Design and Analysis of RTOS and Interrupt Based Data Handling System for Nanosatellites

Akshit Akhoury
Manipal Institute of Technology
Manipal Academy of Higher Education
Manipal, Udupi-576104
7759844684
akshit.akhoury@gmail.com

Arun Ravi
Manipal Institute of Technology
Manipal Academy of Higher Education
Manipal, Udupi-576104
9886715144
arunravi789@gmail.com

Krishna Birla
Manipal Institute of Technology
Manipal Academy of Higher Education
Manipal, Udupi-576104
8674916754
krishnabirla16@gmail.com

Shaleen Kalsi
Manipal Institute of Technology
Manipal Academy of Higher Education
Manipal, Udupi-576104
7204517521
shaleen.kalsi@gmail.com

Rohit Sarkar
Manipal Institute of Technology
Manipal Academy of Higher Education
Manipal, Udupi-576104
9765268459
rohitsarkar5398@gmail.com

Subhojit Ghorai
Manipal Institute of Technology
Manipal Academy of Higher Education
Manipal, Udupi-576104
9106954307
subhojit04ghorai@gmail.com

***Abstract*—In this paper, we describe the design and working of the data handling system of a Nanosatellite that houses three interconnected microcontrollers, each present on a different PCB. Each microcontroller handles and performs a set of tasks to ensure the smooth and proper functioning of the satellite. A brief description of the evolution of the system organization and the motivation behind the choice of the microcontrollers has been provided. An in-depth explanation of the tasks and their distribution among the three microcontrollers follows. The scheduling of jobs on two of the microcontrollers is brought about through the use of a Real-Time Operating System (RTOS), Micrium OS-III, which allows the system to be sensitive to the priorities and time constraints of each task. An in-depth qualitative analysis of the application of the RTOS has been presented along with a vigorous quantitative analysis through the use of Segger SystemView and the Sampled Graph feature in IAR. In contrast to this OS-based implementation, the third microcontroller is run and controlled purely through interrupts from the other two processors. The paper explains the use of a partial OS based and partial interrupt based task switching model and lists out the advantages and limitations of the same. The paper also describes the various stages involved in the onboard processing of images obtained from the thermal camera, which includes image compression and data encoding algorithms before transmitting that help in reducing data loss during transmission and allow error detection and correction upon reception of the payload data.**

TABLE OF CONTENTS

978-1-5386-6854-2/19$31.00 ©2019 IEEE

1. INTRODUCTION

A satellite system requires extensive data acquisition and processing to be done in real time which is done by the onboard data handling system (ODHS). The onboard data handling system provides the software to synchronize the various hardware components and the activities of other subsystems such as Communications (Comms.), Attitude determination and control (ADCS), Electronics and power (EPS), etc.

This paper describes the implementation of an RTOS and interrupt based software model for the onboard data handling system of a nanosatellite whose primary mission involves the urban heat mapping of the Indian Subcontinent using a Thermal Imaging Camera. The satellite also consists of a secondary payload, an electro-dynamic tether that will deorbit the satellite after mission duration. The paper demonstrates the efficient functioning of the system by explaining the mechanisms of inter-processor communication, task synchronization, and scheduling, and various error management schemes implemented. The described system handles the tasks of controlling satellite mechanisms, including antenna deployment, running the algorithms to control the attitude of the satellite, and collecting payload data (from a thermal camera). It handles the processing and transmission of payload data and health data (Beacon) to the ground station using two separate communication links.

Section 2 of the paper describes the system architecture, which includes the evolution of the system over time, the tasks performed by each processor, the scheduling algorithm and the communication mechanism between the processors. Section 3 gives a comparison between a traditional RTOS based system and a system using both an RTOS and interrupts for data handling. Section 4 presents future work which can be done on this topic and the lessons learned.

