POLITECNICO DI TORINO

Department of Electronics and Telecommunication

THESIS

Optimization Of SPI Communication in Precise Farming

Supervisor:
Prof. Massimo Violante

Candidate:
Abdelazim Mansour Abdelazim Mansour

March 2020

Acknowledgement

I would like to express my deep and sincere gratitude to my supervisor, Prof. Massimo Violante, Polytechnic University of Turin. His wide knowledge and his logical way of thinking have been of great value for me. His understanding, support and personal guidance have provided a good basis for the present thesis.

I would like also to thank Eng. Paolo Doz, Abinsula S.R.L. to give me the opportunity to work in such a professional environment. He was very helpful, supportive, encouraging without his consecutive comment, I could not finish the thesis in a professional way.

Also, I'm very grateful to all my colleagues in Abinsula Turin family for helping me and providing the perfect environment to complete this thesis.

Last but not least, I would like to express my gratitude for each and every member of my family for the moral support. I would not have been in this place in my life without their endless motivation.

Abstract

Nowadays, the automobile industry is one of the most important industry, the automobile has changed over the time from depending on heavily mechanical system to have integrated with electronics control system. This has happened mainly due to the advent of semiconductor technology at various levels of application in the automobile, such as body electronics, safety and security, and powertrain and chassis control, among others, which has helped in improving performance levels and operating efficiencies of vehicles.

To control all of these features an enormous number of processors, actuator and sensors have been added inside cars. As result we have multiple communication protocols and between each module and between each processor to help in transferring the data and synchronize the operation inside the car. To achieve high efficiency of the communication we need high bus speed to move the data between the modules.

The topic of this work is to enhance and optimize the communication between two processors (IMX6 and SPC560b) inside automobile for precise farming application. Inside our system we have two processors, one receiving the data from other modules over CAN bus and after that sending the data to the main processors over a SPI communication protocol. The main processor is an Embedded Linux based operating system. All the work has been achieved in the low level to improve our communication protocol so it will be effective in all applications. We have presented our result from a time consumed and from number of bits send over the bus.

Table of Contents

Chapter one

Introduction

1.1 Overview

The rapid evolution of the embedded systems industry and their capabilities in the past two decades makes it vital for the automobile evolution industry. Every year automobile manufactures relay on embedded systems to add a multiple function and make the automobile more and more smart.

Nowadays, embedded systems have a huge variety of application industrial equipment, medical devices, weapon control systems, aerospace systems, home automation systems, home appliance devices, entertainment devices, academic equipment, and so on.

1.2 What is embedded system

An embedded system is basically a computer system which consist of computer processor, computer memory, and input/output peripheral devices that has a dedicated function within a larger mechanical or electrical system.

There is a slight difference between an embedded system and desktop computer as in the last one it is more a general-purpose computing system which could be used for more than one function. But embedded system is built to serve a single purpose or a small set of purposes which at the end will serve one function .so it's unique in its characteristics, functionality and response. So, the embedded system must be low cost, low power, fast, small and dedicated to it function. Also, the response time is crucial in some application. The delay must be calculated to meet the requirements of the application.

Hence, the embedded systems are portable, have a special purpose and a lot of them are mass produced products, so we need to optimize them by enhance the response time and increase the response, reliability and performance and reducing consumed power, cost and size.

Figure 1.1 Embedded System Example

1.3 Embedded system in automotive industry

As we know, the automobile has changed over the decades from depending on heavily mechanical system. Then by the time these mechanical systems have been integrated into electronic control system. This has happened mainly due to the advent of semiconductor technology at various levels of application in the automobile, such as body electronics, safety and security, and powertrain and chassis control, among others, which has helped in improving performance levels and operating efficiencies of vehicles.

Automotive embedded systems are typically microcontroller systems with one or few dedicated functions, usually with real time computation constraints. In automobiles, embedded systems are used in infotainment and telematics, ADAS, airbags, head-up displays, electronic brake systems, power steering, and active suspension, among others [1]

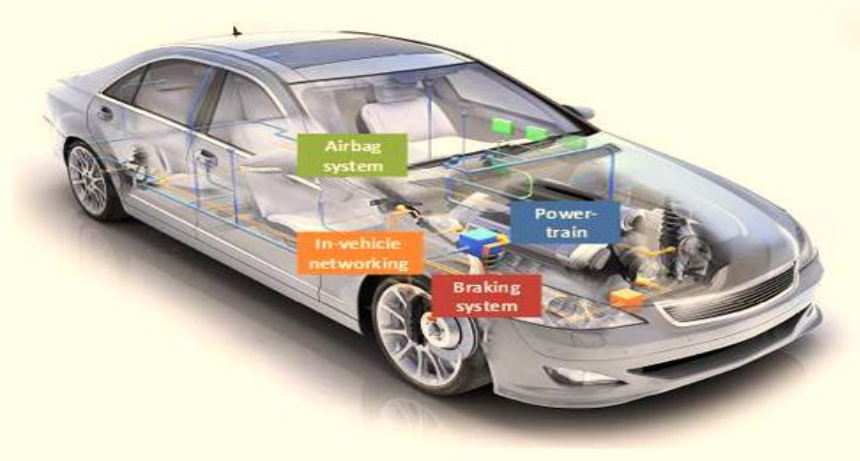


Figure 1.2 Embedded systems inside a car

1.4 Operating system for embedded system

An embedded operating system (OS) is a specialized operating system designed to perform a specific task for a device or system that is not a computer. An embedded operating system's main job is to run the code that allows the device to do its job. The embedded OS also makes the device's hardware accessible to the software that is running on top of the OS. So, it acts like a connection layer between the hardware and software application.

The purpose of an embedded operating system is:

- to ensure the embedded system operates in an efficient and reliable manner by managing hardware and software resources
- to provide an abstraction layer to simplify the process of developing higher layers of software
- to act as a partitioning tool.

The embedded OS (operating system) achieves some functions via a kernel which includes memory management and I/O system management components.

It comes down to balancing between utilizing the system's resources (i.e., keeping the CPU, I/O, etc. as busy as possible) – with task throughput to process as many tasks as possible in a given amount of time – with fairness and ensuring that task starvation does not occur when trying to achieve a maximum task throughput. The key for developers is to note that embedded operating systems impacts effectiveness and performance, and also not to underestimate the impact of an embedded OS's internal design.

The key differentiators between embedded operating systems in this regard are:

1. **Memory Management Scheme**, i.e., virtual memory swapping scheme and page faults
2. **Scheduling Scheme**, i.e., throughput, execution time, and wait time
3. **Performance**, i.e.,
    - Response time, to make the context switch to a ready task and waiting time of task in ready queue
    - Turnaround time, how long a process takes to complete running
    - Overhead, the time and data needed to determine which tasks will run next
    - Fairness, what are the determining factors as to which processes get to run.[2]

There are many types of operating system for embedded system

- QNX
- VxWorks
- INTEGRITY
- ThreadX
- MicroC/OS2
- embOS
- SafeRTOS
- Linux
- eCos
- FreeRTOS
- RTAI
- Coscox
- Rocket OS
- Windows CE

1.4.1 FreeRTOS

FreeRTOS is a real–time operating system (RTOS) for microcontrollers and small microprocessors. Distributed freely under the MIT open source license, FreeRTOS includes a kernel and a growing set of libraries suitable for use across all industry sectors. FreeRTOS is built with an emphasis on reliability, accessibility, and ease of use.

FreeRTOS is totally free and could be used in commercial application for free. Also, there is a lot of reasons which makes FreeRTOS a good choice:

- Has a minimal ROM, RAM and processing overhead. Typically an RTOS kernel binary image will be in the region of 6K to 12K bytes.
- Is very simple – the core of the RTOS kernel is contained in only 3 C files.
- Has a migration path to Safe**RTOS**, which includes certifications for the medical, automotive and industrial sectors.
- Provides a single and independent solution for many different architectures and development tools.
- Contains a pre-configured example for each port. No need to figure out how to setup a project – just download and compile!
- Has an excellent, monitored, and active free support forum.
- Has the assurance that commercial support is available should it be required.
- Is very scalable, simple and easy to use.
- FreeRTOS offers a smaller and easier real time processing alternative for applications where eCOS, embedded Linux (or Real Time Linux) and even uCLinux won't fit, are not appropriate, or are not available.[3]

1.4.2 Embedded Linux

Embedded Linux is a type of Linux operating system/kernel that is designed to be installed and used within embedded devices and appliances. It is a compact version of Linux that offers features and service in line with the operating and application requirements of the embedded system

Though in Embedded Linux, we are utilizing the same Linux kernel, which will be a little bit different than the standard Linux OS. Embedded Linux is specifically customized for embedded systems. Therefore it is has a much smaller size, which requires less power for processing and has minimal features. In the view of these requirements of the underlying embedded system, the Linux kernel is edited and optimized as an embedded Linux version.

To increase the use of Linux in embedded applications, many industrial groups have been formed. Among the others:

- CE Linux Forum , founded in 2003 to aid the inclusion of embedded features in the main Linux kernel branch.
- Linux Foundation , formerly Open Source Development Labs.
- Linux Phone Standards Forum , created in 2004 to pursue a standard applications environment for Linux-based mobile phones and "converged devices".
- LiMo Foundation , founded in 2006 by Motorola, NEC, Panasonic, Samsung, DoCoMo and Vodafone to establish a set of interfaces and standard reference components to improve the third-party mobile phone developer base.

- Embedded Linux Consortium , until its 2005 closure and transfer of operations to the Linux Foundation, included IBM, Intel, LynuxWorks and others, and focused on application programming interface standardization. The Embedded Linux Consor- tium produced the ELCPS (Embedded Linux Consortium Platform Specification) which was intended as a guide to developers of embedded Linux devices as to what functionality should be included in order to provide a standard platform supporting application portability.

Most embedded hardware requires some type of software initialization and memory management. The software that directly deals with and controls this hardware is called a device driver. All embedded systems that require software have, at least a device driver software in their system software layer. Device drivers are the software libraries that initialize the hardware and manage access to the hardware by higher layers of software.

Device drivers are basically specified on two types either they are architecture-specific or generic. A device driver which is architecture-specific manages the hardware that is integrated into the master microprocessor (include in the microprocessor architecture). As an example of the architecture-specific drivers that it is initialize and enable components within a master processor include on-chip memory, integrated memory managers (memory management units (MMUs)), and floating-point hardware. A device driver which is generic, manages hardware that is located on the board and not integrated onto the master microprocessor. In a generic driver, there are typically architecture-specific portions of source code, because the master processor is the central control unit and to gain access to anything on the board usually means going through the master processor. However, the generic driver also manages board hardware that is not specific to that particular processor, which means that a generic driver can be configured to run on a variety of architectures that contain the related board hardware for which the driver is written. Generic drivers include code that initializes and manages access to the remaining major components of the board, including board buses (I2C, PCI, PCMCIA, etc.), off-chip memory (controllers, level 2+ cache, Flash, etc.), and off-chip I/O (Ethernet, RS-232, display, mouse, etc.).[4]

1.4.3 OSEK

OSEK (***O****ffene* ***S****ysteme und deren Schnittstellen für die* ***E****lektronik in* ***K****raftfahrzeugen*; English: "*Open Systems and their Interfaces for the Electronics in Motor Vehicles*") is a standards body for an Embedded operating system which has specifications for a communications stack and a network management protocol for automotive embedded systems. It has also produced other related specifications. OSEK was designed to provide a standard software architecture for the various electronic control units (ECUs) throughout a car.

Cleary there were some excellent open source RTOS alternatives (FreeRTOS, eCos, Jaluna, etc). but there are some problems with these operating systems, from an automotive perspective, like:

- alternatives are quite bulky and their real-time performance is not quite "real" enough
- The API of most alternatives (typically POSIX) is not ideally suited to the combination of synchronous and asynchronous task scheduling
- The OSEK API was designed by automotive suppliers

Some of the advantages of OSEK are :

- Clear savings in costs and development time
- Enhanced quality of the software of control units
- Standardized interfacing features
- Utilization of the existing resources in the vehicle, enhance the performance of the overall system without requiring additional hardware
- Absolute independence with regards to individual implementation, as the specification does not prescribe implementation aspects
- Overhead in terms of memory usage and code footprint

1.4.4 INTEGRITY

Green Hills Software Integrity 178B operating system is the first, and only, certified Common Criteria Evaluation Assurance Level (EAL) 6+ operating system on the market. Green Hills Software uses Integrity as the basis for a secure PC operating system called Integrity PC and includes Padded Cell Virtualization, a secure hypervisor running within Integrity PC. Integrity Global Security LLC has been formed as a subsidiary of Green Hills Software to market Integrity PC.

Integrity also leverages advanced features in Intel's vPro chipset and the Trusted Computing Group's TPM for a trusted boot process. Integrity LLC engineers even claim to thwart hypervisor root kit.

As one of the first RTOSes to leverage hardware memory- management units (MMUs), INTEGRITY is a true, hard real-time operating system that never sacrifices real-time performance for security and protection. INTEGRITY can respond to events in nanoseconds, guaranteed.

All INTEGRITY kernel services have been carefully optimized to minimize the overhead of system calls. Complex system calls can be suspended to allow others to execute. INTEGRITY uses a true real-time scheduler that supports multiple priority levels and enables complete control over CPU percentage allocation.

INTEGRITY’s unique memory quota system keeps one address space from exhausting the memory of any other. To ensure adequate kernel memory, INTEGRITY requires that kernel memory not be used for messages, semaphores, or other kernel objects created in response to process requests. Instead, the kernel performs all services requested by a process using the memory resources that the requesting process supplies.

The modern architecture of INTEGRITY is well suited for multicore processors targeting embedded systems. INTEGRITY provides complete Asymmetrical Multiprocessing (AMP) and Symmetrical Multiprocessing (SMP) support that is optimized for embedded and real time use. Embedded system designers can select the multiprocessing architecture that is right for the task. When coupled with the advanced multicore debugging features found in the Green Hills MULTI® toolsuite, developers will reduce their time to market while increasing system performance and reliability.

Built on the INTEGRITY separation kernel, Green Hills Platforms provide complete, pre-integrated solutions customized to application specific requirements for a range of applications like Automotive platform, Secure mobile device platform , Avionics Platform , Secure networking platform, industrial safety platform , software defined platform , medical devices platform and wireless devices platform.[5]

Chapter two

Communication Protocols in automotive environment

2.1 Introduction

Vehicle communications have been growing rapidly in last decade with growing of the electronics circuit and communication in the vehicle as replacing for mechanical systems. We started to have more and more of the Electronic Control Modules (ECMs) in the vehicle interacting with other ECMs. As we refer existing technologies in automotive, vehicles contains at least guess more than 30 ECMs in the average vehicle. Some luxury vehicles claim having for more than 90 ECMs. This makes automotive communication protocols more challenging and need rapid improvement as users and systems designer.

