. Sets Port K bits 2 and 1 to inputs for
//           the hatch sensors, and sets bit 0 to an output for the
//           hatch control.
// Parameters: void
// Return Value: void
//-----------------------------------------------------------------
void digital_io_init(void)
{
  //Set Port A to digital input.
  //Data Direction Register for Port A, 0x00 = all input.
  DDRA = 0x00;
  //Set Port B to digital output.
  //Data Direction Register for Port B, 0xFF = all output.
  DDRB = 0xFF;

  //Use bits 2-0 on Port K for the hatch control system.
  //Data Direction Register for Port K, 0x01 = bits 2-1 inputs,
  //bit 0 output.
  DDRK = 0x01;
  //Clear the hatch_state.
  (void)memset((void*)(&hatch_state), 0, sizeof(hatch_state_info));
  //Clear hatch open timer.
  hatch_open_timer = 0;
  //Configure the rain sensor.
  rain_sensor.Index = HATCH_RAIN_INDEX;
  rain_sensor.Wet = HATCH_RAIN_WET;
  rain_sensor.Dry = HATCH_RAIN_DRY;
}
//------------------------------------------------------------------

//------------------------------------------------------------------
// Function: update_hatch_timers - Updates timers used for hatch
//           control. Updated every 10[ms] by RTI.
// Parameters: void
// Return Value: void
//------------------------------------------------------------------
void update_hatch_timers(void)
{
  UINT8 retval;

  //Update hatch open timer.
  if (hatch_open_timer > 0)
  {
    hatch_open_timer--;
    if (hatch_open_timer == 0)
    {
      //Hatch open time expired. Close the hatch.
      (void)set_hatch_state(HATCH_STATE_CLOSED);
    }
  }

  //Update rain sensor.
  retval = get_atd_in(rain_sensor.Index, &(rain_sensor.Value));
  if (retval == OK)
  {
    if (rain_sensor.Value < HATCH_RAIN_WET)
    {
      //The rain sensor is reading less than the wet threshold (rain).
      if (hatch_state.Error != ERROR_RAIN)
      {
        //Set the hatch error to rain.
        hatch_state.Error = ERROR_RAIN;
        //Close the hatch.
        (void)set_hatch_state(HATCH_STATE_CLOSED);
      }
    }
    else if (rain_sensor.Value > HATCH_RAIN_DRY)
    {
      if (hatch_state.Error == ERROR_RAIN)
      {
        //Rain was previously detected, but now the sensor is reading
        //greater than the dry threshold (not raining).
        //Set the hatch error to not raining.
        hatch_state.Error = OK;
      }
    }
  }

  //Update hatch control timer.
  if (hatch_state.Timeout > 0)
  {
    hatch_state.Timeout--;
    //Move through hatch control system.
    process_hatch_status();
  }
}
//------------------------------------------------------------------

//------------------------------------------------------------------
// Function: process_hatch_status - Move through hatch control system.
// Parameters: void
// Return Value: void
//------------------------------------------------------------------
void process_hatch_status(void)
{
  UINT8 control;
  UINT8 state;

  //Get hatch control (moving, not moving).
  (void)get_hatch_control(&control);
  //Get hatch state (none, open, closed, invalid).
  (void)get_hatch_state(&state);

  if (control == HATCH_MOVE_START) //Hatch is moving.
  {
    if (hatch_state.Error != OK &&
        hatch_state.Error != ERROR_RAIN) //Error.
    {
      //Stop moving.
      (void)set_hatch_control(HATCH_MOVE_STOP);
    }
    else if (hatch_state.Target == state) //Success.
    {
       //Stop moving.
      (void)set_hatch_control(HATCH_MOVE_STOP);
      if (hatch_state.Target == HATCH_STATE_OPEN) //Open.
      {
        //Set open timeout.
        hatch_open_timer = HATCH_TIMEOUT;
      }
    }
```

```
else if (hatch_state.Timeout == 0) //Timeout.
{
  //Stop moving.
  (void)set_hatch_control(HATCH_MOVE_STOP);
  hatch_state.Error = ERROR_TIMEOUT;
}
else //No error. Target not reached. Process current info.
{
  switch (state)
  {
    case HATCH_STATE_NONE:
      //Neither open or closed. Increment move time.
      hatch_state.MoveTime++;
      break;
    case HATCH_STATE_OPEN:
      if (hatch_state.Start == HATCH_STATE_NONE)
      {
        //Started in between sensors. Set start state to open.
        hatch_state.Start = HATCH_STATE_OPEN;
        //Reset move time.
        hatch_state.MoveTime = 0;
      }
      else if (hatch_state.Start == HATCH_STATE_OPEN &&
               hatch_state.MoveTime != 0)
      {
        //Hatch started open, moved, and opened again.
        //Closed sensor never found.
        //Try to close. Keep moving for half of move time.
        hatch_state.Timeout = hatch_state.MoveTime / 2;
        //reset move time.
        hatch_state.MoveTime = 0;
      }
      break;
    case HATCH_STATE_CLOSED:
      if (hatch_state.Start == HATCH_STATE_NONE)
      {
        //Started in between sensors. Set start state to closed.
        hatch_state.Start = HATCH_STATE_CLOSED;
        //Reset move time.
        hatch_state.MoveTime = 0;
      }
      else if (hatch_state.Start == HATCH_STATE_CLOSED &&
               hatch_state.MoveTime != 0)
      {
        //Hatch started closed, moved, and closed again.
        //Open sensor never found.
        //Stop moving with hatch closed.
        (void)set_hatch_control(HATCH_MOVE_STOP);
        //Set hatch error.
        hatch_state.Error = ERROR_TIMEOUT;
      }
      break;
    case HATCH_STATE_INVALID:
      //Invalid state reached. Stop moving.
```

```
          (void)set_hatch_control(HATCH_MOVE_STOP);
          hatch_state.Error = ERROR_STATE;
          break;
      }
    }
  }
}

//------------------------------------------------------------------
// Function: get_digital_in - Get the digital input channel value.
// Parameters: Index - Index of digital input channel (0 - 7).
//             Data - Pointer to 1-byte variable to hold the value.
// Return Value: Error code. 0 == success.
//------------------------------------------------------------------
UINT8 get_digital_in(UINT8 Index, UINT8 *Data)
{
  UINT8 retval = OK;

  if (!Data) //Invalid pointer.
  {
    retval = ERROR_POINTER;
  }
  else if (Index > 7) //Invalid index.
  {
    retval = ERROR_INDEX;
  }
  else
  {
    //Read the channel result.
    *Data = (PORTA >> Index) & 0x01;
  }

  return retval;
}

//------------------------------------------------------------------

//------------------------------------------------------------------
// Function: get_digital_out - Get the digital output channel value.
// Parameters: Index - Index of digital output channel (0 - 7).
//              Data - Pointer to 1-byte variable to hold the value.
// Return Value: Error code. 0 == success.
//------------------------------------------------------------------
UINT8 get_digital_out(UINT8 Index, UINT8 *Data)
{
  UINT8 retval = OK;

  if (!Data) //Invalid pointer.
  {
    retval = ERROR_POINTER;
  }
  else if (Index > 7) //Invalid index.
  {
    retval = ERROR_INDEX;
  }
  else
  {
    //Read the channel result.
    *Data = (PORTB >> Index) & 0x01;
  }

  return retval;
}
//------------------------------------------------------------------

//------------------------------------------------------------------
// Function: set_digital_out - Set the digital output channel value.
// Parameters: Index - Index of digital output channel (0 - 7).
//             Data - 1-byte variable with the output value.
// Return Value: Error code. 0 == success.
//------------------------------------------------------------------
UINT8 set_digital_out(UINT8 Index, UINT8 Data)
{
  UINT8 retval = OK;

  if (Index > 7) //Invalid index.
  {
    retval = ERROR_INDEX;
  }
  else if (Data > 1) //Invalid value.
  {
    retval = ERROR_DATA;
  }
  else
  {
    if (Data == 1)
      PORTB |= 0x01 << Index; //Set output (1).
    else
      PORTB &= ~(0x01 << Index); //Clear output (0).
  }

  return retval;
}
//------------------------------------------------------------------

//------------------------------------------------------------------
// Function: get_hatch_control - Get the hatch control value. This is
//           the digital output line that determines if the hatch motor
//           is moving (1 == moving).
// Parameters: Data - Pointer to 1-byte variable to hold the value.
// Return Value: Error code. 0 == success.
//------------------------------------------------------------------
UINT8 get_hatch_control(UINT8* Data)
{
  UINT8 retval = OK;

  if (!Data) //Invalild pointer.
  {
    retval = ERROR_POINTER;
  }
  else
  {
    //Read the hatch control value.
    *Data = PORTK & 0x01;
  }

  return retval;
}
//------------------------------------------------------------------

//------------------------------------------------------------------
// Function: set_hatch_control - Set the hatch control value. This is
//           the digital output line that determines if the hatch motor
//           is moving (1 == moving).
// Parameters: Data - 1-byte variable with the hatch control value.
// Return Value: Error code. 0 == success.
//------------------------------------------------------------------
UINT8 set_hatch_control(UINT8 Data)
{
  UINT8 retval = OK;

  if (Data > 1) //Invalid data.
  {
    retval = ERROR_DATA;
  }
  else
  {
    if (Data == 1)
      PORTK |= 0x01; //Set control (bit 0 == 1 for moving).
    else
      PORTK &= ~0x01; //Clear control (bit 0 == 0 for not moving).
  }

  return retval;
}
//------------------------------------------------------------------

//------------------------------------------------------------------
// Function: get_hatch_state - Get the hatch state, the digital input
//           lines that correspond to the hatch position (none==0x00,
//           open==0x01, closed==0x02, error==0x03).
// Parameters: Data - Pointer to 1-byte variable to hold the value.
// Return Value: Error code. 0 == success.
//------------------------------------------------------------------
UINT8 get_hatch_state(UINT8* Data)
{
  UINT8 retval = OK;

  if (!Data) //Invalid pointer.
  {
    retval = ERROR_POINTER;
  }
  else
  {
    //Read the hatch state.
    *Data = (PORTK & 0x06) >> 0x01; //Bits 2-1
  }

  return retval;
}
//------------------------------------------------------------------

//------------------------------------------------------------------
// Function: set_hatch_state - Set the hatch state. Move the hatch
//           until the correct state is reached (open==0x01,
//           closed==0x10).
// Parameters: Data - 1-byte variable with the hatch state.
// Return Value: Error code. 0 == success.
//------------------------------------------------------------------
UINT8 set_hatch_state(UINT8 Data)
{
  UINT8 retval = OK;
  UINT8 state;

  if (hatch_state.Error != OK && //Error. Do nothing.
      hatch_state.Error != ERROR_RAIN)
  {
    retval = hatch_state.Error;
  }
  else if (Data == HATCH_STATE_NONE ||  //Invalid hatch state.
           Data == HATCH_STATE_INVALID)
  {
    retval = ERROR_DATA;
  }
  else if (hatch_state.Error == ERROR_RAIN && //Rain. Allow close.
           Data != HATCH_STATE_CLOSED)
  {
    retval = hatch_state.Error;
  }
  else
  {
    //Get the current hatch state.
    retval = get_hatch_state(&state);
    if (retval == OK)
    {
      if (state != Data) //Current state does not match new state.
      {
        //Set hatch state info.
        hatch_state.Start = state;
        hatch_state.Target = Data;
        hatch_state.MoveTime = 0;
        hatch_state.Timeout = HATCH_TIMEOUT;
        //Start moving the hatch.
        (void)set_hatch_control(HATCH_MOVE_START);
      }
      if (state == HATCH_STATE_OPEN) //Hatch is currently open.
      {
        //Reset the hatch open timer.
        hatch_open_timer = HATCH_TIMEOUT;
      }
    }
  }

  return retval;
}

//-------------------------------------------------------------------
// Function: get_hatch_error - Get the hatch error. Attempts to set the
//           hatch state are ignored until this error is cleared.
// Parameters: Data - Pointer to 1-byte variable to hold the value.
//             Clear - Boolean value to indicate if the error should be
//             cleared (1 == clear the error).
// Return Value: Error code. 0 == success.
//-------------------------------------------------------------------
UINT8 get_hatch_error(UINT8* Data, UINT8 Clear)
{
  UINT8 retval = OK;

  if (!Data) //Invalid pointer.
  {
    retval = ERROR_POINTER;
  }
  else
  {
    //Read the hatch error.
    *Data = hatch_state.Error;
    //Check if the error should be cleared. Do not clear the error if
    //it is equal to ERROR_RAIN.
    if (Clear != 0 && hatch_state.Error != ERROR_RAIN)
    {
      //Clear the hatch error.
      hatch_state.Error = OK;
    }
  }

  return retval;
}
//-------------------------------------------------------------------

