Open Source RTOS Implementation for On-Board Computer (OBC) in STUDSAT-2

Bheema Rajulu
Project STUDSAT
Nitte Meenakshi Institute of Tech.
Bangalore - 560064, India
bheemarajulu90@gmail.com

Sankar Dasiga
Project STUDSAT
Nitte Meenakshi Institute of Tech.
Bangalore - 560064, India
dasiga@hotmail.com

Naveen R. Iyer
Project STUDSAT
M. S. Ramaiah Institute of Tech.
Bangalore - 560054, India
iyernaveenr@gmail.com

***Abstract*— STUDSAT-1 is the first pico-satellite developed in India by the undergraduate students from seven engineering colleges across South India. After the successful launch of STUDSAT-1 on 12th July, 2010, the team is developing STUDSAT-2 which is India's first twin satellite mission. STUDSAT-2 is undertaken by the undergraduate students from seven engineering colleges from the State of Karnataka, India. STUDSAT-2 consists of two Nano Satellites each having the dimension of 30 x 30 x 20 cm cube and weighing less than 10 kg. Each satellite has two payloads, one is the CMOS Camera with a resolution of 80 m and other is the ISL facility. The satellites are in along-the-track constellation architecture with the Master Satellite sending position data of the first image location on orbit and velocity of master satellite from GPS module to the Slave satellite via ISL for improving temporal resolution. On-board computer is the brain of a satellite whose operating system is designed to operate in resource constrained environments and is often tailored to the specific needs of its host system, while a real time operating system ensures that interrupts and other time critical tasks are processed when required. In order to maximize the accessibility the operating system must be inexpensive, and readily available, which demands an open-source OS. Implementation of a Real-Time Operating System (RTOS) provides several priority levels for tasks execution. FreeRTOS fulfills the requirements by having real-time capabilities, full availability of the source code, functional development environment, cross-platform support and community support. After an extensive literature survey, the ARM microcontroller with low power and high performance capability, STM32 ARM Cortex-M4 with 168 MHz CPU having 210 DMIPS is chosen as the main controller that supports FreeRTOS that can run on ARM Cortex-M cores. The On-Board Computer controls and coordinates the tasks of all other subsystems of the satellite. The different tasks executed by these subsystems have multi-threading and preemptive multitasking characteristics. A real-time operating system, FreeRTOS, uses advanced task scheduling techniques and a preemptive kernel, which allows multi-threading of processes to occur. This paper presents the technical justication for using FreeRTOS in STUDSAT-2. It presents implementation of FreeRTOS on STM32Cortex M4 controller which will increase the functional integration and performance of small satellite that simplifies software development, enables code modularity, and results in maintainable and expandable high-performance that will reduce the effort required to develop software for STUDSAT-2A/2B Mission.**

Index Terms- STUDSAT-2, Satellite, RTOS, Inter-Satellite Link (ISL), Twin Satellite Mission, OnBoard Computer (OBC).

TABLE OF CONTENTS

978-1-4799-1622-1/14/$31.00

1. INTRODUCTION

Real-time systems are omnipresent in almost every application. Some of the examples include industrial plant control, missile guidance, computer on-board an aircraft, path planning task of a robot, etc. Nowadays all systems are designed for real-time response to stimuli. A real-time system is subject to operational deadlines to respond to events.

A Real-time operating system (RTOS) is an operating system which is used to write good embedded software for complex systems in order to ensure the application meets its processing deadlines. Typically, deeply embedded real-time applications include a mix of both hard and soft real-time requirements. Soft real-time requirements are those that state a time deadline, which when breached, does not render the system useless whereas hard real-time requirements are those that state a time deadline, which when breached, results in absolute system failure.

In this paper we elucidate the technical justification for using FreeRTOS in STUDSAT-2. We present the implementation of FreeRTOS on STM32Cortex M4 controller to increase the functional integration and overall performance of the small satellite. Utilization of FreeRTOS simplifies software development, enables code modularity, and results in maintainable and expandable high-performance that will reduce the effort required to develop software for STUDSAT-2A/2B Mission.

The rest of the paper is organized as follows: Section 2 reviews the general OBC system design and its objectives, requirements and functions. Section 3 explains the OBCs hardware selection which includes the OBC Bus and Memory. Section 4 discusses the OBCs software which includes the significance of a RTOS for STUDSAT-2 and the implementation of FreeRTOS for the OBC. Section 5 describes the initialization of the OBC system with the help of flow charts which are self - explanatory. Section 6 is the concluding note which also discusses the scope for future.

Real-time software architecture and its implementation on a small satellite platform were explored and also, its pros and cons were discussed in [1]. Utilization of Atmel AT91SAM7SE-256 controller as the OBC for Nano/Pico satellite applications was studied [2]. Implementation of FreeRTOS on S3C44B0 processor was studied [3]. The con-

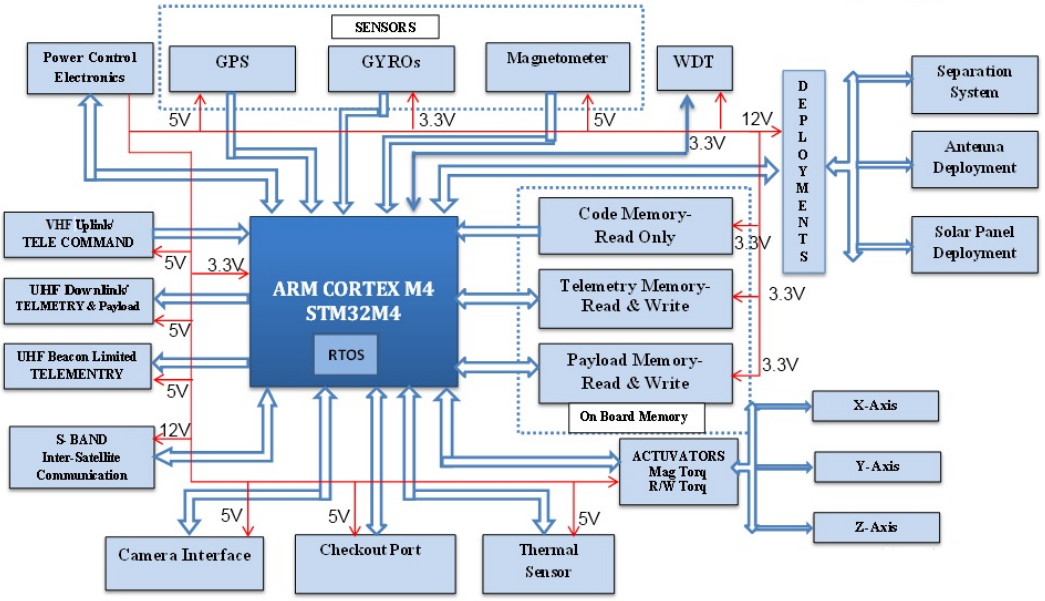

Figure 1. **Overall OBC System.**

cept of hierarchical scheduling using FreeRTOS was studied [4]. The implementation of Stream Control Transmission Protocol (SCTP) in RTEMS, an open source RTOS, for data communication over satellite networks was proposed in [5]. Design of a system for handling high speed image data from satellite and advanced image compression techniques was explained in [6]. A novel algorithm for GPS data acquisition, data processing and implementation for robot navigation was proposed in [7].

