I^2CIP: Inter-Integrated Circuit Intra-networking Protocols

Design Report for a Hardware Design Specification for a Bus-Switched Intra-Network of Hot-Swap Modules of I^2C Targets and a Software Library of Intra-Network Communications Protocols for Rapid Implementation of Plug-and-Play Embedded Systems

Jayden Lefebvre - Founder & CEO, Lead Engineer

Northumberland County, ON, Canada

Primary Contact Email: contact@peapodtech.com

Revision 1.0

PeaPod Technologies Inc.

November 2025

Contents

1 Design Abstract

The Intra-Integrated Circuit Intra-networking Protocols (I²CIP) is a hardware design specification and communications protocols suite that enables hot-swappable, plug-and-play modular components for embedded systems. By leveraging existing I²C components and enhancing them with a bus-switching architecture and dynamic communication protocols, I²CIP allows for rapid prototyping and deployment of embedded systems with minimal configuration. Through this approach, I²CIP aims to revolutionize the way embedded systems are designed and implemented, providing a flexible and scalable solution for developers and engineers.

From the I²C Specification Version 7 (2021, *NXP Semiconductors*): an 8-bit-oriented one-ended ("controller"-driven) bidirectional (read & write) serial communication over a 2-wire bus (data "SDA" & clock "SCL") for integrated circuit devices ("targets"), including (but not limited to):

- **Remote Multi-Channel Ports**: GPIO banks, internal-clock PWM drivers, Analog-to-Digital and Digital-to-Analog converters, etc.
- **System Devices**: real-time clocks, LCD screens, microcontrollers, etc.
- **Data Storage Devices**: EEPROM, SRAM, FRAM, etc.
- **Digital Sensors**: temperature, humidity, light, acceleration, pressure, etc.

It is advantageous to view a collection of devices on an I2C bus as an intra-network, analogous to the Internet, under certain conditions:

- **Dynamic Routing**: *Device reachability is subject to change.* This enables physical "plug-and-play" functionality.
- **Modularity**: *Devices can be added to the network as physical collections.* This enables lifecycle management for physical collections of devices, and informs dynamic routing.

In order to achieve an intra-network of I2C devices, we propose a suite of protocols built on top of I2C which together form the I2C Intranet Protocol, along with a compatible hardware specification.

The I²C Specification can be imagined as an incomplete analogue to the Internet's OSI Model, with the following layers defined:

SC1. **Physical Layer**

(a) **VDD & GND**: e.g. +5 VDC

(b) **SDA & SCL**: Pull-Up Bias Resistors (e.g. 10 kΩ)

SC2. **Data Link Layer**

(a) **Controller**: Bus Speed Control, Start & Stop Conditions, Multi-Controller Arbitration

(b) **Targets**: 7-bit Device Addressing, Acknowledgement ("ACK")

(c) **Packet Structure**: "Read" & "Write" Flags, 8-/16-bit Register Addressing, Byte-Stream Data

The **Network** (data routing), **Transport** (data delivery), and **Session** (transmission context) layers of the OSI model analogy are not defined by the I²C Specification. The following proposed extensions to the I²C Specification, the focus of the **I²CIP** design, are intended to fill this gap, enabling **Presentation** and **Application** layer functionality to be rapidly implemented by developers for embedded systems (e.g. control systems).

2 Design Description

The I²CIP technology is designed to facilitate seamless communication and interoperability between various I²C devices in a modular embedded system environment. It achieves this by defining a set of hardware and software standards that enable hot-swapping of components, dynamic routing of data, and efficient management of device resources.

For the purposes of effective intra-network communication across switched subnetworks, it is proposed that a "fully-qualified address" be implemented at the controller level, comprising routing information that encodes the I²C bus, switch address, and subnetwork number, alongside the target device address, for each target device.

Suppose a dedicated target device E consisting of EEPROM memory containing routing information for all devices on all subnetworks of one switch. If this EEPROM device E is granted a fixed address on a consistent subnetwork on each switch, the controller can reliably retrieve routing information for all devices on all subnetworks of any switch by querying each switch's dedicated EEPROM device.

Together, the EEPROM device E, the switch device X, and all devices on all subnetworks of the switch X comprise a **Module**.

