ON-SITE PROGRAMMABLE PLC

Student – Rhitvik Kumawat | Advisor: Prof. Matthew Campisi

NEW YORK UNIVERSITY Report on an Advanced Project in Embedded Systems

Introduction

The following report entails an approach in the existential embedded paradigm that can eliminate the requirement of programming or the requirement to code/learn embedded programming languages to generate firmware architectures for non-critical embedded systems. We also discuss the benefits and disadvantages of the currently used methods that address the above cause and the abstractions introduced in the current project that eliminates the requirement of programming. Further discussed is the Instruction Set Architecture (ISA) of the ARM Cortex M-4 and how the configuration of a Mealy FSM can render the system's states reconfigurable by mimicking the abstraction of an ISA. We also delve into some associated contingencies caused because of the introduction of a novel abstraction layer in the firmware that will act pseudo kernel and how the processing of the instruction renders the functions executed by the processor core non-reentrant. Lastly, we briefly discuss the future scope and how the addition of a few hardware can render the system capable of being deployed as an out of the box controller for numerous embedded systems applications.

Embedded Systems

Modern embedded systems are ubiquitous and are ingrained in the modern societies and our daily lives. Whether it is an automation of an industry, or we can push a button in our keychain to unlock the car door, embedded systems have enabled us to interface computational technology with the outside world.

Types of embedded systems:

Based on functional requirements:

- Stand Alone Embedded Systems: Does not require host sumpter to run/control.
- Real-Time Embedded Systems: Execute tasks with fixed deadlines.
- Networked Embedded System: Access/Control/Generate network affiliated tasks.
- Mobile Embedded System: Portable and battery-operated devices.

Based on the performance of the microcontroller:

- Small Scale: Uses 8-Bit microcontrollers. E.g. soldering irons with temperature control.
- Medium scale: Uses 16-32 Bit microcontrollers. E.g. Calculators, vending machines.
- Sophisticated: For scalable & complex tasks like self-driving cars. E.g. NVIDIA jettison development board.

Characteristics of Embedded Systems:

- Reactive: Responds to input from physical environment or atleast reads it.
- Dependability: Impact on physical world, may have safety implications.
- Efficiency: Use limited resources effectively.
- Reliability: Must not fail in the field while handing tasks (especially critical tasks).

Constituents of Embedded Systems:

Basically, every embedded system can be divided into three parts:

- Power Supply: Supplies power to the system to execute task.
- Processor: Performs the computational tasks and processes the data received/stored.
- Peripherals: Assists the processor via dedicated hardware. E.g. Timers, IOs, ADC, PWM.

Microcontroller

A microcontroller is a small computational on a silicon chip along with peripherals and constitute one of the most basic entity if embedded systems. Microcontroller has a processing core(s) along with all the requisite peripherals like program memory, instruction decoders that enables it to perform computation without external hardware like traditional computational devices. Along with this, microcontrollers are also embedded with all the hardware that can allow it to interface with the external elements outside the silicon chip.

Some of the examples of such hardware are: GPIOs, Timers, Communication Buses, EEPROM etc. These peripherals, at times, also alleviates the microcontroller's processors by working independently to the CPU core and sometimes sharing the computational load. E.g. some FIFO buffers of an ADC channel have the capability of averaging the sampled data acquired from the analog voltage source. This will in turn alleviate the CPU inside the controller.

Having discussed the microcontroller briefly, lets understand a concept on which the CPU architectures are based on, the ISA. An ISA or Instruction Set Architecture is a contract or agreement between the hardware making the processor to function that guarantees a correct execution of the entered instructions provided, the instructions are compliant with the defined ISA. This is exactly what compilers do, they process the information provided by the user in a form of a code or program into a machine-readable format which complies with the ISA.

A few other terms:

Microcode is an implementation of the ISA in a processor that is physically makes the things happen in the architecture that have been implemented. Wikipedia defines as a Computer hardware technique that interposes a layer of organization between the CPU hardware and the programmer-visible instruction set architecture of the computer.