2. System Architecture

System Evolution

The satellite was initially a single processor system, and the processor in use was an MSP430. The MSP430 was chosen due to its space legacy, low price and low power requirement (around 0.00155 watts). Another MSP430 was later added, as a single processor system proved insufficient in handling the various tasks which were needed to be carried out by the satellite. To improve the efficiency of the dual processor system, distribution of the tasks among the processors was necessary. The primary processor was responsible for handling payload data transmission and processing, along with running the attitude control and power management algorithms, while the secondary processor was responsible for transmission of beacon data, processing telecommands and mechanism handling (deployment of antennas, the opening of tether door and tether deployment) for the satellite. Later on, the MSP430, responsible for payload data handling (the primary processor) was replaced by an STM32F2, this was because the MSP430 did not have a DCMI interface to receive data from the camera which is the primary payload.

The pointing accuracy of the satellite had to be improved to enable the camera to take pictures of the region of interest. To improve the pointing accuracy of the satellite Reaction wheels needed to be added. The reaction wheel assembly consists of motors, motor drivers and flywheels. However, due to the shortage of pins on the STM32F2 and shortage in code memory on the MSP430 the reaction wheel assembly could not function. This led to the addition of the third processor (STM32L4). The STM32L4 was preferred due to its low power requirement (around 0.028 watts). The system architecture is shown by Figure 1.

Tasks Performed Onboard by Processors

STM32F2—The STM32F207IG[1] microcontroller, running at 120MHz, resides on the primary PCB with other components including the sensors required by the Attitude Determination and Control Systems (ADCS), payload transmitter, Flash memory, and SRAM, etc. It runs a real-time operating system (RTOS), Micrium OS-III[2], in which each function performed by the microcontroller has been implemented as a real-time task. The STM32F2 carries most of the load of the satellite since it runs the algorithms required for controlling the attitude and also interfaces with multiple devices onboard using different communication protocols. It is connected directly to the STM32L4 MCU (located on the tertiary PCB) via an I2C bus for which it acts as the master. Other peripherals connected to the MCU include the thermal imaging camera[3] (using UART and DCMI), the GPS (using UART), the payload transmitter (using SPI), the different ADCS sensors (using I2C), the Flash memory (using SPI) and the SRAM (using FSMC).

The primary task of STM32F2 is to execute the algorithms required for ADCS. The ADCS cycle starts with determining the current orbital parameters to locate the satellite in the assumed euclidean space and also to calculate the ideal orientation of the satellite.The calculation of the degree of deviation from the desired attitude of the satellite requires collecting data from the onboard sensors. This acquired data is then fed into the attitude determination algorithm to calculate the current orientation of the satellite with respect to the ideal orientation. The result is converted into actuation data, which holds the information about actual PWM values to be generated on the actuators which in turn can produce the required amount of torque to ultimately bring the system within the accepted margin of stabilization error.

