

Prepared by

Rajendra Prasad Sahu  
110113070 ,ICE  
NITT  
  
Guided by  
Mr. Vivek Sanadhya, SO-H+  
DACS, RCnD, BARC, Mumbai

INTERNSHIP REPORT

Embedded programming on ATMEL AVR microcontroller

**INTRODUCTION:**

This report contains the work of Rajendra Prasad Sahu, an UG student of NIT Trichy, learnt during his internship period dated from 20th May, 2016 to 8th July, 2016 at the Distributed Automation and Control Section (DACS) of the Reactor and Control Division (RCnD), BARC, Mumbai. His guide for the training period was Mr. Vivek Sanadhya (SO-H+), the Head of the same section.

The aim of this internship session was to learn the basic framework of an embedded control system built around ATmega128 and also carry out interfacing tasks like:

1. Display Multiplexing
2. Keypad ~ LCD Interfacing
3. Port Expander Interfacing
4. LCD interfacing on SPI
5. SPI interface between a Keypad, a Master controller and a Port Expander.

**ATmega128:**The ATmega128 is a low power CMOS 8-bit microcontroller based on the AVR enhanced RISC architecture. By executing powerful instructions in a single clock cycle, the ATmega128 achieves throughputs approaching 1 MIPS per MHz allowing the system designer to optimize power consumption versus processing speed.

Some of the remarkable features of this microcontroller are:

• **Advanced RISC Architecture**

– 133 Powerful Instructions – Most Single Clock Cycle Execution

– 32 x 8 General Purpose Working Registers + Peripheral Control Registers

– Up to 16MIPS Throughput at 16MHz

– On-chip 2-cycle Multiplier

• **High Endurance Non-volatile Memory segments**

– 128Kbytes of In-System Self-programmable Flash program memory

– 4Kbytes EEPROM

– 4Kbytes Internal SRAM

– Write/Erase cycles: 10,000 Flash/100,000 EEPROM

– SPI Interface for In-System Programming

• **Peripheral Features**

– Two 8-bit Timer/Counters with Separate Prescaler and Compare Modes

– Two 16-bit Timer/Counters with Separate Prescaler, Compare and Capture Mode

– Master/Slave SPI Serial Interface

• **Special Microcontroller Features**

– Power-on Reset and Programmable Brown-out Detection

– Internal Calibrated RC Oscillator

– External and Internal Interrupt Sources

– 53 Programmable I/O Lines

The features that are going to be extensively used for this training are:

1. **General Purpose Input Output**
2. **Timers**
3. **Serial Peripheral Interface**
4. **External Interrupt**

**GPIO:**

The AVR I/O ports are the path to the outside world. This is the most frequently used module of the controller. So learning them properly is the first step.

An abused I/O port is fairly easy to burn out with excessive current or static damage. Most all the I/O ports are floating inputs that can build up large static charge

Some cautionary steps that can be advised are:

* Never carry your AVR board in a non-static dissipative bag.
* Dry fall days are perfect for creating conditions for ESD damage.

All ports have read-modify-write capability, i.e., you can change pin direction, pin value, or pin pull-up resistor without effecting any other pins in the port.

Control of all ports and pins is done with three registers:

**DDRx:** This register is used to configure a pin as an input/output. Writing 1 in a particular bit of this register configures that particular pin of x Port to output or else if reset to 0 then configured as input. The default value of this register is 0x00 which means default direction is input. As x can take 6 characters, 6 such DDRx registers exist.

**PORTx:** This register has two functions:

* + Determines the driven pin value if the pin is an output
  + Determines if a pullup is present if the pin is an input

In output mode, setting a bit to 1 means driving that pin by a digital output of 1, resetting a bit to 0 means driving that pin by a digital output of 0.  
In input mode setting it to 1 means it will be read as digital input of 1 unless an external circuit is driving that pin which will force the pin to take that value. If reset to 0 it means the pin is configured to tristate unpredicted state.

**PINx:** This is a read only register which stores the status of each pin of a port. When pin configured as input, this register can be used to read the inputs.

**TIMERS:** AVR has two 8 bit and two 16 bit counters. For lower frequency applications 16 bit counter can be used. Each timer is a counter in a basic sense which is clocked by the CPU clock. When these counters reach their maximum value i.e. 0xFF -8 bit, 0xFFFF -16 bit , an overflow flag is set and interrupt can be enabled to execute a routine. This routine becomes the action which has to be timed and the time taken by the counter to reach the max value becomes the time period. The counter counts 1 up/down for one activating clock pulse.

