CHAPTER – 1

INTRODUCTION

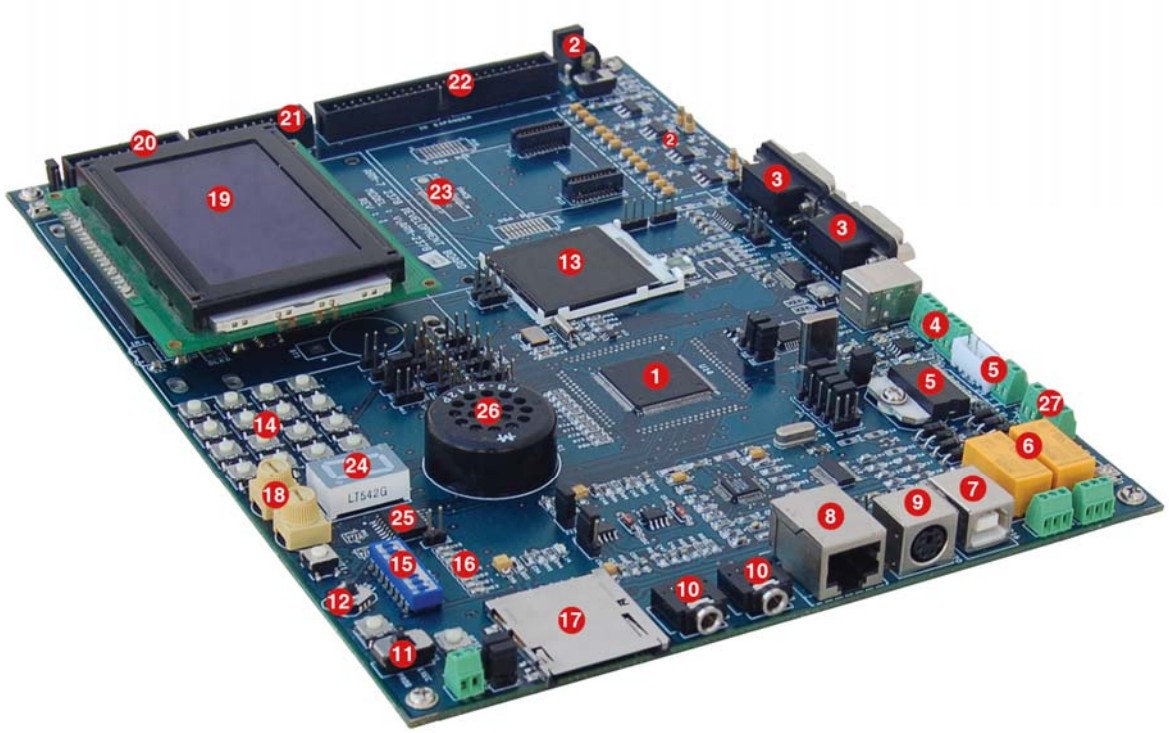
The ViARM-2378 Development System is a full featured Development Board for ARM-7 Core NXP LPC2378 ARM Processor. It has been designed to allow students and engineers to easily exercise and explore the capabilities of ARM-7 Core. It allows LPC2378 ARM Processor to be interfaced with external Circuits and a broad range of peripheral devices, allowing a user to concentrate on Software Development.

The LPC2378 Micro-controller is based on a 32/16 Bit ARM7TDMI-s CPU with real time Emulation and Embedded Trace support that combines with the microcontroller with embedded high-speed 512KB flash memory. It can work with 16-bit Thumb Mode.

With Useful Implemented peripherals, plentiful practical code examples and a broad set of additional on board Peripherals (10/100Mbps Ethernet, MMC/SD, ADC, DAC, RTC, USB etc.,). ViARM Development boards make fast and reliable tools that can satisfy the needs of experienced engineers and beginners alike.

ViARM-2378 Development Boards achieve their small size through Modern SMD technology and Multi layer Design. All Controller signals and ports extend from the controller to high-density pitch Connectors of the board.

ViARM-2378 Hardware Manual Describes the board's design and functions and also includes the Circuit Diagrams and Component Layout.



ViARM-2378 DEVELOPMENT BOARD

ViARM-2378 Development Board Parts Details

1. NXP LPC2378 Micro controller (TQFP-144 Packaging).
2. Power supply section.
3. UART.
4. CAN Port.
5. Stepper Motor.
6. Relay.
7. USB 2.0 Device Connector.
8. 10/100 Base T Ethernet Connector.
9. PS2- Keyboard connector.
10. Stereo Jack for USB Audio Device.
11. Prog/Exec Switch.
12. Joystick.
13. TFT LCD.
14. 4 x 4 Matrix Keypad.
15. 8 Way DIP switch.
16. LED.
17. SD Card Socket.
18. Analog input Trimmer.
19. 128 x 64 Pixels Graphics LCD.
20. Jtag Connector.
21. ADC, DAC and PWM Expansion slot.
22. 50Pin Expansion Header.
23. J-Trace.
24. Seven Segment Display.
25. Serial EEPROM.
26. Speaker.
27. Temperature Sensor.

About LPC2378 CPU...

LPC-2378 is an ARM-based microcontroller for applications requiring serial communications for a variety of purposes. This microcontroller incorporate a 10/100 Ethernet MAC, USB 2.0 Full Speed interface, four UARTs, two CAN channels, an SPI interface, two Synchronous Serial Ports (SSP), three I2C interfaces, an I2S interface, and a Mini Bus. It has 8-bit data/16-bit address parallel bus is available.

Features

* ARM7TDMI-S processor, running at up to 72 MHz.

* Up to 512kB on-chip Flash Program Memory with In-System Programming (ISP) and In-Application Programming (IAP) capabilities. Single Flash sector or full chip erase in 400ms and 256 bytes programming in 1ms. Flash program memory is on the ARM local bus for high performance CPU access.

* Up to 32KB of SRAM on the ARM local bus for high performance CPU access.

* 16KB Static RAM for Ethernet interface. Can also be used as general purpose SRAM.

* 8KB Static RAM for USB interface. Can also be used as general purpose SRAM.

* Dual AHB system that provides for simultaneous Ethernet DMA, USB DMA, and program execution from on-chip Flash with no contention between those functions. A bus bridge allows the Ethernet DMA to access the other AHB subsystem.

* External memory controller that supports static devices such as Flash and SRAM.

* Advanced Vectored Interrupt Controller, supporting up to 32 vectored interrupts.

* General Purpose AHB DMA controller (GPDMA) that can be used with the SSP serial interfaces, the I2S port, and the SD/MMC card port, as well as for memory-to-memory transfers.

* Serial Interfaces:

  - Ethernet MAC with associated DMA controller. These functions reside on an independent AHB bus.

  - USB 2.0 Device with on-chip PHY and associated DMA controller.

  - Four UARTs with fractional baud rate generation, one with modem control I/O, one with IrDA support, all with FIFO. These reside on the APB bus.

  - Two CAN channels with Acceptance Filter/Full CAN mode reside on the APB bus.

  - SPI controller, residing on the APB bus.

  - Two SSP controllers with FIFO and multi-protocol capabilities. One is an alternate for the SPI port, sharing its interrupt and pins. The SSP controllers can be used.

  - Three I2C Interfaces reside on the APB bus. The second and third I2C interfaces are expansion I2Cs with standard port pins rather than special open drain I2C pins.

  - I2S (Inter-IC Sound) interface for digital audio input or output, residing on the APB bus. The I2S interface can be used with the GPDMA.

* Other APB Peripherals:

  - Secure Digital (SD) / Multi-Media Card (MMC) memory card interface.

  - Up to 104 general-purpose I/O pins.

  - 10 bit A/D converter with input multiplexing among 8-pins.

  - 10 bit D/A converter.

  - Four general purpose Timers with two capture inputs each and up to four compare output pins each. Each Timer block has an external count input.

- One PWM/Timer block with support for 3 phase motor control. The PWM has two external count inputs.

- Real Time Clock with separate power pin, clock source can be the RTC oscillator or the APB clock.

- 2KB Static RAM powered from the RTC power pin, allowing data to be stored when the rest of the chip is powered off.

- Watchdog Timer. The watchdog timer can be clocked from the internal RC oscillator, the RTC oscillator, or the APB clock.

* Standard ARM Test/Debug interface for compatibility with existing tools.

* Emulation Trace Module.

* Support for real-time trace.

* Single 3.3V power supply (3.0V to 3.6V).

* Four reduced power modes: Idle, Sleep, Power Down, and Deep Power down.

* Four external interrupt inputs. In addition every PORT0/2 pin can be configured as an edge sensing interrupt.

* Processor wakeup from Power Down mode via any interrupt able to operate during Power Down mode (includes external interrupts, RTC interrupt, and Ethernet wakeup
Interrupt).

* Two independent power domains allow fine- tuning of power consumption based on needed features.

* Brown out detect with separate thresholds for interrupt and forced reset.

* On-chip Power on Reset.

* On-chip crystal oscillator with an operating range of 1MHz to 24MHz.

* 4 MHz internal RC oscillator that can optionally be used as the system clock. For USB and CAN application, an external clock source is suggested to be used.

* On-chip PLL allows CPU operation up to the maximum CPU rate without the need for a high frequency crystal. May be run from the main oscillator, the internal RC oscillator, or the RTC oscillator.

* Boundary scans for simplified board testing.

* Versatile pin function selections allow more possibilities for using on-chip peripheral

Applications

* Industrial control
* Medical systems
* Access control
* Communication Gateway
* General-purpose application.

ARCHITECTURE OVERVIEW

The LPC2378 consists of an ARM7TDMI-S CPU with emulation support, the ARM7 Local Bus for closely coupled, high speed access to the majority of on-chip memory, the AMBA Advanced High-performance Bus (AHB) interfacing to high speed on-chip peripherals and external memory, and the AMBA Advanced Peripheral Bus (APB) for connection to other on-chip peripheral functions. The microcontroller permanently configures theARM7TDMI-S processor for little-endian byte order.

AHB peripherals are allocated a 2MB range of addresses at the very top of the 4GB ARM memory space. Each AHB peripheral is allocated a 16KB address space within the AHB address space. Lower speed peripheral functions are connected to the APB bus. The AHB to APB Bridge interfaces the APB bus to the AHB bus. APB peripherals are also allocated a 2MB range of addresses, beginning at the 3.5GB address point. Each APB peripheral is allocated a 16KB address space within the APB address space.

ARM7TDMI Processor

The ARM7TDMI-S is a general- purpose 32-bit microprocessor, which offers high performance and very low power consumption. The ARM architecture is based on Reduced Instruction Set Computer (RISC) principles, and the instruction set and related decode mechanism are much simpler than those of micro programmed Complex Instruction Set Computers. This simplicity results in a high instruction throughput and impressive real-time interrupt response from a small and cost-effective processor core.

Pipeline techniques are employed so that all parts of the processing and memory systems can operate continuously. Typically, while one instruction is being executed, its successor is being decoded, and a third instruction is being fetched from memory. The ARM7TDMI-S processor also employs a unique architectural strategy known as THUMB, which makes it ideally suited to high-volume applications with memory restrictions, or applications where code density is an issue. The key idea behind THUMB is that of a super-reduced instruction set.

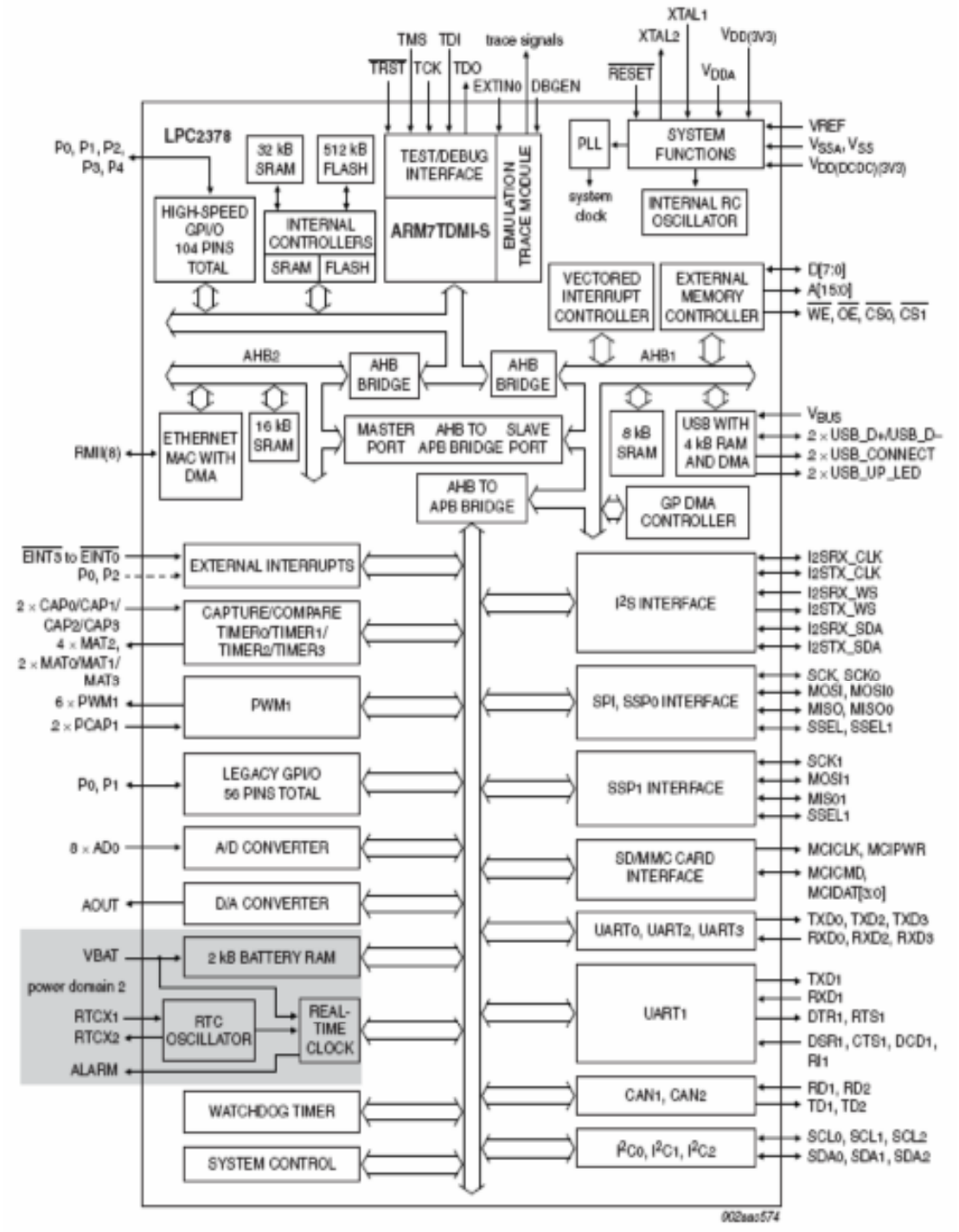
Essentially, the ARM7TDMI-S processor has two instruction sets:

* The standard 32-bit ARM instruction set.
* A 16-bit THUMB instruction set.

The THUMB set's 16-bit instruction length allows it to approach twice the density of standard ARM code while retaining most of the ARM's performance advantage over a traditional 16-bit processor using 16-bit registers. This is possible because THUMB code operates on the same 32-bit register set as ARM code.

THUMB code is able to provide up to 65% of the code size of ARM, and 160% of the performance of an equivalent ARM processor connected to a 16-bit memory system. The ARM7TDMI-S processor is described in detail in the ARM7TDMI-S Datasheet that can be found on official ARM website.

BLOCK DIAGRAM



CHAPTER - 2

HARDWARE DETAILS

ViARM-2378 Development Board features a number of peripheral Devices. In order to enable these devices before programming, you need to check if appropriate jumpers have been properly set.

ViARM-2378 Development Board populated with ARM7 Core LPC2378 CPU.

On Board Peripherals

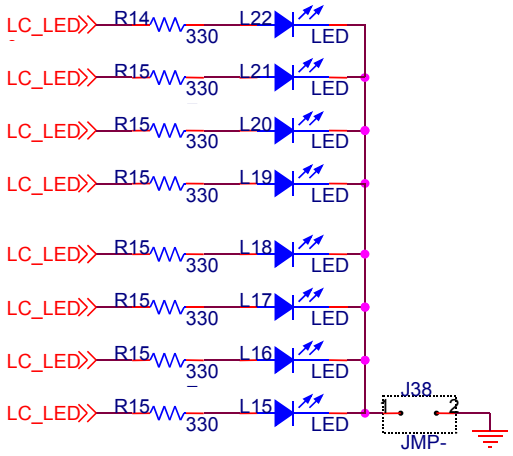
1. 8 Digital Outputs – LED.
2. 8 Digital Inputs - Switch.
3. 4 x 4 Matrix Keypad.
4. Character Based LCD (16 x 2 & 20 x 4).
5. Graphics LCD (128 x 64).
6. Two RS232 Port.
7. SD Card Interface.
8. I2C Peripherals.
    * Real Time Clock.
    * Serial EEPROM.
    * 7 Segment Display.
9. Two SPDT Relay.
10. Temperature Sensor.
11. Stepper Motor Interface.
12. IrDA.
13. 10/100 Base T Ethernet Interface.
14. USB 2.0 Interface.

15. 50-Pin Expansion Header.
16. Two CAN Port.
17. Accelerometer.
18. Joystick.
19. PS/2 Keyboard connector.
20. Jtag Connector.
21. J-Trace Connector.
22. USB Audio Device.
23. Digital To Analog Converter.
24. Analog To Digital Converter.
25. SPI Interface
26. Zigbee
27. OLED Interface
28. TFT LCD Interface

1. LED's

Light Emitting Diodes are the most commonly used components, usually for displaying Pin's Digital State. ViARM-2378 has 8 LED's that are connected to the Microcontroller Port Line.

While we using LEDs, close the Jumper J38.

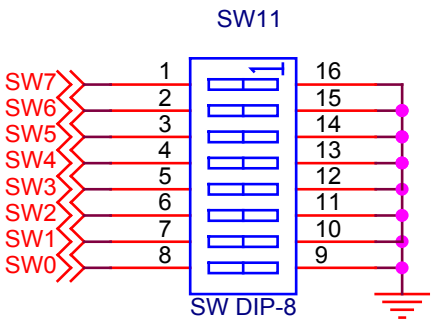


Used Port Lines:

LED0 – LED7 : P3.0 – P3.7

2. SWITCHES

Switches are devices that have two positions - ON and OFF, which have a toggle to establish or break a connection between two contacts. The ViARM-2378 Development Board has one 8-Way Dip Switch.

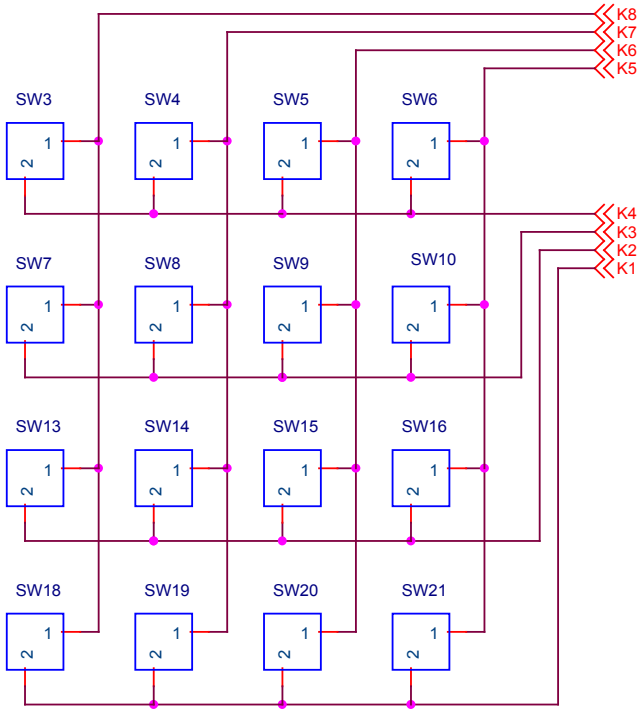


Used Port Lines:

SW0 – SW7 : P4.0 - P4.7

3. Matrix KEY

For Pin reduction, we can connect the Keys in the form of Matrix. In ViARM-2378 has 16 keys which are connected in Matrix format.



Used Port lines

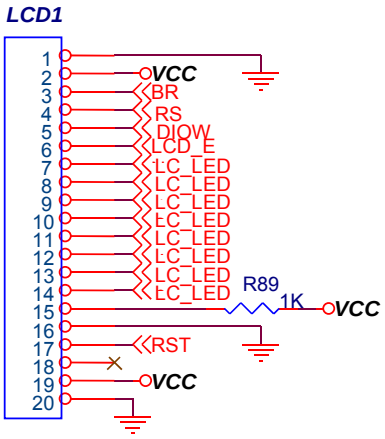
K1 - K8 : P4.8 - P4.15

Scan Lines : P4.8 - P4.11

Read lines : P4.12 - P4.15

4. LCD (Liquid Crystal Display) - Character Based LCD

A standard character LCD is probably the most widely used data Visualization component. Usually it can display four lines of 20 alphanumeric characters, each made up of 5x8 Pixels. The Character LCD communicates with the microcontroller via 8-bit data bus. The Connection to the Microcontroller is shown in below.



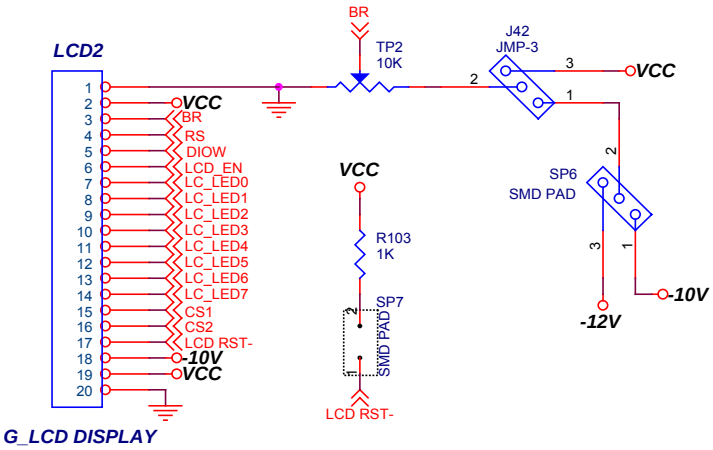
Used Port lines:

Data lines : P3.0 - P3.7
RS : P0.21
DIOW : P1.28
LCDEN : P3.23

5. Graphics LCD

A Graphics LCD (GLCD) allows advanced visual messages to be displayed. While a character LCD can display only alphanumeric characters, a GLCD can be used to display messages in the form of drawings and bitmaps.

The Most commonly used graphic LCD has the screen resolution of 128x64 Pixels. Before a GLCD is connected, the user needs to set the Jumpers. The GLCD's Contrast can be adjusted using the Potentiometer.



While we using the GLCD, close the jumpers J13 to J16 as Downward Direction and close the jumper J42 as left side direction

Used Port lines:

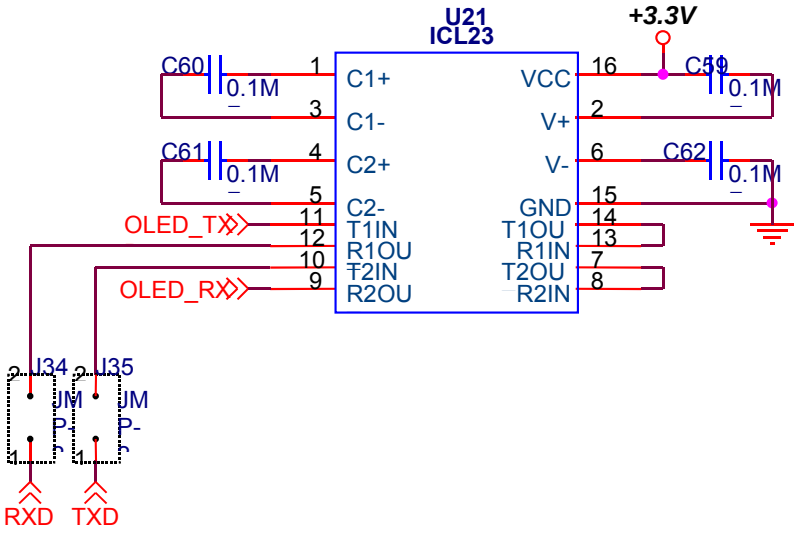
Data lines	:	P3.0 - P3.7
RS	:	P0.12
DIOW	:	P1.28
LCDEN	:	P3.23
CS1	:	P4.29
CS2	:	P4.28

Note:

Make sure to turn off the Power supply before placing GLCD on development board. If the Power supply is connected while placing, GLCD unit can be permanently damaged.

6. RS-232 Communication

RS-232 Communication enables point-to-point Data transfer. It is commonly used in data acquisition applications, for the transfer of data between the microcontroller and a PC. Since the Voltage levels of a microcontroller and PC are not directly compatible with each other, a level transition buffer such as the MAX232 must be used.



ViARM-2378 Development board has Two UART Termination at 9 pin D-type male connector.

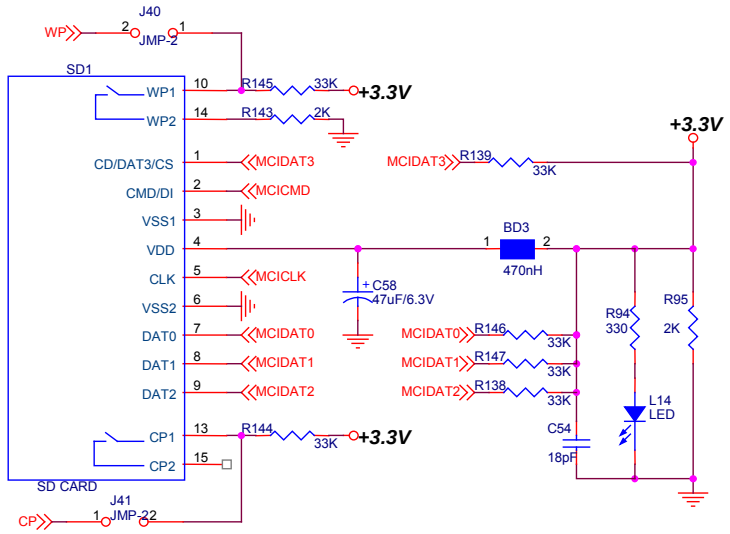
While we using the UART1, close the jumpers J1 & J2

While we using the UART2, close the jumpers J3 & J4

7. MMC/SD (Multimedia CARD)

SD card us used as storage media for a portable devices, in a form that can easily be removed for access by a PC. For example, a digital camera would use an SD card for storing image files. With an SD card reader user can easily transfer the data from SD card to PC and Vice versa. Controller on ViARM-2378 communicates with SD via SPI Communications.

To enable SD card, user to set the Jumpers as shown below. Operating Voltage of SD card is 3.3V DC.



While using SD card close the jumpers J40 and J41.

8. I²C Peripherals

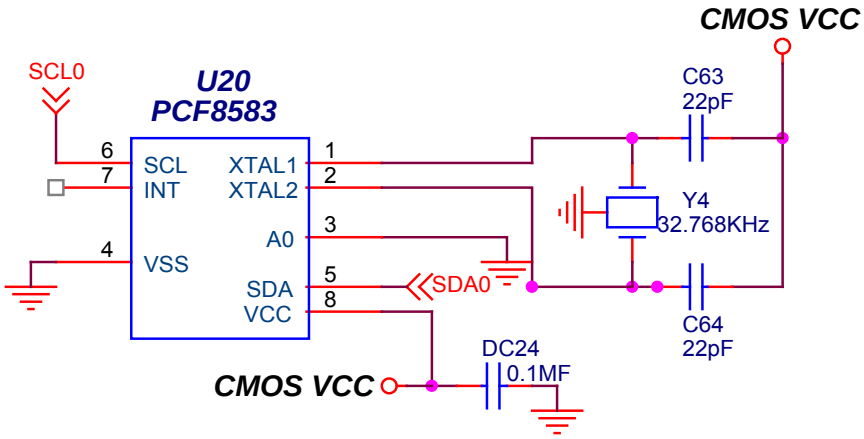
I2C Stands for Inter integrated **Circuit Communications**. This module is built into many arm Controllers. It allows I2C serial communication between two or more devices at a high speed. I2C is a synchronous protocol that allows a master device to initiate communication with a slave device. Data is exchanged between these devices. I2C is also Bi-Directional. This is implemented by an " Acknowledge" System.

In ViARM-2378 Development Board consist of three I2C based Peripherals.

1. I2C Based Real Time Clock
2. I2C Based Serial EEPROM
3. I2C Based Seven Segment Display

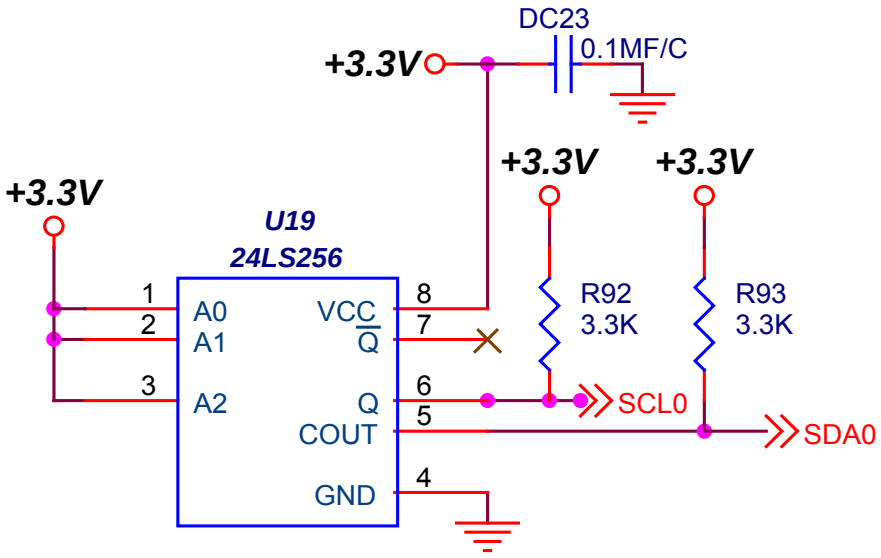
Real Time Clock

Communication : Two Wire Communication
Protocol : I^2C
Hardware IC : PCF8583
Application : Clock and calendar



EEPROM

Serial-interface EEPROM's are used in a broad spectrum of consumer, automotive, telecommunication, medical, industrial and PC related markets. Primarily used to store personal preference data and configuration/setup data, Serial EEPROM's are the most flexible type of non-volatile memory utilized today. Compared to other NVM solutions, Serial EEPROM devices offer a lower pin count, smaller packages, lower voltages, as well as lower power consumption.



I²C Based Seven Segment Display

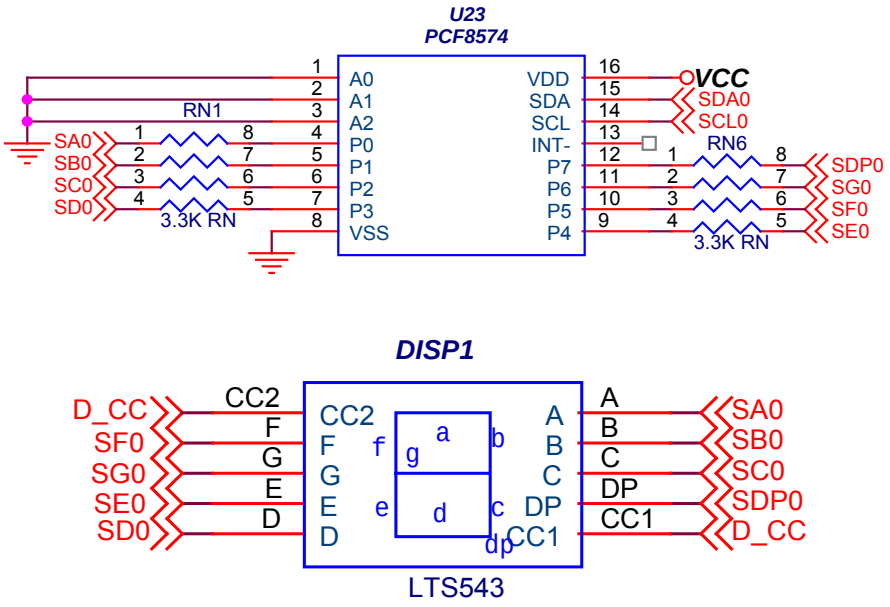
A **seven-segment display,** less commonly known as a **seven-segment indicator**, is a form of display device that is an alternative to the more complex dot-matrix displays. Seven-segment displays are commonly used in electronics as a method of displaying decimal numeric feedback on the internal operations of devices.

A seven-segment display, as its name indicates, is composed of seven elements. Individually on or off, they can be combined to produce simplified representations of the Hindu-Arabic numerals. Each of the numbers and may be represented by two or more different glyphs on seven-segment displays.

The seven segments are arranged as a rectangle of two vertical segments on each side with one horizontal segment on the top and bottom. Additionally, the seventh segment bisects the rectangle horizontally. There are also fourteen-segment displays and sixteen-segment displays (for full alphanumeric); however, these have mostly been replaced by dot-matrix displays.

Often the seven segments are arranged in an *oblique*, or italic, arrangement, which aids readability.

The letters A to G refers to the segments of a 7-segment display.

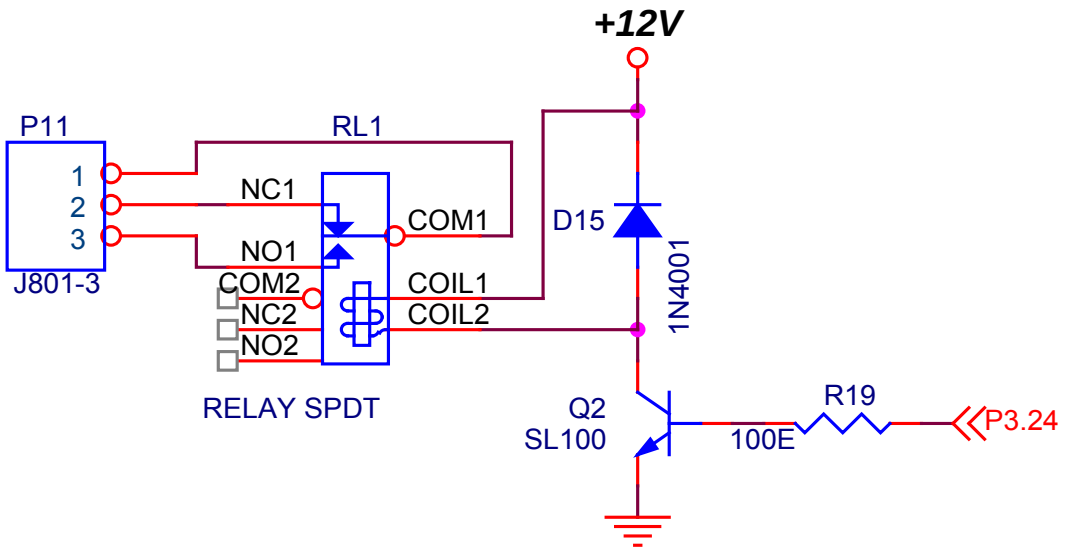


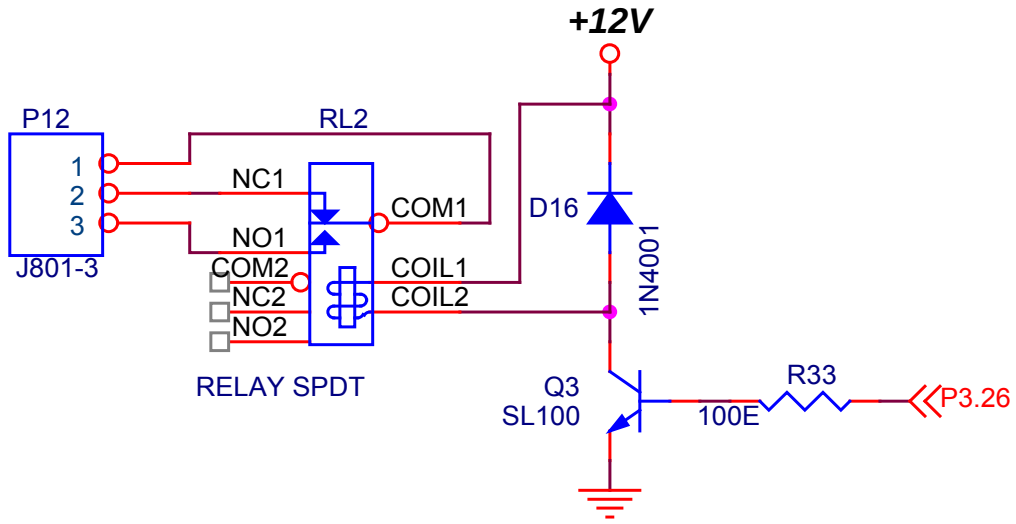
9. Relay

A **relay** is an electrical switch that opens and closes under the control of other an electrical circuit. In the original form, the switch is operated by an electromagnet to open or close one or many sets of contacts.

When a current flows through the coil, the resulting magnetic field attracts an armature that is mechanically linked to a moving contact. The movement either makes or breaks a connection with a fixed contact. When the current to the coil is switched off, a force approximately half as strong as the magnetic force returns the armature to its relaxed position. Usually this is a spring, but gravity is also used commonly in industrial motor starters. Most relays are manufactured to operate quickly. In a low voltage application, this is to reduce noise. In a high voltage or high current application, this is to reduce arcing.

If the coil is energized with DC, a diode is frequently installed across the coil, to dissipate the energy from the collapsing magnetic field at deactivation, which would otherwise generate a spike of voltage and might cause damage to circuit components. If the coil is designed to be energized with AC, a small copper ring can be crimped to the end of the solenoid. This "shading ring" creates a small out-of-phase current, which increases the minimum pull on the armature during the AC cycle. By analogy with the functions of the original electromagnetic device, a solid-state relay is made with a thyristor or other solid-state switching device. To achieve electrical isolation, a light-emitting diode (LED) is used with a phototransistor.



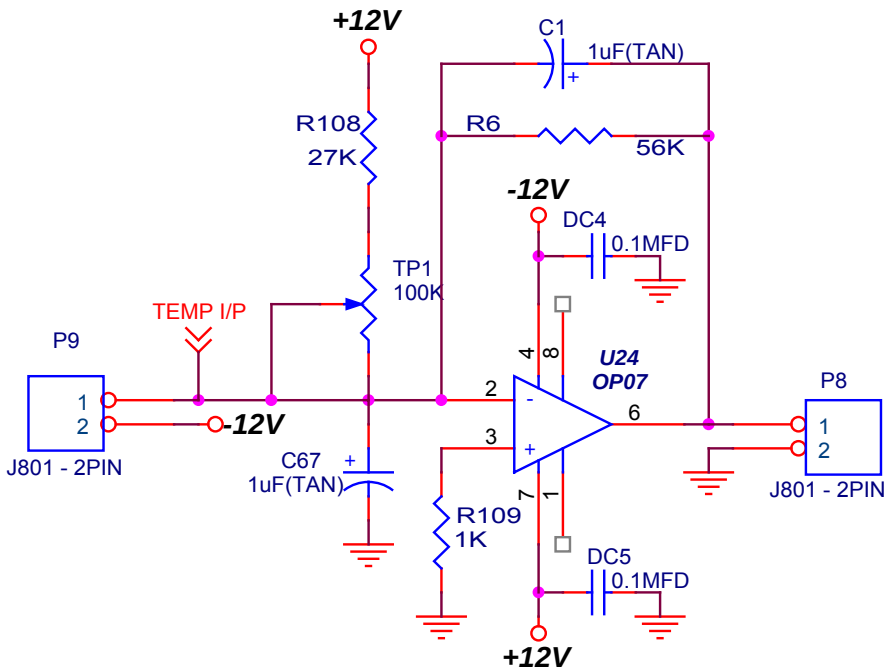


Used Port lines

P3.24 and P3.26.

10. Temperature Sensor

A signal conditioner is a device that converts one type of electronic signal into a another type of signal. Its primary use is to convert a signal that may be difficult to read by conventional instrumentation into a more easily read format. In performing this conversion a number of functions may take place. They include:



1. Amplification

When a signal is amplified, the overall magnitude of the signal is increased. Converting a 0-10mV signal to a 0 -10V signal is an example of amplification.

Electrical Isolation

Electrical isolation breaks the galvanic path between the input and output signal. That is, there is no physical wiring between the input and output. The input is normally transferred to the output by converting it to an optical or magnetic signal then it is reconstructed on the output. By breaking the galvanic path between input and output, unwanted signals on the input line are prevented from passing through to the output. Isolation is required when a measurement must be made on a surface with a voltage potential far above ground. Isolation is also used to prevent ground loops.

Linearization

Converting a non-linear input signal to a linear output signal. This is common for thermocouple signals.

Cold Junction Compensation

Used for thermocouples. The thermocouple signal is adjusted for fluctuations in room temperature.

Excitation

Many sensors require some form of excitation for them to operate. Strain gages and RTDs are two common examples. thermocouple input

11. Stepper Motor

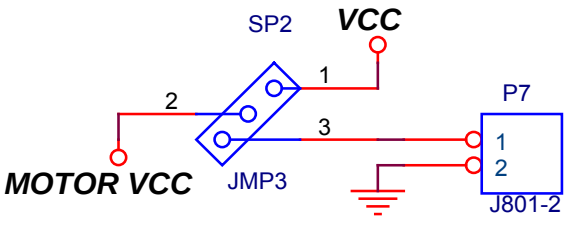
Stepper motors operate differently from normal DC motors, which simply spin when voltage is applied to their terminals. Stepper motors, on the other hand, effectively have multiple "toothed" electromagnets arranged around a central metal gear, as shown at right. To make the motor shaft turn, first one electromagnet is given power, which makes the gear's teeth magnetically attracted to the electromagnet's teeth. When the gear's teeth are thus aligned to the first electromagnet, they are slightly offset from the next electromagnet. So when the next electromagnet is turned on and the first is turned off, the gear rotates slightly to align with the next one, and from there the process is repeated. Each of those slight rotations is called a "step." In that way, the motor can be turned a precise angle. There are two basic arrangements for the electromagnetic coils: bipolar and unipolar.

Theory

A step motor can be viewed as a DC motor with the number of poles (on both rotor and stator) increased, taking care that they have no common denominator. Additionally, soft magnetic material with many teeth on the rotor and stator cheaply multiplies the number of poles (reluctance motor). Like an AC synchronous motor, sinusoidal current, allowing a stepless operation, ideally drives it but this puts some burden on the controller. When using an 8-bit digital controller, 256 micro steps per step are possible. As a digital-to-analog converter produces unwanted Ohmic heat in the controller, pulse-width modulation is used instead to regulate the mean current. Simpler models switch voltage only for doing a step, thus needing an extra current limiter: for every step, they switch a single cable to the motor. Bipolar controllers can switch between supply.

voltage, ground, and unconnected. Unipolar controllers can only connect or disconnect a cable, because the voltage is already hard wired. Unipolar controllers need center-tapped windings. It is possible to drive Unipolar stepper motors with bipolar drivers. The idea is to connect the output pins of the driver to 4 transistors. The transistor must be grounded at the emitter and the driver pin must be connected to the base. Collector is connected to the coil wire of the motor.

The torque they produce rates stepper motors. Synchronous electric motors using soft magnetic materials (having a core) have the ability to provide position-holding torque (called detent torque, and sometimes included in the specifications) while not driven electrically. To achieve full rated torque, the coils in a stepper motor must reach their full rated current during each step. The voltage rating (if there is one) is almost meaningless. The motors also suffer from EMF, which means that once the coil is turned off it starts to generate current because the motor is still rotating. Their needs to be an explicit way to handle this extra current in a circuit otherwise it can cause damage and affect performance of the motor.