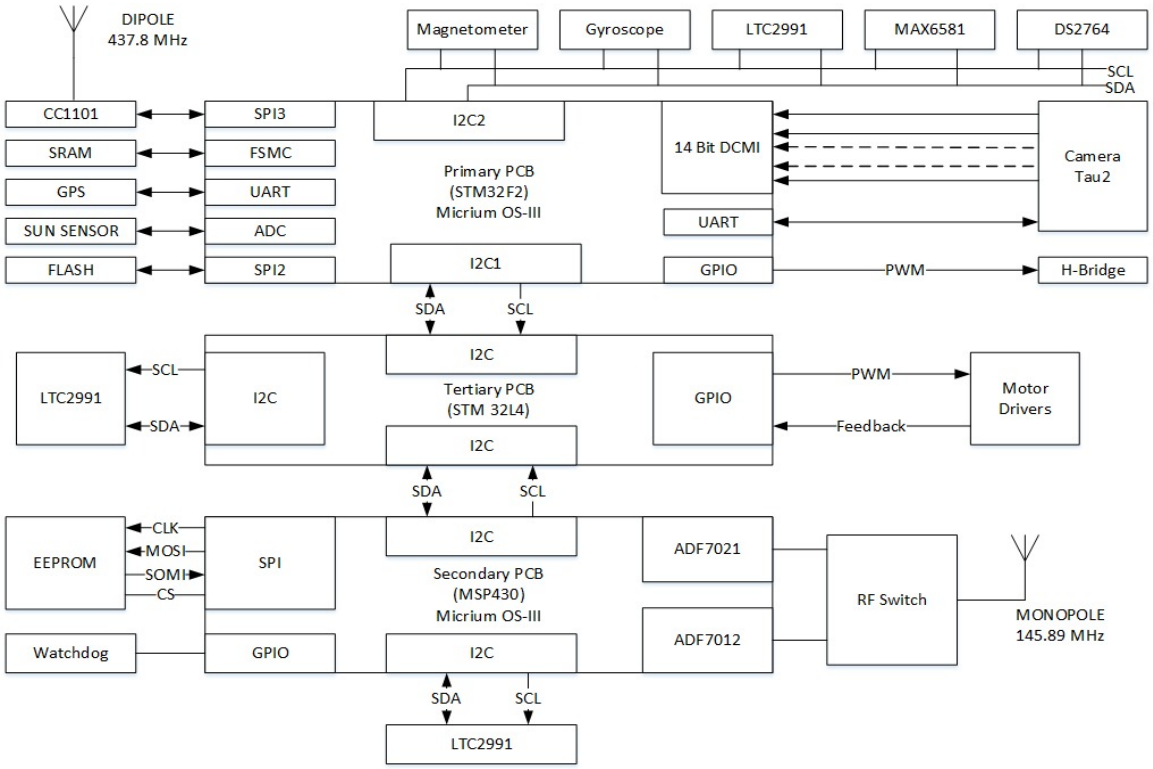

Figure 1. System Architecture

Among the two sets of actuators present on board, the magnetorquers and reaction wheels, the choice of actuator depends on various physical factors such as power budget, error margin, and region of application. The magnetorquers are directly controlled by the STM32F2 using dedicated timers for generating PWM signals on the GPIO pins whereas the reaction wheels are controlled by the STM32L4 to which the actuation data is sent over the I2C bus for application.

Apart from the line of tasks including actuation and stabilization, other lines of tasks on STM32F2 include capturing, processing and transmitting payload data, calibrating the orbit propagation, monitoring and optimizing battery use and generating housekeeping data. STM32F2 interfaces to the thermal camera using the UART protocol to issue click commands and configure data rate but uses the Digital Camera Memory Interface (DCMI) to extract the raw data directly in the form of 14-bit wide pixels. The images are clicked and stored in the SRAM using the FSMC interface. The SRAM can store a maximum of three images at a time, which are read for processing and transmission. The images are 2-D arrays of 512 pixels height and 640 pixels width, which are read from the SRAM as tiles of 16 x 16 pixels and fed into a lossless compression algorithm. The output of the compression algorithm is a set of compressed packets obtained from one image which are further processed to handle fragmentation loss and transmission loss by adding prefixes containing index and length.The packets are finally stored into the flash in contiguous memory locations with the timestamp specific to every image.