All ECMs are demanding more information to be available in-vehicle. We need diffident bus communication based on type of ECMs. Type of ECMs will decided by its criticality of Application and that makes communication more complicated. In this chapter we will discuss few major automotive communication protocol. Each has its own importance in vehicle based on its Advantages. Some protocols are expensive in terms of controller or very complicated for implementation than other but it provided more robust and efficient data flow in-vehicles.[6]

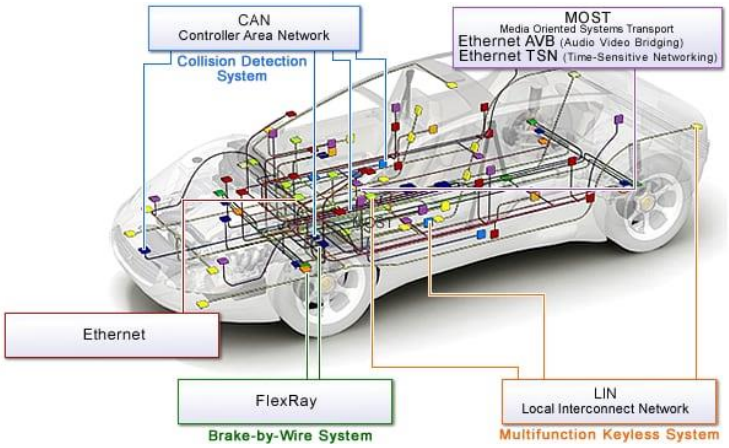


Figure 2.1 Communication network inside car

Also, inside each ECM there is some communication protocol used between ECUs for sending and receiving data. Choosing between all these types of the communication depend on the hardware availability in each ECU, the communication, baud rate needed, and the application used ‘’serial or parallel ‘’ or ‘’synchronous or asynchronous’’.

2.2 Internal bus Communication protocol

2.2.1 SPI

The Serial Peripheral Interface Bus or SPI bus is a synchronous serial data link standard named by Motorola which working as full duplex communication protocol. Devices communicate in master/slave mode where the master device start the communication. Multiple slave devices are allowed with individual slave select (chip select) line where the master can choose between them which one to be communicated with. Sometimes SPI is called a ”four wire” serial bus.

SPI is a synchronous data bus, which means that it uses separate lines for data and a clock that keeps both sides in perfect sync. The clock is an oscillating signal that tells the receiver exactly when to sample the bits on the data line or to start to send its own data. This could be on the rising edge (low to high) or on falling edge (high to low) of the clock signal “the datasheet will specify which one to be used” depending on memory register initialization. When the receiver detects that edge, it will immediately look at the data line to read the next bit. Because the clock is sent along with the data, specifying the speed(baud rate) isn't important, although devices will have a top baud rate at which they can receive and send bits.

Data transmission

In SPI, only one side generates the clock signal (usually called CLK or SCK for Serial ClocK). This side that generates the clock is called the "master", and the other side is called the "slave". There is always only one master (which is almost always the microcontroller), but there can be multiple slave connected to the same master.
When data is sent from the master to a slave, it's sent on a data line called MOSI "Master Out / Slave In". If the slave needs to send a response back to the master,or send some information periodically, so the master will continue to generate a prearranged number of clock cycles, and the slave will put the data onto a third data line called MISO, "Master In / Slave Out" to be received by the master.

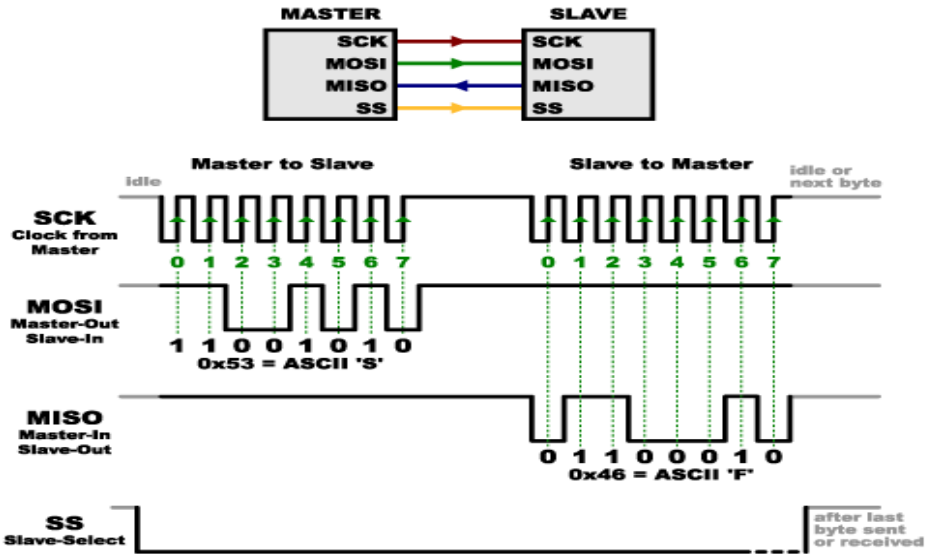


Figure 2.2 SPI communication buses

Because the master always generates the clock signal, it must know in advance when a slave should return data or answer back a communication and how much data will be returned. This is very different than asynchronous serial, where random amounts of data could be sent from master to slave or the opposite at any time without previous knowledge. In actual project this isn't a problem, as SPI is generally used to communicate with sensors that have a very specific function and known data structure. So, the master first configures the clock, using a frequency less than or equal to the maximum frequency the slave device supports. Such frequencies are commonly in the range of 1-70 MHz.

For example, if you send the command for "read data" to a slave, you know that the slave will always send back the reply, for example, two bytes in return. (In cases where you might want to return a variable amount of data, you could always specify two bytes in the beginning of the communication carrying the information of the length of the data and then have the master will know how many byte he expected and adjust the number of the clock cycle to retrieve the whole amountof data).

Note that SPI is a full duplex communication (has separate send and receive lines), and thus, in certain situations, you can transmit and receive data *at the* same time (for example, requesting a new sensor reading while retrieving the data from the previous one). Your device's datasheet will tell you if this is possible or not.

Transmissions normally involve two shift registers of some given word size, such as eight bits, one in the master and one in the slave; they are connected in a ring. Data is usually shifted out with the most significant bit first, while shifting a new least significant bit into the same register. After that register has been shifted out, the master and slave have exchanged register values. Then each device takes that value and does something with it, such as writing it to memory. If there is more data to exchange, the shift registers are loaded with new data and the process repeats.

Slave selects

This signal tells the slave that it should wake up and receive / send data and is also crucial when multiple slaves are present to select the one you'd like to send /receive data with. The SS line is normally held high, which disconnects the slave from the SPI bus “active low logic”.
Before data is sent to the slave, the line is brought low, which activates the slave. When you're done using the slave, the line is made high again. In a shift register, this corresponds to the "latch" input, which transfers the received data to the output lines.

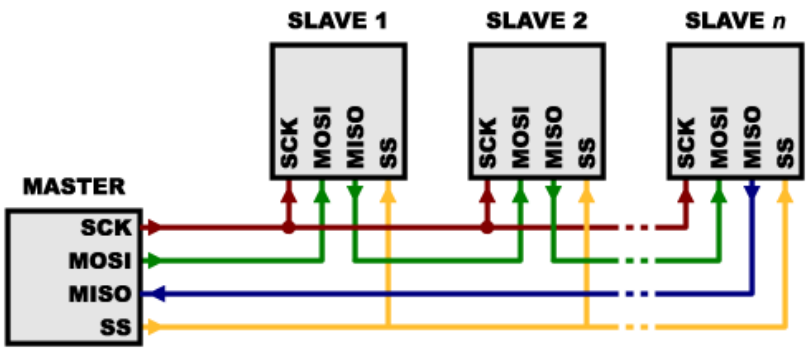


Figure 2.3 SPI communication with multiple slave

Pros and cons of SPI

Advantages

- Full duplex communication
- Higher throughput than I2C or SMBus
- Uses many fewer pins on IC packages, and wires in board layouts or connectors, than parallel interfaces
- At most one ”unique” bus signal per device (SS); all others are shared

- Complete protocol flexibility for the bits transferred
  - Not limited to 8-bit words
  - Arbitrary choice of message size, content, and purpose
- Extremely simple hardware interfacing
  - Typically lower power requirements than I2C or SMBus due to less circuitry (including pullups)
  - No arbitration or associated failure modes
  - Slaves use the master’s clock, and don’t need precision oscillators
  - Transceivers are not needed

Disadvantages

- Requires more pins on IC packages than I2C
- No in-band addressing; out-of-band chip select signals are required on shared buses
- No hardware flow control
- No hardware slave acknowledgement (the master could be talking to nothing and not know it)
- Supports only one master device
- Without a formal standard, validating conformance is not possible
- Only handles short distances compared to RS-232, RS-485, or CAN-bus

Clock polarity and phase

In addition to setting the clock frequency, the master might have the possibility to configure the clock polarity and phase with respect to the data. Motorola SPI Block Guide names these two options as CPOL and CPHA (for clock "pol"arity and "pha"se) respectively.

The timing diagram is shown to the right. The timing is further described below and applies to both the master and the slave device.

- CPOL determines the polarity of the clock. The polarities can be converted with a simple inverter.
  - CPOL=0 is a clock which is idle at 0, and each cycle consists of a pulse of 1. That is the leading edge is a rising edge, and the trailing edge is a falling edge.

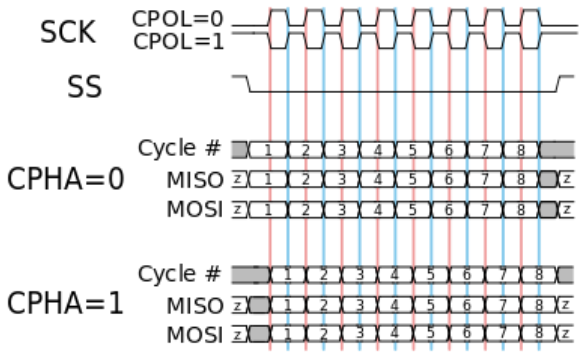


Figure 2.4 SPI communication clock signal

  - CPOL=1 is a clock which is idle at 1, and each cycle consists of a pulse of 0. That is, the leading edge is a falling edge, and the trailing edge is a rising edge.
- CPHA determines the timing (i.e. phase) of the data bits relative to the clock pulses. Conversion between these two forms is non-trivial.
  - For CPHA=0, the "out" side changes the data on the trailing edge of the preceding clock cycle, while the "in" side captures the data on (or shortly after) the leading edge of the clock cycle. The out side holds the data valid until the trailing edge of the current clock cycle. For the first cycle, the first bit must be on the MOSI line before the leading clock edge.
  - An alternative way of considering it is to say that a CPHA=0 cycle consists of a half cycle with the clock idle, followed by a half cycle with the clock asserted.
  - For CPHA=1, the "out" side changes the data on the leading edge of the current clock cycle, while the "in" side captures the data on (or shortly after) the trailing edge of the clock cycle. The out side holds the data valid until the leading edge of the following clock cycle. For the last cycle, the slave holds the MISO line valid until slave select is deasserted.
  - An alternative way of considering it is to say that a CPHA=1 cycle consists of a half cycle with the clock asserted, followed by a half cycle with the clock idle.

The MOSI and MISO signals are usually stable (at their reception points) for the half cycle until the next clock transition. SPI master and slave devices may well sample data at different points in that half cycle.This adds more flexibility to the communication channel between the master and slave.

2.2.2 SPI on MPC5607B

The SPI in the board is DSPI "**D**eserial **S**erial **P**eripheral **I**nterface", which provides a synchronous serial bus for communication between the MCU and an external peripheral device.

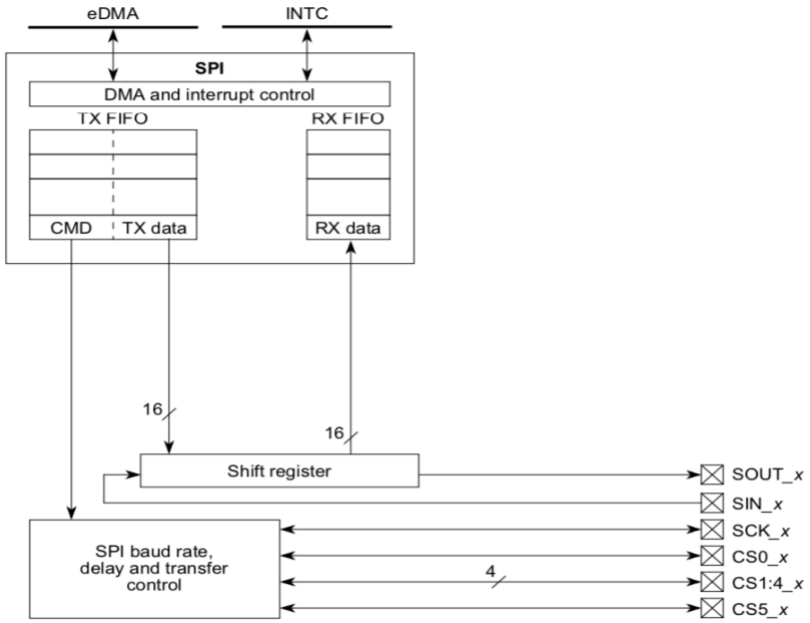


Figure 2.4 DSPI block diagram

The DSPI has one configuration, namely serial peripheral interface (SPI), in which the DSPI operates as a basic SPI or a queued SPI.