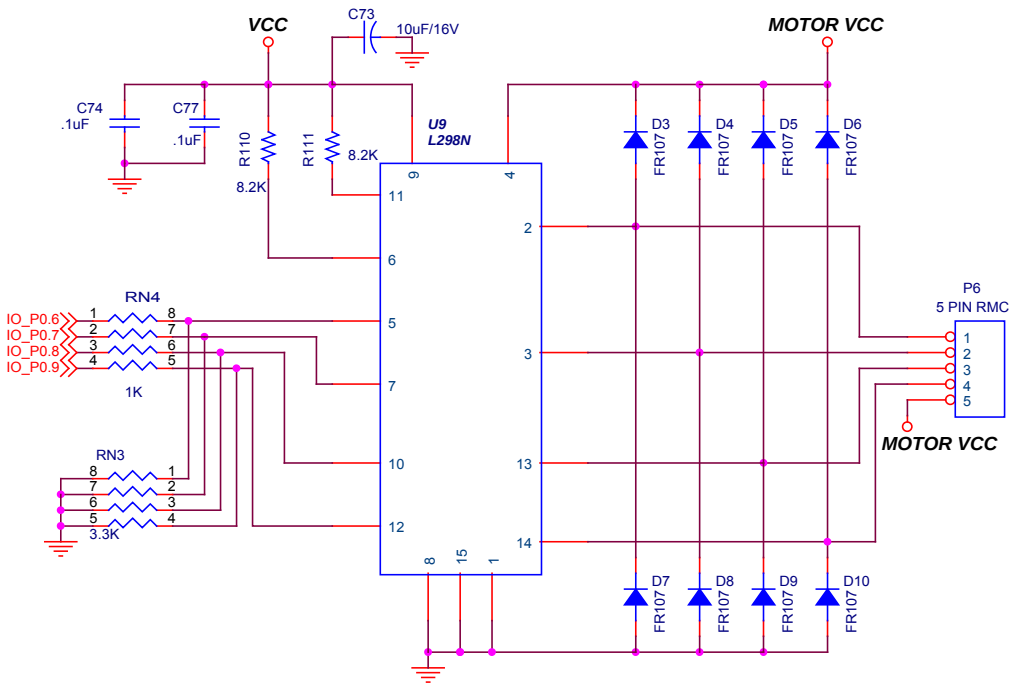


Jumper Position

Closed 1 and 2 - Internal voltage for stepper motor.

Closed 2 and 3 - External voltage for stepper motor.

In ViARM-2378 Development board consists of stepper motor driver, which is shown in below.



Used Port Lines

Coil 1 : P0.6

Coil 2 : P0.7

Coil 3 : P0.8

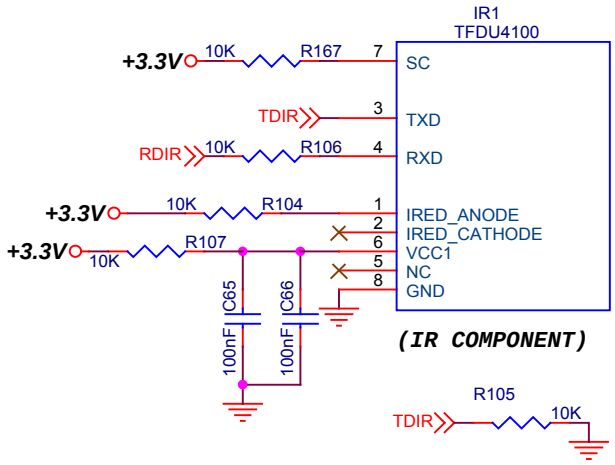
Coil 4 : P0.9

While we using the Stepper motor close Jumpers: J12-J15 as upward direction.

12. IrDA

Infrared Data Association, a group of device manufacturers that developed a standard for transmitting data via infrared light waves. Increasingly, computers and other devices (such as printers) come with IrDA ports. This enables you to transfer data from one device to another without any cables. For example, if both your laptop computer and printer have IrDA ports, you can simply put your computer in front of the printer and output a document, without needing to connect the two with a cable.

IrDA ports support roughly the same transmission rates as traditional parallel ports. The only restrictions on their use are that the two devices must be within a few feet of each other and there must be a clear line of sight between them.



While we using the IrDA, close the above Jumpers J13 and J14 as upward direction.

13. Ethernet

Ethernet is a large, diverse family of frame-based computer networking technologies that operates at many speeds for local area networks (LANs). The name comes from the physical concept of the ether. It defines a number of wiring and signaling standards for the physical layer, through means of network access at the Media Access Control (MAC)/Data Link Layer, and a common addressing format.

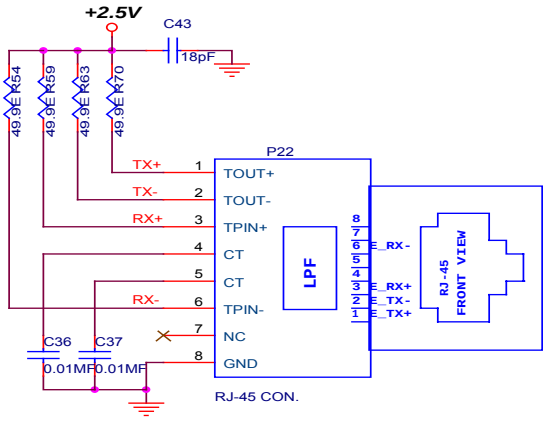
Ethernet has been standardized as IEEE802.3. The combination of the twisted pair versions of Ethernet for connecting end systems to the network, along with the fiber optic versions for site backbones, has become the most widespread-wired LAN technology. It has been in use from the 1990s to the present, largely replacing competing LAN standards such as coaxial cable Ethernet, token ring, FDDI, and ARCNET. In recent years, Wi-Fi, the wireless LAN standardized by IEEE802.11, has been used instead of Ethernet for many home and small office networks and in addition to Ethernet in larger installations.

Ethernet was originally based on the idea of computers communicating over a shared coaxial cable acting as a broadcast transmission medium. The methods used show some similarities to radio systems, although there are major differences, such as the fact that it is much easier to detect collisions in a cable broadcast system than a radio broadcast. The common cable providing the communication channel was likened to the ether and it was from this reference that the name "Ethernet" was derived.

From this early and comparatively simple concept, Ethernet evolved into the complex networking technology that today powers the vast majority of local computer networks. The coaxial cable was later replaced with point-to-point links connected together by hubs and/or switches in order to reduce installation costs, increase reliability, and enable point-to-point management and troubleshooting. Star LAN was the first step in the evolution of Ethernet from a coaxial cable bus to a hub-managed, twisted-pair network. The advent of twisted-pair wiring enabled Ethernet to become a commercial success.

Above the physical layer, Ethernet stations communicate by sending each other data packets, small blocks of data that are individually sent and delivered. As with other IEEE 802 LANs, each Ethernet station is given a single 48 bit MAC address, which is used both to specify the destination and the source of each data packet. Network interface cards (NICs) or chips normally do not accept packets addressed to other Ethernet stations. Adapters generally come programmed with a globally unique address, but this can be overridden, either to avoid an address change when an adapter is replaced, or to use locally administered addresses.

Despite the very significant changes in Ethernet from a thick coaxial cable bus running at 10 Mbit/s to point-to-point links running at 1 Gbit/s and beyond, all generations of Ethernet (excluding very early experimental versions) share the same frame formats (and hence the same interface for higher layers), and can be readily (and in most cases, cheaply) interconnected.



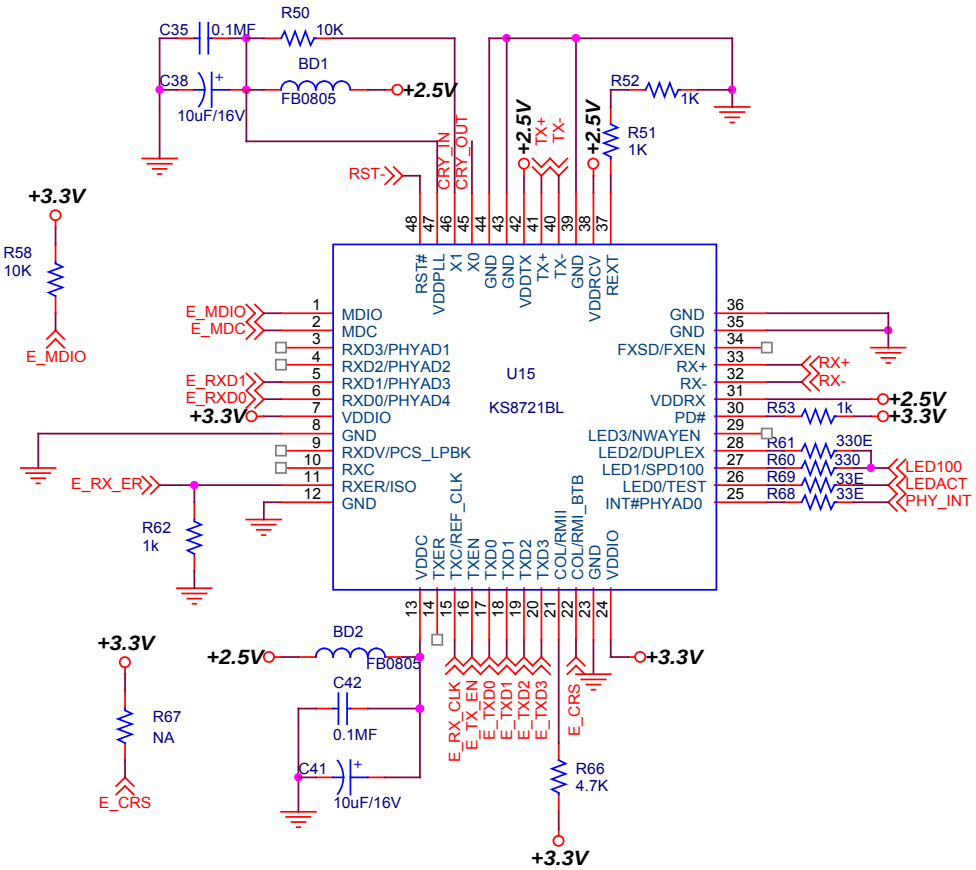
Due to the ubiquity of Ethernet, the ever-decreasing cost of the hardware needed to support it, and the reduced panel space needed by twisted pair Ethernet, most manufacturers now build the functionality of an Ethernet card directly into PC motherboards, obviating the need for installation of a separate network card.

ABOUT KS8721BL (10/100Base-TX/FX PHYSICAL LAYER TRANSCEIVER)

The KS8721BL is a 10BASE-T, 100BASE-TX and 100 BASE-FX physical layer transceiver providing M11/RM11 interfaces to MACS and switches. Using a unique mixed-signal design that extends signaling distance while reducing power consumption, the KS8721BL represents Micrel's fourth generation single-port fast Ethernet PHY.

Micrel's architecture dissipates an ultra-low 275mW in power, including transmit output derivers. This translates to increased reliability while enabling development in today's power sensitive applications. Multiple power down modes, accessed via configuration pins or registers, allow further levels of power saving through manual selection or automatic detection of cable energy.

The KS8721BL automatically configures itself for 100Mbps or 10Mbps and full-or half-duplex operation using an on-chip auto negotiation algorithm. Combined with auto MDI/MDIX for automatic correction and detection of crossover and straight through cables, the KS8721BL/SL provides an easy to use high performance solution for 10BASE-T, 100BASE-TX and 100BASE-FX applications.



LPC2378 port lines are connected with 10/100Base-TX/FX transceiver are,

E_TXD0 : P1.0

E_TXD1 : P1.1

E_TX_EN : P1.4

E_CRS : P1.8

E_RXD0 : P1.9

E_RXD1 : P1.10

E_RX_ER : P1.14

E_RX_CLK : P1.15

E_MDC : P1.16

E_MDIO : P1.17

PHY_INT : P0.30

Applications

LAN on Motherboard

Media Converter

Set-Top Box

Cable/DSL Modem

Network Printer

Game console

Cable configuration

1. Cross over cable - PC to PC or PC to Kit communication
2. Straight Cable - PC to HUB communication

14. Universal Synchronous Bus (USB)

Universal Serial Bus (USB) is a serial bus standard to interface devices. A major component in the legacy-free PC, USB was designed to allow peripherals to be connected using a single standardized interface socket, to improve plug-and-play capabilities by allowing devices to be connected and disconnected without rebooting the computer (hot swapping). Other convenient features include powering low-consumption devices without the need for an external power supply and allowing some devices to be used without requiring individual device drivers to be installed.

USB is intended to help retire all legacy serial and parallel ports. USB can connect computer peripherals such as mouse devices, keyboards, PDAs, game pads and joysticks, scanners, digital cameras and printers. For many devices such as scanners and digital cameras, USB has become the standard connection method. USB is also used extensively to connect non-networked printers; USB simplifies connecting several printers to one computer. USB was originally designed for personal computers, but it has become commonplace on other devices such as PDAs and video game consoles. In 2004, there were about 1 billion USB devices in the world.

The USB Implementers Forum (USB-IF), an industry standards body incorporating leading companies from the computer and electronics industries, standardizes the design of USB. Notable members have included Apple Inc., Hewlett-Packard, NEC, Microsoft, Intel and Agere.

Device Classes

Devices that attach to the bus can be full-custom devices requiring a full-custom device driver to be used, or may belong to a device class. These classes define an expected behavior in terms of device and interface descriptors so that the same device driver may be used for any device that claims to be a member of a certain class. An operating system is supposed to implement all device classes so as to provide generic drivers for any USB device. The Device Working Group of the USB Implementers Forum decides upon device classes.

Example device classes include:

1. 0x03: USB human interface device class ("HID"), keyboards, mice, etc.
2. 0x08: USB mass storage device class used for USB flash drives, memory card readers, digital audio players etc.
3. 0x09: USB hubs.
4. 0x0B: Smart card readers.
5. 0x0E: USB video device class, web cam-like devices, motion image capture devices.
6. 0xE0: Wireless controllers, for example Blue tooth dongles.

USB mass-storage

A Flash Drive, a typical USB mass-storage device USB implements connection to storage devices using a set of standards called the USB mass storage device class (referred to as MSC or UMS). This was initially intended for traditional magnetic and optical drives, but has been extended to support a wide variety of devices. USB is not intended to be a primary bus for a computer's internal storage: buses such as ATA (IDE), Serial ATA (SATA), and SCSI fulfill that role. However, USB has one important advantage in that it is possible to install and remove devices without opening the computer case, making it useful for external drives. Today a number of manufacturers offer external portable USB hard drives, or empty enclosures for drives, that offer performance comparable to internal drives. These external drives usually contain a translating device that interfaces a drive of conventional technology (IDE, ATA, SATA, ATAPI, or even SCSI) to a USB port. Functionally, the drive appears to the user just like another internal drive. Other competing standards that allow for external connectivity are SATA and Fire wire.

Human-interface devices (HIDs)

Mice and keyboards are frequently fitted with USB connectors, but because most PC motherboards still retain PS/2 connectors for the keyboard and mouse as of 2007, are generally supplied with a small USB-to-PS/2 adaptor so that they can be used with either USB or PS/2 ports. There is no logic inside these adaptors: they make use of the fact that such HID interfaces are equipped with controllers that are capable of serving both the USB and the PS/2 protocol, and automatically detect which type of port they are plugged in to. Joysticks, keypads, tablets and other human-interface devices are also progressively migrating from MIDI, PC game port, and PS/2 connectors to USB. Apple Macintosh computers have used USB exclusively for all wired mice and keyboards since January 1999.

USB signaling

The USB standard uses the NRZI system to encode data.

USB signals are transmitted on a twisted pair of data cables, labeled D+ and D−. These collectively use half-duplex differential signaling to combat the effects of electromagnetic noise on longer lines. D+ and D− usually operate together; they are not separate simplex connections. Transmitted signal levels are 0.0–0.3 volts for low and 2.8–3.6 volts for high.

USB supports three data rates:

1. A **Low Speed** rate of 1.5Mbit/s (153.6kB/s) that is mostly used for Human Interface Devices (HID) such as keyboards, mice, and joysticks.

2. A **Full Speed** rate of 12Mbit/s (1.2MB/s). Full Speed was the fastest rate before the USB 2.0 specification and many devices fall back to Full Speed. Full Speed devices divide the USB bandwidth between them in a first-come first-served basis and it is not uncommon to run out of bandwidth with several isochronously devices. All USB Hubs support Full Speed.

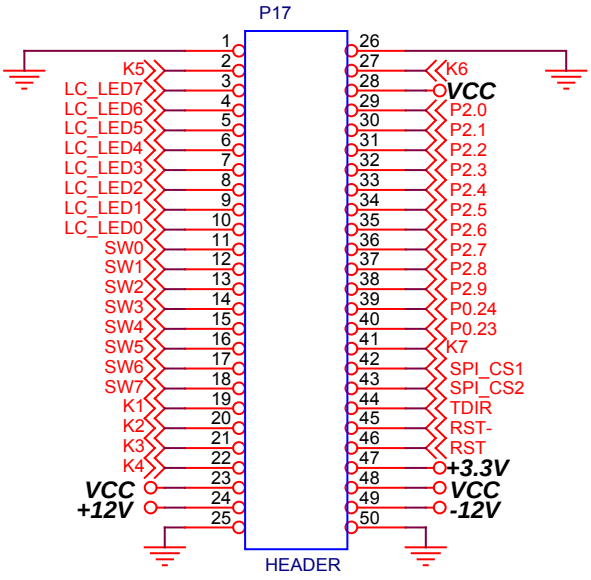
3. A **Hi-Speed** rate of 480Mbit/s (48MB/s).

Though Hi-Speed devices are commonly referred to as "USB 2.0" and advertised as "up to 480Mbit/s", not all USB 2.0 devices are Hi-Speed. The actual throughput currently (2006) attained with real devices is about half of the full theoretical (48MB/s) data throughput rate.

Most hi-speed USB devices typically operate at much slower speeds, often about 3MB/s overall, sometimes up to 10-20 MB/s. LPC2378 CPU has Built in USB2.0 (12Mbits) compatible Devices. We are providing all those 2378 examples

15. Direct Port Access

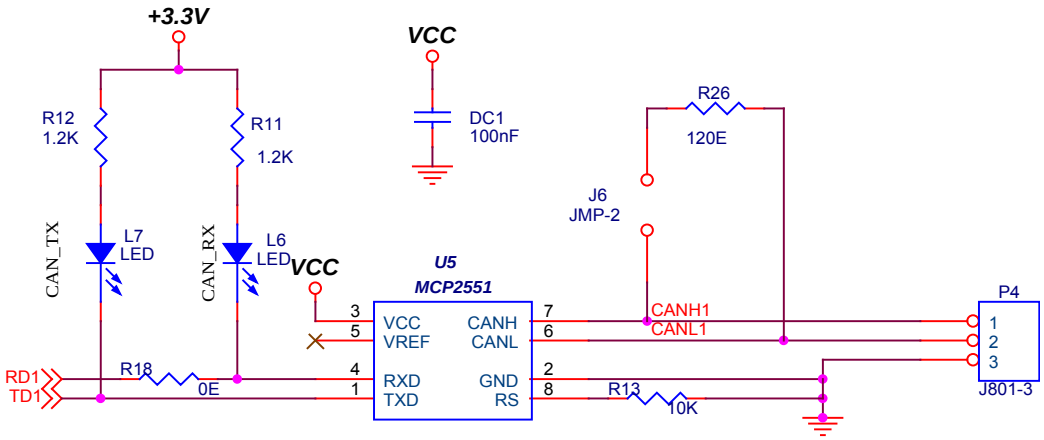
All the Controller input/output pins can be accessed via 50 Pin Box type header. This connector can be used for system expansion with external boards. Ensure that the on-board peripherals are disconnected from micro controller by setting the appropriate jumpers, while external peripherals are using the same pins.

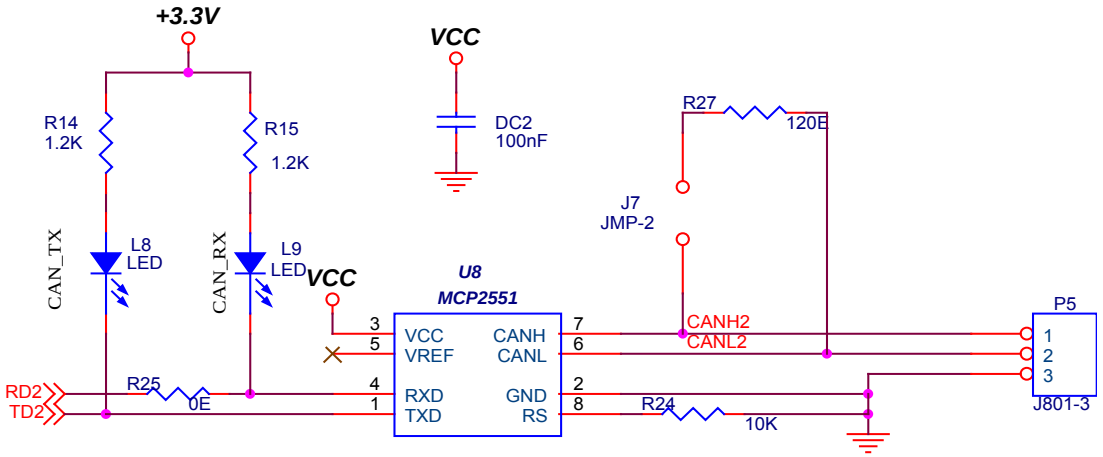


16. CAN

CAN is a serial bus network of microcontrollers that connects devices, sensors and actuators in a system or sub-system for real-time control applications. There is no addressing scheme used in controller area networks, as in the sense of conventional addressing in networks (such as Ethernet). Rather, messages are broadcast to all the nodes in the network using an identifier unique to the network. Based on the identifier, the individual nodes decide whether or not to process the message and also determine the priority of the message in terms of competition for bus access. This method allows for uninterrupted transmission when a collision is detected, unlike Ethernets that will stop transmission upon collision detection.

Controller area networks were first developed for use in automobiles. Equipped with an array of sensors, the network is able to monitor the systems that the automobile depends on to run properly and safely. Beyond automobiles, controller area networks can be used as an embedded communication system for microcontrollers as well as an open JOYSTICK communication system for intelligent devices.

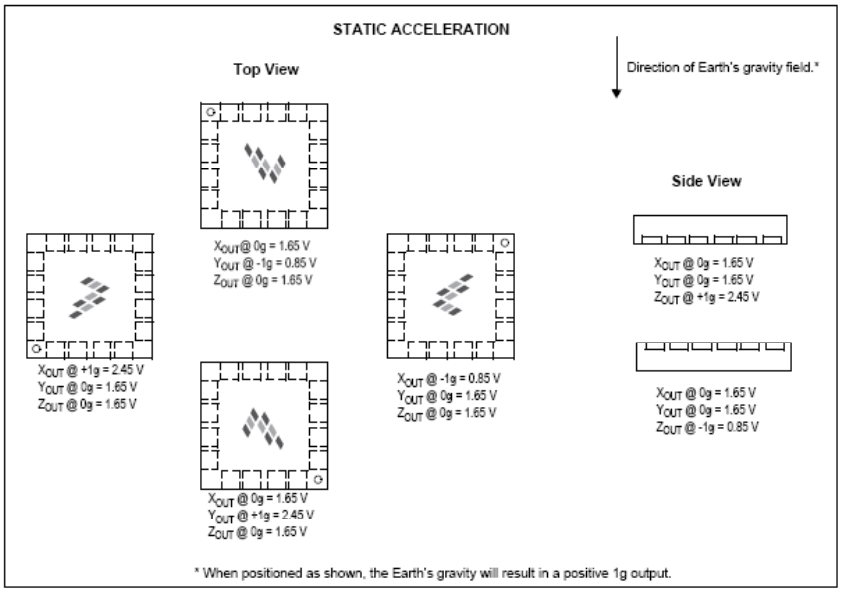


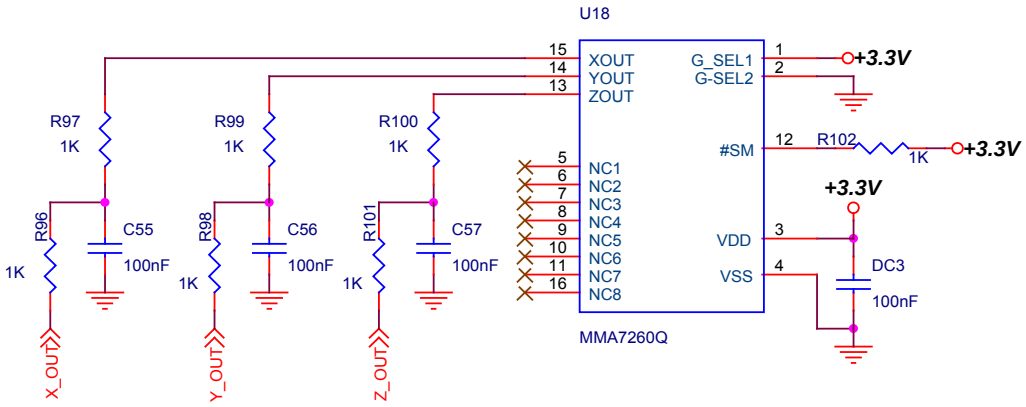


17. ACCELEROMETER

The 3-Axis Accelerometer consists of three –5 to + 5g accelerometers mounted in one small block. Using the appropriate data collection hardware and software, you can graph any of these components, or calculate the magnitude of then acceleration. The 3-Axis Accelerometer can be used for a wide variety of experiments and demonstrations, both inside the lab and outside. There are sample data collected with the 3-axis

The MMA7260Q low cost capacitive micro machined accelerometer features signal conditioning, a 1-pole low pass filter, temperature compensation and g- Select which allows for the selection among for sensitivities. Zero-g offset full-scale span and filter cut-off are factory set and require no external devices. Includes a Sleep Mode that makes it ideal for handheld battery powered electronics.





Used Port Lines

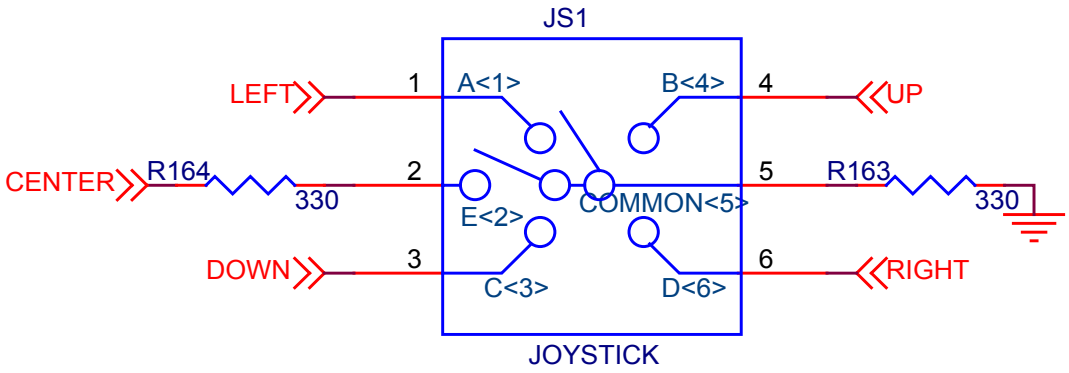
X_OUT : P0.24

Y_OUT : P0.23

Z_OUT : P0.12

18. JOYSTICK

It is an input device first found on arcade game machines, then home game systems, and finally on computers. It consists of any stick-like object attached to a base that can be pushed in four or more directions. Usually there's a button in the vicinity of the joystick. Joysticks come in many shapes and sizes, with the typical arcade joystick having a large ball at the top that can be easily gripped; but lately, cheap button pads in many home gaming systems have replaced joysticks. However, smaller joysticks that can be pushed around with a single finger have been added to some of the pads as well.



Used Port Lines

Up : P1.18

Down : P1.19

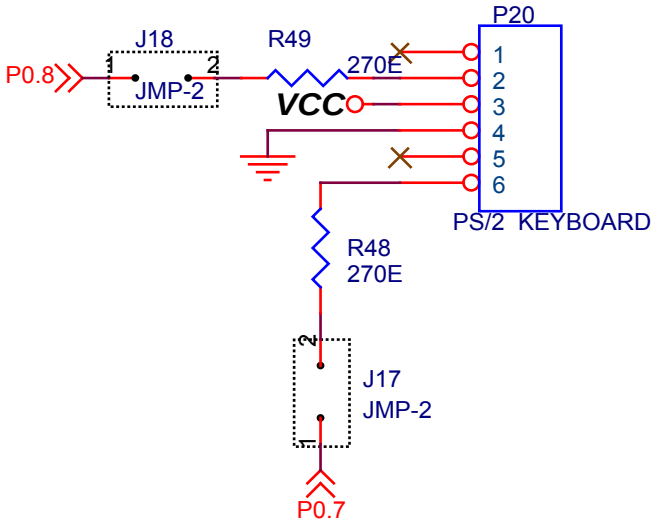
Right : P1.22

Left : P1.27

Center : P1.25

19. PS2 KEYBOARD

The PS/2 connector is used for connecting a keyboard and a mouse to a PC compatible computer system. PS/2 ports are designed to connect the digital I/O lines of the microcontroller in the external device directly to the digital lines of the microcontroller. Its name comes from the IBM Personal System/2 series of personal computers, with which it was introduced in 1987. The keyboard connector replaced the larger 5-pin DIN used in the IBM PC/AT design.

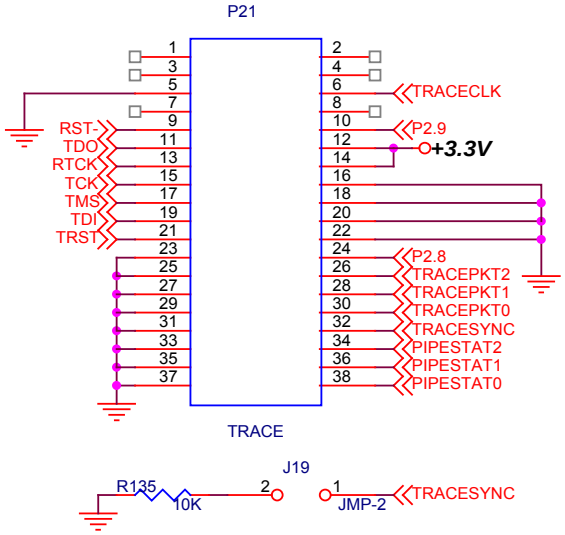


While using PS2 Keyboard close the jumper J17 and J18.

Used Port Lines : P0.7 & P0.8

20. J-TRACE

J-Trace is a JTAG emulator designed for ARM cores which includes trace (ETM) support. It connects via USB to a PC running Microsoft Windows 2000 or XP. J-Trace has a built-in 20-pin JTAG connector and a built in 38-pin JTAG+Trace connector, which is compatible with the standard 20-pin connector and 38-pin connector defined by ARM.



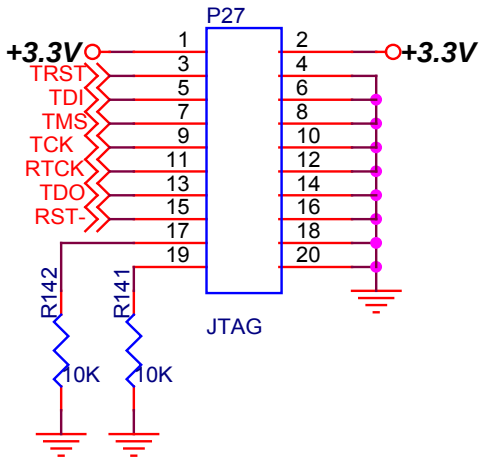
While we using J-Trace close the jumpers J26 to J31 as upward direction and also close J20, J21 as Downward direction.

Used Port Lines : P2.0 To P2.7

21. J-TAG

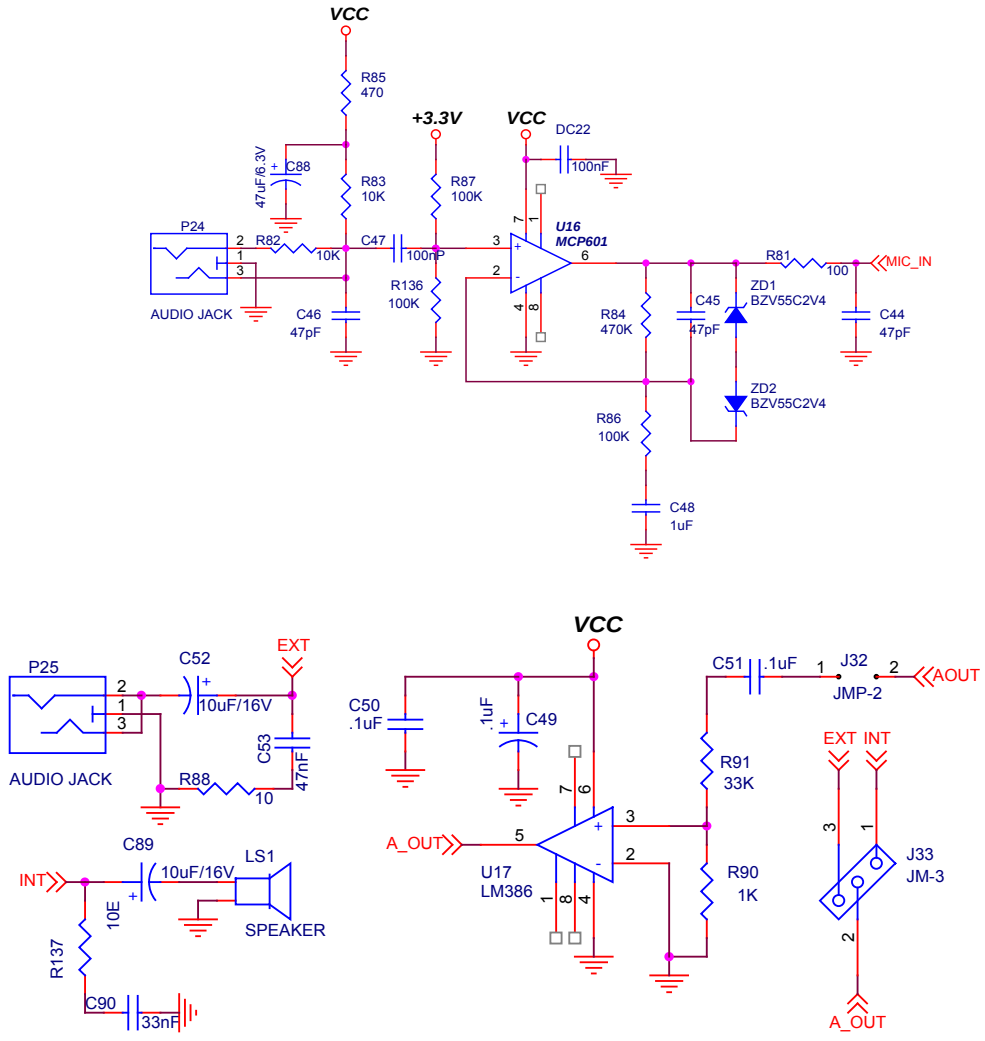
JTAG (Joint Test Action Group) is the usual name used for the IEEE 1149.1 standard entitled Standard Test Access Port and Boundary-Scan Architecture for test access ports used for testing printed circuit boards using boundary scan. While designed for printed circuit boards, JTAG is nowadays primarily used for accessing sub-blocks of integrated circuits, and is also useful as a mechanism for debugging embedded systems, providing a convenient "back door" into the system. When used as a debugging tool, an in-circuit emulator - which in turn uses JTAG as the transport mechanism - enables a programmer to access an on-chip debug module which is integrated into the CPU, via the JTAG interface.

The debug module enables the programmer to debug the software of an embedded system.



22. USB - AUDIO DEVICE

The Audio Device Class Definition applies to all devices or functions embedded in composite devices that are used to manipulate audio, voice, and sound-related functionality. This includes both audio data (analog and digital) and the functionality that is used to directly control the audio environment, such as Volume and Tone Control. The Audio Device Class does not include functionality to operate transport Mechanisms that are related to the reproduction of audio data, such as tape transport mechanisms or CDROM drive control.



23. DIGITAL TO ANALOG CONVERSION (DAC)

In Electronics, a digital-to-analog converter (DAC or D-to-A) is a device for converting a digital (usually binary) code to an analog signal (current, voltage or electric charge). Digital-to-analog converters are interfaces between the abstract digital world and analog real life.

An analog-to-digital converter (ADC) performs the reverse operation.

A DAC usually only deals with pulse-code modulation (PCM) encoded signals. The job of converting various compressed forms of signals into PCM is left to codecs.

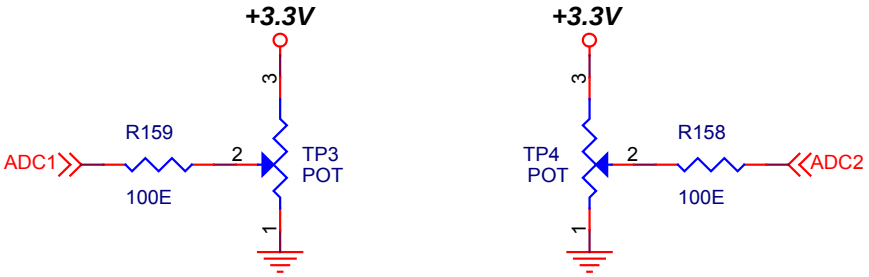
The DAC fundamentally converts finite-precision numbers (usually fixed-point binary numbers) into a physical quantity, usually an electrical voltage. Normally the output voltage is a linear function of the input number. Usually these numbers are updated at uniform sampling intervals and can be thought of as numbers obtained from a sampling process. These numbers are written to the DAC, sometimes along with a clock signal that causes each number to be latched in sequence, at which time the DAC output voltage changes rapidly from the previous value to the value represented by the currently latched number. The effect of this is that the output voltage is *held* in time at the current value until the next input number is latched resulting in a piecewise constant output. This is equivalently a zero-order hold operation and has an effect on the frequency response of the reconstructed signal.

LPC2378 CPU has Single channel DAC, which is terminated at P29 connector 1st Pin.

24. ANALOG TO DIGITAL CONVERSION (ADC)

ViARM - 2378 Development board has two potentiometers for working with A/D Converter. All Potentiometers outputs are in the range of 0V to 3.3V. Each Potentiometer can be connected on two different analog inputs. Jumpers group enables the connection between Potentiometer and LPC2378 Analog inputs.

Applications of A/D conversion are various. ARM processor takes analog signal from its input pin and translates it into a digital value. Basically, you can measure any analog signal that fits in range acceptable by LPC2378. That range is 0V to 3.3V. Simultaneously, user can give external analog source. For that purpose, analog input signals are terminated at 20pin Box type header (P23).



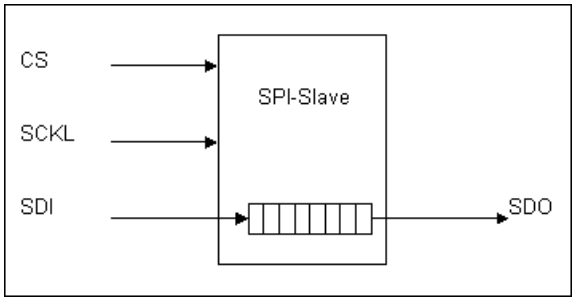
While we using the on-chip ADC1 and ADC2, close the Jumpers J36 and J37 as downward direction.

25. SPI INTERFACE

The Principle

The Serial Peripheral Interface is used primarily for a synchronous serial communication of host processor and peripherals. However, a connection of two processors via SPI is just as well possible and is described at the end of the chapter.

In the standard configuration for a slave device two control and two data lines are used. The data output SDO serves on the one hand the reading back of data, offers however also the possibility to cascade several devices. The data output of the preceding device then forms the data input for the next IC.



1. SPI Slave

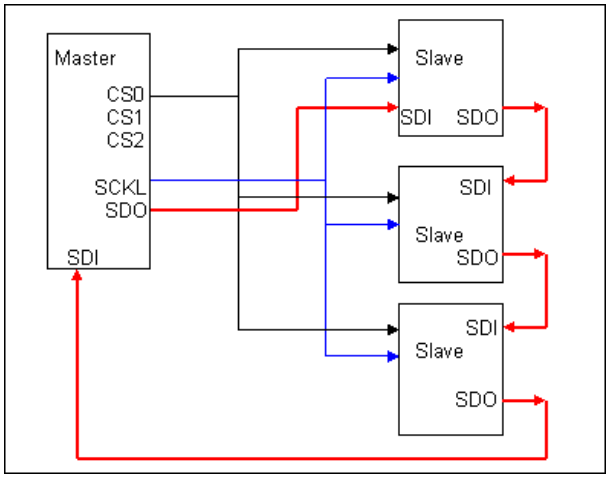
There is a MASTER and a SLAVE mode. The MASTER device provides the clock signal and determines the state of the chip select lines, i.e. it activates the SLAVE it wants to communicate with. CS and SCKL are therefore outputs.

The SLAVE device receives the clock and chip select from the MASTER, CS and SCKL are therefore inputs.

This means there is one master, while the number of slaves is only limited by the number of chip selects.

A SPI device can be a simple shift register up to an independent subsystem. The basic principle of a shift register is always present. Command codes as well as data values are serially transferred, pumped into a shift register and are then internally available for parallel processing. Normally the shift registers are 8Bit or integral multiples of it. Of course there also exist shift registers with an odd number of bits. For example two cascaded 9Bit EEPROMs can store 18Bit data.

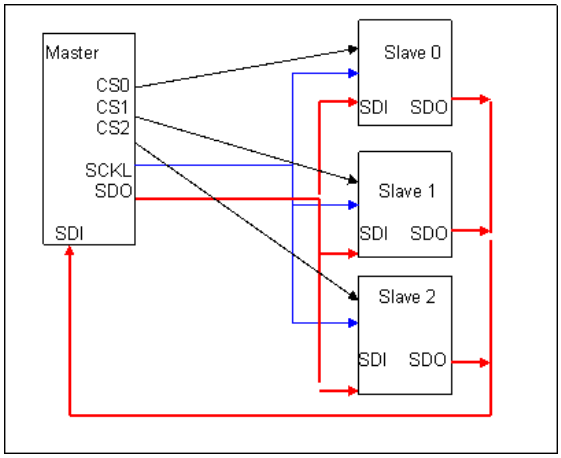
If a SPI device is not selected, its data output goes into a high-impedance state (hi-Z), so that it does not interfere with the currently activated devices. When cascading several SPI devices, they are treated as one slave and therefore connected to the same chip select.



2. Cascading several SPI devices

In illustration 2 the cascaded devices are evidently looked at as one larger device and receive therefore the same chip select. The data output of the preceding device is tied to the data input of the next, thus forming a wider shift register.

If independent slaves are to be connected to a master an other bus structure has to be chosen, as shown in illustration 3. Here, the clock and the SDI data lines are brought to each slave. Also the SDO data lines are tied together and led back to the master. Only the chip selects are separately brought to each SPI device. Last not least both types may be combined.



3. Master with Independed slave

It is also possible to connect two micro controllers via SPI. For such a network, two protocol variants are possible. In the first, there is only one master and several slaves and in the second, each micro controller can take the role of the master. For the selection of slaves again two versions would be possible but only one variant is supported by hardware. The hardware supported variant is with the chip selects, while in the other the selection of the slaves is done by means of an ID packed into the frames. The assignment of the IDs is done by software. Only the selected slave drives its output, all other slaves are in high-impedance state. The output remains active as long as the slave is selected by its address. The first variant, named single-master protocol, resembles the normal master-slave communication. The micro controller configured as a slave behaves like a normal peripheral device.