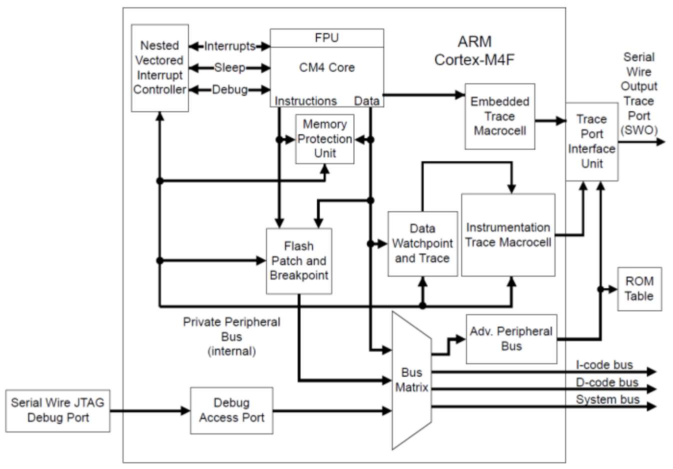
An interrupt is a reentrant function that is executed preemptively during the runtime of a program if it is invoked.

Having understood the ISA definitions and its importance, let's dive into the realm of the ARM CPU architecture and learn more about it.

ARM Cortex M-4 architecture

The Cortex-M4 processor is developed to address digital signal control markets that demand an efficient, easy-to-use blend of control and signal processing capabilities. The combination of high-efficiency signal processing functionality with the low-power, low cost and ease-of-use benefits of the Cortex-M family of processors satisfies many markets. These industries include motor control, automotive, power management, embedded audio and industrial automation markets. (Snippet taken from this website).

ARM feature a 32-Bit instruction set ISA bit in order to optimize the code density, it also has a compressed 16-Bit ISA (Thumb).


Nested Vectored Interrupt Controller
Interrupts
Sleep
Debug
FPU
CM4 Core
Instructions
Data
ARM Cortex-M4F
Memory Protection Unit
Embedded Trace Macrocell
Trace Port Interface Unit
Serial Wire Output Trace Port (SWO)
Flash Patch and Breakpoint
Data Watchpoint and Trace
Instrumentation Trace Macrocell
ROM Table
Private Peripheral Bus (internal)
Adv. Peripheral Bus
Bus Matrix
I-code bus
D-code bus
System bus
Serial Wire JTAG Debug Port
Debug Access Port

CPU Block Diagram of ARM Cortex M4F

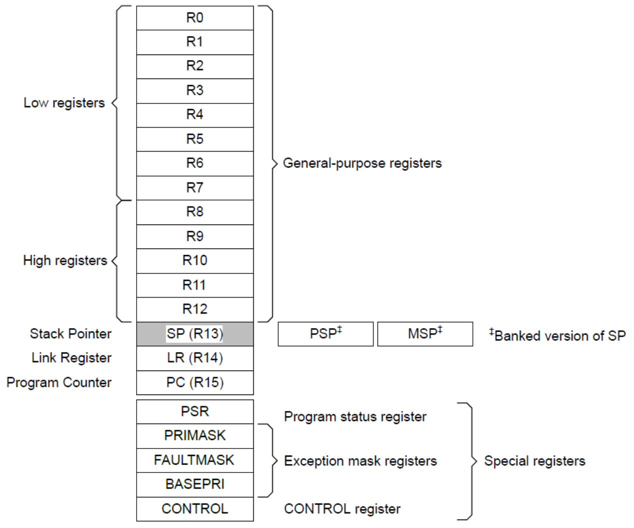
The Cortex-M4F has two modes of operation:

- Thread mode: Used to execute application software. The processor enters Thread mode when it comes out of reset.
- Handler mode: Used to handle exceptions. When the processor has finished exception processing, it returns to Thread mode.