The 16-bit shift register in the master and the 16-bit shift register in the slave are linked by the SOUT_*x* and SIN_*x* signals to form a distributed 32-bit register. When a data transfer operation is performed, data is serially shifted a pre-determined number of bit positions. Because the registers are linked, data is exchanged between the master and the slave; the data that was in the master's shift register is now in the shift register of the slave, and vice versa. At the end of a transfer, the TCF bit in the DSPI*x*_SR is set to indicate a completed transfer. [7]

Features

The DSPI supports these SPI features:

- Full-duplex, three-wire synchronous transfers
- Master and slave mode
- Buffered transmit and receive operation using the TX and RX FIFOs, with depths
- four entries Visibility into TX and RX FIFOs for ease of debugging
- FIFO bypass mode for low-latency updates to SPI queues
- Up to 6 peripheral chip selects, expandable to 64 with external demultiplexer
- Deglitching support for up to 32 peripheral chip selects with external demultiplexer
- Programmable transfer attributes on a per-frame basis
  - 6 clock and transfer attribute registers
  - Serial clock with programmable polarity and phase
  - Programmable delays
    - CS to SCK delay
    - SCK to CS delay
    - Delay between frames
  - Programmable serial frame size of 4 to 16 bits, expandable with software control
  - Continuously held chip select capability
- Two DMA conditions for SPI queues residing in RAM or flash
  - TX FIFO is not full (TFFF)
  - RX FIFO is not empty (RFDF)
- 6 interrupt conditions:
  - End of queue reached (EOQF)
  - TX FIFO is not full (TFFF)
  - Transfer of current frame complete (TCF)
  - RX FIFO is not empty (RFDF)
  - FIFO overrun (attempt to transmit with an empty TX FIFO or serial frame received while RX FIFO is full) (RFOF) or (TFUF)
- Modified SPI transfer formats for communication with slower peripheral devices
- Supports all functional modes from QSPI subblock of QSMCM (MPC500 family)
- Continuous serial communications clock (SCK)[7]

Modes of operation

The DSPI has five modes of operation. These modes can be divided into two categories:

- Module-specific: Master, Slave, and Module Disable modes
- MCU-specific: External Stop and Debug modes

The module-specific modes are entered by host software writing to a register. The MCU-specific modes are controlled by signals external to the DSPI. An MCU-specific mode is a mode that the entire device may enter, in parallel to the DSPI being in one of its module-specific modes.

Master mode

Master mode allows the DSPI to initiate and control serial communication. In this mode the SCK, CS*n* and SOUT signals are controlled by the DSPI and configured as outputs.

In SPI configuration, master mode transfer attributes are controlled by the SPI command in the current TX FIFO entry. The CTAS field in the SPI command selects which of the eight DSPI*x*_CTARs are used to set the transfer attributes. Transfer attribute control is on a frame by frame basis.

Slave mode

Slave mode allows the DSPI to communicate with SPI bus masters. In this mode the DSPI responds to externally controlled serial transfers. The DSPI cannot initiate serial transfers in slave mode. In slave mode, the SCK signal and the CS0_*x* signal are configured as inputs and provided by a bus master. CS0_*x* must be configured as input and pulled high. If the internal pullup is being used then the appropriate bits in the relevant SIU_PCR must be set (SIU_PCR [WPE = 1], [WPS = 1]).

Module Disable mode

The module disable mode is used for MCU power management. The clock to the non-memory mapped logic in the DSPI is stopped while in module disable mode. The DSPI enters the module disable mode when the MDIS bit in DSPI*x*_MCR is set.

Debug mode

Debug mode is used for system development and debugging. If the device enters debug mode while the FRZ bit in the DSPI*x*_MCR is set, the DSPI halts operation on the next frame boundary. If the device enters debug mode while the FRZ bit is cleared, the DSPI behavior is unaffected and remains dictated by the module-specific mode and configuration of the DSPI. [7]

Memory map

All the registers can be accessed by simply read and write into memory. As there is 6 DSPI modules in the MPC5607BRM so, we have 6 base addresses to access each register of them.

- 0xFFF9_0000 (DSPI_0)
- 0xFFF9_4000 (DSPI_1)
- 0xFFF9_8000 (DSPI_2)
- 0xFFF9_C000 (DSPI_3)
- 0xFFFA_0000 (DSPI_4)
- 0xFFFA_4000 (DSPI_5)

Then we add the address pf the specified register we want to access.[7]

Baud rate generator

The SCK_*x* frequency and the delay values for serial transfer are generated by dividing the system clock frequency by a prescaler and a scaler with the option of doubling the baud rate.

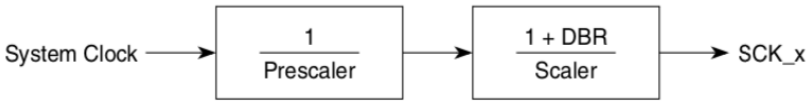


Figure 2.5 Communications clock prescalers and scalers

The baud rate is the frequency of the serial communication clock (SCK_*x*). The system clock is divided by a baud rate prescaler (defined by DSPI*x*_CTAR[PBR]) and baud rate scaler (defined by DSPI*x*_CTAR[BR]) to produce SCK_*x* with the possibility of doubling the baud rate. The DBR, PBR, and BR fields in the DSPI*x*_CTARs select the frequency of SCK_*x* using the following formula:

$$\text{SCK baud rate} = \frac{fsys}{PBRPrescalerValue} ¥ \frac{1+DBR}{BRScalerValue}$$

DSPI DMA / Interrupt Request Select

The DSPI*x*_RSER serves two purposes:

- It enables flag bits in the DSPI*x*_SR to generate DMA requests or interrupt requests.
- It selects the type of request to generate.[7]

The bit description for the type of requests the are supported is shown in the following figure.

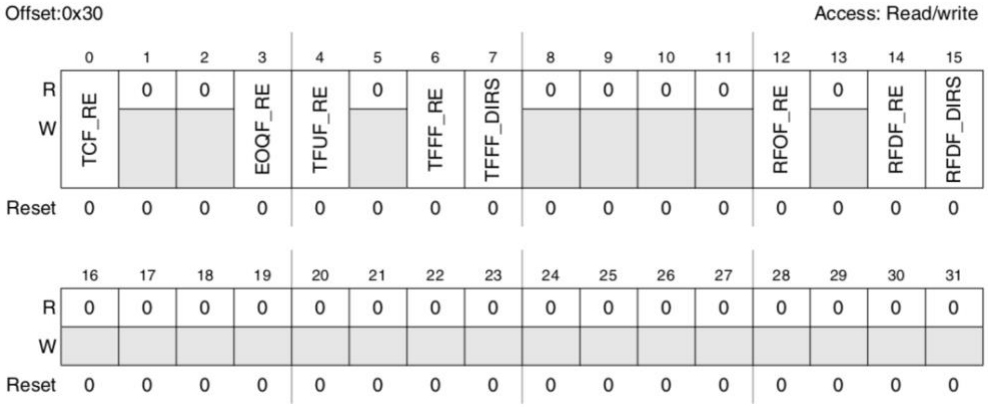


Figure 2.6 DSPI DMA / Interrupt Request Select and Enable Register (DSPIx_RSER)

In the following table there is a description for every bit in the register.

Field	Description
TCF_RE	Transmission complete request enable Enables TCF flag in the DSPI*x*_SR to generate an interrupt request. 0 TCF interrupt requests are disabled 1 TCF interrupt requests are enabled
EOQF_RE	DSPI finished request enable Enables the EOQF flag in the DSPI*x*_SR to generate an interrupt request. 0 EOQF interrupt requests are disabled 1 EOQF interrupt requests are enabled
TFUF_RE	Transmit FIFO underflow request enable The TFUF_RE bit enables the TFUF flag in the DSPI*x*_SR to generate an interrupt request. 0 TFUF interrupt requests are disabled 1 TFUF interrupt requests are enabled
TFFF_RE	Transmit FIFO fill request enable Enables the TFFF flag in the DSPI*x*_SR to generate a request. The TFFF_DIRS bit selects between generating an interrupt request or a DMA requests. 0 TFFF interrupt requests or DMA requests are disabled 1 TFFF interrupt requests or DMA requests are enabled
TFFF_DIRS	Transmit FIFO fill DMA or interrupt request select Selects between generating a DMA request or an interrupt request. When the TFFF flag bit in the DSPI*x*_SR is set, and the TFFF_RE bit in the DSPI*x*_RSER is set, this bit selects between generating an interrupt request or a DMA request. 0 Interrupt request is selected 1 DMA request is selected
RFOF_RE	Receive FIFO overflow request enable Enables the RFOF flag in the DSPI*x*_SR to generate an interrupt requests. 0 RFOF interrupt requests are disabled 1 RFOF interrupt requests are enabled
RFDF_RE	Receive FIFO drain request enable Enables the RFDF flag in the DSPI*x*_SR to generate a request. The RFDF_DIRS bit selects between generating an interrupt request or a DMA request. 0 RFDF interrupt requests or DMA requests are disabled 1 RFDF interrupt requests or DMA requests are enabled
RFDF_DIRS	Receive FIFO drain DMA or interrupt request select Selects between generating a DMA request or an interrupt request. When the RFDF flag bit in the

	DSPI*x*_SR is set, and the RFDF_RE bit in the DSPI*x*_RSER is set, the RFDF_DIRS bit selects between generating an interrupt request or a DMA request. 0 Interrupt request is selected 1 DMA request is selected

Table 2.1 DSPIx_RSER field descriptions

2.2.3 Inter-Integrated Circuit (I2C)

I2C is a serial protocol ,multi-master,multi-slave ,synchronous, packet switched and single-ended for two-wirecommunication interface to connect low-speed devices such as microcontrollers, A/D and D/A converters, EEPROMs, I/O interfaces and other similar peripherals in any embedded systems. It was invented by Philips (NXP now) and now it is exist in almost all major IC manufacturers.

I2C bus is popular communication protocol because it is simple to use, there can be more than one master, only upper bus speed is defined and there are two wires with pull-up resistors which are needed to connect unlimited number of I2C devices.

Each I2C slave device needs an unique address to be communicated with during the communication. The initial I2C specifications defined maximum clock frequency of 100 kHz. This was increased to 400 kHz later as Fast mode. There is also another high speed mode which can go up to 3.4 MHz and there is other one is 5 MHz as an ultra-fast mode.

Data Transmission

In the I2C there are only two wires during the communication one for the data and the other bus for the synchronized clock. The figure below illustrates an example of an I2C bus.

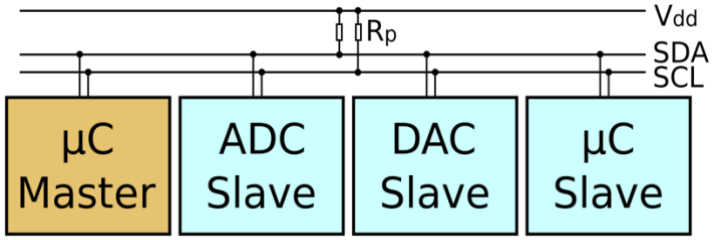


Figure2.6 I2C communication with multiple slave

SDA (Serial Data) – The line for data to allow the master and slave to send and receive data.
SCL (Serial Clock) – The line which carries the clock signal.

With I2C, data is transferred in messages. Messages are divide into frames of data. Each message has an address frame which contains the binary address of the slave, and after that, one or more data frames that contain the data needed to be transmitted. The message also includes read/write bits, start and stop conditions and ACK/NACK bits between each data frame.

Start Condition: The SDA line changes from high “high voltage level” to low “low voltage level” before the SCL line goes from high to low.

Stop Condition: The SDA line goes from a low to a high after the SCL line switches from low to high.

Address Frame: is 7 or 10 unique bit sequence to each slave which identifies the slave when the master wants to send/receive data to it.

Read/Write Bit: is a single bit which specify whether the master is requesting data from it (high voltage level) or sending data to the slave (low voltage level).

ACK/NACK Bit: Each frame in a message is followed at the end by an acknowledge/no-acknowledge bit. If an address frame or data frame was correctly received, an ACK bit is returned to the sender to notify him from the receiving device.
The figure below shows the anatomy of the I2C message.

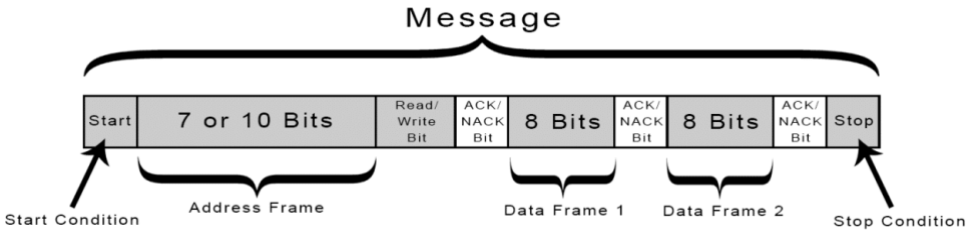


Figure2.7: anatomy of I2C message

Addressing

I2C doesn't have slave select lines like SPI for slave select, so it needs another way to tell the slave that data is being sent to it.So, the sender does this by addressing. The address frame is always the first frame sent after the start bit in a new message.

The master sends the address of the slave which wants to communicate with on the bus to every slave connected to it. Then, each slave compares the address received from the master to its own address. If the address matches, the receiver sends a low voltage ACK bit back to the sender. If the address doesn't match, the receiver does nothing and the SDA line remains high.

The address frame includes a single bit at the end of the frame which informs the slave whether the master wants to send data to it or receive data from it. If the master wants to write data to the slave, the read/write bit is a low. If the master is requesting data from the slave, So the bit is a high.

The data frame is always 8 bits long, and it is sent with the most significant bit first. Each data frame is always followed by an ACK/NACK bit to make sure that the frame has been received the data successfully. The ACK bit must be received by either the slave or the master (depending on who is sending the data) before the next data frame should be sent.

After all of the data have been sent, the master can send a stop order to the slave to stall the transmission. The stop condition is a voltage transition which goes from low to high on the SDA line after changes from low to high transition on the SCL line, where the SCL line remaining high.

Transmission Handling

- The master sends the start condition to every connected slave by switching the SDA line from a high voltage level to a low voltage level *before* switching the SCL line from high to low
- The master sends each slave the 7 or 10 bit address of the slave it wants to communicate with, along with the read/write bit.
- Each slave compares the address sent from the master to its own address. If the address matches, the slave returns an ACK bit by pulling the SDA line low for one bit. If the address from the master does not match the slave's own address, the slave leaves the SDA line high.
- The master sends or receives the data frame.
- After each data frame has been transferred, the receiving device returns another ACK bit to the sender to acknowledge successful receipt of the frame.
- To stop the data transmission, the master sends a stop condition to the slave by switching SCL high before switching SDA high.

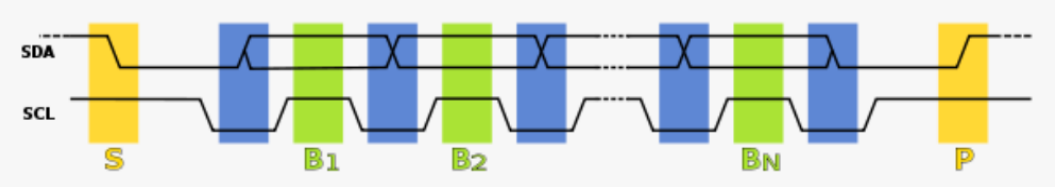


Figure2.8 I2C bus communication

Pros and cons of I2C

Advantages

- Only uses two wires
- Supports multiple masters and multiple slaves
- ACK/NACK bit gives confirmation that each frame is transferred successfully
- Hardware is less complicated than with UARTs
- Well known and widely used protocol

Disadvantages

- Slower data transfer rate than SPI
- The size of the data frame is limited to 8 bits
- More complicated hardware needed to implement than SPI

2.2.4 I2C on MPC5607B

This bus is suitable for applications requiring occasional communications over a short distance between a number of devices. It also provides flexibility, allowing additional devices to be connected to the bus for further expansion and system development.