**Precaling:** The clock provided to the counter need not be CPU clock. It can be prescaled by dividing the CPU clock by multiples of 4 giving a lower clock to the counter.

For basic timing operations we need three basic R/W registers.

1. TCCRx  
2. TCNTx  
3.TIMSK

**Timer/Counter Control Register TCCRx:**

**• Bit 2:0 – CS02:0: Clock Select**

The three clock select bits select the clock source to be used by the Timer/Counter by prescaling it with multiples of 4.

**Timer/Counter Register TCNTx:**

The Timer/Counter Register gives direct access, both for read and write operations, to the

Timer/Counter unit 8-bit counter. Writing to the TCNT0 Register blocks (removes) the compare match on the following timer clock. Writing 0x00 to this register starts the counter.

**Timer/Counter Interrupt Mask Register TIMSK:**

When the TOIE bit is written to one, and the I-bit in the Status Register is set (one), the

Timer/Counter Overflow interrupt is enabled.

**Calculating the frequency of an 8 bit counter =** (Clock to counter/256)

Clock to counter = (CPU clock/Prescaler )

To decrease the frequency further, the no of overflows can be counted to a certain value to execute the action needed to be timed.

**Serial Peripheral Interface:**The Serial Peripheral Interface (SPI) allows high-speed synchronous data transfer between the

Atmel AVR ATmega128 and peripheral devices or between several AVR devices. The

ATmega128 SPI includes the following features:

•**Full-duplex, Three-wire Synchronous Data Transfer**

• **Master or Slave Operation**

• **LSB First or MSB First Data Transfer**

SPI uses 4 pins for communications (which is described later in this post) while the other communication protocols available on AVR use lesser number of pins like 2 or 3. Here are some of the advantages of SPI:

* Extremely easy to interface! (It took me much less time to setup and transmit data through SPI as compared to I2C and UART!)
* Full duplex communication
* Less power consumption as compared to I2C
* Higher hit rates (or [throughput](http://en.wikipedia.org/wiki/Throughput))

**Pin Description**

The SPI typically uses 4 pins for communication, wiz. MISO, MOSI, SCK, and SS.

**MISO** –. Data transfer from *Slave* to *Master* takes place through this channel. This is the PB3 pin in ATmega128.

**MOSI**– Data transfer from *Master* to *Slave* takes place through this channel. This is the PB2 pin in ATmega128.

**SCK** – This is the SPI clock line (since SPI is a synchronous communication). This is the PB2 pin in ATmega128.

**SS** – This stands for *Slave Select*. This pin would be discussed in detail later in the post. This is the PB0 pin in ATmega128.

**Register Descriptions**

**SPCR:**

**Bit7: SPIE–SPI Interrupt Enable**- Setting it enables SPI Interrupt if global interrupt s are enabled

**Bit 6: SPE – SPI Enable-** Setting it enables SPI

**Bit 5: DORD – Data Order**: Set this bit to 1 if you want to transmit LSB first, else set it to 0

**Bit 4: MSTR – Master/Slave Select**: Setting this configures the device as master.

**Bit 1,0: SPR1, SPR0 – SPI Clock Rate Select**  
These bits, along with the SPI2X bit in the SPSR register (discussed next), are used to choose the oscillator frequency divider, which prescales the F\_CPU

**SPSR:**  
The SPI Interrupt Flag is set whenever a serial transfer is complete. An interrupt is also generated if SPIE bit (bit 7 in SPCR) is enabled and global interrupts are enabled. This flag is cleared when the corresponding ISR is executed.

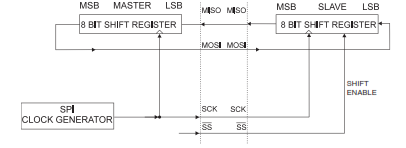
**SPDR:** The SPI Data register is an 8-bit read/write register. This is the register from where we read the incoming data, and write the data to which we want to transmit. The 7th bit is obviously, the Most Significant Bit (MSB), while the 0th bit is the Least Significant Bit (LSB).

Now we can relate it to bit 5 of SPCR – the DORD bit. When DORD is set to 1, then LSB, i.e. the 0th bit of the SPDR is transmitted first, and vice versa.

# **The Slave Select (SS’) Pin** SS’ (means SS complemented) works in active low configuration. Which means to select a particular slave, a LOW signal must be passed to it.