Specifications:

Architecture	Armv7E-M
Bus Interface	3x AMBA AHB-Lite interface (Harvard bus architecture) AMBA ATB interface for CoreSight debug components
ISA Support	Thumb/Thumb-2
Pipeline	3-stage + branch speculation
DSP Extension	Single cycle 16/32-bit MAC Single cycle dual 16-bit MAC 8/16-bit SIMD arithmetic Hardware Divide (2-12 Cycles)
Floating-Point Unit	Optional single precision floating point unit → IEEE 754 compliant
Memory Protection	Optional 8 region MPU with sub regions and background region
Bit Manipulation	Integrated Bit Field Processing Instructions & Bus Level Bit Banding
Interrupts	Non-maskable Interrupt (NMI) + 1 to 240 physical interrupts
Interrupt Priority Levels	8 to 256 priority levels
Wake-up Interrupt Controller	Optional
Sleep Modes	Integrated WFI and WFE Instructions and Sleep on Exit capability, Sleep & Deep Sleep Signals, Optional Retention Mode with Arm Power Management Kit
Debug	Optional JTAG and Serial Wire Debug ports. Up to 8 Breakpoints and 4 Watchpoints
Trace	Optional Instruction Trace (ETM), Data Trace (DWT), and Instrumentation Trace (ITM)

Register Set



Cortex-M4F Register Set

R0 to R7 – General Purpose Reg. Low
R7 to R12 – General Purpose Reg. High
SP = (ASP value)? Process SP: Main SP
LR – stores the return information for subroutines, function calls & exceptions
PC - stores the current program address
PSR - has three functions, and the reg bits are assigned to different functions: • APSR, bits 31:27, bits 19:16 • EPSR, bits 26:24, 15:10 • IPSR, bits 7:0
PRIMASK - prevents activation of all exceptions with programmable priority
FAULTMASK - prevents activation of all exceptions except for the NMI
BASRPRI - defines the minimum priority for exception processing
CONTROL - controls the stack used and the privilege level for software execution when the processor is in Thread mode, and indicates whether the FPU state is active

Note: The register type shown in the register descriptions refers to type during program execution in Thread mode and Handler mode. Debug access can differ.

CMSIS

CMSIS stands for Cortex M Software Interface Standard and allows to access the registers and buffers inside the ARM Cortex-M Microcontroller via data structures.

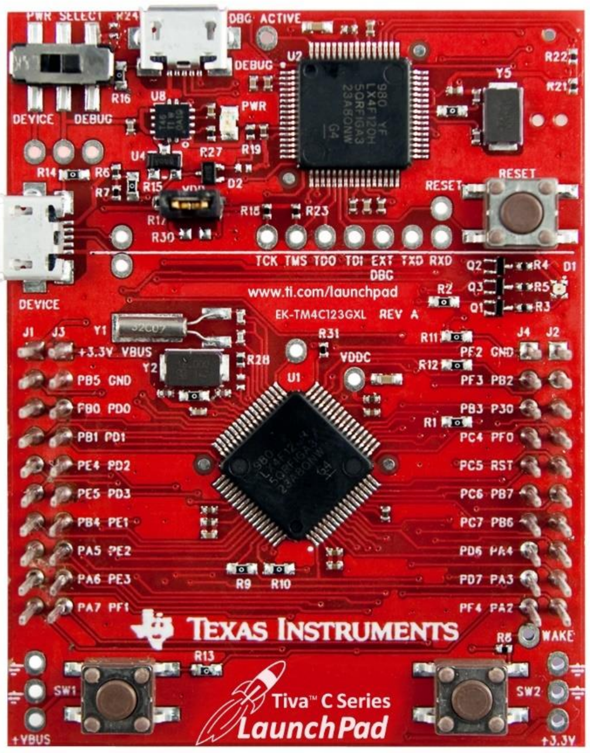
CMSIS provides interfaces to processor and peripherals, real-time operating systems, and middleware components. It includes a delivery mechanism for devices, boards, and software and enables the combination of software components from multiple vendors.