Data and Control Lines of the SPI

The SPI requires two control lines (CS and SCLK) and two data lines (SDI and SDO). Motorola names these lines MOSI (Master-Out-Slave-In) and MISO (Master-In-Slave-Out). The chip select line is named SS (Slave-Select). With CS (Chip-Select) the corresponding peripheral device is selected. This pin is mostly active-low. In the unselected state the SDO lines are hi-Z and therefore inactive. The master decides with which peripheral device it wants to communicate. The clock line SCLK is brought to the device whether it is selected or not. The clock serves as synchronization of the data communication. The majority of SPI devices provide these four lines. Sometimes it happens that SDI and SDO are multiplexed.

SPI Configuration

Because there is no official specification, what exactly SPI is and what not, it is necessary to consult the data sheets of the components one wants to use. Important are the permitted clock frequencies and the type of valid transitions. There are no general rules for transitions where data shouls be latched. Although not specified by Motorola, in practice four modes are used. These four modes are the combinations of CPOL and CPHA. In table 1, the four modes are listed.

SPI-mode	CPOL	CPHA
0	0	0
1	0	1
2	1	0
3	1	1

4. SPI - Modes

If the phase of the clock is zero, i.e. CPHA = 0, data is latched at the rising edge of the clock with CPOL = 0, and at the falling edge of the clock with CPOL = 1. If CPHA = 1, the polarities are reversed. CPOL = 0 means falling edge, CPOL = 1 rising edge.

The micro controllers from Motorola allow the polarity and the phase of the clock to be adjusted. A positive polarity results in latching data at the rising edge of the clock. However data is put on the data line already at the falling edge in order to stabilize. Most peripherals, which can only be slaves, work with this configuration. If it should become necessary to use the other polarity, transitions are reversed.

The Different Peripheral Types

The question is of course, which peripheral types exist and which can be connected to the host processor. The available types and their characteristics are now discussed. Peripheral types can be subdivided into the following categories:

- Converters (ADC and DAC)
- Memories (EEPROM and FLASH)
- Real Time Clocks (RTC)
- Sensors (temperature, pressure)
- Others (signalmixer, potentiometer, LCD controller, UART, CAN controller, USB controller, amplifier)

In the three categories converters, memories and RTCs, there is a great variety of component. Devices belonging to the last both groups are more rarely.

There are lots of converters with different resolutions, clock frequencies and number of channels to choose from. 8,10,12 up to 24Bit with clock frequencies from 30ksps up to 600ksps.

Memory devices are mostly EEPROM variants. There are also a few SPI flash memories. Capacities range from a couple of bits up to 64KBit. Clock frequencies up to 3MHz. Serial EEPROMS SPI are available for different supply voltages (2.7V to 5V) allowing their use in low-voltage applications. The data retention time duration from 10 years to 100 years. The permitted number of write accesses is 1 million cycles for most components. By cascading memory devices any number of bits/word can be obtained.

RTCs are ideally suited for serial communication because only small amounts of data have to be transferred. There is also a great variety of RTCs with supply voltages from 2.0V. In addition to the standard functions of a "normal" clock, some RTCs offer an alarm function, non-volatile RAM etc. Most RTCs come from DALLAS and EPSON.

The group of the sensors is yet weakly represented. Only a temperature and a pressure sensor could be found.

CAN and USB controllers with SPI make it easier to use these protocols on a micro controller and interfacing a LCD via SPI saves the troublesome parallel wiring.

Note

While we using SPI Interface close the jumpers J12 to J15 as downward direction.

26. ZIGBEE

ZigBee is a technological standard, based on the IEEE 802. 15.4 standard, which was created specifically for control and sensor networks. Within the broad organization of the Institute of Electrical and Electronics Engineers (IEEE), the 802 group is the section that deals with network operations and technologies. Group 15 works more specifically with wireless networking, and Task Group 4 drafted the 802.15.4 standard for a low data rate wireless personal area network (WPAN). The standard for this WPAN specifies not only a low data rate but also low power consumption and low complexity, among other things.

The data rate is limited to 250 kbps in the global 2.4 GHz Industrial, Scientific, Medical (ISM) band, 20 kbps in the 868 MHz band used in Europe, and 40 kbps in the 915 MHz band used in North America and Australia. The rate is depending on band. low data is (20-250 kbps). Zigbee main features are low data rate, low power consumption and small packet device.

The ZigBee standard is built on top of this IEEE standard, addressing remote monitoring and control for sensory network applications. This standard was created by an organization known as the ZigBee Alliance, which is composed of a large number of companies and industry leaders striving to enable such control devices based on said standard.

DEVICE TYPES

There are three different types of ZigBee device:

ZigBee coordinator(ZC)

The most capable device, the coordinator forms the root of the network tree and might bridge to other networks. There is exactly one ZigBee coordinator in each network since it is the device that started the network originally. It is able to store information about the network, including acting as the Trust Centre & repository for security keys.

ZigBee Router (ZR)

As well as running an application function a router can act as an intermediate router, passing data from other devices.

ZigBee End Device (ZED)

Contains just enough functionality to talk to its parent node (either the coordinator or a router); it cannot relay data from other devices. This relationship allows the node to be asleep a significant amount of the time thereby giving you the much quoted long battery life. A ZED requires the least amount of memory, and therefore can be less expensive to manufacture than a ZR or ZC.

HOW ZIGBEE WORKS

ZigBee basically uses digital radios to allow devices to communicate with one another. A typical ZigBee network consists of several types of devices. A network coordinator is a device that sets up the network, is aware of all the nodes within its network, and manages both the information about each node as well as the information that is being transmitted/received within the network.

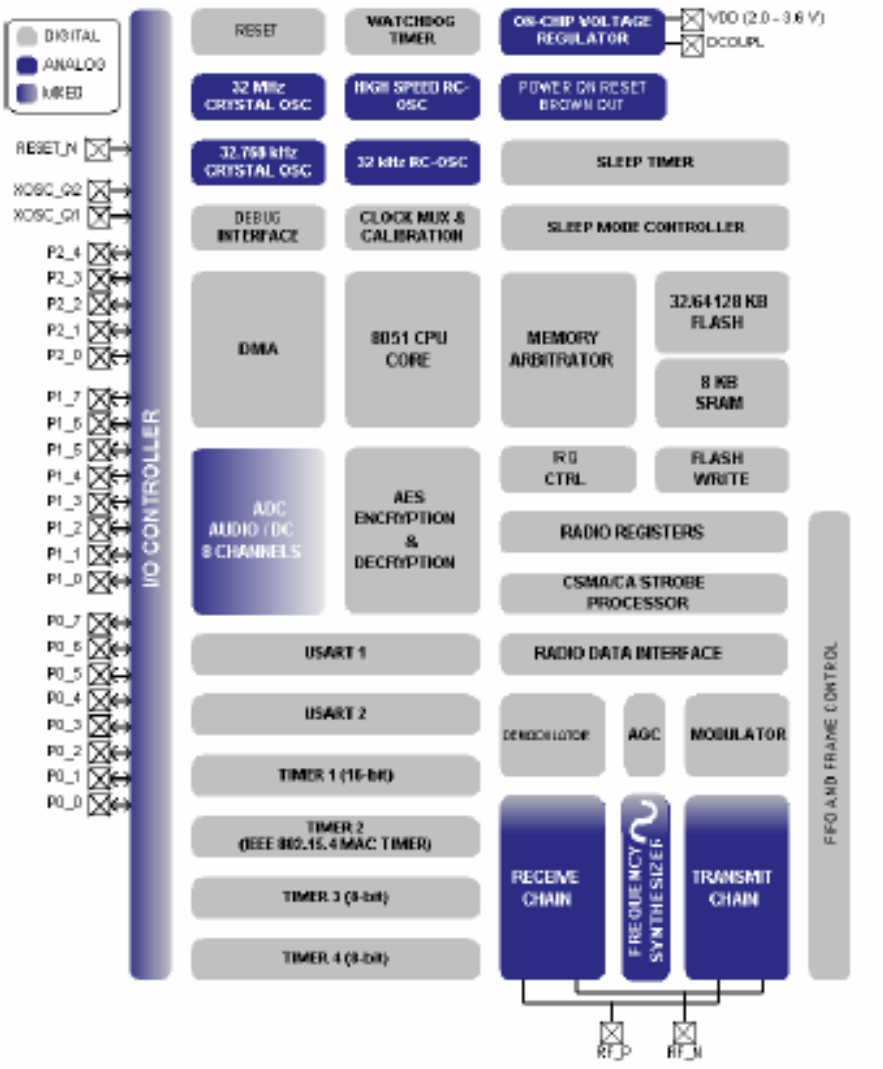
Every ZigBee network must contain a network coordinator. Other Full Function Devices (FFD's) may be found in the network, and these devices support all of the 802.15.4 functions. They can serve as network coordinators, network routers, or as devices that interact with the physical world. The final device found in these networks is the Reduced Function Device (RFD), which usually only serve as devices that interact with the physical world.

The figure given later introduces the concept of the ZigBee network topology. Several topologies are supported by ZigBee, including star, mesh, and cluster tree. Star and mesh networking are both shown in the figure above. As can be seen, star topology is most useful when several end devices are located close together so that they can communicate with a single router node. That node can then be a part of a larger mesh network that ultimately communicates with the network coordinator. Mesh networking allows for redundancy in node links, so that if one node goes down, devices can find an alternative path to communicate with one another. Given below provide an example of how mesh networking allows for multiple paths between devices.

ABOUT CC2431…

Applications

- *2.4 GHz IEEE 802.15.4 systems*
- *ZigBee® systems*
- *Home/building automation*
- *Industrial Control and Monitoring*
- *Low power wireless sensor networks*
- *PC peripherals*
- *Set-top boxes and remote controls*
- *Consumer Electronics*



Description

The CC2431 comes in three different flash versions: CC2431F32/64/128, with 32/64/128 KB of flash memory respectively. The CC2431 is a true System-on-Chip (SoC) solution specifically tailored for IEEE 802.15.4 and ZigBee® applications. It enables ZigBee® nodes to be built with very low total bill-ofmaterial costs. The CC2431 combines the excellent performance of the leading CC2420 RF transceiver with an industry-standard enhanced 8051 MCU, 32/64/128 KB flash memory, 8 KB RAM and many other powerful features. Combined with the industry leading ZigBee® protocol stack (Z-Stack™) from Texas Instruments, the CC2431 provides the market's most competitive ZigBee® solution. The CC2431 is highly suited for systems where ultra low power consumption is required. This is ensured by various operating modes. Short transition times between operating modes further ensure low power consumption.

Key Features

* **RF/Layout**

  - 2.4 GHz IEEE 802.15.4 compliant RF transceiver (industry leading CC2420 radio core)
  - Excellent receiver sensitivity and robustness to interferers
  - Very few external components
  - Only a single crystal needed for mesh network systems
  - RoHS compliant 7x7mm QLP48 package

* **Low Power**

  - Low current consumption (RX: 27 mA, TX:27 mA, microcontroller running at 32 MHz)
  - Only 0.5 µA current consumption in power down mode, where external interrupts or the RTC can wake up the system o 0.3 µA current consumption in stand-by mode, where external interrupts can wake up the system
  - Very fast transition times from low-power modes to active mode enables ultra low average power consumption in low duty cycle systems
  - Wide supply voltage range (2.0V - 3.6V)

* Micro controller

- High performance and low power 8051microcontroller core
- 32, 64 or 128 KB in-system programmable flash
- 8 KB RAM, 4 KB with data retention in all power modes
- Powerful DMA functionality
- Watchdog timer
- One IEEE 802.15.4 MAC timer, one general16-bit timer and two 8-bit timers
- Hardware debug support

* Peripherals

- CSMA/CA hardware support.
- Digital RSSI / LQI support
- Battery monitor and temperature sensor
- 12-bit ADC with up to eight inputs and configurable resolution
- AES security coprocessor
- Two powerful USARTs with support for several serial protocols
- 21 general I/O pins, two with 20mA sink/source capability

DESCRIPTION OF CC4321 DEVELOPMENT KIT

1. CC2431DK content

The development kit contains the following:

- 2 x SmartRF04EB
- 10 x SOC_BB (Battery Board)
- 2 x Evaluation Modules CC2431EM
- 10 x Evaluation Modules CC2431EM
- 12 x 2.4GHz Antennas
- 2 x USB cables
- 1 x RS232 Serial cable
- 1 x 10-wire flat cable for using SmartRF04EB as emulator for external target systems
- 1 x Quick start guide

SmartRF04EB with CC2430EM

Battery board with CC2431EM

The SmartRF04DK Development Kit includes a sample application for location calculation and demonstration. The sample application allows you to:

- Evaluate the SmartRF®04 products. Apply power the boards and the plug and play kit can be used for range testing and location engine testing
- Use SmartRF® Studio to perform RF evaluation and measurements. The radio can be easily configured to measure sensitivity, output power and other RF parameters.

- Develop and test your own firmware. The CC2431DK includes a USB interface that can be used as an emulator interface for the CC2431 or CC2431. All I/O ports are available on pin connectors on the edge of the board to allow easy access for testing or for external I/O. SWRU07

2. System overview

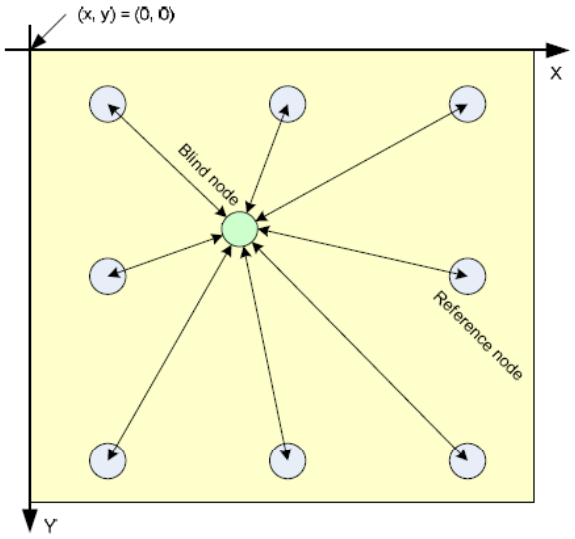


Figure 1: Location Estimation

The location algorithm used in CC2431 Location Engine is based on RSSI (Received Signal Strength Indicator) values. The RSSI value will decrease with increasing distance. An important parameter related to the performance of the location estimate is the absolute tolerance of the RSSI measurement of the RF-transceiver. This is specific to our transceiver and is provided in the datasheet. The main feature of the CC2431 Location Engine is that the location estimation is performed in each Blind Node, hence the algorithm is decentralized.

This reduces the amount of data transferred on the RF interface, since only the calculated position is transmitted. The location engine implemented in CC2431 can use up to 16 Reference Nodes for each calculation. The data that is necessary to do a calculation are X, Y and RSSI values for each of the Reference Nodes used in the system setup. In addition it needs two RF transmission parameters (A and N), please refer to application note AN042 for a description of each parameter.

2.1 Grid

To map each location to a distinct place in the natural environment, a two dimensional grid is used. The direction will, in the following, be named X and Y. The CC2431 Location Engine can only handle two dimensions, but it's possible to handle a third dimension in software (i.e. to represent levels in a building). The point named (X, Y) = (0, 0) is located in upper left corner of the grid.

2.2 Nodes

The CC2431 Location Engine uses the RSSI value combined with the physical location of the Reference Nodes to calculate its own position. Any number of Reference Nodes can be used in the system, but a node can only calculate its position if it is within range of at least three Reference Nodes1. Experiments have shown that one Reference Node for each 100 m2 gives good location estimates. The nodes should be placed in a grid with one node for each 10 meters in both directions.

2.2.1 Reference Node

A node which has static location is called a Reference Node. This node must be configured with an X and a Y value that correspond to the physical location. The main task for a Reference Node is to provide “reference” packets to the Blind Node. Reference packets contain the X and Y coordinates of the Reference Node.

2.2.2 Blind Node

A Blind Node will communicate with its nearest Reference Nodes, collecting X, Y and RSSI values for each of these nodes. Then it uses the location engine hardware to calculate its position based on the collected parameters from several Reference Nodes.

2.2.3 Dongle

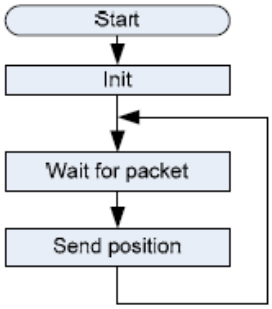
The Dongle will communicate with the entire network; it can request or configure the X,Y values of all Reference Nodes and the A and N values of the Blind Nodes via the Z-location Engine PC Application. The Z-location Engine can also configure any Blind Node to automatically make a periodic position calculation and report (by default the Blind Node is waiting for a command to perform a position calculation.)

2.3 RSSI

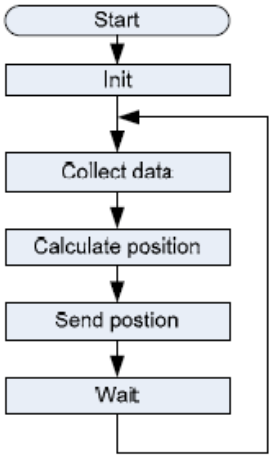
The RSSI value is typically in the range -40 dBm to -90 dBm, where -40 dBm is the highest value. -40 dBm is approximately the measured signal strength on distance of one meter. Input to the location engine hardware is the absolute value of the RSSI in dBm, so the range will typically be from 40 to 90, where 40 is the highest signal strength.

2.4 Program flow

The program flow for both Reference Node and Blind Node are shown in Figure 2 and Figure 3. The figures are simplified.



Reference Node



Blind Node

2.5 Communication Flow

The Z-location Engine is configured to periodically query the entire network – the X,Y of all Reference Nodes is requested and all Blind Nodes are commanded to perform a position calculation. When a Blind Node performs a position calculation, the Over-the-Air (OTA) message traffic can be observed as three phases: broadcast, data collection, & position calculation and reporting.

Broadcast Phase

The Blind Node sends out a 1-hop broadcast to learn the network address of all Reference Nodes that are within radio range. Then the Blind Node sends out a blast of several 1-hop broadcast messages, and any Reference Node receiving such a message shall make a running average of the RSSI of the packets received from a particular Blind Node.

Data Collecting Phase

After the broadcast phase, the Blind Node will send a 1-hop unicast message to every Reference Node in radio range requesting the average RSSI calculated during the broadcast blast.

Position Calculating Phase

In this phase the Blind Node calculates the position and transmits it to the Dongle.
Note :

While we are zigbee close the jumpers J22 to J25 as upward direction.

27. OLED Interface

1. Introduction

Short for ***o****rganic* ***l****ight-****e****mitting diode*, a display device that sandwiches carbon-based films between two charged electrodes, one a metallic cathode and one a transparent anode, usually being glass. The organic films consist of a hole-injection layer, a hole-transport layer, an emissive layer and an electron-transport layer. When voltage is applied to the OLED cell, the injected positive and negative charges recombine in the emissive layer and create electro luminescent light. Unlike LCDs, which require backlighting, OLED displays are emissive devices - they emit light rather than modulate transmitted or reflected light.

The **µOLED** is a compact & cost effective all in one 'SMART" OLED Display with an embedded graphics controller that will deliver 'stand-alone' functionality to your project. The 'simple to use' embedded commands not only control background color but can produce text in a variety of sizes as well as draw shapes (which can include user definable bitmapped characters such as logos) in 256 or 65,536 colors whilst freeing up the host processor from the 'processor hungry ' screen control functions. This means a simple micro controller with a standard serial or USB interface can drive the **µOLED** module with total ease. Figures below show some of the graphics capability of the **µOLED**.

2. µOLED Features

The **µOLED** is aimed at being integrated into a variety of different applications via a wealth of features designed to facilitate any given functionality quickly and cost effectively and thus reduce 'time to market'.

These features are as follows:

- 96 x 64 pixel resolution, 256 or 65K true to life colors, Enhanced OLED screen.
- 0.96" diagonal. Module Size: 32.7 x 23.0 x 4.9mm. Active Display Area: 20mm x 14mm.
- No backlighting with near 180° viewing angle.
- Easy 5 pin interface to any host device: VCC, TX, RX, GND, RESET
- Voltage supply from 3.3V to 6.0V, current @ 40mA nominal when using a 5.0V supply source. Note: The module may need to be supplied with a voltage greater than 4.0 volts when using it with a SD memory card.

- Serial RS-232 (0V to 3.3V) with auto-baud feature (300 to 256K baud). If interfacing to a system greater than 3.6V supply, a series resistor (100 to 220 Ohms) is required on the RX line.
- Powered by the 4D-LABS **GOLDELOX** processor (also available as separate OEM chips for volume users).
- Optional USB to Serial interface via the 4D micro-USB (**µUSB-MB5** or **µUSB-CE5**) modules.
- Onboard micro-SD (**µSD**) memory card adaptor for storing of icons, images, animations, etc. 64Mb to 4Gig µSD memory cards can be purchased separately.
- Three selectable font sizes (5x7, 8x8 and 8x12) for ASCII characters as well as user defined bitmapped characters (32 @ 8x8)
- Built in graphics commands such as: LINE, CIRCLE, RECTANGLE, TEXT, USER BITMAP, BACKGROUND COLOUR, PUT PIXEL, IMAGE, etc. just to name a few

3. µOLED Serial Command Set

The heart of the µOLED-96-G1 is the easy to understand command set. This comprises of a handful of easy to learn instructions that can draw lines, circles, squares, etc, to provide a full text and graphical user interface. The commands are sent to the µOLED-96-G1 via its serial connection (5 pin header). The command set is grouped into 3 sections:

- General Command Set
- Display Specific Command Set
- Extended Command Set

Each Command set is described in detail in the following sections.

NOTE!

The RX and the TX signals are at 3.3V levels. If interfacing to a host system running at voltages greater than 3.6V levels, then a 100 to 220 Ohms series resistor must be inserted between the Host TX and the µOLED RX signals as shown in the diagram below.

4. Command Protocol

The following are each of the commands with the correct syntax. Please note that all command examples listed below are in hex (**00**hex). Due to the high colour depth of the µOLED, a pixel colour value will not fit into a single byte, a byte can only hold a maximum value of 255. Therefore the colour is represented as a 2 byte value, **colour**(msb:lsb). The most significant byte (msb) is transmitted first followed by the least significant byte (lsb). This format is called the big endian. So for a 2 byte colour value of **013F**hex the byte order can be shown as (**01**hex),(**3F**hex).

NOTE:

When transmitting the command and data bytes to the µOLED, do not include any separators such as commas ‘,’ or spaces ‘ ‘ or brackets ‘(‘ ‘)’ between the bytes. The examples show these separators purely for legibility; these must not be included when transmitting data to the µOLED.

When a command is sent, the µOLED will reply back with a single acknowledge byte called the **ACK** (**06**hex). This tells the host that the command was understood and the operation is completed. It will take the µOLED anywhere between 1 to several milliseconds to reply back with an **ACK**, depending on the command and the operation the µOLED has to perform. If the µOLED receives a command that it does not understand it will reply back with a negative acknowledge called the **NAK** (**15**hex).

If a command that has 5 bytes but only 4 bytes are sent, the command will not be executed and the µOLED will wait until another byte is sent before trying to execute the command. There is no timeout on the µOLED when incomplete commands are sent. The µOLED will reply back with a **NAK** for each invalid command it receives. For correct operation make sure the command bytes are sent in the correct sequence.

5. General Command Set

5.1. Add User Bitmapped Character (A)

Syntax : **cmd, char#, data1, data2, ……., dataN**
cmd : **41**hex, **A**ascii
char# : bitmap character number to add to memory:
0 to 31 (**00**h to **1F**h), 32 characters of 8x8 format.
data1 to dataN : Number of data bytes that make up the composition and format of the bitmapped character. The 8x8 bitmap composition is 1 byte wide (8bits) by 8 bytes deep which makes N = 1x8 = 8.
Description: This command will add a user defined bitmapped character into the internal memory.

Example1:

41hex, **01**hex, **18**hex, **24**hex, **42**hex, **81**hex, **81**hex, **42**hex, **24**hex, **18**hex

This adds and saves user defined 8x8 bitmap as character number 1 into memory as seen below.

5.2. Set Background Colour (B)

Syntax : **cmd, colour**(msb:lsb)
cmd : **42**hex, **B**ascii
colour(msb:lsb) : pixel colour value: 2 bytes (16 bits) msb:lsb 65,536 colours to choose from
Black = **0000**hex, **0**dec
White = **FFFF**hex, **65,535**dec, **1111111111111111**bin

Description :

This command sets the current background colour. Once this command is sent, only the background colour will change. Any other object on the screen with a different colour value will not be affected.

Example :

42hex, **FFFF**hex Set the background colour to value 65,535 (**white**).

5.3 Text button (b)

Syntax : cmd, state, x, y, buttonColour(msb:lsb), **font, Text Colour**(msb:lsb), **textWidth, textHeight, char1, .., charN, terminator**

cmd : 62hex, **b**ascii

state : Specifies whether the displayed button is drawn as **UP** (not pressed) or **DOWN** (pressed). 0 = Button Down (pressed) 1 = Button Up (not pressed)

x : top left horizontal start position of the button

y : top left vertical start position of the button

buttonColour(msb:lsb) **:** 2 byte button colour value

font : 0 = 5x7 font, 1 = 8x8 font, 2 = 8x12 font. This has precedence and does not affect the Font command.

textColour(msb:lsb) **:** 2 byte text colour value

textWidth : horizontal size of the character, effects the width of the button.

textHeight : vertical size of the character, effects the height of the button.

char1..charN : string of ASCII characters (limit the string to line width)

terminator : the string must be terminated with **00**hex

Description :

This command will place a Text button similar to the ones used in a PC Windows environment. **(x, y)** refers to the top left corner of the button and the size of the button is automatically calculated and drawn on the screen with the text relatively justified inside the button box. The button can be displayed in an UP (button not pressed) or DOWN (button pressed) position by specifying the appropriate value in the **state** byte. Separate button and text colours provide many variations in appearance and format.

5.4 Draw Circle (C)

Syntax : cmd, x, **y**, **rad**, **colour**(msb:lsb)

cmd : 43hex**, C**ascii

x : circle centre horizontal position. **0**dec to **95**dec (**00**hex to **5F**hex).

y : circle centre vertical position. **0**dec to **63**dec (**00**hex to **37F**hex).

rad : radius size of the circle. **0**dec to **63**dec (**00**hex to **3F**hex).

colour(msb:lsb) **:** 2 byte circle colour value

Description :

This command will draw a coloured circle centred at **(x, y)** with a radius determined by the value of **rad**. The circle can be either solid or wire frame (empty) depending on the value of the Pen Size (see **Set Pen Size** command). When Pen Size = 0 circle is solid, Pen Size = 1 circle is wire frame.

Example :

43hex**, 20**hex**, 20**hex**, 10**hex, **00**hex**, 1F**hex Draws a RED circle (**001F**hex) centred at x = **32**dec (**3F**hex) and y = **32**dec (**3F**hex) with a radius of **16**dec (**20**hex).

5.5 Block copy & Paste (Screen Bitmap Copy) (c)

Syntax : cmd, xs, **ys**, **xd**, **yd**, **width**, **height**

cmd : 63hex**, c**ascii

xs: top left horizontal start position of block to be copied (source)

ys: top left vertical start position of block to be copied (source)

xd: top left horizontal start position of where copied block is to be pasted (destination)

yd: top left vertical start position of where the copied block is to be pasted (destination)

width: width of block to be copied (source)

height: height of block to be copied (source)

Description :

This command copies an area of a bitmap block of specified size. The start location of the block to be copied is represented by **xs, ys** (top left corner) and the size of the area to be copied is represented by **width** and **height** parameters. The start location of where the block is to be pasted (destination) is represented by **xd, yd** (top left corner).

This is a very powerful feature for animating objects, smooth scrolling, implementing a windowing system or copying patterns across the screen to make borders or tiles.

5.6 Display User Bitmapped Character (D)

Syntax : cmd, char#, x, **y**, **colour**(msb:lsb)

cmd : 44hex, **D**ascii

char# : which user defined character number to display from the selected group. **0**dec to **31**dec (**00**hex to **1F**hex), of 8x8 format.

x : horizontal display position of the character. **0**dec to **95**dec (**00**hex to **5F**hex).

y : vertical display position of the character. **0**dec to **63**dec (**00**hex to **3F**hex).

colour(msb:lsb) **:** 2 byte bitmap colour value

Description :

This command displays the previously defined user bitmapped character at location **(x, y)** on the screen. User defined bitmaps allow drawing & displaying unlimited graphic patterns quickly & effectively.

Example 1:

44hex, **01**hex , **00**hex, **00**hex, **F8**hex, **00**hex Display 8x8 bitmap character number 1 at x = 0, y = 0, colour = red

Example 2:

44hex, **01**hex, **08**hex, **00**hex, **07**hex, **E0**hex Display 8x8 bitmap character number 1 at x = 8, y = 0, colour = green

Example 3:

44hex , **01**hex, **10**hex, **00**hex, **00**hex, **1F**hex Display 8x8 bitmap character number 1 at x = 16, y = 0, colour = blue

5.7 Erase Screen (E)

Syntax : cmd

cmd : 45hex, **E**ascii

Description :

This command clears the entire screen using the current background colour.

Example :

45hex Clear the screen.

5.8 Set Font Size (F)

Syntax : cmd, size

cmd : 46hex, **F**ascii
size : **= 00**hex : 5x7 small size font
= 01hex : 8x8 medium size font
= 02hex : 8x12 large size font

Description :

This command will change the size of the font according to the value set by **size**. Changes take place after the command is sent. Any character on the screen with the old font size will remain as it was.

Example1: 46hex, **00**hex Select small 5x7 fonts

Example1: 46hex, **01**hex Select medium 8x8 fonts

Example1: 46hex, **02**hex Select large 8x12 fonts

5.9 Draw TrianGle (G)

Syntax : cmd, x1, **y1**, **x2**, **y2**, **x3**, **y3**, **colour**(msb:lsb)

cmd : 47hex**, G**ascii

x1, y1, x2, y2, x3, y3 : 3 vertices of the triangle. These must be specified in an anticlockwise fashion.

colour(msb:lsb) **:** 2 byte triangle colour value

Description : This command draws a Solid/Empty triangle. The vertices must be specified in an anti-clock wise manner, i.e. **x2 < x1, x3 > x2, y2 > y1, y3 > y1**.

A solid or a wire frame triangle is determined by the value of the Pen Size setting, i.e.

0 = solid, 1 = wire frame.

5.10 Draw Polygon (g)

Syntax : cmd, vertices, x1, **y1**, **.. .. .. .. xn**, **yn**, **colour**(msb:lsb)

cmd : 67hex**, g** ascii

vertices : number of vertices from 3 to 7. Specifies the number of vertices of the polygon.

(x1, y1) .. .. .. (xn, yn) : vertices of the polygon. These can be specified in any fashion.

colour(msb:lsb) **:** 2 byte polygon colour value

Description :

This command draws an Empty/Wire Frame polygon. Up to 7 vertices can be specified in any manner. Currently only a wire frame polygon is supported.

5.11 Display Image (I)

Syntax : cmd, x, **y**, **width**, **height**, **colourMode**, **pixel1**, .. **pixelN**

cmd : 49hex**, I**ascii

x : Image horizontal start position (top left corner)

y : Image vertical start position (top left corner)

width : horizontal size of the image

height : vertical size of the image

colourMode : 8dec = 256 colour mode, 8bits/1byte per pixel

16dec = 65K colour mode, 16bits/2bytes per pixel (msb:lsb)

pixel1..pixelN : image pixel data and N is the total number of pixels

N = height x width when colourMode = 8

N = height x width x 2 when colourMode = 16

Description :

This command displays a bitmap image on to the screen with the top left corner specified by **(x, y)** and size of the image specified by **width** and **height** parameters. This command is more effective than using the “Put Pixel” command, where there are no overheads in specifying the **x, y** location of each pixel.

5.12 Draw Line (L)

Syntax : cmd, x1, **y1**, **x2**, **y2**, **colour**(msb:lsb)

cmd : 4Chex, **L**ascii

x1 : horizontal position of line start. **0**dec to **95**dec (**00**hex to **5F**hex).

y1 : vertical position of line start. **0**dec to **63**dec (**00**hex to **3F**hex).

x2 : horizontal position of line end. **0**dec to **95**dec (**00**hex to **5F**hex).

y2 : vertical position of line end. **0**dec to **63**dec (**00**hex to **3F**hex).

colour(msb:lsb) **:** 2 byte line colour value

Description :

This command will draw a coloured line from point **(x1, y1)** to point **(x2, y2)** on the screen.

Example :

4Chex, **00**hex, **00**hex, **5F**hex, **3F**hex, **FF**hex, **Ff**hex Draws a white line from (x1=0, y1=0) to (x2=95, y2=63).

5.13 Opaque / Transparent Text (O)

Syntax : cmd, mode

cmd : 4Fhex, **O**ascii

mode : = 00hex : Transparent Text, objects behind the text can be seen.
= 01hex: Opaque Text, objects behind text is blocked by background

Description :

This command will change the attribute of the text so that an object behind the text can either be blocked or transparent. Changes take place after the command is sent. This command will change the attribute so that when a character is written, it will either write just the character alone (Transparent Mode) so any original character will be seen as well as the new, or overwrite any existing data with the new character.

Example1: 4Fhex, **00**hex Transparent Text Mode
Example2: 4Fhex, **01**hex Opaque Text Mode

5.14 Put Pixel (P)

Syntax : cmd, x, **y**, **colour**(msb:lsb)

cmd : 50hex**, P**ascii

x : horizontal pixel position. **0**dec to **127**dec (**00**hex to **7F**hex).

y : vertical pixel position. **0**dec to **127**dec (**00**hex to **7F**hex).

colour : pixel colour value: 2 bytes (16 bits) msb, lsb 65,536 colours to choose from
Black = **0000**hex, **0**dec
White = **FFFF**hex, **65,535**dec, **1111111111111111**bin

Description :

This command will put a coloured pixel at location **(x, y)** on the screen.

Example :

50hex**, 01**hex**, 0A**hex**, FF**hex**, Ff**hex Puts a white (**FFFF**hex) pixel at location x = **01**dec (**01**hex) and y = **10**dec (**0A**hex).

5.15 Set pen Size (p)

Syntax : cmd, size

cmd : 70hex**, p** ascii

size : = **00**hex : All objects such as circles, rectangles, triangles, etc are solid
= **01**hex : All objects are wire frame (empty)

Description :

This command determines if certain graphics objects are drawn in solid or wire frame fashion.

Example1 : 70hex**, 00**hex All objects will be drawn solid

Example1 : 70hex**, 01**hex All objects will be drawn wire frame.

5.16 Read Pixel (R)

Syntax : cmd, x, y

cmd : 52hex, **R**ascii

x : horizontal pixel position. **0**dec to **95**dec (**00**hex to **5F**hex).

y : vertical pixel position. **0**dec to **63**dec (**00**hex to **3F**hex).

Description :

This command will read the colour value of pixel at location **(x, y)** on the screen and return it to the host. This is a useful command when for example a white pointer is moved across the screen and the host can read the colour on the screen and switch the colour of the pointer when it's on top of a light coloured area.

Example : 52hex, **01**hex, **01**hex

µOLED reply : 00hex, **1F**hex

Reads a blue (**001F**hex) pixel at location x = **1**dec (**01**hex) and y = **1**dec (**01**hex).

5.17 Draw rectangle (r)

Syntax : cmd, x1, **y1**, **x2**, **y2**, **colour**(msb:lsb)

cmd : 72hex**, r** ascii

x1 : top left horizontal start position of rectangle. **0**dec to **95**dec (**00**hex to **5F**hex).

y1 : top left vertical start position of rectangle . **0**dec to **63**dec (**00**hex to **3F**hex).

x2 : bottom right horizontal end position. **0**dec to **95**dec (**00**hex to **5F**hex).

y2 : bottom right vertical end position. **0**dec to **63**dec (**00**hex to **3F**hex).

colour(msb:lsb) **:** 2 byte rectangle colour value

Description :

This command will draw a rectangle of specified area on the screen. **x1, y1** refers to the top left corner of the area and **x2, y2** refers to the bottom right hand corner of the rectangle on the screen. If colour is chosen to be that of the background then the effect will be erasure. If Pen Size value was previously set to 0 rectangle will be solid, otherwise wire frame if value was 1.

Example :

70hex**, 00**hex**, 00**hex**, 10**hex**, 10**hex**, 00**hex**, 1F**hex Draws a RED (001Fhex) rectangle that has its top left corner at x1=0, y1=0 and its bottom right corner at x2=16, y2=16.

5.18 Place String of Ascii Text (unformatted) (S)

Syntax : cmd, x, **y**, **font**, **colour**(msb:lsb), **width, height, char1, .. , charN, Terminator**

cmd : 53hex**, S**ascii

x : the horizontal start position of string (in pixels).

y : the vertical start position of string (in pixels).

font : 0 = 5x7 font, 1 = 8x8 font, 2 = 8x12 font. This has precedence over the Font command but does not effect the previous font selection.

colour(msb:lsb) **:** 2 byte colour value of the string.

width : horizontal size of the string characters, n x normal size

height : vertical size of the string characters, m x normal size

char1..charN : string of ASCII characters (max 256 characters)

terminator : the string must be terminated with **00**hex

Description :

This command allows the display of a string of bitmapped (unformatted) ASCII characters. The horizontal start position of the string is specified by **x** and the vertical position is specified by **y**. The string must be **terminated** with **00**hex. The sizes of the characters are determined by the **width** and **height** parameters. If the length of the string is longer than the maximum number of characters per line then, a wrap around will occur on to the next line. Maximum string length is **256 bytes**.

5.19 Place string of Ascii Text (formatted) (s)

Syntax : cmd, column, **row**, **font**, **colour**(msb:lsb), **char1,.., charN, terminator**

cmd : 73hex**, s**ascii

column : horizontal start position of string:

0 - 15 for 5x7 font, **0 - 11** for 8x8 and 8x12 font.

row : vertical start position of string:

0 - 7 for 5x7 and 8x8 font, **0 – 4** for 8x12 font.

font : 0 = 5x7 font, 1 = 8x8 font, 2 = 8x12 font. This has precedence over the Font command.

colour(msb:lsb) **:** 2 byte colour value of the string.

char1..charN : string of ASCII characters (max 256 characters)

terminator : the string must be terminated with **00**hex

Description :

This command allows the display of a string of ASCII characters. The horizontal start position of the string is specified by **column** and the vertical position is specified by **row**. The string must be **terminated** with **00**hex. If the length of the string is longer than the maximum number of characters per line then, a wrap around will occur on to the next line. Maximum string length is **256 bytes**.

5.20 Place Text Character (formatted) (T)

Syntax : cmd, char, column, row, **colour**(msb:lsb)

cmd : 54hex**, T**ascii

char : inbuilt standard ASCII character, **32**dec to **127**dec (**20**hex to **7F**hex)

column : horizontal position of character, see range below:

0 - 15 for 5x7 font, **0 - 11** for 8x8 and 8x12 font.

row : vertical position of character:

0 - 7 for 5x7 and 8x8 font, **0 – 4** for 8x12 font.

colour(msb:lsb) **:** 2 byte colour value of the character.

Description :

This command will place a coloured ASCII character (from the ASCII chart) on the screen at a location specified by **(column, row)**. The position of the character on the screen is determined by the predefined horizontal and vertical positions available, namely 0 to 15 columns by 0 to 7 rows.

Example :

54hex**, 41**hex**, 00**hex**, 00**hex**, FF**hex, **Ff**hex Place character 'A' (**41**hex) at column = 0, row = 0, colour = white (65,535).

5.21 Place text Character (unformatted) (t)

Syntax : cmd, char, **x**, **y**, **colour**(msb:lsb), **width**, **height**

cmd : 74hex**, t**ascii

char : inbuilt standard ASCII character, **32**dec to **127**dec (**20**hex to **7F**hex)

x : the horizontal position of character (in pixel units).

y : the vertical position of character (in pixel units).

colour(msb:lsb) **:** 2 byte colour value of the character.

width : horizontal size of the character, n x normal size

height : vertical size of the character, m x normal size

Description :

This command will place a coloured built in ASCII character anywhere on the screen at a location specified by **(x, y)**. Unlike the '**T**' command, this option allows text of any size (determined by **width** and **height**) to be placed at any position. The font of the character is determined by the '**Font Size**' command.

5.22 OLED DisplaY Control Functions (Y)

Syntax : cmd, mode, value

cmd : 59hex, **Y**ascii

mode : = 00hex : **BACKLIGHT CONTROL**.

value = XXhex: has no effect as there is no backlighting on the OLED display. This is only retained for legacy.

mode : = 01hex : **DISPLAY ON/OFF**.

value = 00hex: Display OFF
= 01hex: Display ON

mode : = 02hex : **OLED CONTRAST**.

value = 0dec to **15**dec : Contrast range (default = 15dec)

mode : = 03hex : **OLED POWER-UP/POWER-DOWN**.

value = 00hex: OLED Power-Down
= 01hex: OLED Power-Up

Note:

It is important that the **µOLED** be issued with the Power-Down command before switching off the power. This command switches off the internal voltage boosters and current amplifiers and they need to be turned off before main power is removed. If the power is removed without issuing this command, the OLED display maybe damaged (over a period of time). This command also turns off the display. This command need not only be issued to shutdown but can be issued to conserve power by turning off the display and the backlight. The Power-Up command does not need to be executed when applying power. If a Power-Down command has been issued and Power is not switched off, the Power-Up command can be sent to Power the display back up again.

5.23 Version/Device Info Request (V)

Syntax : cmd, output

Response : device_type, hardware_rev, firmware_rev, horizontal_res, vertical_res

cmd : 56hex, **V**ascii

output : 00hex : outputs the version and device info to the serial port only.

01hex : outputs the version and device info to the serial port as well as to the screen.

device_type : this response indicates the device type.

00hex = micro-OLED.
01hex = micro-LCD.
02hex = micro-VGA.

hardware_rev : this response indicates the device hardware version.

firmware_rev : this response indicates the device firmware version.

horizontal_res : this response indicates the horizontal resolution of the display.

22hex : 220 pixels
28hex : 128 pixels
32hex : 320 pixels
60hex : 160 pixels
64hex : 64 pixels
76hex : 176 pixels
96hex : 96 pixels

vertical_res : this response indicates the vertical resolution of the display. See horizontal_res above for resolution options.

Description :

This command requests all the necessary information from the module about its characteristics and capability.

6. Display Specific Command Set

Different OLED display panels that are used in the µOLED range of intelligent display modules have certain built in features that are controlled directly by the embedded driver controller. These features otherwise would be too cumbersome to implement in firmware and would require resources that are not available. The Display Specific Command set utilises these built in hardware features directly. These are detailed in this section.

Display Specific Command Set

1. Display Scroll Control ($S)

Syntax : spCmd, cmd, register, data

spCmd : 24hex, **$**ascii

cmd : 53hex, **S**ascii

reg : Scroll Control Register.

register data
0x00 (Scroll Enable/Disable) 0 = Disable, 1 = Enable
0x01 Reserved XXX
0x02 (Scroll Speed) 1 = fast, 2 = normal, 3 = slow

data : Scroll register data. Refer to above for detail.

Description : This command allows control of screen scrolling.

2. Dim Screen Area ($D)

Syntax : spCmd, cmd, x, y, width, height

spCmd : 24hex**, $**ascii

cmd : 44hex**, D**ascii

x : horizontal start position of screen area to dim (top left corner)

y : vertical start position of screen area to dim (top left corner)

width : horizontal size of the area to dim

height : vertical size of the area to dim

Description : This command allows a portion of the screen to be dimmed to achieve certain effects such as highlight control, etc.