**When set as input, the SS’ pin should be given as HIGH (Vcc) on as Master device, and a LOW (Grounded) on a Slave device.**

When we are communicating between multiple devices working on SPI through the same bus, the SS’ pin is used to select the slave to which we want to communicate with.



**EXTERNAL INTERRUPT:**

The external interrupt sources on ATmega128 are PD1, PD0. These pins must be configured as inputs to accept interrupt request from external circuit. Now it has be decided that what kind of transition would act as an interrupt request. To decide this two bits has been reserved in EICRA and EICRB registers for each interrupt.

For External Interrupt 0 bits 1:0 has been reserved i.e. ISC01, ISC00

**AVR Studio 5.1:**

This is the IDE that will be used for this for this series of tasks.  
AVR Studio 5.1 is the new professional Integrated Development Environment (IDE) for

writing and debugging AVR applications in Windows XP/7/8/10 environments.

AVR Studio 4 supports the following development tools: ICE50, JTAGICE, ICE200,

STK500, and AVRISP. It uses the AVR GNU C Compiler.

The features may be listed as:

* Rich code editor for C/C++ and Assembly featuring the powerful Visual Assist extension
* Cycle correct simulator with advanced debug functionality
* Atmel Software Framework allowing creation of modular applications and providing building blocks for a prototype on any AVR platform
* Debugging on actual devices using Debugging Tools
* Rich SDK to enable tight integration of customer plugins
* Compatible with many Microsoft Visual Studio plugins

**In System Programming (ISP):**

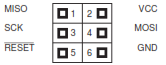
In-System Programming (ISP), also called In-Circuit Serial Programming (ICSP), is the ability of some programmable logic devices, microcontrollers, and other embedded devices to be programmed while installed in a complete system, rather than requiring the chip to be programmed prior to installing it into the system. This is the mode of programming to be used in upcoming tasks.

Using a simple Three-wire SPI interface, the In-System Programmer communicates serially with the AVR microcontroller, reprogramming all non-volatile memories on the chip.

In-System Programming eliminates the physical removal of chips from the system.

This will save time, and money, both during development in the lab, and when updating the software or parameters in the field.

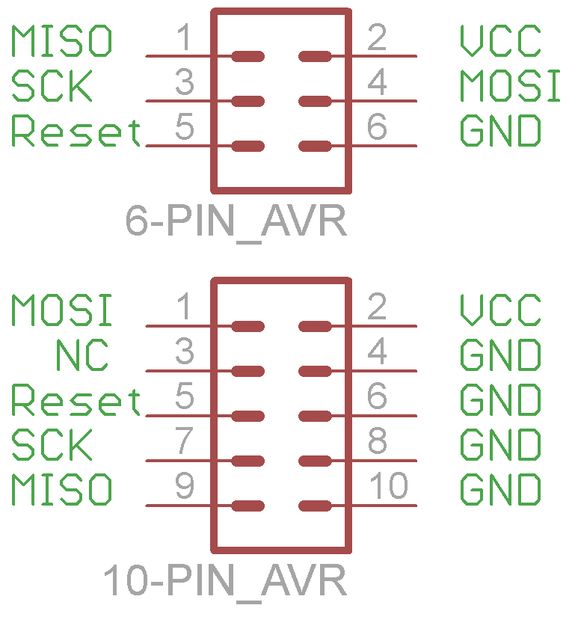
A conventional ISP port consists of 6 pins



**STK500:**

STK500 is a starter kit that can act as an ISP programmer for ATmega128. It provides with two ISP ports one of 6 pins and another one of 10 pins.

The port top views are viewed to be like this.



**Steps to successfully program an ATmega128 via ISP using STK500**

1. Build the hex of your code.
2. Connect the 6-pin or the 10-pin ISP header to the target ISP port with latched pin as the first one.
3. Switch on the STK500 and check for a green signal at the STATUS LED.
4. Add STK500 device in your AVR Studio and select the device i.e. ATmega128.
5. Apply the settings.
6. Read the device signature.
7. Burn the hex to the flash memory by pressing the “Program” tab.

**DISPLAY MULTIPLEXING:**

**Multiplexed displays** are electronic [display devices](https://en.wikipedia.org/wiki/Display_device) where the entire display is not driven at one time. Instead, sub-units of the display (typically, rows or columns for a [dot matrix](https://en.wikipedia.org/wiki/Dot_matrix) display or individual characters for a character oriented display, occasionally individual display elements) are [multiplexed](https://en.wikipedia.org/wiki/Multiplexing), that is, driven one at a time, but the electronics and the [persistence of vision](https://en.wikipedia.org/wiki/Persistence_of_vision) combine to make the viewer believe the entire display is continuously active.