The interface is designed to operate up to 100 kbps in Standard Mode and 400 Kbps in Fast Mode. The device is capable of operating at higher baud rates, up to a maximum of module clock/20 with reduced bus loading. Actual baud rate can be less than the programmed baud rate and is dependent on the SCL rise time. SCL rise time is dependent on the external pullup resistor value and bus loading. The maximum communication length and the number of devices that can be connected are limited by a maximum bus capacitance of 400 pF.

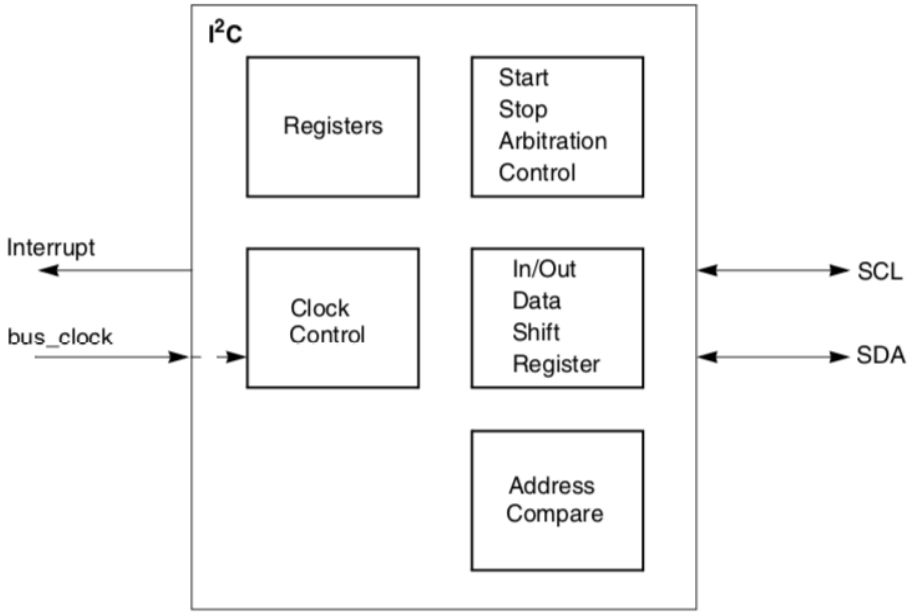


Figure2.9 block diagram I2C

Features

The I2C module has the following key features:

- Compatible with I2C Bus standard
- Multi-master operation
- Software programmable for one of 256 different serial clock frequencies
- Software selectable acknowledge bit
- Interrupt driven byte-by-byte data transfer
- Arbitration lost interrupt with automatic mode switching from master to slave
- Calling address identification interrupt
- Start and stop signal generation/detection
- Repeated start signal generation
- Acknowledge bit generation/detection
- Bus busy detection

Features currently not supported:

- No support for general call address
- Not compliant to ten-bit addressing

Memory map

The memory map for the I2C module is given below in Table. The total address for each register is the sum of the base address for the I2C module and the address offset for each register. The base address is **0xFFE3_0000.**[7]

Address offset	Register
0x0	I^2C Bus Address Register (IBAD)
0x1	I^2C Bus Frequency Divider Register (IBFD)
0x2	I^2C Bus Control Register (IBCR)
0x3	I^2C Bus Status Register (IBSR)
0x4	I^2C Bus Data I/O Register (IBDR)
0x5	I^2C Bus Interrupt Config Register (IBIC)

Table 2.2 I2C memory map

All registers are accessible via 8-bit, 16-bit or 32-bit accesses. However, 16-bit accesses must be aligned to 16-bit boundaries, and 32-bit accesses must be aligned to 32-bit boundaries. As an example, the IBDF register for the frequency divider is accessible by a 16-bit read/write to address Base + 0x000, but performing a 16-bit access to Base + 0x001 is illegal.

Steps of initialization sequence

Initialization sequence

Reset will put the I2C Bus Control Register to its default state. Before the interface can be used to transfer serial data, an initialization procedure must be carried out, as follows:

1. Update the Frequency Divider Register (IBFD) and select the required division ratio to obtain SCL frequency from system clock.
2. Update the I2C Bus Address Register (IBAD) to define its slave address.
3. Clear the IBCR[MDIS] field to enable the I2C interface system.
4. Modify the bits of the I2C Bus Control Register (IBCR) to select Master/Slave mode, Transmit/Receive mode and interrupt enable or not. Optionally also modify the bits of the I2C Bus Interrupt Config Register (IBIC) to further refine the interrupt behavior.

Generation of START

After completion of the initialization procedure, serial data can be transmitted by selecting the 'master transmitter' mode. If the device is connected to a multi-master bus system, the state of the I2C Bus Busy bit (IBB) must be tested to check whether the serial bus is free.
If the bus is free (IBB=0), the start condition and the first byte (the slave address) can be sent. The data written to the data register comprises the slave calling address and the LSB, which is set to indicate the direction of transfer required from the slave.[7]

The bus free time (i.e., the time between a STOP condition and the following START condition) is built into the hardware that generates the START cycle. Depending on the relative frequencies of the system clock and the SCL period, it may be necessary to wait until the I2C is busy after writing the calling address to the IBDR before proceeding with the following instructions.

Post-transfer software response

Transmission or reception of a byte will set the data transferring bit (TCF) to 1, which indicates one byte communication is finished. The I2C Bus interrupt bit (IBIF) is set also; an interrupt will be generated if the interrupt function is enabled during initialization by setting the IBIE bit. The IBIF (interrupt flag) can be cleared by writing 1 (in the interrupt service routine, if interrupts are used).

The TCF bit will be cleared to indicate data transfer in progress whenever data register is written to in transmit mode, or during reading out from data register in receive mode. The TCF bit should not be used as a data transfer complete flag as the flag timing is dependent on a number of factors including the I2C bus frequency. This bit may not conclusively provide an indication of a transfer complete situation. It is recommended that transfer complete situations are detected using the IBIF flag

Software may service the I2C I/O in the main program by monitoring the IBIF bit if the interrupt function is disabled. Note that polling should monitor the IBIF bit rather than the TCF bit since their operation is different when arbitration is lost.

Note that when a “Transfer Complete” interrupt occurs at the end of the address cycle, the master will always be in transmit mode, i.e. the address is transmitted. If master receive mode is required, indicated by R/W bit sent with slave calling address, then the Tx/Rx bit at Master side should be toggled at this stage. If Master does not receive an ACK from Slave, then transmission must be re-initiated or terminated.

In slave mode, IAAS bit will get set in IBSR if Slave address (IBAD) matches the Master calling address. This is an indication that Master-Slave data communication can now start. During address cycles (IAAS=1), the SRW bit in the status register is read to determine the direction of the subsequent transfer and the Tx/Rx bit is programmed accordingly. For slave mode data cycles (IAAS=0), the SRW bit is not valid. The Tx/Rx bit in the control register should be read to determine the direction of the current transfer.[7]

Transmit/receive sequence

Follow this sequence in case of Master Transmit(Address/Data):

1. Clear IBSR[IBIF].
2. Write data in Data Register (IBDR).
3. IBSR[TCF] bit will get cleared when transfer is in progress.
4. IBSR[TCF] bit will get set when transfer is complete.
5. Wait for IBSR[IBIF] to get set, then read IBSR register to determine its source:
    - TCF = 1 i.e. transfer is complete.
    - No Acknowledge condition (RXAK = 1) is found
    - IBB = 0 i.e. Bus has transitioned from Busy to Idle state.
    - If IBB = 1, ignore check of Arbitration Loss (IBAL = 1).

- Ignore Address Detect (IAAS = 1) for Master mode (valid only for Slave mode).

6. f) Check RXAK in IBSR for an acknowledge from slave.

Follow this sequence in case of Slave Receive(Address/Data):

1. Clear IBSR[IBIF].
2. IBSR[TCF] will get cleared when transfer is in progress for address transfer.
3. IBSR[TCF] will get set when transfer is complete.
4. Wait for IBSR[IBIF] to get set. Then read IBSR register to determine its source:
   — Address Detect has occurred (IAAS = 1) - determination of Slave mode.
5. Clear IBIF.
6. Wait until IBSR[TCF] bit gets cleared (that is, "Transfer under Progress" condition is reached for data transfer).
7. Wait until IBSR[TCF] bit gets cleared(proof that Transfer Completes from "Transfer under Progress" state).
8. Wait until IBSR[IBIF] bit gets set. To find its source, check if:
   — TCF = 1 i.e. reception is complete
   — IBSR[IBB] = 0, that is, bus has transitioned from Busy to Idle state — Ignore Arbitration Loss (IBAL = 1) for IBB = 1
   — Ignore No Acknowledge condition (RXAK = 1) for receiver
9. Read the Data Register (IBDR) to determine data received from Master.

Sequence followed in case of Slave Transmit (Steps 1–4 of Slave Receive for Address Detect, followed by 1–6 of Master Transmit for Data Transmit).
Sequence followed in case of Master Receive (Steps 1–6 of Master Transmit for Address dispatch, followed by 5–8 of Slave Receive for Data Receive).[7]

Generation of STOP

A data transfer ends with a STOP signal generated by the 'master' device. A master transmitter can simply generate a STOP signal after all the data has been transmitted.

If a master receiver wants to terminate a data transfer, it must inform the slave transmitter by not acknowledging the last byte of data which can be done by setting the transmit acknowledge bit (TXAK) before reading the 2nd last byte of data. Before reading the last byte of data, a STOP signal must first be generated.

Generation of repeated START

At the end of data transfer, if the master still wants to communicate on the bus, it can generate another START signal followed by another slave address without first generating a STOP signal.

2.3 Main bus communication protocol

Due to the high number of Electronics modules in the vehicle, there are some communication protocol exist strongly in the market like LIN, CAN, FlexRay, Ethernet and others.

CAN and LIN are among the most popular communication protocols in-vehicle and have existed as standardized. It was expected that LIN and CAN communication networks will continue to dominate vehicle communication over a few decade. FlexRay is in market from last decade and it is getting popular. Ethernet and MOST are demand for today's OEM as tomorrow's technology. Those protocols needs to be more standardized and needed to be enhancement for automotive communication.

2.3.1 LIN

Local interconnect network (LIN) is widely known as one of the low cost automotive networks that complements the existing automotive multiplex networks. It is a serial communication protocol. LIN is one of the most important factor for the implementation of a vehicle network in order to have further quality, enhancement and cost reduction of vehicles.

The standardization which have been done rreducing the existing of low-end multiplex solutions and also cut the cost of development, service , production and logistics in vehicle electronics. This network is built based on master-slave architecture. One of the network node is chosen to control all communication happened and this is called as LIN master.

LIN master does the role of bus arbiter. LIN master sends message in the form of token. Then, the token understood as a request by LIN slave. Slave can reply to token in three ways i.e. it can reply by send the data, receive data or ignore data. Token and the data together form the LIN message. Up to 64 messages might be defined in the local interconnect network. Each message of them is available to be received by any node or even by a master if it has slave task. Though, LIN is known as centrally controlled message distribution system. [6]

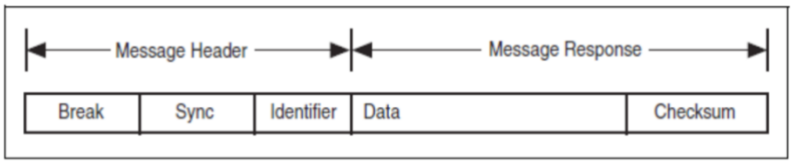
Main properties of the LIN network are:

- Single-master / multiple-slave concept
- Low cost silicon implementation based on common UART/SCI interface hardwar
- Self-synchronization without quartz or ceramics resonator in the slave nodes
- Deterministic signal transmission
- Low cost single-wire implementation
- Speed up to 20kbit/s.

LIN message format

LIN Frame format – The frame format is spitted into two parts, i.e. message header and message response.The message header consists of sync, break, and identifier in the meantime the message response contains the data and checksum. The token is referred to message header. The message header is always sent from master task. The master header is used for synchronization.and the message response is sent by slave task.

The below figure illustrate the LIN frame format.



2.10 LIN frame format

Break: LIN frame begins with break, that has 13 dominant bits (nominal bits) followed by a break delimiter that is of one bit (nominal) recessive. This is part called as a start-of-frame which is notice to all nodes on the bus.

Sync - The sync part is the second field which is transmitted by the master task in the header. The sync field lets the slave devices which perform automatic baud rate detection to calculate the period of the baud rate and adjust their internal baud rates so they can synchronize with the bus.

ID: This field is the final one which is transmitted in the header part with the help of master task. This field gives identification for each message sent on network. It also determines which nodes will respond to each transmission. A slave task usually listens for frames IDs andmake sure that their respective parties and determines if they are publisher or subscriber. LIN provides 64 Ids, divided as from 0 to 59 IDs used for signal carrying frames. And, ID 60 and 61 used for diagnostic data,and the 62 is used for user-defined extensions while 63 is kept reserved for future protocol enhancements.

Data Bytes: Data bytes part usually sent by the slave task in the response. This field consists from one to eight bytes as a payload data bytes.

Checksum: This is transmitted by slave task in the response. LIN bus use of one of two checksum algorithms which is used to calculate and check the value in the eight-bit checksum field. Classic checksum is calculated by summing the data bytes alone, on the other hand enhanced checksum is calculated by summing the data bytes and the protected ID.[6]

2.3.2 Control Area Network (CAN)

The Controller Area Network was developed by Robert Bosch GmbH for automotive applications in the early 1980s and publicly released in 1986. The Bosch CAN specification became an ISO standard (ISO 11898) in 1993 (CAN 2.0A), and extended in 1995 to permit longer device identifiers (CAN 2.0B) [8]. Typically, CAN interconnects a network of modules (or nodes) using two wire, twisted pair cable.

Many companies implement CAN devices. In the Freescale MPC 5xx series of processors, the CAN device is called the TouCAN module; in the MPC 55xx series it's called FlexCAN. CAN is a serial, multimaster which means that when the bus is free, any node can send a message (multimaster), and all nodes may receive and act on the message (multicast).

