A PROJECT REPORT ON

Developing Library to interface DHT11 with HiFive1 Board

SUBMITTED TO

C-DAC HYDERABAD

IN PARTIAL FULFILLMENT OF THE REQUIREMENT FOR THE AWARD OF

Submitted By:

P Vinay Kumar	180850330054
Polisetty Tejaswi	180850330058
Pothuri Sai Sowmya Sree	180850330060
Pragya Pachouri	180850330061
Prashant Patil	180850330062

CENTRE FOR DEVELOPMENT OF ADVANCED COMPUTING ,HYDERABAD

https://cdac.in/index.aspx?id=hyd

CERTIFICATE

This is to certify that this is a bonafide record of project work "**Developing Library to interface DHT11 with HiFive1 Board"** done by **180850330054, 180850330058, 180850330060, 180850330061, 180850330062** under the guidance of**Mr.SunnyBijwada**in partial fulfillment of the requirement of **Diploma In Embedded System Design** at C-DAC Hyderabad for academic session of August 2018.

PROJECT GUIDE

Mr. Sunny B

(Project Engineer)

ACKNOWLEDGEMENT

The satisfaction and euphoria that accompany the success of any task would be incomplete without the mention of the people who made it possible, whose constant guidance and encouragement crowned my effort with success.

We would like to thank our mentor **Mr. Sunny B** for identifying our area of work, reviewing it at every stage and for his patient valuable hours serving as our project guide. His continuous valuable guidance and encouragement made us to complete this project.

We extend our thanks to the entire faculty of CDAC (DESD) Hyderabad who have encouraged us throughout this project.

P Vinay Kumar180850330058
Polisetty Tejaswi180850330059
Pothuri Sai Sowmya Sree 180850330060
Pragya Pachouri180850330061
Prashant Patil180850330062

ABSTRACT

Ever wondered the wonders, a renaissance architecture could do? It's definitely a yes, when its RISC-V .Here we are with a testament that elaborates how this 32-bit ,Arduino –compatible ,efficient ISA(Instruction Set Architecture)enabled us exploring the highly in-use sensor DHT-11 which is used for measuring both Humidity & Temperature.RISC-V, a new open source ISA based architecture is rapidly gaining acceptance in embedded space. Several core packages e.g. gcctoolchain, linux kernel, binutils, newlib, qemu has already been ported for it.

The fact that the architecture has less number of libraries, facilitated the idea of developing one such user-defined library to interface the RISCV –Board (HiFive1) with DHT11, using standard C concepts.

Our project mainly focuses on using FreeSDK (GCC 6.1.0 toolchain) that provides everything required compiling, customizing, and debugging C and/or RISC-V assembly programs, RISC-V enabled GDB and OpenOCDetc for dumping one's own software and run it over the board. Here, based on the deep analysis done on how the DHT11 communicates with the host ,we have tried implementing this algorithm in terms of C and developed a GCC compliant library for DHT11,which can further be used by anyone who needs it in future . The RISC-V ISA is free and open-source and can be used royalty-free for any purpose, permitting anyone to design, manufacture and sell RISC-V chips and software.While not the first open-architecture[1] ISA, it is significant because it is designed to be useful in a wide range of devices. The instruction set also has a substantial body of supporting software, which avoids a usual weakness of new instruction sets.

Developed by University Of California and many other collaborations, RISCV has an upstream support for NewLib, and other Operating Systems like Fedora and RTOS based OS.

For testing the output of the project, the source code is being dumped using the given toolchain, the sensor senses the values from the environment in 40 data bits and sends it to the microcontroller after getting an acknowledgement from it.

Now why Risc-V ?The question is obvious .Though the processor is still in its infancy its the only one supporting completely open source instruction set..This chip proves been more efficient as it saves both time and money and is being used for both industrial and customer–level benefits,and no doubt working with it feels great!!!

TABLE OF CONTENTS

1. LITERATURE SURVEY

- Riscv on its full swing -Embedded Linux on RISC-V Architecture -- Status Report By Khem Raj ,in Embedded Linux Conference,Europe

Most Linux users have heard about the open source RISC-V ISA and its potential to challenge proprietary Arm and Intel architectures. Most are probably aware that some RISC-V based CPUs, such as SiFive's 64-bit Freedom U540 found on its HiFive Unleashed board, are designed to run Linux. What may come as a surprise, however, is how quickly Linux support for RISC-V is evolving.

In his ELC presentation, called "Embedded Linux on RISC-V Architecture -- Status Report," Raj, who is an active contributor to RISC-V, as well as the OpenEmbedded and Yocto projects, revealed the latest updates for RISC-V support in the Linux kernel and related software. The report has a rather short shelf life, admitted Raj: "The software is developing very fast so what I say today may be obsolete tomorrow -- we've already seen a lot of basic tool, compilers, and toolchain support landing upstream."

- SiFive launches RISC-V processor cores for real-time applications-Venture Beat

https://venturebeat.com/2018/10/31/sifive-launches-risc-v-processor-cores-for-real-timeapplications/

SiFive wants to exploit the commercial opportunities of RISC-V, an open chip architecture that challenges more-closed rivals, such as Arm and Intel. Today it is announcing the availability of its SiFive Core IP 7 Series.
These high-performance RISC-V cores are designed for embedded devices and real-time applications. The 7 Series includes the E7, S7 and the U7 series. The SiFive Core IP E7 Series provides a heterogeneous, customizable architecture with hard real-time capabilities, and the SiFive Core IP S7 series brings high performance 64-bit architectures to the embedded markets, while the SiFive Core IP U7 Series is a Linux-capable applications processor with a highly configurable memory architecture for domain-specific customization.

2. TECHNOLOGY USED

RISC-V (pronounced "risk-five") is an Open source hardware instruction set architecture (ISA) based on established reduced instruction set computer (RISC) principles.

2.1.Significance:

The RISC-V authors aim to provide several CPU designs freely available under BSD licenses, which allow derivative works, such as RISC-V chip designs, to be either open and free, as is RISC-V, or closed and proprietary.

The RISC-V authors have substantial research and user-experience validating their designs in silicon and simulation. The RISC-V ISA is a direct development from a series of academic computer-design projects. It was originated in part to aid such projects.

2.2.Software:

The RISC-V has a specification for user-mode instructions, and a preliminary specification for a general-purpose privileged instruction set, to support operating system

The design software includes a design compiler, Chisel, which can reduce the designs to Verilog for use in devices. This includes verification data for testing core implementations.

Available RISC-V software tools include a GNU Compiler Collection (GCC) toolchain (with GDB, the debugger), an LLVM toolchain, the OVPsim simulator (and library of RISC-V Fast Processor Models), the Spike simulator, and a simulator in QEMU.

Operating system support exists for the Linux kernel, FreeBSD, and NetBSD, but the supervisor-mode instructions are unstandardized as of 10 November 2016, so this support is provisional. The preliminary FreeBSD port to the RISC-V architecture was upstreamed in February 2016, and shipped in FreeBSD 11.0. Ports of Debian and Fedoraare stabilizing. A port of Das U-Boot exists. UEFI Spec v2.7 has defined the RISC-V binding and a tianocore port has been done by HPE engineers and is expected to be up streamed. There is a preliminary port of the seL4 microkernel. A simulator exists to run a RISC-V Linux system on a web browser using JavaScript.