//-------------------------------------------------------------------
// Function: get_rain_index - Get the rain sensor ATD index.
// Parameters: Data - Pointer to 1-byte variable to hold the result.
// Return Value: Error code. 0 == success.
//-------------------------------------------------------------------
UINT8 get_rain_index(UINT8* Data)
{
  UINT8 retval = OK;

  if (!Data) //Invalid pointer.
  {
    retval = ERROR_POINTER;
  }
  else
  {
    //Get the ATD index.
    *Data = rain_sensor.Index;
  }

  return retval;
}
//-------------------------------------------------------------------

//-------------------------------------------------------------------
// Function: set_rain_index - Set the rain sensor ATD index.
// Parameters: Data - 1-byte variable with the index.
// Return Value: Error code. 0 == success.
//-------------------------------------------------------------------
UINT8 set_rain_index(UINT8 Data)
{
  UINT8 retval = OK;

  if (Data > 7) //Invalid index.
  {
    retval = ERROR_INDEX;
  }
  else
  {
    //Set the ATD index.
    rain_sensor.Index = Data;
  }

  return retval;
}
//-------------------------------------------------------------------

//-------------------------------------------------------------------
// Function: get_rain_wet - Get the rain sensor wet threshold.
// Parameters: Data - Pointer to 2-byte variable to hold the result.
// Return Value: Error code. 0 == success.
//-------------------------------------------------------------------
UINT8 get_rain_wet(UINT16* Data)
{
  UINT8 retval = OK;

  if (!Data) //Invalid pointer.
  {
    retval = ERROR_POINTER;
  }
  else
  {
    //Get the wet threshold.
    *Data = rain_sensor.Wet;
  }
  return retval;
}
//------------------------------------------------------------------

//------------------------------------------------------------------
// Function: set_rain_wet - Set the rain sensor wet threshold.
// Parameters: Data - 2-byte variable with the wet threshold.
// Return Value: Error code. 0 == success.
//------------------------------------------------------------------
UINT8 set_rain_wet(UINT16 Data)
{
  UINT8 retval = OK;
  UINT16 Resolution;

  retval = get_atd_resolution(&Resolution);

  if (retval == OK)
  {
    if (Data > Resolution) //Invalid data.
    {
      retval = ERROR_DATA;
    }
    else
    {
      //Set the wet threshold.
      rain_sensor.Wet = Data;
    }
  }

  return retval;
}
//------------------------------------------------------------------

//------------------------------------------------------------------
// Function: get_rain_dry - Get the rain sensor dry threshold.
// Parameters: Data - Pointer to 2-byte variable to hold the result.
// Return Value: Error code. 0 == success.
//------------------------------------------------------------------
UINT8 get_rain_dry(UINT16* Data)
{
  UINT8 retval = OK;

  if (!Data) //Invalid pointer.
  {
    retval = ERROR_POINTER;
  }
  else
  {
    //Get the dry threshold.
    *Data = rain_sensor.Dry;
  }

  return retval;
}
```

```
//-------------------------------------------------------------------

//-------------------------------------------------------------------
// Function: set_rain_dry - Set the rain sensor dry threshold.
// Parameters: Data - 2-byte variable with the wet threshold.
// Return Value: Error code. 0 == success.
//-------------------------------------------------------------------
UINT8 set_rain_dry(UINT16 Data)
{
  UINT8 retval = OK;
  UINT16 Resolution;

  retval = get_atd_resolution(&Resolution);

  if (retval == OK)
  {
    if (Data > Resolution)
    {
      retval = ERROR_DATA;
    }
    else
    {
      //Set the dry threshold.
      rain_sensor.Dry = Data;
    }
  }

  return retval;
}
//-------------------------------------------------------------------

//-------------------------------------------------------------------
// error_defines.h - Error defines header file for ProjectICI.
// Includes the defines for all IDI error codes.
//-------------------------------------------------------------------

//-------------------------------------------------------------------
#ifndef INCLUDE_ERROR_DEFINES_H
#define INCLUDE_ERROR_DEFINES_H
//-------------------------------------------------------------------

//-------------------------------------------------------------------
//success (0x00)
#define OK 0 //success.
//errors (0x01-0xFF)
#define ERROR_TIMEOUT  0x01 //time expired while processing command.
#define ERROR_COMMAND  0x02 //invalid command.
#define ERROR_DATA     0x03 //invalid data.
#define ERROR_INDEX    0x04 //invalid channel index.
#define ERROR_POINTER  0x05 //NULL pointer.
#define ERROR_OVERFLOW 0x06 //buffer overflow.
#define ERROR_BUSY     0x07 //processing command.
#define ERROR_STATE    0x08 //invalid hatch state.
#define ERROR_RAIN     0x09 //rain sensor detected rain.
```

```
#define ERROR_UNKNOWN  0xFF //unknown error.
//-------------------------------------------------------------------

//-------------------------------------------------------------------
#endif
//-------------------------------------------------------------------

//-------------------------------------------------------------------
// ici.h - ICI control system header file for ProjectICI.
// Handles the initialization and control of the ICI control system.
// Processes incoming messages, executes message commands, and
// generates a response.
//-------------------------------------------------------------------

//-------------------------------------------------------------------
#ifndef INCLUDE_ICI_H
#define INCLUDE_ICI_H
//-------------------------------------------------------------------

//-------------------------------------------------------------------
#include "datatypes.h"
//-------------------------------------------------------------------

//-------------------------------------------------------------------
//Size of message data buffer.
#define ICI_DATA_SIZE 128

//ICI commands (0x0000-0xFFFF).
//Get ATD input channel.
#define ICI_COMMAND_GET_ATD_IN        0x0001
//Get digital input channel.
#define ICI_COMMAND_GET_DIGITAL_IN    0x0002
//Get digital output channel.
#define ICI_COMMAND_GET_DIGITAL_OUT   0x0003
//Set digital output channel.
#define ICI_COMMAND_SET_DIGITAL_OUT   0x0004
//Pass data to SCI0 and wait for response.
#define ICI_COMMAND_PASS_SCI0         0x0005
//Pass data to SCI1 and wait for response.
#define ICI_COMMAND_PASS_SCI1         0x0006
//Get hatch control (START, STOP).
#define ICI_COMMAND_GET_HATCH_CONTROL 0x0007
//Set hatch control (START, STOP).
#define ICI_COMMAND_SET_HATCH_CONTROL 0x0008
//Get hatch state (NONE, OPEN, CLOSED, INVALID).
#define ICI_COMMAND_GET_HATCH_STATE   0x0009
//Set hatch state (OPEN, CLOSED).
#define ICI_COMMAND_SET_HATCH_STATE   0x000A
//Get hatch control error.
#define ICI_COMMAND_GET_HATCH_ERROR   0x000B
//Get all ATD input channels.
#define ICI_COMMAND_GET_ATD_IN_ALL    0x000C
//Write data to SCI0 transmit buffer.
#define ICI_COMMAND_GET_SCI0          0x000D
```

```
//Read data from SCI0 receive buffer.
#define ICI_COMMAND_SET_SCI0          0x000E
//Write data to SCI1 transmit buffer.
#define ICI_COMMAND_GET_SCI1          0x000F
//Read data from SCI1 receive buffer.
#define ICI_COMMAND_SET_SCI1          0x0010
//Get device information.
#define ICI_COMMAND_GET_DEVICE_INFO   0x0011
//Get the rain sensor ATD index.
#define ICI_COMMAND_GET_RAIN_INDEX    0x0012
//Set the rain sensor ATD index.
#define ICI_COMMAND_SET_RAIN_INDEX    0x0013
//Get the rain sensor wet threshold.
#define ICI_COMMAND_GET_RAIN_WET      0x0014
//Set the rain sensor wet threshold.
#define ICI_COMMAND_SET_RAIN_WET      0x0015
//Get the rain sensor dry threshold.
#define ICI_COMMAND_GET_RAIN_DRY      0x0016
//Set the rain sensor dry threshold.
#define ICI_COMMAND_SET_RAIN_DRY      0x0017

//ICI message states.
#define ICI_MESSAGE_NONE     0x00 //No message.
#define ICI_MESSAGE_NEW      0x01 //New message.
#define ICI_MESSAGE_WAITING  0x02 //Waiting to finish message.
#define ICI_MESSAGE_FINISHED 0x03 //Finished message.

//ICI interface types.
#define ICI_TYPE_TCP    0x00 //Message to/from TCP.
#define ICI_TYPE_SERIAL 0x01 //Message to/from serial.

//ICI device information.
#define ICI_DEVICE_FIRMWARE 0x01000003 //Firmware version 1.0.0.3.

//ICI message header.
typedef struct
{
  UINT16 Command; //Command.
  UINT16 NumData; //Number of bytes in Data.
} ici_message_header;

//ICI message (header and data).
typedef struct
{
  UINT16 Command; //Command.
  UINT16 NumData; //Number of bytes in Data.
  UINT8 Data[ICI_DATA_SIZE]; //Data.
} ici_message;

//ICI message state and information.
typedef struct
{
  UINT8 State; //Message state.
  UINT8 From;  //Message interface.
  UINT8 Index; //Index of message interface.
} ici_message_info;

//ICI device information (64 bytes).
typedef struct
{
  UINT32 FirmwareVersion;  //Firmware version.
  UINT16 AtdResolution;    //ATD resolution.
  UINT16 HatchOpenTimeout; //Hatch open timeout [ms].
  UINT16 SerialTimeout;    //Serial timeout [ms].
  UINT8 ExtraB[54];        //Extra bytes.
} ici_device_info;
//-------------------------------------------------------------------

//-------------------------------------------------------------------
//ICI control system initialization.
void ici_init(void);
//Move through ICI control system.
void ici_run(void);
//Clear the ICI message.
void ici_clearmessage(void);
//Process the ICI message.
UINT8 ici_process_message(UINT8 From, UINT8 Index, UINT8* Data,
  UINT16 NumData);
//Fill the ICI device information structure.
UINT8 get_device_info(ici_device_info* device_info);
//Swap the order of the fields in the ICI device information structure.
UINT8 swap_device_info(ici_device_info* device_info);
//Swap the order of the bytes in the array.
void swap_byte_order(UINT8* Address, int Size);
//-------------------------------------------------------------------

//-------------------------------------------------------------------
#endif
//-------------------------------------------------------------------