7. Extended Command Set

The following commands are related to the µOLED-96-G1 extended command set and they are described in this section. The µOLED-96-G1 has an integrated micro-SD (**µSD**) memory card adaptor and can accept memory cards of any size from 64Mb up to 4Gig for storing of text, images, icons, animations, movie clips and all other graphics objects. To utilise the Extended Command set, a **µSD** memory card must be inserted into the module since all of these commands are based around the memory card.

You will find references being made to "**Objects**" throughout this section. An object can be simply defined as those commands that reside inside the memory card (programmed/downloaded previously) and can be displayed on the screen by the **"Display Object from Memory Card"** command. The idea of programming objects into the memory card is so that they can be automatically replayed back like a slide show without any host processor intervention.

There are also some commands that can only reside inside the card and must be executed from there. These commands will return a NAK if executed live from the serial link.

7.1. Initialise Memory Card (@i)

Syntax : extCmd, cmd

extCmd : 40hex, **@**ascii

cmd : 69hex, **i** ascii

Description :

This command initialises the **µSD** memory card. The memory card is always initialised upon Power-Up or Reset cycle, if the card is present. If the card is inserted after the power up or a reset then this command must be used to initialize the card.

7.2. Read Sector Data from Memory Card (@R)

Syntax : extCmd, cmd, SectorAddress(hi:mid:lo)

extCmd : 40hex, **@**ascii

cmd : 52hex, **R**ascii

SectorAddress(hi:mid:lo)**:** A 3 byte sector address. Sector Address range from 0 to 16,777,215 depending on the capacity of the card. Each sector is 512 bytes in size. There are 2048 sectors per every 1Mb of card memory.

Description :

This command provides a means of reading data back from the memory card in lengths of 512 bytes. It maybe useful in validating the data that was stored previously using the Write Sector command. Once this command is sent, the µOLED will return 512 bytes of data relating to that particular sector.

7.3. Write Sector Data to Memory Card (@W)

Syntax : extCmd, cmd, SectorAddress(hi:mid:lo), **data**(1)**, .. , data**(512)

extCmd : 40hex**, @**ascii

cmd : 57hex**, W**ascii

SectorAddress(hi:mid:lo)**:** A 3 byte sector address. Sector Address range from 0 to 16,777,215 depending on the capacity of the card. Each sector is 512 bytes in size. There are 2048 sectors per every 1Mb of card memory. **data**(1)**, .. , data**(512)**:** 512 bytes of sector data. The data length must be 512 bytes long. Unused bytes must be padded even if not all are used.

Description :

This command allows downloading of objects such as images and other commands for storage that can be retrieved and used later on. It can also be used as general purpose storage for user specific data. Downloads must always be limited to 512 bytes in length. For large objects such as images, the data must be broken up into multiple sectors (chunks of 512 bytes) and this command then maybe used many times until all of the data is written into the card. If the data block to be written is less than 512 bytes in length, then make sure the rest of the remaining data are padded with 00hex or FFhex (it can be anything).

If only few bytes of data are to be written then the **Write Byte** command can be used.

Once this command message is sent, the µOLED will take a few milliseconds to write the data into its memory card and at the end of which it will reply back with an **ACK**(06hex) if the write cycle was successful. If there was a problem in writing the data to the card a **NAK**(15hex) will be sent back without any write attempts. Only **data**(1) to **data**(512) are stored in the card. Other bytes in the command message such as Sector Address are not stored.

7.4. Read Byte Data from Memory Card (@r)

Syntax : extCmd, cmd

extCmd : 40hex, **@**ascii

cmd : 72hex, **r** ascii

Description :

This command provides a means of reading a single byte of data back from the memory card. Before this command can be used the card memory address location must be set using the **Set Memory Address** command. Once this command is sent, the µOLED will return 1 byte of data relating to that memory location set by the memory Address pointer. The memory Address location pointer is automatically incremented to the next address location.

7.5. Write Byte Data to Memory Card (@w)

Syntax : extCmd, cmd, data

extCmd : 40hex, **@**ascii

cmd : 77hex, **w** ascii

data : 1 byte of memory card data.

Description :

This command allows writing single bytes of data to the memory card. This is useful for writing small chunks of data relating to graphics objects or user application specific data for general purpose storage. For large data blocks it is more efficient to use the **Write Sector Data** command described in the previous section. Before this command can be used the card memory address location must be set using the **Set Memory Address** command. Once this command is sent, the Moled will write 1 byte of data relating to that memory location set by the memory Address pointer. The memory Address location pointer is automatically incremented to the next address location.

Only the **data** byte is stored in the card. Other bytes in the command message are not stored.

7.6. Set Memory Address (@A)

Syntax : extCmd, cmd, Address(Umsb:Ulsb:Lmsb:Llsb)
extCmd : 40hex, **@**ascii
cmd : 41hex, **A**ascii
Address(Umsb:Ulsb:Lmsb:Llsb)**:** A 4 byte memory card address for byte wise access.

Description :

This command sets the card memory Address pointer for byte wise reads and writes. After a byte read or write the Address pointer is automatically incremented internally to the next Address location.

7.7. Copy Screen to Memory Card (@C)

Syntax : extCmd, cmd, x, y, width, height, SectorAddress(hi:mid:lo)
extCmd : 40hex, **@**ascii
cmd : 43hex, **C**ascii
x : Screen horizontal start position (top left corner)
y : Screen vertical start position (top left corner)
width : horizontal size of the screen area to be copied
height : vertical size of the screen area to be copied
SectorAddress(hi:mid:lo)**:** A 3 byte sector address where the copied screen area is to be stored.

Description :

This command copies an area of the screen of specified size. The start location of the block to be copied is represented by **x, y** (top left corner) and the size of the area to be copied is represented by **width** and **height** parameters. This is similar the **Block Copy and Paste** command but instead of the copied screen area being pasted to another location on the screen it is stored into the memory card. The stored screen image can then be later recalled from the memory card and redisplayed onto the screen at the same or different location by using the **Display Image/Icon from Memory Card** command.

This is a very powerful feature for animating objects, smooth scrolling, or implementing a windowing system.

7.8. Display Image/Icon from Memory Card (@I)

Syntax : extCmd, cmd, x, y, width, height, colourMode, SectorAddress(hi:mid:lo)

extCmd : 40hex, **@**ascii

cmd : 49hex, **I**ascii

x : Screen horizontal start position (top left corner)

y : Screen vertical start position (top left corner)

width : horizontal size of the Image/Icon

height : vertical size of the Image/Icon

colourMode : 8dec = 256 colour mode, 8bits/1byte per pixel

16dec = 65K colour mode, 16bits/2bytes per pixel

SectorAddress(hi:mid:lo)**:** A 3 byte memory card sector address of a previously stored Image or an Icon that is about to be displayed.

Description : This command displays a bitmap image or an icon on to the screen that has been previously stored at a particular sector address in the memory card. The screen position of the image to be displayed is specified by **(x, y)** and the size of the image by **width** and **height** parameters.

If the previously stored image was in 8 bit colour format (1 byte per pixel) or 16 bits (2 bytes per pixel) then this must be specified in the **colourMode** byte parameter. Do not store an image/icon in one colour format then display it in another colour format, this will result in a corrupted image display.

Notes:

- The **Copy Screen to Memory Card** command always stores that part of the screen as a 16 bit image, i.e. 2 bytes per pixel.
- The images or icons when stored into the memory card must be sector boundary aligned, i.e. the object start location must be at the start of a sector boundary.

7.9. Display Object from Memory Card(@O)

Syntax : extCmd, cmd, Address(Umsb:Ulsb:Lmsb:Llsb)

extCmd : 40hex, **@**ascii

cmd : 4Fhex, **O**ascii

Address(Umsb:Ulsb:Lmsb:Llsb)**:** A 4 byte (32 bit) memory address of a previously stored Object that is about to be displayed.

Description:

Some of the commands can be stored as objects in the memory card which can be later recalled by the host on demand and displayed or executed. The user must make sure the 32 bit address of each stored command/object is known before using this feature.

For example, a series of images can be stored as icons and later displayed as the application requires them. The table at the end of this section lists all of the commands that can be stored as objects within the memory card.

7.10. Run Program from Memory Card (@P)

Syntax : extCmd, cmd, Address(Umsb:Ulsb:Lmsb:Llsb)

extCmd : 40hex, **@**ascii

cmd : 50hex, **P**ascii

Address(Umsb:Ulsb:Lmsb:Llsb)**:** A 4 byte memory card address for the internal command execution.

Description :

The Run command forces the 32bit internal memory pointer to jump to the specified address and automatically start executing commands, from the memory card without any further interaction by the host processor. It will sequentially execute any valid memory related commands and display objects until it gets to the end of the memory. It is advisable to have the **Exit Program** or the **Jump to Address** commands at the end of the user composed program so that the pointer does not run off so to speak.

7.11. Delay (07hex) (memory card command only)

Syntax : cmd, value(msb:lsb)

cmd : 07hex

value(msb:lsb) **:** A 2 byte delay value in milliseconds. Maximum value of 65,535 milliseconds or 65.5 seconds.

Description :

When objects from the memory card such as images are displayed sequentially, a delay can be inserted between subsequent objects. A delay basically has the same effect as a NOP (No Operation) which can be used to determine how long the object stays on the screen before the next object is displayed.

7.12. Set Counter (08hex) (memory card command only)

Syntax : cmd, value

cmd : 08hex

value : A 1 byte counter value that can be used with **Decrement Counter** and **Jump to Address If Counter Not Zero** commands to form loops. Practical values should be between 2 and 255.

Description :

A series of images that might be part of an animation may need to be redisplayed over and over to achieve a lengthy viewing. This command when used in conjunction with Decrement Counter and Jump to Address If Counter Not Zero commands allow the user to determine exactly how many times the series of images are looped.

For example, we may want to animate the Globe rotating. Let's say we have 10 image slides of the Globe at different rotated positions residing in the memory card. When the images are displayed sequentially, the effective duration will only be the length of time it takes to display the 10 image frames. We can increase that length by looping through the animation a number of times depending on the value set in the counter.

When the display reaches the end of the last frame and encounters the Decrement Counter followed by Jump to Address If Counter Not Zero commands, the counter will be decremented and then the internal pointer will jump to the memory Address specified in the “Jump to Address If Counter Not Zero” command. This sequence will repeat until the value in the counter reaches zero. The following demonstrates how this maybe used:

Address (dec)	**Command**
00000000	Set Counter (value = 25),
00000002	Display Image from Memory Card (image1),
00000012	Delay(10ms),
00000015	Display Image from Memory Card (image2),
00000025	Delay(10ms),
	…,
00000119	Display Image from Memory Card (image10),
00000129	Delay(10ms),
00000132	Decrement Counter
00000134	Jump to Address if Counter Not Zero (Address = 00000002)

Note :

The above example is typical of how a series of commands might be loaded into the memory card and then executed by using the Run Program from Memory Card command. The commands would off course be the series of hex codes.

7.13. Decrement Counter (09hex) (memory card command only)

Syntax : cmd, value

cmd : 08hex

Description :

Decrements the counter. See detailed description on how this command can be used effectively in the **Set Counter** command section.

7.14. Jump to Address If Counter Not Zero (0Ahex) (memory card command only)

Syntax : cmd, Address(Umsb:Ulsb:Lmsb:Llsb)

cmd : 0Ahex

Address(Umsb:Ulsb:Lmsb:Llsb)**:** A 4 byte (32 bit) memory jump address if the counter is not zero.

Description :

If the internal counter is not zero the program pointer will jump to the specified address. If the counter is zero then it will continue executing the next command. Please see detailed description on how this command can be used effectively in the **Set Counter** command section.

7.15. Jump to Address (0Bhex) (memory card command only)

Syntax : cmd, Address(Umsb:Ulsb:Lmsb:Llsb)
cmd : 0Bhex
Address(Umsb:Ulsb:Lmsb:Llsb)**:** A 4 byte (32 bit) memory jump address.

Description :

This command will force the internal 32 bit program memory pointer to jump unconditionally to the specified address and start executing commands from there.

7.16. Exit Program from Memory Card (0Chex)

Syntax : cmd
Cmd : 0Chex

Description :

This command forces the program to stop executing from the memory card and ready to accept and execute commands from the host via the serial interface. When the internal program memory pointer encounters this command it will force the command execution from memory card to stop. It can also be sent via the serial port while the program is running and commands are being executed from the memory card.

28. TFT LCD Interface (Thin Film Transistor Liquid Crystal Display):

Introduction

Liquid crystal was discovered by the Austrian botanist Fredreich Rheinizer in 1888. "Liquid crystal" is neither solid nor liquid (an example is soapy water). In the mid-1960s, scientists showed that liquid crystals when stimulated by an external electrical charge could change the properties of light passing through the crystals.

The early prototypes (late 1960s) were too unstable for mass production. But all of that changed when a British researcher proposed a stable, liquid crystal material (biphenyl). Today's color LCD TV and LCD Monitors have a sandwich-like structure with liquid crystal filled between two glass plates.

There have been countless millions of Nokia cell phones sold world-wide and this economy of scale has made it possible for the hobbyist and experimenter to procure the LCD graphic display from these phones at a reasonable price. Sparkfun Electronics sells a model 6100 for $19.95 (US). I've seen sources for this display on EBay for $7.99 (US) plus $10.00 shipping (from Hong Kong, so shipping is a bit slow). The Swedish web shop Jelu has this display for about $20.00 (US) also (see photograph below).

Olimex uses these displays on their more sophisticated development boards, so this tutorial will be geared to the Olimex SAM7-EX256 board shown on the front cover.

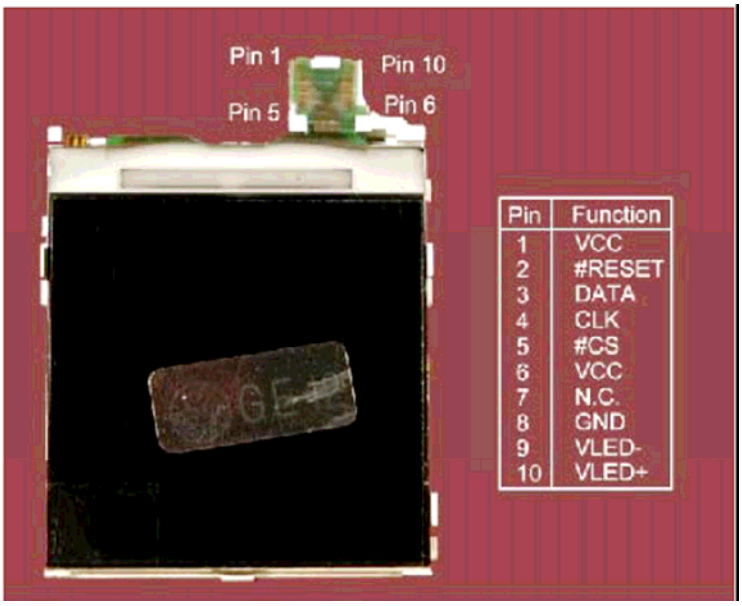


Figure 1. Nokia 6100 LCD Display (from Jelu web site)

The important specifications for this display are as follows:

132 x 132 pixels

12-bit color rendition (4 bits red, 4-bits green, 4-bits blue)

3.3 volts

9-bit SPI serial interface (clock/data signals)

The major irritant in using this display is identifying the graphics controller; there are two possibilities (Epson S1D15G00 or Philips PCF8833). The LCD display sold by the German Web Shop Jelu has a Leadis LDS176 controller but it is 100% compatible with the Philips PCF8833). So how do you tell which controller you have? Some message boards have suggested that the LCD display be disassembled and the controller chip measured with a digital caliper – well that's getting a bit extreme.

Here's what I know. The Olimex boards have both display controllers possible; if the LCD has a GE-12 sticker on it, it's a Philips PCF8833. If it has a GE-8 sticker, it's an Epson controller. The older Sparkfun 6100 displays were Epson, their web site indicates that the newer ones are an Epson clone. Sparkfun software examples sometimes refer to the Philips controller so the whole issue has become a bit murky. The trading companies in Honk Kong have no idea what is inside the displays they are selling. A Nokia 6100 display that I purchased from Hong Kong a couple of weeks ago had the Philips controller.

I was not happy with any of the driver software examples I had inspected; they all seemed to be "mash-ups" – collections of code snippets for both types of controllers mixed together. None of these examples matched exactly the Philips PCF8833 or the Epson S1D15G00 user manuals, which can be downloaded from these links.

So I set out to write a driver based solely on the LCD controller manufacturer's manual. This is not to say that I didn't have my own mysteries. I had to "invert" the entire display and "reverse" the RGB order to get the colors to work out properly for the Philips controller. The Epson S1D15G00 user manual is a poor English translation and nearly incomprehensible. To keep this tutorial simple, I will not address the issues of scrolling or partial displays (to conserve power) since these are rarely-used features.

I used the Olimex SAM7-EX256 evaluation board as the execution platform. This is an ARM7 board with many peripherals that is an excellent way to learn about the ARM architecture at a reasonable price ($120 from Sparkfun). I also used the YAGARTO/Eclipse platform as the cross-development environment which is explained in great detail in my tutorial "**Using Open Source Tools for AT91SAM7 Cross Development**" which can be downloaded from the following link:

Hardware connection issues are also not the subject of this tutorial; you can download the Olimex schematic for the SAM7-EX256 board to see their design for a hardware interface to the Nokia 6100 LCD display.

LCD Display Orientation

The Nokia 6100 display has 132 x 132 pixels; each one with 12-bit color (4 bits RED, 4 bits GREEN and 4 bits BLUE). Practically speaking, you cannot see the first and last row and columns. The normal orientation is as follows:

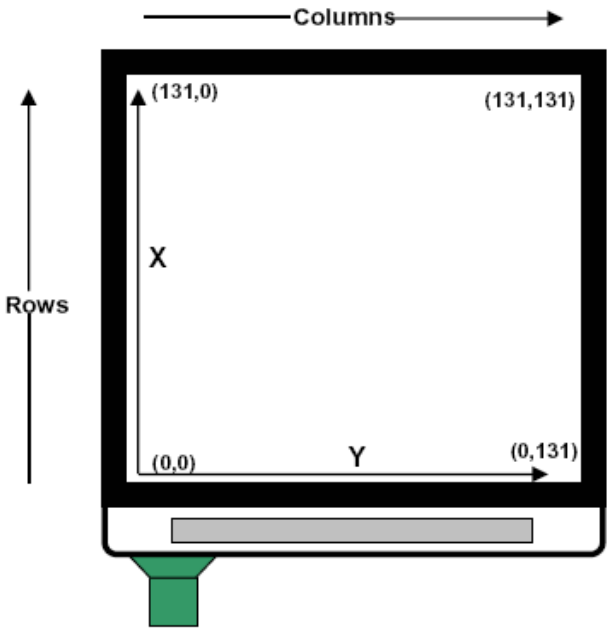


Figure 2. Default Orientation of Nokia 6100 LCD Display

That, of course, is upside-down on the Olimex SAM7-EX256 board if the silk-screen lettering is used as the up/down reference. So I set the "mirror x" and "mirror y" command to rotate the display 180 degrees, as shown below. This will be the orientation used in this tutorial (it is so easy to change back, if you desire).

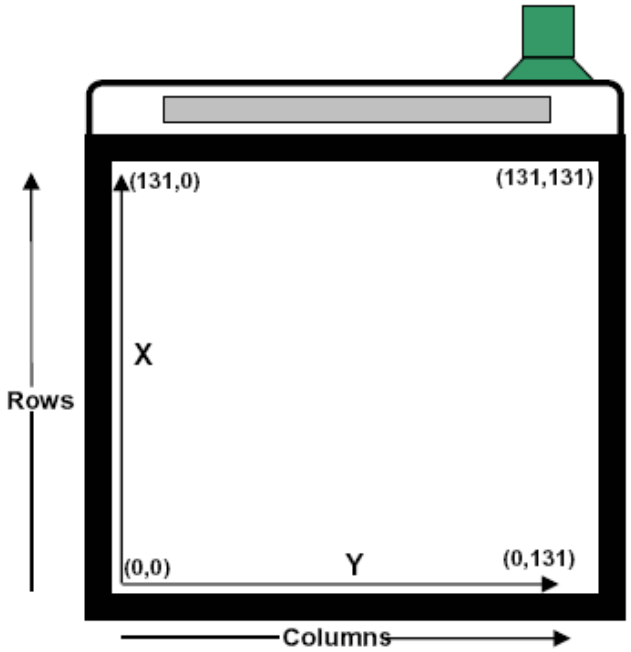


Figure 3. Tutorial Orientation of Nokia 6100 LCD Display

Communication with the Display

The Nokia 6100 uses a two-wire serial SPI interface (clock and data). The ARM7 microcontroller SPI peripheral generates the clock and data signals and the display acts solely as a slave device. Olimex elected to not implement the MISO0 signal that would allow the ARM microcontroller to read from the LCD display (you could read some identification codes, status, temperature data, etc). Therefore, the display is strictly **write-only**!

We send 9 bits to the display serially, the ninth bit indicates if a command byte or a data byte is being transmitted. Note in the timing diagram below from the Philips manual, the ninth bit (command or data) is clocked out first and is LOW to indicate a command byte or HIGH to indicate a data byte.

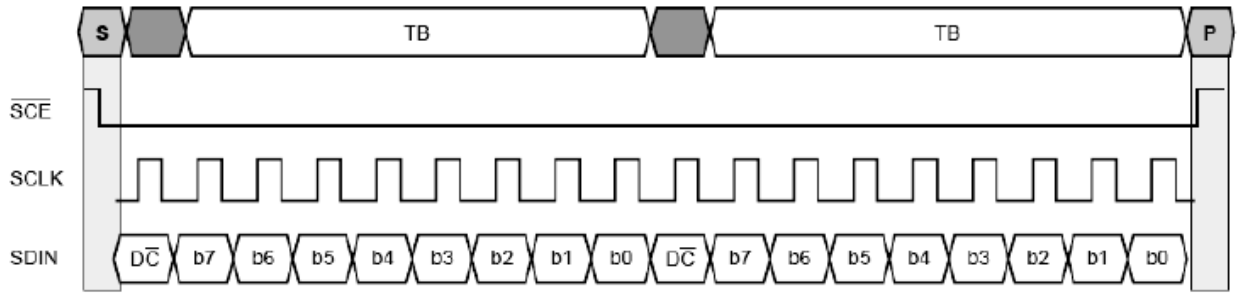


Figure 4. SPI serial interface sends commands and data bytes

How fast can this SPI interface be run? Since the PCF8833 data sheet specifies that the serial clock SCLK period be no less than 150 nsec, dividing the board's master clock (48054841 Hz) by 8 gives a period of 166 nsec. Thus we can safely run the SPI interface at 6 MHz. I have run the SPI interface at 16 MHz and it still worked, but that is tempting fate.

The SAM7-EX256 board uses an ARM7 microprocessor; so commands or data are submitted to the SPI peripheral as unsigned integers (32 bits) wherein only the lower 9 bits are used.

For example, to send a command we **clear** bit 8 to specify this is a command transmission. The lowest 8 bits contain the desired PCF8833 command.

```
unsigned int command;                  // PCF8833 command byte
while ((pSPI->SPI_SR &
AT91C_SPI_TXEMPTY) == 0);              // wait for the previous transfer to complete
command = (command & (~0x0100));       // clear bit 8 - indicates a "command" byte
pSPI->SPI_TDR = command;               // send the command
```

Likewise, to send a data byte we **set** bit 8 to specify that this is a data transmission. The lowest 8 bits contain the desired PCF8833 data byte.

```
unsigned int data;                     // PCF8833 data byte
while ((pSPI->SPI_SR &
AT91C_SPI_TXEMPTY) == 0);              // wait for the previous transfer to complete
data = (data | 0x0100);                // set bit 8 - indicates a "data" byte
pSPI->SPI_TDR = data;                  // send the command
```

Both snippets have a "wait until TXEMPTY" to guarantee that a new command/data stream is not started before the previous one has completed. This is quite safe as you will never get stuck forever in that wait loop.

The LCD driver has three functions supporting the SPI interface to the LCD:

```
InitSpi( )                    - sets up the SPI interface #1 to communicate with
the LCD
WriteSpiCommand(command)      - sends a command byte to the LCD
WriteSpiData(data)            - sends a data byte to the LCD
```

Using these commands is quite simple; for example, to initialize the SPI interface and then set the contrast for the Philips controller:

```
InitSpi( );                   // Initialize SPI interface to LCD
WriteSpiCommand(SETCON);      // Write contrast (command 0x25)
WriteSpiData(0x30);           // contrast 0x30 (range is -63 to +63)
```

The hardware interface uses five I/O port pins; four bits from PIOA and one bit from PIOB, as shown in Table 1 and Figure 5 below.

PA2	**LCD Reset (set to low to reset)**
PA12	**LCD chip select (set to low to select the LCD chip)**
PA16	**SPI0_MISO Master In - Slave Out (not used in LCD nterface)**
PA17	**SPI0_MOSI Master Out - Slave In pin (Serial Data to LCD slave)**
PA18	**SPI0_SPCK Serial Clock (to LCD slave)**
PB20	**backlight control (normally PWM control, 1 = full on)**

Table 1. I/O port bits used to support the SPI interface to the LCD Display

Note in Table 1 above that Olimex elected not to support the SPIO_MOSI – Master In bit (PA16) which would have allowed the user to read from the display. The LED backlight needs a lot of current, so a 7-volt boost converter is used for this purpose. The backlight can be turned on and off using PB20. It looks like you might be able to PWM the backlight, but I doubt anyone would want the backlight to be at half brightness.

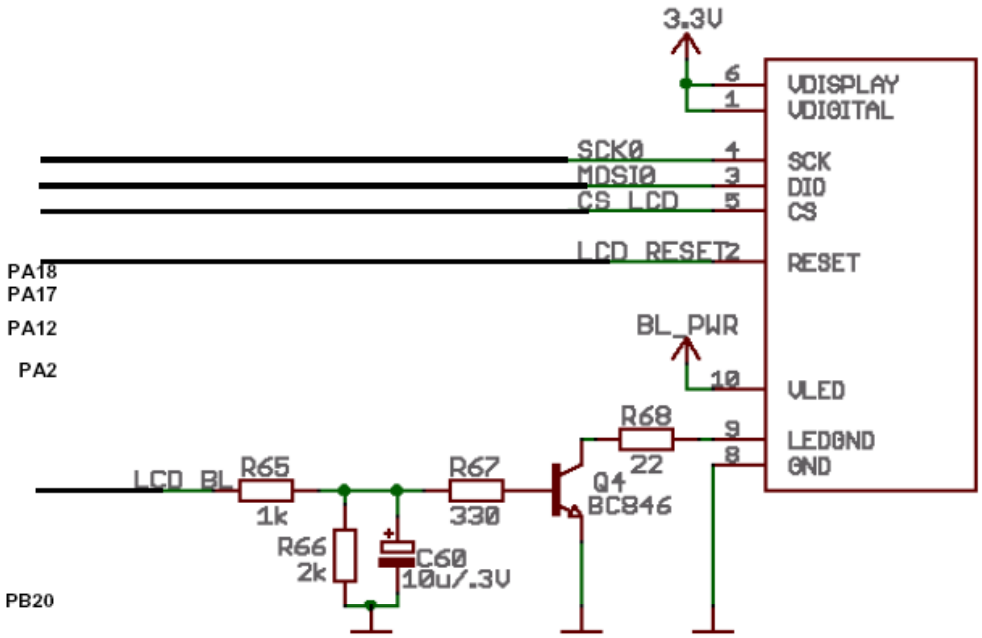


Figure 5. Hardware Interface to Nokia 6100 LCD Display (Olimex design)

Addressing Pixel Memory

The Philips PCF8833 controller has a 17424 word memory (132 x 132), where each word is 12 bits (4-bit color each for red, green and blue). You address it by specifying the address of the desired pixel with the Page Address Set command (rows) and the Column Address Set command (columns).

The Page Address Set and Column Address Set command specify two things, the starting pixel and the ending pixel. This has the effect of creating a drawing box. This sounds overly complex, but it has a wrap-around and auto-increment feature that greatly simplifies writing character fonts and filling rectangles.

The pixel memory has 132 rows and 132 columns, as shown in below

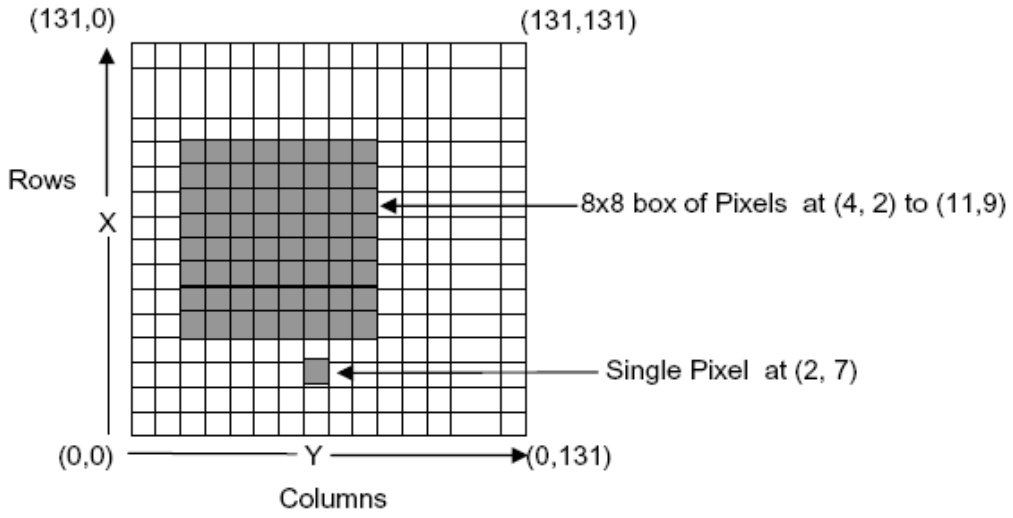


Figure 6. Philips PCF8833 Pixel Memory

To address a single pixel, just specify the same location for the starting pixel and the ending pixel on each axis. For example, to specify a single pixel at (2, 7) use the following sequence.

```
WriteSpiCommand(PASET);     // Row address set (command 0x2B)
WriteSpiData(2);            // starting x address
WriteSpiData(2);            // ending x address (same as start)
WriteSpiCommand(CASET);     // Column address set (command 0x2A)
WriteSpiData(7);            // starting y address
WriteSpiData(7);            // ending y address (same as start)
```

To address a rectangular area of pixels, just specify the starting location and the ending location on each axis, as shown below. For example, to define a drawing rectangle from (4, 3) to (11, 9) use the following sequence.

```
WriteSpiCommand(PASET);     // Row address set (command 0x2B)
WriteSpiData(4);            // starting x address
WriteSpiData(11);           // ending x address
WriteSpiCommand(CASET);     // Column address set (command 0x2A)
WriteSpiData(3);            // starting y address
WriteSpiData(9);            // ending y address
```

Once the drawing boundaries have been established (either a single pixel or a rectangular group of pixels), any subsequent memory operations are confined to that boundary. For instance, if you try to write more pixels than defined by the boundaries, the extra pixels are discarded by the controller. The Epson S1D15G00 controller has essentially the same memory layout as the Philips/NXP PCF8833.

12-Bit Color Data

The Philips PCF8833 LCD controller has three different ways to specify a pixel's color.

1. **12 bits per pixel** (native mode)

Selection of the native 12 bits/pixel mode is accomplished by sending the Color Interface Pixel Format command (0x3A) followed by a single data byte containing the value 3.

This encoding requires a Memory Write command and 1.5 subsequent data bytes to specify a single pixel. The bytes are packed so that two pixels will occupy three sequential bytes and the process repeats until the drawing boundaries are used up. Figure 7 illustrates the 12 bits/pixel encoding.

0	0	1	0	1	1	0	0	RAMWR command (memory write)
R	R	R	R	G	G	G	G	Data: Red and green from 1st pixel
B	B	B	B	R	R	R	R	Data: Blue from 1st pixel; red from 2nd pixel
G	G	G	G	B	B	B	B	Data: Green and blue from 2nd pixel

Figure 7. Color encoding for 12 bits / pixel - example illustrates sending 2 pixels

You might pose the question “What happens if I specify a single pixel with just two data bytes. Will the 4-bits of red information from the next pixel (usually set to zero) perturb the neighboring pixel? The answer is no since the PCF8833 controller writes to display RAM only when it gets a complete pixel. The straggler red bits from the next pixel wait for the completion of the remaining colors which will never come. Appearance of any command will cancel the previous memory operation and discard the unused pixel information. To be safe, I added a NOP command in the LCDSetPixel( ) function to guarantee that the unused red information from the next pixel is discarded.

Figure 8 demonstrates how to send a single pixel using 12-bit encoding. Note that the last 4 red bits from the next pixel will be ignored.

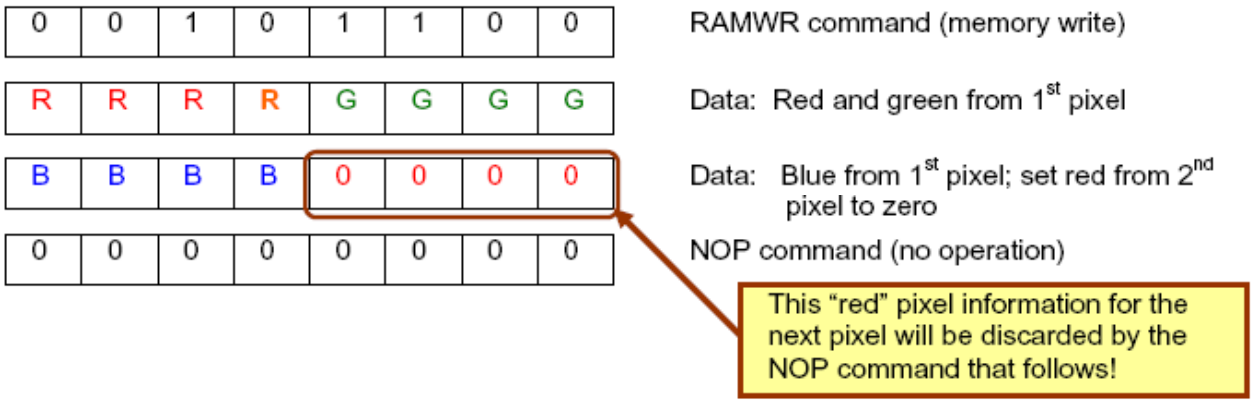


Figure 8. Color encoding for 12 bits / pixel - example illustrates sending 1pixel

2. 8 bits per pixel

Selection of the reduced resolution 8 bits/pixel mode is accomplished by sending the Color Interface Pixel Format command (0x3A) followed by a single data byte containing the value 2.

This encoding requires a Memory Write command and a single subsequent data byte to specify a single pixel. The data byte contains all the color information for one pixel. The color information is encoded as 3 bits for red, 3 bits for green and 2 bits for blue, as shown in Figure 9 below

Figure 9. Color encoding for 8 bits – per – pixel

The important thing to note here is that this 8-bit color encoding will be converted to the 12-bit encoding by the Color Table that you set up in advance. This Color Set table will convert 3-bit RED to 4-bit RED, 3-bit GREEN to 4-bit GREEN and 2-bit BLUE to 4-bit BLUE. This is made possible by the specification of the 20 entry color table in the initialization step.

```
WriteSpiCommand(RGBSET);        // Define Color Table (command 0x2D)
WriteSpiData(0);                // red 000 value
WriteSpiData(2);                // red 001 value
WriteSpiData(5);                // red 010 value
WriteSpiData(7);                // red 011 value
WriteSpiData(9);                // red 100 value
WriteSpiData(11);               // red 101 value
WriteSpiData(14);               // red 110 value
WriteSpiData(16);               // red 111 value
WriteSpiData(0);                // green 000 value
WriteSpiData(2);                // green 001 value
WriteSpiData(5);                // green 010 value
WriteSpiData(7);                // green 011 value
WriteSpiData(9);                // green 100 value
WriteSpiData(11);               // green 101 value
WriteSpiData(14);               // green 110 value
WriteSpiData(16);               // green 111 value
WriteSpiData(0);                // blue 000 value
WriteSpiData(6);                // blue 001 value
WriteSpiData(11);               // blue 010 value
WriteSpiData(15);               // blue 011 value
```

Consider the following points. The resolution of the Nokia 6100 display is 132 x 132 pixels, 12 bits/pixel. Since the 8 bits/pixel encoding is converted by the color table to 12 bits/pixel, there is no saving of display memory. The 8 bits/pixel encoding would use about 1/3 less data bytes to fill an area, so there would be a performance gain in terms of the number of bytes transferred. The 8 bits/pixel encoding would make a photograph look terrible. In the author's view, there's very little to be gained by using this mode in an ARM microcontroller environment. Therefore, I elected to not implement the color table and 8-bit encoding in this driver.

3. 16 bits per pixel

Selection of 16 bits/pixel mode is accomplished by sending the Color Interface Pixel Format command (0x3A) followed by a single data byte containing the value 5.

This encoding requires a Memory Write command and two subsequent data bytes to specify a single pixel. The color information is encoded as 5 bits for RED, 6 bits for GREEN and 5 bits for BLUE, as shown in Figure 10 below

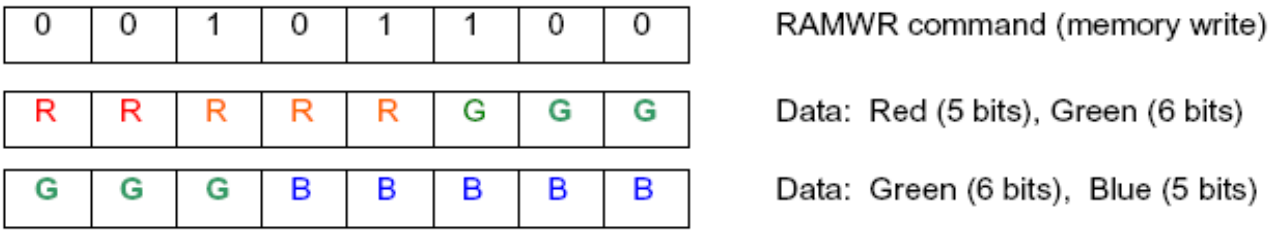


Figure10. Color encoding for 16 bits – per - pixel

This pixel encoding is converted by the controller using a dithering technique to the 12-bit data for the pixel RAM. The net effect is to give 65k color variations. My view is that nobody is going to display the Mona Lisa on this tiny display, so 16-bit color encoding would be rarely used. I did not include support for it in the driver software, but you could easily add it if you desire. The Epson S1D15G00 controller supports the 8-bit and 12-bit modes, but not the 16-bit mode.

Wrap-Around and Auto Increment

The wrap-around feature is the cornerstone of the controller's design and it amazes me how many people ignored it in drawing rectangles and character fonts. This feature allows you to efficiently draw a character or fill a box with just a simple loop – taking advantage of the wrap-around after writing the pixel in the last column and auto-incrementing to the next row. Remember how the pixel was addressed by defining a "drawing box"? If you are planning to draw an 8 x 8 character font, define the drawing box as 8 x 8 and do a simple loop on 64 successive pixels. The row and column addresses will automatically increment and wrap back when you come to the end of a row, as shown in Figure 11 below.

The rules for Auto-incrementing and Wrap-Around are as follows.

- Set the column and row address to the bottom left of the drawing box.
- Set up a loop to do all the pixels in the box. Specifically, since three data bytes will specify the color for two pixels, the loop will typically iterate over ½ the total number of pixels in the box.
- Writing three memory bytes will illuminate two pixels (12-bit resolution). Each pixel written automatically advances the column address. When the "max" column address pixel is done, the column address wraps back to the column starting address **AND** the row address increments by one. Now keep writing memory bytes until the next row is illuminated and so on.

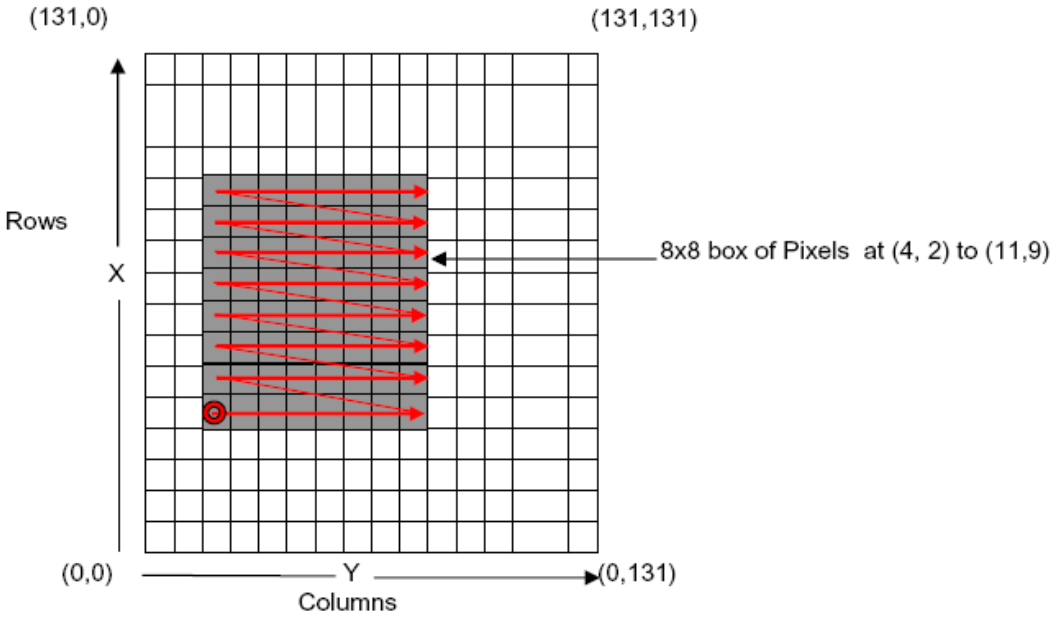


Figure 11. Drawing Box permits auto-increment and wrap-around.

To illustrate this technique, Figure 12 shows the code to fill an 8 x 8 box shown above. Note that we set the row and column address just once (pointing to the lower left corner). Then we do a single Memory Write command followed by three data bytes done 33 times. The grand total is 106 SPI transmissions.

Compare that to the implementation where you address each pixel, set Memory Write and feed two bytes of color data for each pixel. The grand total would be 576 SPI transmissions. The advantage gained using the auto-increment and wrap-around features is obvious.

```
// Row address set (command 0x2B)
WriteSpiCommand(PASET);
WriteSpiData(4);
WriteSpiData(11);
// Column address set (command 0x2A)
WriteSpiCommand(CASET);
WriteSpiData(2);
WriteSpiData(9);
// Write Memory (command 0x2C)
WriteSpiCommand(RAMWR);
// loop on total number of pixels / 2
for (i = 0; i < ((((11 - 4 + 1) * (9 - 2 + 1)) / 2) + 1); i++) {
        // use the color value to output three data bytes covering two pixels
        WriteSpiData((color >> 4) & 0xFF);
        WriteSpiData(((color & 0xF) << 4) | ((color >> 8) & 0xF));
        WriteSpiData(color & 0xFF);

}
```

Figure 12. Code Snippet to Fill an 8 x 8 box

Code to use this technique to draw a character font is similar, but at each pixel you have to determine if the font calls for a foreground color or the background color.

Initializing the LCD Display (Philips PCF8833)

This was a surprise to me but the Philips PCF8833 does not quite boot into a “ready to display” mode after hardware reset. The following is the minimal commands/data needed to place it into 12-bit color mode.