SC3. **Network Layer**: Fully-Qualified Addressing ("FQA")

SC4. **Transport Layer**: Switch & Target Ping Prior to Target Control with Quality-of-Service 2 ("only-once" delivery) via ACK

SC5. **Session Layer**: Target Discovery & Module Configuration via Dedicated EEPROM Target

2.1 Hardware Specification

- **Electrical Isolator**: Prevents ground loops and protects sensitive components (**ISO1540**).
- **Level Shifters**: Allows communication between devices operating at different voltage levels.
- **Modules**: A physical collection of one or more I²C devices connected to a common switch, including one SPRT EEPROM and the switch itself.
  - **Switching Multiplexer** (MUX): A bus-switching multiplexer (**TCA9548A**). Eight possible MUX addresses (0x70-0x77) enables the instantiation of up to 8 modules per I²C bus.
  - **SPRT EEPROM**: A non-volatile memory device (**24LC32**) located on MUX bus 0 at default address 0x50. Stores a Static Partial Routing Table (SPRT) encoding intra-network location information for all devices on one switch's subnetworks.
  - **Stuck-Bus Buffer**: A buffer that holds the I²C bus state during module hot-swapping (**TCA4307**).

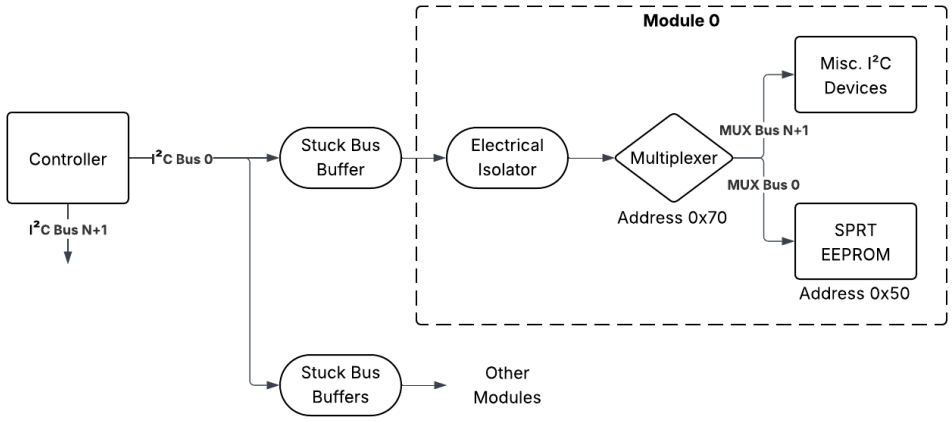

Figure 1: I²CIP Hardware Architecture Diagram

2.2 Software Protocols

- **Fully-Qualified Addressing** (FQA): Enables unique identification of each device in the network. Encodes I²C bus (3 bits; MSB 0-2), module number (3 bits; MSB 3-5), multiplexer bus (3 bits; MSB 6-8), and device address (7 bits; MSB 9-15) into a single 16-bit address.
- **Quality-of-Service Protocol** (QSP): Pings devices prior to and after any transmission, including multiplexers. Ensures transmission integrity and device availability using acknowledgments and retries with timeouts. Provides error level feedback to the Dynamic Routing Protocol (DRP).
- **Bus Switching Protocol** (BSP): Multiplexer control protocol that manages switching between multiple I²C buses on a single module.
- **Device Lookup Protocol** (DLP): Enables groups of device FQAs to be looked up by ID.
- **Reverse Device Lookup Protocol** (RDLP): Enables devices' IDs to be looked up by their FQA.
- **Intra-network Discovery Protocol** (IDP): Reads SPRT EEPROMs to build two parallel working maps of the intra-network topology: a hash table for DLP and a binary tree for RDLP.
- **Dynamic Routing Protocol** (DRP): Utilizes the working maps generated by IDP and the error level feedback from QSP to inform routing decisions for data transmissions; i.e. if a module is unreachable, DRP will remove the module's devices from the working maps.