2.3.RISC-V ISA Base:

RISC-V has a modular design, consisting of alternative base parts, with added optional extensions. The ISA base and its extensions are developed in a collective effort between industry, the research community and educational institutions. The base specifies instructions (and their encoding), control flow, registers (and their sizes), memory and addressing, logic (i.e., integer) manipulation, and ancillaries. The base alone can implement a simplified general-purpose computer, with full software support, including a general-purpose compiler.

2.4.RISC-VISA Extensions:

RISC-V has standardized a series of **standard extensions** beyond the integer base instructions which can be implemented or omitted as desired depending on the design goals (e.g. energy/area/performance/storage goals).

By default, only the core ISA must be implemented presenting great opportunity for area and energy optimization.

However, additional functionality is sometimes desired. RISC-V comes with a series of standard extensions that enable additional functionality beyond the core ISA such as floating point and operations and bitmanipulation.

Extensions can be implemented and omitted as desired. Those extensions are:

- **A** - Atomic instructions
- **B** - Bit manipulation instructions
- **C** - Compressed instructions
- **D** - Double-precision floating-point instructions
- **E** - Embedded applications, resource-constrained subset
- **F** - Single-precision floating-point instructions
- **G** - General (I + M + A + F + D)
- **I** - Integer base instructions
- **J** - Dynamically translated languages
- **L** - Decimal floating point instructions
- **M** - Integer multiplication and division instructions
- **N** - User-level interrupt instructions

  P - Packed-SIMD instructions
- **Q** - Quad-precision floating-point instructions
- **T** - Transactional Memory instructions
- **V** - Vector operations instructions.

2.5.Register Set:

RISC-V has 32 (or 16 in the embedded variant) integer registers, and, when the floating-point extension is implemented, 32 floating-point registers. Except for memory access instructions, instructions address only registers.

The first integer register is a zero register, and the remainder are general-purpose registers. A store to the zero register has no effect, and a read always provides 0. Using the zero register as a placeholder makes for a simpler instruction set. E.g., `move rx to ry` becomes `add r0 to rx and store in ry.`

Control and status registers exist, but user-mode programs can access only those used for performance measurement and floating-point management.

No instructions exist to save and restore multiple registers. Those were thought to be needless, too complex, and perhaps too slow.

2.6.Memory access:

Like many RISC designs, RISC-V is a load–store architecture: instructions address only registers, with load and store instructions conveying to and from memory.

Memory consists of and is addressed as 8-bit bytes, with words being in little-endian order. Words, up to the register size, can be accessed with the load and store instructions.

Accessed memory addresses need not be aligned to their word-width, but accesses to aligned addresses will always be the fastest.

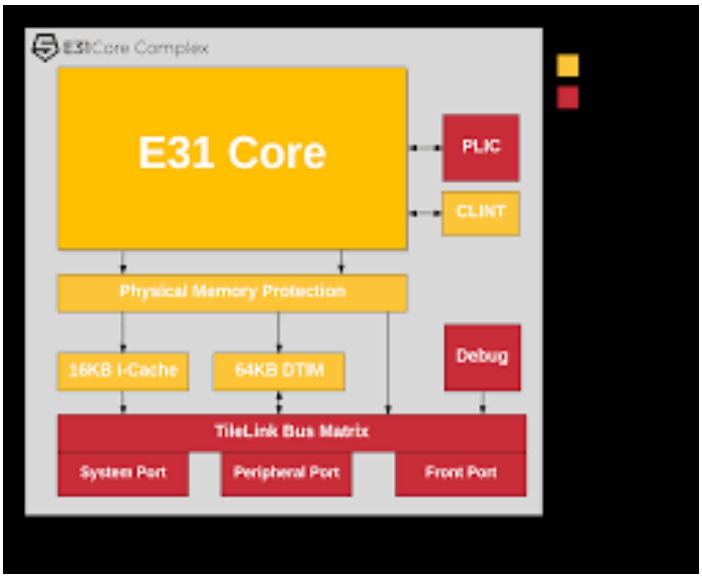
RISC-V manages memory systems that are shared between CPUs or threads by ensuring a thread of execution always sees its memory operations in the programmed order. But between threads and I/O devices, RISC-V is simplified: It doesn't guarantee the order of memory operations, except by specific instructions.

Like many RISC instruction sets (and some complex instruction set computer (CISC) instruction sets, such as x86 and IBM System/360 families), RISC-V lacks address-modes that write back to the registers. For example, it does not auto-increment.

RISC-V is little-endian to resemble other familiar, successful computers, for example, x86. This also reduces a CPU's complexity and costs slightly because it reads all sizes of words in the same order. For example, the RISC-V instruction set decodes starting at the lowest-addressed byte of the instruction. The specification leaves open the possibility of non-standard big-endian or bi-endian systems.

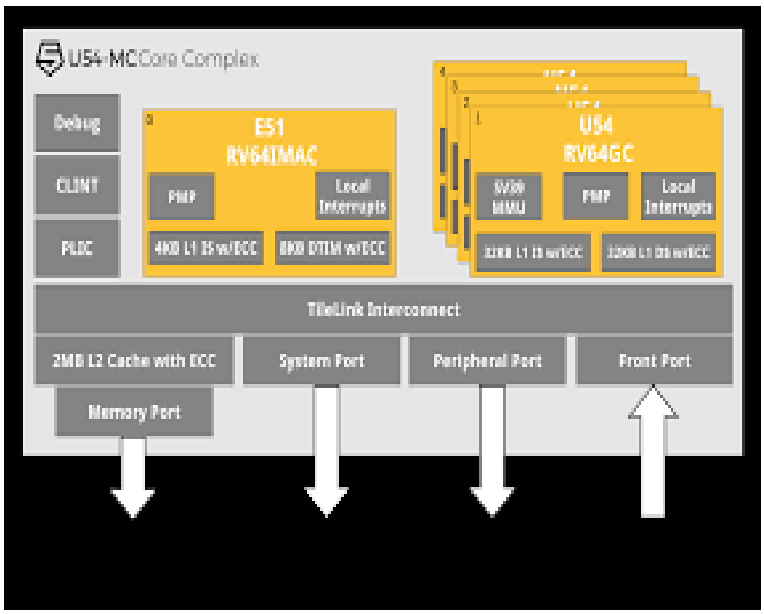
2.7.External debug system:

There is a preliminary specification for RISC-V's hardware-assisted debugger. The debugger will use a transport system such as Joint Test Action Group (JTAG) or Universal Serial Bus (USB) to access debug registers. A standard hardware debug interface may support either a standardized abstract interface or instruction feeding.



2.8.32-IMAC Core Architecture

Crowd Supply HiFive1 Arduino-Compatible RISC-V Development Kit features the Freedom E310, a commercially available RISC-V SoC designed for microcontroller, embedded, IoT, and wearable applications. The FE310 features SiFive's E31 CPU Coreplex, a high-performance, 32-bit RV32IMAC core.



Hifive1 Board

2.9.Risc V CPU

The FE310 is among the fastest microcontrollers in the market. Additional features include a 16KB L1 Instruction Cache, a 16KB Data SRAM scratchpad, hardware multiply/divide, and a debug module. Other features include flexible clock generation with on-chip oscillators and PLLs, and a wide variety of peripherals including UARTs, QSPI, PWMs, and timers. Multiple power domains and a low-power standby mode ensure a wide variety of applications can benefit from the FE310.