The node that initiates the message is called the transmitter; any node not sending a message is called a receiver. Messages are assigned static priorities, and a transmitting node will remain a transmitter until the bus becomes idle or until it is superseded by a node with a higher priority message through a process called arbitration. A CAN message may contain up to 8 bytes of data. A message identifier describes the data content and is used by receiving nodes to determine the destination on the network. Bit rates up to 1 Mbit/s are possible in short networks (≤ 40 m). Longer network distances reduce the available bit rate (125 kbit/s at 500 m, for example). “High speed” CAN is considered to be 500 kbit/s.

Layered Architecture ofCAN

The following figure shows the layered architecture of CAN according to OSI reference model.

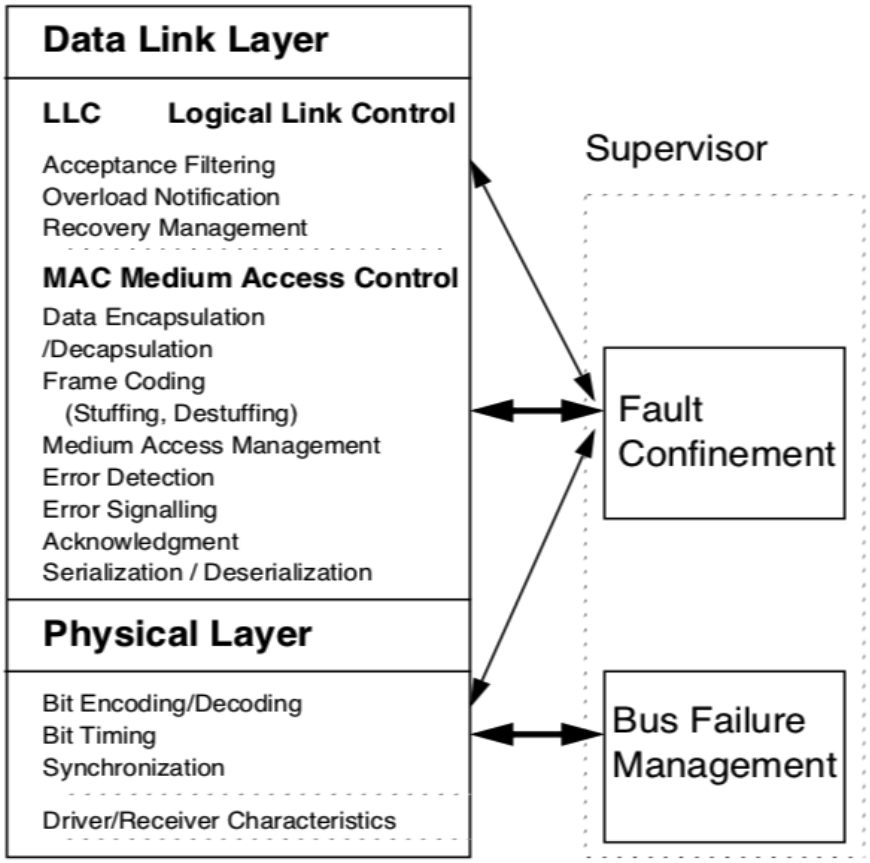


Figure 2.11 Layered Architecture of CAN according to the OSI Reference Model

Physical Layer - It defines how signals are actually transmitted on the bus. Hence it specifies the description of Bit Timing, Bit Encoding, and Synchronization.

Data Link Layer - DLL is divided into two sub-layers; first one is Media Access Control (MAC) and other one Logical Link Control (LLC).

Media Access Control - The MAC sub-layer represents the core of the CAN protocol. It presents messages received from the LLC sub-layer and accepts messages to be transmitted

to the LLC sub-layer. The MAC sub layer is responsible for Message Framing, Arbitration, Acknowledgement, Error Detection and Signaling. It's supervised by a management entity called Fault Confinement which is self-checking mechanism for distinguishing short disturbances from permanent failures.

Logical Link Control - The LLC sub-layer is concerned with Message Filtering, Overload Notification and Recovery Management.

Message Structure

There are four types of CAN messages or "frames" Data Frame, Remote Frame, Error Frame and Overload Frame. The data frame is the standard CAN message, broadcasting data from the transmitter to the other nodes on the bus.

A remote frame is broadcast by a transmitter to request data from a specific node. An error frame may be transmitted by any node that detects a bus error.

Overload frames are used to introduce additional delay between data or remote frames. CAN 2.0A and 2.0B data frames are illustrated in blow Figure and in Tables .

The difference between a CAN 2.0A and a CAN 2.0B message is that CAN 2.0B supports both 11 bit (standard) and 29 bit (extended) identifiers. Standard and extended frames may exist on the same bus, and even have numerically equivalent identifiers. In this case, the standard frame will have the higher priority.

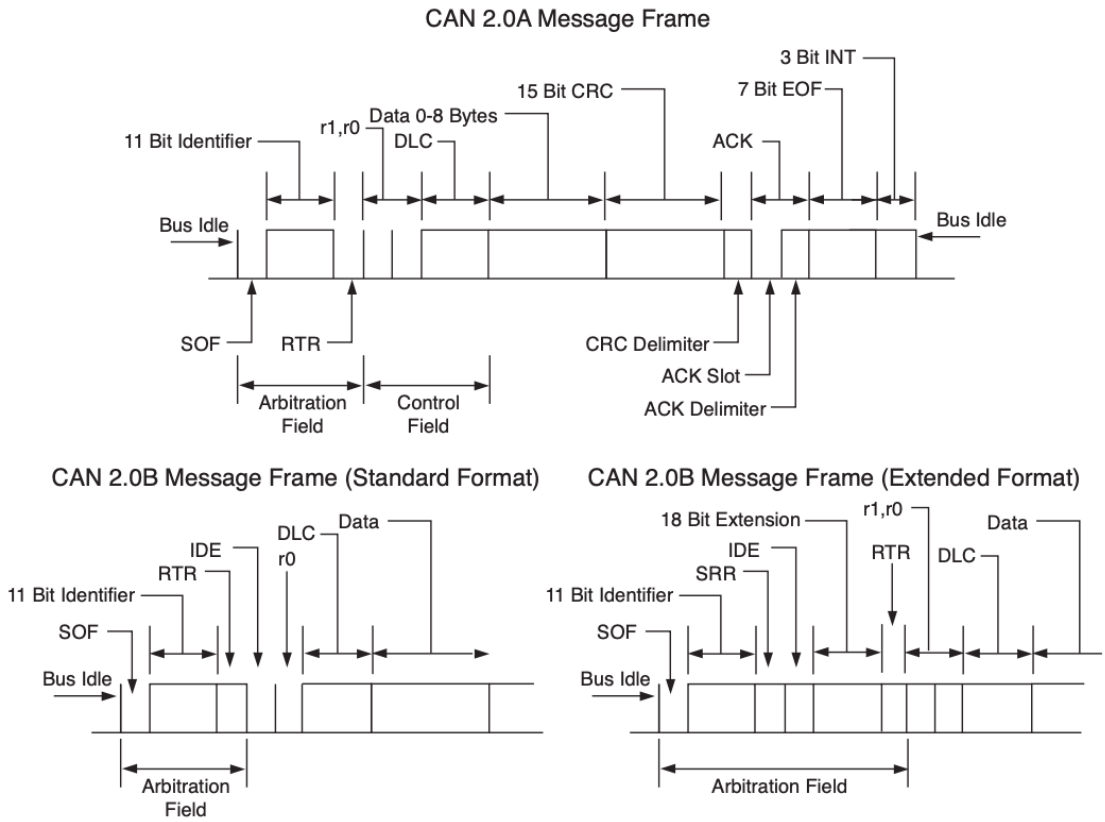


Figure 2.12 Can message format

Field	Length (bits)	Description
Start of Frame (SOF)	1	Must be dominant
Identifier	11	Unique identifier indicates priority
Remote Transmission Request (RTR)	1	Dominant in data frames; recessive in remote frames
Reserved	2	Must be dominant
Data Length Code (DLC)	4	Number of data bytes (0–8)
Data Field	0–8 bytes	Length determined by DLC field
Cyclic Redundancy Check (CRC)	15	
CRC Delimiter	1	Must be recessive
Acknowledge (ACK)	1	Transmitter sends recessive; receiver asserts dominant
ACK Delimiter	1	Must be recessive
End of Frame (EOF)	7	Must be recessive

Table 2.3 CAN 2.0A Message Frame

Field	Length (bits)	Description
Start of Frame (SOF)	1	Must be dominant
Identifier – Standard and Extended Formats	11	Unique identifier corresponds to Base ID in Extended Format
Identifier – Extended Format	29	Comprised of 11 bit Base ID and 18 bit Extended ID
Remote Transmission Request (RTR) – Standard and Extended Formats	1	Dominant in data frames; recessive in remote frames. In Standard Format, the 11 bit identifier is followed by the RTR bit.
Substitute Remote Request (SRR) – Extended Format	1	Must be recessive. SRR is transmitted in Extended Frames at the position of the RTR bit in Standard Frames. In arbitration between standard and extended frames, recessive SRR guarantees the standard message frame prevails.
IDE – Standard and Extended Frames	1	Must be recessive for Extended Format; dominant for Standard Format.
Reserved r0 – Standard Format	1	Must be dominant
Reserved r1, r0 – Extended Format	2	Must be recessive
Data Length Code (DLC)	4	Number of data bytes (0–8)
Data Field	0–8 bytes	Length determined by DLC field
Cyclic Redundancy Check (CRC)	15	
CRC Delimiter	1	Must be recessive
Acknowledge (ACK)	1	Transmitter sends recessive; receiver asserts dominant
ACK Delimiter	1	Must be recessive
End of Frame (EOF)	7	Must be recessive

Table2.4 CAN 2.0B Message Frame

CAN Data Frame

It-s composed of seven fields as explained above in the table. All network nodes waiting to transmit synchronize with the SOF and begin transmitting at the same time. An arbitration scheme determines which of the nodes attempting to transmit will actually control the bus.

Arbitration

The arbitration field of the CAN message consists of an 11- or 29-bit identifier and a remote transmission (RTR) bit. The CAN arbitration scheme is called “carrier sense multiple access with collision detection” or CSMA/CD, and assures that the highest priority message is broadcast.

Message priority is determined by the numerical value of the identifier in the arbitration field, with the lowest numerical value having the highest priority. Non-destructive, bit-wise arbitration resolves conflicts among competing transmitters.

This means that the bus can be thought of as acting like an AND gate: If any node writes a dominant (0) bit on the bus, every node will read a dominant bit regardless of the value written by that node. Every transmitting node always reads back the bus value for each bit transmitted. If a node transmits a recessive bit and reads back a dominant bit, it immediately stops transmitting.

The RTR bit simply distinguishes between data frames and remote frames. In data frames, the RTR bit must be dominant; in remote frames it must be recessive.

Error Handling

CAN implements five levels of error detection. At the message level, it performs cyclic redundancy checks(CRC), frame checks and acknowledgment checks. Bit level checks consist of monitoring and stuffing.

Cyclical redundancy errors are detected using a 15 bit CRC computed by the transmitter from the message content. Each receiver accepting the message recalculates the CRC and compares it against the transmitted value. A difference between the two calculations causes an error flag to be set.

Frame checks that will flag an error are the detection by a receiver of an invalid bit in the CRC delimiter, ACK delimiter, EOF or 3-bit interframe space.

Finally, each receiving node writes a dominant bit into the ACK slot of the message frame that is read by the transmitting node. If a message is not acknowledged (perhaps because the receiver has failed), an ACK error is flagged.

At the bit level, we have already noted that each transmitted bit is “read back” by the transmitter. If the monitored value is different than the value being sent, a bit error is detected.

Additionally, bit errors are detected by stuffing.After five consecutive identical bits have been transmitted, a bit of the opposite polarity will be inserted (“stuffed”) by the transmitter into the bit stream (bits are stuffed from the SOF through the CRC field). Receivers automatically “de-stuff” the message. If any node detects six consecutive bits of the same level, a stuff error is flagged.

In addition to error detection, bit stuffing assures that there are enough edges in the non-return to zero (NRZ) bit stream to maintain synchronization.

CAN Error Frame

If a transmitting or receiving node detects an error, it will immediately abort the transmission and broadcast an error frame consisting of an error flag made up of six dominant bits and an error flag delimiter made up of eight recessive bits. Since this bit string violates the bit stuffing rule, all other nodes respond by transmitting error flags, too. After a sufficient number of errors are detected, a node will eventually turn itself off.

Robustness, especially in manufacturing and automotive environments where CAN is prevalent, requires that the network determine whether errors are transient (due to voltage spikes, noise or some other temporary condition) or permanent failure of the node due to defective hardware. Consequently, nodes store and track the number of errors detected.

A node may be in one of three modes depending on the error count: If the count in either the transmit or receive buffer of a node is greater than zero and less than 128, the node is considered “error active,” indicating that, although the node remains fully functional, at least one error has been detected.

An error count between 128 and 255 puts the node in “error passive” mode. An error passive node will transmit at a slower rate by sending 8 recessive bits before transmitting again or recognizing the bus to be idle.

Error counts above 255 will cause the node to enter “bus off” mode, taking itself off-line.

Receive errors increment the error count by 1; transmit errors increment the count by 8. Subsequent error-free messages decrement the error count by 1. If the error count returns to zero, a node will return to normal mode. A node in the bus off condition may become error active after 128 occurrences of 11 consecutive recessive bits have been monitored. A message is considered valid by the transmitter if there is no error until the EOF. Corrupted messages are automatically retransmitted as soon as the bus is idle.

CAN Remote Frame

A node that requires data from another node on the network can request a transmission by sending a Remote Frame. For example, the microprocessor controlling the central locking on your car may need to know the state of the transmission gear selector from the powertrain controller (is the car in “park?”). A remote frame is the same as a data frame, without the data field (with the RTR bit recessive).

CAN Overload Frame

If a CAN node receives messages faster than it can process them, then an Overload Frame will be generated to provide extra time between successive Data or Remote frames. Similar to an Error Frame, the Overload Frame has two fields: an overload flag consisting of six dominant bits, and an overload delimiter consisting of eight recessive bits. Unlike error frames, error counters are not incremented.

The Interframe Space consists of a three recessive bit Intermission and the bus idle time between Data or Remote Frames. During the intermission, no node is permitted to initiate a transmission (if a dominant bit is detected during the Intermission, an Overload Frame will be generated). The bus idle time lasts until a node has something to transmit, at which time the detection of a dominant bit on the bus signals a SOF.