CMSIS-Core (Cortex-M) implements the basic run-time system for a Cortex-M device and gives the user access to the processor core and the device peripherals. In detail it defines:

- Hardware Abstraction Layer (HAL) for Cortex-M processor registers with standardized definitions for the SysTick, NVIC, System Control Block registers, MPU registers, FPU registers, and core access functions.
- System exception names to interface to system exceptions without having compatibility issues.
- Methods to organize header files that makes it easy to learn new Cortex-M microcontroller products and improve software portability. This includes naming conventions for device-specific interrupts.
- Methods for system initialization to be used by each MCU vendor. For example, the standardized SystemInit() function is essential for configuring the clock system of the device.
- Intrinsic functions used to generate CPU instructions that are not supported by standard C functions.
- A variable to determine the system clock frequency which simplifies the setup the SysTick timer.

CMSIS is publicly developed on GitHub.

TM4C123GH6PM



TI TM4C123GH6PM Evaluation Board

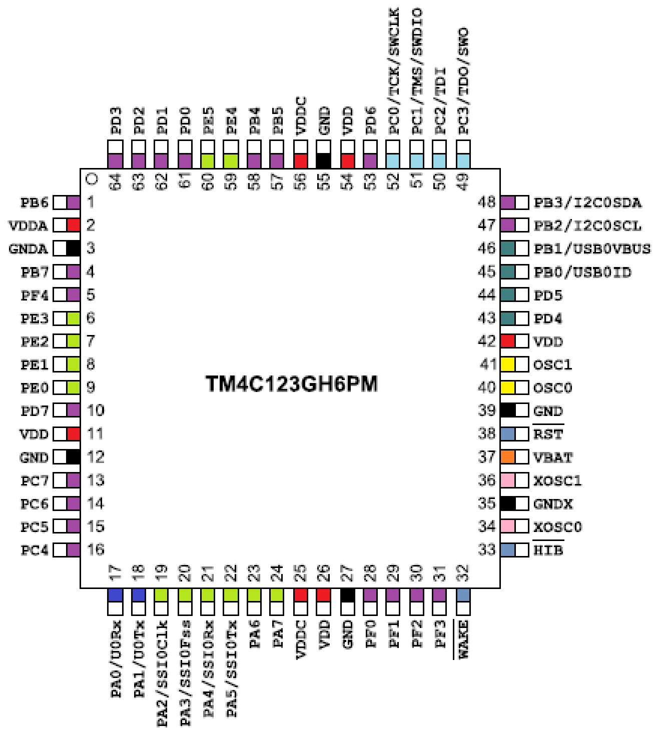
Description

The TM4C123G Launch Pad Evaluation Kit is a low-cost evaluation platform for ARM Cortex-M4F based microcontrollers from Texas Instruments. The design of the TM4C123G Launch Pad highlights the TM4C123GH6PM microcontroller with a USB 2.0 device interface and hibernation module.

Features

High Performance TM4C123GH6PM MCU:

- 80MHz 32-bit ARM Cortex-M4-based microcontrollers CPU
- 256KB Flash, 32KB SRAM, 2KB EEPROM
- Two Controller Area Network modules
- USB 2.0 Host/Device/OTG + PHY
- Dual 12-bit 2MSPS ADCs
- Motion control PWMs
- 8 UART, 6 I2C, 4 SPI



LM4F120H5QR Microcontroller Pinout

Reason for using this microcontroller is its multiple peripherals and a good amount of flash memory and SRAM. These specifications can prove themselves to be beneficial for a reconfigurable state machine implementation that will be implemented via data structures dynamically created by the user while programming his/her project. More on this in the next few sections.

TM4C123GH6PM have two LM4F120H5QR microcontrollers, one for the user and one for the debug access. Except for a few pins, all the pins are 5 Volt tolerant but to keep the system safe and work with a single power source, we will stick to the logic defined voltage, i.e. 3.3 Volts. This does introduce an added complexity in the project associated with the LCD Display that usually works on 5 Volts, but a small addition of the reverse charge pump can be implemented to mitigate the problem.