Specifications:

- Microcontroller: SiFive Freedom E310 (FE310)
- CPU: SiFive E31 CPU
- Architecture: 32-bit RV32IMAC
- Speed: 320+ MHz
- Performance: 1.61DMIPs/MHz
- Memory: 16KB Instruction Cache, 16 KB Data Scratchpad
- Other Features: Hardware Multiply/Divide, Debug Module, Flexible Clock Generation with on-chip oscillators and PLLs
- Operating Voltage: 3.3V and 1.8V
- Input Voltage: 5V USB or 7-12VDC Jack

- IO Voltages: Both 3.3V or 5V supported
- Digital I/O Pins: 19
- PWM Pins: 9
- SPI Controllers/HW CS Pins: 1/3
- External Interrupt Pins: 19
- External Wakeup Pins: 1
  Flash Memory: 128Mbit Off-Chip (ISSI SPI Flash)
- Host Interface (microUSB): Program, Debug, and Serial Communication
- Dimensions: 68x51mm
- Weight: 22 g

SOFTWARE:

- Freedom E SDK
- Arduino IDE Support

3. SYSTEM ANALYSIS

3.1.HIFIVE1 (FE310-G000)

The FE310-G000 is the first Freedom E300 SoC, and forms the basis of the HiFive1 development board for the Freedom E300 family. The FE310-G000 is built around the E31 Core Complex instantiated in the Freedom E300 platform and fabricated in the TSMC CL018G 180nm process. This manual describes the specific configuration for the FE310-G000. The SiFive FE310-G000 is compatible with all applicable RISC-V standards, and this document should be read together with the official RISC-V user-level, privileged, and external debug architecture specifications.

3.2.RISC-V Core

The FE310-G000 includes a SiFive E31 RISC-V core, which is a high-performance single-issue in-order execution pipeline, with a peak sustainable execution rate of one instruction per clock cycle. The RISC-V core supports Machine mode only as well as the standard Multiply, Atomic, and Compressed RISC-V extensions (RV32IMAC**).**

3.3.On Chip Memory System

The FE310-G000 memory system has Data Tightly Integrated Memory subsystem (DTIM) optimized for high performance. The data subsystem has a DTIM size of 16 KiB. The instruction subsystem consists of a 16 KiB 2-way instruction cache. The system mask ROM is 8KiB in size and contains simple boot code. The system ROM also holds the platform configuration string and debug ROM routines.

3.4.Interrupts

TheFE310-G000includesaRISC-Vstandardplatform-levelinterruptcontroller(PLIC),whichsupports 51 global interrupts with 7 priority levels. This Core Complex also provides the standard RISC-V machine-mode timer and software interrupts via the Core Local Interruptor (CLINT).

3.5.Always-On (AON) Block

The AON block contains the reset logic for the chip, an on-chip low-frequency oscillator, a watchdog timer, connections for an off-chip low-frequency oscillator,thereall-time clock,a programmable powermanagement unit, and 16×32 bit backup registers that retain state while the rest of the chip is in a low-power mode.It can be used to put the system in sleep and wake it up .

	HiFive1	Arduino 101	Arduino Zero	Arduino Uno
Microcontroller	Freedom E310	Intel Curie Module	Atmel ATSAMD21G18	Atmel ATmega328P
Open-Source RTL?	Yes	No	No	No
CPU Speed	320+ MHz	32 MHz	48 MHz	16 MHz
Bits	32-bit	32-bit	32-bit	8-bit
CPU Core	SiFive E31	Intel Quark SE	ARM Cortex M0+	AVR
CPU ISA	RISC-V RV32IMAC	x86	ARMv6-M	AVR
Performance				
DMIPs/MHz*	1.61	1.3	0.93	0.30
Total Dhrystones*	515.2	41.6	44.64	5
DMIPS/mW*	3.16	0.35	-	0.10
Board Specs				
IO Voltage	3.3 V and 5 V	3.3 V and 5 V	3.3 V Only	5 V Only
Digital IO	19	14	14	14
PWM	9	4	10	6
SRAM [kB]	16	24	32	2
Flash [kB]	16384	196	256	32
USB	Micro	Regular	2 Micro	Regular

** HiFive1 DMIPs/mW measured at 1.61 V, 200 MHz operation. Intel Dhrystone data and DMIPs/mW taken from their datasheet and product material. Arduino Uno DMIPs/mW estimated based on ATmega328P datasheet and this site.*

3.6.GPIO Complex

The GPIO complex manages the connection of digital I/O pads to digital peripherals, including SPI, UART, and PWM controllers, as well as for regular programmed I/O operations. FE310-G000 has two additional QSPI controllers in the GPIO block, one with four chip selects and one with one. FE310-G000 also has two UARTs. FE310-G000 has three PWM controllers, two with 16-bit precision and one with 8-bit precision.

3.7.Quad-SPI Flash

A dedicated quad-SPI(QSPI)flash interface is provided to hold code and data for the system. It supports burst reads of 32 bytes over TileLink to accelerate instruction cache refills.

The QSPI can be programmed to support eXecute-In-Place (XIP) modes to reduce SPI command overhead on instruction cache refills. The QSPI interface also supports single-word data reads over the primary TileLink interface, as well as programming operations using memory-mapped control registers.

3.8.Hardware Serial Peripheral Interface

Hardware serial peripheral interfaces (SPI) are available and provide a means for serial communication between the FE310-G000 and off-chip devices

3.9.Universal Asynchronous Receiver/Transmitter

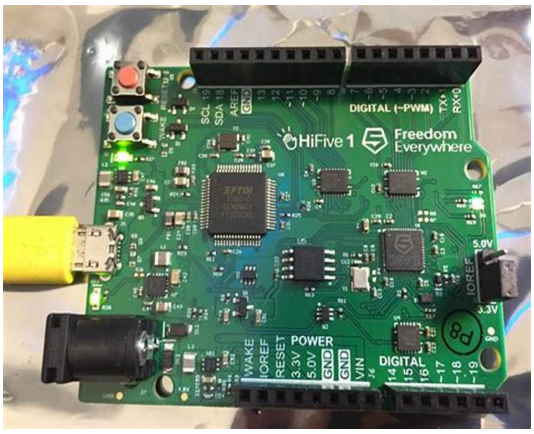
Multiple universal asynchronous receiver/transmitter (UARTs) are available and provide a means for serial communication between the FE310-G000 and off-chip devices.

3.10.Pulse Width Modulation

The pulse width modulation (PWM) peripheral can generate multiple types of waveforms on GPIO output pins, and can also be used to generate several forms of internal timer interrupt.

3.11.Debug Support

The debug module is accessed over JTAG, and has support for two programmable hardware breakpoints. The debug RAM has 28 bytes of storage. A four-wire 1149.1JTAG connection is used to connect the external debugger to the internal debug module.



Hifive1

4. INTERFACING

4.1.Boot and Run

The HiFive1 comes programmed with a simple bootloader and a demo software program which prints to the UART and cycles through the RGB LED in a rainbow pattern. You can respond on the UART to indidcate that the LEDs are changing and get a “PASS” message. This program will be overwritten in the SPI Flash when you program new software into the board with the SDK, but the bootloader code will not be modified.