2. On-Board Computer

The On Board Computer (OBC) for STUDSAT-2 is ARM based Cortex- M4 microcontroller with DSP and FPU instructions combined to 168 MHz performance. The performance and the features of the chosen processor are based on functional, operational and interface requirements & protocols. The overall system architecture and requirements of the subsystems is accomplished by the open-source real-time operating system (RTOS) which support various types of peripheral interfaces, telemetry storage and data processing.

The block diagram shown in Figure 1 illustrates the various peripherals used for the project and the general system design. The various peripherals for the C&DH system consist of a set of memories, a microcontroller supervisor IC and temperature sensors at various locations of the satellite. A Read-/Write Non-Volatile Radiation Resistant Memory is used for storing the telemetry data. And for storing the images from the CMOS image sensor a fast and lard read/write memory is used. The controller will be supervised by a watchdog timer to catch runaway programs by resetting the controller. The I/O Board provides serial peripheral interface (SPI), I2C, USART, ADC etc. to communicate with sub-modules of other sub-systems.

Objectives

The Command and Data Handling (C&DH) System is the brain of the satellite. Its functions are:

- To develop a working hardware prototype of the STUDSAT OBC subsystem
- To perform the house keeping functions which includes logging the telemetry data in the memory periodically
- To detect and correct faults
- To implement the higher layers of the communication protocol
- To interface with the electrical and ground support equipment, the launcher, the ground segment, with the other two spacecraft in the constellation for telemetry possibly received over the Inter-Spacecraft Links

Requirements

OBC requirements are derived from mission requirements and are defined at system level requirements. OBC System requirements can be subdivided into three different groups:

- Functional Requirements: Tasks, subsystem control, task scheduling, data processing, communication subsystem and component level, data storage, etc.
- Operational Requirements: Operations modes (states), power modes, autonomy, failure detection isolation and recovery (FDIR), etc.)
- Interface Requirements and protocols: Compatibility with hardware and software interfaces, standards and protocols.

Functions of OBC

The tasks that are performed by the OBC can be listed as follows:

- Stabilize the satellite in the orbit
- Establish a communication link with ground station
- Monitor the health of the satellite
- To capture images of the earth
- To send the image to the ground station
- To communicate with the other satellite through Inter-Satellite Link

3. HARDWARE SELECTION

ARM CORTEX-M4F-based STM32F407VGT6 is used as main processor for device monitoring, status maintenance and interfacing with other peripherals. The Cortex-M4 processor has a wide variety of highly efficient signal processing features applicable to digital signal control markets and features necessary for the OBC. More information of STM32F4 Discovery board can be inferred from the data sheet.

Figure 2 shows an example of a figure spanning a single column.

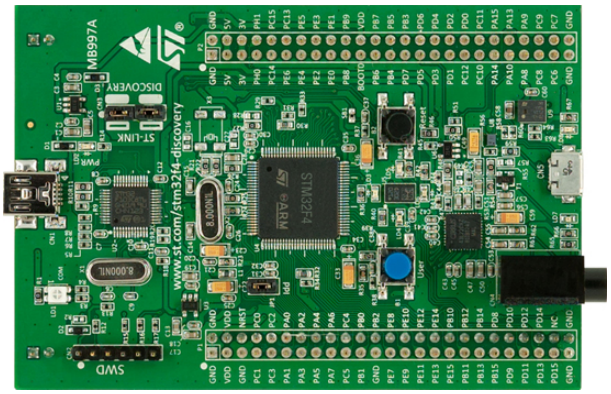

Figure 2. **STM32F4DISCOVERY.**

The project utilizes STM32F4 Discovery board as its onboard computer. A snapshot of the board is as shown in the above figure.

The OBC hardware has been designed with all possible interfaces and control circuitry with STM32F4 Discovery board.

OBC Bus

One Wire Interface Board—This board allows the onboard computer to act as the 1-wire bus master to control the devices connected to the bus. This board also contains a mounted thermal sensor for monitoring the temperature of the box containing the onboard computer as well as several additional circuit boards.

Magnetic Coil Control Board—This board provides the onboard computer with a GPIO connection to control the magnetic coils for attitude adjustments.

Reaction Wheels Control Board (PWM)—This board provides access for the data bus to the heaters and reaction wheels.

Magnetometer Interface Board (USART)—This board provides the serial conversion of magnetic field in xyz coordinates to allow the onboard computer to take readings over the data bus from the magnetometer.

GPS Interface Board (USART)— This board converts the LVTTL signal from the GPS unit to an RS-232 signal for connection to the onboard computer.

Communication (SPI/USART)—The communication system with transmitter and receiver for the radio frequency communication link and provides an SPI data link to the onboard computer.

Electronic Power Board (I2C)—This board handles the voltage regulation for the different subsystems.

Payload CMOS camera (DCMI)—Provided 10 bit parallel data bus with SDA and SCL data bus for storing image and reading image data from payload memory.

Inter-satellite Board (USART)—This board is designed to mount the ISL S-Band transceivers on and connects to the computer via a USART data link.

Memory

General Requirements—The general requirements of both the program and data memories are as follows:

- Low power consumption.
- Non-volatile, so that data is not lost in the event of a power failure.
- Serial bus interface to minimize space of bus on PCB and failure rate.
- The maximum number of write cycles expected is assumed to about 11,000.

This is based on the satellite passing the NASTRAC ground station six times a day, for the mission lifetime of five years, with each pass initiating a write cycle. Obviously, the memory should be able to write/erase more times than this.

Program Memory Requirements— The microcontroller has its own non-volatile program memory. However, on most devices this is small limiting the size of the program, and thus range of tasks, the microcontroller can run. In order to increase the flexibility of the mission, external program memory will be used. There is no upper limit to the required program memory size. During the mission, it is expected to have the ability to upload new programs to the spacecraft. Thus it is essential to use fast writing, non-volatile secure memory for the external program memory. Three types memories: EEPROM, Flash, and SRAM chosen because this computer will operate in atmosphere where radiation sometimes leads to bit-flips in temporary memory (Flash), which results in unpredictable software errors. A programmable safety timer, called a watchdog timer, resets the processor if bit-flip occurs, and the processor reloads the boot-software from EEPROM (24LC02 (EEPROM)) to Flash. It is essential that the boot-software always works correctly, and therefore must be stored in a memory, where bit-flips do not occur [8]. The Flash memory is used to store the main operating software and other subroutines, and a copy of this software will also be stored in EEPROM (permanent memory). The 512K SRAM temporary memory will be used to run the software and subroutines and also as a place to store the measured values from the payload sensors.

Data Memory Requirements (AT45DB161D)—For the initial payload, 500 Kbytes is needed to store one photo taken by the camera. The transmitter transmits at 9.6Kbits/s, and in the STUDSAT-2 low orbit, the maximum time it will be able to transmit data to the ground station will be around ten minutes. This means the maximum data that can be transmitted in one pass is 720 Kbytes. However, in practice, it will typically be about half of this. As one photo takes 500 Kbytes that means that in one pass over the ground station, either one photo of around 500 Kbytes or 500 Kbytes of telemetry data can be transmitted, but not at the same time. Thus, some orbits will store only a photo, while other orbits will store only telemetry data, at the discretion of the ground station operator [8].