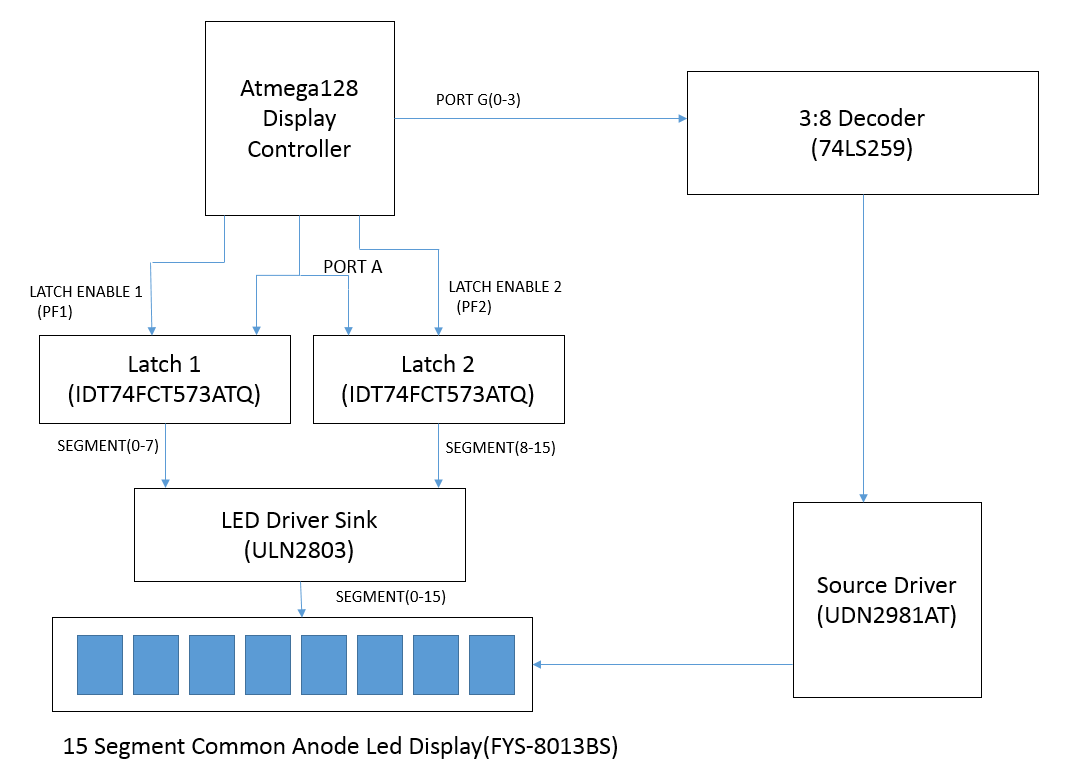
A multiplexed display has several advantages compared to a non-multiplexed display:

* fewer wires (often, far fewer wires) are needed
* simpler driving electronics can be used
* both lead to reduced cost
* reduced power consumption   
  The task is to drive a character oriented display that will display one character at one time. A 15 segment common anode LED can be used as the basic display unit. 8 such display units will comprise of the display panel to display a string of 8 alphanumeric characters.