//-------------------------------------------------------------------
// ici.c - ICI control system source file for ProjectICI.
// Handles the initialization and control of the ICI control system.
// Processes incoming messages, executes message commands, and
// generates a response.
//-------------------------------------------------------------------

//-------------------------------------------------------------------
#include "ici.h"
#include "error_defines.h"
#include "atd.h"
#include "digital_io.h"
#include "serial_io.h"
#include "tcp_server.h"
#include <string.h>
//-------------------------------------------------------------------

//-------------------------------------------------------------------
```

```
//ICI message state and information.
ici_message_info message_info;
//ICI message.
ici_message message;
//------------------------------------------------------------------

//------------------------------------------------------------------
// Function: ici_init - ICI control system initialization. Clears ICI
//           message state and information.
// Parameters: void
// Return Value: void
//------------------------------------------------------------------
void ici_init(void)
{
  //Clear ICI message state and information.
  ici_clearmessage();
}
//------------------------------------------------------------------

//------------------------------------------------------------------
// Function: ici_run - Move through ICI control system.
// Parameters: void
// Return Value: void
//------------------------------------------------------------------
void ici_run(void)
{
  UINT8 retval = OK;
  UINT16 NumData;
  UINT8 i;

  //Check if message is new.
  if (message_info.State == ICI_MESSAGE_NEW)
  {
    //Execute message command.
    switch (message.Command)
    {
      case ICI_COMMAND_GET_ATD_IN:
        //Get ATD input channel.
        if (message.NumData == 0) //Invalid data.
        {
          retval = ERROR_DATA;
        }
        else
        {
          //Zero number of bytes in message response.
          message.NumData = 0;
          //First byte in the received message data is the index of the
          //ATD input channel.
          //First byte is reserved for the ACK/NAK.
          //Get the ATD input channel and store result in the message
          //data buffer (skip ACK/NAK byte).
          retval = get_atd_in(message.Data[0],
                              (UINT16*)(message.Data+1));
          if (retval == OK)
          {
            //Set number of bytes in the message response to the size
            //of the ATD input channel result (2 bytes).
            message.NumData = 2;
            //Convert the result to little endian.
            swap_byte_order(message.Data+1, message.NumData);
          }
        }
        //Finished processing message.
        message_info.State = ICI_MESSAGE_FINISHED;
        break;
      case ICI_COMMAND_GET_DIGITAL_IN:
        //Get digital input channel.
        if (message.NumData == 0) //Invalid data.
        {
          retval = ERROR_DATA;
        }
        else
        {
          //Zero number of bytes in message response.
          message.NumData = 0;
          //First byte in the received message data is the index of the
          //digital input channel.
          //First byte is reserved for the ACK/NAK.
          //Get the digital input channel and store the result in the
          //message data buffer (skip ACK/NAK byte).
          retval = get_digital_in(message.Data[0], message.Data+1);
          if (retval == OK)
          {
            //Set number of bytes in the message response to the size
            //of the digital input channel result (1 byte).
            message.NumData = 1;
          }
        }
        //Finished processing message.
        message_info.State = ICI_MESSAGE_FINISHED;
        break;
      case ICI_COMMAND_GET_DIGITAL_OUT:
        //Get digital output channel.
        if (message.NumData == 0) //Invalid data.
        {
          retval = ERROR_DATA;
        }
        else
        {
          //Zero number of bytes in message response.
          message.NumData = 0;
          //First byte in the received message data is the index of the
          //digital output channel.
          //First byte is reserved for the ACK/NAK.
          //Get the digital output channel and store the result in the
          //message data buffer (skip ACK/NAK byte).
          retval = get_digital_out(message.Data[0], message.Data+1);
          if (retval == OK)
```

```
      {
        //Set number of bytes in the message response to the size
        //of the digital output channel result (1 byte).
        message.NumData = 1;
      }
    }
    //Finished processing message.
    message_info.State = ICI_MESSAGE_FINISHED;
    break;
  case ICI_COMMAND_SET_DIGITAL_OUT:
    //Set digital output channel.
    if (message.NumData < 2)
    {
      retval = ERROR_DATA;
    }
    else
    {
      //Zero number of bytes in message response.
      message.NumData = 0;
      //First byte in the received message data is the index of the
      //digital output channel.
      //Second byte is digital output channel value.
      //Set the digital output channel.
      retval = set_digital_out(message.Data[0], message.Data[1]);
    }
    //Finished processing message.
    message_info.State = ICI_MESSAGE_FINISHED;
    break;
  case ICI_COMMAND_PASS_SCI0:
    //Pass data to SCI0 and wait for response.
    if (message.NumData == 0) //Invalid data.
    {
      retval = ERROR_DATA;
    }
    else
    {
      //First byte in received message data is the expected value
      //of the last byte in the serial response to this message.
      //Pass the message data over SCI0 (skip terminator byte).
      retval = send_serial(0, message.Data+1, message.NumData-1);
      //Zero number of bytes in message response.
      message.NumData = 0;
      if (retval == OK)
      {
        //Wait for the serial response to this message.
        message_info.State = ICI_MESSAGE_WAITING;
      }
      else
      {
        //Finished processing message.
        message_info.State = ICI_MESSAGE_FINISHED;
      }
    }
    break;
```

```
case ICI_COMMAND_PASS_SCI1:
  //Pass data to SCI1 and wait for response.
  if (message.NumData == 0) //Invalid data.
  {
    retval = ERROR_DATA;
  }
  else
  {
    //First byte in received message data is the expected value
    //of the last byte in the serial response to this message.
    //Pass the message data over SCI1 (skip terminator byte).
    retval = send_serial(1, message.Data+1, message.NumData-1);
    //Zero number of bytes in message response.
    message.NumData = 0;
    if (retval == OK)
    {
      //Wait for the serial response to this message.
      message_info.State = ICI_MESSAGE_WAITING;
    }
    else
    {
      //Finished processing message.
      message_info.State = ICI_MESSAGE_FINISHED;
    }
  }
  break;
case ICI_COMMAND_GET_HATCH_CONTROL:
  //Get hatch control (START, STOP).
  //Zero number of bytes in message response.
  message.NumData = 0;
  //First byte in message response is reserved for the ACK/NAK.
  //Get the hatch control and store result in the message data
  //buffer (skip ACK/NAK byte).
  retval = get_hatch_control(message.Data+1);
  if (retval == OK)
  {
    //Set the number of bytes in the message response to the size
    //of the hatch control result (1 byte).
    message.NumData = 1;
  }
  //Finished processing message.
  message_info.State = ICI_MESSAGE_FINISHED;
  break;
case ICI_COMMAND_SET_HATCH_CONTROL:
  //Set hatch control (START, STOP).
  if (message.NumData == 0) //Invalid data.
  {
    retval = ERROR_DATA;
  }
  else
  {
    //Zero number of bytes in message response.
    message.NumData = 0;
    //First byte in the received message data is the hatch
```

```
      //control value.
      //Set the hatch control value.
      retval = set_hatch_control(message.Data[0]);
    }
    //Finished processing message.
    message_info.State = ICI_MESSAGE_FINISHED;
    break;
  case ICI_COMMAND_GET_HATCH_STATE:
    //Get hatch state (NONE, OPEN, CLOSED, INVALID).
    //Zero number of bytes in message response.
    message.NumData = 0;
    //First byte is reserved for the ACK/NAK.
    //Get the hatch state and store the result in the message data
    //buffer (skip ACK/NAK byte).
    retval = get_hatch_state(message.Data+1);
    if (retval == OK)
    {
      //Set the number of bytes in the message response to the size
      //of the hatch state result (1 byte).
      message.NumData = 1;
    }
    //Finished processing message.
    message_info.State = ICI_MESSAGE_FINISHED;
    break;
  case ICI_COMMAND_SET_HATCH_STATE:
    //Set hatch state (OPEN, CLOSED).
    if (message.NumData == 0) //Invalid data.
    {
      retval = ERROR_DATA;
    }
    else
    {
      //Zero number of bytes in message response.
      message.NumData = 0;
      //First byte in the received message data is the hatch
      //state value.
      //Set the hatch state value.
      retval = set_hatch_state(message.Data[0]);
    }
    //Wait for the hatch to reach the new state.
    message_info.State = ICI_MESSAGE_WAITING;
    break;
  case ICI_COMMAND_GET_HATCH_ERROR:
    //Get hatch control error (NONE, OPEN, CLOSED, INVALID) and
    //clear error.
    //Zero number of bytes in message response.
    message.NumData = 0;
    //First byte is reserved for the ACK/NAK.
    //Get the hatch error and store the result in the message data
    //buffer (skip ACK/NAK byte).
    retval = get_hatch_error(message.Data+1, 1);
    if (retval == OK)
    {
      //Set the number of bytes in the message response to the size
      //of the hatch error result (1 byte).
      message.NumData = 1;
    }
    //Finished processing message.
    message_info.State = ICI_MESSAGE_FINISHED;
    break;
  case ICI_COMMAND_GET_ATD_IN_ALL:
    //Get all ATD input channels.
    //Zero number of bytes in message response.
    message.NumData = 0;
    //First byte is reserved for the ACK/NAK.
    //Get the ATD input channels and store result in the message
    //data buffer (skip ACK/NAK byte).
    for (i = 0; i < 8 && retval == OK; i++)
    {
      retval = get_atd_in(i, (UINT16*)(message.Data+1+i*2));
      if (retval == OK)
      {
        //Set number of bytes in the message response to the size
        //of the ATD input channel result (2 bytes).
        message.NumData += 2;
        //Convert the result to little endian.
        swap_byte_order(message.Data+1+i*2, 2);
      }
    }
    //Finished processing message.
    message_info.State = ICI_MESSAGE_FINISHED;
    break;
  case ICI_COMMAND_GET_SCI0:
    //Read data from the SCI0 receive buffer.
    //Zero number of bytes in message response.
    message.NumData = 0;
    //First byte is reserved for the ACK/NAK.
    //Get the serial response and store the result in the message
    //data buffer (skip ACK/NAK byte).
    retval = receive_serial_all(0, message.Data+1,
                                &message.NumData,
                                ICI_DATA_SIZE-1);
    //Finished processing message.
    message_info.State = ICI_MESSAGE_FINISHED;
    break;
  case ICI_COMMAND_SET_SCI0:
    //Write data to the SCI0 transmit buffer.
    retval = send_serial(0, message.Data, message.NumData);
    //Zero number of bytes in message response.
    message.NumData = 0;
    //Finished processing message.
    message_info.State = ICI_MESSAGE_FINISHED;
    break;
  case ICI_COMMAND_GET_SCI1:
    //Read data from the SCI1 receive buffer.
    //Zero number of bytes in message response.
    message.NumData = 0;
    //First byte is reserved for the ACK/NAK.
    //Get the serial response and store the result in the message
    //data buffer (skip ACK/NAK byte).
    retval = receive_serial_all(1, message.Data+1,
                                &message.NumData,
                                ICI_DATA_SIZE-1);
    //Finished processing message.
    message_info.State = ICI_MESSAGE_FINISHED;
    break;
  case ICI_COMMAND_SET_SCI1:
    //Write data to the SCI1 transmit buffer.
    retval = send_serial(1, message.Data, message.NumData);
    //Zero number of bytes in message response.
    message.NumData = 0;
    //Finished processing message.
    message_info.State = ICI_MESSAGE_FINISHED;
    break;
  case ICI_COMMAND_GET_DEVICE_INFO:
    //Get device information.
    message.NumData = 0;
    //First byte is reserved for the ACK/NAK.
    //Get the device information and store result in the message
    //data buffer (skip ACK/NAK byte).
    retval = get_device_info((ici_device_info*)(message.Data+1));
    if (retval == OK)
    {
      //Set the number of bytes in the message response to the size
      //of the ICI device information structure (64 bytes).
      message.NumData = sizeof(ici_device_info);
      //Convert the result to little endian.
      retval =
        swap_device_info((ici_device_info*)(message.Data+1));
    }
    //Finished processing message.
    message_info.State = ICI_MESSAGE_FINISHED;
    break;
  case ICI_COMMAND_GET_RAIN_INDEX:
    //Get the rain sensor ATD index.
    //Zero number of bytes in message response.
    message.NumData = 0;
    //First byte is reserved for the ACK/NAK.
    //Get the rain sensor index and store the result in the message
    //data buffer (skip ACK/NAK byte).
    retval = get_rain_index(message.Data+1);
    if (retval == OK)
    {
      //Set the number of bytes in the message response to the size
      //of the rain sensor index (1 byte).
      message.NumData = 1;
    }
    //Finished processing message.
    message_info.State = ICI_MESSAGE_FINISHED;
    break;
  case ICI_COMMAND_SET_RAIN_INDEX:
    //Set the rain sensor ATD index.
```