4.2.Setting the baud rate

Using a terminal emulator such as GNU screen on Linux,open a console connection from the host computer to the HiFive1.

>sudo screen /dev/ttyUSB1 115200

If you are running on Ubuntu-style Linux, the below is an example of steps you may need to follow to access your dev kit without sudo permissions:

1. With your board’s debug interface connected, make sure your device shows up with the lsusb command:

> lsusb

Bus XXX Device XXX: ID 0403:6010 Future Technology Devices International, Ltd FT2232C Dual USB-UART/FIFO IC

2. Set the udev rules to allow the device to be accessed by the plugdev group:

> sudo vi /etc/udev/rules.d/99-openocd.rules

# These are for the HiFive1 Board SUBSYSTEM=="usb", ATTR{idVendor}=="0403", ATTR{idProduct}=="6010", MODE="664", GROUP="plugdev"
SUBSYSTEM=="tty", ATTRS{idVendor}=="0403", ATTRS{idProduct}=="6010", MODE="664", GROUP="plugdev"
# These are for the Olimex Debugger for use with E310 Arty Dev Kit
SUBSYSTEM=="usb", ATTR{idVendor}=="15ba",
ATTR{idProduct}=="002a", MODE="664", GROUP="plugdev"
SUBSYSTEM=="tty", ATTRS{idVendor}=="15ba", ATTRS{idProduct}=="002a", MODE="664", GROUP="plugdev"

3 .See if your board shows up as a serial device belonging to the plugdev group:

> ***ls /dev/ttyUSB****

/dev/ttyUSB0 /dev/ttyUSB1

> ls -l /dev/ttyUSB1

crw-rw-r-- 1 root plugdev 188, 1 Nov 28 12:53 /dev/ttyUSB1

4.Add yourself to the plugdev group.

> sudo usermod -a -G plugdev <your user name>(hp in our case)

5. Log out and log back in, then check that you're now a member of the plugdev group:

> groups

plugdev

Now you should be able to access the serial (UART) and debug interface without sudo permissions.

4.3.Software Development Flow

Software Development with the Freedom E SDK Compiling the Freedom E SDK Toolchain.

The Freedom E Software Development Kit provides everything required to compile, customize, and debug C and/or RISC-V assembly programs: GCC 6.1.0 cross-compilation toolchain, RISC-V enabled GDB and OpenOCD, etc.

To clone the Freedom E SDK git repository:

>git clone --recursive https://github.com/sifive/freedom-e-sdk.git

Install all the necessary packages described in the repository's README.md file. To build the software toolchain:

>cd freedom-e-sdk
> make tools

To keep your software toolchain up to date with the upstream repository:

>cd freedom-e-sdk
>git pull origin master
> git submodule update --init –recursive
> make tools

Compiling Software Programs

To build a C program that will be loaded by the debugger/programmer into the SPI Flash, use the FreedomESDKtocompile. An example is provided in the software/demo gpio directory.

To build the program:

>cd freedom-e-sdk
>make software PROGRAM=demo gpio BOARD=freedom-e300-hifive1

To compile the Dhrystone benchmark instead:

>cd freedom-e-sdk
> make software PROGRAM=dhrystone BOARD=freedom-e300-hifive1

Uploading Software Programs

To upload the program to the SPI flash, connect the board's debug interface , Then execute:

>cd freedom-e-sdk
>make upload PROGRAM=<your desired program> BOARD=freedom-e300-hifive1

Debugging Running Programs

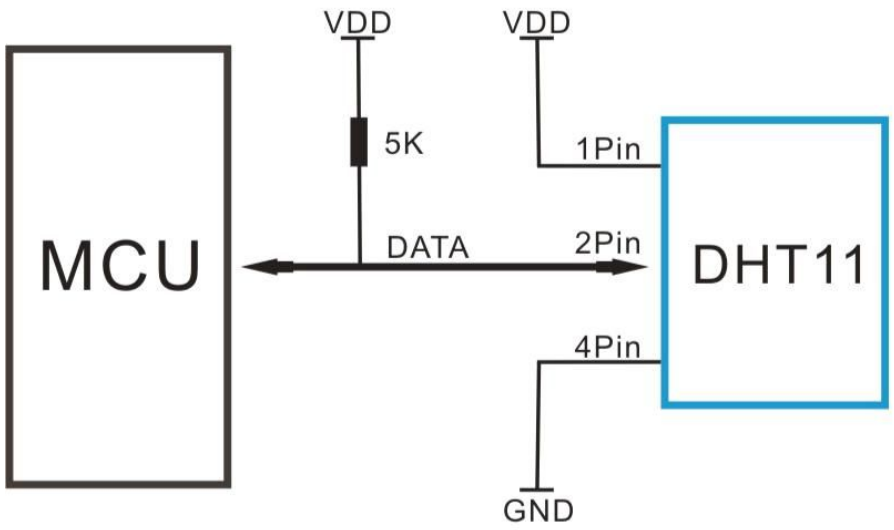
To debug your program with GDB, connect your board and launch the debugger:

>cd freedom-e-sdk
>make run debug PROGRAM=<your desired program> BOARD=freedom-e300-hifive1

This will automatically launch OpenOCD and GDB, connect to the board, and halt the currently running program. You can step through the running program with stepi, or load the new program using load. The usual suite of GDB commands are available to set breakpoints etc.

5. METHODOLOGY

5.1.Block Diagram



Microprocessor and DHT11 of connection typical application circuit as shown above, DATA pull the microprocessor I / O ports are connected.

- Typical application circuit recommended in the short cable length of 20 meters on the 5.1K pull-up resistor, the resistance of greater than 20 meters under the pull-up resistor on the lower of the actual situation.
- When using a 3.5V voltage supply cable length shall not be greater than 20cm. Otherwise, the line voltage drop will cause the sensor power supply shortage, caused by measurement error.
- Each read out the temperature and humidity values are the results of the last measurement For real-time data, sequential read twice, but is not recommended to repeatedly read the sensors, each read sensor interval is greater than 5 seconds can be obtained accurate data.

5.2. Power and Pin

DHT11's power supply is 3-5.5V DC. When power is supplied to the sensor, do not send any instruction to the sensor in within one second in order to pass the unstable status. One capacitor valued 100nF can be added between VDD and GND for power filtering.

5.3.Communication Process: Serial Interface (Single-Wire Two-Way)

Single-bus data format is used for communication and synchronization between MCU and DHT11 sensor. One communication process is about 4ms.Data consists of decimal and integral parts. A complete data transmission is **40bit**, and the sensor sends **higher data bit** first.

5.4.Data format:

8bit integral RH data + 8bit decimal RH data + 8bit integral T data + 8bit decimal T data + 8bit check sum. If the data transmission is right, the check-sum should be the last 8bit of "8bit integral RH data + 8bit decimal RH data + 8bit integral T data + 8bit decimal T data".