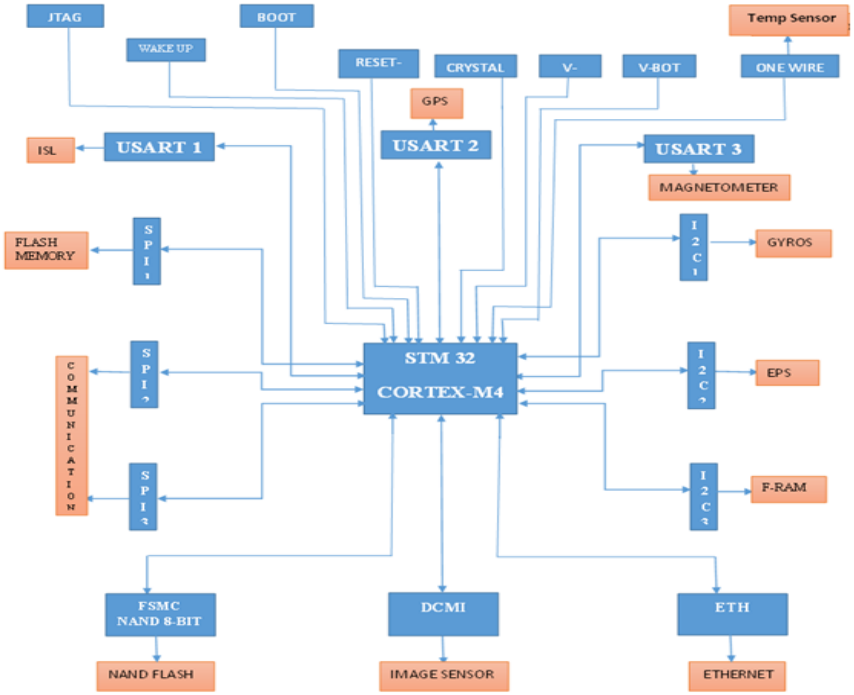

Figure 3. **Peripheral Interfacing.**

4. SOFTWARE

Figure 3 shows the STM32 M4 interfaced with various pheripherals like Camera, Magnetometer, Gyros, ISL Module 2.4GHz, Power System and Beacon Module for checking the overall software functionality (using FreeRTOS as operating system).

Requirement of Real-Time Operating System

When STUDSAT-1 was developed the idea of utilizing a real-time operating system did not strike us. However, after its launch we did come to realize how utilization of a RTOS could have significantly reduced the complexity of the code and saved a lot of time.

Exploring the prospects of using a RTOS for Project STUDSAT-2 led to the following advantages:

- Code Complexity: Utilization of RTOS leads to a significant reduction of the complexity of the code. By allowing the project to be decomposed into independent threads or processes, and using OS services such as message queues, mutexes, semaphores, event flags, etc. to communicate and synchronize, the project becomes more manageable.
- Time critical nature of the system: As discussed in the introduction section of this paper, the need to meet hard real-time deadlines is another major issue where RTOS could help. With a priority-based pre-emptive RTOS, the scheduler can help ensure deadlines are met.
- Testing: Testing of individual functions becomes easier when these functions are split into a number of tasks so that each task can be separately tested and verified. Task interfaces can be exercised without the need to add instrumentation that may have changed the behaviour of the module under test [10].
- Code reuse: Tasks facilitate code resue within a project thereby saving time in rewriting huge chunks of code again and again.Greater modularity and less module interdependencies facilitates code reuse across projects [10].
- Etcetera

Above are just few of the points which justify the use of RTOS for the project. A thorough study of the capabilities of a RTOS led to more such advantages of using a RTOS for the project.

In order to implement all the functions required by the OBC a real-time operating system is used. A real time operating system (RTOS) is basically a task-scheduling program that responds to events such as interrupts in real time, i.e. a few milliseconds at most. A good RTOS will hide the complexity of the underlying hardware from the tasks running, making tasks easier to develop.

An RTOS provides the following services to the tasks running:

- Decides which task to run next.
- Ensures that a task does not take up resources for too long, delaying other tasks.
- Handles hardware interrupts for tasks, and calls the appropriate task that is waiting for the interrupt.
- Passes messages between tasks so they can communicate with each other.

An important feature of the RTOS is multitasking. This allows several tasks to run as though they are running in parallel. Obviously this is impossible on most processors, and what actually happens is the tasks are allowed to run for short periods before switching to the next task. All important registers and data are saved between each task switch so

that the task may resume in exactly the same state it was before the switch next time it is called. More details on how multitasking and other such features can be implemented using FreeRTOS is discussed in the sections below.

Comparison & Feasibility Studies

One of the major requirements of Project STUDSAT-2 was an open-source RTOS (mainly for economic feasibility). Also, when an open-source RTOS could do the job and share the same capabilities as that of a non-open-source RTOS, then there is no need to incur additional costs. Following are the major aspects of a RTOS we looked at for carrying out comparison studies:

• Features: Basic features of a RTOS include tasks, task scheduling, queues, semaphores, memory management, etc. Although there are several more features, the said features constitute a minimum requirement.

• Size: Memory constraint is a big issue while considering using a RTOS. The microcontroller chosen for the project (STM32F4VGT6) required a small RTOS.

• Performance: One of the major requirements include that the RTOS take minimal time to boot. Memory allocation mechanism gives a clearer picture of the performance of the RTOS in the chosen hardware.

• Documentation & Support: Any RTOS has to have good documentation and technical support so that critical issues can be easily and professionally addressed.

• Cost & Licensing Model: What kind of license does the RTOS offer? Is it open-source? The definition of open-source may not be uniform across all vendors of RTOS. It is important to check what features are available as open source, that is, how much of the RTOS is truly open-source. These are few of the questions that need to be answered.

Since there are several RTOSes in the market which are open source it may not be possible to test each and every available open-source RTOS on the hardware due to time constraints. However, a comparison study is really important for the project.

Comparison studies require brainstorming sessions. The first step for comparison studies would be to lookup for the list of available open-source RTOSes. And then look up for comparison studies already performed and made available online. Based on the online reviews of several open-source RTOSes a select few can be rounded off. Then theoretical studies over each of these RTOSes could lead to rounding off to an even smaller number of RTOSes which makes it viable to test each of these on the hardware.

Nevertheless, it is upto the user to test and verify each of the open source RTOSes available for a specific application.

Keeping the above points in mind, we finalised on FreeRTOS. FreeRTOS is a good choice if the user is looking for a small OS (for microcontroller-based applications). Having tested quite a few like ChibiOS/RT (which lacked good documentation), uCOS-II (which lacked a robust multitasking feature (set by assigning same priority to several tasks in FreeRTOS)), etc., we were convinced that FreeRTOS provides us with all that is required for our project.

RTOSes keep getting updated with the latest features and thereby, there may or may not be a best choice for the application.

The section below discribes why we chose FreeRTOS over other RTOSes.

Why FreeRTOS?

Functionally, most small RTOSes do basically the same thing. So differentiation comes from non-functional attributes like quality control, IP infringement protection, design choices (at which point do you trade off size, speed, portability, etc.), ecosystem, user base, if there are options for commercial support should any legal team ever need it, etc.