First, we do a hardware reset with a simple manipulation of the port pin. Reset is asserted low on this controller.

```
// Hardware reset
LCD_RESET_LOW;
Delay(20000);
LCD_RESET_HIGH;
Delay(20000);
```

The controller boots into SLEEPIN mode, which keeps the booster circuits off. We need to exit sleep mode which will also turn on all the voltage booster circuits.

```
// Sleep out (command 0x11)
WriteSpiCommand(SLEEPOUT);
```

This is still a mystery to me, but I had to invert the display and reverse the RGB setting to get the colors to work correctly in this particular display. If you have trouble, consider removing this command.

```
// Inversion on (command 0x20)
WriteSpiCommand(INVON);        // seems to be required for this
                               Controller
```

For this driver, I elected to use the 12-bit color pixel format exclusively.

```
// Color Interface Pixel Format (command 0x3A)
WriteSpiCommand(COLMOD);
WriteSpiData(0x03);            // 0x03 = 12 bits-per-pixel
```

In setting up the memory access controller, I elected to use the “mirror x” and “mirror y” commands to reorient the x and y axes to agree with the silk screen lettering on the Olimex board. If you want the default orientation, send the data byte 0x08 instead. Finally, I had to reverse the RGB color setting to get the color information to work properly. You may want to experiment with this setting.

```
// Memory access controller (command 0x36).
WriteSpiCommand(MADCTL);
WriteSpiData(0xC8);    // 0xC0 = mirror x and y, reverse rgb
```

I found that setting the contrast varies from display to display. You may want to try several different contrast data values and observe the results on your display.

```
// Write contrast (command 0x25)
WriteSpiCommand(SETCON);
WriteSpiData(0x30); // contrast 0x30
Delay(2000);
```

Now that the display is initialized properly, we can turn on the display and we’re ready to start producing characters and graphics.

```
// Display On (command 0x29)
WriteSpiCommand(DISPON);
```

Note

While we are using TFT LCD close the jumper J22 to J25 as downward direction.

CHAPTER - 3

DEVELOPMENT TOOLS

ARM IAR Embedded Workbench

The ARM IAR Embedded Workbench IDE is a very powerful Integrated Development Environment that allows you to develop and manage complete embedded application projects.

Creating Application Project

ARM IAR Embedded Workbench integrated development environment (IDE) demonstrates a typical development cycle shows how you use the compiler and the linker to create a small application for the Arm core. For instance, creating a workspace, setting up a project with C source files, and compiling and linking and application.

Setting up Create a New Project

Using the IAR Embedded Workbench IDE, you can design advanced project models. You create a workspace to which you add one or several projects. There are ready-made project templates for both application and library projects. Each project can contain a hierarchy of groups in which you collect your source files. For each project you can define one or several build configurations.

Because the application in this tutorial is a simple application with very few files, the Tutorial does not need an advanced project model.

We recommend that you create a specific directory where you can store your entire project files. In this tutorial we call the directory projects. You can find all the files needed for the tutorials in the arm\tutor directory. Make a copy of the tutor directory in your Projects directory. Before you can create your project you must first create a workspace.

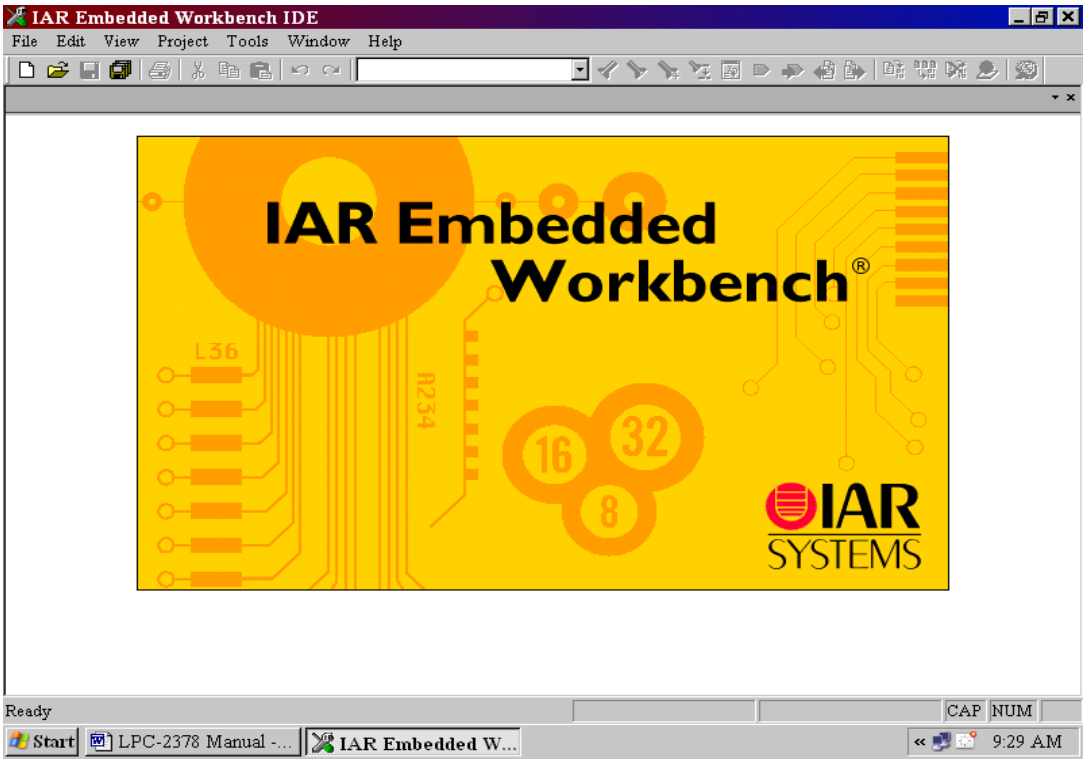
CREATING A WORKSPACE WINDOW

The first step is to create a new workspace for the tutorial application. When you start The IAR Embedded Workbench IDE for the first time, there is already a ready-made Workspace, which you can use for the tutorial projects. If you are using that workspace, you can ignore the first step.

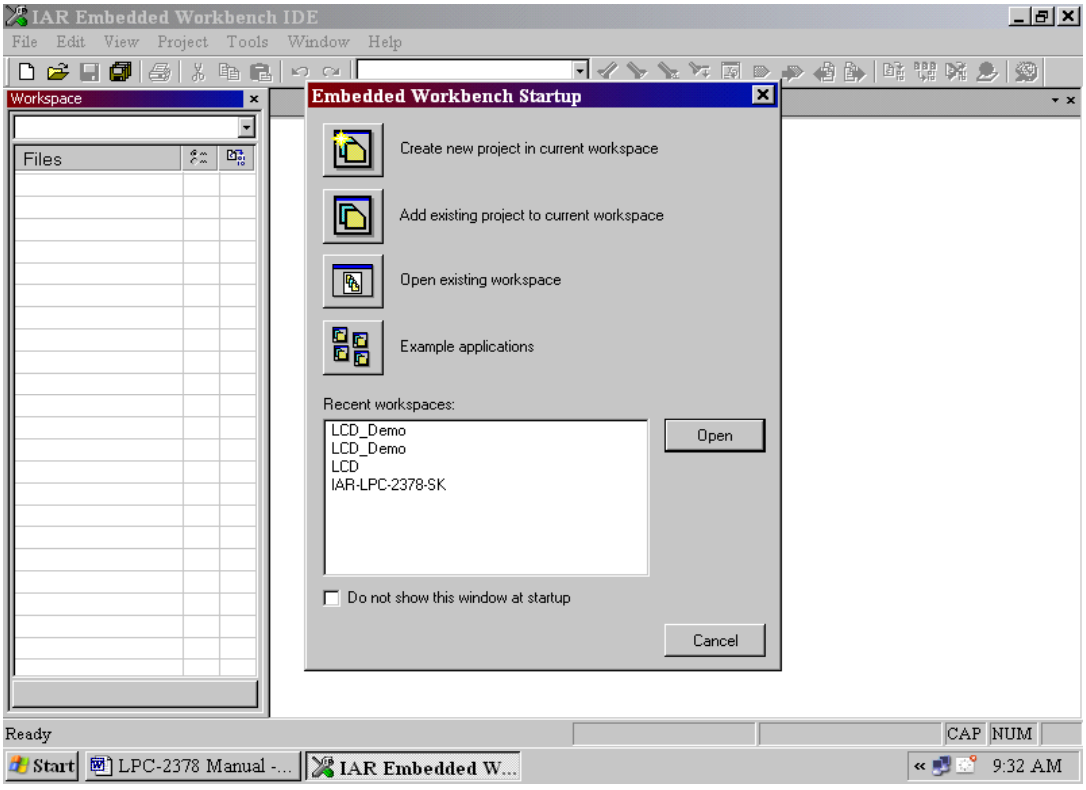
Choose **File>New>Workspace**. Now you are ready to create a project and add it to the Workspace.

CREATING THE NEW PROJECT

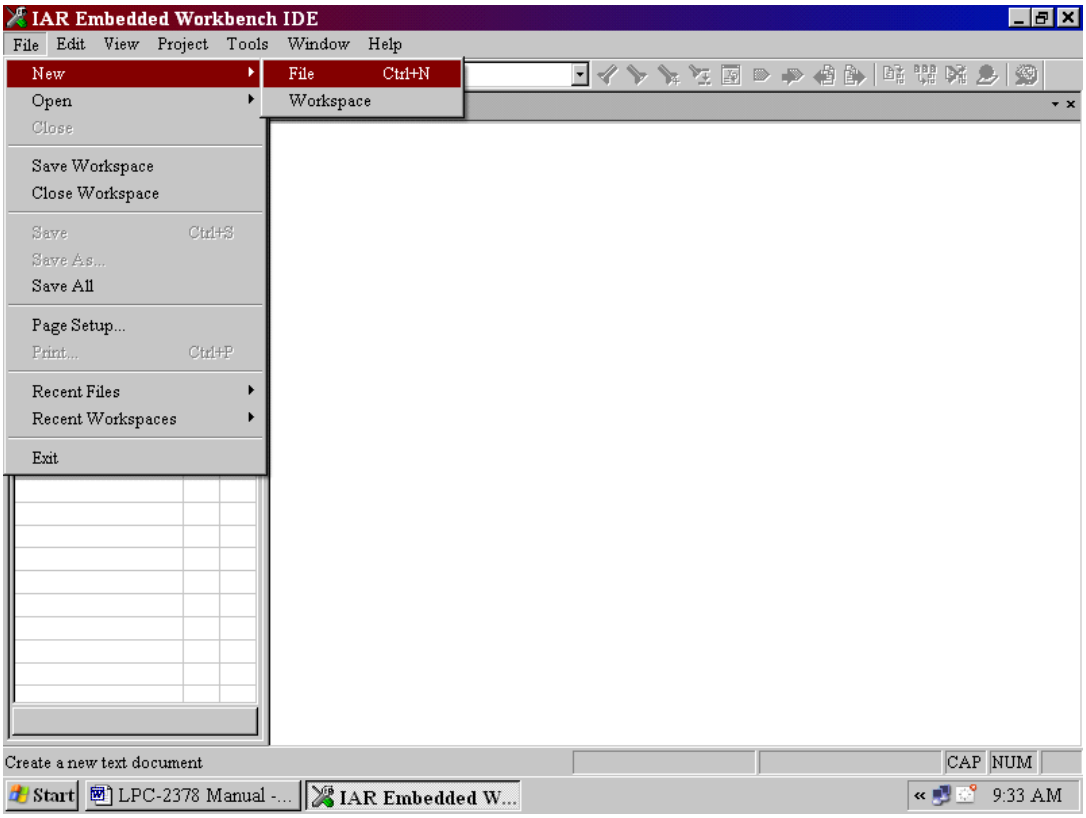
1. Run the Application **IAR Embedded Workbench**.



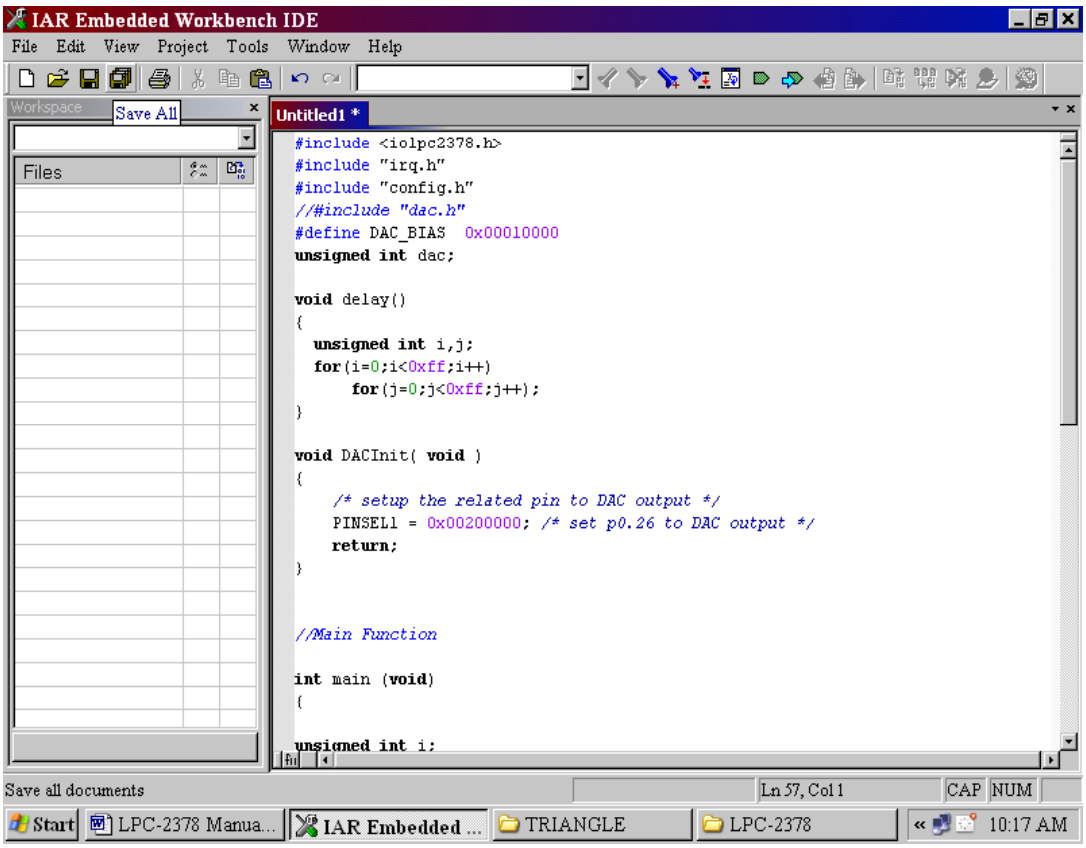
2. Click **Cancel** Button.



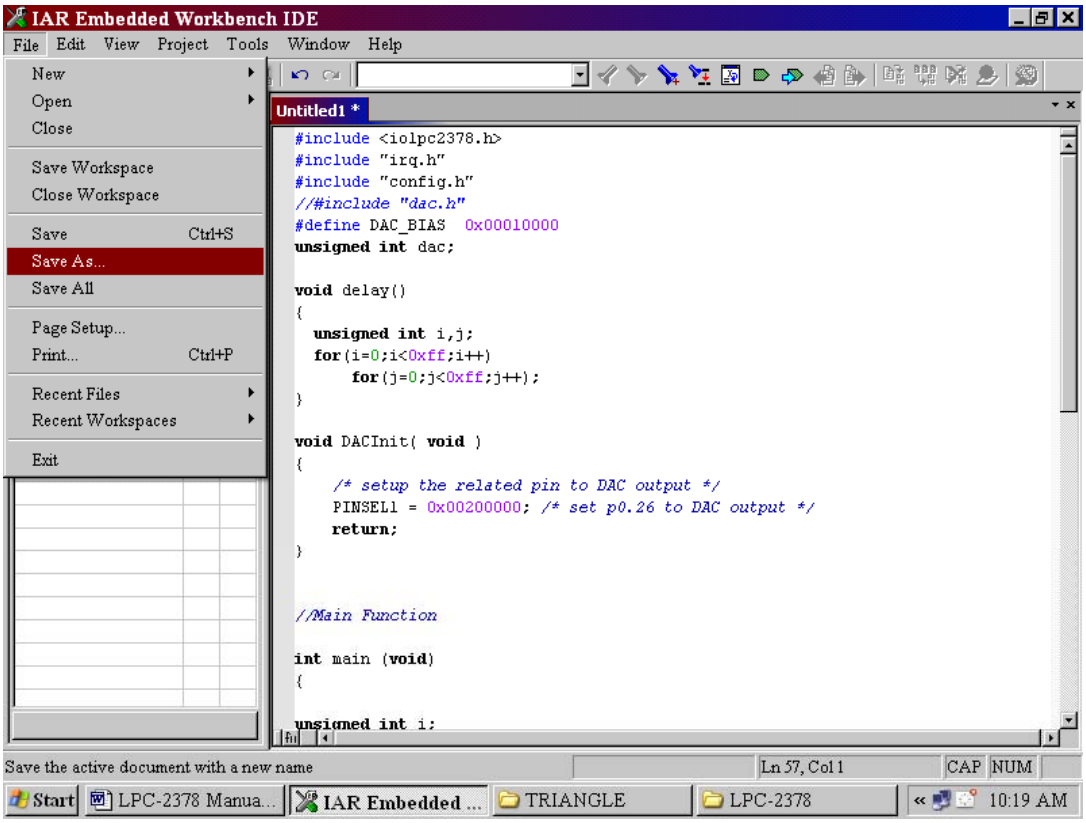
3. **New>File** Menu, for creating a new '**C** 'file.



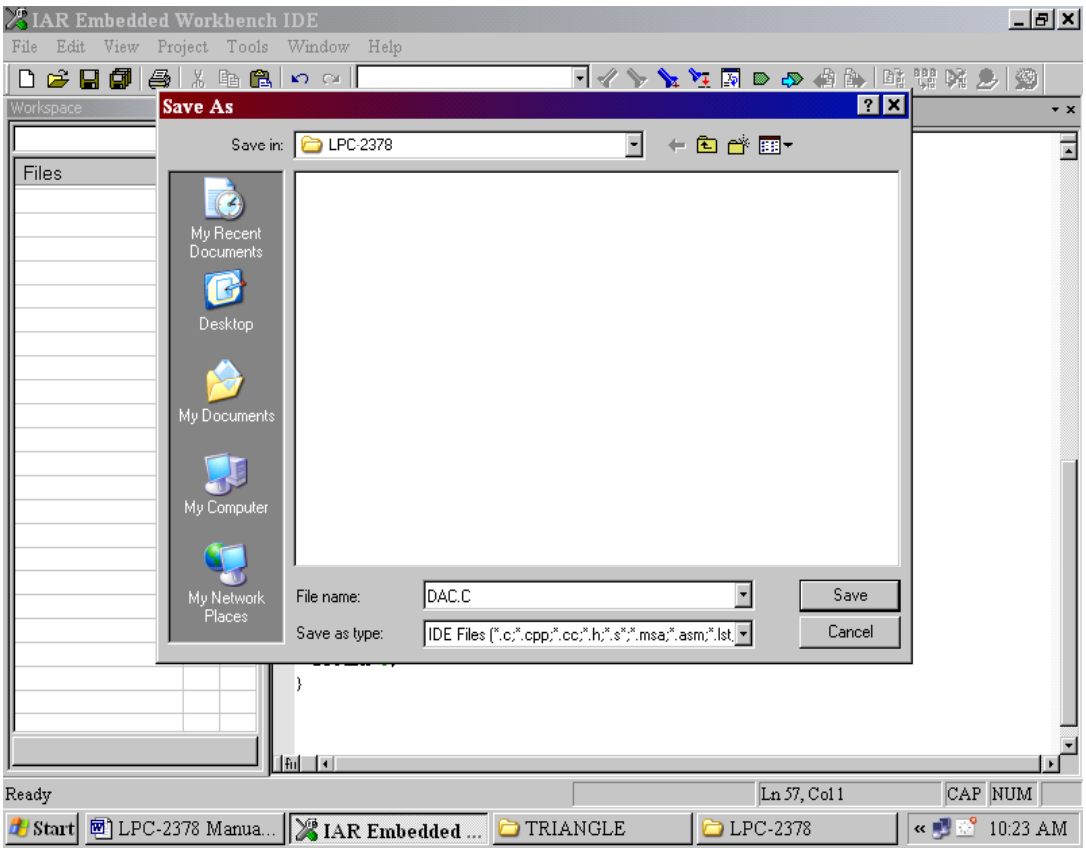
4. Creating your **'C'** file.



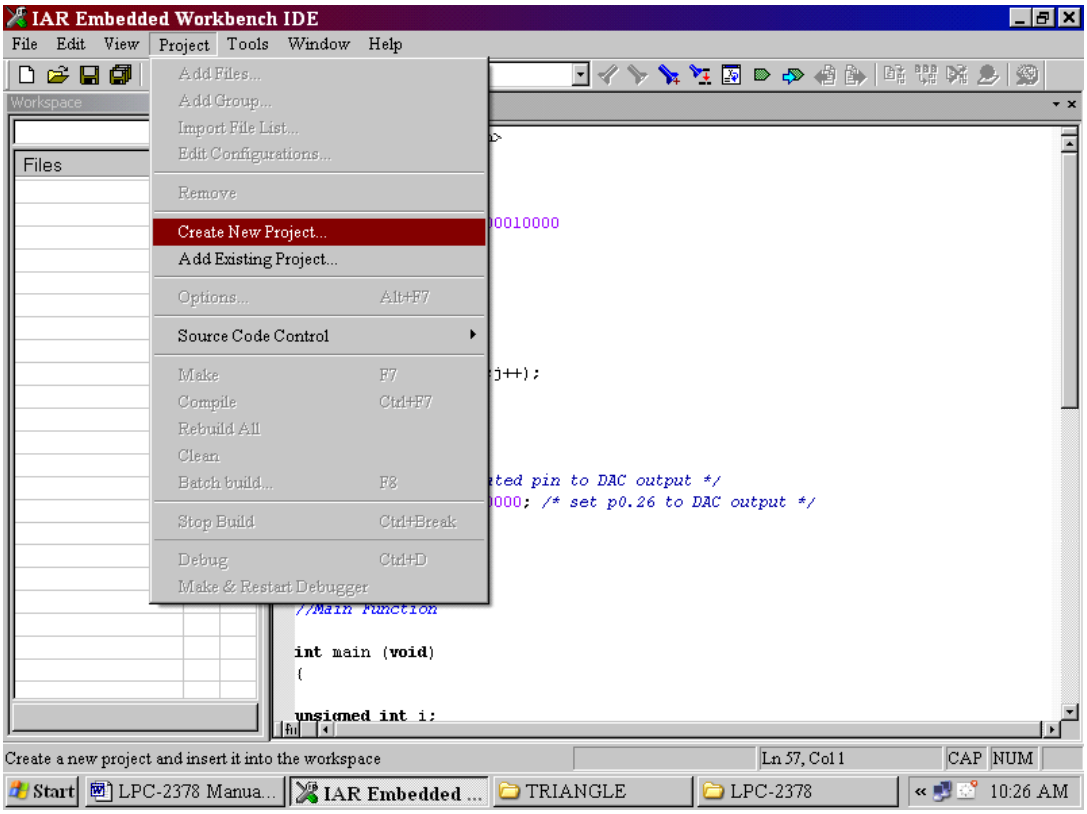
5. **File>Save As,** for saving a file.



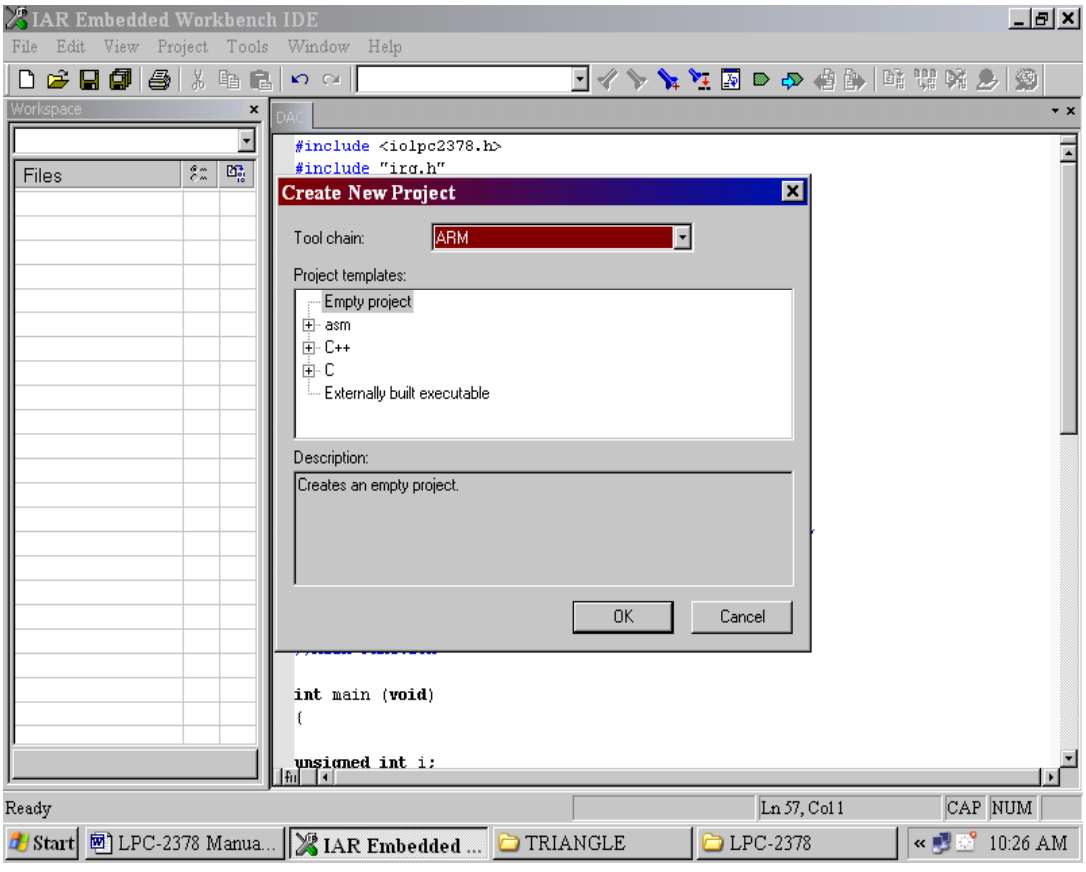
6. Give your File name Eg: **DAC.C** and click **Save** Button.



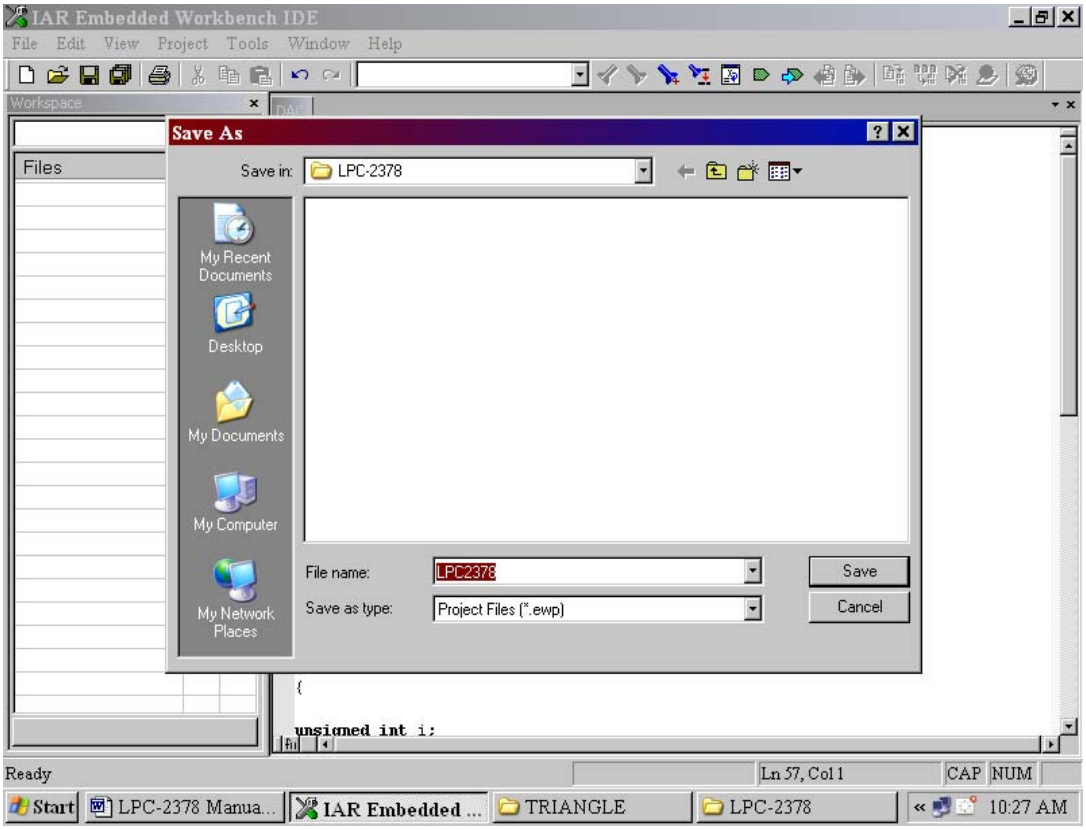
7. Select Button **Project >Create New Project** Menu, for creating a project.



8. Click **OK** Button following window will open.

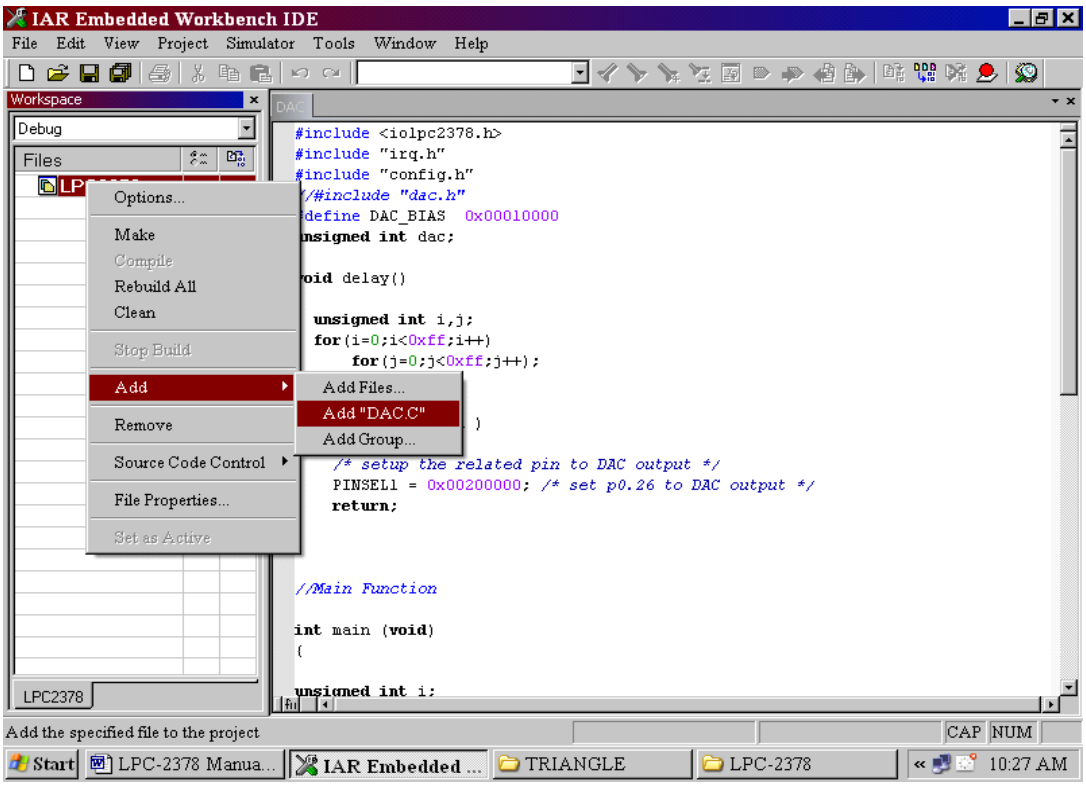


9. Give Project Name Eg: **LPC2378** and click **SAVE** Button.

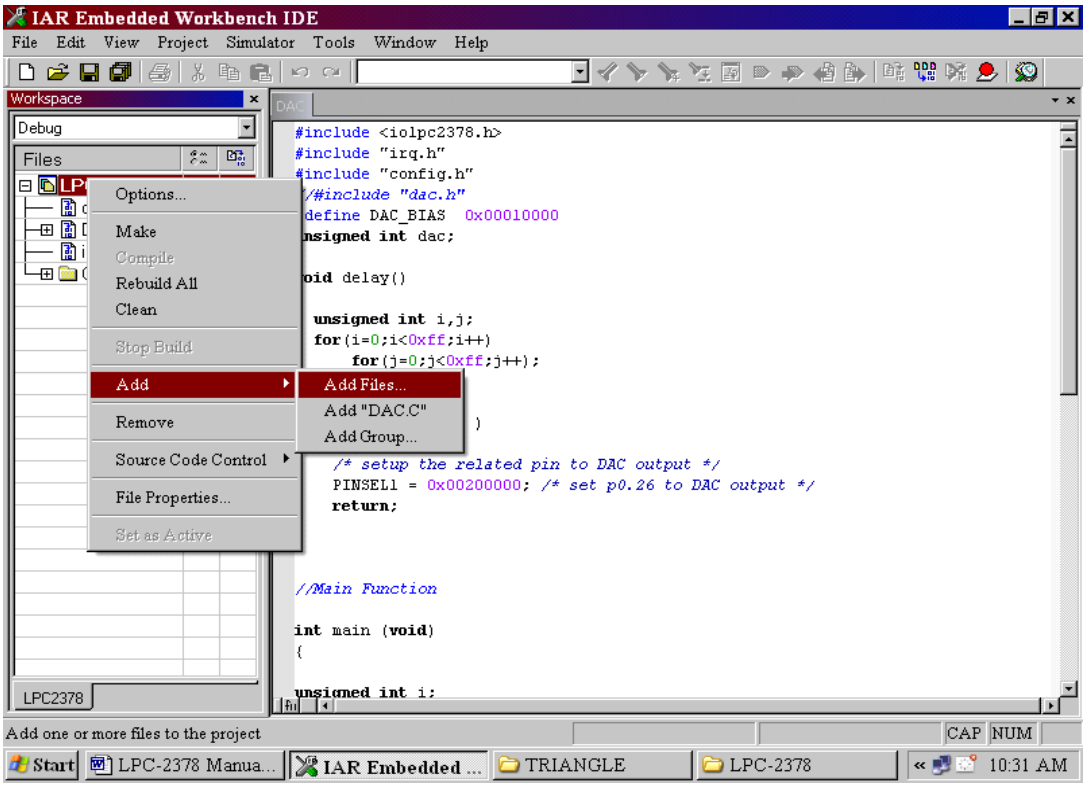


ADDING THE FILES TO THE PROJECT

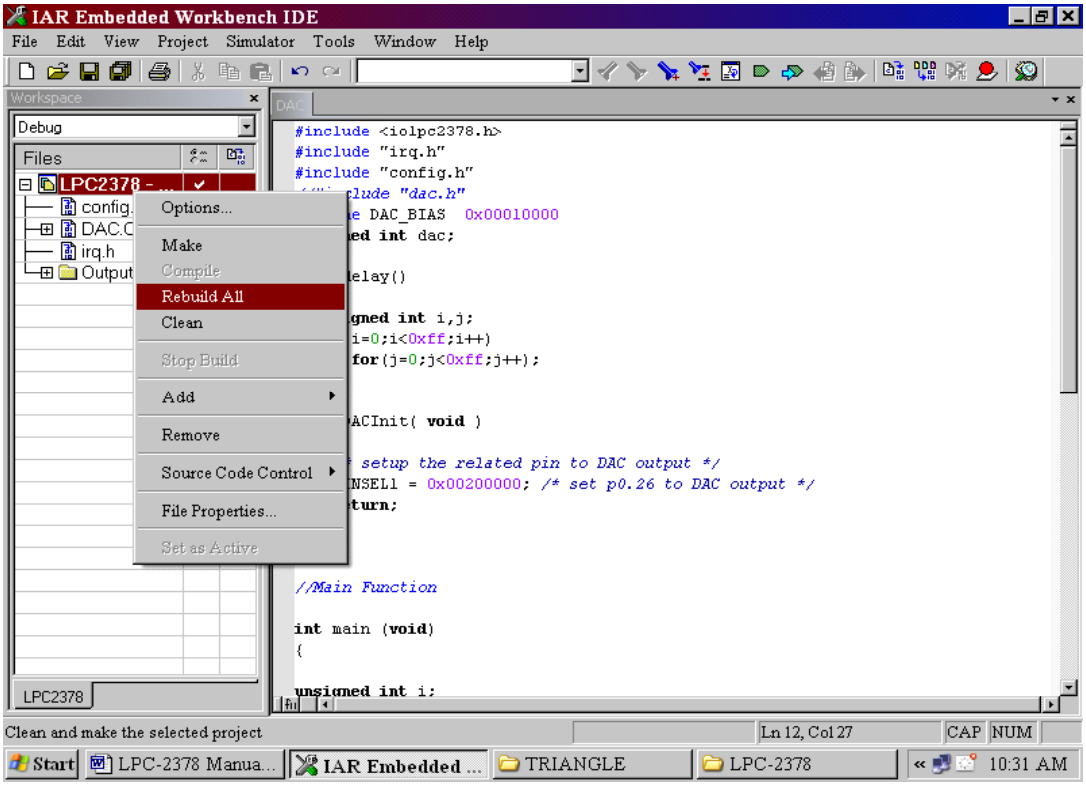
10. Right Click the project name option for select a 'C' file in workspace window. Select **Add >Add DAC.C** Menu.



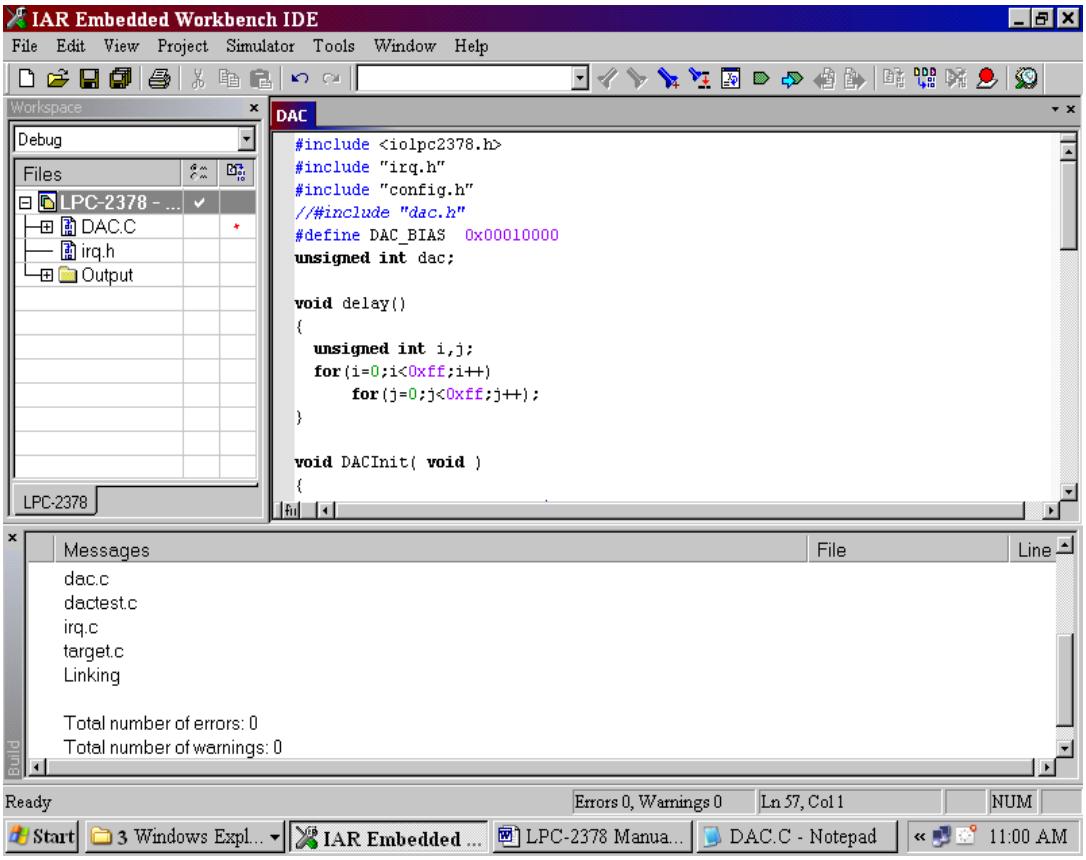
11. Right Click the Project name option in workspace window for adding a header files. Choose **Add** >**Add Files** menu. Before that the header files are store the path at where your '**C**' files is located.



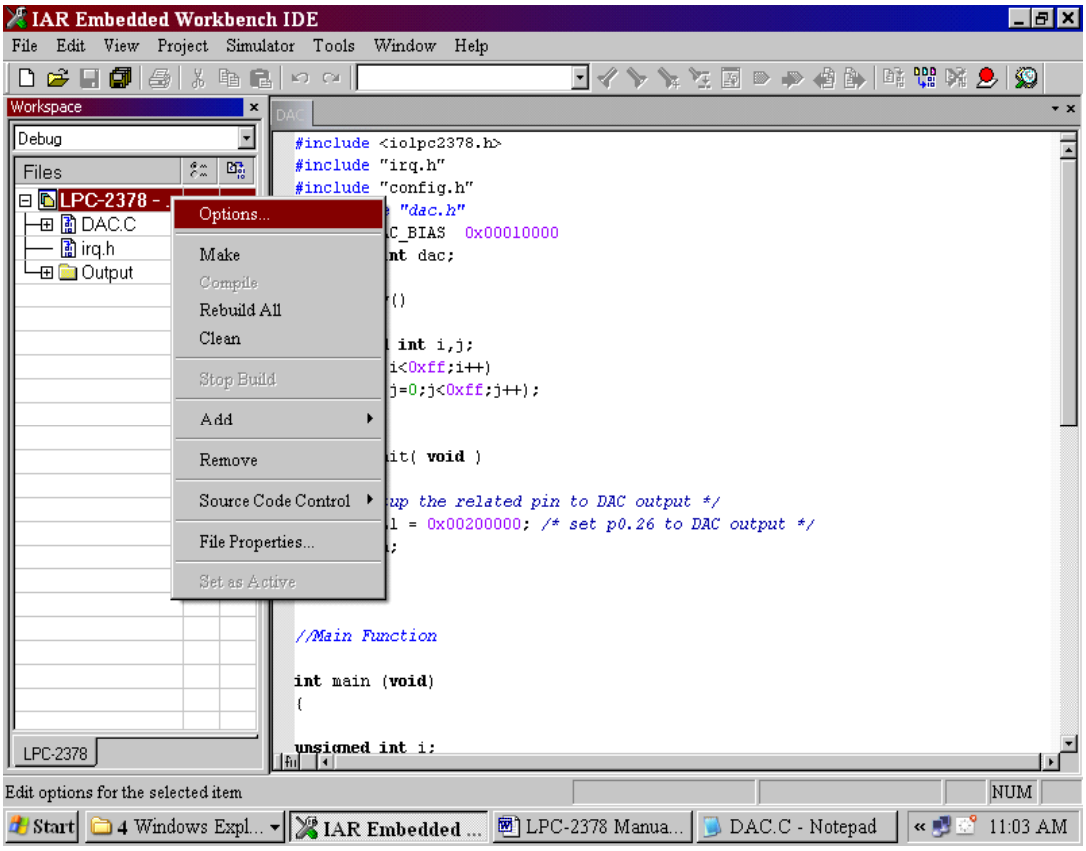
12. Select **LPC2378>Rebuild** all menu, for checking any error in your **'C'** file.