Example 1: 40 data is received:

0011 0101	0000 0000	0001 1000	0000 0000	0100 1101
High humidity 8	Low humidity 8	High temp. 8	Low temp. 8	Parity bit

Calculation：

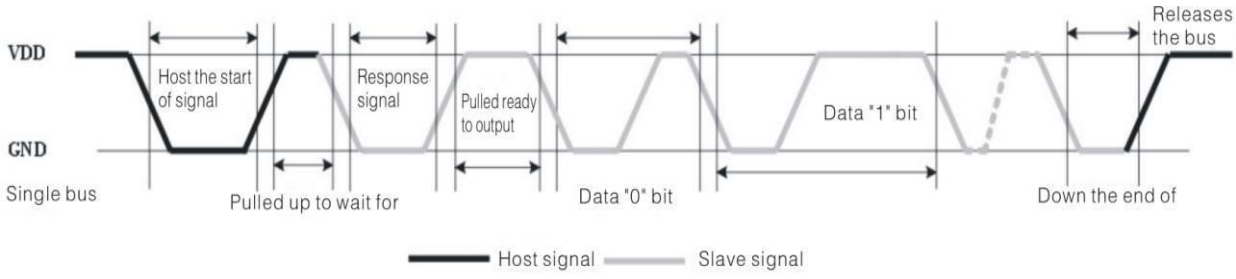
0011 0101+0000 0000+0001 1000+0000 0000= 0100 1101
Received data is correct：
Humidity：0011 0101=35H=53%RH
Temperature：0001 1000=18H=24℃
Example 2: 40 data is received:

0011 0101	0000 0000	0001 1000	0000 0000	0100 1001
High humidity 8	Low humidity 8	High temp. 8	Low temp. 8	Parity bit

Calculation：

0011 0101+0000 0000+0001 1000+0000 0000＝ 0100 1101
01001101≠0100 1001
The received data is not correct, give up, to re-receive data.
Data Timing Diagram

User host (MCU) to send a signal, DHT11 converted from low-power mode to high-speed mode, until the host began to signal the end of the DHT11 send a response signal to send 40bit data, and trigger a letter collection. The signal is sent as shown.



Data Timing Diagram

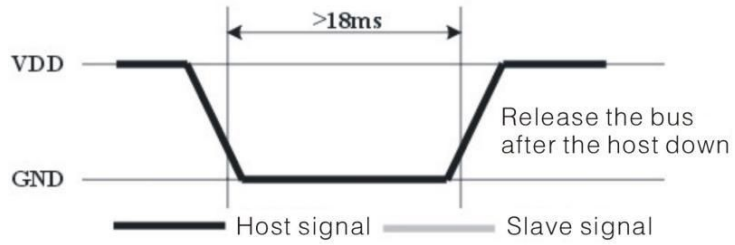
Communication between the master and slave can be done through the following steps (peripherals (such as microprocessors) read DHT11 the data of steps).

Step 1:

After power on DHT11 (DHT11 on after power to wait 1S across the unstable state during this period can not send any instruction), the test environment temperature and humidity data, and record the data, while DHT11 the DATA data lines pulled by pull-up resistor has been to maintain high; the DHT11 the DATA pin is in input state, the moment of detection of external signals.

Step 2:

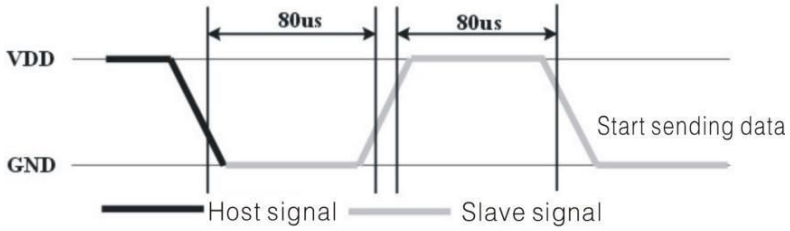
Microprocessor I / O set to output at the same time output low, and low hold time can not be less than 18ms, then the microprocessor I / O is set to input state, due to the pull-up resistor, a microprocessor/ O DHT11 the dATA data lines also will be high, waiting DHT11 to answer signal, send the signal as shown:



Host sends a start signal

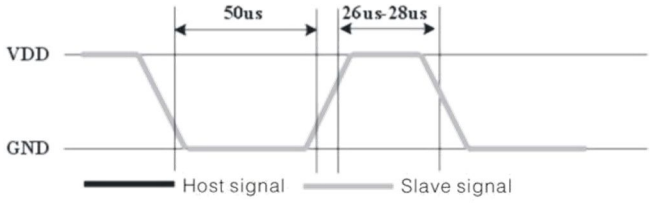
Step 3:

DATA pin is detected to an external signal of DHT11 low, waiting for external signal low end the delay DHT11 DATA pin in the output state, the output low of 80 microseconds as the response signal, followed by the output of 80 micro-seconds of high notification peripheral is ready to receive data, the microprocessor I / O at this time in the input state is detected the I / O low (DHT11 response signal), wait 80 microseconds highdata receiving and sending signals as shown:

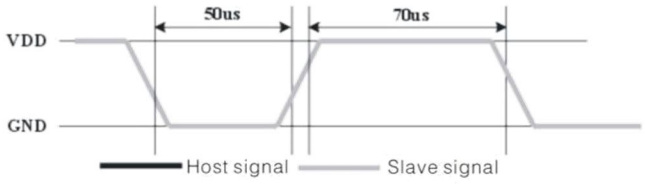


Step 4:

Output by DHT11 the DATA pin 40, the microprocessor receives 40 data bits of data "0" format is low level of 50 microseconds and 26-28 microseconds according to the changes in the I / O level, bit data "1" format is high level of low plus, 50 microseconds to 70 microseconds. Bit data "0", "1" signal format as shown:



Bit data "0" bit format



Bit data "1" bit format

5.5End signal:

Continue to output the low 50 microseconds after DHT11 the DATA pin output 40 data, and changed the input state, along with pull-up resistor goes high. But DHT11 internal re-test environmental temperature and humidity data, and record the data, waiting for the arrival of the external signal.