```
    if (message.NumData == 0) //Invalid data.
    {
      retval = ERROR_DATA;
    }
    else
    {
      //Zero number of bytes in message response.
      message.NumData = 0;
      //First byte in the received message data is the rain sensor
      //index.
      //Set the sensor index.
      retval = set_rain_index(message.Data[0]);
    }
    //Finished processing message.
    message_info.State = ICI_MESSAGE_FINISHED;
    break;
  case ICI_COMMAND_GET_RAIN_WET:
    //Get the rain sensor wet threshold.
    //Zero number of bytes in message response.
    message.NumData = 0;
    //First byte is reserved for the ACK/NAK.
    //Get the sensor wet threshold and store result in the message
    //data buffer (skip ACK/NAK byte).
    retval = get_rain_wet((UINT16*)(message.Data+1));
    if (retval == OK)
    {
      //Set the number of bytes in the message response to the size
      //of the sensor wet threshold (2 bytes).
      message.NumData = 2;
      //Convert the result to little endian.
      swap_byte_order(message.Data+1, message.NumData);
    }
    //Finished processing message.
    message_info.State = ICI_MESSAGE_FINISHED;
    break;
  case ICI_COMMAND_SET_RAIN_WET:
    //Set the rain sensor wet threshold.
    if (message.NumData < 2) //Invalid data.
    {
      retval = ERROR_DATA;
    }
    else
    {
      //Convert the threshold to big endian.
      swap_byte_order(message.Data, message.NumData);
      //Zero number of bytes in message response.
      message.NumData = 0;
      //First two bytes in the received message data are the rain
      //sensor wet threshold (little endian).
      //Set the sensor wet threshold.
      retval = set_rain_wet(*(UINT16*)message.Data);
    }
    //Finished processing message.
    message_info.State = ICI_MESSAGE_FINISHED;
```

```
      break;
    case ICI_COMMAND_GET_RAIN_DRY:
      //Get the rain sensor dry threshold.
      //Zero number of bytes in message response.
      message.NumData = 0;
      //First byte is reserved for the ACK/NAK.
      //Get the sensor dry threshold and store result in the message
      //data buffer (skip ACK/NAK byte).
      retval = get_rain_dry((UINT16*)(message.Data+1));
      if (retval == OK)
      {
        //Set the number of bytes in the message response to the size
        //of the sensor dry threshold (2 bytes).
        message.NumData = 2;
        //Convert the result to little endian.
        swap_byte_order(message.Data+1, message.NumData);
      }
      //Finished processing message.
      message_info.State = ICI_MESSAGE_FINISHED;
      break;
    case ICI_COMMAND_SET_RAIN_DRY:
      //Set the rain sensor dry threshold.
      if (message.NumData < 2) //Invalid data.
      {
        retval = ERROR_DATA;
      }
      else
      {
        //Convert the threshold to big endian.
        swap_byte_order(message.Data, message.NumData);
        //Zero number of bytes in message response.
        message.NumData = 0;
        //First two bytes in the received message data are the rain
        //sensor dry threshold (little endian).
        //Set the sensor dry threshold.
        retval = set_rain_dry(*(UINT16*)message.Data);
      }
      //Finished processing message.
      message_info.State = ICI_MESSAGE_FINISHED;
      break;
    default:
      //Invalid command.
      retval = ERROR_COMMAND;
      //Zero number of bytes in message response.
      message.NumData = 0;
      //Finished processing message.
      message_info.State = ICI_MESSAGE_FINISHED;
      break;
  }
}

//Check if message is waiting.
if (message_info.State == ICI_MESSAGE_WAITING)
{
```

```
//Continue processing command.
switch (message.Command)
{
  case ICI_COMMAND_PASS_SCI0:
    //Continue processing the pass to SCI0 command.
    //Check for a transmit error.
    retval = send_serial_check(0);
    if (retval == OK)
    {
      //First byte is reserved for the ACK/NAK.
      //Get the serial response and store the result in the message
      //data buffer (skip ACK/NAK byte).
      retval = receive_serial(0, message.Data+1, &message.NumData,
                              ICI_DATA_SIZE-1);
    }
    if (retval != OK)
    {
      //Finished processing message.
      message_info.State = ICI_MESSAGE_FINISHED;
    }
    else
    {
      //Check if the serial terminator was found.
      //First byte in received message data is the expected value
      //of the last byte in the serial response to this message.
      if (message.NumData > 0 &&
          (message.Data[message.NumData] == message.Data[0] ||
           message.Data[message.NumData] == '?'))
      {
        //Finished processing message.
        message_info.State = ICI_MESSAGE_FINISHED;
      }
    }
    break;
  case ICI_COMMAND_PASS_SCI1:
    //Continue processing the pass to SCI1 command.
    //Check for a transmit error.
    retval = send_serial_check(1);
    if (retval == OK)
    {
      //First byte is reserved for the ACK/NAK.
      //Get the serial response and store the result in the message
      //data buffer (skip ACK/NAK byte).
      retval = receive_serial(1, message.Data+1, &message.NumData,
                              ICI_DATA_SIZE-1);
    }
    if (retval != OK)
    {
      //Finished processing message.
      message_info.State = ICI_MESSAGE_FINISHED;
    }
    else
    {
      //Check if the serial terminator was found.
```

```
        //First byte in received message data is the expected value
        //of the last byte in the serial response to this message.
        if (message.NumData > 0 &&
            (message.Data[message.NumData] == message.Data[0] ||
             message.Data[message.NumData] == '?'))
        {
          //Finished processing message.
          message_info.State = ICI_MESSAGE_FINISHED;
        }
      }
      break;
    case ICI_COMMAND_SET_HATCH_STATE:
      //Continue processing the set hatch state command.
      //Check for a hatch control system error.
      retval = get_hatch_error(message.Data+1, 0);
      if (retval == OK)
      {
        //First byte is reserved for the ACK/NAK.
        //Store the hatch control system error in the message data
        //buffer (skip ACK/NAK byte).
        (void)memcpy(&retval, message.Data+1, sizeof(UINT8));
      }
      if (retval == OK)
      {
        //First byte in the received message data is the hatch
        //state value.
        //Get the current hatch state and store the result in the
        //message data buffer (skip hatch state byte).
        retval = get_hatch_state(message.Data+1);
      }
      if (retval != OK)
      {
        //Finished processing message.
        message_info.State = ICI_MESSAGE_FINISHED;
      }
      else
      {
        //Check if the hatch state has reached the dedired value.
        //First byte in the received message data is the hatch
        //state value.
        //Second byte in the message data is the current hatch
        //state value.
        if (message.Data[0] == message.Data[1])
        {
          //Finished processing message.
          message_info.State = ICI_MESSAGE_FINISHED;
        }
      }
      break;
  }
}

//Check if message is finished.
if (message_info.State == ICI_MESSAGE_FINISHED)
```

```
  {
    //Fill in the ACK/NAK byte in message response.
    message.Data[0] = retval;
    //Increment the number of bytes to include the size of the
    //ACK/NAK byte.
    message.NumData++;
    //Calculate the total size of the message response (header and
data).
    NumData = message.NumData + sizeof(ici_message_header);
    //Convert the command and numdata to little endian.
    swap_byte_order((UINT8*)&message.Command, sizeof(message.Command));
    swap_byte_order((UINT8*)&message.NumData, sizeof(message.NumData));
    //Send the message response.
    switch (message_info.From)
    {
      case ICI_TYPE_TCP: //Send response over TCP.
        tcp_server_send(message_info.Index, (UINT8*)&message, NumData);
        break;
      case ICI_TYPE_SERIAL: //Send response over serial.
        (void)send_serial(message_info.Index, (UINT8*)&message,
                          NumData);
        break;
    }
    //Clear the message.
    ici_clearmessage();
  }
}
//------------------------------------------------------------------

//------------------------------------------------------------------
// Function: ici_clearmessage - Clear the ICI message.
// Parameters: void
// Return Value: void
//------------------------------------------------------------------
void ici_clearmessage(void)
{
  (void)memset(&message_info, 0, sizeof(ici_message_info));
  (void)memset(&message, 0, sizeof(ici_message));
}
//------------------------------------------------------------------

//------------------------------------------------------------------
// Function: ici_process_message - Process the ICI message. Parse the
//           incoming Data into command, numdata, and data.
// Parameters: From - Message interface (TCP, serial, ...).
//             Index - Index of message interface.
//             Data - Raw message.
//             NumData - Number of bytes in the raw message.
// Return Value: Error code. 0 == success.
//------------------------------------------------------------------
UINT8 ici_process_message(UINT8 From, UINT8 Index, UINT8* Data,
                          UINT16 NumData)
{
  UINT8 retval = OK;
```

```
ici_message_header busy_header;
UINT8 busy_response[sizeof(ici_message_header)+1];

if (!Data) //Invalid pointer.
{
  retval = ERROR_POINTER;
}
else if (NumData < sizeof(ici_message_header) || //Not enough data.
         NumData > sizeof(ici_message)) //NumData exceeds capacity.
{
  retval = ERROR_DATA;
}
else if (message_info.State != ICI_MESSAGE_NONE)
{
  //Message in progress.
  //Copy raw message command into busy response header.
  (void)memcpy(&busy_header, Data, sizeof(busy_header.Command));
  //Set number of bytes in the busy response to the size of the
  //ACK/NAK byte.
  busy_header.NumData = 1;
  //Convert the number of bytes to little endian.
  swap_byte_order((UINT8*)&busy_header.NumData,
                  sizeof(busy_header.NumData));
  //Build the busy response.
  //Copy the comand and number of bytes into the busy response.
  (void)memcpy(&busy_response, &busy_header,
               sizeof(ici_message_header));
  //Fill in the ACK/NAK byte in message response.
  busy_response[sizeof(ici_message_header)] = ERROR_BUSY;
  //Send response.
  switch (From)
  {
    case ICI_TYPE_TCP: //Send response over TCP.
      tcp_server_send(Index, busy_response,
                      sizeof(ici_message_header)+1);
      break;
    case ICI_TYPE_SERIAL: //Send response over serial.
      (void)send_serial(Index,busy_response,
                        sizeof(ici_message_header)+1);
      break;
  }
}
else
{
  //Load the raw message into the ICI message.
  //Set the message state to NEW.
  message_info.State = ICI_MESSAGE_NEW;
  //Set the message interface.
  message_info.From = From;
  message_info.Index = Index;
  //Set the index of the message interface.
  (void)memcpy(&message, Data, NumData);
  //Convert the command and numdata to big endian.
  swap_byte_order((UINT8*)&message.Command, sizeof(message.Command));
```