**COMPONENTS REQUIRED:**

1. **15 Segment Common Anode Led Display:** For displaying alphanumeric characters 15 segment LED is needed. All of the 15 segments have a common anode. Placing the anode at high and cathode at low will lit up the led. As the anode is common to all of the 15 segments, it is the cathode that will decide that the LED will glow or not. Corresponding cathode combinations to display a specific character can be calculated by using any 15 segment LED datasheet. Component named FYS-8013BS LED is suitable for this purpose. 8 such LEDs will make the display panel which needs to be multiplexed.
2. **Display Controller:** The microcontroller used for this assignment is an ATmega128. This controller will provide the data segment (cathode combination) to lit up a specific alphanumeric and also the control signal (anode signal) to multiplex between the 8 LEDs.
3. **3:8 Decoder(active high outputs):** To multiplex between the 8 LEDs this decoder will be used which takes 3 select signals and decode it into 8 outputs. The 8 outputs will be driving 8 anodes of the display panel. As a decoder works only 1 of the 8 outputs will be high at a time. The 3 select signals will be provided by the microcontroller. Component named 74LS259 is suitable for this purpose.
4. **Two Data Latches:** For displaying a character 15 data signals are required but a microcontroller has a maximum of 8 pins in a port and using two data ports doesn’t seem like a good idea. It’s efficient to use 2 data latches to which the microcontroller can successively write the upper and lower 8 bits of the 15 bit data using 1 data port. Component named IDT74FCT573ATQ is suitable for this purpose. This latch one enable signal and one output enable signal. Data is successively written onto the latches by enabling and disabling them. Then a common output enable signal is generated to obtain the 15 bit data signal.
5. **Source Driver IC:** The decoder output voltage level and LED anode voltage requirements may not match. It is advisable to use an anode driver IC for the LEDs The decoder output will activate these drivers to drive 1 anode of the 8. This IC’s output voltage level is well matched with the LED anode requirement. Component named UDN2981AT is suitable for this purpose.
6. **LED Driver IC:** The microcontroller low power data signals aren’t enough to drive the high power LED segments. LED driver ULN2803 can be used to produce a high power 15 bit data input for LED. Taking the 16 bit low power input from the two latches it produces 15 bit high power output.

**DIAGRAM ILLUSTRATION:**

**PROGRAMMING FLOW:**

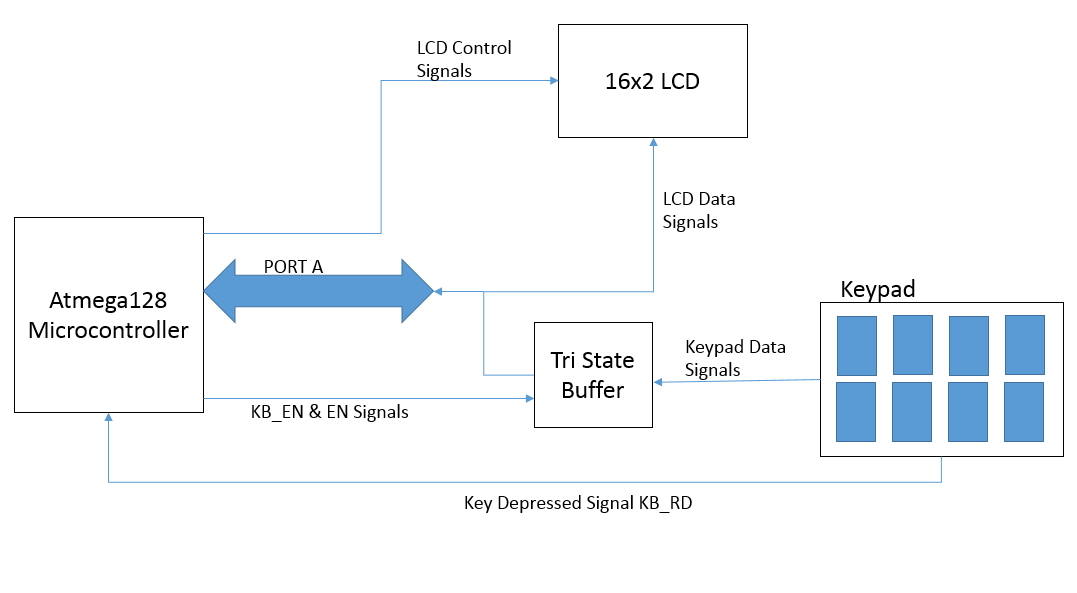
1. Initialise the select signals and data port as GPIO output pins.
2. Configure the TIMER0 to appropriate settings which will be running an Interrupt Service Routine when the timer overflows.
3. Enable the timer which would perform the following at such a rate that the discontinuous enabling and disabling of LEDs exceeds the persistence of vision.
4. Write consecutively the upper and lower 8 bits into the 2 latches by first enabling and then disabling the latch.
5. Enable the common Output enable signal to obtain the 15 bit data.
6. Now the select the required LED by sending appropriate select signals to decoder.
7. Disable the decoder.

**OBSERVATION:**On decreasing the frequency at which the ISR is executed, it is observed that display panel starts to show flickering. The minimum frequency above which no flickering is observed is found to be around 250 Hz. As there 8 LEDs, the frequency contribution for each LED can be calculated as 250/8=31.25 Hz which is almost close to persistence of vision limit i.e. 30 Hz.