6.TESTING AND RESULTS

6.1.GPIO Testing by LED blinking

Code

```
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>
#include "platform.h"
#include <string.h>
#include "plic/plic_driver.h"
#include "encoding.h"
#include <unistd.h>
#include "stdatomic.h"

#define LED1 PIN_3_OFFSET

#define LED2 PIN_5_OFFSET

#define LED3 PIN_6_OFFSET



static void _putc(char c) {
  while ((int32_t) UART0_REG(UART_REG_TXFIFO) < 0));
  UART0_REG(UART_REG_TXFIFO) = c;
}

int _getc(char * c){
  int32_t val = (int32_t) UART0_REG(UART_REG_RXFIFO);

if (val > 0) {
    *c =  val & 0xFF;
    return 1;
  }
  return 0;
}


static void _puts(const char * s) {
  while (*s != '\0'){
    _putc(*s++);
  }
}
```

```
//DELAY
void wait_ms(uint64_t ms)
{
      static const uint64_t ms_tick = RTC_FREQ/1000;
      volatile  uint64_t  *  mtime    =  (uint64_t*)  (CLINT_CTRL_ADDR  +
CLINT_MTIME);
      uint64_t then = (ms_tick * ms) + *mtime;
      while(*mtime<then);
}

int main(int argc, char **argv)
{



  PRCI_REG(PRCI_HFROSCCFG) |= ROSC_EN(1);
  // Run off 16 MHz Crystal for accuracy. Note that the
  // first line is
  PRCI_REG(PRCI_PLLCFG) = (PLL_REFSEL(1) | PLL_BYPASS(1));
  PRCI_REG(PRCI_PLLCFG) |= (PLL_SEL(1));
  // Turn off HFROSC to save power
  PRCI_REG(PRCI_HFROSCCFG) &= ~(ROSC_EN(1));

  // Configure UART to print
  GPIO_REG(GPIO_OUTPUT_VAL) |= IOF0_UART0_MASK;
  GPIO_REG(GPIO_OUTPUT_EN)  |= IOF0_UART0_MASK;
  GPIO_REG(GPIO_IOF_SEL)    &= ~IOF0_UART0_MASK;
  GPIO_REG(GPIO_IOF_EN)     |= IOF0_UART0_MASK;

  // 115200 Baud Rate
  UART0_REG(UART_REG_DIV) = 138;
  UART0_REG(UART_REG_TXCTRL) = UART_TXEN;
  UART0_REG(UART_REG_RXCTRL) = UART_RXEN;

  // Wait a bit to avoid corruption on the UART.
  // (In some cases, switching to the IOF can lead
  // to output glitches, so need to let the UART
  // reciever time out and

uint32_t a;
printf("Enter value\n\r");
scanf("%d",&a);
printf("the vcaue is %d",a);
/*
unsigned long   stime = *(volatile unsigned long *)(CLINT_CTRL_ADDR +
CLINT_MTIME);
```

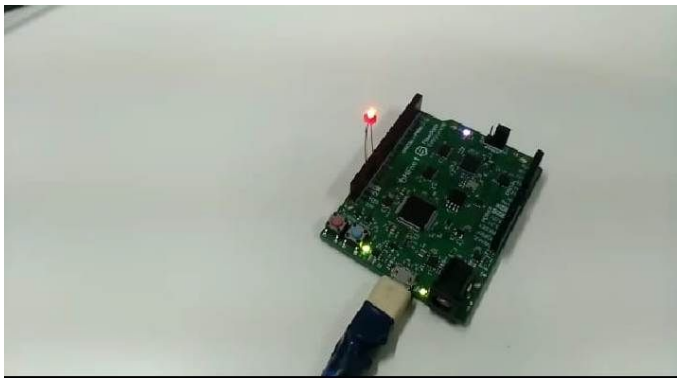
```
 printf("RTC FREQUENCY: %d\n\r",RTC_FREQ);
//printf("RTC FREQUENCY: %d\n\r",CORE_FREQ);

     //LED INITIALIZING
GPIO_REG(GPIO_OUTPUT_EN)   |=  (0x1 << LED1) | (0X1 << LED2) | (0X1 <<
LED3);

     while(1)
     {
     static const uint64_t ms_tick = RTC_FREQ/1000;

GPIO_REG(GPIO_OUTPUT_VAL)  =  (0x1 << LED1) | (0X0 << LED2) | (0X0 <<
LED3) ;
printf("RED ON\n\r");
wait_ms(1000) ;
unsigned long    time = *(volatile unsigned long *)(CLINT_CTRL_ADDR +
CLINT_MTIME);

//uint32_t curtime =
printf("Time: %f",(time - stime) ;

GPIO_REG(GPIO_OUTPUT_VAL)  =  (0x0 << LED1) | (0X1 << LED2) | (0X0 <<
LED3) ;
printf("GREEN ON\n\r");
wait_ms(1000) ;

//printf("Time:    %f",(((volatile    uint64_t*)(CLINT_CTRL_ADDR    +
CLINT_MTIME) - *  stime)/263041843)) ;


GPIO_REG(GPIO_OUTPUT_VAL)  =  (0x0 << LED1) | (0X0 << LED2) |    (0X1
<< LED3) ;
printf("BLUE ON\n\r");
wait_ms(1000) ;

//printf("Time:  %f",(-  *  ((volatile  uint64_t*)(CLINT_CTRL_ADDR  +
CLINT_MTIME) - *  stime)/ms_tick)) ;

     }*/

}
```

6.1.1.RESULTS FOR SINGLE LED:

```
GPIO_REG(GPIO_OUTPUT_EN)   |=  (0x1 << LED);
while(1)
{
        GPIO_REG(GPIO_OUTPUT_VAL)  =  (0x1 << LED);
        printf("LED ON\n\r");
        wait_ms(1000) ;
        GPIO_REG(GPIO_OUTPUT_VAL)  =  (0x0 << LED);
        printf("LED OFF\n\r");
        wait_ms(1000) ;
}
```

```
LED ON
LED OFF
LED ON
LED OFF
LED ON
LED OFF
LED ON
LED OFF
```

Output display for single Led in minicom

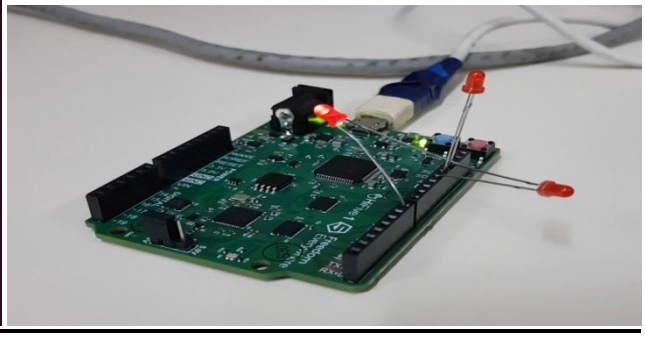


Single Led blinking-Code and Output

6.1.2.FOR 3 LEDS:

```
//LED INITIALIZING
GPIO_REG(GPIO_OUTPUT_EN)   |=  (0x1 << LED1) | (0X1 << LED2) | (0X1 << LED3);
while(1)
{

        GPIO_REG(GPIO_OUTPUT_VAL)  =  (0x1 << LED1) | (0X0 << LED2) | (0X0 << LED3) ;
        printf("RED ON\n\r");
        wait_ms(1000) ;
        GPIO_REG(GPIO_OUTPUT_VAL)  =  (0x0 << LED1) | (0X1 << LED2) | (0X0 << LED3) ;
        printf("GREEN ON\n\r");
        wait_ms(1000) ;
        GPIO_REG(GPIO_OUTPUT_VAL)  =  (0x0 << LED1) | (0X0 << LED2) | (0X1 << LED3) ;
        printf("BLUE ON\n\r");
        wait_ms(1000) ;
}
```

```
RED ON
GREEN ON
BLUE ON
RED ON
GREEN ON
BLUE ON
RED ON
GREEN ON
BLUE ON
RED ON
GREEN ON
BLUE ON
RED ON
GREEN ON
BLUE ON
```