A good comparison of various open-source RTOS for STM32 and Cortex-M3 MCUs (which is applicable for Cortex-M4 MCUs like STM32F407VGT6 also) can be found in [9].

Features—FreeRTOS has all the necessary features a RTOS generally has and is therefore suitable for the project. All of the features are also easy to implement.

Size—This depends on the compiler, architecture and RTOS kernel configuration. FreeRTOS is a relatively small application. The minimum core of FreeRTOS is only three source (.c) files and a handful of header files, totalling just under 9000 lines of code, including comments and blank lines. FreeRTOS running on an AVR has a footprint (the amount of ROM used) of approximately 4.4 kilobytes. RTLinux (RTOS microkernel that runs the entire Linux operating system), on the other hand, is relatively scalable. The Linux kernel can be stripped of functionality which is not needed. But even given that, the footprint is generally measured in megabytes (or in some rare cases hundreds of kilobytes). Similarly, QNX is another RTOS which seemed viable but is a "heavier" RTOS compared to FreeRTOS. A typical application takes 5 to 10 kilobytes of ROM space. STM32F407VGT6 has upto 1 Mbyte of flash memory and upto 192+4 Kbyte of SRAM including 64 Kbyte of CCM (core coupled memory) data RAM which makes it suitable for any FreeRTOS-based application. More details on the amount of RAM and ROM utilization of FreeRTOS can be found in [11].

Performance—Time taken for booting is one of the major issues that is to be taken into consideration while judging the performance of any RTOS. A detailed description of the each of the processes that are executed while booting and its timing in FreeRTOS can be found in [11]. Again, the performance depends on how efficiently the code is written and the hardware on which the RTOS is used. Although few of the small open-source RTOSes (like ChibiOS/RT) may closely compete with FreeRTOS in performance, other aspects like documentation & support, features, etc. make FreeRTOS a more attractive choice.

Documentation & Support—FreeRTOS is a truly cross platform defacto standard. It supports 34 architectures (that is, architectures, not devices) and 18 tool chains till date. Even the time investment required can be lower than with commercial solutions. Availability of documentation for several microcontroller architectures, swift technical support and massive user base makes FreeRTOS a good choice and reins it supreme when compared to ChibiOS/RT. Also, since FreeRTOS has a massive user base, the solutions to most of the common technical problems are available readily online.

Cost & Licensing—License details of FreeRTOS can be found in [12]. FreeRTOS is 'moderated open source', which means that, when FreeRTOS is downloaded, the code is truly free and open source to the user. However, contributions back to FreeRTOS are made publicly available while simultane-

ously kept separate from the 'official' code. This allows the FreeRTOS project to strictly control the quality of the code, ensuring a robust product. It also provides assurance to end users that their open source choice has no unintentional or unknown IP contamination. The business model allows users the freedom to access, use, evaluate, distribute, etc. the code, completely free of charge. However, they have the security of knowing there is a third party infrastructure there should they ever need commercial licensing or support contracts.

QNX RTOS is not open-source and therefore can be used only for a limited time period. The project duration is long and therefore limited time period softwares are not feasible for the project. Micrium's uCOS is another example of a RTOS which competes closely with FreeRTOS. Although uCOS is known for its high reliability (even more than FreeRTOS as it has been used in several space research projects), it is not open-source.

Another speciality of FreeRTOS includes code quality, which can be quantified in a number of ways. Independent analysis of the code has proven its robustness and quality. FreeRTOS is professional grade and robust, yet still free. Users can have the confidence in knowing that SafeRTOS, which originated from FreeRTOS, has been independently certified by TUV SUD for use in safety critical systems.

These are all the points where FreeRTOS scores extremely highly. The design choice is subjective of course, and different solutions are best suited to different applications, so there is no "best" and no "right or wrong" for that category.

FreeRTOS has become the de facto standard for microcontrollers, and comes top in class in the EE times embedded market surveys for most used and most considered for next project, shows independently that the model works extremely well.

Implementation of FreeRTOS

FreeRTOS software download consists of demo, source and license files. Demo file consists of example projects for a variety of boards. For OBC either one of these sample projects can be modified or a project can be created from the scratch using a project management software. Instead of modifying an existing demo (which makes use of IAR Embedded Workbench software) we chose to create a project from the scratch as it allows us to use any software we choose to integrate with FreeRTOS. We chose uVision Keil for the OBC. Creating a project from the scratch allows us to organize/group all the files and also allows us to add or delete files as per our requirement. On the other hand a pre-configured demo includes even those files which are not required for the OBC.

Firstly, we listed out the tasks required by the OBC. Table 1 illustrates the list of tasks devised for the system [16]. There are three types of tasks based on the timing in this software architecture [16].

• Periodic Task: This task must take action on a regular basis.

• Periodic Update Task: This task will collect data and place it in a global memory area for use by other tasks on regular basis.

• Aperiodic Task: This task will run as a result of some external command respond, such as from the ground.

The tasks are assigned a priority based on criticality of the task and execution time. Each task will have a message queue assigned to it and will block on the message queue, with a timeout equal to its period. If the task is aperiodic, it will have an infinite timeout. Using this mechanism, there are two ways to activate a sleeping task: send a message to its message queue, which will be processed immediately (as long as another, higher priority task isnt already active) or wait for the timeout period to expire (assuming the task is periodic). The task manager is the FreeRTOS real-time operating system (RTOS). The task manager is responsible for context switching between active tasks, based on their priority.

Table 1. **Task Interface**

Task	Task Name	Timing
Task Manager		Continuous
Camera	tCamera	Periodic Update
GPS	tGPS	Periodic Update
Attitude Determination	tAttDetr	Periodic
Inter-Satellite Communication	tISL	Periodic
STUDSAT Beacon	tBeacon	Periodic
Power	tPower	Periodic
Mode Manager	tModeMgr	Periodic
Mission Timeline Manager	tMTM	Aperiodic
Uplink Communication	tUplink	Aperiodic
Downlink Communication	tDownlink	Aperiodic

• Camera (tCamera) : This task is responsible for capturing image and serve buffer to compress on board image using JPEG2000 algorithm. Following threads run in the Camera Task (tCamera):

```
Cam_GetStatus();
Cam_set_mode();
Cam_TakePicture();
Cam_Jpeg_Compress();
Cam_Transmit_Image_to_Buffer();
Cam_Rec_Image_from_Buffer();
Cam_LowPower();
Cam_PowerOK();
Cam_telemetry.();
Cam_PictureData();
```

• Intersatellite Link (tISL): This task is responsible for communication with the other satellite through Inter-Satellite Link (2.4Ghz) and establish communication between two satellites. The communication would be master salve configuration. OBC will determine when the data needs to be gathered for the telemetry buffer and when to communicate with the slave satellite. Following Threads run in Intersatellite Link (tISL):

```
ISL_GetStatus();
ISL_Get_TelemBuffer();
ISL_Get_IMIBuffer();
ISL_Get_ImageData();
ISL_Send_Cross Send();
ISL_GetResponse();
ISL_Get_Cross_GPS();
ISL_End_Link();
```