Also, to keep the HMI (Human Machine Interface) to a minimal, 2 push buttons that pre present on the Tiva-C board are used in addition to two additional GPIOs that enable to poll a rotary encoder. One of the push buttons provided on the board can be programmed as a GPIO as well as a non-maskable interrupt (NMI), giving it the ability to execute a preprogrammed structure in a preemptive way.

Hardware Abstraction Layer

Before we continue with the system design of the project, there is an additional concept we need to keep in mind. It is idea of abstraction. Hardware abstraction basically allows the programmers to write device independent applications that can be interfaced to the hardware with the interpretation of the compiler that the programmer does not need to know. This means that now a concept of higher-level language can be introduced that can allow a programmer to work on the higher levels of the program instead of delving into the gory details of the specific hardware the user is working with.

With the introduction of HAL, one does not need to understand and remember the register aggresses of the microcontroller but can very easily access a specific register of any microcontroller with the help of the datasheet and the macro definitions header file that can be included in the project and the specific memory location can be programmed with a more easy to understandable keyword. Not only this makes the code easy to read and understand, the developed code is more portable and less susceptible to mistakes.

Another example of observing the HAL is Arduino. One of the most successful open source hardware development platforms that enables everyone to code microcontroller by introducing a HAL of its own, burned into the bootloader that the developer programs using the IDE provided. Given this platform does carry a memory overhead and certain constraints exercised in the bootloader does force the developer to relinquish control over certain parts but this does allow a plethora of developers to turn their ideas into reality by just using the libraries provided by the vendor. The tradeoffs associated does impact a certain percentage of developers, but this introduced abstraction allows a lot of people to step into the paradigm of embedded system that is generally considered arcane.

Similarly, what I have tried to do in this project is to eliminate the requirement of programming all together and enable people to develop/prototype firmware by making a unified firmware that can enable the user to configure programmable states and behavior. The way it is achieved is by simulating the of an instruction execution in an ISA of an architecture in the firmware and store the functions set by the user in a defined structure. This structure does not store the functions itself but a certain piece of information that will later be matched to a LUT (look up table) programmed in the microcontroller as a function. Since there is a considerable amount of hardware both performance and memory wise, the algorithms defined by the user can be considerably long in size. In other words, most of the users will be able to successfully prototype their respective application provided that their hardware requirements do not surpass the specifications setup by the firmware. Not only this will allow the users to prototype at fast rates, but the prototyping rates will be further decreased (discussed in the later sections).

Project Specifications

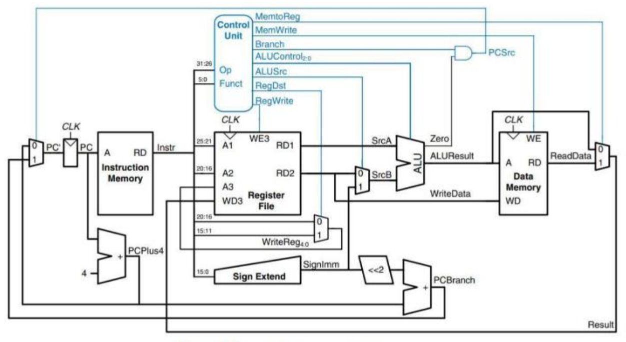
The developed firmware features the following specifications:

- 4 General Purpose Input Output (GPIO)
- 2 Timers (16-Bit Wide)
- 2 Pulse Width Modulation (PWM)
- 1 Non Maskable Interrupt (NMI) Pin
- 1 UART
- 1 Twin Wire Interface (I2C/TWI) Port
- 1 Serial Peripheral Interface (SPI) Bus
- 1 Controlled Area Network (CAN) Bus
- 1 Internal Temperature Sense Read

The reason that the hardware is limited to such a limited scope is because the structure definitions by the system will cause a memory allocation overhead and concurrent/stacked function definition can render the flow of the application code to corrupt or the stack to overflow. Especially when the concept is in nebulous phase and reliability is preferred over the features, it will be a better idea to take this approach.