3 Design Operations

3.1 Device Drivers

For each type of device connected to an I²CIP network, a corresponding device driver must be implemented within the host system's software stack. These drivers are responsible for managing communication with their respective devices, interpreting data, and providing a standardized interface for higher-level applications. These classes inherit from a common `Device` base class (which contains the **FQA** data and implements the **QSP** and **BSP** protocols during communications subroutines) as well as inheriting from either a `InputInterface` or `OutputInterface` (or `IOInterface`) abstract class, depending on the device's functionality. The `Interface` classes provide a templated set of methods for caching data read from input devices and writing data to output devices, while standardizing argument and return passing across all device drivers. The template parameter schema `<G, A, S, B>` represents the input buffer type ('G' for "get"), input argument type ('A'), output buffer type ('S' for "set"), and output argument type ('B'), respectively.

For example, the `EEPROM` class (the device driver for the 24LC32 IC) inherits from the `Device` base class and the `IOInterface` abstract class, with template parameters `<char*, uint16_t, char*, uint16_t>`. In this order, these template parameters represent the input buffer type (character array), input argument (read length), output buffer type (character array), and output argument (write length), respectively.

3.2 State Management

The fundamental building block of abstraction in I²CIP state management is the `DeviceGroup` class, which represents a collection of I²C devices of identical type. Each `DeviceGroup` contains an array of device driver instances, a factory (function pointer) for instantiating new devices, a handler (function pointer) for parsing JSON arguments into device-specific argument structures, and a cleanup (function pointer) for releasing those argument structures. The function pointers are passed to the `DeviceGroup` constructor from each device driver class, which defines them as static methods. This is achieved through a templated static factory method in the `DeviceGroup` class which takes one template parameter (the device driver class type) and returns a pointer to a newly instantiated `DeviceGroup` of that type.

Rather than managing `DeviceGroup` instances directly, they are grouped together within a higher-level `Module` class, which represents a physical I²CIP module. Each `Module` instance contains an hash table of `DeviceGroup` instances (by ID), implementing **DLP**. A single global-scope binary search tree is maintained by all `Module` instances, implementing **RDLP**.

3.3 Modular Networking

The `Module` class is the primary abstraction for managing I²CIP modules and devices within the host system's software stack. It defines three call operators: one taking an FQA and one taking a device ID, each updating a device's internal state based on the provided arguments, and one with no arguments that performs a module self-test that attempts to ping the MUX and rediscover the SPRT EEPROM. The method `discoverEEPROM` attempts to ping the SPRT EEPROM, adds it to the module's internal hash table of `DeviceGroup` instances (as well as the global binary search tree), and calls `parseEEPROMContents` (see below).

The `Module` class is a virtual class, with several virtual methods that must be implemented in a derived class. These methods include:

- `protected deviceGroupFactory`, which takes a device ID and returns a pointer to a newly instantiated `DeviceGroup` of the corresponding type (or `nullptr` if the ID is unrecognized) - allows the user to link custom device drivers into the I²CIP framework.
- `public parseEEPROMContents`, which takes the contents of the SPRT EEPROM - for example, implementing **IDP** (as defined in the `JSONModule` derived class).
- `public handleCommand` and `handleConfig`, which take `JSONObject` instances representing commands and configuration data, respectively - allows the user to define custom commands and configuration options for their specific module implementation.

3.4 Static Partial Routing Table

The Static Partial Routing Table (SPRT) is a critical component of the I²CIP technology, serving as a non-volatile memory storage for intra-network location information of all devices connected to a module's switch. Each module contains one SPRT EEPROM that holds the FQAs and IDs of its connected devices, allowing for efficient device discovery and routing within the network. The SPRT contents are UTF-8 encoded JSON, structured as an array of objects representing each multiplexer bus:

```
// Example SPRT JSON Contents
[
    // Array of MUX buses
    {
        // MUX bus 0
        "24LC32" : [ 80 ],    // Decimal encoding of device address 0x50
        "SHT31" : [ 68, 69 ], // Two SHT31 devices on MUX bus 0, 0x44 and 0x45
        "MCP23017" : [ 32 ],  // Another example device, 0x20
        // etc.
    },
    // Up to MUX bus 7
    { },
    {
        "24LC32" : [ 80, 81 ], // E.g. two more different EEPROM devices on MUX bus 2
    },
    { }, { }, { }, { }, { }
]
```