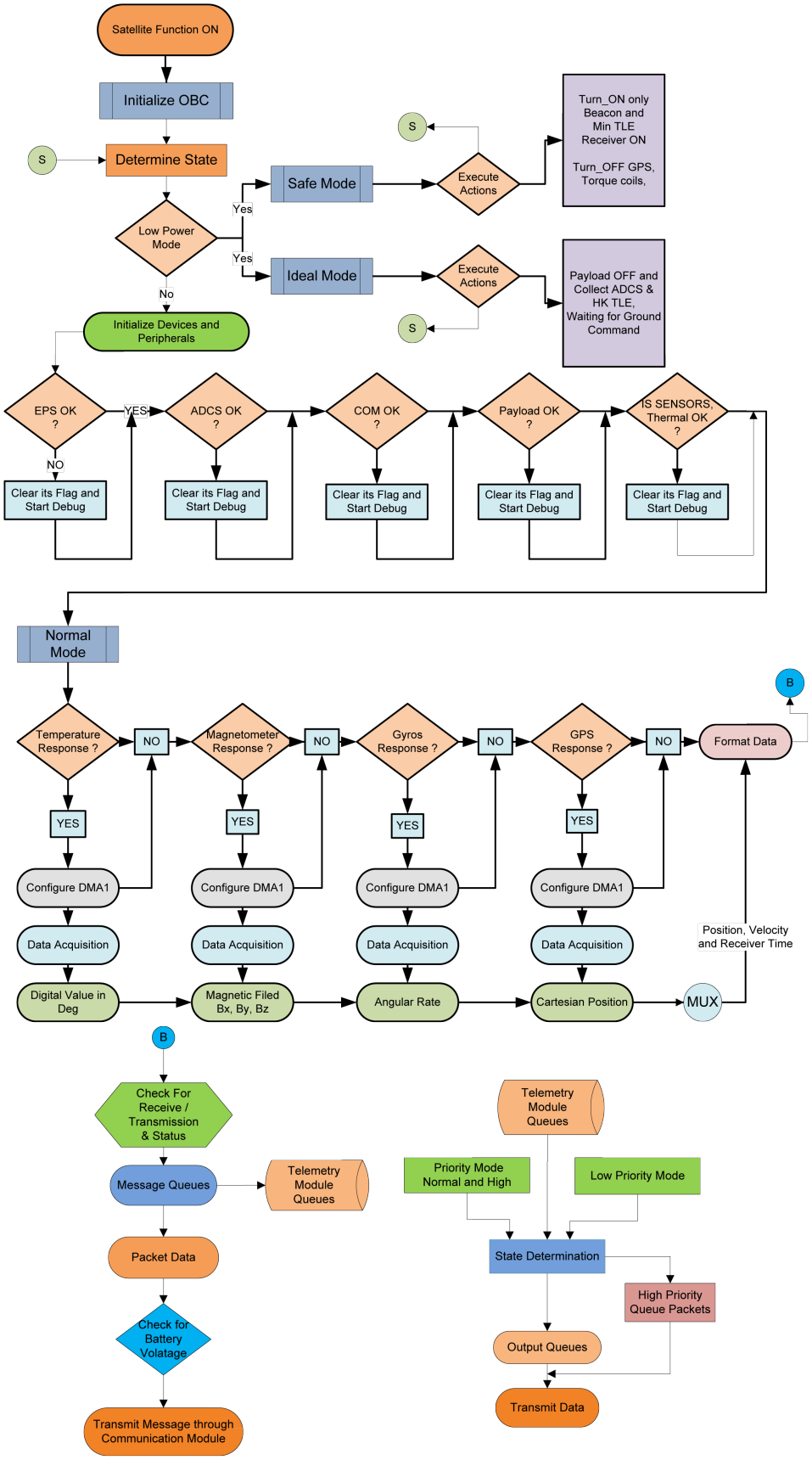

Figure 4. Flowchart illustrating initialization of OBC and device startup.

• Communication Tasks: Communication tasks can be classified as Uplink Task and Downlink Task. These tasks are responsible for establishing a communication link with ground station for both uplink and downlink communication.
Uplink Communication (tUplink): This task is responsible for receiving commands from ground station and parsing the data to execute suitable actions in the satellite.
Downlink Communication (tDownlink): This task is responsible for sending telemetry data and payload data that is requested by ground station. The message should be formatted in AX.25 protocol for decoding message in ground station.
The OBC_Comm_Downlink performs the task of forming packets of the information to be sent.

OBC_Comm_Downlink():	OBC_Uplink():
Com_GetStatus	Com_RxNewSoftware
Com_PictureData	Com_RxSyncData
Com_LOGData	Com_RxCommand
Com_TxSyncData	Com_RxDtmf_Command
Com_LowPower	Com_RxSetDebugOut
Com_PowerOK	Com_RxTransmit_Image
Com_RxNewFlightPlan	Com_RxNewAcsParam
Com_RxKeplerElements	Com_COMStatus
Com_RxTimeSync	

• Beacon Task (tBeacon): The satellite will send beacon signals to the ground station. The Beacon signal contains information like the satellite ID. This information is in the form of packets, which are formed according to the AX.25 protocol.

• Power (tPower): This task is responsible for handling all power related tasks on the OBC. These tasks include software overcurrent protection and power switch control with data flow control command for Regulator Current and Voltage bus. Following threads run in Power (tPower):

```
Power_GetStatus();
Power_SetBootMode(); Power_SystemReset ();
Power_TurnOnSubsytem();
Power_TurnOffSubsytem ();
Power_reset_theDog();
Power_BasicBeaconOff();
Power_telemetry();
Power_ErrorShutDown();
Power_PCUPowerOK(); Power_PCULowPower();
```

• Attitude Determination (tAttDetr): This task is responsible for updating Keplerian elements, performing mathematical calculations and provide actuation and determination. Data acquisition of magnetometer XYZ position, gyro XYZ position, sun vector, orbit propagation and error detection. Following Threads run in Attitude Determination (tAttDetr):

```
ADCS_GetStatus
ADCS_Telemetry
ADCS_UpdateKeplerElements
ADCS_navigation_guidance
ADCS_SunData
ADCS_SwitchADCSMode
ADCS_Lowpower
ADCS_PowerOk
ADCS_TimeSynchronized
```

• Mode Manager (tModeMng): This task is responsible for maintaining the mode state of the satellite. The mode manager will determine the state based the request and conditions received from the OBC which are associated with different subsystems like EPS, COMM, ADCS and thermal systems [16].

• Mission Timeline Manager (tMTM): This task is responsible for managing the timeline for future events [16].

5. INITIALIZATION OF OBC & DEVICE STARTUP

The data ow shown in gure 4 describes the initialization of OBC and device startup required to send telemetry data from the sensors to communication module. All the communication between tasks occur using message queues. The ight software is implemented with FreeRTOS for operation of command and data handling.

Satellite Modes of Operation

The ight software is implemented with three distinct spacecraft modes ->Safe Mode ->Ideal Mode->Normal Mode.

Apart from the normal operation of the satellite the following operating modes are defined for the extended convenience & optimum functioning of the satellite and for power saving functions.

Start-up Mode—This will be a one-time operational mode which will be active only after a few hours after the launch of the Satellite into the orbit.

Safe Mode (Low power) —In this mode only the essential modules are powered ON and only the Housekeeping & Maintenance functions are done in this mode.