```
    swap_byte_order((UINT8*)&message.NumData, sizeof(message.NumData));
  }

  return retval;
}
//-------------------------------------------------------------------

//-------------------------------------------------------------------
// Function: swap_byte_order - Fill the ICI device information
//           structure.
// Parameters: device_info - Pointer to ici_device_info structure.
//             Size - Number of bytes in the array.
// Return Value: Error code. 0 == success.
//-------------------------------------------------------------------
UINT8 get_device_info(ici_device_info* device_info)
{
  UINT8 retval = OK;

  if (!device_info) //Invalid pointer.
  {
    retval = ERROR_POINTER;
  }
  else
  {
    device_info->FirmwareVersion = ICI_DEVICE_FIRMWARE;
    (void)get_atd_resolution((UINT16*)&device_info->AtdResolution);
    //NOTE: Return timeouts in [ms]. Assume RTI is set to 10[ms].
    device_info->HatchOpenTimeout = HATCH_TIMEOUT*10;
    device_info->SerialTimeout = SERIAL_TIMEOUT*10;
  }

  return retval;
}
//-------------------------------------------------------------------

//-------------------------------------------------------------------
// Function: swap_device_info - Swap the order of the fields in the
//           ICI device information structure.
// Parameters: device_info - Pointer to ici_device_info structure.
//             Size - Number of bytes in the array.
// Return Value: void
//-------------------------------------------------------------------
UINT8 swap_device_info(ici_device_info* device_info)
{
  UINT8 retval = OK;

  if (!device_info) //Invalid pointer.
  {
    retval = ERROR_POINTER;
  }
  else
  {
    swap_byte_order((BYTE*)&device_info->FirmwareVersion,
                    sizeof(device_info->FirmwareVersion));
    swap_byte_order((BYTE*)&device_info->AtdResolution,
                    sizeof(device_info->AtdResolution));
    swap_byte_order((BYTE*)&device_info->HatchOpenTimeout,
                    sizeof(device_info->HatchOpenTimeout));
    swap_byte_order((BYTE*)&device_info->SerialTimeout,
                    sizeof(device_info->SerialTimeout));
  }

  return retval;
}
//------------------------------------------------------------------

//------------------------------------------------------------------
// Function: swap_byte_order - Swap the order of the bytes in the
//           array.
// Parameters: Address - Pointer to array of bytes.
//             Size - Number of bytes in the array.
// Return Value: void
//------------------------------------------------------------------
void swap_byte_order(UINT8* Address, int Size)
{
  UINT8 temp;

  //Set offsets to start and end of array.
  int StartOffset = 0, EndOffset = Size-1;
  while(StartOffset < EndOffset)
  {
    //Get a copy of the start byte.
    temp = Address[StartOffset];
    //Set the start byte to the end byte.
    Address[StartOffset] = Address[EndOffset];
    //Set the end byte to the start byte.
    Address[EndOffset] = temp;
    //Move start offset forward one byte.
    StartOffset++;
    //Move end offset back one byte.
    EndOffset--;
  }
}
//------------------------------------------------------------------

//------------------------------------------------------------------
// Main.c - Main file for ProjectICI.
// ProjectICI is a simple control system that supports embedded
// ethernet for communicating with the ICI software interface. The
// software interface is run from a personal computer, and the software
// communicates with the microcontroller through a TCP/IP connection.
// The control system supports 8 analog input channels, 8 digital input
// channels, and 8 digital output channels. It also supports 2 serial
// ports, and will pass information from ethernet to serial, and from
// serial to ethernet. The control system uses 2 additional digital
// inputs for reading the hatch position (open and closed), and 1
// additonal digital output for controlling the hatch (moving and not
// moving). The control system will automatically close the hatch
// after it has been open for 1 minute. Analog channel 0 is assumed to
// be a rain sensor, and the control system will close the hatch when
// rain is detected.
//-------------------------------------------------------------------

//-------------------------------------------------------------------
#include "debug.h"
#include "datatypes.h"
#include "MC9S12NE64.h"
//OpenTCP files
#include "timers.h"
#include "system.h"
#include "ethernet.h"
#include "arp.h"
#include "ip.h"
#include "tcp_ip.h"
//NE64 files
#include "address.h"
#include "ne64driver.h"
#include "ne64api.h"
#include "mBuf.h"
#include "ne64config.h"
//ProjectICI files
#include "tcp_server.h"
#include "serial_io.h"
#include "atd.h"
#include "digital_io.h"
#include "rti.h"
#include "ici.h"
//-------------------------------------------------------------------

//-------------------------------------------------------------------
//Network interface
struct netif localmachine;
//Determines if flow control packets are sent in full duplex.
extern tU16 gotxflowc;
//Determines if link is active.
extern tU08 gotlink;

#if USE_SWLED //use software to drive EPHY status indicators on port L.
  tU16 LEDcounter=0;
#endif
//-------------------------------------------------------------------

//-------------------------------------------------------------------
// Function: main - mc9s12ne64 kernel. Initialize the ICI system. Watch
//            for Ethernet frames, process the frames, and then update
//           the TCP connection. Update the tcp server and the ICI
//           control system.
// Parameters: void
// Return Value: void
//-------------------------------------------------------------------
void main(void)
{
```

```
INT16 len; //temporary variable for length of ethenet frame

//System clock initialization.
//The oscillator output clock signal (OSCCLK) is 25[MHz]. Set the
//phase-locked loop clock (PLLCLK) to twice the OSCCLK, or 50[MHz].
//This results in a bus clock of PLLCLK/2, or 25[MHz].
CLKSEL=0;          //Reset the CRG clock select register.
CLKSEL_PLLSEL = 0; //De-select the PLLCLK (select the OSCCLK).
PLLCTL_PLLON = 0;  //Turn off the PLL circuitry.
SYNR = 0;          //Set PLL the multiplication factor to 0.
REFDV = 0;         //Set PLL the reference divider to 0.
PLLCTL = 192;      //Turn on the clock monitor and the PLL circuitry.
PLLCTL_PLLON = 1;  //Turn on the PLL circuitry.
while(!CRGFLG_LOCK); //Wait for PLL to reach the desired frequency.
CLKSEL_PLLSEL = 1; //Select the PLLCLK.

INTCR_IRQEN = 0;   //Disable the IRQ interrupt (enabled after reset).

//Set the network information.
localmachine.localip = *((UINT32 *)ip_address); //IP address.
localmachine.defgw   = *((UINT32 *)ip_gateway); //Default gateway.
localmachine.netmask = *((UINT32 *)ip_netmask); //Subnet mask.
localmachine.localHW[0] = hard_addr[0]; //Ethernet (MAC) address.
localmachine.localHW[1] = hard_addr[1];
localmachine.localHW[2] = hard_addr[2];
localmachine.localHW[3] = hard_addr[3];
localmachine.localHW[4] = hard_addr[4];
localmachine.localHW[5] = hard_addr[5];

//Initialize system services.
timer_pool_init(); //OpenTCP timers.
atd_init();        //Atd converter.
digital_io_init(); //Digital I/O
serial_io_init();  //Serial I/O
//Initialize all buffers.
mBufInit ();
//Initialize all network layers.
EtherInit();

//Enable interrupts
__asm CLI;

//Initialize ARP.
arp_init();
//Initialize TCP.
(void)tcp_init();

//Initialize TCP server.
(void)tcp_server_init();
//Initialize ICI control system.
ici_init();
//Inititlize RTI (RTI must be configured to 10[ms] for OpenTCP).
RTI_Init();
//Enable RTI.
```

```
  RTI_Enable ();

  //Main loop
  for (;;)
  {
#if USE_SWLED //use software to drive EPHY status indicators on port L.
    UseSWLedRun();
#endif
    if (gotlink)
    {
      //Try to receive ethernet frame
      if( NETWORK_CHECK_IF_RECEIVED() == TRUE )
      {
        switch( received_frame.protocol)
        {
          case PROTOCOL_ARP: //ARP frame.
            process_arp (&received_frame);
            break;
          case PROTOCOL_IP:    //IP frame.
            len = process_ip_in(&received_frame);
            if(len < 0)
              break;
            switch (received_ip_packet.protocol)
            {
              case IP_ICMP:  //ICMP/IP frame.
                process_icmp_in (&received_ip_packet, len);
                break;
              case IP_UDP:   //UDP/IP frame.
                process_udp_in (&received_ip_packet,len);
                break;
              case IP_TCP:   //TCP/IP frame.
                process_tcp_in (&received_ip_packet, len);
                break;
              default:
                break;
            }
            break;
          default:
            break;
        }
        //Discard received frame.
        NETWORK_RECEIVE_END();
      }
      //Manage arp cache tables.
      arp_manage();
      //Manage opened TCP connections (retransmissions, timeouts, ...).
      tcp_poll();
      //Manage TCP server.
      tcp_server_run();
      //Manage ICI control system.
      ici_run();
    }
    else
    {
      //NO LINK
    }
  }
}
//-------------------------------------------------------------------

//-------------------------------------------------------------------
// rti.h - RTI header file for ProjectICI.
// Handles the initialization and control of the Real-Time interrupt.
// The interrupt is set to ~10[ms]. This setting (10[ms]) is required
// for the OpenTCP implementation. The digital hatch control and serial
// timers also assume the RTI is set to 10[ms]. The RTI updates the
// timers for OpenTCP, the hatch control system, and the serial I/O.
//-------------------------------------------------------------------

//-------------------------------------------------------------------
#ifndef INCLUDE_RTI_H
#define INCLUDE_RTI_H
//-------------------------------------------------------------------

//-------------------------------------------------------------------
//Real-Time Interrupt initialization.
void RTI_Init (void);
//Enable the Real-Time Interrupt.
void RTI_Enable (void);
//Disable the Real-Timer Interrupt.
void RTI_Disable (void);
//-------------------------------------------------------------------

//-------------------------------------------------------------------
#endif
//-------------------------------------------------------------------

//-------------------------------------------------------------------
// rti.c - RTI source file for ProjectICI.
// Handles the initialization and control of the Real-Time interrupt.
// The interrupt is set to ~10[ms]. This setting (10[ms]) is required
// for the OpenTCP implementation. The digital hatch control and serial
// timers also assume the RTI is set to 10[ms]. The RTI updates the
// timers for OpenTCP, the hatch control system, and the serial I/O.
//-------------------------------------------------------------------

//-------------------------------------------------------------------
#include "timers.h"
#include "digital_io.h"
#include "serial_io.h"
#include "MC9S12NE64.h"
//-------------------------------------------------------------------

//-------------------------------------------------------------------
// Function: rti_init - Real-Time Interrupt initialization. Sets the
//           interrupt to 10[ms] and disables the RTI.
// Parameters: void
// Return Value: void
```

```
//-----------------------------------------------------------------
void RTI_Init (void)
{
  //Disable the RTI.
  //CRG Interrupt Enable Register.
  //RTI Enable bit 7, 0 = RTI disabled.
  CRGINT_RTIE = 0;
  //RTI Control Register.
  //Real-time Interrupt Prescale Rate Select Bits 6-4, 111 = 2^16.
  //Real-time Interrupt Modulus Counter Bits 3-0, 0011 = 4.
  //25000000/(4*2^16) = 95.367[ticks/sec].
  //1/95.367 = 10.486[ms]
  RTICTL_RTR = 0x73;
}
//-----------------------------------------------------------------