**KEYPAD~LCD INTERFACING:** The Keypad and LCD are needed to be interfaced in such a way that the system responds to an input at the keypad and display the output at the LCD. The same data port will be multiplexed for sending data signals to LCD as well as receiving data signals from Keypad **COMPONENTS REQUIRED:**

1. **8 KEYS KEYPAD:** This keypad contains 8 auto release switches which will show change in input only for the instant the switch is pressed. In that instant only the microcontroller has to catch the data and take necessary action based on the input. This is possible by having a key depressed signal which will be an external interrupt to the microcontroller and will be requested only when a key has been pressed. This signal can act as a keyboard read signal which is taken at PD0. This keypad has 8 data signals which will be showing the instantaneous values at each key tapping.
2. **TRI STATE BUFFER:** As the keypad shows an instantaneous change, a buffer will be required to save the data until the controller gets hold of it. Also the same port will be getting used for LCD, so some time will be consumed to change the data direction of the port (from o/p direction for LCD to i/p direction for Keypad). Component named IDT74FCT54ICTQ is suitable for this purpose. It is a CMOS octal tri state buffer which means it will produce an output of high impedance when no input is applied to or the buffer is not enabled. So there’s no chance of data confusion between no input and low input. This buffer has two active low enable signals KB\_EN and EN signal which can be provided by PE2 and PE5 pins.
3. **16x2 LCD:** This is a normal 16X2 LCD used for regular microcontroller projects. It has a 8 bit data port. For its operation, control signals RS, R/W and EN signal.

**DIAGRAM ILLUSTRATION:**



**PROGRAMMING FLOW:**

1. Initialise PORT A as GPIO output port because in normal the program would be running to drive the LCD. Also initialize the data directions to the associated LCD control signals. Most of the control signals will be provided by PORT E.
2. Initialise the active low KB\_EN signal.
3. Initialise PD0 as an external interrupt for the key depressed signal. As a falling edge input in key depressed signal acts as an interrupt, configure the external interrupt such that it’s active only when there’s a falling edge.
4. Configure the LCD with the initialisation commands of 16x2LCD, clear, the cursor on 1st line 1st position etc.
5. Display some test message on LCD
6. Reading the data from keypad

* The first step must be to enable the buffer to obtain the keypad data. Enable and then disable the buffer using the active low KB\_EN and EN signals.
* Change the direction of PORT A to receive the keypad data using DORD A.
* Receive the data using PIN C register.
* Display the received data on LCD after clearing the LCD for a new message.
* Again display the normal state message after clearing the LCD.

**PORT** **EXPANDER:**A port expander can be any device to which one existing or on board port becomes two or more. Such expanders offer the advantage of allowing more devices of a particular port type to be utilized at the same time. A major downside is that, for example, a 3Gbit/s port might have a hub or expander installed and now be able to accommodate 6 devices, but at a maximum of 3Gbit/s throughput bandwidth divided by the said 6 devices, or by however many are plugged in and being used. For this purpose MAX7301AAX GPIO port expander will be used which has an SPI interface. This port expander can be used to drive a relay circuit which will be considered as digital outputs from port expander.

**COMPONENTS REQUIRED:**

1. **Master Controller:** This microcontroller will serve as the master device for the SPI communication to the port expander. This master will be sending configuration as well as data commands to the port expander. It will be an one way communication from master to slave (port expander). It is to be noted that ATmega128 has a 8 bit SPI buffer whereas port expander MAX7301AAX has a 16 bit SPI buffer.So the master will have to write two successive 8 bit data into the salve.
2. **Port expander:** The port expander to be used for this task is MAX7301AAX which is a   
   compact, serial-interfaced I/O expander (GPIO) peripheral that provides microprocessors with up to 28 ports. Each port is individually user configurable to either a logic input or logic output. It has got a 4 wire SPI interface. The interface has three inputs, Clock (SCLK), Chip Select (CS), and Data In (DIN), and one output, Data Out (DOUT). CS must be low to clock data into or out of the device, and DIN must be stable when sampled on the rising edge of SCLK. DOUT provides a copy of the bit that was input 15.5 clocks earlier, or upon a query it outputs internal register data, and is stable on the rising edge of SCLK.

Controlling the MAX7301 requires sending a 16-bit word. The first byte, D15 through D8, is the command address, and the second byte, D7 through D0, is the data byte. The command address is the address of the register to which we want to write data. This address can be referred in the datasheet of MAX7301 for further clarification.