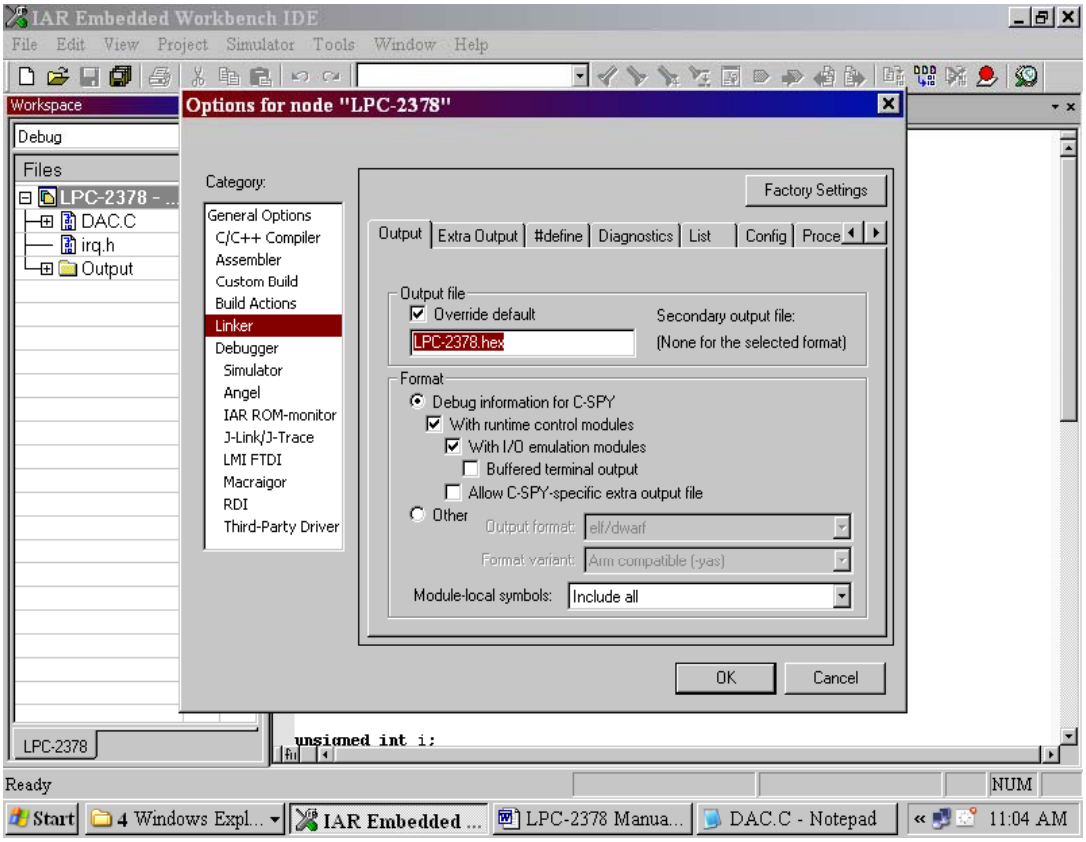
13. If your '**C**' file has no error the following window will be displayed.



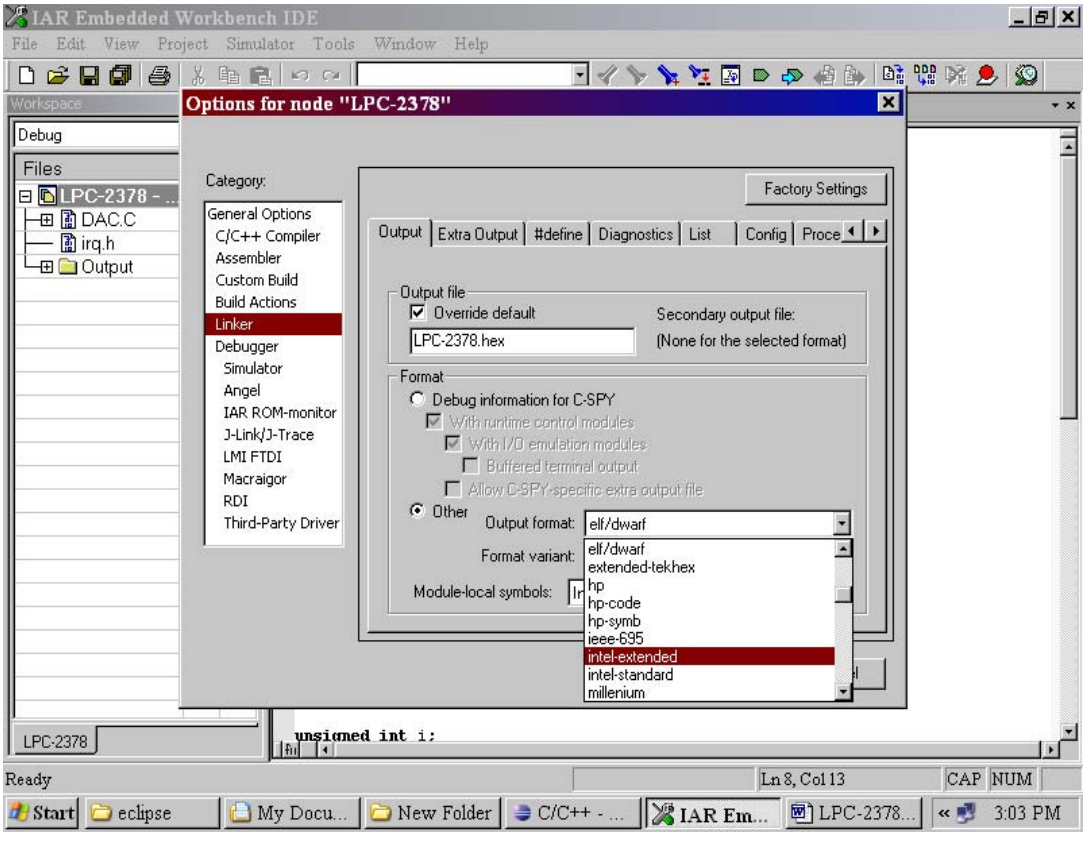
14. Select **LPC2378 >Options** menu



15. Select Linker Category. Enable output option file and give File name.hex Eg: **LPC2378.hex**.

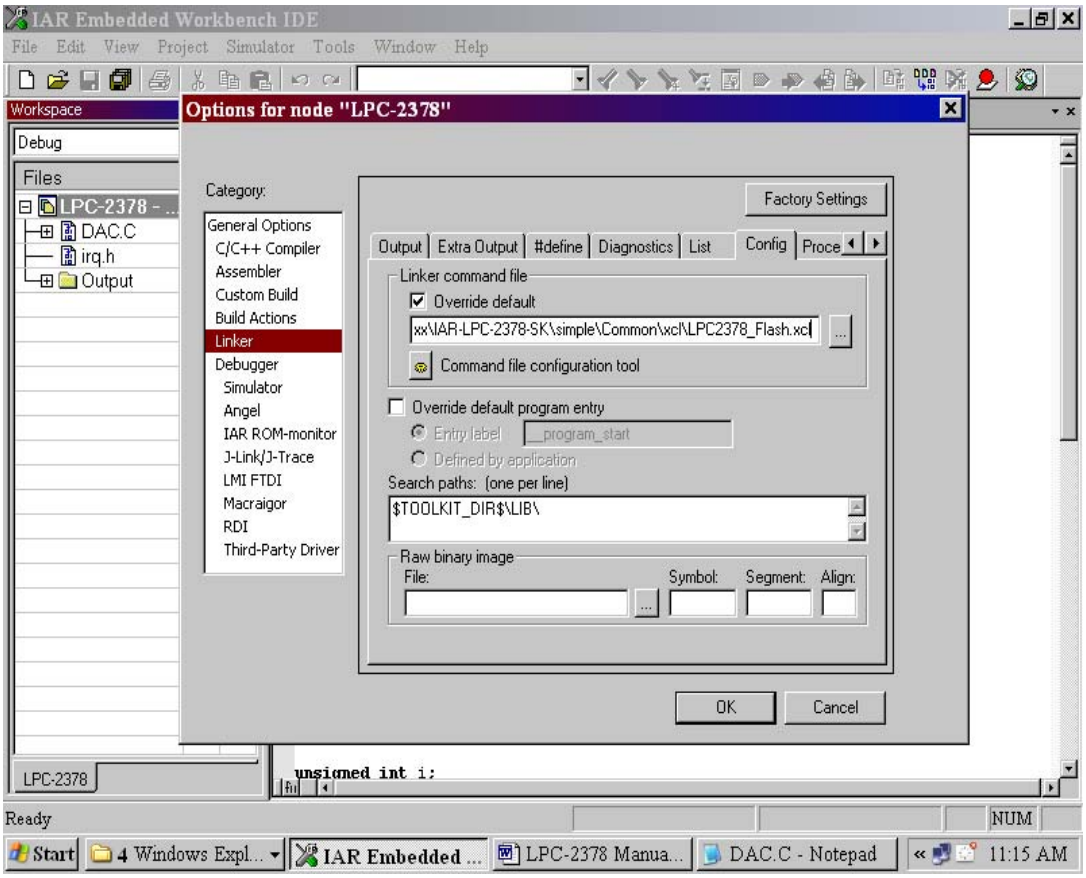


16. Enable **Other** Menu and select **Intelextented** option.



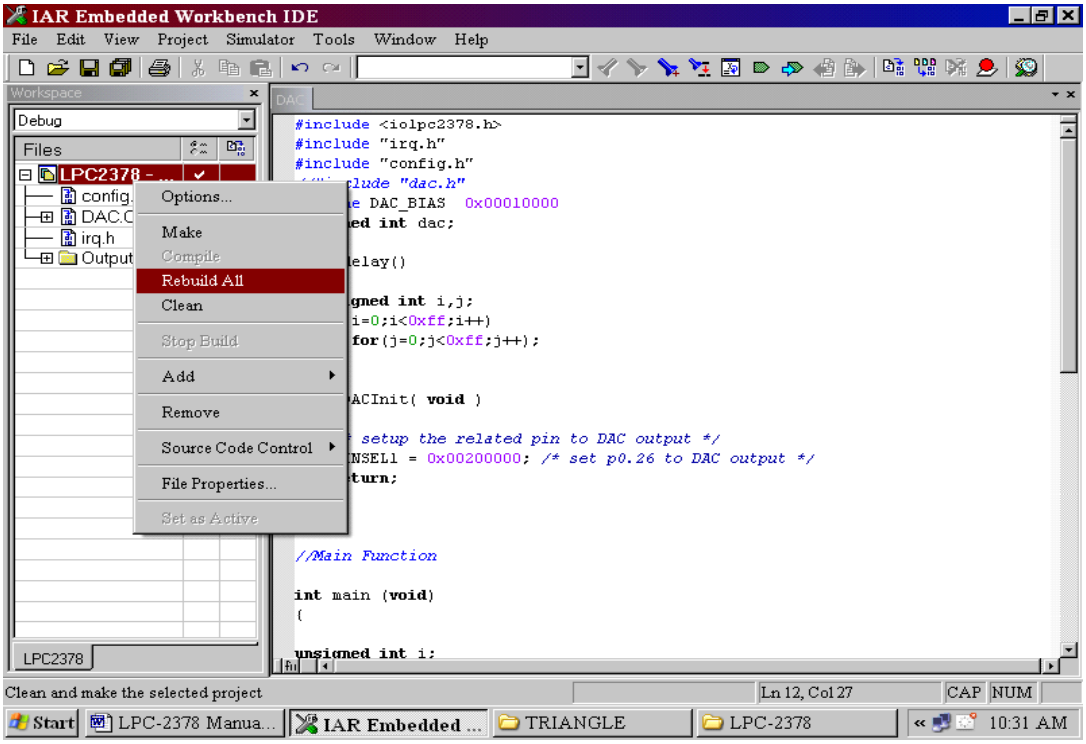
17. Select **Config** Menu. Enable **Overide default** option and click Browse button for choosing a **LPC2378-Flash-XCL** file.

Path: C:\Program Files\IAR Systems\Embedded Workbench 4.0 Kick Start \Arm\Examples\NxP\LPC23XX\IAR-LPC-2378\SK\simple\common\XCL\ LPC2378 FlashXCL and click OK button.

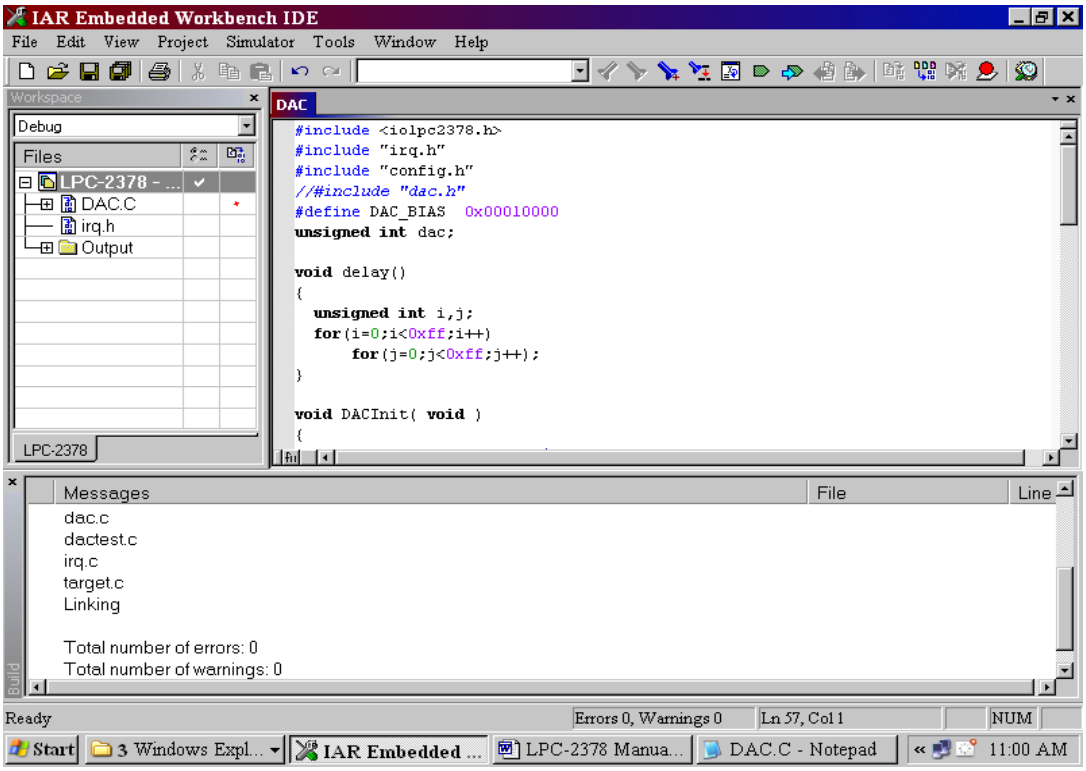


BUILTING THE PROJECT

18. Select **LPC2378 >Rebuild All** Menu, for Builting a project.



19. If your project has no error Builting completed and **Hex** file generated.



This **Hex** file will be downloaded to the VIARM-2378 controller by **Flash Magic** Software

FLASH ISP UTILITY

In- System Programming (ISP) : In System Programming is programming or Reprogramming the on-chip flash memory, using the boot-loader software and a serial port.

The LPC2378 Micro controller provides on-chip boot-loader software that allows programming of the internal flash memory over the serial channel. Pulling port pin P2.10 low during reset of the Microcontroller activates the boot-loader. ViARM-2378 Board contains circuits for Controlling Pin P2.10 over Toggle switch.

ViARM-2378 Development Board has an Reset switch and Mode Selection Switch (Programming /Execution Switch). Through this one, can change either Programming mode or Execution Mode.

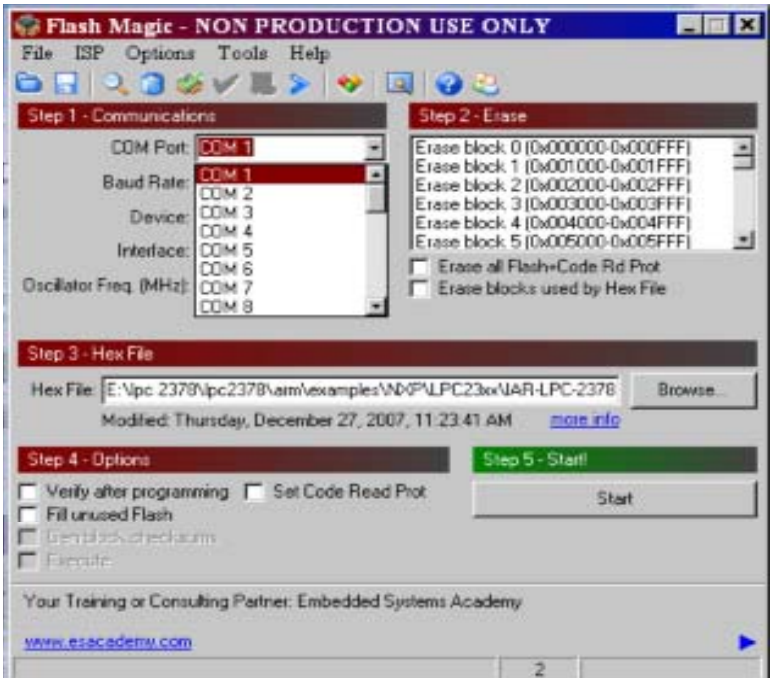
Philips provides a utility program for In-System Flash (ISP) programming called **Flash Magic** Software.

For now, it is assumed that the program to be downloaded is already developed and there exist a HEX-file to be downloaded. This HEX-file represents the binary image of the application program.

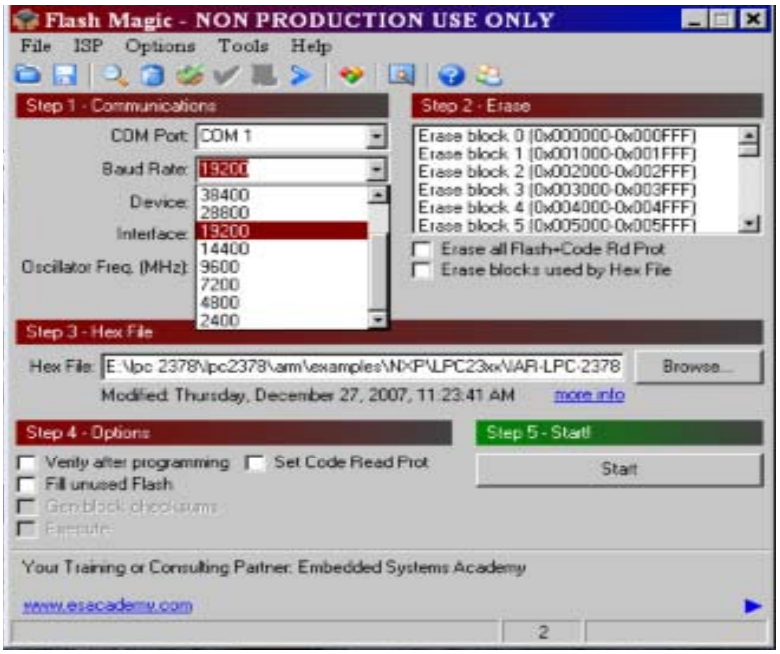
FLASH MAGIC

1. Run **FLASH MAGIC** Application, the following window will appear.

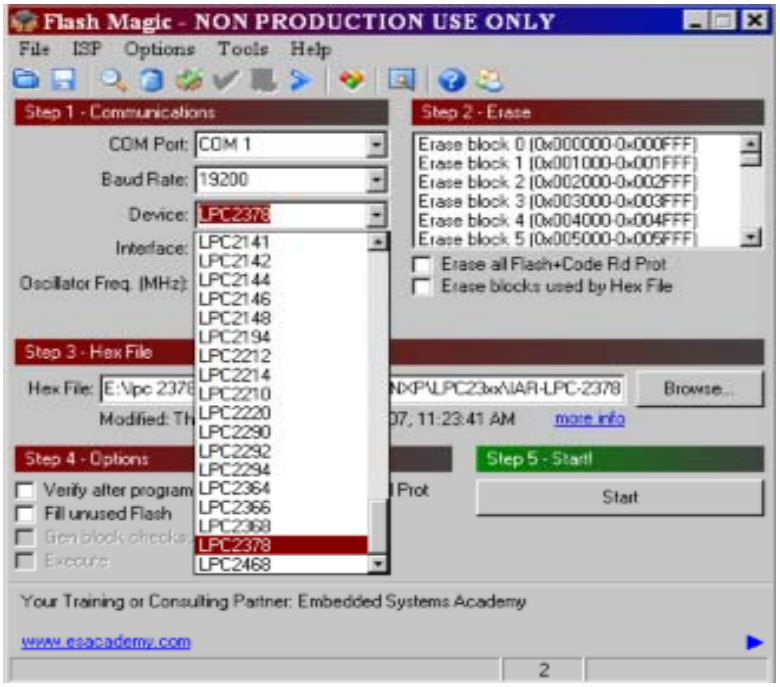
2. In **Step-1** Section, Select communication port Eg:**COM1**



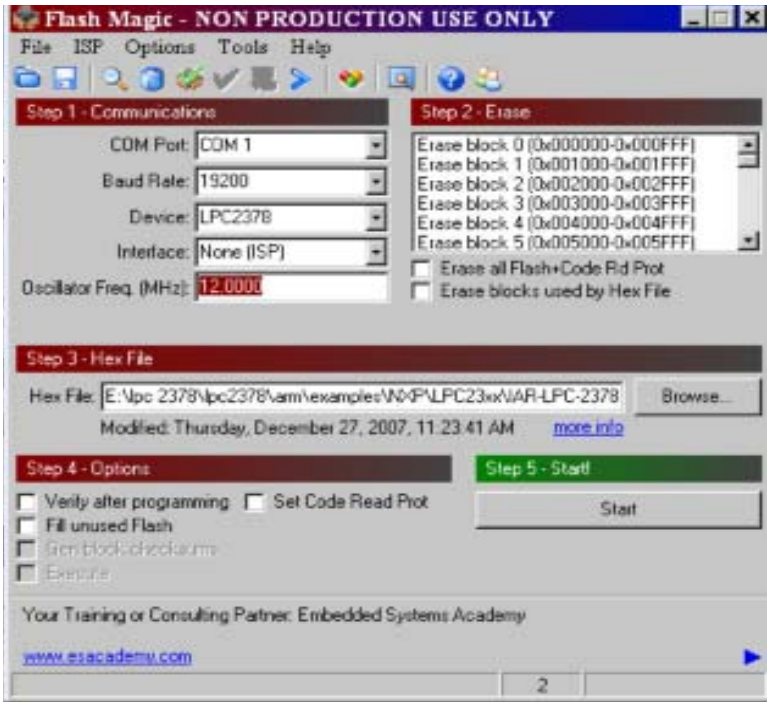
3. Select Baud Rate **19200**



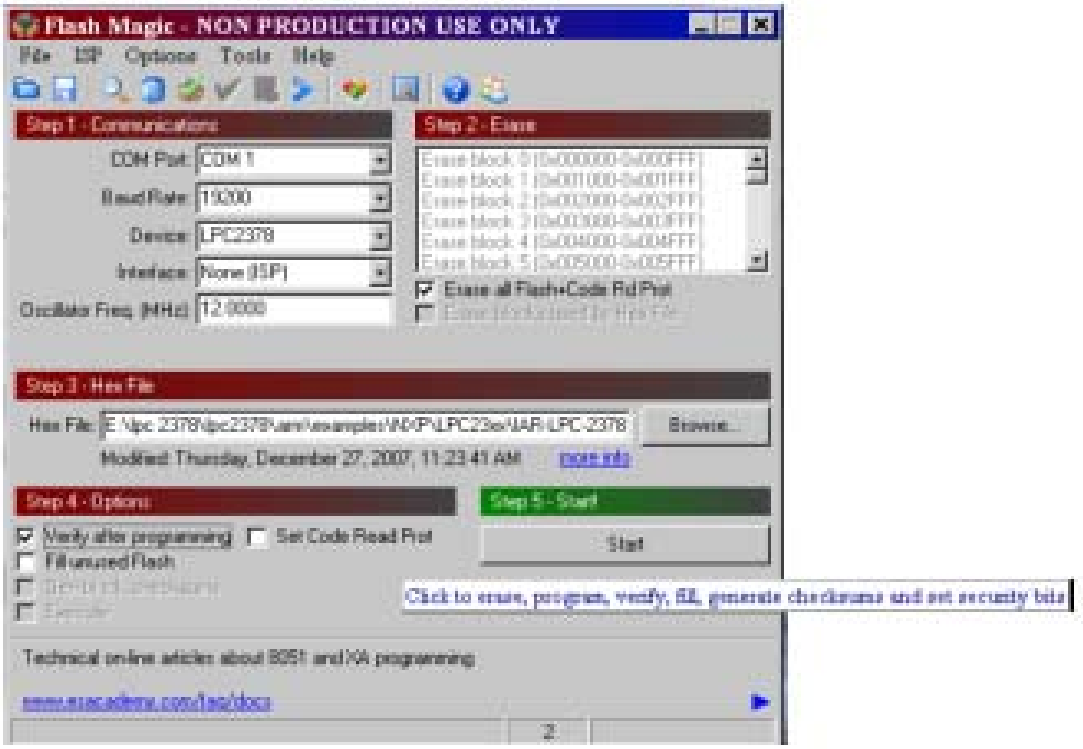
4. Select Micro-controller Device **LPC2378.**



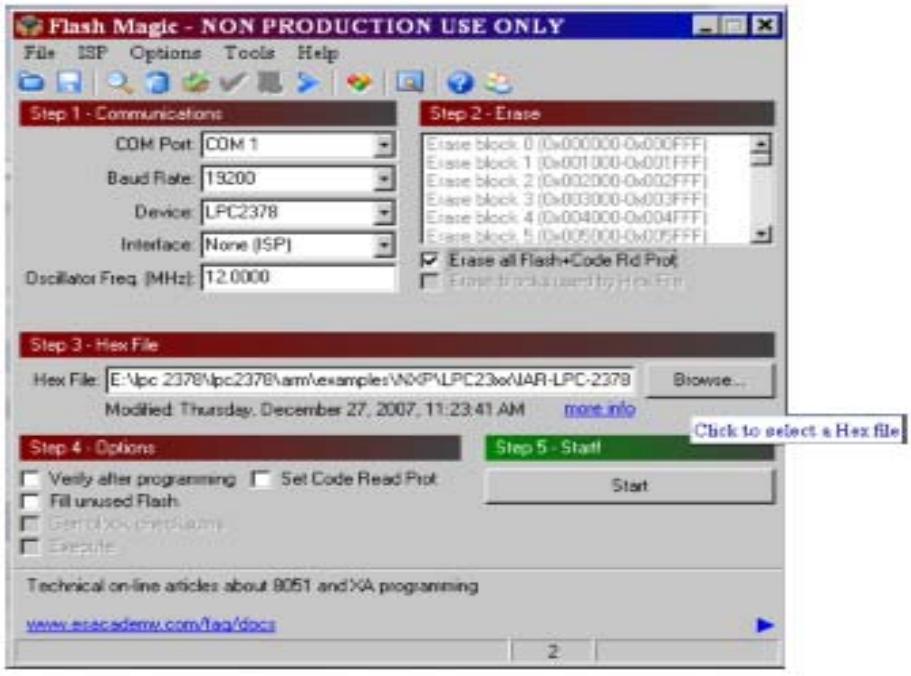
5. Select Oscillator frequency **12Mhz.**



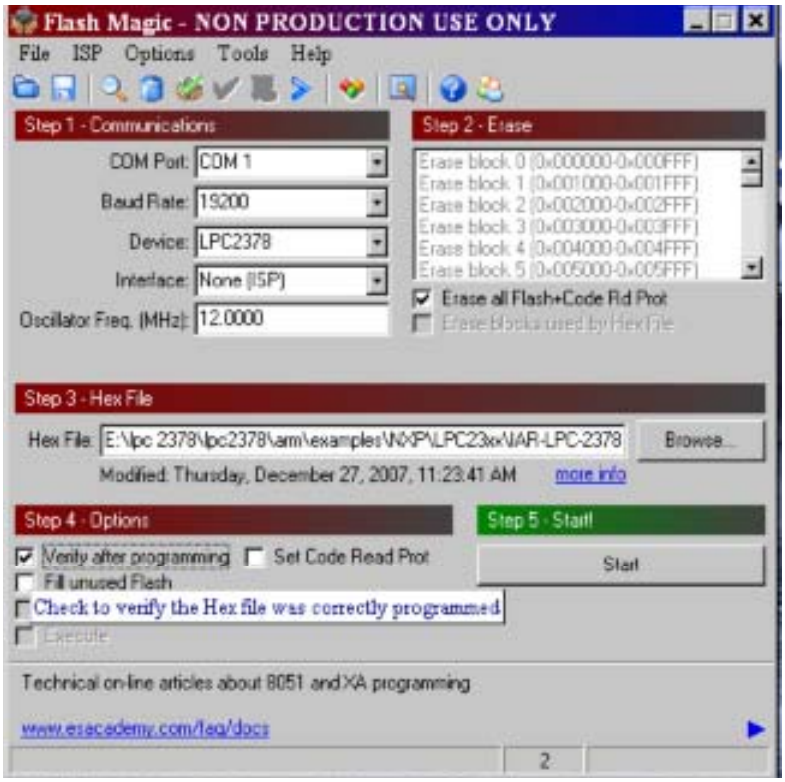
5. Enable **Erase Blocks** for Erasing the memory locations of micro-controller, before programming in **Step-2** Section.



7. Click the **Browse** Button, for Select the **HEX** file to be downloaded in **Step-3** Section.



8. Enable **Verify After Programming** Option in Step-4 Section, for verifying data's in memory locations after programming success



9. Select **Start** Button in Step-5, for **hex** file downloaded to the LPC – 2378 Controller.

10. If given HEX file is properly downloaded, the window as shown in below.

11. The downloaded Program will start after changing the Mode of Execution (Change the Toggle Switch to **Execution Mode** and Press **Reset Key**).

CHAPTER – 4
ABOUT RTOS

REAL TIME SYSTEMS

A RTOS is the readymade operating system like a WINDOWS, XP for your Pentium computer applications, used for real time embedded applications. Like any OS it consists of a kernel, board level routines & other utilities (device drivers) to support the applications. In RTOS time constraints is very critical. The embedded operating system need only less memory & also possible for the developer to write his/her own operating system kernel.

µC/OS-II

Memory is allocated and de-allocated from a pool with deterministic, often constant, timing. µC/OS-II, The Real time kernel is a completely portable, ROMable, scalable, preemptive, real-time, multi tasking kernel for microprocessors and micro controllers. **µC/OS-II** can manage up to 64 application tasks and provides the following services:

1. Semaphores
2. Event Flags
3. Mutual Exclusion Semaphores (to reduce priority inversions)
4. Message Mailboxes
5. Message Queues
6. Task Management (Create, Delete, Change Priority, Suspend/Resume etc.)
7. Fixed Sized Memory Block management
8. Time Management

TASK MANAGEMENT

It services that create a task, delete a task, check the size of task's stack, changes a task priority, suspend and resume a task.

OS FUNCTIONS

- OSTaskCreate ()
- OSinit ()
- OSStart ()
- OS_STK
- OSTaskNameSet ()

1. OSTaskCreate ()

OSTaskCreate creates a task so it can managed by μC/OS-II. Tasks can be created either prior to the start of multitasking or by a running a task.

Syntax

OSTaskCreate(void(*task)(void *pd),void *pdata,OS_STK *ptos, INT8U prio)

Eg1:

OSTaskCreate (Task1, 0, &Task1stk [99], 1);

2. OS_STK

Each task requires its own stack; however μC/OS-II allows each task to have a different stack size, which allows you to reduce the amount of RAM needed in your applications.

Eg2:

```
OS_STK      Task1stk [100];
OS_STK      Task2stk [100];
OS_STK      Task3stk [100];
OS_STK      Task4stk [300];
```

3. OSInit ()

OSInit () initializes μC/OS-II and must be called prior to calling OSStart (), which actually starts multitasking.

Eg3:

```
Void main ()
{
 OSInit ();                          //Initialize μC/OS-II
  OSTaskCreate (Task1, 0, &Task1stk [99], 1);
  .
  .
OSStart ();                          //start multitasking
}
```

4. OSStart ()

OSStart () starts multitasking under μC/OS-II. This function is typically called from your startup code but after you call OSInit (). Refer Eg3.

5. OSTaskNameSet ()

OSTaskNameSet () allows you to assign name to a task. This function is typically used by a debugger to allow associating a name to a task.

Syntax:

Void OSTaskNameSet (INT8U prio, INT8U *pname, INT8U *err);

Eg4: OSTaskNameSet (1,"T1", &err);

TIME MANAGEMET

OS FUNCTIONS:

- OSTimeDly ()
- OSTimeDlyHMSM ()
- OSTimeTick ()

1. OSTimeDly ()

OSTimeDly () allows a task to delay itself for an integral number of clock ticks. Rescheduling always occurs when the number of clock ticks is greater than zero. Valid delays range from one to 65,535 ticks.

Syntax:

Void OSTimeDly (INT16U TICKS);

Eg4:

```
Void Task1(void *pdata)
{
  While (1)
  {
    FIO4SET0 = 0x04;
    OSTimeDly (30);    //delay task for 30 ticks
}
}
```

2. OSTimeDlyHMSM ()

OSTimeDlyHMSM () allows a task to delay itself for a user specified amount of time specified in hours, minutes, seconds, and milliseconds. This format is more convenient and natural than ticks. Rescheduling always occurs when at least one parameter is zero.

Syntax:

OSTimeDlyHMSM (INT8U hours, INT8U minutes, INT8U seconds, INT8U milliseconds);

Eg5:

```
Void task1()
{
  GLCD_Printf("NXP LPC2378",&Font_7x8);
  OSTimeDlyHMSM(0,0,5,0);
}
```

3. OSTimeTick ()

OSTimeTick () processes a clock tick.

Syntax:

Void OSTimeTick (void)

Eg6:

```
Static void  Tmr_TickInit (void)
{

CPU_INT32U pclk_freq;
CPU_INT32U rld_cnts;

VICIntSelect &= ~(1 << VIC_TIMER0);     /* Configure the timer interrupt as an
                                           IRQ source */
VICVectAddr4 =(CPU_INT32U)Tmr_TickISR_Handler;/* Set the vector address*/

VICIntEnable  = (1 << VIC_TIMER0);       /* Enable the timer interrupt source*/

pclk_freq =   BSP_CPU_PclkFreq(PCLK_TIMER1);  /* Get the peripheral clock
                                                 frequency*/
```

```
rld_cnts  =   pclk_freq / OS_TICKS_PER_SEC;  /* Calculate the # of counts
                                                necessary for the OS ticker    */
T0TCR   =  (1 << 1);     /* Disable and reset counter 0 and the prescale counter 0   */
T0TCR   =   0;            /* Clear the reset bit*/
T0PC    =   0;            /* Prescaler is set to no division*/
T0MR0   =   rld_cnts;
T0MCR  =   3;           /* Interrupt on MR0 (reset TC), stop TC*/
T0CCR   =   0;           /* Capture is disabled*/.
T0EMR   =   0;           /* No external match output*/.
T0TCR  = 1;              /* Enable timer 0 */
}
```

SAMPLE PROGRAMS:

1. PROGRAM FOR LED'S AND GRAPHICS LCD USING µC/OS-II

```
#include<includes.h>
#include<images.h>

#define  DEF_FALSE  0
#define  DEF_TRUE    1
unsigned char data[]="DATA FROM LPC2378 DEVELOPMENT
BOARD...\n";
unsigned int value;
void *MessageStorage[10];

OS_STK        Task1stk[100];
OS_STK        Task2stk[100];
OS_STK        Task3stk[100];
OS_STK        Task4stk[300];

void uCdetails();
char KeyRead(char);

void Task1(void *pdata);
void Task2(void *pdata);
void Task3(void *pdata);
void Task4(void *pdata);
```

```
void Task1(void *pdata)

{
  GLCD_ClearDisplay();
  GLCD_LocateCursor(25,2);
  GLCD_Printf("uC/OS - II",&Font_7x8);
  GLCD_LocateCursor(5,3);
  GLCD_Printf("VI Microsystems",&Font_7x8);
  GLCD_LocateCursor(20,4);
  GLCD_Printf("NXP LPC2378",&Font_7x8);
  OSTimeDlyHMSM(0,0,5,0);
  GLCD_ClearDisplay();

  uCdetails();

  for(;;)
  {
    value = (CPU_INT32U)OSCPUUsage;
    GLCD_LocateCursor(70,3);
    GLCD_DisplayValue(value,3,0);

    value = (CPU_INT32U)OSTime;
    GLCD_LocateCursor(70,5);
    GLCD_DisplayValue(value,8,0);

    value = (CPU_INT32U)OSCtxSwCtr;
    GLCD_LocateCursor(70,6);
    GLCD_DisplayValue(value,8,0);
    OSTimeDly(20);

  }
}

void Task2(void *pdata)
{
  while(1)
  {
    FIO4SET0 = 0x04;
    OSTimeDly(30);
    FIO4CLR0 = 0x04;
    OSTimeDly(20);

  }
}
```

```
void Task3(void *pdata)
{
  for(;;)
  {
    FIO4SET0 = 0x10;
    OSTimeDly(50);
    FIO4CLR0 = 0x10;
    OSTimeDly(50);
  }
}

void Task4(void *pdata)
{
  for(;;)
  {
    FIO4SET0 = 0x40;
    OSTimeDly(70);
    FIO4CLR0 = 0x40;
    OSTimeDly(70);
  }
}

void main()
{
  CPU_INT08U      err;
  BSP_IntDisAll();
  BSP_Init();
  GLCD_Init();
  GLCD_ClearDisplay();
  OSInit();
  OSTaskCreate(Task1, 0, &Task1stk[99], 1);
  OSTaskCreate(Task2, 0, &Task2stk[99], 2);
  OSTaskCreate(Task3, 0, &Task3stk[99], 3);
  OSTaskCreate(Task4, 0, &Task4stk[299], 4);
  OSTaskNameSet(1,"T1", &err);
  OSTaskNameSet(2,"T2", &err);
  OSTaskNameSet(3,"T3", &err);
  OSTaskNameSet(4,"T4", &err);
  OSStart();
}
```

```
void uCdetails()
{
  GLCD_LocateCursor(20,0);
  GLCD_Printf("OS DETAILS",&Font_7x8);
  GLCD_LocateCursor(0,1);
  GLCD_Printf("Version :V",&Font_7x8);
  value = (CPU_INT32U)OSVersion();
  GLCD_LocateCursor(80,1);
  GLCD_DisplayValue(value,1,2);

  GLCD_LocateCursor(0,2);
  GLCD_Printf("TickRate:",&Font_7x8);
  value = (CPU_INT32U)OS_TICKS_PER_SEC;
  GLCD_LocateCursor(70,2);
  GLCD_DisplayValue(value,3,0);
  GLCD_LocateCursor(88,2);
  GLCD_Printf("/SEC",&Font_3x6);

  GLCD_LocateCursor(0,3);
  GLCD_Printf("CPU Used:",&Font_7x8);
  GLCD_LocateCursor(88,3);
  GLCD_Printf("%",&Font_7x8);

  GLCD_LocateCursor(0,4);
  GLCD_Printf("CPU Freq:",&Font_7x8);
  value = (CPU_INT32U)BSP_CPU_ClkFreq() / 1000000L;
  GLCD_LocateCursor(70,4);
  GLCD_DisplayValue(value,2,0);
  GLCD_LocateCursor(84,4);
  GLCD_Printf("MHz",&Font_7x8);

  GLCD_LocateCursor(0,5);
  GLCD_Printf("OS Ticks:",&Font_7x8);

  GLCD_LocateCursor(0,6);
  GLCD_Printf("Ctx Swt :",&Font_7x8);
}
```

PROCDEURE

1. To write a multi tasking program using μC/OS-II real time kernel.

2. Include following μC/OS-II header files.

3. That files are taken from below folders are

APP

- OS_CFG.H
- MAIN.C

BSP

- BSP.C
- BSP.H

PORT

- OS_CPU.H
- OS_CPU_A.ASM
- OS_CPU_C.C
- OS_DBG.C
- OS_DCC.C

UC_CPU

- CPU.H
- CPU_A.S
- CPU_DEF.H

UCOS-II

- OS_CORE.C
- OS_FLAG.C
- OS_MBOX.C
- OS_MEM.C
- OS_MUTEX.C
- OS_Q.C
- OS_SEM.C
- OS_TASK.C
- OS_TIME.C
- OS_TMR.C
- UCOSII.H

4. Compile your project. And executing the program.

2. PROGRAM FOR LED'S USING μC/OS-II

```
#include<includes.h>

OS_STK        Task1stk[100];
OS_STK        Task2stk[100];
OS_STK        Task3stk[100];

void Task1(void *pdata);
void Task2(void *pdata);
void Task3(void *pdata);
void Task1(void *pdata)

{
  BSP_Init();
  FIO4DIR0=0xff;
  for(;;)
  {
    FIO4SET0 = 0x01;
    OSTimeDly(10);
    FIO4CLR0 = 0x01;
    OSTimeDly(10);
  }
}

void Task2(void *pdata)
{
  for(;;)
  {
    FIO4SET0 = 0x04;
    OSTimeDly(20);
    FIO4CLR0 = 0x04;
    OSTimeDly(20);
  }
}

void Task3(void *pdata)
{
  for(;;)
  {
    FIO4SET0 = 0x10;
    OSTimeDly(30);
    FIO4CLR0 = 0x10;
    OSTimeDly(30);
```

```
  }
}

void main()
{
  BSP_IntDisAll();
  OSInit();
  OSTaskCreate(Task1, 0, &Task1stk[99], 1);
  OSTaskCreate(Task2, 0, &Task2stk[99], 2);
  OSTaskCreate(Task3, 0, &Task3stk[99], 3);
  OSStart();
}
```

NOTE

The Procedure as same as program 1

CHAPTER – 5
SOFTWARE EXAMPLE

Example – 1 Program For 8 - Bit LED Interface

```
#include <iolpc2378.h>
#include "irq.h"
#include "config.h"

void delay()
{
  unsigned int i,j;
  for(i=0;i<0x3fff;i++)
  for(j=0;j<0xff;j++);
}
int main (void)
{
  unsigned int Fdiv;
  TargetResetInit();

    PINSEL0 = 0x00000050;                 /* RxD0 and TxD0 */
    U0LCR = 0x83;                         /* 8 bits, no Parity, 1 Stop bit */
    Fdiv = ( Fpclk / 16 ) / 19200 ;       /*baud rate */
    U0DLM = Fdiv / 256;
    U0DLL = Fdiv % 256;
    U0LCR = 0x03;                         /* DLAB = 0 */
    U0FCR = 0x07;                         /* Enable and reset TX and RX
     FIFO.
    FIO3DIR = 0X008000FF;

    while(1)
    {
    FIO3PIN = 0X000000ff;
    delay();
    FIO3PIN = 0X00000000;
    delay();
    }

}
```

Example – 2 Program For 8-Way DIP Switch Interface

```
#include <iolpc2378.h>
#include "irq.h"
#include "config.h"

unsigned int k;
unsigned int dat[] = {0x00,0x01,0x02,0x04,0x08,0x10,0x20,0x40,0x80};
void delay()
{
  unsigned int i,j;
  for(i=0;i<0x1Fff;i++)
  for(j=0;j<0xff;j++);
}
int main (void)
{
  TargetResetInit();
  FIO4PIN = 0X0;
  FIO4DIR = 0XFFFF0000;
  FIO3DIR = 0X008000FF;
  while(1)
  {
    FIO3PIN = FIO4PIN ;
  }

}
```