//-----------------------------------------------------------------
// Function: rti_enable - Enable the Real-Time Interrupt.
// Parameters: void
// Return Value: void
//-----------------------------------------------------------------
void RTI_Enable (void)
{
  //Reset the interrupt request flag.
  //CRG Flags Register, 0x80 = clear the RTI flag.
  CRGFLG = CRGINT_RTIE_MASK;
  //Enable the RTI.
  //CRG Interrupt Enable Register.
  //RTI Enable bit 7, 1 = RTI enabled.
  CRGINT_RTIE = 1;
}
//-----------------------------------------------------------------

//-----------------------------------------------------------------
// Function: rti_disable - Disable the Real-Time Interrupt.
// Parameters: void
// Return Value: void
//-----------------------------------------------------------------
void RTI_Disable (void)
{
  //Disable the RTI.
  //CRG Interrupt Enable Register.
  //RTI Enable bit 7, 0 = RTI disabled.
  CRGINT_RTIE = 0;
}
//-----------------------------------------------------------------

//-----------------------------------------------------------------
// Function: realtime_interrupt - Handles the Real-Time Interrupt.
//           Updates the timers for OpenTCP, the hatch control system,
//           and the serial I/O.
// Parameters: void
// Return Value: void
//-----------------------------------------------------------------
```

```
#pragma CODE_SEG NON_BANKED
interrupt void realtime_interrupt (void)
{
  //Reset the interrupt request flag.
  //CRG Flags Register, 0x80 = clear the RTI flag.
  CRGFLG = CRGINT_RTIE_MASK;

  //update OpenTCP timers
  decrement_timers();
  //update hatch timers
  update_hatch_timers();
  //update serial timers
  update_serial_timers();
}
#pragma CODE_SEG DEFAULT
//-------------------------------------------------------------------

//-------------------------------------------------------------------
// serial_io.h - Serial I/O header file for ProjectICI.
// Handles the initialization and control of the serial input and
// output on two serial ports, SCI0 and SCI1.
//-------------------------------------------------------------------

//-------------------------------------------------------------------
#ifndef INCLUDE_SERIAL_IO_H
#define INCLUDE_SERIAL_IO_H
//-------------------------------------------------------------------

//-------------------------------------------------------------------
#include "datatypes.h"
//-------------------------------------------------------------------

//-------------------------------------------------------------------
//set serial timeout, number of real-time interrupts.
//assumes RTI set to 10[ms].
#define SERIAL_TIMEOUT 1000 //10[s].
//Serial specifications.
#define SERIAL_DATA_SIZE 128 //Size of transmit and receive buffers.
#define SERIAL_NUM_PORTS 2   //Number of serial ports.

//Serial state and information.
typedef struct
{
  UINT8 Data[SERIAL_DATA_SIZE]; //Transmit/receive buffers.
  UINT8 ReadOffset;             //Start of transmit/receive data.
  UINT8 WriteOffset;            //End of transmit/receive data.
  UINT16 Timeout;               //Total time for transmit/receive.
} serial_buffer;
//-------------------------------------------------------------------

//-------------------------------------------------------------------
//Serial I/O initialization.
void serial_io_init(void);
//Write data to the serial port.
```

```
UINT8 send_serial(UINT8 Index, UINT8* Data, UINT16 NumData);
//Check the serial transmission time.
UINT8 send_serial_check(UINT8 Index);
//Read data from the serial port.
UINT8 receive_serial(UINT8 Index, UINT8* Data, UINT16* NumData,
                     UINT16 MaxNumData);
//Read all data from the serial port.
UINT8 receive_serial_all(UINT8 Index, UINT8* Data, UINT16* NumData,
                         UINT16 MaxNumData);
//Updates timers used for serial I/O.
void update_serial_timers(void);
//-----------------------------------------------------------------

//-----------------------------------------------------------------
#endif
//-----------------------------------------------------------------

//-----------------------------------------------------------------
// serial_io.c - Serial I/O source file for ProjectICI.
// Handles the initialization and control of the serial input and
// output on two serial ports, SCI0 and SCI1.
//-----------------------------------------------------------------

//-----------------------------------------------------------------
#include "serial_io.h"
#include "MC9S12NE64.h"
#include "error_defines.h"
#include <string.h>
//-----------------------------------------------------------------

//-----------------------------------------------------------------
//Serial receive buffers.
serial_buffer volatile RxBuffer[SERIAL_NUM_PORTS];
//Serial transmit buffers.
serial_buffer volatile TxBuffer[SERIAL_NUM_PORTS];
//-----------------------------------------------------------------

//-----------------------------------------------------------------
// Function: serial_io_init - Serial I/O initialization. Sets SCI0 and
//           SCI1 to use 1 start bit, 8 data bits, and 1 stop bit, no
//           parity, and 9600 baud. Enable the receiver full interrupt.
// Parameters: void
// Return Value: void
//-----------------------------------------------------------------
void serial_io_init(void)
{
  //Clear the receive and transmit buffers.
  (void)memset((void*)RxBuffer, 0,
             sizeof(serial_buffer)*SERIAL_NUM_PORTS);
  (void)memset((void*)TxBuffer, 0,
             sizeof(serial_buffer)*SERIAL_NUM_PORTS);

  //Configure SCI0.
  //Set the data format and disable parity.
  //SCI Control Register 1
  //Data Format Mode Bit 4, 0 = 1 start, 8 data and 1 stop bit.
  SCI0CR1_M = 0;
  //Parity Enable Bit 1, 0 = no parity.
  SCI0CR1_PE = 0;
  //Enable the transmitter and receiver.
  //SCI Control Register 2.
  //Transmitter Enable Bit 3, 1 = enable transmitter.
  SCI0CR2_TE = 1;
  //Receiver Enabled Bit 2, 1 = enable receiver.
  SCI0CR2_RE = 1;
  //Set Baud rate.
  //SCI Baud Rate Registers.
  //25000000/(16*163) = 9585 ~= 9600
  SCI0BDH = 0;
  SCI0BDL = 163;
  //Enable interrupts for the receive data register full (RDRF) flag.
  //SCI Control Register 2.
  //Receiver Full Interrupt Enable Bit 5, 1 = enable RIE.
  SCI0CR2_RIE = 1;

  //Configure SCI1.
  //Set the data format and disable parity.
  //SCI Control Register 1
  //Data Format Mode Bit 4, 0 = 1 start, 8 data and 1 stop bit.
  SCI1CR1_M = 0;
  //Parity Enable Bit 1, 0 = no parity.
  SCI1CR1_PE = 0;
  //Enable the transmitter and receiver.
  //SCI Control Register 2.
  //Transmitter Enable Bit 3, 1 = enable transmitter.
  SCI1CR2_TE = 1;
  //Receiver Enabled Bit 2, 1 = enable receiver.
  SCI1CR2_RE = 1;
  //Set Baud rate.
  //SCI Baud Rate Registers.
  //25000000/(16*163) = 9585 ~= 9600
  SCI1BDH = 0;
  SCI1BDL = 163;
  //Enable interrupts for the receive data register full (RDRF) flag.
  //SCI Control Register 2.
  //Receiver Full Interrupt Enable Bit 5, 1 = enable RIE.
  SCI1CR2_RIE = 1;
}
//------------------------------------------------------------------

//------------------------------------------------------------------
// Function: update_serial_timers - Updates timers used for serial I/O.
// Parameters: void
// Return Value: void
//------------------------------------------------------------------
void update_serial_timers(void)
{
  UINT8 i;
  for (i = 0; i < SERIAL_NUM_PORTS; i++)
  {
    //Update receive timers.
    if (RxBuffer[i].Timeout > 0) RxBuffer[i].Timeout--;
    //Update transmit timers.
    if (TxBuffer[i].Timeout > 0) TxBuffer[i].Timeout--;
  }
}
//------------------------------------------------------------------

//------------------------------------------------------------------
// Function: sci0_interrupt - Handles SCI0 Interrupts.
//           Reads serial input and stores it in the receive buffer.
//           Writes serial output stored in the transmit buffer.
// Parameters: void
// Return Value: void
//------------------------------------------------------------------
#pragma CODE_SEG NON_BANKED
interrupt void sci0_interrupt(void)
{
  UINT8 Offset;
  UINT8 Data;

  if (SCI0SR1_RDRF)
  {
    //Read the data and reset RDRF.
    Data = SCI0DRL;
    //Find next position in the receive buffer (WriteOffset).
    Offset = (UINT8)((RxBuffer[0].WriteOffset+1) % SERIAL_DATA_SIZE);
    //Check if the receive buffer is full.
    if (Offset != RxBuffer[0].ReadOffset)
    {
      //Receive buffer is not full. Add the data to the receive buffer.
      RxBuffer[0].Data[RxBuffer[0].WriteOffset] = Data;
      //Move WriteOffset to the next position.
      RxBuffer[0].WriteOffset = Offset;
      //Reset the receive timer.
      RxBuffer[0].Timeout = SERIAL_TIMEOUT;
    }
  }

  if (SCI0SR1_TDRE)
  {
    //Check if serial transmit data is present.
    if (TxBuffer[0].ReadOffset != TxBuffer[0].WriteOffset)
    {
      //Write the data in the transmit buffer (ReadOffset).
      SCI0DRL = TxBuffer[0].Data[TxBuffer[0].ReadOffset];
      //Move ReadOffset to the next position.
      TxBuffer[0].ReadOffset =
        (UINT8)((TxBuffer[0].ReadOffset+1) % SERIAL_DATA_SIZE);
      //Reset the transmit timer.
      TxBuffer[0].Timeout = SERIAL_TIMEOUT;
    }
    else
    {
      //Nothing to transmit. Disable the transmit interrupt.
      SCI0CR2_SCTIE = 0;
    }
  }
}
//------------------------------------------------------------------

//------------------------------------------------------------------
// Function: sci1_interrupt - Handles SCI0 Interrupts.
//           Reads serial input and stores it in the receive buffer.
//           Writes serial output stored in the transmit buffer.
// Parameters: void
// Return Value: void
//------------------------------------------------------------------
interrupt void sci1_interrupt(void)
{
  UINT8 Offset;
  UINT8 Data;

  if (SCI1SR1_RDRF)
  {
    //Get the data and reset RDRF.
    Data = SCI1DRL;
    //Find next position in the receive buffer (WriteOffset).
    Offset = (UINT8)((RxBuffer[1].WriteOffset+1) % SERIAL_DATA_SIZE);
    //Check if the receive buffer is full.
    if (Offset != RxBuffer[1].ReadOffset)
    {
      //Receive buffer is not full. Add the data to the receive buffer.
      RxBuffer[1].Data[RxBuffer[1].WriteOffset] = Data;
      //Move WriteOffset to the next position.
      RxBuffer[1].WriteOffset = Offset;
      //Reset the receive timer.
      RxBuffer[1].Timeout = SERIAL_TIMEOUT;
    }
  }

  if (SCI1SR1_TDRE)
  {
    //Check if serial transmit data is present.
    if (TxBuffer[1].ReadOffset != TxBuffer[1].WriteOffset)
    {
      //Write the data in the transmit buffer (ReadOffset).
      SCI1DRL = TxBuffer[1].Data[TxBuffer[1].ReadOffset];
      //Move ReadOffset to the next position.
      TxBuffer[1].ReadOffset =
        (UINT8)((TxBuffer[1].ReadOffset+1) % SERIAL_DATA_SIZE);
      //Reset the transmit timer.
      TxBuffer[1].Timeout = SERIAL_TIMEOUT;
    }
    else
    {
      //Nothing to transmit. Disable the transmit interrupt.
      SCI1CR2_SCTIE = 0;
    }
  }
}
#pragma CODE_SEG DEFAULT
//------------------------------------------------------------------