Emergency Mode—The satellite is put under emergency mode during Extreme operating conditions caused due to either partial module failure or low power availability, In this mode only the Power system and the Beacon is kept on and rest of the modules are suspended until the errors are rectified or the normal operating conditions are restored.

Communication Priority Mode— In this mode the priority of operation is given to the Communication modules, all other modules are put under minimal operative mode so that maximum power is provided to the Communication module for the duration in which the satellite happens to be in the Cone of Window.

Payload Priority Mode—In this mode maximum CPU usage and power is given to the payload requirements to ensure smooth and uninterrupted payload mission. The satellite will operate in this mode during the exclusive payload operation.

Debug Mode—This mode is mostly activated along with the Emergency mode or Safe mode. This mode is activated during the maintenance operation, where the Runtime software errors can be debugged by up-command.

6. SOFTWARE TESTING & SIMULATION

The softwares used for OBC are as follows:

• RTOS: The OBC makes use of FreeRTOS. It is GPL Licensed operating system that targets microcontroller. It supports various degrees of multitasking, ranging from a preemptive scheduler & co-operative multitasking to coroutines.

• Compiler and Debugger: The Vision IDE from Keil combines project management, make facilities, source code editing, program debugging, and complete simulation in one

powerful environment which makes it suitable for the project. The ST-LINK/V2 tool can be easily used for JTAG and SWD interface debugging and programming.

Testing

A code was written and executed on STM32F4DISCOVERY board to test few task management features of FreeRTOS. The total flash footprint came upto approximately 11 Kbyte which includes FreeRTOS code, application code and library code. Few code snippets are listed below that show different task management features of FreeRTOS.

The syntax for creating a task using FreeRTOS is shown below. Detailed explanation can be referred from text books on FreeRTOS.

```
portBASE_TYPE xTaskCreate( pdTASK_CODE
    pvTaskCode,
const signed char * const pcName,
unsigned short usStackDepth,
void *pvParameters,
unsigned portBASE_TYPE uxPriority,
xTaskHandle *pxCreatedTask
);
```

Following is a code snippet which shows simple task creation using FreeRTOS.

```
void emptyTask(void *pvParameters)
{
      char *pcTaskName = "Empty Task is
          running\n"; //To display which
          task is running
      for(;;) //Infinite loop
      {
            USART_puts(USART1, pcTaskName);
                //For printing on a terminal
                software through USART1
            Delay();
      }
}
int main( void )
{
      init_USART1(9600); // initialize
          USART1 @ 9600 baud
      USART_puts(USART1, "USART
          Initialization complete !\r\n");

      //Task Creation
      xTaskCreate(emptyTask, ( signed char *
          ) "EmptyTask", 240, NULL,
          EmptyTaskPriority, NULL);

      //Execute the task
      vTaskStartScheduler();// This should
          never return.
      // Will only get here if there was
          insufficient memory to create
      // the idle task.
      return 0;
}
```

The output looks like the following:

```
USART Initialization complete !
Empty Task is running
Empty Task is running
Empty Task is running
```

Following is a code snippet which shows multitasking using FreeRTOS.

```
void tUplink (void *pvParameters)
{
      /* Variable declarations and other
          initializations*/
      const char *pcTaskName = \r\ntUlink
          task running\r\n;
      uint8_t i=0;
      /*  .*/

      for( ;; ){
            /* Print out the name of this
                task. */
            USART_puts(USART1, pcTaskName);
            /* tUplink  code  */
      }
}
void tDownlink (void *pvParameters)
{
      /* Variable declarations and other
          initializations*/
      const char *pcTaskName = \r\ntDlink
          task running\r\n;
      uint8_t i=0;
      /*  .*/

      for( ;; ){
            /* Print out the name of this
                task. */
             USART_puts(USART1, pcTaskName);
            /* tDownlink code */
      }
}

int main( void )
 {
      init_USART1(9600); // initialize
          USART1 @ 9600 baud
      USART_puts(USART1, "USART
          Initialization complete !\r\n");

      //Task Creation
      xTaskCreate( tUplink, "tUplink",
          configMINIMAL_STACK_SIZE, NULL, 1,
          &hUplink ); //Note that the
          priority for both the tasks are
          set the same (whisi is 1)
      xTaskCreate( tDownlink, "tDownlink",
          configMINIMAL_STACK_SIZE, NULL, 1,
          &hDownlink ); //This means that on
          execution, both tasks will be
          running alternately which makes it
          look like its running
          simultaneously
      //Similarly, any number of tasks can
          be set the same priority for
          multitasking

      //Execute the task
       vTaskStartScheduler(); // This should
           never return.

       // Will only get here if there was
           insufficient memory to create
       // the idle task.
      return 0;
}
```

The output looks like the following:

```
USART Initialization complete !
tUlink task running
```

```
tDlink task running
tUlink task running
tDlink task running
tUlink task running
tDlink task running
```

Following is a code snippet which shows dynamic priority scheduling using FreeRTOS. An initial priority for the task is assigned by the uxPriority parameter of the xTaskCreate() API function. The priority can be changed by using the vTaskPrioritySet() API function. The maximum number of priorities can be set by defining a compile time configuration constant in the configMAX_PRIORITIES within FreeRTOSConfig.h header file. Higher the configMAX_PRIORITIES value the more RAM the kernel will consume.

```
void sampleTask1(void *pvParameters)
{
      const char *pcTaskName = "sampleTask1
          is running\n";
      unsigned portBASE_TYPE uxPriority;
      uxPriority = uxTaskPriorityGet(NULL);
      for(;;) {
            Delay();
            USART_puts(USART1, pcTaskName);
            USART_puts(USART1, "About to
                raise the sampleTask2
                priority...\n");
            vTaskPrioritySet(xTask2Handle,
                (uxPriority+1));
      }
}

void sampleTask2(void *pvParameters)
{
      const char *pcTaskName = "sampleTask2
          is running\n";
      unsigned portBASE_TYPE uxPriority;
      uxPriority = uxTaskPriorityGet(NULL);
      for(;;) {
            Delay();
            USART_puts(USART1, pcTaskName);
            USART_puts(USART1, "About to
                lower the sampleTask2
                priority...\n");
            vTaskPrioritySet(NULL,
                (uxPriority-2));
      }
}

int main( void )
{
      init_USART1(9600); // initialize
          USART1 @ 9600 baud
      USART_puts(USART1, "USART
          Initialization complete !\r\n");

      //Task Creation
            xTaskCreate(sampleTask1, (
                signed char * )
                "sampleTask1", 240, NULL,
                sampleTask1Priority, NULL);
            xTaskCreate(sampleTask2, (
                signed char * )
                "sampleTask2", 240, NULL,
                sampleTask2Priority,
                &xTask2Handle);

      //Execute the task
      vTaskStartScheduler(); // This should
          never return.
       // Will only get here if there was
           insufficient memory to create
       // the idle task.
      return 0;
}
```

The output looks like the following:

```
USART Initialization complete !
sampleTask1 is running
About to raise the sampleTask2 priority...
sampleTask2 is running
About to lower the sampleTask2 priority...
sampleTask1 is running
About to raise the sampleTask2 priority...
sampleTask2 is running
About to lower the sampleTask2 priority...
```