System Design

To understand the system design of this on-site programmable PLC, Let's take a step back to understand the concept the way an instruction is executed in an ISA (here taken a single cycle MIPS ISA) and from there, we will try to relate how a structure definition can mimic as a fetched instruction and how we can program it.



A single cycle MIPS ISA

Components Definition

- Program counter (PC) register: This is a 32-Bit register that contains the address of the next instruction to be executed by the processor.
- Decode Unit: This block takes as input some or all the 32 bits of the instruction and computes the proper control signals to be utilized for other blocks. These signals are generated based on the type and the content of the instruction being executed.
- Register File: This block contains 32 32-bit registers. The register file supports two independent register reads and one register write in one clock cycle. 5 bits are used to address the register file.

- ALU: This block performs operations such as addition, subtraction, comparison, etc. It uses the control signals generated by the Decode Unit, as well as the data from the registers or from the instruction directly. It computes data that can be written into one of the registers (including PC). You will implement this block by referring to the instruction set.
- Instruction and Data Memory: The instruction memory is initialized to contain the program to be executed. The data memory stores the data and is accessed using load word and store word instructions.

How it works?

The instruction fetched by the machine contains the information about the current instruction. This will be decoded by the instruction decoder and will feed the information in the control unit and register file. The control unit will determine what kind of instruction is fetched, what and how the data will be processed. We must keep in mind the all of this is happening at the hardware level and the instruction fetched from the instruction memory will determine the state of the machine that is processing the information and acting according to a defined contract, the ISA.

How the PLC's system works?

Having understood how an instruction is processed in an ISA, we can conceptualize an architecture defined inside a firmware that have already been deployed and running on the microcontroller that can utilize a state machine mechanism to take instruction from the user and store them in the structure that will act as the instruction memory for our firmware introduced abstraction.

The functions that the user can set will be in an agreement with the functions written in the coded state machine. The user need not to worry about this as the navigation menu will not allow the user to step out of bounds of the architecture defined. A limited number of peripherals that will be defined in the firmware to ensure that there are no conflicts or risks involved with overflow of stack generated during the runtime of the application.

The way this works is as follows:

The LCD display will ask the number of peripherals the user wants to use.

The user can rotate the rotary encoder switch and program the peripherals he/she wants to use in his/her application on sun time. For the purpose of demonstration, we will use all the GPIOs and all the communication as well as NMI pin.

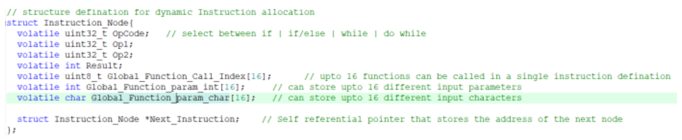
Now once the system stores all the user defined peripheral definitions in a global variable, it will automatically direct the navigation of program to every peripheral in a sequence that is like any other microcontroller routine. It will first ask to setup the initial settings of the peripherals which can include the pin mode of operation, PWM frequency, initial duty cycle, ADC settings, timer setting etc. But again, it will impose some restrictions on the user because of the limited hardware. Some of the constraints being: if GPIO is defined as an input then they will be setup in a pull-up configuration and this setting cannot be changed by the user. Similarly, the Timers defined will be defaulted to 16-bits and the user will not have the privilege to access or change this setting to 32-Bit mode of operation. The serial communication ports also have a very limited flexibility because we are still in a very early Development stage of this project. The later updates will allow the user to select the settings in the respective serial communication ports, baud, endianness and the clock polarity.

Once the setup is complete, we can start programming the runtime setup that will run continuously like the void loop () function in an Arduino. Now, let us investigate how the structure definitions inside the firmware will enable us to make dynamically programmable runtime application.