2.3.3 CAN FD

CAN FD (Controller Area Network Flexible Data-Rate) is an extension to the original CAN bus protocol that was specified in ISO 11898-1.Developed in 2011 and released in 2012 by Bosch, CAN FD was developed to meet the need for increasing the data transfer rate up to 5 times faster and with larger messages size in the Automotive Electronic Control Unit (ECU)s of modern vehicles. As in the classic CAN, CAN FD protocol is designed to transmit sensor data, control commands and communication errors, between electronic sensor devices and microcontrollers.

Hence, Both CAN and CAN FD are almost the same, they share a lot of specification between each other's. Like, Arbitration, Error handling, error detection and more. We will highlight the significant differences between them.

Why CAN FD

The bandwidth requirement of the new automotive applications has been increasing gradually. This is mainly due to the variety, volume and Velocity of data from sensors and other sources being fed to in the in-vehicle network of control units. Automotive ECU reprogramming is another area where there is large size binary files are required to be transferred over the in-vehicle network.

The bit rate and payload limitation of CAN started preventing activities like automotive ECU flashing, and faster communication for ADAS applications.

Message structure

There are two types of CAN frames in the CAN FD protocol. One of them is the basic format and the other on is the extended format. The below figure shows the anatomy of both messages.

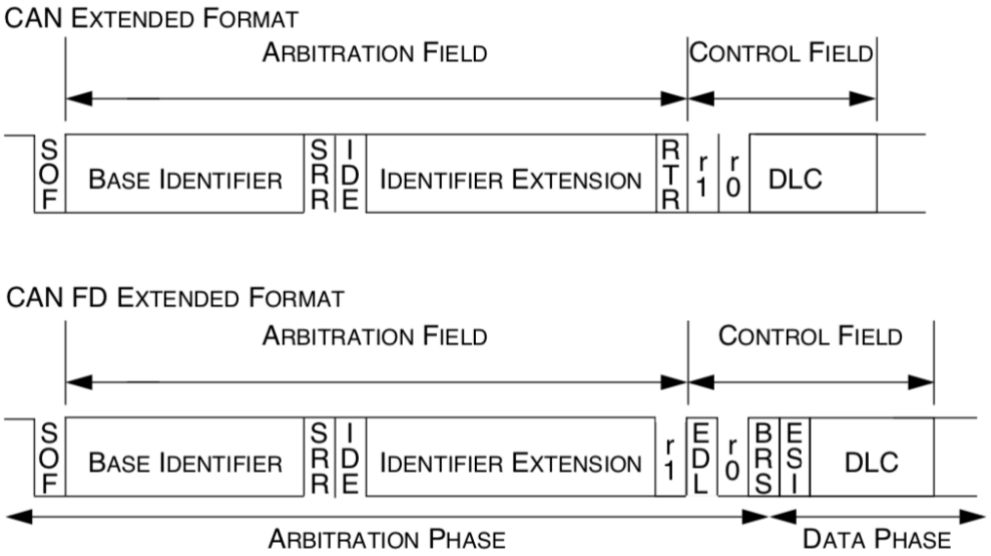


Figure 2.13 CAN FD packet-s anatomy

Classical CAN Vs CAN FD

Some of the differences which are also the main advantages are quite obvious. It includes the higher bit rate and support for larger payload.

1. **Increased data rate:** From a max of 1 Mb/s to up to 8 Mb/s, the change is glaring. Increased data rate is highly beneficial in re-programming the applications like ADAS (Advanced Driver Assistance System), as large data packets are needed to be transmitted.
2. **Large Payloads**: CAN FD supports 64 bytes of data field as compared to 8 bytes in CAN. This huge leap in payload size translates into faster and more efficient in-vehicle network communication among the vehicle ECUs. End-of-line software upgrade also becomes faster affair with CAN FD.
3. **Message Format**: The message frame format has undergone some crucial changes in CAN FD bus standard. The most primary being the ability to send the control data (arbitration and acknowledgment) with a different bit rate (usually 500 Kb/s) and send the actual data at a higher bit rate.
4. **Data bit rate dependency**: Propagation delay is quite common in CAN bus standard due to the transceivers and cable length. Propagation time is nothing but the time required to send the signal to the most distant node and get it back. The bit-rate will directly depend on the propagation delay as there is a reciprocal relation between them. There is no such dependence of bit rate on signal propagation delay.

The Compatibility of CAN FD and Classical CAN

A Classical/Classicalal CAN controller will not be compatible with CAN FD, however, the vice-versa is true. Backward compatibility very much exists between CAN FD and classical CAN. It implies that both CAN and CAN FD nodes can be used together as CAN FD is backward compatible with Classical CAN. However, special hardware would be required.

Following are some of the reasons why CAN and Can FD backward compatibility is possible:

- Dependence of data bit-rate on transmission characteristics and physical layer and not on propagation delay.
- Some of the compatibility issues can be resolved by the transceivers that feature passive partial networking and are both CAN and CAN FD tolerant.

2.3.4 Socket CAN

The socketcan package is an implementation of CAN protocols (Controller Area Network) for Linux. While there have been other CAN implementations for Linux based on character devices, SocketCAN uses the Berkeley socket API, the Linux network stack and implements the CAN device drivers as network interfaces. The CAN socket API has been designed as similar as possible to the TCP/IP protocols to allow programmers, familiar with network programming, to easily learn how to use CAN sockets.

Why Socket API

There have been CAN implementations for Linux before SocketCAN so the question here is, why we have started another project. Most existing implementations come as a device driver for some CAN hardware, they are based on character devices and provide comparatively little functionality. Usually, there is only a hardware-specific device driver which provides a character device interface to send and receive raw CAN frames, directly to/from the controller hardware. Queueing of frames and higher-level transport protocols like ISO-TP have to be implemented in user space applications. Also, most character-device implementations support only one single process to open the device at a time, similar to a serial interface. Exchanging the CAN controller requires employment of another device driver and often the need for adaption of large parts of the application to the new driver's API.

SocketCAN was designed to overcome all of these limitations. A new protocol family has been implemented which provides a socket interface to user space applications and which builds upon the Linux network layer, enabling use all of the provided queueing functionality. A device driver for CAN controller hardware registers itself with the Linux network layer as a network device, so that CAN frames from the controller can be passed up to the network layer and on to the CAN protocol family module and also vice-versa. Also, the protocol family module provides an API for transport protocol modules to register, so that any number of transport protocols can be loaded or unloaded dynamically.

Socket CAN concept

As discussed before the main goal of SocketCAN is to provide a socket interface to user space applications which builds upon the Linux network layer. In contrast to the commonly known TCP/IP and ethernet networking, the CAN bus is a broadcast medium that has no MAC-layer addressing like ethernet. The CAN-identifier (can_id) is used for arbitration on the CAN-bus. Therefore the CAN-IDs have to be chosen uniquely on the bus. When designing a CAN-ECU network the CAN-IDs are mapped to be sent by a specific ECU. For this reason a CAN-ID can be treated best as a kind of source address.

Linux networking subsystem Vs Socket CAN

First, The Linux networking subsystem, is widely known as the implementation of the TCP/IP protocol suite, and it is highly flexible. It contains several other networking protocols like ATM, ISDN and the kernel part of the official Linux stack. The figure in the right shows the network subsystem.

Starting from the application level, there is the standard POSIX socket API showing the interface to the kernel. Underneath there is the protocol layer, consisting of protocol families (here PF_INET) which implement different networking protocols. Within each family you can notice several protocols (here like TCP and UDP).

Below this level there is the routing and packet scheduling layer exists then followed by the last one which consists of the drivers for the networking hardware.

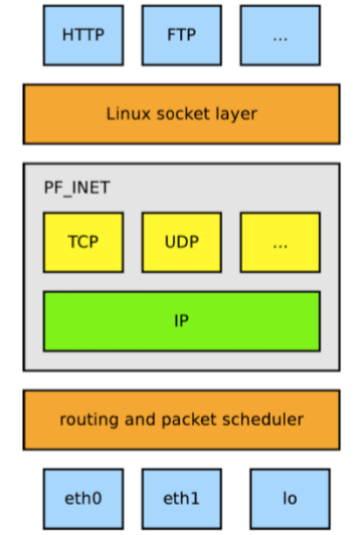


Figure 2.14 Linux network subsystem

In order to put the CAN networking to the Linux kernel, CAN support has been added to the existing networking subsystem. This primarily consists of two parts:

1. A new protocol family **PF_CAN** including a **CAN_RAW** protocol,
2. the drivers for various CAN networking devices.

This approach brings several design advantages over the earlier mentioned character-device based solutions:

- Taking benefit of the existing and established POSIX socket API to assist the application developer.
- The new protocol family is developed against established abstractions layers, the socket layer above and the packet scheduler below.
- CAN network device drivers implement the same standardized networking driver model as Ethernet drivers.
- Communication protocols and complex filtering can be implemented inside the kernel.
- Support for multi-user and multi- application access to a single CAN interface is possible.

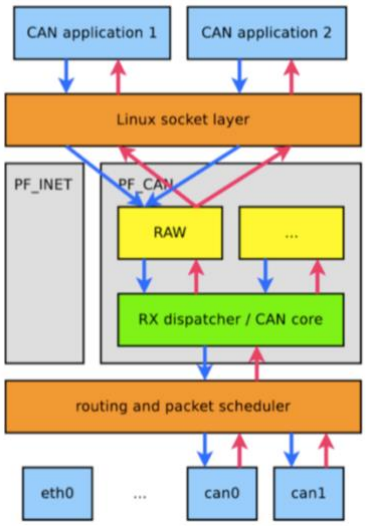


Figure 2.15 Linux network include CAN

Socket API Usage

It is like TCP/IP, you first need to open a socket for communicating over a CAN network. Since SocketCAN implements a new protocol family, you need to pass PF_CAN as the first argument to the socket system call. Currently, there are two CAN protocols to choose from, the raw socket protocol and the broadcast manager (BCM).

After the successful creation of the socket, you would normally use the bind system call to bind the socket to a CAN interface (which is different from TCP/IP due to different addressing). After binding (CAN_RAW) or connecting (CAN_BCM) the socket, you can read and write from/to the socket or use send, sendto, sendmsg.

2.3.5 FlexRay

FlexRay is an automotive network communication protocol developed by FlexRay Consortium. In 2000 BMW, Daimler-Chrysler, Philips and Motorola joined together to create a new automotive network scheme which would better adjust the increasing demand for advanced applications in automation. It was widely known as FlexRay.

FlexRay is designed to be faster and more reliable than CAN and TTP, but it is also more expensive. FlexRay is very flexible and fault tolerance communication protocol. It supports high data rate i.e. 10Mbit/s. It supports star and party line bus topology and it has two independent data channels which are used for fault tolerance.[9]

Basic architecture

FlexRay Cluster or FlexRay System consists of several electronic control units (ECU) each one of them with a bus interface connected to one or two communication channels. The bus works on a time cycle concept which is divided into two parts dynamic segment and the static segment. The static segment is reallocated into separate parts for individual communication types. It gives stronger real-time guarantee than its previous CAN bus. The dynamic segment operates more like CAN, with nodes taking control of the bus as available, allowing event-triggered behavior.

The below figure explains the basic building blocks of FlexRay.It shows CHI (controller host interface) ,Media access control, POC (protocol operation control), clock sync and frame/symbol processing. Controller host interface exists between host and microcontroller. It handles configuration and also the status data. Protocol operation control reacts to protocol conditions and host commands. MAC schedules the bus and writes accesses. It assemblesn the message header. Frame and symbol processing handles received messages and executes timing and error check.

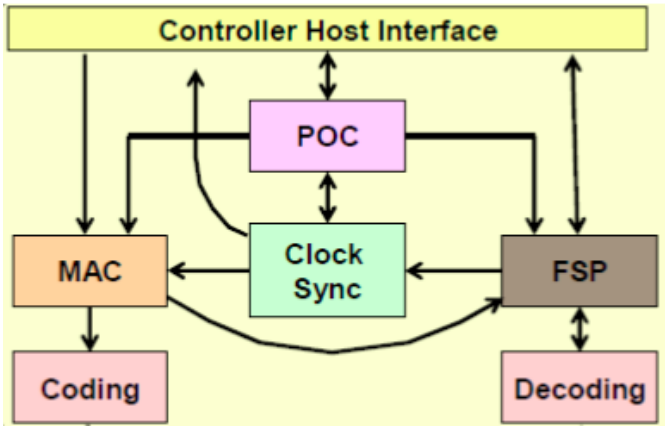


Figure2.16 basic building block of FlexRay

Communication Cycle

The communication cycle consists of network idle time, static segment, dynamic segment, symbol window. The below figure shows the anatomy of the cycle.

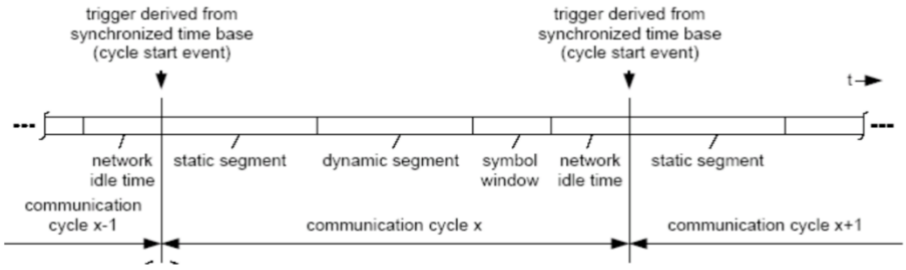


Figure2.17 the communication cycle

Static Segment - Communication cycle consist of static segment which has configurable no of static slots. Static slots have identical number of macroticks. Static segment is divided into static slots of equal duration. Slot duration defined as per cluster. Both channels have identical slot timings. Slot has slot ID. For each node, specific time slot is given. 16 slots can be assignable per node. If there is no frame configured then the slot will remain empty.

Dynamic Segment - This segment is optional. It contains configurable number of minislots. All minislots has same no of macroticks. At the start of dynamic segment, all slot counters set to zero. The node can send a message when it has message with ID matching with the slot counter. Once the message sent, all nodes detect a new dynamic slot. If there is no message transmitted all the slot counters increase again. Low value ID gives high priority of being transmitted. Dynamic slot is variable in size.

Symbol window - This is optional segment. It contains macroticks. A single symbol can be sent within a symbol window. Arbitration among different senders is not provided by protocol for the symbol window. If arbitration required for symbol window then it can be achieved with the help of higher level protocol.

Network idle time - Communication cycle contains network idle time. Node calculates and applies clock correction terms during this network idle time.