Output display for 3 Leds over minicom **3 Leds blinking-Code and Output**

6.2.Reading values from the pins

Code

```
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>
#include "platform.h"
#include <string.h>
#include "plic/plic_driver.h"
#include "encoding.h"
#include <unistd.h>
#include "stdatomic.h"

#define SEND PIN_8_OFFSET

#define REC PIN_12_OFFSET

int pow(int x)
{
      int p = 1;
      while(x > 0)
      {
            p = p * 2;
            x--;
      }
      return p;
}



static void _putc(char c) {
  while ((int32_t) UART0_REG(UART_REG_TXFIFO) < 0);
```

```
  UART0_REG(UART_REG_TXFIFO) = c;
}

int _getc(char * c){
  int32_t val = (int32_t) UART0_REG(UART_REG_RXFIFO);
  if (val > 0) {
    *c =  val & 0xFF;
    return 1;
  }
  return 0;
}


static void _puts(const char * s) {
  while (*s != '\0'){
    _putc(*s++);
  }
}

//DELAY
void wait_ms(uint64_t ms)
{
     static const uint64_t ms_tick = RTC_FREQ/1000;
     volatile uint64_t * mtime = (uint64_t*) (CLINT_CTRL_ADDR +
CLINT_MTIME);
     uint64_t then = (ms_tick * ms) + *mtime;
     while(*mtime<then);
}

int main(int argc, char **argv)
{



  PRCI_REG(PRCI_HFROSCCFG) |= ROSC_EN(1);
  // Run off 16 MHz Crystal for accuracy. Note that the
  // first line is
  PRCI_REG(PRCI_PLLCFG) = (PLL_REFSEL(1) | PLL_BYPASS(1));
  PRCI_REG(PRCI_PLLCFG) |= (PLL_SEL(1));
  // Turn off HFROSC to save power
  PRCI_REG(PRCI_HFROSCCFG) &= ~(ROSC_EN(1));

  // Configure UART to print
  GPIO_REG(GPIO_OUTPUT_VAL) |= IOF0_UART0_MASK;
  GPIO_REG(GPIO_OUTPUT_EN)  |= IOF0_UART0_MASK;
  GPIO_REG(GPIO_IOF_SEL)    &= ~IOF0_UART0_MASK;
  GPIO_REG(GPIO_IOF_EN)     |= IOF0_UART0_MASK;

  // 115200 Baud Rate
  UART0_REG(UART_REG_DIV) = 138;
  UART0_REG(UART_REG_TXCTRL) = UART_TXEN;
  UART0_REG(UART_REG_RXCTRL) = UART_RXEN;
```

```
  // Wait a bit to avoid corruption on the UART.
  // (In some cases, switching to the IOF can lead
  // to output glitches, so need to let the UART
  // reciever time out and



     //uint64_t  *    stime  =  (volatile  uint64_t*)(CLINT_CTRL_ADDR  +
CLINT_MTIME);

//LED INITIALIZING

     uint8_t z ,y;
GPIO_REG(GPIO_OUTPUT_EN)   |=  (0x1 << SEND) ;

//GPIO_REG(GPIO_OUTPUT_VAL)  |=  0x00 ;

GPIO_REG(GPIO_OUTPUT_VAL)  |=  (0x0 << SEND) ;
     while(1)
     {


          GPIO_REG(GPIO_INPUT_EN)  |=  (0x1 << REC) ;
          GPIO_REG(GPIO_INPUT_VAL) = 0x00;
          y = REC ;
          printf("OFFSET: %d\n\r",y);

          z = GPIO_REG(GPIO_INPUT_VAL) & (pow(REC));
          printf("value: %d\n\r",z);

          printf("value1: %d\n\r",GPIO_REG(GPIO_INPUT_VAL));
          /*z = GPIO_REG(GPIO_INPUT_VAL)  |=  (0x1 << REC) ;
          printf("value: %d\n\r",*z);

          if((GPIO_REG(GPIO_INPUT_VAL)  |=  (0x1 << REC)) == 0)
               printf("HIGH\n\r");
          else
               printf("LOW\n\r");*/
          wait_ms(1000) ;

     }

}
```

6.2.1.READING OF LOW VALUE:

```
uint8_t z ;
GPIO_REG(GPIO_OUTPUT_EN)   |=  (0x1 << SEND) ;
GPIO_REG(GPIO_OUTPUT_VAL)  |=  (0x0 << SEND) ;
while(1)
{
        GPIO_REG(GPIO_INPUT_EN)  |=  (0x1 << REC) ;
        z = GPIO_REG(GPIO_INPUT_VAL) & (pow(REC));
        printf("value: %d\n\r",z);
        wait_ms(1000) ;
}
```

```
core freq at 29048543 Hz
value: 0
value: 0
value: 0
value: 0
value: 0
value: 0
value: 0
value: 0
value: 0
value: 0
```

Output for Low Value 0 over Minicom

6.2.2.READING OF HIGH VALUE:

```
uint8_t z ;
GPIO_REG(GPIO_OUTPUT_EN)   |=  (0x1 << SEND) ;
GPIO_REG(GPIO_OUTPUT_VAL)  |=  (0x1 << SEND) ;
while(1)
{
        GPIO_REG(GPIO_INPUT_EN)  |=  (0x1 << REC) ;
        z = GPIO_REG(GPIO_INPUT_VAL) & (pow(REC));
        printf("value: %d\n\r",z);
        wait_ms(1000) ;
}
```

```
value: 16
value: 16
value: 16
value: 16
value: 16
value: 16
value: 16
value: 16
```

Reading high value (16= 2^4(here 4 as offset))