//------------------------------------------------------------------
// Function: send_serial - Write data to the serial port. Load data
//           into the transmit buffer and enable transmit interrupt to
//           send the data.
// Parameters: Index - Index of SCI port (0 - 1).
//              Data - Pointer to byte array with the output values.
//              NumData - Number of bytes in the data array.
// Return Value: Error code. 0 == success.
//------------------------------------------------------------------
UINT8 send_serial(UINT8 Index, UINT8* Data, UINT16 NumData)
{
  UINT8 retval = OK;
  UINT8 Offset;
  UINT16 i;

  if (!Data) //Invalid pointer.
  {
    retval = ERROR_POINTER;
  }
  else if (!NumData) //Number of bytes is 0.
  {
    retval = ERROR_DATA;
  }
  else if (NumData > SERIAL_DATA_SIZE) //NumData exceeds capacity.
  {
    retval = ERROR_OVERFLOW;
  }
  else if (Index >= SERIAL_NUM_PORTS) //Invalid index.
  {
    retval = ERROR_INDEX;
  }
  else
  {
    //Disable all interrupts while writing to transmit/receive buffers.
    __asm SEI;
    //Clear the TxBuffer.
    TxBuffer[Index].ReadOffset = TxBuffer[Index].WriteOffset;
    //Set the transmit timeout.
    TxBuffer[Index].Timeout = SERIAL_TIMEOUT;
    //Load the TxBuffer.
    for (i = 0; i < NumData && retval == OK; i++)
    {
      //Find next position in transmit buffer (WriteOffset).
      Offset = (UINT8)((TxBuffer[Index].WriteOffset+1) %
               SERIAL_DATA_SIZE);
      if (Offset != TxBuffer[Index].ReadOffset)
      {
        //Add the data byte to the transmit buffer.
        TxBuffer[Index].Data[TxBuffer[Index].WriteOffset] = Data[i];
        //Move WriteOffset to the next position.
        TxBuffer[Index].WriteOffset = Offset;
      }
    }
    //Clear the RxBuffer.
    RxBuffer[Index].ReadOffset = RxBuffer[Index].WriteOffset;
    //Set the receive timeout.
    RxBuffer[Index].Timeout = SERIAL_TIMEOUT;
    //Enable transmitter interrupt.
    //SCI Control Register 2.
    //Transmitter Interrupt Enable Bit 7, 1 = enable TIE.
    switch (Index)
    {
      case 0:
        SCI0CR2_SCTIE = 1;
        break;
      case 1:
        SCI1CR2_SCTIE = 1;
        break;
    }
    //Enable all interrupts.
    __asm CLI;
  }

  return retval;
}
//------------------------------------------------------------------

//------------------------------------------------------------------
// Function: send_serial_check - Check the serial transmission time.
// Parameters: Index - Index of SCI port (0 - 1).
// Return Value: Error code. 0 == success.
//------------------------------------------------------------------
UINT8 send_serial_check(UINT8 Index)
{
  UINT8 retval = OK;

  if (Index >= SERIAL_NUM_PORTS) //Invalid index.
  {
    retval = ERROR_INDEX;
  }
  else
  {
    //Check if serial data is present and check the transmit time.
    if (TxBuffer[Index].ReadOffset != TxBuffer[Index].WriteOffset &&
        TxBuffer[Index].Timeout == 0)
    {
      //The transmission is not complete and the time has expired.
      retval = ERROR_TIMEOUT;
    }
  }

  return retval;
}
//-----------------------------------------------------------------

//-----------------------------------------------------------------
// Function: receive_serial - Read data from the serial port. Read one
//           byte from the receive buffer, store it in the data array,
//           and increment the number of data bytes.
// Parameters: Index - Index of SCI port (0 - 1).
//             Data - Pointer to byte array to hold the values.
//             NumData - Number of bytes in the data array.
//             MaxNumData - Maximum number of bytes in the data array.
// Return Value: Error code. 0 == success.
//-----------------------------------------------------------------
UINT8 receive_serial(UINT8 Index, UINT8* Data, UINT16* NumData,
                      UINT16 MaxNumData)
{
  UINT8 retval = OK;

  if (!Data) //Invalid pointer.
  {
    retval = ERROR_POINTER;
  }
  else if (!NumData) //Invalid pointer.
  {
    retval = ERROR_POINTER;
  }
  else if (*NumData >= MaxNumData) //Number of bytes exceeds capacity.
  {
    retval = ERROR_OVERFLOW;
  }
  else if (Index >= SERIAL_NUM_PORTS) //Invalid index.
  {
    retval = ERROR_INDEX;
  }
  else
  {
    //Check if serial receive data is present.
    if (RxBuffer[Index].ReadOffset != RxBuffer[Index].WriteOffset)
    {
      //Add a byte to the Data buffer.
      Data[*NumData] =
        RxBuffer[Index].Data[RxBuffer[Index].ReadOffset];
      //Move ReadOffset to the next position.
      RxBuffer[Index].ReadOffset =
        (UINT8)((RxBuffer[Index].ReadOffset+1) % SERIAL_DATA_SIZE);
      //Increment the number of bytes in the Data buffer.
      (*NumData)++;
    }
    //Check the receive time.
    else if (RxBuffer[Index].Timeout == 0)
    {
      //The receive time has expired.
      retval = ERROR_TIMEOUT;
    }
  }

  return retval;
}
//-------------------------------------------------------------------

//-------------------------------------------------------------------
// Function: receive_serial_all - Read data from the serial port. Read
//           all data from the receive buffer, store it in the data
//           array, and set the number of data bytes.
// Parameters: Index - Index of SCI port (0 - 1).
//             Data - Pointer to byte array to hold the values.
//             NumData - Number of bytes in the data array.
//             MaxNumData - Maximum number of bytes in the data array.
// Return Value: Error code. 0 == success.
//-------------------------------------------------------------------
UINT8 receive_serial_all(UINT8 Index, UINT8* Data, UINT16* NumData,
                         UINT16 MaxNumData)
{
  UINT8 retval = OK;

  if (!Data) //Invalid pointer.
  {
    retval = ERROR_POINTER;
  }
  else if (!NumData) //Invalid pointer.
  {
    retval = ERROR_POINTER;
  }
  else if (Index >= SERIAL_NUM_PORTS) //Invalid index.
  {
    retval = ERROR_INDEX;
  }
  else
  {
    //Loop through the receive buffer while serial data is present,
    //receive time has not expired, and the number of bytes in the
    //Data buffer.is less than the maximum number of bytes.
    while (RxBuffer[Index].ReadOffset != RxBuffer[Index].WriteOffset &&
           RxBuffer[Index].Timeout != 0 &&
           *NumData < MaxNumData)
    {
      //Add a byte to the Data buffer.
      Data[*NumData] =
        RxBuffer[Index].Data[RxBuffer[Index].ReadOffset];
      //Move ReadOffset to the next position.
      RxBuffer[Index].ReadOffset =
        (UINT8)((RxBuffer[Index].ReadOffset+1) % SERIAL_DATA_SIZE);
      //Increment the number of bytes in the Data buffer.
      (*NumData)++;
    }
    //Check the receive time.
  }

  return retval;
}
//------------------------------------------------------------------

//------------------------------------------------------------------
// tcp_server.h - TCP server header file for ProjectICI.
// Handles the initialization and control of the TCP server. Opens TCP
// server client sockets, manages the sockets, and handles events.
//------------------------------------------------------------------

//------------------------------------------------------------------
#ifndef INCLUDE_TCP_SERVER_H
#define INCLUDE_TCP_SERVER_H
//------------------------------------------------------------------

//------------------------------------------------------------------
#include "datatypes.h"
#include "system.h"
#include "tcp_ip.h"
//------------------------------------------------------------------

//------------------------------------------------------------------
//TCP server specifications.
#define NO_OF_TCP_SESSIONS 1    //Number of TCP servers.
#define TCP_SERVER_PORT    1024 //TCP server port.
#define TCP_DATA_SIZE      132  //Size of TCP transmit/receive buffer.
                                //matches size of ici_message.

//TCP server states.
#define TCPS_STATE_FREE     0 //Free.
#define TCPS_STATE_RESERVED 1 //Reserved (connect requested).
#define TCPS_STATE_ACTIVE   2 //Active (connected).

//TCP server state and information.
typedef struct
{
  UINT8 State;                //Server state.
  INT8 OwnerSocket;           //TCP socket.
  UINT8 Data[TCP_DATA_SIZE]; //Transmit/receive buffer.
  UINT16 NumData;            //Number of bytes in the data buffer.
  UINT16 Offset;             //Number of transmitted bytes.
  UINT16 Unacked;            //Number of unacknowledged bytes
} tcps_server_state;
//------------------------------------------------------------------

//------------------------------------------------------------------
//TCP server initialization.
INT8 tcp_server_init(void);
//Move through TCP server control system.
void tcp_server_run(void);
//Process TCP server events.
```

```
INT32 tcp_server_eventlistener(INT8 cbhandle, UINT8 event, UINT32 par1,
UINT32 par2);
//Clear the TCP server.
void tcp_server_clearsession(UINT8 ses);
//Delete the TCP server.
void tcp_server_deletesession(UINT8 ses);
//Find the TCP server.
INT16 tcp_server_searchsession(UINT8 soch);
//Reserver the TCP server.
INT16 tcp_server_bindsession(UINT8 soch);
//Activate the TCP server.
void tcp_server_activatesession(UINT8 ses);
//Write data to the TCP socket.
void tcp_server_send(UINT8 ses, UINT8* Data, UINT16 NumData);
//-------------------------------------------------------------------

//-------------------------------------------------------------------
#endif;
//-------------------------------------------------------------------

//-------------------------------------------------------------------
// tcp_server.c - TCP server source file for ProjectICI.
// Handles the initialization and control of the TCP server. Opens TCP
// server client sockets, manages the sockets, and handles events.
//-------------------------------------------------------------------

//-------------------------------------------------------------------
#include "debug.h"
#include "datatypes.h"
#include "globalvariables.h"
#include "tcp_ip.h"
#include "tcp_server.h"
#include "ici.h"
#include <string.h>
//-------------------------------------------------------------------

//-------------------------------------------------------------------
//TCP server status (1 == enabled).
UINT8 tcp_server_enabled = 0;
//TCP server state and information.
tcps_server_state volatile tcps[NO_OF_TCP_SESSIONS];
//-------------------------------------------------------------------

//-------------------------------------------------------------------
// Function: tcp_server_init - TCP server initialization. Initializes
//           servers, opens the server sockets, and sets sockets to
//           listen.
// Parameters:   void
// Return Value: Index of last socket initialized.
//-------------------------------------------------------------------
INT8 tcp_server_init(void)
{
  UINT8 i;
  INT8 soch;
  //Initialize the TCP servers.
  for(i = 0; i < NO_OF_TCP_SESSIONS; i++)
  {
    //Clear the TCP server.
    tcps[i].State = TCPS_STATE_FREE;
    tcps[i].OwnerSocket = -1;
    (void)memset(tcps[i].Data, 0, TCP_DATA_SIZE);
    tcps[i].NumData = 0;
    tcps[i].Offset = 0;
    tcps[i].Unacked = 0;

    //Get a TCP server socket.
    soch =  tcp_getsocket(TCP_TYPE_SERVER, TCP_TOS_NORMAL,
                          TCP_DEF_TOUT,
                          tcp_server_eventlistener);

    //Check if a valid socket was found.
    if(soch < 0)
    {
      RESET_SYSTEM();
    }

    //Assign the socket to the server.
    tcps[i].OwnerSocket = soch;

    //Refresh the watchdog timer.
    kick_WD();