Frame structure

The whole structure is showed in the figure below. In the beginning there are some separate bits. Each one of them has function.

Reserved bit: For a transmitting node, reserved bit should set to logical '0' and for a receiving node no need to set reserved bit.
Payload preamble indicator: In the static segment frame, this bit indicates the presence of a NM vector at the beginning of the payload and in the dynamic segment frame; this bit indicates the presence of a message ID at the beginning of the payload.
Null frame indicator: If the payload segment contains no valid data then this bit is set to zero otherwise set to one if the payload segment contains data.

Sync frame indicator: This bit is set to zero then no receiving node shall consider the frame for synchronization. If this bit is set to one then all receiving nodes shall utilize the frame for synchronization if it meets other acceptance criteria.
Startup frame indicator : The startup frame indicator set to one in the sync frames of coldstart nodes. Every coldstart node can transmit exactly one frame per communication cycle and channel with the startup frame indicator set to one.

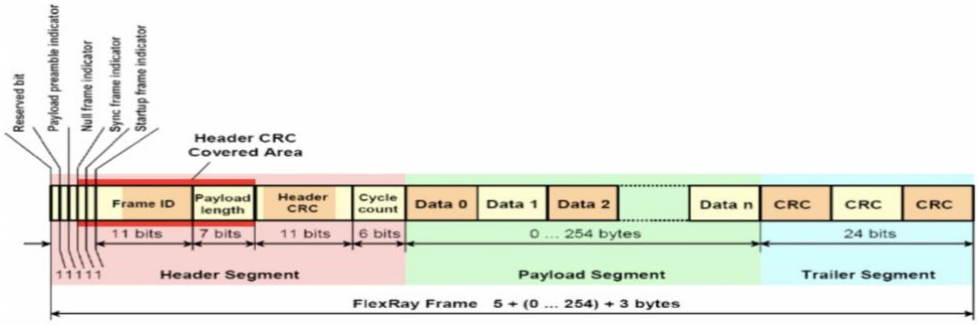


Figure2.18 FlexRay's frame anatomy

Frame ID - A frame ID used once on each channel in a communication cycle. It ranges from 1 to 2047.
Payload Length - It is 7 bits. For static segment payload length is fixed and it is different for dynamic segment.
Header CRC - It has 11 bits. This is calculated over sync frame indicator, startup frame indicator, frame ID and payload length. For a transmitted frame CC does not calculate header CRC. Header CRC of transmitted frames are calculated offline and then it gives to CC by means of configuration. CCC calculate header CRC of a received frame to check whether CRC is correct or not.
Payload segment (0 - 254 bytes) - First 0 to 12 bytes of the payload segment may optionally be used as NM vector in static segment frame transmission. In dynamic segment frame transmission, first two bytes of payload segment may optionally use as message ID field. NM Vector (optional) - The length of NM Vector shall be configured to the same value for all nodes in a cluster. At the transmitting node the NMVector is written by the host as application data.
Message ID - It is 16 bit and optional. In dynamic segment frame, first two bytes of payload is used as receiver filterable data. Message ID is written at transmitting node with the help of host application data.
FlexRay trailer segment - It contains 24 bit CRC for the frame. Frame CRC is computed over header segment and payload segment of frame.

Chapter Three

System architecture

In this chapter we will discuss the whole system architecture and communication network between the microprocessors and the actuators. Also, we will discuss the communication layer between the two microprocessors and all the data handling inside each block.

Mainly we will explain the network static architecture and the communication sequence diagram of the system.

3.1 Hardware architecture

The ECU application processor operating system based on Linux using as much standard open source software as possible. The ECU architecture is based on 3 processors .Mainly all our work is between the Processor and Companion processors and we will refer to them as the main processor which is the IMX6 and the Companion processor which is SPC as CCPU.

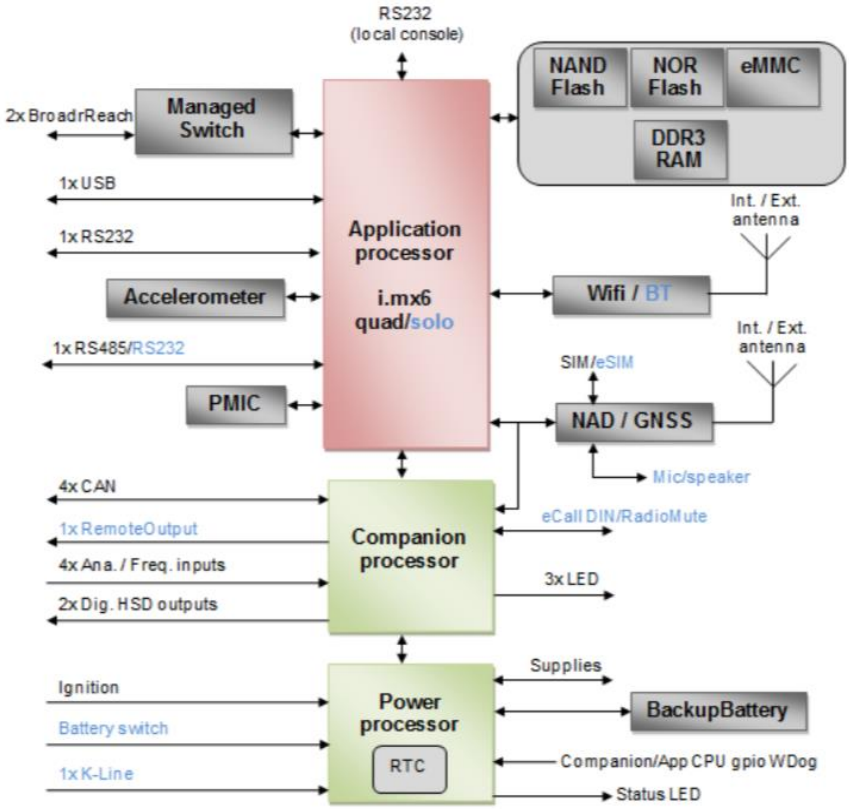


Figure 3.1 Hardware architecture

The basic architecture depends on 3 processor the main point that we are focusing on in this work is forwarding the data from the CCPU to the IMX6 over an SPI.
the CCPU has 4 CAN interface which they are receiving the data from the whole system and after these data is forwarded over the SPI to the IMX6.Also the CCPU forward the RTC (Rea time clock) received from the third processor over the I2C to the IMX6 over the SPI bus.
The below figure shows the communication between the three processors.

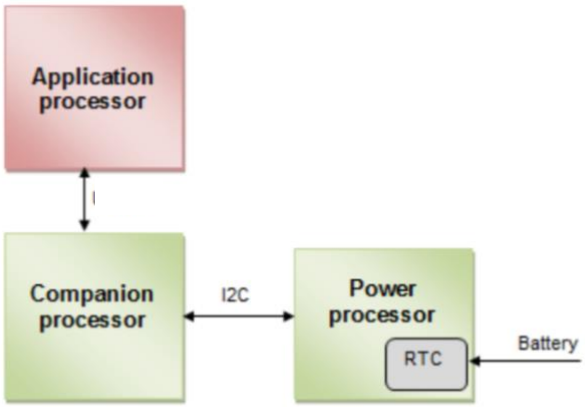


Figure 3.2 Communication between the 3 processors

Also, when we need to open a CAN socket in the CCPU it should be done from the IMX6 to the CCPU by an interface packet which give the command to open/close the Socket and after that the CCPU should response with the status of the socket. Also, the IMX6 can send command to know the status of the socket in the CCPU. Moreover, after every sent packet the receiver should check if he needs to reply by an acknowledge message or not and after that send it. All this communication is only related to the RAW packets but the whole system is could send and receive packet by ISOTP, BCM and J-1939 and those are not in our area of interest. So basically, we are only interested in the RAW CAN packets.

3.2 Static architecture

In this point we will explain the whole communication layers and blocks between the IMX6 and the CCPU starting from the CAN interface till the socket API in the IMX6.the below figure show all the blocks.

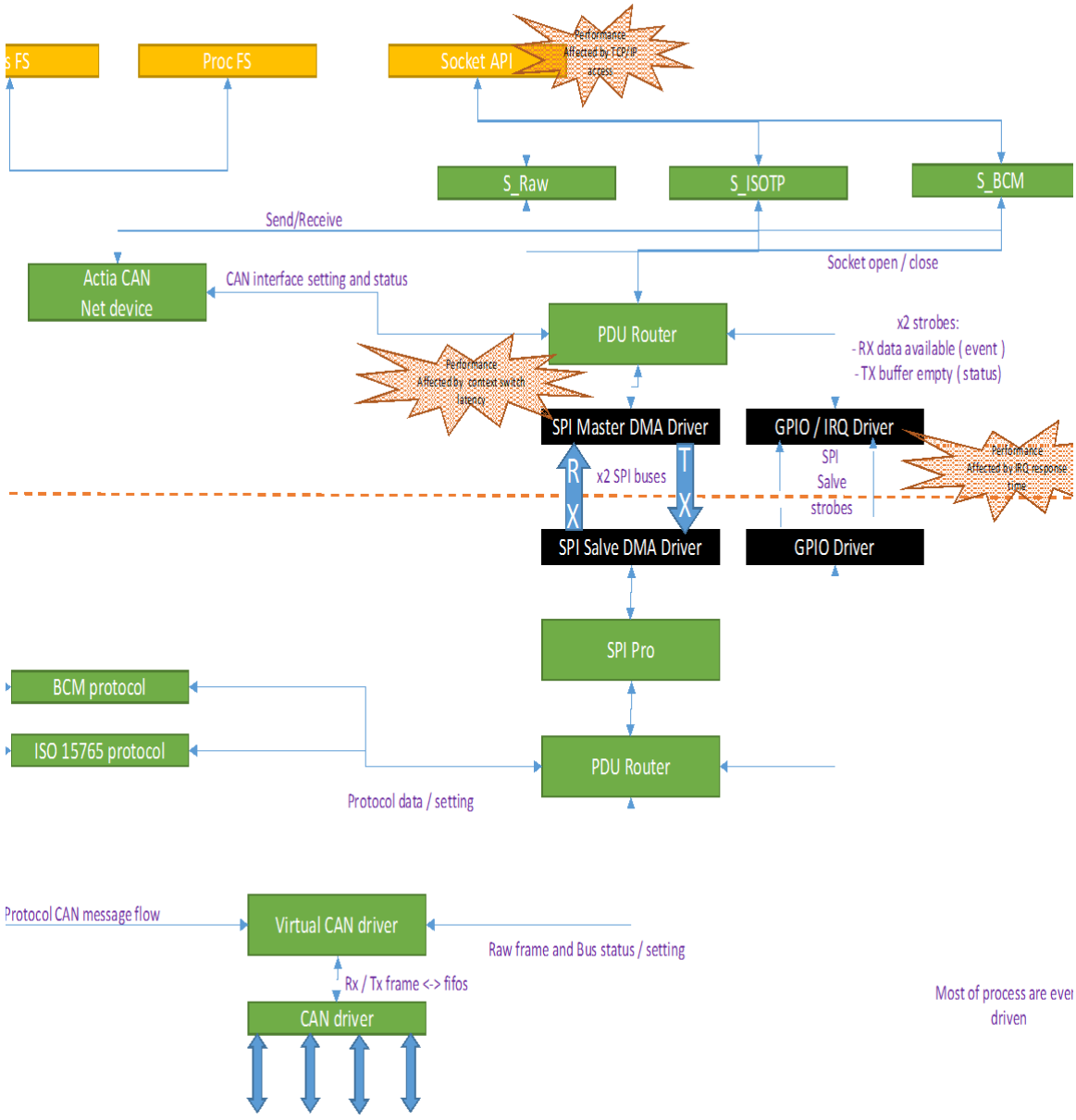


Figure 3.4 Static architecture

As we can see from the previous diagram that the communication layer is divided in different blocks. We can call them stages.
From the receiving point the packet is received by the CAN driver from the system after that it forwarded to the virtual CAN driver because we are only interested in the RAW packages to it handed to the PDU router where it has been processed depending on it type.
If it a data type so it contains important information regarding to the CAN ID and DLC and the payload of the packet. It is the important information handed from the virtual CAN driver after checking the packet's CRC and the detect all recessive bits also the RTR bit all this type of information used during the CAN bus. The PDU router starts to add the its own header data like the ID number and the packet type also add the it is own packet length and send also the channel ID to define which can interface received this information, socket ID and the protocol ID to define if it is RAW , BCM or ISOTP.

There is also other type of information could be sent it is about the interface status if it is ready or there is some error happened on it. This packet called interface receive status. It has two type of information, first if the interface is ready so it sends the interface ID, the status of the interface and the type of the information. Second if there is an error so it sends the interface ID, the setup of the interface, the type of error (receive error or transmit error) and the what is the error. All this data in the two cases is accompanied by the packet ID and the packet type.

Also, the system packet which is received from the third processor and has the RTC after received by I2C driver it is delivered to the PDU router and then add the ID, packet type, the length of packet in PDU router and the payload ID which recognize the payload.

The last packet which is sent depend on the status of the received the packets from the SPI is the acknowledge packet. This packet has its ID, type and the status of the acknowledge.

After all this data is gathered and handled in the PDU router they passed to the FIFO register to be handled by the SPI driver. In the stage we should take into account the data we have in only in the FIFO. These data now are being prepared in the shape of the ISO 15765-2 where the data is shaped depending on its length as single frame and send it or first frame followed by consecutive frame. With each frame added the header of the packet which is the packet number and the CRC data which is calculated in this stage after data added in the PDU router. After that these data is delivered to DMA driver to be sent over the SPI bus.

Then these data are received in the by the DMA driver in the IMX6 and being handled by the SPI driver to calculate the CRC and check if it is right or not to processed by the IMX6 or dumped and request to send it again. Then the data is passed to the PDU router in the IMX6 to be handled depending on the type added by the PDU router in the CCPU.

After that all the data is being transferred to the net device which have all the information related to the communication happened from/to the IMX6 and all the peripherals connected to.

At the end the data may be sent to one of the socket API of the IMX6 depending on it RAW, BCM or ISOTP.

In the other direction there are also packets sent to the CCPU. One of those packets is the IOCTL which has the information to open and setup the CAN interfaces of the CCPU. This packet is sent from the net device has the setup data, the channel ID which is the interface ID. Then there is other information added to the packet in the PDU router like the IOCTL ID , the packet length and of course the packet number and the packet type.

The Second packet is the interface packet is sent from the net device with the necessary information like the channel ID and the setup data after that the packet ID and the packet type added by the PDU router.