1. **Relay Driver IC (ULN2803):** This IC consists of 8 NPN Darlington connected open collector transistors suitable for interfacing between low level digital circuitry such as microcontroller and higher current/ =voltage requirements of output changeover relays.

**Writing Device Registers**

The MAX7301 contains a 16-bit shift register into which DIN data are clocked on the rising edge of SCLK, when CS is low. When CS is high, transitions on SCLK have no effect. When CS goes high, the 16 bits in the Shift register are parallel loaded into a 16-bit latch. The 16 bits in the latch are then decoded and executed. The MAX7301 is written to using the following sequence:

1. Take SCLK low.
2. Take CS low. This enables the internal 16-bit shift register.
3. Clock 16 bits of data into DIN—D15 first, D0 last— observing the setup and hold times (bit D15 is low, indicating a write command).
4. Take CS high (either while SCLK is still high after clocking in the last data bit, or after taking SCL low).
5. Take SCLK low (if not already low).

**PROGRAMMING FLOW:**

1. Initialise the SPI pins i.e. MOSI, MISO, SCLK, CS with their correct data directions. Note is to be taken to keep the CS pin initially high as it is an active low pin.
2. Configure the SPCR register such as SPIE bit, MASTER bit, SCLK rate bits etc.
3. Disable the transition detection control by writing data 0x01 into command address 0x04. The program may behave abnormally if transition detection is not disabled.
4. Configure the ports wished to be used as input or output ports by writing input/output mode into command address. Refer the datasheet for further clarification.
5. As for this task the ports are configured as output, the next step must be to clock the output data values into corresponding command address.

**PRECAUTIONS:**To ensure proper transmission of data, pull down and then up the CS pin for each 16 bit data.  
As 16 bit of data can’t transmitted at once, two successive 8 bit data has to be written consecutively to the SPDR of master to fill up the SPDR of slave once.

**SPI DISPLAY:** This is a simple task of displaying a message from delivered from a master controller to a slave controller which has a LCD interface.

**COMPONENTS REQUIRED:**

1. **Master Controller:** This controller will act as the master of the SPI communication sending the messages to be displayed. An ATmega128 is taken for this purpose.
2. **Slave Controller:** This controller will act as the slave of the SPI communication receiving the messages in its SPDR continuously. This controller would also be interfaced with a LCD to display the received messages.
3. **16x2 LCD:** To display the received messages on slave controller.

**PROGRAMMING FLOW:**

Two independent source codes has to be developed and written on master as well as on slave. Master’s code will deal with starting the transmission and sending the message on the SPI. Slave’s code will deal with receiving the message and displaying it on LCD.  
**MASTER CODE:**

1. Initialise and configure the pins required for SPI master. Keep the CS pin high as it is an active low pin.
2. Load the data into SPDR.
3. Start the transmission by pulling down the CS pin.
4. Wait for the transmission to complete by continuously polling the SPSR SPIF flag.
5. When the flag is set pull up the CS pin to stop transmission.
6. Repeat the process for each character.  
   **SLAVE CODE:**
7. Initialise the pins required for SPI slave and LCD interfacing.
8. Give the LCD initialisation commands.
9. Enable the SPI Interrupt which will be requested when the serial transfer is complete.
10. Writing the ISR to receive the data from SPDR and displaying it on LCD.  
      
    **SPI INTERFACE:**

The task is to develop an interface between a keypad and a port expander such that pressing a key will display a particular pattern of LED interfaced on port expander unless another key is pressed which will show another pattern.

**COMPONENTS REQUIRED:**

**Master Controller:** This controller acts as the Master for all of the SPI communications. This controller will read the input from slave controller interfaced with the keypad and then transmit the received data to the port expander on SPI to display the corresponding pattern. For this task there will be two slaves, so the master has to have a decoder devoted for multiplexing between the slaves.

**Slave Controller:** This controller acts as the slave of the SPI communication. It’s main function is to capture the keypad data and then let the master controller read this data from the slave. Both the master and slave controllers are chosen to be ATmega128.

**Port** **Expander:** This IC is used to receive the keypad pattern from Master controller and receive the corresponding pattern of LED.

**PROGRAMMING FLOW:**

**MASTER CODE:** The master must perform a set of actions at a particular rate. As it has to show output for any change in input on the keypad within an instant. This rate’s period must not be greater than the time required to press a key that will be in the range of 100ms. The master must perform the following actions within a particular period.