Example – 3 Program For 4 x 4 Matrix Keypad Interface

```
#include <iolpc2378.h>
#include "irq.h"
#include "config.h"

unsigned int k;
unsigned int Read_Key;
unsigned char scan [] = {
0x00000E00,0x00000D00,0x00000B00,0x00000700};
unsigned int i;
void delay()
{
  unsigned int i,j;
  for(i=0;i<0xff;i++)
  for(j=0;j<0xff;j++);
}
```

```
void send_serial_data(unsigned char serial)
{

   while((U0LSR & 0x20)==0);
  U0THR = serial;
 }

int main (void)
{
   unsigned int Fdiv;
   TargetResetInit();

   PINSEL0 = 0x00000050;          /* RxD0 and TxD0 */
   U0LCR = 0x83;                  /* 8bits, no Parity, 1 Stop bit */
   Fdiv = ( Fpclk / 16 ) / 19200 ;    /*baud rate */
   U0DLM = Fdiv / 256;
   U0DLL = Fdiv % 256;
   U0LCR = 0x03;                  /* DLAB = 0 */
   U0FCR = 0x07;                  /* Enable and reset TX and RX FIFO. */
   FIO4DIR = 0X00000FFF;
   FIO3DIR = 0X008000FF;

  while(1)
  {

     FIO4SET = 0X00000e00;        /*First Row*/
     Read_Key = FIO4PIN;
     Read_Key = (Read_Key & 0xf000) >> 12 ;

     if((Read_Key==0x07))
       {
       send_serial_data('0');
       FIO3PIN = 0X00000000;
       }
     if((Read_Key==0x0b))
       {
       send_serial_data('1');
       FIO3PIN = 0X00000001;
       }
     if( (Read_Key==0x0d))
       {
       send_serial_data('2');
       FIO3PIN = 0X00000002;
       }
```

```
if((Read_Key==0x0e))
  {
  send_serial_data('3');
  FIO3PIN = 0X00000003;
  }

FIO4CLR = 0X00000e00;
delay();
FIO4SET = 0X00000d00;                /*Second Row*/
Read_Key = FIO4PIN;
Read_Key = (Read_Key & 0xf000) >> 12 ;

  if((Read_Key==0x07))
    {
    send_serial_data('4');
    FIO3PIN = 0X00000004;
    }
 if((Read_Key==0x0b))
    {
    send_serial_data('5');
    FIO3PIN = 0X00000005;
    }
 if( (Read_Key==0x0d))
    {
    send_serial_data('6');
    FIO3PIN = 0X00000006;
    }
 if( (Read_Key==0x0e))
    {
    send_serial_data('7');
    FIO3PIN = 0X00000007;
    }
FIO4CLR = 0X00000d00;
delay();

FIO4SET = 0X00000b00;                /*Third Row*/
Read_Key = FIO4PIN;
Read_Key = (Read_Key & 0xf000) >> 12 ;

  if((Read_Key==0x07))
     {
     send_serial_data('8');
     FIO3PIN = 0X00000008;
     }
```

```
        if((Read_Key==0x0b))
          {
           send_serial_data('9');
           FIO3PIN = 0X00000009;
          }
        if( (Read_Key==0x0d))
          {
          send_serial_data('a');
          FIO3PIN = 0X0000000a;
          }
        if( (Read_Key==0x0e))
          {
          send_serial_data('b');
          FIO3PIN = 0X0000000b;
          }
        FIO4CLR = 0X00000b00;
        delay();

      FIO4SET = 0X00000700;                /*Fourth Row*/
      Read_Key = FIO4PIN;
      Read_Key = (Read_Key & 0xf000) >> 12 ;

          if((Read_Key==0x07))
           {
           send_serial_data('c');
           FIO3PIN = 0X0000000c;
            }
         if((Read_Key==0x0b))
           {
            send_serial_data('d');
            FIO3PIN = 0X0000000d;
           }
         if((Read_Key==0x0d))
           {
           send_serial_data('e');
           FIO3PIN = 0X0000000e;
           }
         if( (Read_Key==0x0e))
           {
           send_serial_data('f');
           FIO3PIN = 0X0000000f;
           }
         FIO4CLR = 0X00000700;
         delay();
    }
  return 0;
}
```

Example – 4 Program For Serial Interface

```
#include<iolpc2378.h>
#include "type.h"
#include "target.h"
#include "irq.h"
#include "uart.h"

#include <intrinsics.h>
unsigned int ADC_VALUE=0;
#define ADC_CLK     1000000       /* set to 1Mhz */
#include "config.h"

void  PLL_Init (void)
{
  unsigned int  m;
  unsigned int  n;
  unsigned int  clk_div;
  unsigned int  clk_div_usb;

   m          =  24;              /PLL Multiplier = 20, MSEL bits = 12 - 1=11 */
   n          =  1;               /* PLL Divider =  1, NSEL bits =  1 - 1 =  0    */
   clk_div    =  4;               /*Configure the ARM Core clock divCCLKSEL=
                                  6 -1 */
   clk_div_usb =                   /* Configure the USB clock divider to 6, SBSEL
                                  = 6 - 1    */

  if ((PLLSTAT & 0x02000000) > 0)
  {
    PLLCON  &= ~0x02;             /* Disconnect the PLL  */
    PLLFEED  =  0xAA;             /* PLL register update sequence, 0xAA, 0x55 */
    PLLFEED  =  0x55;
  }
  PLLCON   &= ~0x01;              /* Disable the PLL    */
  PLLFEED   =  0xAA;              /* PLL register update sequence, 0xAA, 0x55  */
  PLLFEED   =  0x55;
  SCS      &= ~0x10;              /* OSCRANGE = 0, Main OSC is between 1 and
                                    20 Mhz */
  SCS      |=  0x20;              /* OSCEN = 1, Enable the main oscillator */

  while ((SCS &  0x40) == 0)

  CLKSRCSEL = 0x01;               /* Select main OSC, 12MHz, as the PLL clock
                                     source */
```

```
  PLLCFG    = (m << 0)
                | (n << 16);          /* Configure the PLL multiplier and divider */
  //PLLCFG    = 11;                   /* Configure the PLL multiplier and divider*/
  PLLFEED   = 0xAA;                   /* PLL register update sequence, 0xAA, 0x55  */
  PLLFEED   = 0x55;
  PLLCON   |= 0x01;                   /* Enable the PLL */
  PLLFEED   = 0xAA;                   /* PLL register update sequence, 0xAA, 0x55  */
  PLLFEED   = 0x55;
  CCLKCFG   = clk_div;                /*Configure the ARM Core Processor clock
                                        divider    */
  USBCLKCFG = clk_div_usb; /* Configure the USB clock divider */

  while ((PLLSTAT & 0x04000000) == 0)
  PCLKSEL0  = 0xAAAAAAAA;/*Set peripheral clocks to be half of main clock
*/
  PCLKSEL1  = 0x22AAA8AA;
  PLLCON   |= 0x02;                   /* Connect the PLL. The PLL is now the active
                                        clock source  */
  PLLFEED   = 0xAA;                  /* PLL register update sequence, 0xAA, 0x55  */
  PLLFEED   = 0x55;
  while ((PLLSTAT & 0x02000000) == 0)

  PCLKSEL0 = 0x55555555;  /* PCLK is the same as CCLK */
  PCLKSEL1 = 0x55555555;
}
void send_serial_data(unsigned char serial)
{

  while((U0LSR & 0x20)==0);
   U0THR = serial;
}

void adc_serial_tx(unsigned int ch)
{

    unsigned int t1000,t100,t10,t1,temp;
    t1000 = ch   / 1000;
    temp  = ch   % 1000;
    t100  = temp / 100;
    temp  = temp % 100;
    t10   = temp / 10;
    t1    = temp % 10;
    send_serial_data(t1000+0x30);
    send_serial_data(t100 +0x30);
```

```
    send_serial_data(t10+0x30);
    send_serial_data(t1+0x30);
    send_serial_data(0x0d);
    send_serial_data(0x0a);

}

    int putchar(int ch)
{
  if (ch == '\n')
  {
    while (!(U0LSR & 0x20));
    U0THR = 0x0d;
  }
  while (!(U0LSR & 0x20));
  return (U0THR = ch);
 }

void main()
{
     unsigned long int val;
     unsigned int Fdiv;
     PLL_Init();

     PCONP |= (1 << 12);
     PINSEL1 = 0X00054000;          //AD0.0 AS ANALALOG INPUT
     PINSEL0 = 0x00000050;          // RxD0 and TxD0

     U0LCR = 0x83;                  // 8 bits, no Parity, 1 Stop bit
     Fdiv = (60000000 / 16 ) / 19200 ; //baud rate
     U0DLM = Fdiv / 256;
     U0DLL = Fdiv % 256;
     U0LCR = 0x03;                  // DLAB = 0
     U0FCR = 0x07;                  // Enable and reset TX and RX FIFO.
    send_serial_data('a');
   install_irq( UART0_INT, (void *)UART0Handler, HIGHEST_PRIORITY ) ;
     while(1)
     {
       send_serial_data('a');
       printf("\n\r Welcome");
     }

         }
```

Example – 5 Program for SPDT Relay Interface

```
#include <iolpc2378.h>
#include "irq.h"
#include "config.h"

void delay()
{
  unsigned int i,j;
  for(i=0;i<0x2Fff;i++)
    for(j=0;j<0xff;j++);
}
int main (void)
{
  TargetResetInit();

  FIO3PIN = 0X0;
  FIO3DIR = 0XFFFFFFFF;

  while(1)
  {
    FIO3SET = 0x05000000;
    delay();
    FIO3CLR = 0x05000000;
    delay();

  }
}
```

Example – 6 Program for Stepper Motor Interface

```
#include <iolpc2378.h>
#include "irq.h"
#include "config.h"

void delay()
{
  unsigned int i,j;
  for(i=0;i<0xff;i++)
    for(j=0;j<0xff;j++);
}
int main (void)
{
  TargetResetInit();
```

```
  IO0DIR  = 0XFFFFFFFF;

  while(1)
  {
    for(k=0;k<10;k++)
    {
      IO0SET = 0X00000240;
      delay();
      IO0CLR = 0X00000240;
      IO0SET = 0X00000140;
      delay();
      IO0CLR = 0X00000140;
      IO0SET = 0X00000180;

      delay();
      IO0CLR = 0X00000180;
      IO0SET = 0X00000280;
      delay();
      IO0CLR = 0X00000280;
    }
  }
}
```

Example – 7 Program for Joystick Interface

```
#include <iolpc2378.h>
#include "irq.h"
#include "config.h"
unsigned int k;
unsigned int i;

int main (void)
{
  unsigned long int joy_stick;
  TargetResetInit();

    FIO3DIR = 0X000000FF;  //LED0.
    IO1DIR  = 0X0000FFFF;
    FIO4PIN = 0X00000000;
    while(1)
    {
        joy_stick = IO1PIN ;
        joy_stick = (joy_stick & 0XFFFF0000) >> 16 ;

        if((joy_stick & 0XFF) ==0xfb)
         FIO3PIN = 0XF0;
       if((joy_stick & 0XFF) ==0xf7)
         FIO3PIN = 0X0F;

        if((joy_stick & 0XFF) ==0xBF)
         FIO3PIN = 0X44;

        joy_stick = IO1PIN ;
        joy_stick = joy_stick  >> 20 ;

        if((joy_stick & 0xff)==0x7F)
          FIO3PIN = 0X22;

        if((joy_stick & 0x20) ==0x0)
          FIO3PIN = 0XFF;
    }
      return 0;
}
```

Example – 8 Program for ADC Interface

```
#include<iolpc2378.h>
#include "irq.h"
#include "config.h"

unsigned int ADC_VALUE=0;
#define ADC_CLK     1000000                    set to 1Mhz */
void send_serial_data(unsigned char serial)

{
  while((U0LSR & 0x20)==0);
     U0THR = serial;
}

void adc_serial_tx(unsigned int ch)
{
   unsigned int t1000,t100,t10,t1,temp;
   t1000 = ch   / 1000;
   temp  = ch   % 1000;
   t100   = temp / 100;
   temp  = temp % 100;
   t10    = temp / 10;
   t1      = temp % 10;

   send_serial_data(t1000+0x30);
   send_serial_data(t100 +0x30);
   send_serial_data(t10+0x30);
   send_serial_data(t1+0x30);
   send_serial_data(0x0d);
   send_serial_data(0x0a);
}
void main()
{
     unsigned long int val;
     unsigned int Fdiv;
     TargetResetInit();
     PCONP |= (1 << 12);
     PINSEL1 = 0X00054000;        //AD0.0 AS ANALALOG INPUT
     PINSEL0 = 0x00000050;        // RxD0 and TxD0
     U0LCR = 0x83;                // 8 bits, no Parity, 1 Stop bit
     Fdiv = ( Fpclk / 16 ) / 19200 ;   // baud rate
     U0DLM = Fdiv / 256;
     U0DLL = Fdiv % 256;
```

```
    U0LCR = 0x03;              // DLAB = 0
    U0FCR = 0x07;              // Enable and reset TX and RX FIFO.

      AD0CR = ( 0x01 << 0 ) | /*SEL=1,select channel 0~7 on ADC0 */
       ( ( Fpclk / ADC_CLK - 1 ) << 8 ) |  /* CLKDIV = Fpclk / 1000000 - 1 */
       ( 1 << 16 ) |           /* BURST = 0, no BURST, software controlled */
       ( 0 << 17 ) |           /* CLKS = 0, 11 clocks/10 bits */
       ( 1 << 21 ) |           /* PDN = 1, normal operation */
       ( 0 << 22 ) |           /* TEST1:0 = 00 */
       ( 1 << 24 ) |            /* START = 0 A/D conversion stops */
       ( 0 << 27 );            /* EDGE = 0 (CAP/MAT singal falling,trigger A/D
                               conversion) */

        while(1)
    {
       while((AD0GDR & 0X80000000)!=0X80000000);
              val = (AD0GDR>>6)& 0x3ff
        adc_serial_tx(val);
    }
}
```

Example – 9 Program DAC Interface

```
#include <iolpc2378.h>
#include "dac.h"

void DACInit( void )
{
   PINSEL1 = 0x00200000;          /* set p0.26 to DAC output */
   return;
}
```

Example – 10 Program Accelerometer Interface

```
#include<iolpc2378.h>
#include "irq.h"
#include "config.h"

unsigned int ADC_VALUE=0;
#define ADC_CLK     1000000                 /* set to 1Mhz */
unsigned long int val;
unsigned long int x,y,z;

void send_serial_data(unsigned char serial)
{

  while((U0LSR & 0x20)==0);
     U0THR = serial;
}

void  adc_serial_tx(unsigned int ch)
{

    unsigned int t1000,t100,t10,t1,temp;
    t1000 = ch   / 1000;
    temp  = ch   % 1000;
    t100  = temp / 100;
    temp  = temp % 100;
    t10   = temp / 10;
    t1    = temp % 10;

    send_serial_data(t1000+0x30);
    send_serial_data(t100 +0x30);
    send_serial_data(t10+0x30);
    send_serial_data(t1+0x30);
}

unsigned long  adc_0()
{

    AD0CR = ( 0x01 << 0 ) |             /* SEL=1,select channel 0~7 on ADC0 */
        ( ( Fpclk / ADC_CLK - 1 ) << 8 ) |  /* CLKDIV = Fpclk / 1000000 - 1 */
        ( 1 << 16 ) |                   /* BURST = 0, no BURST, software
                                           controlled */
        ( 0 << 17 ) |                   /* CLKS = 0, 11 clocks/10 bits */
        ( 1 << 21 ) |       /* PDN = 1, normal operation */
```

```
        ( 0 << 22 ) |        /* TEST1:0 = 00 */
        ( 1 << 24 ) |        /* START = 0 A/D conversion stops */
        ( 0 << 27 );         /* EDGE = 0 (CAP/MAT signal falling, trigger A/D
                             conversion) */
        while((AD0GDR & 0X80000000)!=0X80000000);
        val = (AD0GDR>>6)& 0x3ff
       return(val);
}

unsigned long adc_1()
{

    AD0CR = ( 0x01 << 1 ) |          /* SEL=1,select channel 0~7 on ADC0 */
        ( ( Fpclk / ADC_CLK - 1 ) << 8 ) |  /* CLKDIV = Fpclk / 1000000 - 1 */
        ( 1 << 16 ) |                /* BURST = 0, no BURST, software
controlled */
        ( 0 << 17 ) |                /* CLKS = 0, 11 clocks/10 bits */
        ( 1 << 21 ) |                /* PDN = 1, normal operation */
        ( 0 << 22 ) |                /* TEST1:0 = 00 */
        ( 1 << 24 ) |                /* START = 0 A/D conversion stops */
        ( 0 << 27 );                 /* EDGE = 0 (CAP/MAT signal
                                     falling, trigger A/D conversion) */
        while((AD0GDR & 0X80000000)!=0X80000000);
        val = (AD0GDR>>6)& 0x3ff
        return(val);
}

unsigned long  adc_6()
{
           AD0CR = ( 0x01 << 6 ) |   /* SEL=1,select channel 0~7 on ADC0 */
        ( ( Fpclk / ADC_CLK - 1 ) << 8 ) |  /* CLKDIV = Fpclk / 1000000 - 1 */
        ( 1 << 16 ) |    /* BURST = 0, no BURST, software controlled */
        ( 0 << 17 ) |    /* CLKS = 0, 11 clocks/10 bits */
        ( 1 << 21 ) |    /* PDN = 1, normal operation */
        ( 0 << 22 ) |    /* TEST1:0 = 00 */
        ( 1 << 24 ) |    /* START = 0 A/D conversion stops */
        ( 0 << 27 );     /* EDGE = 0 (CAP/MAT signal falling, trigger A/D
                           conversion) */

        while((AD0GDR & 0X80000000)!=0X80000000);
        val = (AD0GDR>>6)& 0x3ff
        return(val);
}
```

```
void main()
{
     unsigned int Fdiv;
     TargetResetInit();
     PCONP |= (1 << 12);
     PINSEL1 = 0X00054000
     PINSEL0 = 0x00000050;

     U0LCR = 0x83;
     Fdiv = ( Fpclk / 16 ) / 19200 ;
     U0DLM = Fdiv / 256;
     U0DLL = Fdiv % 256;
     U0LCR = 0x03;
     U0FCR = 0x07;              .

    AD0CR = ( 0x01 << 0 ) |          /* SEL=1,select channel 0~7 on ADC0 */
        ( ( Fpclk / ADC_CLK - 1 ) << 8 ) |  /* CLKDIV = Fpclk / 1000000 - 1 */
        ( 1 << 16 ) |                /* BURST = 0, no BURST, software
                                     controlled */
        ( 0 << 17 )  = 0, 11 clocks/10 bits */
        ( 1 << 21 ) |                /* PDN = 1, normal operation */
        ( 0 << 22 ) |                /* TEST1:0 = 00 */
        ( 1 << 24 ) |                /* START = 0 A/D conversion stops */
        ( 0 << 27 );                 /* EDGE = 0 (CAP/MAT signal falling,
                                     trigger A/D conversion) */
    while(1)
    {
      y = adc_0();
      x = adc_1();
      z = adc_6();

      send_serial_data('X');
      send_serial_data(':');
      send_serial_data(' ');
      adc_serial_tx(x);
      send_serial_data(' ');
      send_serial_data(' ');
      send_serial_data('Y');
      send_serial_data(':');
      send_serial_data(' ');
      adc_serial_tx(y);
      send_serial_data(' ');
      send_serial_data(' ');
```

```
        send_serial_data('Z');
        send_serial_data(':');
        send_serial_data(' ');
        adc_serial_tx(z);
        send_serial_data(' ');
        send_serial_data(' ');

        send_serial_data(0x0d);
        send_serial_data(0x0a);

    }
}
```

Example – 11 Program Colour LCD Interface

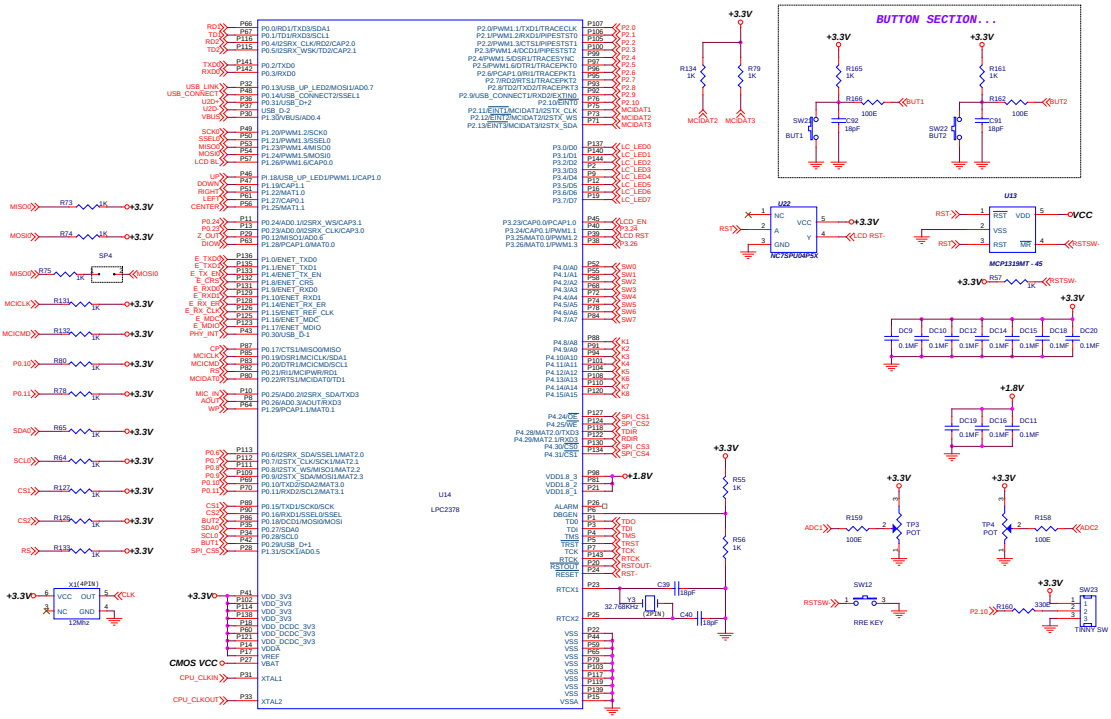
```
#include<iolpc2378.h>
#include<system.h>
#include<glcd_drv.h>
#include<font.h>
#include<math.h>
#include<images.h>
void delay()
{
  for(int Dly = 0xFFFF;Dly;Dly--)
  {
  }
}
char color8(short color){
 char c;
 char r,g,b;
 r=(color & 0x0f00)>>8;
 g=(color & 0x00f0)>>4;
 b=(color & 0x000f);
 c=((r&0x0e)<<4) | ((g & 0x0e)<<1) | ((b & 0x0c)>>2);
 return c;
}
void graphics()
{
  unsigned int  idx;
  unsigned int  x;
  unsigned int  y;
  unsigned int  r;
  unsigned int  g;
  unsigned int  b;
```

```
  unsigned int  temp;
  unsigned int  color;
  for (idx = 512; idx < 1024; idx++)
  {
    x    = rand() % 120;
    y    = rand() % 120;
    r    = rand() % 15;
    g    = rand() % 15;
    b    = rand() % 15;
    temp = idx / 64;
    if (r < temp)
    {
      r = temp;
    }
    if (g < temp)
    {
      g = temp;
    }
    if (b < temp)
    {
      b = temp;
    }
    color = (r << 8) | (g << 4) | b;
    GLCD_FillCircle(x, y, 10, color);
    delay();
  }
}
void LCD_blit(char x,char y,char width,char height,int *buffer)
{
  int i,j,c1,c2;
  y+=2;
  x &= 0xfe;
  GLCD_Send(CMD,PASET);
  GLCD_Send(DATA,y);
  GLCD_Send(DATA,y+height-1);
  GLCD_Send(CMD,CASET);
  GLCD_Send(DATA,x);
  GLCD_Send(DATA,x+width-1);
  GLCD_Send(CMD,RAMWR);
  for (i=0;i<height;i++)
  {
    for (j=0;j<width;j+=2)
    {
      c1=buffer[i+height*j];
```

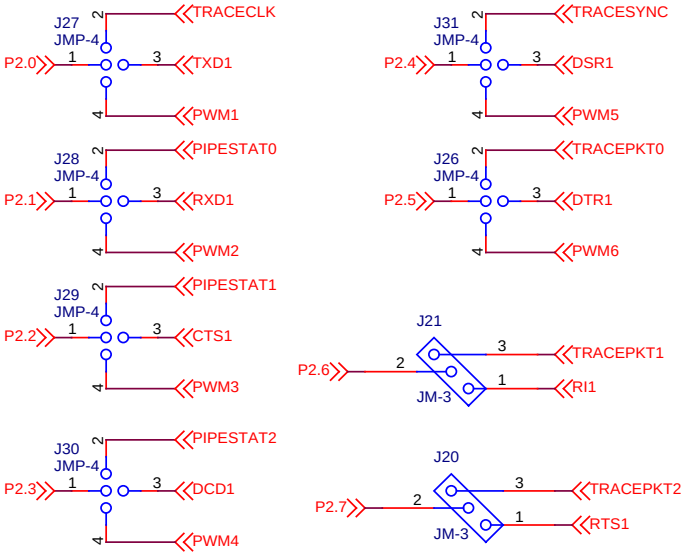
```
      c2=buffer[i+height*j+height];
      GLCD_Send(DATA,(c1 & 0x0ff0)>>4);
      GLCD_Send(DATA,((c1 & 0x000f)<<4) | ((c2 & 0x0f00)>>8));
      GLCD_Send(DATA,(c2 & 0x00ff));
    }
  }
}
void dly2(int a)
{
  long int i;
  long int j;
  j=10000*a;
  for (i=0;i<j;i++)
  {
    ;
  }
}

//buffer for full color animation
int buffer[32][32];

void clear(int *buffer)
{
  int i;
  for(i=0;i<50*50;i++)
    buffer[i]=0;
}

void main()
{

  int dly=1;
  int i;
  PLL_Init();
  MAM_Init();
  SCS |=  1;                    //configure Fast io
  FIO3DIR |= (1UL<<25);         //Reset line
  LCD_LightInit();
  SSP0_Init();
  GLCD_Init();
  LCD_LightSet(0x00);
  GLCD_FillRectangle(0,0,132,132,DarkGreen);
  GLCD_String("VI Microsystems",Font8x16,0,50,LightBlue,Yellow);
}
```

APPENDIX – A

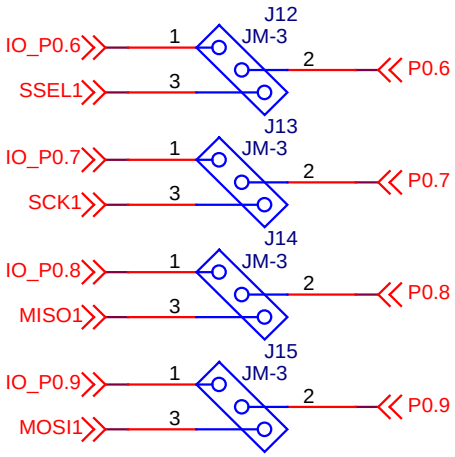
CIRCUIT DIAGRAM

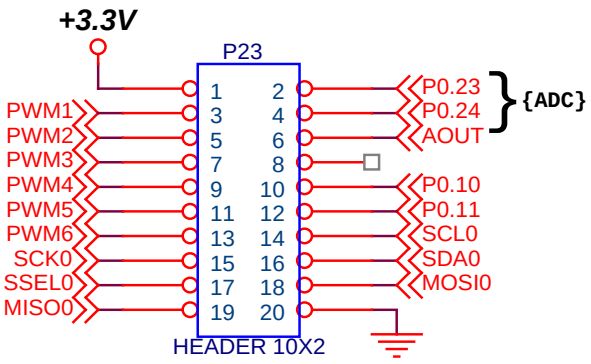


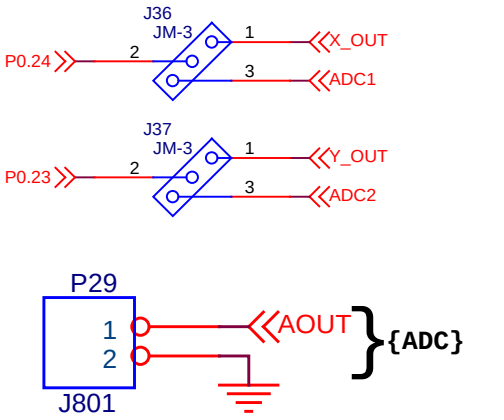
TRACE SIGNALS

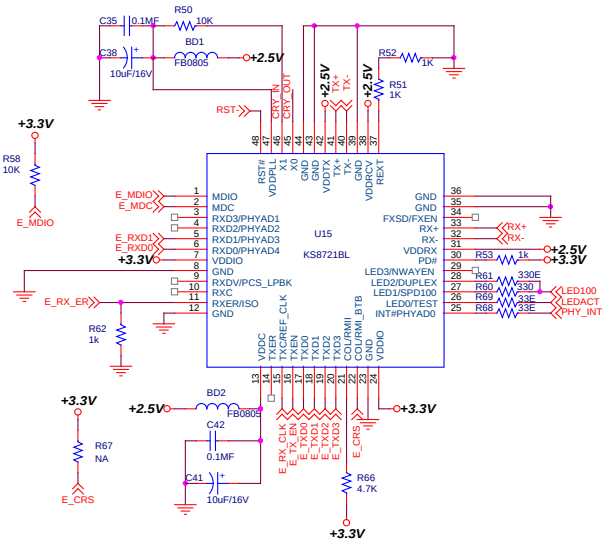


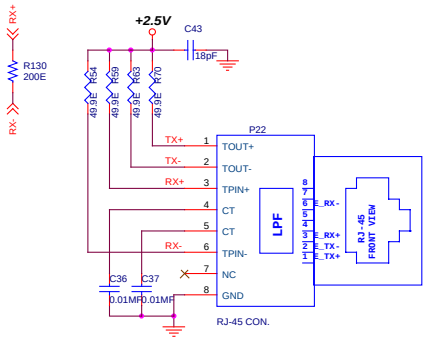
STEPPER MOTOR WITH SPI-1

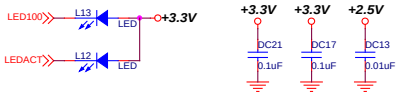




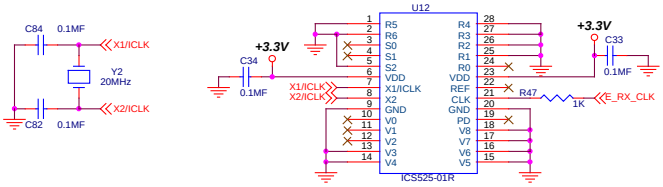


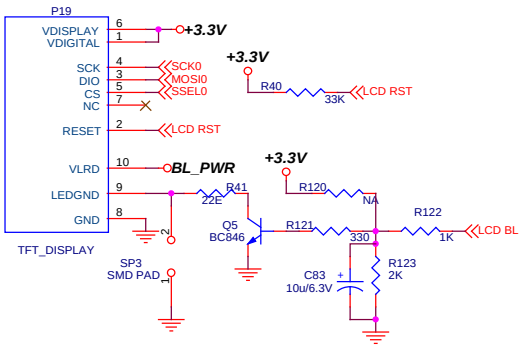


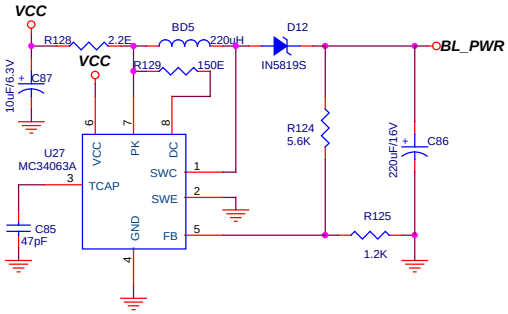




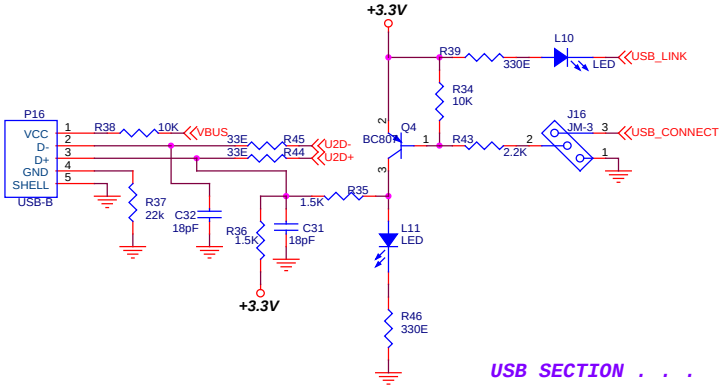
ETHERNET SECTION . . .

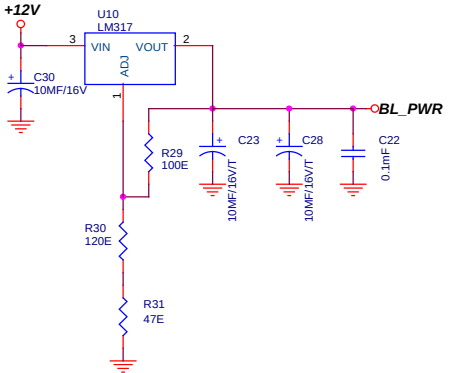




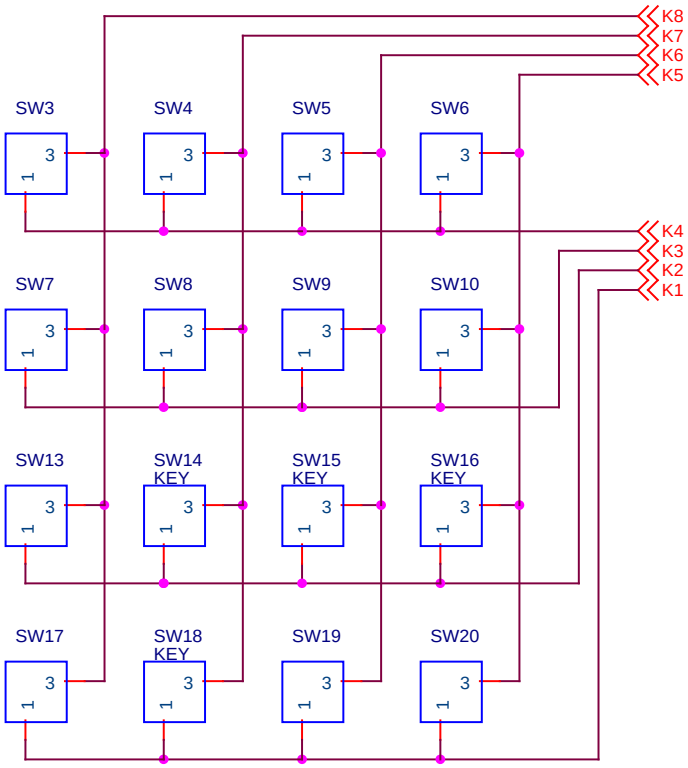


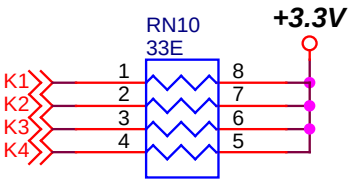
TFT DISPLAY SECTION . . .

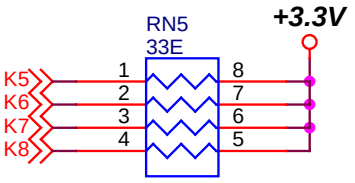


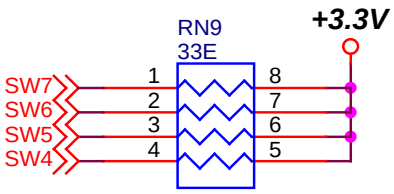


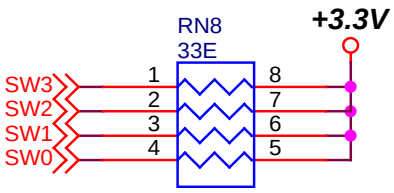
USB SECTION . . .

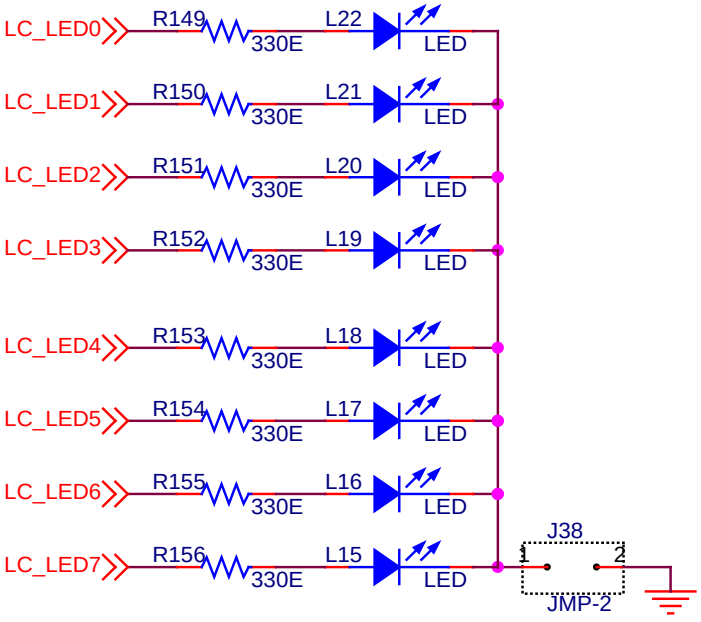


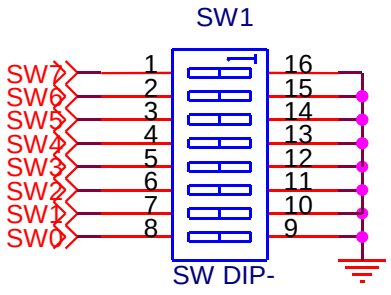






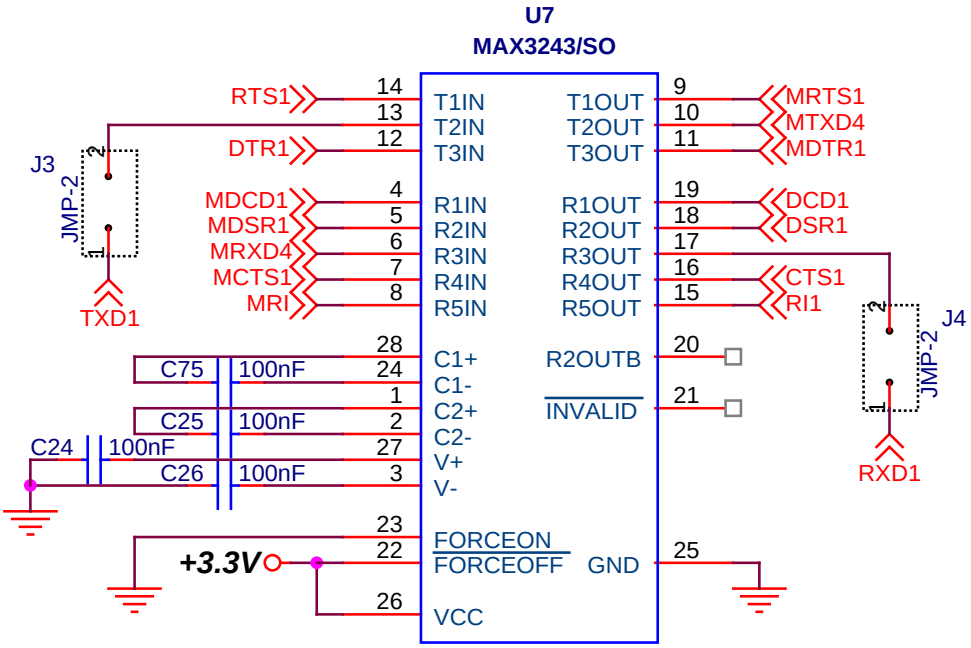


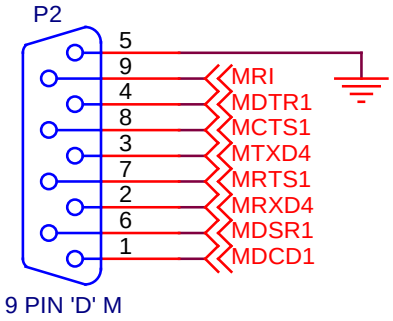


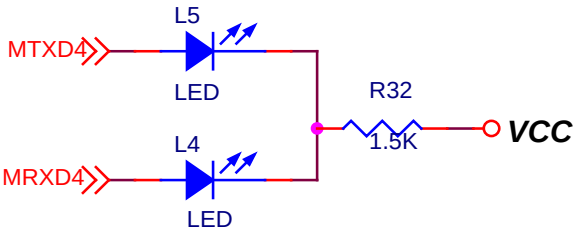


SWITCH & LED SECTION . . .

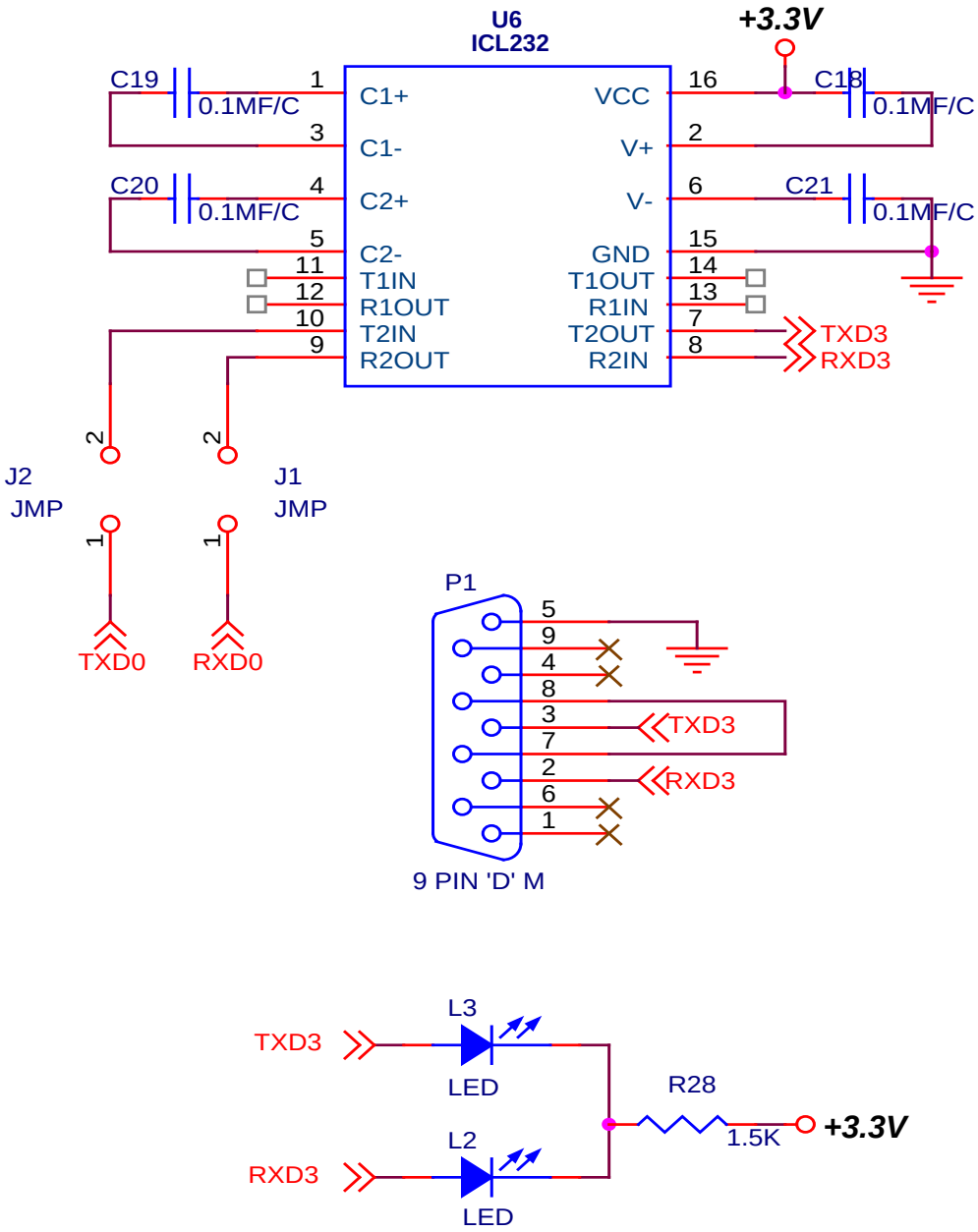
FULL MODEM SECTION . . .