Following is a code snippet which shows task periodicity using FreeRTOS.

```
void emptyTask(void *pvParameters)
{
      char *pcTaskName = "Empty Task is
          running\n"; //To display which
          task is running
      for(;;) {
            USART_puts(USART1, pcTaskName);
                //For printing on a terminal
                software through USART1
            Delay();
      }
}

void periodicTask (void *pvParameters)
{
      const char *pcTaskName = "Periodic
          Task is running\n";
      portTickType xLastWakeTime;
      xLastWakeTime = xTaskGetTickCount();

      //Lets say the mode of execution is as
          per the number of executions, that
          is, after a said number of
          executions the periodic task will
          execute
      x = (number_of_executions * y);
          //number_of_executions=3
      //The code can also be modified to
          execute a periodic task after a
          said time rather than the number
          of executions

      for(;;) {
            Delay();
            USART_puts(USART1, pcTaskName );
            vTaskDelayUntil( &xLastWakeTime,
                (x/portTICK_RATE_MS));
            Delay();
      }
}

int main( void )
{
      init_USART1(9600); // initialize
          USART1 @ 9600 baud
      USART_puts(USART1, "USART
          Initialization complete !\r\n");

      //Task Creation
      xTaskCreate(emptyTask, ( signed char *
          ) "EmptyTask", 240, NULL, 1, NULL);
```

```
            xTaskCreate(periodicTask, (
                signed char * )
                "PeriodicTask", 240, NULL,
                2, NULL); //Note that the
                periodic task has a higher
                priority

        //Execute the task
        vTaskStartScheduler(); // This should
            never return.

         // Will only get here if there was
             insufficient memory to create
         // the idle task.
        return 0;
}
```

The output looks like the following:

```
USART Initialization complete !
Empty Task is running
Empty Task is running
Empty Task is running
Periodic Task is running
Empty Task is running
Empty Task is running
Empty Task is running
Periodic Task is running
Empty Task is running
```

It is worth mentioning that as a subset of the not running state, the aperiodic tasks are in the blocked state, that is, these tasks are waiting for an event to occur and the periodic update tasks are in the ready state, that is, they are in the not running state but are ready to run.

Software Test Logging

Each module should have a minimum of two testing logs on file after completion. Due to the nature of software development and debugging (code, compile, test, repeat, etc) the testing logs will act as a debugging completed log which shows that the software is completed and ready to be integrated into the system.

Full software system tests will only begin once initial versions of modules are completed and the full system can be assembled as a functional software package. The system test will attempt to check every possible system state and execution path. This will ensure that the individual modules do not cause functionality problems with each other or that there are multiple modules attempting to use the same resource at the same time.

Error Checking

Error checking in the hardware can help to eliminate additional software. However, commercial-off-the-shelf (COTS) products that utilize hardware solutions are cost prohibitive for this project. A single processor board for both satellites is utilized because of the higher power requirements of a multiprocessor plan. Onboard error checking systems are adequate for our need. Software redundant storage system has been designed and tested for the computers. The system consists of a read, write, and validate module that implements a three-copy data storage system to fix any errors that may occur during storage. This system has been found to be error free and functions properly on a Compact Flash memory device.

7. Conclusion & Scope for Future

The overall system architecture and requirements of the subsystems is accomplished by the open-source real-time operating system (FreeRTOS) which supports various types of peripheral interfaces, telemetry storage and data processing. FreeRTOS was chosen for the project after performing several comparison and feasibility studies which have been discussed in the earlier sections.

The only drawback of FreeRTOS is that tasks are designed to enter in infinite loops. Satellite function tasks should not be in infinite loop. The tasks should function in the specified conditions without going into infinite loops. To overcome this issue implementation of special techniques to avoid infinite loops is required. Currently, we are working on implementation of certain techniques described in [17] to avoid infinite loops. This would help to modify the existing FreeRTOS code for STUDSAT-2.

Results

Figure 6 shows the image data received from the hardware setup shown figure 5. Camera is interfaced with STM32 and commanded to take picture using UHF communication module. The image stored is sent via 2.4 GHz ISL module. The hex values received for the ISL module is converted to JPEG Image format to display in PC and this experiment carried out to prove the concept of transmitting image from one satellite to other satellite via ISL. The results shows the decoded beacon signal received from the overall satellite subsystems interface as seen in figure 5. The data is received by NASTRAC (Nitte Amateur Satellite Tracking Center) using MiXW (a software TNC program).

Acknowledgments

The authors sincerely thank Dr. M. Annadurai Chairmen, SRC for Small Satellite, ISRO and Shri Amereshwar Knened Project, Director, Small Satellite Project and all the scientists across various centers of ISRO and ISRO Satellite Centre (ISAC) for their motivation and having provided the authors with all the technical and non-technical support. The authors extend their sincere gratitude to Dr. Jharna Majumdar, Prof. CSE and Dean (R&D) of the Nitte Meenakshi Institute of Technology, the Chief Project Coordinator of the Project STUDSAT, for guiding the STUDSAT Team and playing a key role in initiating the Project STUDSAT-2. The authors render special thanks to Mr. Loganathan M., Director Space Program Nitte Meenakshi Institute of Technology, for his support and guidance. The authors are grateful to Visvesvaraya Technological University (VTU), Belgaum for their Monetary support. The authors thank the Management of Nitte Meenakshi Institute of Technology, the lead college of the STUDSAT project for their wholehearted support for the project. The authors thank Prof. N.R. Shetty, Visionary, Advisor, Nitte Meenakshi Institute of Technology and Dr. H.C. Nagaraj, Principal, NMIT for providing excellent infrastructure in the college for the execution of the project and the encouragement given to the team. The authors thank the Principals and the managements of the 6 consortium colleges for their support. The authors also thank Richard Barry, the author of FreeRTOS, and Real Time Engineers Ltd. for technical support.

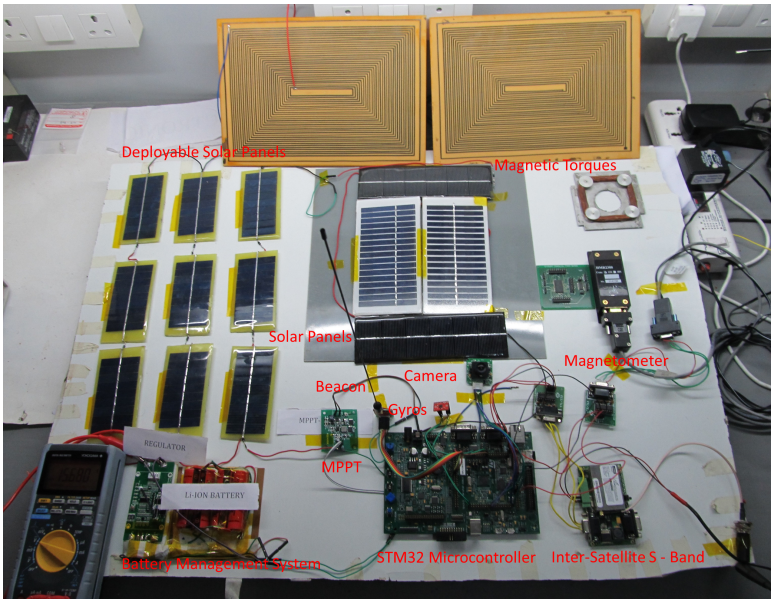

Figure 5. Complete hardware test setup.