1. Read the slave’s SPDR data by pulling the Slave’s CS pin down and then up.
2. Check if there’s a change in input from the keypad.

* If there’s a change in input, then assign the new input as the state.
* If there’s no change in input, then save the previous excited state.

1. Transmit the state to the port expander on SPI to print the corresponding pattern. As port expander SPDR is a 16bit register, two consecutive 8bit data i.e. command address and the data are clocked consecutively.

This set of actions has to be performed at a particular rate say 10Hz. To create this periodic execution, the timers will be used with the interrupts enabled. When the timer expires, the ISR is executed which can set a global flag. Polling this flag until it is set and then performing these actions creates the periodic fashion in which it has to be executed.

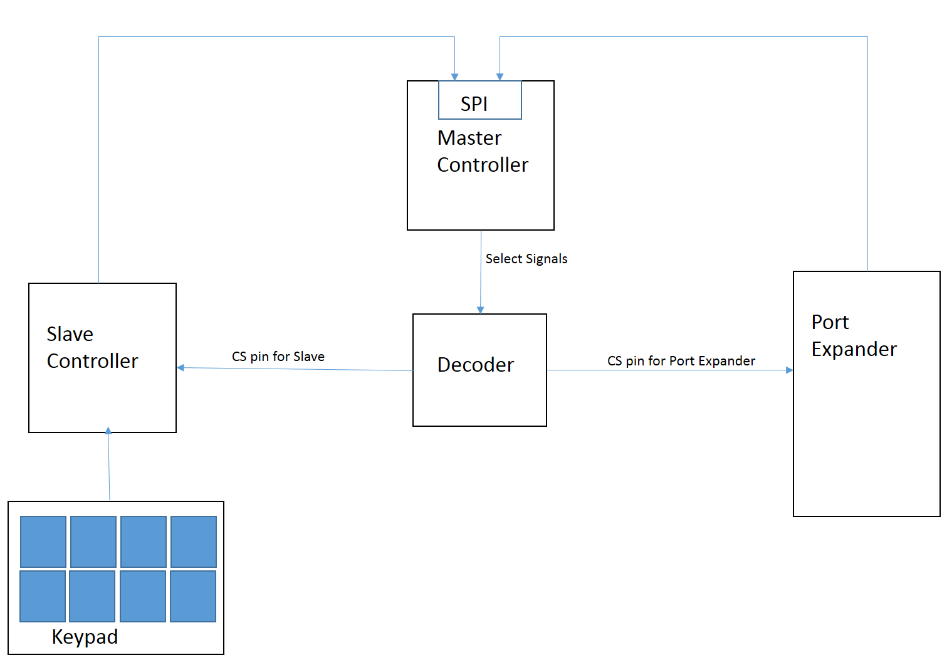
In this way the master will always be refreshing its port expander output based on the input.

**SLAVE CODE:** The slave also has to perform the following actions at a particular rate so that its SPDR register is always refreshed with the latest value of keypad.

1. Enable the tri-state buffer and change the direction of PORT C to read the keypad’s latest value.
2. Print this latest value on the LCD so that user gets to know the status of Keypad.
3. Update this value to the SPDR register.
4. Let the master clock out the SPDR data on SPI. Wait till the transmission completes.

This set of actions has to be performed at a particular rate say 10Hz. To create this periodic execution, a timer is used which when expires generates a software interrupt which shifts the program counter to an ISR to set a flag. Polling of this flag can be done to perform the actions periodically.

**DIAGRAM ILLUSTRATION:**

****

**APPLICATIONS & USES:**

This report constitutes the basic framework of an Embedded Control System. Such types of Embedded Control System find its application in industries, home automation, motion control etc. The interfacing tasks can be magnified in the following ways.

1. Display multiplexing can serve as an all-time display of a sensor such as encoder whose readings have to be displayed to find the exact location of the host body.
2. Digital outputs can be used to switch ON/OFF any external machine
3. Digital inputs like limit switches can also be received on this system.
4. Analogue readings from Sensor in the field can be interfaced using ADCs to process in the microcontroller domain. To communicate with these ADCs, SPI protocol can be used.
5. These type of Embedded systems can also provide analogue driving signals by having a DAC interfaced with it. Communication to the DAC is also be done on the SPI.
6. A mini user interface, between the keypad and the controller can be developed to execute key specific functions. The interface can allow key combinations to act as command to a greater range of functions. Key combination feature can be used to add security to the system through password protection. In this way the functionality of the system is increased.