Structure Definitions

The structure definition is defined by storing the user defined setting in the structure that will be a node in a linked list. Every node will have the ability to store 2 operands, and one result in a 32-bit register. This node will also have a register that will store the opcode for an arithmetic/logical/shifting operation as per the user's will while defining it in the algorithm that later will be executed by passing the said opcode into a global function.

Global_Function_Call_Index is a 16-Bit array that can store the information regarding what function to call during the execution of the current node. These 16 functions will be executed sequentially, and the user can enter the number to set the executions as per his will. Also, if the user does not enter 16 functions then they will be skipped. In case the user wants to use more than 16 functions, he/she can use the next node to program accordingly. The Global_Function_param_int contains the integer values that will be passed to/from the global function and the Global_Function_param_char will store the character values that will be passed to/from the global function. The last node will be a self-referential node that will point to the next node that will be defined.



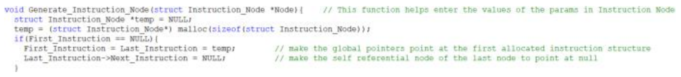
```
// structure defination for dynamic Instruction allocation
struct Instruction_Node{
  volatile uint32_t OpCode;   // select between if | if/else | while | do while
  volatile uint32_t Op1;
  volatile uint32_t Op2;
  volatile int Result;
  volatile uint8_t Global_Function_Call_Index[16];      // upto 16 functions can be called in a single instruction defination
  volatile int Global_Function_param_int[16];     // can store upto 16 different input parameters
  volatile char Global_Function_param_char[16];   // can store upto 16 different input characters

  struct Instruction_Node *Next_Instruction;    // Self referential pointer that stores the address of the next node
};
```

<u>*Sequencing / Queuing the functions*</u>

There are 3 Global Instruction nodes defined in the ral.c file named *First_Instruction, *Last_Instruction and *Y_Node all initialized to null value.

The pointer to the node to first instruction will always point to the address from where the Linked List starts and the pointer to the node to the last instruction will always point to the current instruction. The pointer to the Y_Node will be asked to point to the node from where the user want to initialize the void loop () sequence in his/her application and will become the starting point for the circular linked list.



```
void Generate_Instruction_Node(struct Instruction_Node *Node){    // This function helps enter the values of the params in Instruction Node
  struct Instruction_Node *temp = NULL;
  temp = (struct Instruction_Node*) malloc(sizeof(struct Instruction_Node));
  if(First_Instruction == NULL){
    First_Instruction = Last_Instruction = temp;        // make the global pointers point at the first allocated instruction structure
    Last_Instruction->Next_Instruction = NULL;          // make the self referential node of the last node to point at null
  }
```

<u>*Generating Instructions via DMA*</u>

The first instruction will be defined just as soon the user starts to program the application and a function will allocate it a memory in the heap. The user will enter the values / define the functions in the application and the structures will automatically set the values in the as per the settings selected by the user. When the user is finally done with the application setup, the self-referential node of the last node will be programmed to point at the Y_Node. Now the application will run as per the settings entered during the setup procedure.

Let's learn more about the peripheral definitions and the subsystems that are associated with out project.

DMA Contingencies

We need to keep in mind that since the program is allocating dynamic memory in its runtime there is a risk associated with the memory leakage and heap overflow. So, in order to mitigate some of the risks pertaining to these issues, the microcontroller program will keep notify the user that the heap is now full. At this point the user will need to restructure the application but for the purposes of the current project and keeping in mind the consumer sectors in mind, we need not to worry about it now.

The main issue however be caused by the stack overflow and to deter that, the firmware has been programmed in such a way that there are only non-recursive calls that the user can make. However, the user can mimic the recursive call function by implementing the states on his/her application but at this method will again result in an abstraction of the stack and the real stack formed via the instruction node will be limited to the current node only.