    //Set the TCP server to listening.
    soch = tcp_listen(tcps[i].OwnerSocket, TCP_SERVER_PORT);

    if(soch < 0)
    {
      RESET_SYSTEM();
    }
  }

  //Set the TCP server enabled flag.
  tcp_server_enabled  = 1;

  //Return the last socket initialized.
  return(i);
}
//------------------------------------------------------------------

//------------------------------------------------------------------
// Function: tcp_server_run - Move through TCP server control system.
//           Check if server has data to send and send data.
// Parameters: void
// Return Value: void
//------------------------------------------------------------------
void tcp_server_run(void)
{
```

```
UINT8 i;
INT16 len = 0;
static UINT8 ses = 0;

//Check if the TCP server is enabled.
if( tcp_server_enabled == 0)
  return;

//Process the TCP servers.
for(i = 0; i < NO_OF_TCP_SESSIONS; i++)
{
  //Refresh the watchdog timer.
  kick_WD();

  //Check if this is a valid server.
  if(ses >= NO_OF_TCP_SESSIONS)
    ses = 0;

  //If the server is not listening, set the state to listening.
  if(tcp_getstate(tcps[ses].OwnerSocket) < TCP_STATE_LISTENING)
  {
    (void)tcp_listen(tcps[ses].OwnerSocket, TCP_SERVER_PORT);
    ses++;
    continue;
  }

  //Determine if the server is active and has data to send.
  if(tcps[ses].State != TCPS_STATE_ACTIVE)
  {
    //Server is not active. Nothing to do. Move to next server.
    ses++;
    continue;
  }
  if(tcps[ses].Unacked != 0)
  {
    //Server has unacked data. Waiting for ack. Move to next server.
    ses++;
    continue;
  }
  if(tcps[ses].NumData == 0)
  {
    //Server has no data to send. Move to next server.
    ses++;
    return;
  }

  //Load the network buffer with the data.
  if (tcps[ses].NumData - tcps[ses].Offset <
      NETWORK_TX_BUFFER_SIZE – TCP_APP_OFFSET)
  {
    //The data fits in the network buffer. Copy the remaining data.
    len = tcps[ses].NumData - tcps[ses].Offset;
  }
  else
```

```
    {
      //The data does not fit in network buffer. Copy the size of the
      //network packet.
      len = NETWORK_TX_BUFFER_SIZE - TCP_APP_OFFSET;
    }
    (void)memcpy(&net_buf[TCP_APP_OFFSET],
                 tcps[ses].Data+tcps[ses].Offset, len);

    //Send the data.
    len = tcp_send(tcps[ses].OwnerSocket, &net_buf[TCP_APP_OFFSET],
                   NETWORK_TX_BUFFER_SIZE - TCP_APP_OFFSET, len);
    if(len<0)
    {
      //Error sending data. Clear the server. Move to the next server.
      tcp_server_clearsession(ses);
      ses++;
      return;
    }

    //Set the unacked data to length of sent data.
    tcps[ses].Unacked = len;
    //Move to the next server.
    ses++;
    return;
  }
}
//-------------------------------------------------------------------

//-------------------------------------------------------------------
// Function: tcp_server_eventlistener - Process TCP server events.
// Parameters: cbhandle - Handle to TCP server that generated the
//             event.
//             event - Type of TCP server event.
//             par1 - Generic paramter, depends on the event.
//             par2 - Generic paramter, depends on the event.
// Return Value: Error code - 1 == success.
//-------------------------------------------------------------------
INT32 tcp_server_eventlistener (INT8 cbhandle, UINT8 event, UINT32
par1, UINT32 par2)
{
  INT16 i = 0;
  INT16 session;
  UINT8 Data[TCP_DATA_SIZE]; //receive buffer.

  //par2 is not currently used. Suppress compiler warning.
  par2 = 0;

  //Check if the TCP server is enabled.
  if( tcp_server_enabled == 0)
    return(-1);
  //Check if the TCP server handle is valid.
  if(cbhandle < 0)
    return(-1);
  //Search for the correct TCP server.
```

```
session = tcp_server_searchsession(cbhandle);

//Process the TCP server event.
switch( event )
{
  case TCP_EVENT_CONREQ:
    //Connect request.
    //Create new server.
    session = tcp_server_bindsession(cbhandle);
    //Check if the TCP server was created.
    if(session < 0)
      return(-1);
    return(1);
  case TCP_EVENT_ABORT:
    //Server aborted.
    //Check if the TCP server if valid.
    if(session < 0)
      return(1);
    //Clear the TCP server.
    tcp_server_clearsession((UINT8)session);
    return(1);
  case TCP_EVENT_CONNECTED:
    //Server connected.
    //Check if the TCP server is valid.
    if(session < 0)
      return(-1);
    //Activate the TCP server.
    tcp_server_activatesession((UINT8)session);
    return(1);
  case TCP_EVENT_CLOSE:
    //Server closed.
    //Check if the TCP server is valid.
    if(session < 0)
      return(-1);
    //Delete the TCP server.
    tcp_server_deletesession((UINT8)session);
    return(1);
  case TCP_EVENT_ACK:
    //Server acknowledged.
    //Check if the TCP server is valid.
    if(session < 0)
      return(-1);
    //Move the send offset forward. Clear the unacked bytes.
    tcps[session].Offset += tcps[session].Unacked;
    tcps[session].Unacked = 0;
    //Check if the send is complete.
    if( tcps[session].Offset >= tcps[session].NumData)
    {
      //Finished sending the data. Clear the server.
      tcp_server_clearsession((UINT8)session);
    }
    return(1);
  case TCP_EVENT_DATA:
    //Data received.
```

```
      //Check if the TCP server is valid.
      if(session < 0)
        return(-1);
      //Check if the TCP server is active.
      if(tcps[session].State == TCPS_STATE_ACTIVE)
      {
        //Process the data.
        //par1 is the length of the data.
        if (par1 != 0)
        {
          //Read in the data.
          for (i = 0; i < par1 && i < TCP_DATA_SIZE; i++)
          {
            Data[i] = RECEIVE_NETWORK_B();
          }
          //Process the data.
          (void)ici_process_message(ICI_TYPE_TCP, (UINT8)session, Data,
                                    (UINT16)par1);
        }
        return(1);
      }
      return(1);
    case TCP_EVENT_REGENERATE:
      //Server needs to re-send data.
      //Check if the TCP server is valid.
      if(session < 0)
        return(-1);
      //Check if the TCP server is active.
      if(tcps[session].State != TCPS_STATE_ACTIVE)
        return(-1);
      //Load the network buffer with the data.
      if (tcps[session].NumData - tcps[session].Offset <
          NETWORK_TX_BUFFER_SIZE - TCP_APP_OFFSET)
      {
        //The data fits in network buffer. Copy the remaining data.
        i = tcps[session].NumData - tcps[session].Offset;
      }
      else
      {
        //The data does not fit in the network buffer. Copy size of the
        //network packet.
        i = NETWORK_TX_BUFFER_SIZE - TCP_APP_OFFSET;
      }
      (void)memcpy(&net_buf[TCP_APP_OFFSET],
                   tcps[session].Data+tcps[session].Offset, i);
      //Send the data.
      i = tcp_send(tcps[session].OwnerSocket, &net_buf[TCP_APP_OFFSET],
                   NETWORK_TX_BUFFER_SIZE - TCP_APP_OFFSET, i);
      return(i);
    default:
      return(-1);
  }
}
//-------------------------------------------------------------------
```

```
//------------------------------------------------------------------
// Function: tcp_server_clearsession - Clear the TCP server.
// Parameters: ses - Index of the TCP server.
// Return Value: void
//------------------------------------------------------------------
void tcp_server_clearsession (UINT8 ses)
{
  //Check if the TCP server is valid.
  if (ses < NO_OF_TCP_SESSIONS)
  {
    //Clear the TCP server (leave the server active).
    (void)memset(tcps[ses].Data, 0, TCP_DATA_SIZE);
    tcps[ses].NumData = 0;
    tcps[ses].Offset = 0;
    tcps[ses].Unacked = 0;
  }
}
//------------------------------------------------------------------

//------------------------------------------------------------------
// Function: tcp_server_deletesession - Delete the TCP server.
// Parameters: ses - Index of the TCP server.
// Return Value: void
//------------------------------------------------------------------
void tcp_server_deletesession (UINT8 ses)
{
  //Check if the TCP server is valid.
  if (ses < NO_OF_TCP_SESSIONS)
  {
    //Delete the TCP server (free the server).
    tcps[ses].State = TCPS_STATE_FREE;
    (void)memset(tcps[ses].Data, 0, TCP_DATA_SIZE);
    tcps[ses].NumData = 0;
    tcps[ses].Offset = 0;
    tcps[ses].Unacked = 0;
  }
}
//------------------------------------------------------------------

//------------------------------------------------------------------
// Function: tcp_server_searchsession - Find the TCP server.
// Parameters: soch - Owner of the TCP server.
// Return Value: Index of the TCP server.
//------------------------------------------------------------------
INT16 tcp_server_searchsession (UINT8 soch)
{
  UINT8 i;

  //Find the TCP server.
  for(i=0; i<NO_OF_TCP_SESSIONS; i++)
  {
    if(tcps[i].OwnerSocket == soch) //Server found.
      return(i);
  }
  return(-1);
}
//------------------------------------------------------------------

//------------------------------------------------------------------
// Function: tcp_server_bindsession - Reserve the TCP server.
// Parameters: soch - Owner of the TCP server.
// Return Value: Index of the TCP server.
//------------------------------------------------------------------
INT16 tcp_server_bindsession (UINT8 soch)
{
  UINT8 i;

  //Find the TCP server.
  for(i=0; i<NO_OF_TCP_SESSIONS; i++)
  {
    if(tcps[i].OwnerSocket == soch)
    {
      //Server found. Check if TCP server is free.
      if(tcps[i].State == TCPS_STATE_FREE)
      {
        //TCP server is free. Reserve the TCP server.
        tcps[i].State = TCPS_STATE_RESERVED;
        return(i);
      }
    }
  }
  return(-1);
}
//------------------------------------------------------------------

//------------------------------------------------------------------
// Function: tcp_server_activatesession - Activate the TCP server.
// Parameters: ses - Index of the TCP server.
// Return Value: void
//------------------------------------------------------------------
void tcp_server_activatesession (UINT8 ses)
{
  //Check if the TCP server is valid.
  if (ses < NO_OF_TCP_SESSIONS)
  {
    //Activate the TCP server.
      tcps[ses].State = TCPS_STATE_ACTIVE;
  }
}
//------------------------------------------------------------------

//------------------------------------------------------------------
// Function: tcp_server_send - Write data to the TCP socket. Load data
//           into the TCP server. The TCP server control system will
//           send the data (tcp_server_run).
// Parameters: ses - Index of the TCP server.
//             Data - Pointer to byte array with the output values.
//             NumData - Number of bytes in the data array.
// Return Value: void
//------------------------------------------------------------------
void tcp_server_send (UINT8 ses, UINT8* Data, UINT16 NumData)
{
  //Check if the TCP server and data are valid.
  if (ses < NO_OF_TCP_SESSIONS && Data && NumData)
  {
    //Limit the number of bytes to the size of the TCP server buffer.
    if (NumData > TCP_DATA_SIZE)
    {
      NumData = TCP_DATA_SIZE;
    }
    //Load the TCP server buffer.
    (void)memcpy(tcps[ses].Data, Data, NumData);
    //Set the number of bytes.
    tcps[ses].NumData = NumData;
  }
}
//------------------------------------------------------------------
```