6.3.DHT11 LIBRARY CODE

The precision needed was in microseconds delay ,we could reach milliseconds only.

```
#include <stdio.h>
#include "platform.h"
#include <string.h>
#include "plic/plic_driver.h"
#include "encoding.h"
#include <unistd.h>
#include "stdatomic.h"

#define DPIN PIN_12_OFFSET                                          //
DPIN = DHT-11 Connnected PIN

#define GOT_VALUES 0
#define CHECKSUM_ERROR -1
#define TIMEOUT_ERROR -2
int humidity, temperature ;

uint8_t bits[5] ;
uint8_t cnt = 7 ;
uint8_t idx = 0 ;

static const uint64_t us_tick = RTC_FREQ/1000000 ;

int pow11(int x)
{
      int p = 1;
      while(x > 0)
      {
            p = p * 2;
            x--;
      }
      return p;
}

uint8_t z;

// DELAY IN MILLISECONDS
void wait_us (uint64_t us)
{
      //static const uint64_t us_tick = RTC_FREQ/1000000 ;
      volatile  uint64_t  *  mtime  =  (uint64_t*)  (CLINT_CTRL_ADDR  +
CLINT_MTIME) ;
      uint64_t then = (us_tick * us) + * mtime ;
      while(* mtime < then);
}
```

```
int main (int argc, char **argv)
{
      uint32_t p, z;
      p = pow11(DPIN);
       printf("p val %d\n\r",p);


 // EMPTY BUFFER
      for (int i = 0; i < 5; i++)
            bits[i] = 0 ;

      // REQUEST SAMPLE FROM PROCESSOR
      GPIO_REG(GPIO_OUTPUT_EN)  |= (0X1 << DPIN) ;
      //GPIO_REG(GPIO_PULLUP_EN)  |= (0X1 << DPIN) ;
      GPIO_REG(GPIO_OUTPUT_VAL) |= (0X0 << DPIN) ;
      wait_us(18000) ;                        // 18 milliseconds delay

       GPIO_REG(GPIO_OUTPUT_VAL) |= (0X1 << DPIN) ;
       wait_us(40) ;                         // 40 microseconds delay

      /*GPIO_REG(GPIO_INPUT_EN)   |= (0X1 << DPIN) ;
        GPIO_REG(GPIO_PULLUP_EN)  |= (0X1 << DPIN) ;*/




      GPIO_REG(GPIO_OUTPUT_EN)  &= ~(0x1 << DPIN);
      GPIO_REG(GPIO_PULLUP_EN)  |= (0x1 << DPIN);
      GPIO_REG(GPIO_INPUT_EN)   |= (0X1 << DPIN) ;




      //DHT11 RESPONSE
      uint32_t loopcount = 10000 ;
z = GPIO_REG(GPIO_INPUT_VAL) & (pow11(DPIN));
      while(z == 0x0)
      {
printf("1");
//z = GPIO_REG(GPIO_INPUT_VAL) & (pow11(DPIN));
//printf("value: %d\n\r",z);
           if(loopcount-- == 0)
           {
                printf("TIMEOUT ERROR") ;
                return TIMEOUT_ERROR ;
           }
z = GPIO_REG(GPIO_INPUT_VAL) & (pow11(DPIN));
      }

      loopcount = 10000 ;
      while((GPIO_REG(GPIO_INPUT_VAL) & (pow11(DPIN)) != 0x0))
```

```
      {
printf("2");
//z = GPIO_REG(GPIO_INPUT_VAL) & (pow11(DPIN));
//printf("value: %d\n\r",z);




if(loopcount-- == 0)
            {
                  printf("TIMEOUT ERROR") ;
                  return TIMEOUT_ERROR ;
            }
      }


      // GETTING VALUES
      uint32_t mtime , ntime;

      for(int i = 0; i < 40; i++)
      {
            loopcount = 10000 ;
printf("vvvvvvv %ld\n\r",(GPIO_REG(GPIO_INPUT_VAL) & (pow11(DPIN))));
            while((GPIO_REG(GPIO_INPUT_VAL) & (pow11(DPIN)) ==0x0))
            {
printf("3");
//z = GPIO_REG(GPIO_INPUT_VAL) & (pow11(DPIN));
//printf("value: %d\n\r",z);

if(loopcount-- == 0)
                  {
                        printf("TIMEOUT ERROR") ;
                        return TIMEOUT_ERROR ;
                  }
            }

            mtime = *((uint64_t*) (CLINT_CTRL_ADDR + CLINT_MTIME)) ;
            //unsigned long t =  mtime ;

            loopcount = 10000 ;

            while((GPIO_REG(GPIO_INPUT_VAL) & (pow11(DPIN)) != 0x0))
            {
printf("4");
//z = GPIO_REG(GPIO_INPUT_VAL) & (pow11(DPIN));
//printf("value: %d\n\r",z);

if(loopcount-- == 0)
                  {
                  printf("TIMEOUT ERROR") ;
                  return TIMEOUT_ERROR ;
```

```
                }
            }

            ntime = *((uint64_t*) (CLINT_CTRL_ADDR + CLINT_MTIME)) ;

            if ((ntime - mtime) > (40 * us_tick)){
                bits[idx] |= (0x1 << cnt) ;

if (cnt == 0)
            {
                cnt = 7 ;
                idx++ ;
            }
            else
                cnt-- ;}
      }

      // ASSIGNING VALUES
      // bits[1] and bits[3] have zeros
      humidity    = bits[0];
      temperature = bits[2];

      uint8_t sum = bits[0] + bits[2];

      if (bits[4] != sum)
      {
printf("GOT VALUES\n") ;
            printf("HUMIDITY: %d RH \t TEMPERATURE: %d *c\n",bits[0],
bits[2]) ;
            printf("CHECKSUM ERROR\n") ;
            return CHECKSUM_ERROR ;
      }
      else
      {
            printf("GOT VALUES\n") ;
            printf("HUMIDITY: %d RH \t TEMPERATURE: %d *c\n",bits[0],
bits[2]) ;
            return GOT_VALUES ;
      }
      wait_us(3000000);
}


      *we need to reach the microseconds precision
```

7.FUTURE SCOPE

- Instead of 32 bit, 64 bit RISC-V architecture can be used.
- RISC-V can be an alternative processor in future.
- It's plays a main role in the development for industrial architecture.
- In future, DHT22 can be used in place of DHT11 which has higher accuracy for temperature and humidity and also has good stability for lower temperature.

- A Report from Venture Beats says that:

SiFive wants to exploit the commercial opportunities of RISC-V, an open chip architecture that challenges more-closed rivals, such as Arm and Intel. Today it is announcing the availability of its SiFive Core IP 7 Series.

These high-performance RISC-V cores are designed for embedded devices and real-time applications. The 7 Series includes the E7, S7 and the U7 series. The SiFive Core IP E7 Series provides a heterogeneous, customizable architecture with hard real-time capabilities, and the SiFive Core IP S7 series brings high performance 64-bit architectures to the embedded markets, while the SiFive Core IP U7 Series is a Linux-capable applications processor with a highly configurable memory architecture for domain-specific customization.

Together, the SiFive E7, S7 and U7 Core IP Series will enable the next generation of RISC-V cores that will power 5G, networking, storage, augmented reality, virtual reality, simultaneous location and mapping (SLAM), and sensor fusion functionality.

The cores offer efficient performance and optimized power consumption, appropriate for supporting smart offloads of datacenter workloads, as well as those of extremely power-efficient edge devices.

“We selected SiFive Core IP due to its best-in-class performance, as it was a third the power and a third the area of competing solutions,” said Fadu CEO Jihyo Lee in a statement. “As we target our next generation of advanced memory products, we look forward to seeing SiFive’s 7 Series Core IP bringing truly heterogeneous architectures, customizable 64-bit capability, and intelligence to the embedded space.”

The SiFive 7 Series provides scalable throughput with by 8+1 cores per cluster, 64-bit memory addressability for real-time processors, and in-cluster coherent combination of real-time processors and application processors. These features are currently not available from any other CPU vendors and are unique to SiFive’s Core IP series of processors.

SiFive has 300 employees. The company has raised $63.8 million from investors that include Sutter Hill Ventures, Spark Capital, Osage University Partners, and Chengwei Capital, along with strategic partners Huami, SK Telecom, Western Digital, and Intel Capital.

8.CONCLUSION

RISCV is a newly emerging architecture with less existence of libraries, So by using this library we can be able to sense temperature and humidity in the environment which is the basic necessity for an application in the industrial purpose through hifive1 board.

Apart from this ,CDAC is also a member of RISC-V project, our trial to develop the library for DHT11 ,would help ,many others at the time of requirements and pave a guiding path for future aspects and will be counted as a valuable contribution in this way.

9.REFERENCES

- **https://venturebeat.com/2018/10/31/sifive-launches-risc-v-processor-cores-for-real-time-applications/**
- **https://github.com/**
- **https://www.youtube.com/watch?v=8BCvyUlcRHY-video** RISCV by KHEMRAJ on EMBEDDED LINUX OpenIOT Summit held in Europe
- **https://riscv.org/**
- Getting started with Hifive1 Manual
- **https://en.wikipedia.org/wiki/RISC-V/**

- https://content.riscv.org/wp-content/uploads/2018/07/1100-19.07.18-Vinod-Ganesan-Gopinathan-Muthuswamy-IIT-Madras.pdf