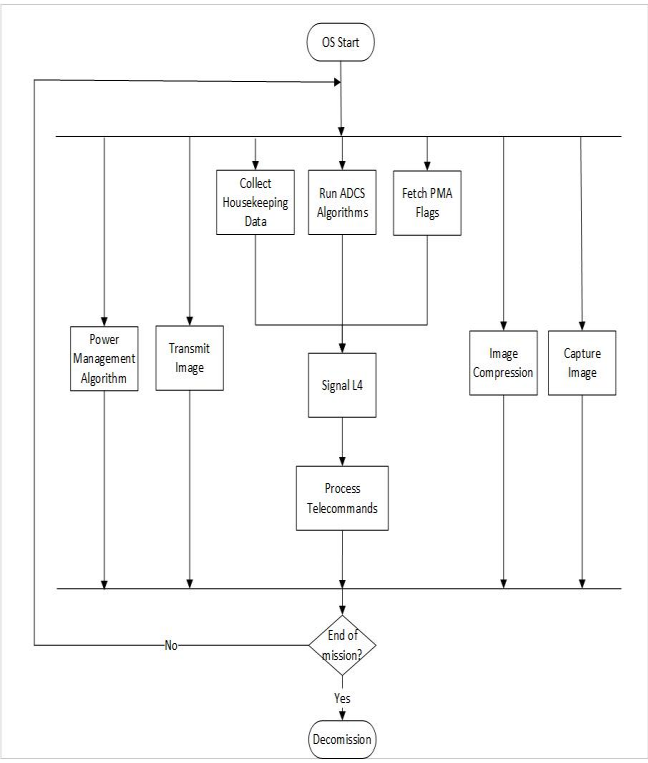

Figure 2. STM32F2 Task Flow

When the satellite is over the ground station range, and the power management algorithm (PMA) allows transmission of data, the compressed data is fetched in chunks of 223 bytes and fed into the RS encoding algorithm which produces a stream of 255 bytes ready for transmission. These byte streams are then handed over to the CC1101 transmitter which implements packetization as a part of its transmission protocol and ultimately transmits those packets to be intercepted by the ground station. A reverse engineered algorithm is used to retrieve the original images, and further image processing is used to enhance and extract as much information as possible out of them.

Orbit propagator is a tool which uses orbital dynamics to predict the location of the satellite in the assumed euclidean space at time t based on the orbital parameters at time $(t-1)$. Like any dynamic system, the orbit propagator accumulates error due to unquantifiable variables and needs calibration, which is provided by interfacing with the onboard GPS using the UART communication protocol and hence the values of orbit propagator are updated.

The satellite's orbit is geocentric which results in periodic eclipses on the satellite making it unable to generate power using its solar panels. Hence, the satellite needs to store the extra power generated and ration its power consumption. This requires strict monitoring and optimization of power use, and the battery is under constant inspection by the microcontroller. Various power modes have been designed to cater to the satellite's power needs which enable or disable some hardware modules and save power for the essential modules of the system to ensure proper functioning. The flow of control on the primary processor is shown in Figure 2.

MSP430—The MSP430F5438A[4] microcontroller, running at 1MHz, resides on the secondary PCB with other components including EEPROM, transceivers, watchdog, etc. Similar to the STM32F207IG ,it also runs a real-time operating system (RTOS), Micrium OS-III where every function has been implemented as a real-time task. The MSP430 interfaces with several peripherals, handles the deployable mechanisms, beacon transactions with the ground station and the telecommand reception, making it the entry point for any ground to satellite communication. It is connected directly to the STM32L4 on the tertiary PCB via an I2C bus for which it acts as the master. Other connections include the connection to the EEPROM (using SPI) and the transceivers.

The primary goal of MSP430 is to constantly determine and transmit the health of the satellite and act as a stand-alone communications hub for the mission even in the worst case scenario where the other two processors fail to deliver. The housekeeping data from all the three PCBs, collected from all devices arrive at MSP430 for the final stage of processing followed by transmission. MSP430 feeds the housekeeping data into a morse code generator to be transmitted once the satellite is within the ground station's acceptance cone. The time in every orbit which overlaps with this acceptance cone is multiplexed to handle transmission and reception in a pseudo-parallel manner by dividing the morse data into four segments that are transmitted with an interval of 60 seconds between the transmission of two consecutive packets, during which the satellite accepts opcodes from the ground station.

The mechanism tasks which include antenna deployment, door deployment, and tether deployment are carried out by MSP430 as a series of predefined events in time and are error handled by various software and hardware means. Each one-time executable task has a flag associated with it which is saved in the EEPROM, to prevent the repetition of tasks in case of a system reset. The MSP430 houses an independent watchdog that initiates a full system reset in case the system hangs. The EEPROM makes sure that the system resumes

from the last dynamic cycle and does not perform any one-time executable task again after resetting by loading the last state of the system in the EEPROM in every cycle.