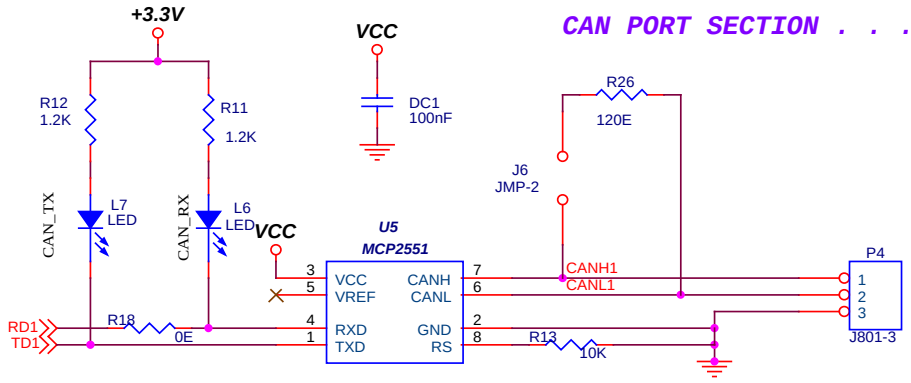




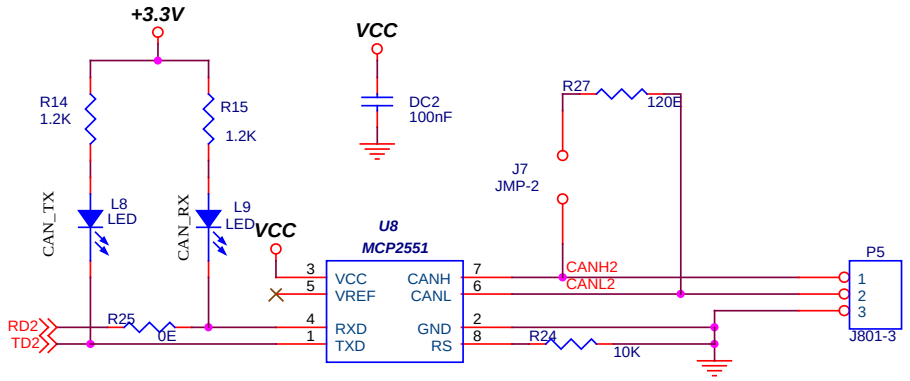
SERIAL PORT SECTION . . .



CAN PORT SECTION . . .

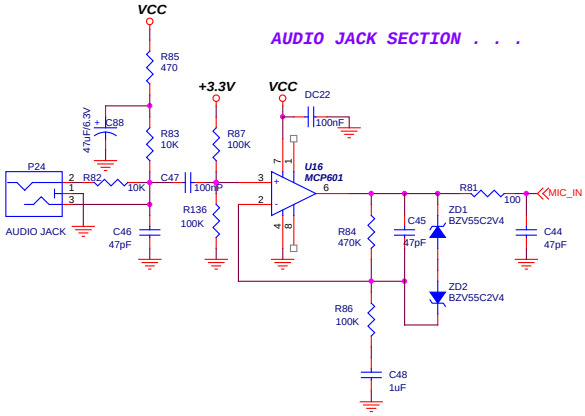


CAN PORT - I

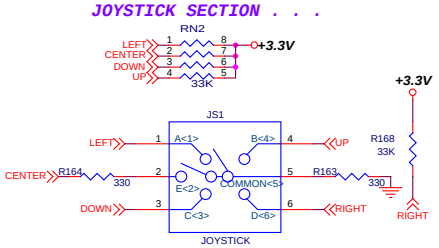


CAN PORT - II

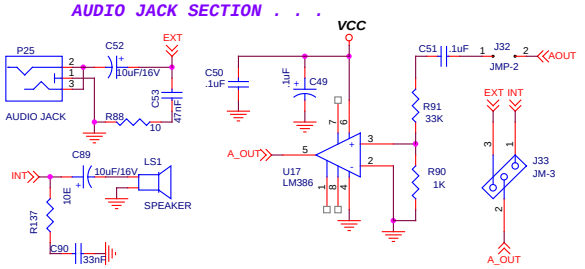
AUDIO JACK SECTION . . .



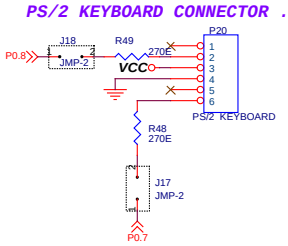
JOYSTICK SECTION . . .

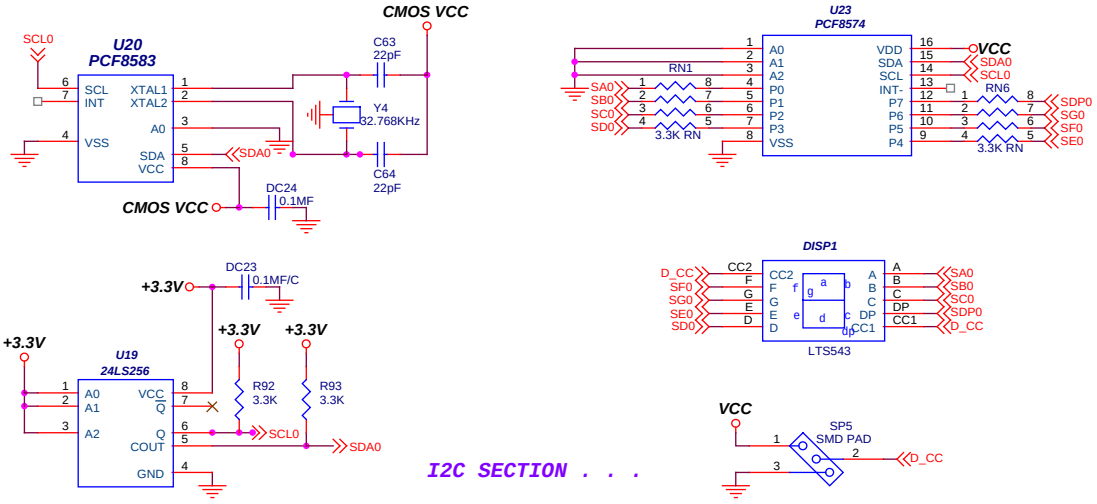


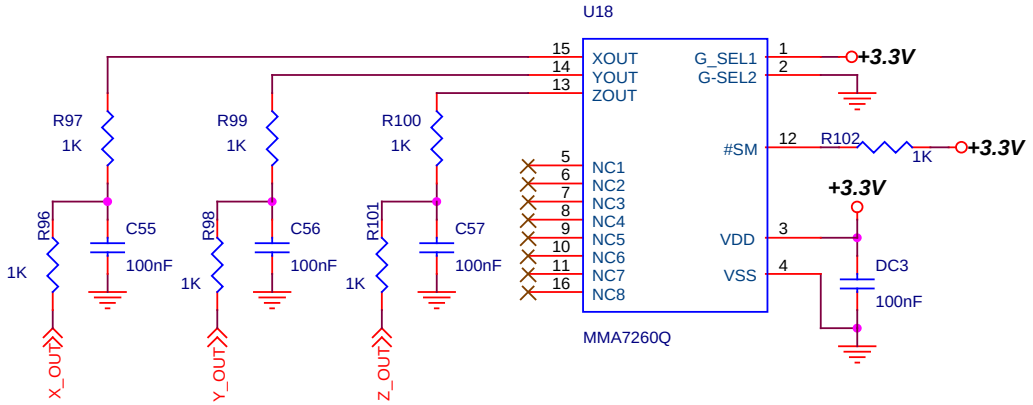
AUDIO JACK SECTION . . .



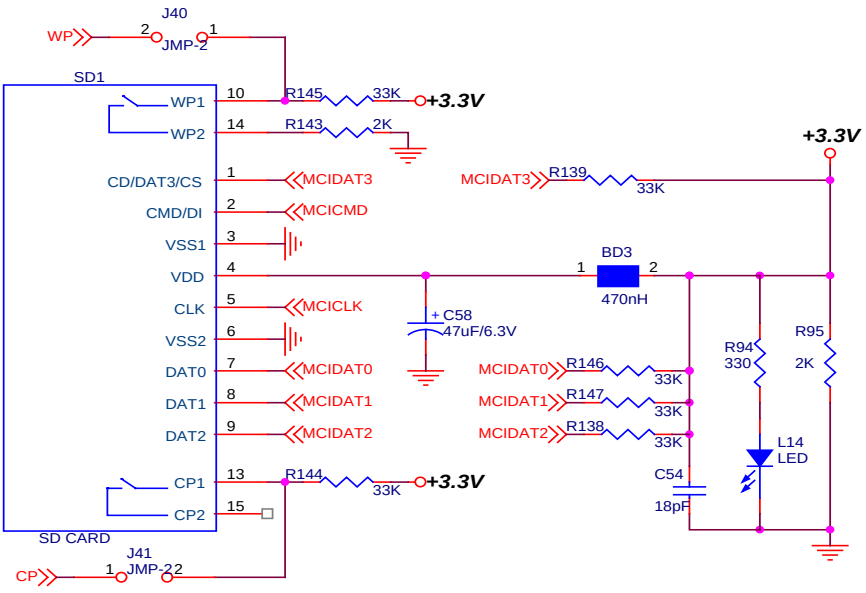
PS/2 KEYBOARD CONNECTOR . . .

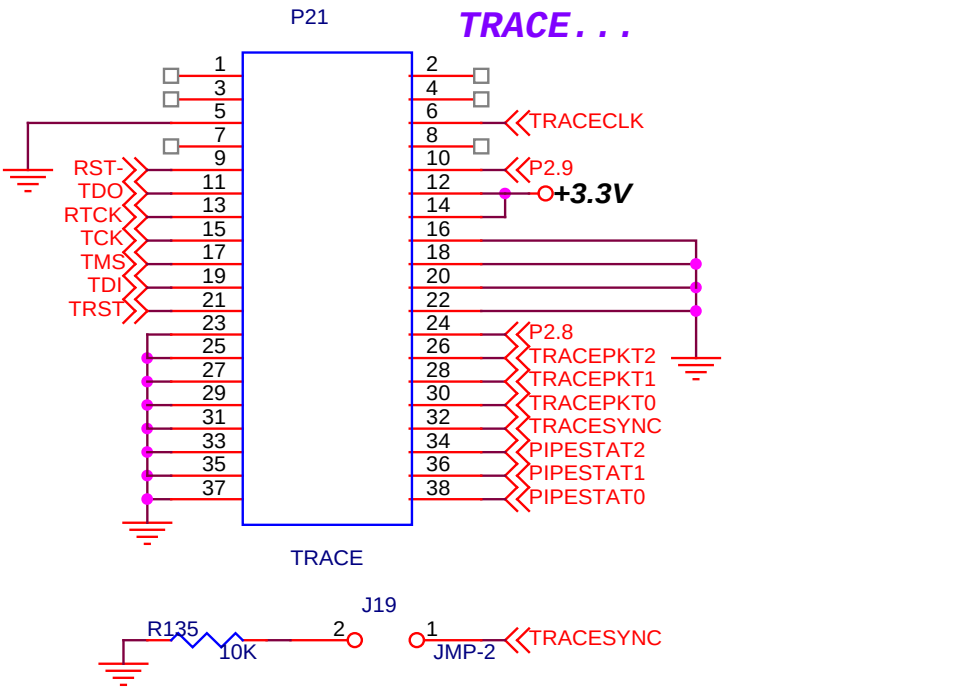


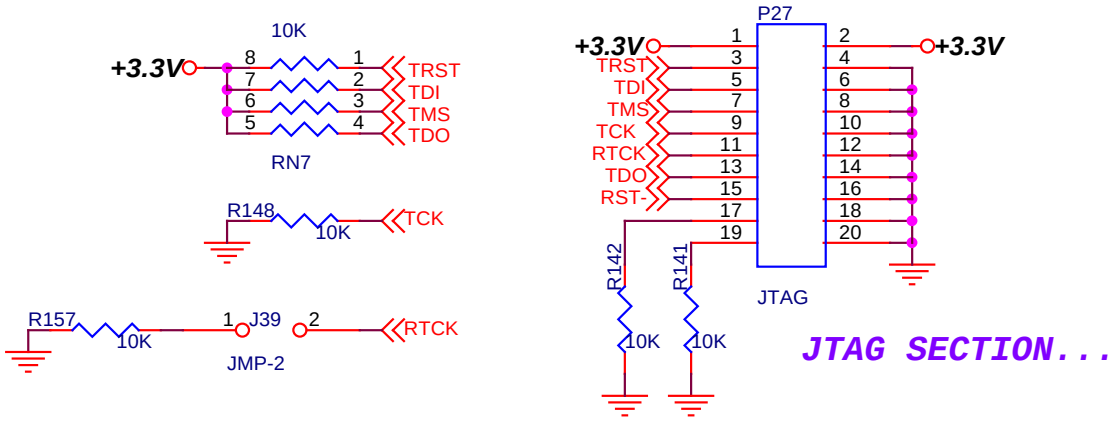


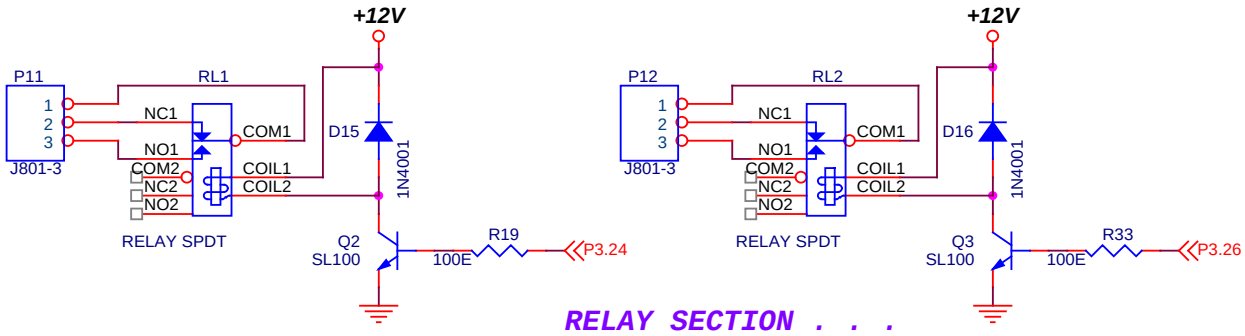


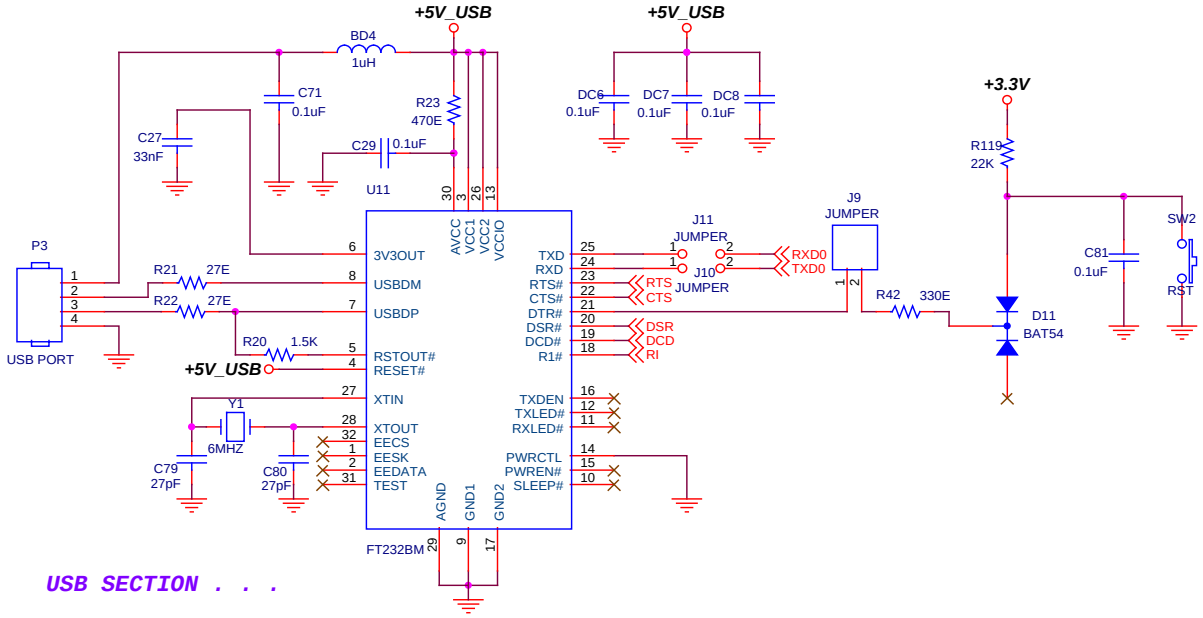
SD CARD SECTION...



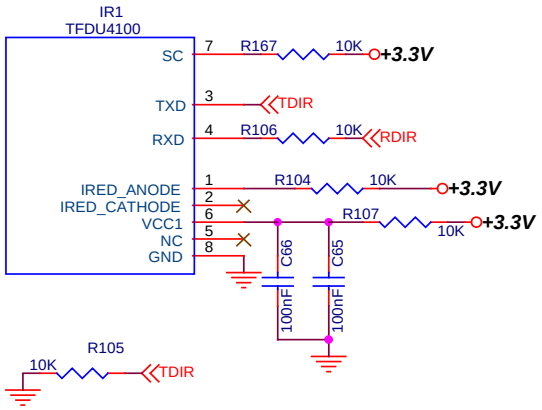

P21
TRACE...
TRACECLK
P2.9
+3.3V
RST-
TDO
RTCK
TCK
TMS
TDI
TRST
P2.8
TRACEPKT2
TRACEPKT1
TRACEPKT0
TRACESYNC
PIPESTAT2
PIPESTAT1
PIPESTAT0
TRACE
J19
R135
10K
JMP-2
TRACESYNC

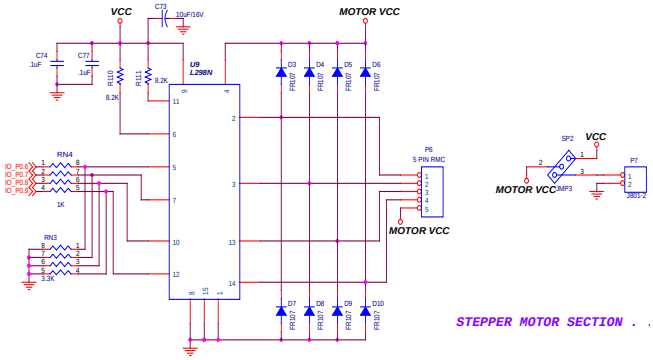

10K
+3.3V
TRST
TDI
TMS
TDO
RN7
R148
10K
TCK
R157
10K
J39
JMP-2
RTCK
P27
+3.3V
+3.3V
TRST
TDI
TMS
TCK
RTCK
TDO
RST-
R142
R141
10K
10K
JTAG
JTAG SECTION...


+12V
P11
RL1
NC1
NO1
COM2
NC2
NO2
COM1
COIL1
COIL2
J801-3
RELAY SPDT
D15
1N4001
Q2
SL100
R19
100E
P3.24
+12V
P12
RL2
NC1
NO1
COM2
NC2
NO2
COM1
COIL1
COIL2
J801-3
RELAY SPDT
D16
1N4001
Q3
SL100
R33
100E
P3.26
RELAY SECTION . . .



USB SECTION . . .

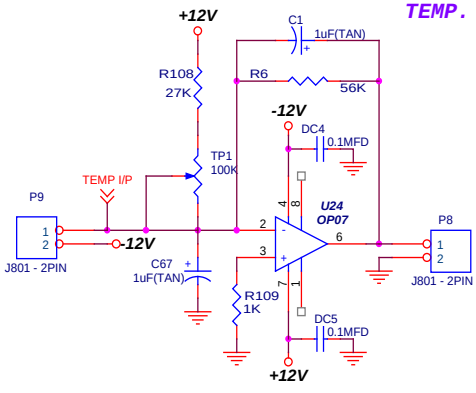


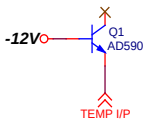


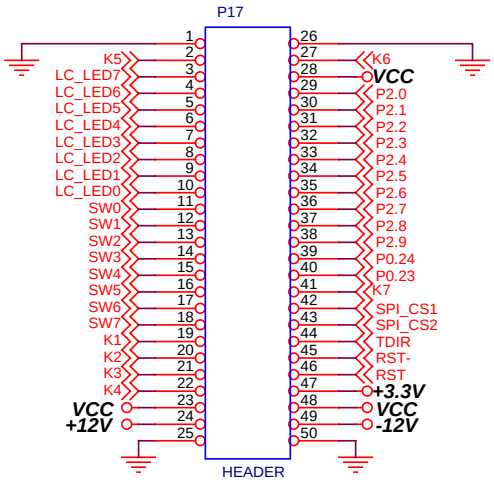
STEPPER MOTOR SECTION . .

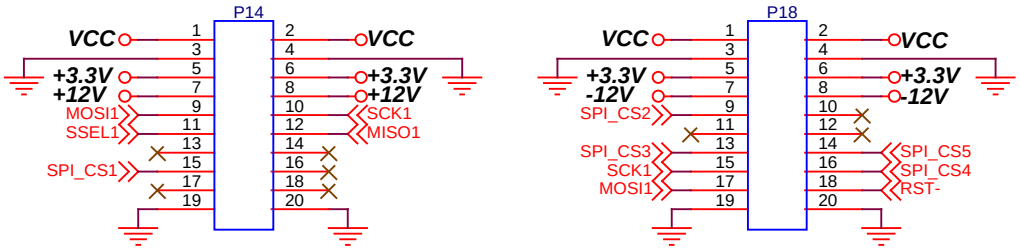
IR SECTION . . .

TEMP. SENSOR SECTION . . .



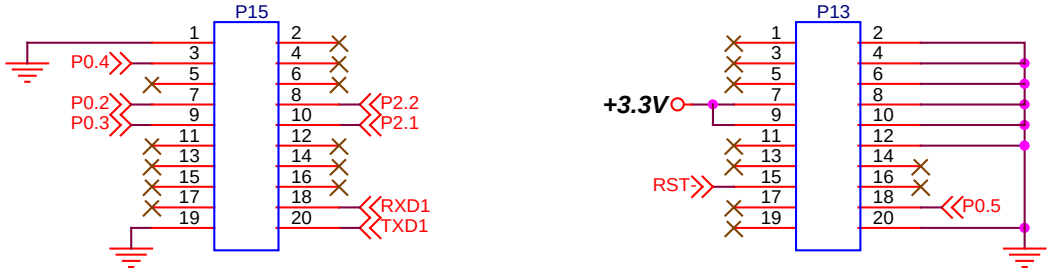






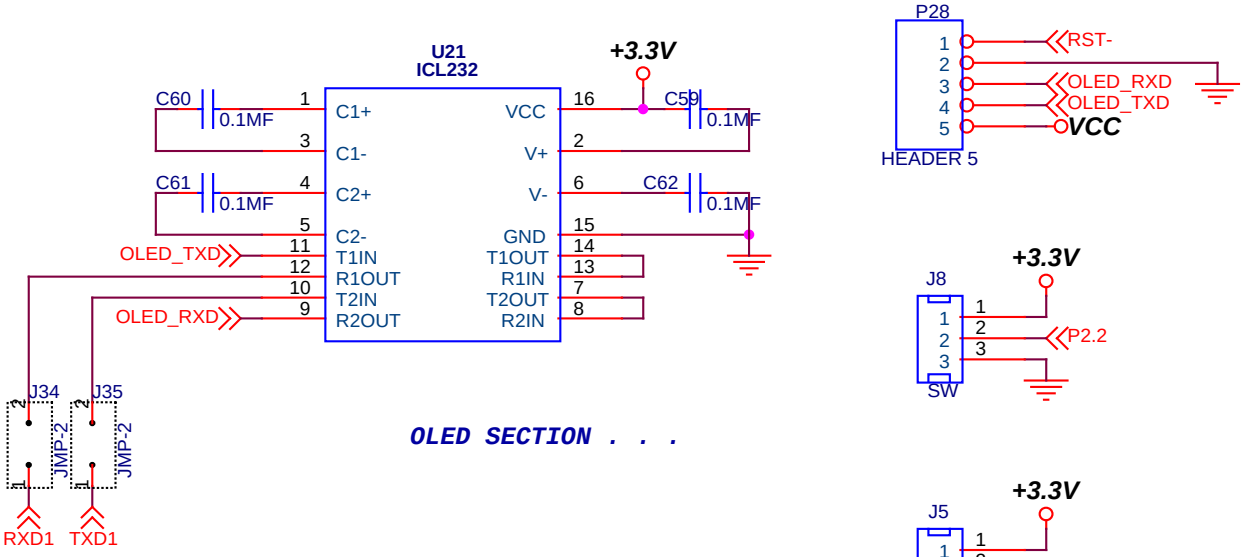
(LEFT CONNECTOR) *(RIGHT CONNECTOR)*

HIGH SPEED ADC & DAC . . .

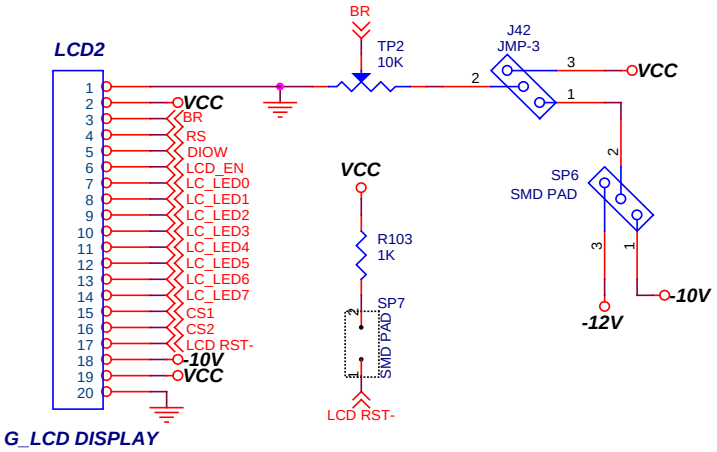


(LEFT CONNECTOR) *(RIGHT CONNECTOR)*

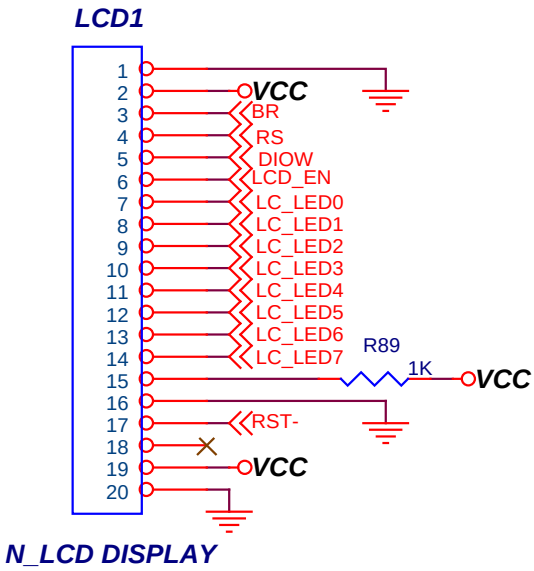
ZIG BEE SECTION . . .



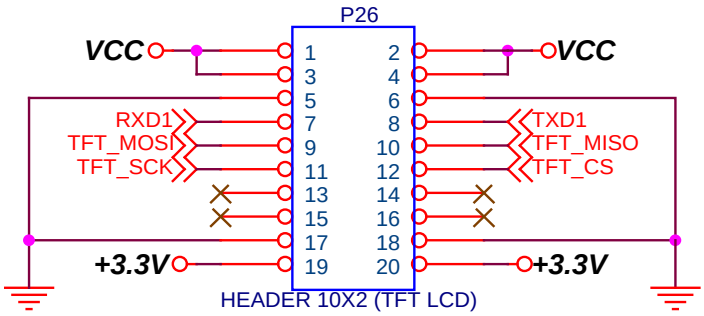
OLED SECTION . . .

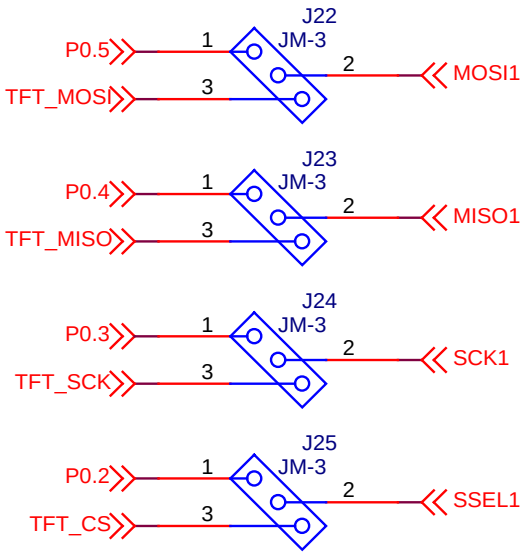


G_LCD DISPLAY



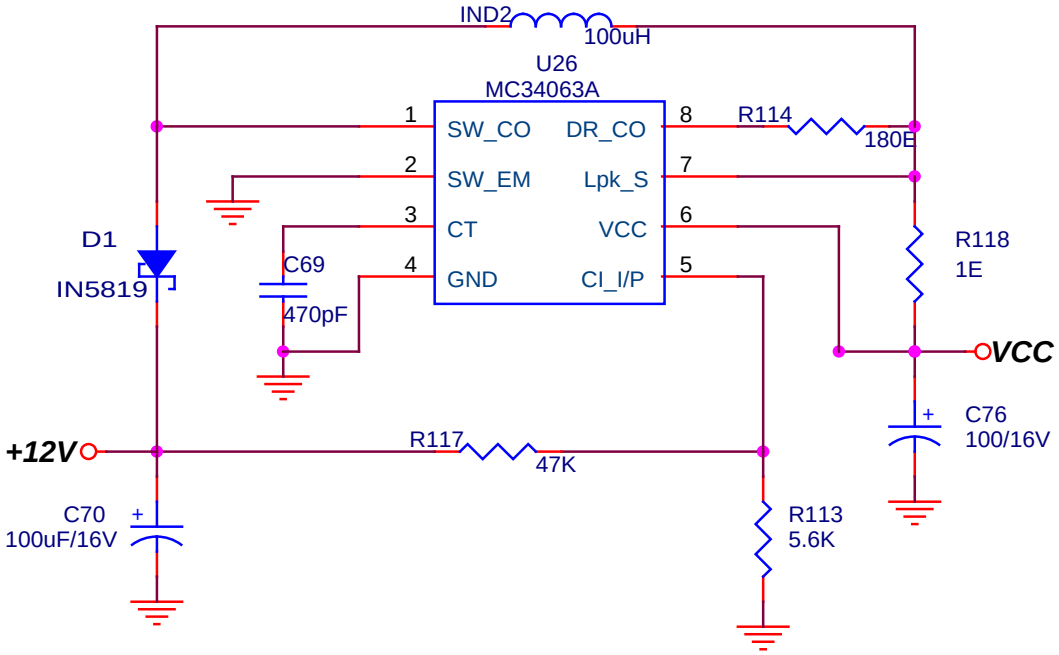
LCD DISPLAY SECTION . . .



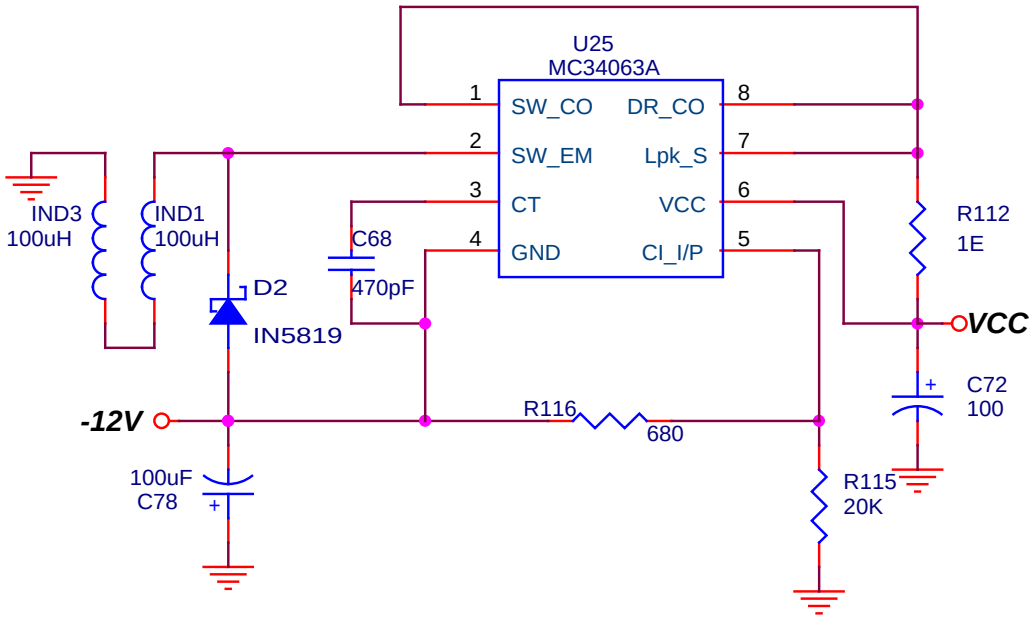


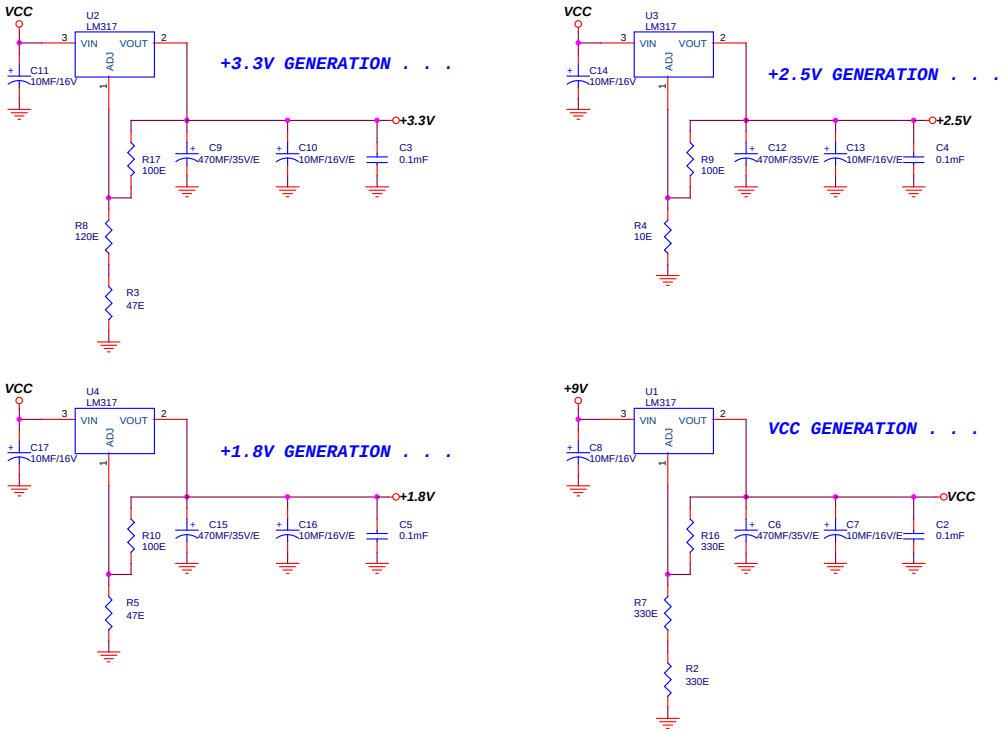
TFT CONNECTOR SECTION . . .

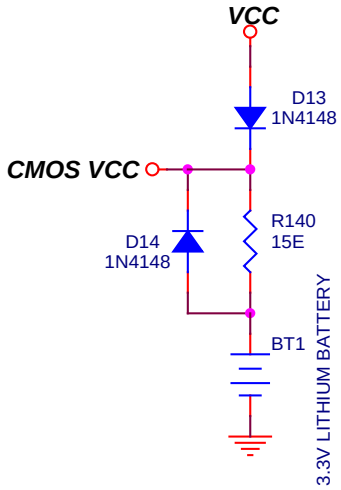
+12V GENERATION . . .

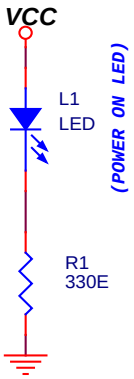


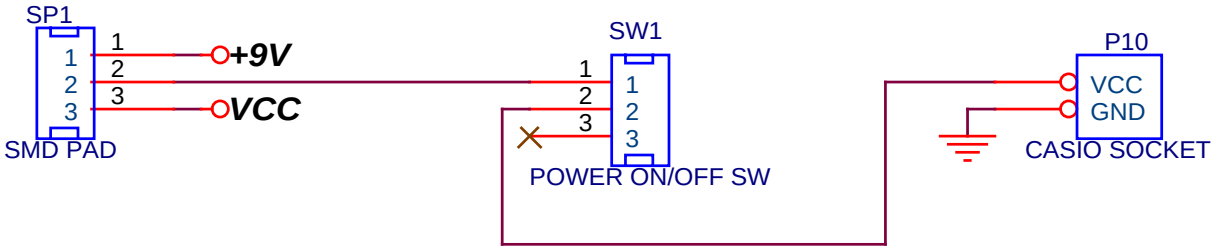
-12V GENERATION . . .




VCC
U2
LM317
VIN
VOUT
ADJ
C11
10MF/16V
+3.3V GENERATION . . .
+3.3V
R17
100E
C9
470MF/35V/E
C10
10MF/16V/E
C3
0.1mF
R8
120E
R3
47E
U3
C14
R9
C12
C13
C4
+2.5V GENERATION . . .
+2.5V
R4
10E
U4
C17
+1.8V GENERATION . . .
+1.8V
R10
C15
C16
C5
R5
+9V
U1
C8
VCC GENERATION . . .
R16
330E
C6
C7
C2
R7
330E
R2
330E

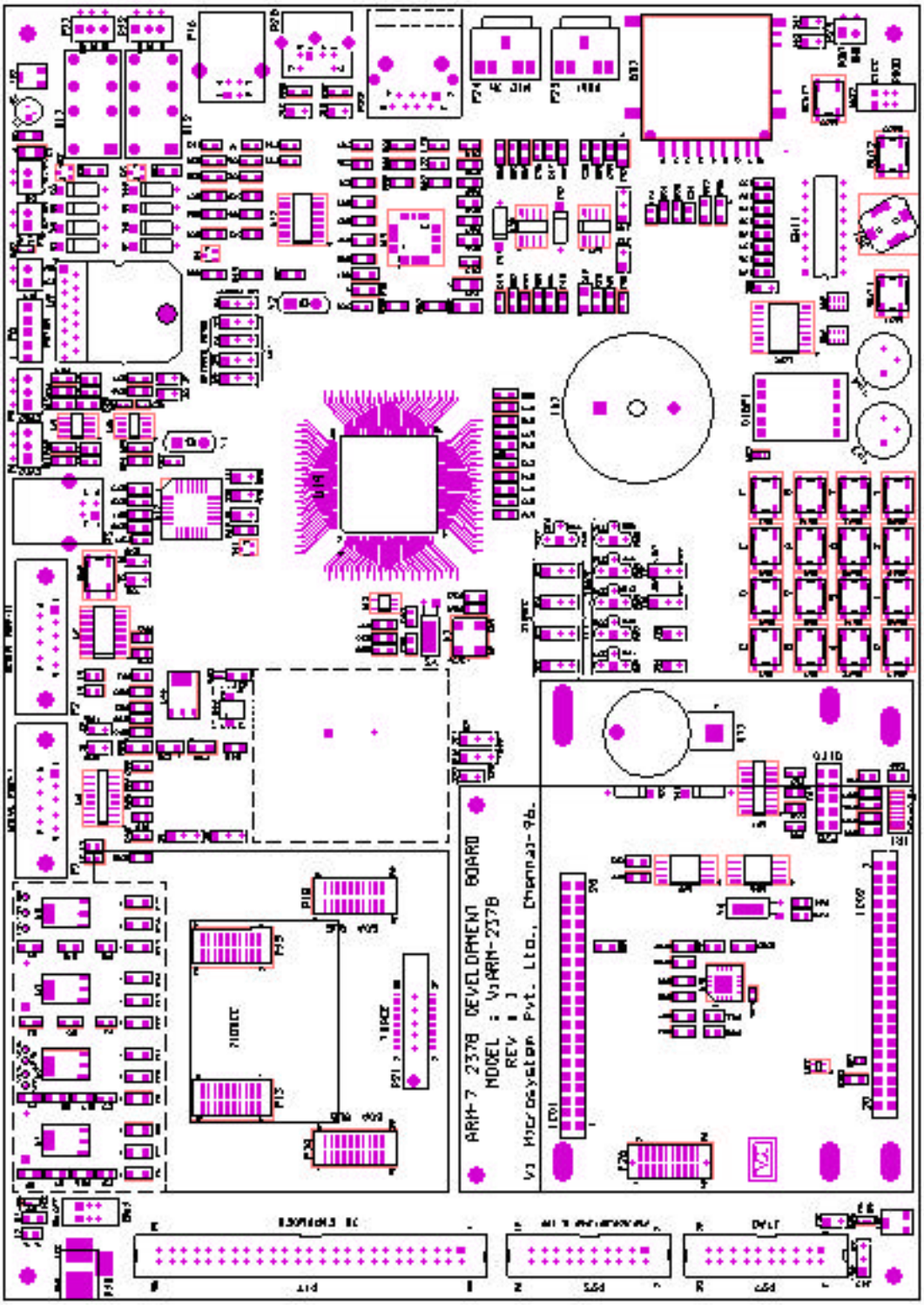

VCC
D13
1N4148
CMOS VCC
D14
1N4148
R140
15E
BT1
3.3V LITHIUM BATTERY


VCC
L1
LED
(POWER ON LED)
R1
330E


SP1
1
2
3
+9V
VCC
SMD PAD
SW1
POWER ON/OFF SW
P10
VCC
GND
CASIO SOCKET

APPENDIX – B

COMPONENT LAYOUT


ARM7 2378 DEVELOPMENT BOARD
MODEL : ViARM-2378
REV : 1
Vi MicroSystem Pvt. Ltd., Chennai-96.