After that all this data is forward to the SPI driver to be handled in the same way like ISO 15765-2 as single frame or first frame followed by the consecutive frame depending on the length of the data received from the PDU router during this handling the SPI calculate the CRC and add the SPI packet ID to be handed to the DMA and sent over the bus.

Then, the DMA in the CCPU receive the data and calculate the CRC to check if there is any error happened during the sending and decide to use it or dump it and request the packet to be sent again. After that the packet is passed to the PDU router which handle the data and process the information related to each channel.

3.3 Communication analysis

3.3.1 PDU Router data handling

There are many types of the information to be exchanged between the IMX6 and the CCPU in both ways. We have four types of frames or information to be sent from the CCPU to IMX6 and three different types of frame or information to be sent from IMX6 to the CCPU. We will discuss every and each frame of them.

Communication from CCPU to IMX6

First, we will discuss the communication from the CCPU to the IMX6 as it is about 90% of the data flow between both processors. One of the most important frames is the send frame. Which has the CAN bus information. After the data received from the CAN bus and passed by the CAN driver and virtual CAN driver it delivered to the the PDU router. Before received by the PDU router the can data is being handled in the way of checking the CRC and removing the bit stuffing and everything and deliver only the ID (4 bytes), DLC (1 byte), Payload (bytes) and time stamp (4 byte).

Then all this data delivered to the PDU router to be dealt with as a payload and the PDU router and its own data like protocol ID (1 byte), channel ID (1 byte), socket ID (2 byte) and the PDU router header which id packet ID (1 byte) and packet type (1 byte) and the the end the total length of this packet (2 byte). So, in total we have in total more (8 byte). After that the data is delivered to the SPI driver to be handled and sent.

The second used frames is the “interface frame received” which is basically sent after receive a command from the IMX6 to open, close or even change the baud rate of the CAN interface so the the CCPU answer by the status of conducting the command as if the interface is ready or there is an error and the type of this error in details .
As a general structure for the frame, it consists of the header for packet ID (1 byte) and packet type (1 byte), channel ID (1 byte), reserved (2 byte) and the then the type of the frame if it is ready frame or the error frame and depending on that the rest of the structure will be change but they are the same in the length which is more (6 byte).

The third type of frames is the acknowledge frame which is used as acknowledgement for each packet needed to be confirmed as per the system requirements.
So, when the PDU router received packet and it is needed to send an acknowledgement so the PDU router send the status of the acknowledgment (4 byte), packet ID (1 byte) and the packet type (1 byte) and in the same structure there is (4 byte) named as reserved.

The last frame is the system frame which has the RTC (Real time clock) as a payload (8 byte), payload ID (1 byte), reserved (3 byte), header (2 byte) for packet ID and type and the length of all this data (2 byte).

Communication from the IMX6 to the CCPU

In this direction of the communication we have only three types of frames sent over the SPI bus. First one of them is the interface frame which sent the information to open or close the CAN interface. It sends this information in (1 byte), the channel ID (1byte), reserved data (3 byte) and the PDU router header which is (2 byte), packet ID and packet type.

The second frame is IOCTL which is sent to configure the CAN interface. The payload has all the information needed to configure the interface maximum is (32 byte), the IOCTL ID (1 byte), channel ID (1 byte),reserved (2 byte) and the PDU router header (2 byte) packet ID and packet type, also the length of all this structure(2 byte).

The last frame is the acknowledge frame which is send by the acknowledge data with the same structure from the CCCPU to the IMX6.

3.3.2 SPI Data Handling

The communication between both processors in both directions is the same as there is a buffer with a size of 384 bytes used for sending and receiving the data over the SPI bus. Basically, the protocol used over the SPI communication is the same like ISOTP protocol “ISO 15765-2”.the protocol idea depend on sending single frame if the size is equal or less than the buffer size or first frame followed by consecutive frame if the size is bigger than the buffer size. The SPI header is the frame ID and CRC each one of them is 2 bytes so total 4 bytes.

When the buffer is ready it is sent using the DMA driver for better usage of CPU resources. When the packet is received the CRC checked and after that handled by the PDU router.

Chapter Four

Optimization Activity and Result

4.1 Method of optimization

As you discussed in the previous chapter, the data is handled in two different layers, SPI driver and PDU Router. So, our optimization activity will be done in these two different layers.

4.1.1 SPI optimization

In the SPI level we have added CRC as a technique of error detection to detect corruption of data that is stored in and/or transmitted between the two processors. Before that the system was using checksum, which is weaker than CRC in case there are more than one error.

The buffer size has been changed from 384 bytes to 64 bytes so we can send the data faster to be processed in smaller frame better than waiting for the 384 bytes. Also, the technique of sending and receiving the packet between the two processors is like ISOTP protocol “ISO 15765-2” so if the data is smaller or equal to the buffer size, the data will be sent as single frame with packet ID and CRC. But if the data is bigger than so it will be divided into first frame and multiple consecutive frames. When we should send a consecutive frame, we send its own CRC byte with it so it could be checked and processed the second it received and no need to wait for the whole data to check the CRC. In case of there is an error happened in the communication and the CRC is not matched, we have added an algorithm to resend the defected message again despite of sending the whole buffer again.

Now, we have a robust, fast and reliable communication protocol ready to handle all the data received from the PDU Router with minimum packet header.

4.1.2 PDU Router optimization

As we explained before there are four types of frames used in the communication between the IMX6 and the CCPU. We have optimized all the added header to each frame of them. The table below show all the optimization applied to the frames.

Frame type	Before optimization			After optimization		
	Type	Size	Total size	Type	Size	Total size
Send frame	• Header • Socket ID • Channel ID • Protocol ID • Length • Payload (Full bus)	2 byte 2 byte 1 byte 1 byte 2 byte 17 byte	25 byte	• Header • Socket ID • Channel ID • Protocol ID • Length • Payload (Full bus)	2 byte **0 byte** 1 byte **0 byte** **1 byte** **13byte**	17 byte
System Frame	• Header • Payload ID • Reserved • Length • Payload	2 byte 1 byte 3 byte 2 byte 8 byte	16 byte	• Header • Payload ID • Reserved • Length • Payload	2 byte 1 byte **0 byte** **1 byte** 8 byte	12 byte
Interface Receive Frame (Error case)	• Header • Reserved • Channel ID • Type • Error setup • RXERR • TXERR	2 byte 2 byte 1 byte 1 byte 2 byte 2 byte 2 byte	12 byte	• Header • Reserved • Channel ID • Type • Error setup • RXERR • TXERR	2 byte **0 byte** 1 byte **0 byte** 2 byte **1 byte** **1 byte**	7 byte
Interface Receive Frame (Ready case)	• Header • Reserved • Channel ID • Type • Ready flag • Reserved	2 byte 2 byte 1 byte 1 byte 1 byte 5 byte	12 byte	• Header • Reserved • Channel ID • Type • Ready flag • Reserved	2 byte **0 byte** 1 byte **0 byte** 1 byte **0 byte**	4 byte
Acknowledge Frame	• Header • Reserved • Status	2 byte 4 byte 4 byte	10 byte	• Header • Reserved • Status	2 byte **0 byte** **2 byte**	4 byte
IOCTL Frame	• Header • Reserved • Channel ID • IOCTL ID • Length • Payload	2 byte 2 byte 1 byte 1 byte 2 byte 8 byte	16 byte	• Header • Reserved • Channel ID • IOCTL ID • Length • Payload	2 byte **0 byte** 1 byte 1 byte **1 byte** 8 byte	13 byte

Interface Frame	• Header • Resaved1 • Channel ID • Reserved2 • Setup	2 byte 2 byte 1 byte 1 byte 2 byte	8 byte	• Header • Resaved1 • Channel ID • Reserved2 • Setup	2 byte **0 byte** 1 byte **0 byte** **1 byte**	4 byte

Table 4.1 PDU router Frame structure

We have optimized by removing or decreasing all the unwanted data or reserved. For example, in the send frame we have optimized the payload by decreasing it from 17 byte to 13 byte in case of full bus i.e. the DLC is 8 by removing 4 byte for the timestamp and all remove all unnecessary data from the header.

4.2 Result

As all work has been done in the low level "Driver level". So, the actual numbers depend on the application which will be applied. Although to show the result we will use multiple use cases with different possibilities of the data on the can bus.

The CAN bus cloud be operated on two different speed 250 Kbs or 500 Kbs. We will conduct all our test and calculation on only one speed 250 Kbs.

First, we should calculate our busload, we have the did the calculation considering that we are sending 10000 packets with a full payload. Theoretical it should take 4.16 sec and practical it took 5.26 sec. This means that the busload is 79.08 %.

Second, we will measure the time consumed to send 10000 packets over the SPI before and after optimization, Also practical and theoretical.

	Before Optimization	After Optimization	Improvemnt
Theortical time for 10000 packet	8.12 sec	5.84 sec	28.07%
Practical time for 10000 packet	9.27 sec	7.95 sec	14.23%

Table 4.2 time consumed for 10000 packets

So, as we can see from the result that we expected to get improvement by 28% but in practical we get around 14%. We managed to decrease the time consumed to send 10000 packets by 1.32 sec.

Third, we will apply some use case to measure our improvement over the SPI bus by calculating the useful bits over the bus which is the CAN data (ID, DLC and Payload) in different scenarios with different DLCs.

	Before Optimization	After Optimization	Improvement
Full bus load (DLC=8)	52%	70.9%	36.3 %
75% bus load (DLC=6)	47.82%	73.3%	53.28 %
62% bus load (DLC=5)	45.45%	71.42%	57.13 %
25% bus load (DLC=2)	36.84%	63.3%	71.8 %

Table 4.3 useful bit over the SPI bus

As we can see from the numbers that we managed to increase the useful bit rate inside the SPI buffer by decreasing the packet headers over the all communication network. In general the useful bits increased from 20 % to 27 % per frame and we reached our highest improvement rate by 70 % improvement --27 % useful data increase – when we receive CAN packets with small payload -2 byte- but if we received the full load -8 byte- this means that our improvement will be 36.3 % --18.9 % useful data increase –

Fourth we also measured the CPU usage during the sending and receiving before and after the optimization and there was significant CPU usage. Also the total CPU usage was the same before and after optimization.

Chapter Five

Conclusion

5.1 Work Conclusion

The automotive industry is one of the most important industry, so the improvement of the automobile vehicle is very crucial to decrease the cost and increase the features and efficiency. This became so achievable by the technological evolution happened in the last in electronics field in the last decade by increasing the electronics component and decreasing the mechanical part everything became feasible.

So, to achieve this improvement and this added feature in the car we should add a lot of ECU to control every feature added inside the car. To utilize this usage of the evolution we should have a good communication network between all the ECU and the module inside the vehicle. With a high-speed baud rate to cover all the data handling inside the network.

The topic work done in the project only focus on achieving high performance on the communication buses and how the data packed and handled inside the microprocessors. Also study and analyze the path of the data in the low-level layer “Driver level” and how is it handled inside the communication layer system architecture starting from the CAN driver passing by the PDU Router to the SPI Driver using the DMA feature till it received by the microprocessor and handled by the PDU Router in the other part then delivered to the CAN Interface Net device reaching one of the APIs.

Basically, the technique used to improve the communication over the bus by utilize the header added to packets over every stage of processing the data and handling it.
The optimization done in two level or two stages of data processing.

First stage done in the SPI driver level after data received from PDU Router and before delivered to the DMA. The Checksum has been changed by the CRC to improve the detection techniques. Also, the packet length has decreased from 384 bytes to 64 bytes so now the data delivered from the one microprocessor to the other has less size and sent in a less time also handled fast. Moreover the technique use in the SPI driver is the same technique used in ISOTP protocol “ISO 15765-2” so we have Synchronize frame in the beginning of the communication then depending on the data available to be sent over we will send them in the single frame packet or if it’s size is bigger than the single frame will be sent as First frame and consecutive frame in all situations the CRC bytes is sent with the packet so it is ready to processed in the after the CRC checked.

Second stage is done in the PDU Router after data received from the CAN bus and the I2C Bus and before being delivered to the SPI driver. Every type of the information was handled by adding different types of the data, some important and some could be removed. So, the optimization has been done by utilizing the different types structure of the added header to the data.

To sum everything up, the added header to the data has been decreased to contain only super important information and the maximum size has been decreased from 25 bytes to 17 bytes. Also, the SPI packet has been decreased from 384 bytes to be 64 bytes so it cloud be sent fast also to utilize less number of data. The time taken to send data over the whole bus is enhanced by 14 % and the percentage of useful bytes over the SPI bus is increased by 36.2%.

5.2 Future Work

As we can see from the followed technique used in optimizing the SPI protocol communication between the two processor in the system and by stacking to the system architecture used in the CCPU. There are two steps of optimization in this system to be done.
One of them related to the driver level and the other one will depend on the application itself.

First step is to reoptimize the packet overhead data in the CCPU before being bypassed to the SPI module to be send. The technique will be advised to use is to optimize in bit level. The meaning is that all data added as packet header will classified in bit needs and merge all of them in some space like two or three bytes maximum and send it as packet overhead. By applying this technique, we will save from one to two bytes on each packet sent.

Second step, this step or level of optimization depending on the application and the data handled (sent or received) over the payload of the CAN bus. In other word, If we know the type of data sent over the CAN bus so we can apply ant technique of data compression and decompression or even data encoding and decoding in both processor so decrease the number of bits sent over the SPI bus and make the protocol fast and efficient. But this technique will depend on the application and we will be different form one application to other.

Bibliography

[1] T. Noergaard, Embedded Systems Architecture,, Elsevier Inc, 2005.
[2] A. Matthews, "Electronics Specifier," The design engineers' resource guide to news, 18 August 2017. [Online]. Available: https://www.electronicspecifier.com/news/analysis/investigating-the-automotive-embedded-systems-market.
[3] "Free RTOS," [Online]. Available:https://www.freertos.org/RTOS.html.
[4] T. Noergaard, Demystifying Embedded Systems Middleware, Elsevier Inc., 2010.
[5] Software, Green Hills, "University of North Carolina," [Online]. Available: https://www.cs.unc.edu/~anderson/teach/comp790/papers/INTEGRITY_RTOS.pdf.
[6] R. Sharma, *In-Vehicular Communication Networking Protocol,* Indianapolis, IN.
[7] Freescale Semiconductor, MPC560B Microcontroller Reference Manual, May 2015.
[8] CAN Specification 2.0, part B http://www.cancia.org/fileadmin/cia/specifications/CAN20B.pdf/
[9] How FlexRay works - http://www.cancia.org/fileadmin/cia/specifications/CAN20B.pdf/