Figure 6. Image Received from the interface.

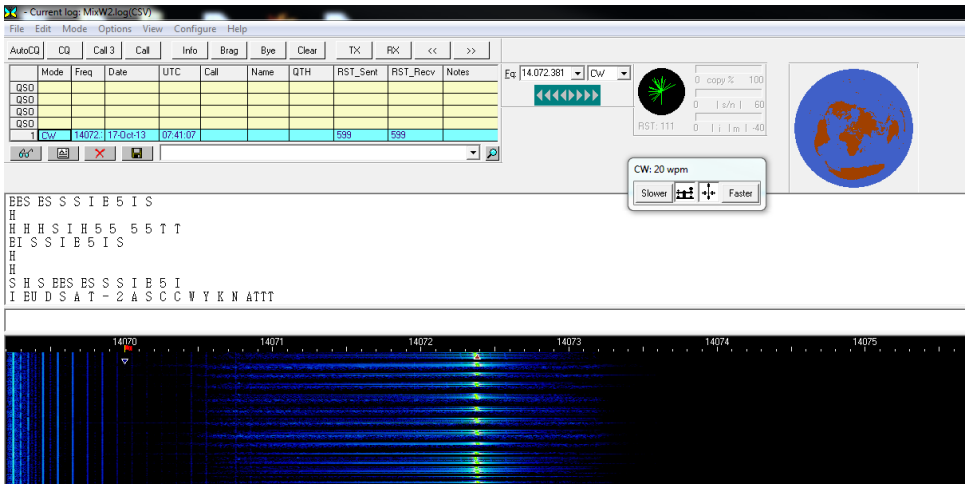

Figure 7. Beacon Received from UHF Beacon Board.

REFERENCES

[1] Caitlyn M. Cooke, Implementation of a Real-Time Operating System on a small Satellite Platform, in Space Grant Undergraduate Research Symposium, Colorado Space Grant Consortium, Boulder, CO, 80309.

[2] Nishchay Mhatre, Mohit Karve, Gautam Akiwate, Shravan Aras, Mr. Sanjeev Krishnan, Varad Deshmukh, Rahul Bedarkar, A MODULAR, GENERIC, LOW-COST ON-BOARD COMPUTER SYSTEM FOR NANO/PICO SATELLITE APPLICATIONS, 62nd International Astronautical Congress 2011, vol.,no.,pp. IAC-11,E2,3,5,x10570 2011.

[3] Shancao Niu; Jing Zhang; Bin Wang; Xiaodong Fu, "Analysis and Implementation of Migrating Real-Time Embedded Operating System FreeRTOS Kernel Based on S3C44B0 Processor," Information Science and Engineering (ISISE), 2012 International Symposium on , vol., no., pp.430,433, 14-16 Dec. 2012.

[4] Rafia Inam, Jukka Mki-Turja, Mikael Sjdin, Moris Behnam, Hard Real-time Support for Hierarchical Scheduling in FreeRTOS*,7th annual workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT 11), p 51-60, Porto, Portugal.

[5] Rahman, M.S.; Atiquzzaman, M.; Ivancic, W.; Eddy, W.; Stewart, D., "Implementation of SCTP in an open source Real-Time Operating System," Military Communications Conference, 2008. MILCOM 2008. IEEE , vol., no., pp.1,7, 16-19 Nov. 2008.

[6] Sarma, T.C.; Srinivas, C.V., "Design and Implementation of High Bit Rate Satellite Image Data Ingest and Processing System," Signal Processing, Communications and Networking, 2007. ICSCN '07. International Conference on , vol., no., pp.149,152, 22-24 Feb. 2007.

[7] Sethu Selvi, S.; Iyer, N.R.; Sandeep, G.S.P., "A facile approach to GPS navigation in unmanned ground vehicles," Advances in Technology and Engineering (ICATE), 2013 International Conference on , vol., no., pp.1,6, 23-25 Jan. 2013.

[8] DTUsat On Board Computer (OBC) System - DTUsat, http://etd.dtu.dk/thesis/200728/imm5238.pdf

[9] Open source Real time Operating Systems for the STM32 and Cortex m3 MCu's, https://sites.google.com/site/stm32discovery/stm32-resources-and-links/open-source-real-time-operating-systems-for-the-stm32-and-cortex-m3-mpus

[10] Why Use FreeRTOS, http://www.freertos.org/FAQWhat.html#WhyUseRTOS

[11] FreeRTOS FAQ - Memory Usage, Boot Times & Context Switch Times, http://www.freertos.org/FAQMem.html

[12] FreeRTOS License Details, http://www.freertos.org/a00114.html

[13] A NEW DESIGN APPROACH OF SOFTWARE ARCHITECTURE FOR AN AUTONOMOUS OBSERVATION SATELLITE Jkrorne Gout, Sara Fleury, LAAS-CNRS

[14] Command and Data Handling Subsystem Design for the Ionospheric Observation Nanosatellite Formation (ION-F) John D. Jensen, Utah State University Dr. Charles M. Swenson (Advisor), Utah State University

[15] USU Software Document AOE

[16] Ion-F Software Architecture AOE

[17] Super Simple Tasker (SST) by Micro Samek and Robert Word in Embedded System Design, 2006

BIOGRAPHY

Bheema Rajulu *working as Research Scholar for Student Small Satellite Project - Project STUDSAT-2, received a B.E in Electical and Electronic Engineering from Nitte Meenakshi Institute of Technology under Visweswaraya Technological University, Belgaum, India in July 2011. He was the core team member of the Project-STUDSAT-1, India's first Pico satellite, which was successfully flown into the orbit on July 12th, 2010 by PSLV. Now he is leading Project STUDSAT-2, India's first Twin Satellite Mission aiming to prove Inter-Satellite Communication. He is the Co-author of several papers out of which one has won the ESSTA (Emerging Scenarios in Space Technology") Best paper Award.*

Sankar Dasiga *has obtained his Bachelors Degree in ECE from Bangalore University in 1984 and Masters Degree in Electronics and Instrumentation from REC, Warangal in 1988. He has 22Years experience in Embedded Systems and Semiconductor Industry. His industry experience includes working in Macmet, Siemens, Philips, Wipro, CISCO, and NXP. He is currently working as a Professor in Department of ECE, Nitte Meenakshi Institute of Technology, Bangalore from 2010. He is a Deputy Project Co-ordinator for Project STUDSAT-2.*

Naveen Rajaraman Iyer *is a final year undergraduate student pursuing Bachelor of Engineering in Instrumentation Technology, in M S Ramaiah Institute of Technology, Bangalore, India. He is a robotics enthusiast who has worked on a variety of projects, conducted workshops on robotics and has also represented his institute in one of the worlds most prestigious robotics competitions (IGVC 2012). Additionally, he has coauthored a paper, titled A facile approach to GPS navigation in unmanned ground vehicles which was published in an IEEE conference, International Conference on Advances in Technology and Engineering (ICATE 2013). He is currently working on research projects in Centre for Cryogenic Technology (CCT) and Department of Electronic Systems Engineering (DESE), Indian Institute of Science (IISc), Bangalore, India.*