Having said this, it is now apparent to the observer that the entire system is a single routing taking different input values from the instruction node. This means that there are a lot of scope of improvements in this function and this function is not that one should deploy as a system critical controller. However, given the nebulous state of this project there does lies some scope in the future versions that can introduce the possibilities of having these systems handle critical workloads.

Protection Features (SW/HW)

Having tested this proof of concept on the breadboard there is lot of scope to add hardware that can make this setup as a out of the box assembly and can be driven by a single power source (although we can always have multiple power systems inside the design that will be fed through the said single power source). Also, specific library can be written and introduced as a feature of this firmware that can utilize the encryption accelerators provided by the microcontroller itself for securing sensitive information in the EEPROM or transmitting it via serial communication protocols.

The memory limiting features that can access the size of the programmed application and limit the further decelerations of the instruction nodes provides a certain level of reliability.

Talking about the logic shifting methods, the system operates does not tolerate voltages exceeding 3.3 Volts (except a few pins). Proper optocoupled / voltage sifting circuitry will enable the setup to read/write logics at 5 volts. An addition of other protection features will make it a true out of the box, plug and play microcontroller alternative that does not require the user to program it via a programming language.

Future Aspects

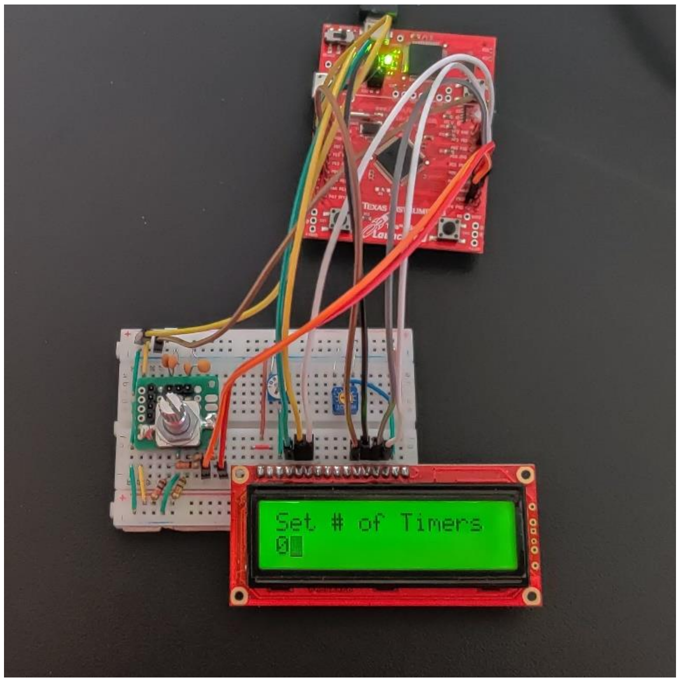
Having worked on this concept, I look forward to open source this project and with a collaborative effort the open source hardware community this project if executed correctly have the potential to be the next big thing in the DIY industry.

The stress testing of the firmware is to be completed jus as soon, I will have the access to the electric workbenches in the college to study the behavior and the wave quality of the signals the microcontroller can produce. I still believe that the structure sued have a good scope of improvement that will further optimize the entire runtime behavior of the machine and memory allocation.

Along with this I also look forward to add default application in the firmware that can enable the user to control certain devices like IR sensors, Hall Sensors, Servo motors and Stepper motors directly, hence making them run with a further optimized scripts while providing the user with ease of setting up his/her application code.

In case we need to expand the features of such a system, porting the firmware will be a very easy task as the entire system is compliant with the CMSIS standard.

For a faster processor execution, the PLL (Phase Locked Loops) can be activated which can push the processor frequency inside the microcontroller from 16 MHz to 80 MHz, but it’s not done keeping in mind a default breadboard based prototype because the 16 MHz oscillations were obtained from full rail to rail swing mode via crystal oscillator. This means that these systems will have higher tolerance to the noisy environment. But this feature can again be enabled out of the box once we have the added circuitry that can make the system properly isolated from the noise sources.



Developed prototyped on Breadboard