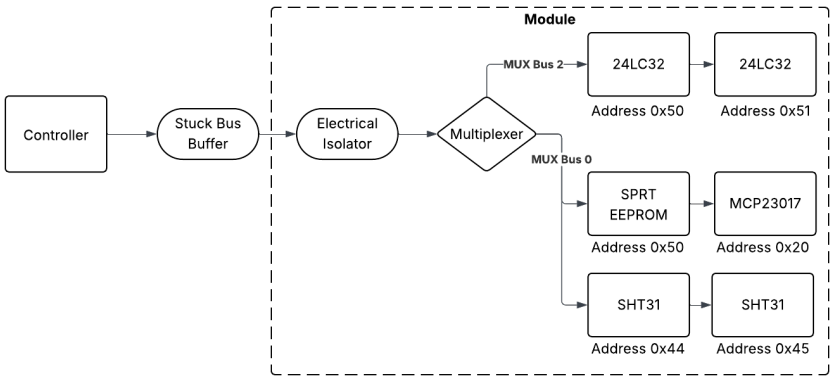

Figure 2: Graphical Representation of Example SPRT Module

3.5 User Interaction

- **MUX API**: Users can control multiplexers through a dedicated static API:
  - `pingMUX`: Attempts to communicate (ACK) with a specified MUX (wire & module number OR FQA).
  - `setBus`: Pings and switches a specified MUX to a specified MUX bus (wire & module number & MUX bus OR FQA).
  - `resetBus`: Pings and resets a specified MUX to deselect all MUX busses (wire & module number OR FQA).
- **Device API**: Users interact with I²CIP devices through the instance API:
  - `ping` and `pingTimeout`: Sets MUX bus and attempts to communicate (ACK) with the device (with optional timeout retries).
  - `writeByte` and `write`: Sets MUX bus and writes one or more bytes (`uint8_t` OR `uint8_t*` with `size_t`) to the device.
  - `writeRegister`: Sets MUX bus and writes one or more bytes (`uint8_t` OR `uint8_t*` with `size_t`) to a specified device register. Register size is specified as either 8-bit or 16-bit using argument overloading (`uint8_t` OR `uint16_t`).
  - `readByte` and `readWord` and `read`: Sets MUX bus and requests one or more bytes (`uint8_t` OR `uint16_t` OR `uint8_t*` with `size_t`) from the device. Note: for `read` the `size_t` argument is passed by reference, the actual number of bytes read is returned through this argument.
  - `readRegisterByte` and `readRegisterWord` and `readRegister`: Sets MUX bus and requests one or more bytes (`uint8_t` OR `uint8_t*` with `size_t`) from a specified device register. Register size is specified as either 8-bit or 16-bit using argument overloading (`uint8_t` OR `uint16_t`). Note: for `readRegister` the `size_t` argument is passed by reference, the actual number of bytes read is returned through this argument.

- **Module API**: Users manage I²CIP modules through the instance API:
  - `parseEEPROMContents`: Virtual. Ideally, implements **IDP** by parsing the contents of the SPRT EEPROM and populating the module's internal hash table of `DeviceGroup` instances (as well as the global binary search tree).
  - `discoverEEPROM`: Attempts to ping and rediscover the SPRT EEPROM on the module. Note: within EEPROM discovery, `parseEEPROMContents` is the only supported way to add devices to the module.
  - Self-Test `operator()`: Performs a module self-test that attempts to ping the MUX and rediscover the SPRT EEPROM.
  - Device Update `operator()`: Templated by device driver class. Takes a device FQA and performs **DLP** to ping the device. Optionally updates the device based on arguments (struct of void pointers) parsed from JSON by the device group's handler function.
  - `handleCommand` and `handleConfig`: Virtual. Ideally, implements custom handling of `JSONObject` instances to manage and modify module behaviour (i.e. control devices).

3.6 Operational Assumptions

Assumptions for operation of the I²CIP technology include:

- All devices are contained within modules that adhere to the I²CIP hardware specification.
- Each module contains a properly formatted SPRT EEPROM with accurate device information.
- All SPRT entries contain device IDs that correspond to actual device types for which a corresponding driver exists.

4 Design Requirements

5 Design Performance

6 Design Potential

References