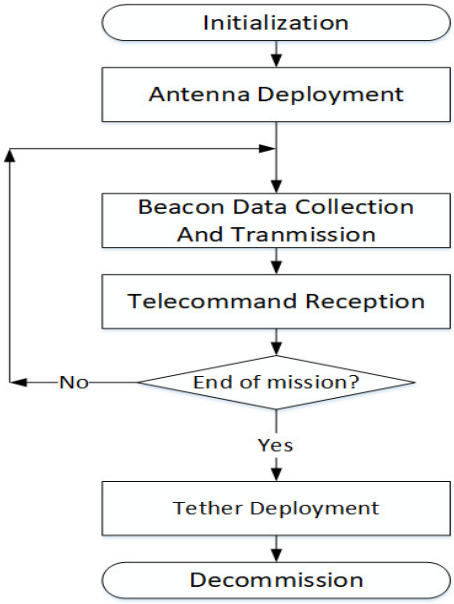

Figure 3. **MSP Task Flow**

The satellite decommissions upon receiving the opcode for it or completing the mission life (whichever is earlier) and stops all communication marking the end of the operational part of the mission. The flow of control on the secondary processor is shown in Figure 3.

STM32L4—The STM32L431CC[5] microcontroller, running at 80MHz, resides on the tertiary PCB with other components including motor drivers and other monitoring devices. It directly runs using a Hardware Abstraction Layer without any operating system layered between them.

It is connected to the other two microcontrollers via different I2C buses hence completing the virtual linear connection between the three microcontrollers. STM32L4 acts as the slave for both its I2C buses and forms a delayed virtual connection by simulating a multi-master environment.

The incoming data from STM32F2 on the I2C bus contains actuation information and housekeeping data to be acted upon and passed to the MSP430 for transmission respectively. The incoming data from MSP430 contains opcodes from the ground station to be passed on to the STM32F2. STM32L4 processes the data coming from STM32L4 before copying it into the outgoing I2C buffers (to the MSP430) by truncating the actuation information and concatenating its housekeeping data which includes rotor lock flags, OCPC status flags and feedback value of motor drivers.

The actuation information contains required RPM values to be applied on the motor drivers for spatial stabilization of the system which are then converted into PWM ratios to be supplied to the motor drivers.

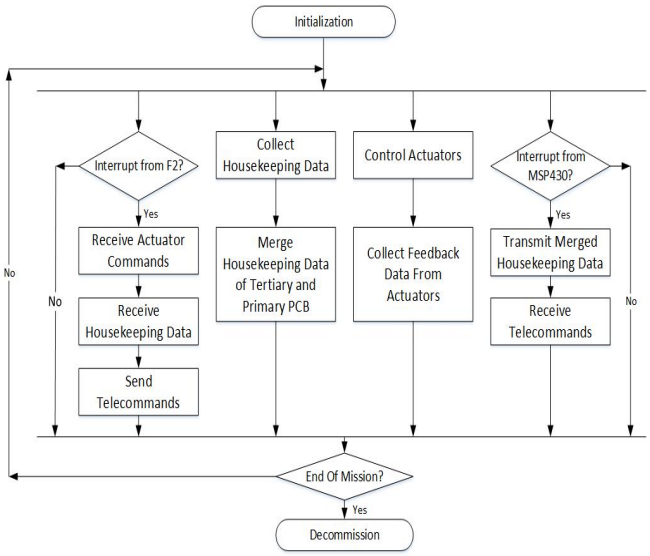

Figure 4. **STM32L4 Task Flow**

The priority of the I2C bus connecting STM32L4 and STM32F2 is greater than the one connecting STM32L4 and MSP430 which ensures uninterrupted transactions on the former at the cost of starvation of the later. However, the solution to the starvation problem is intrinsic to the systems since the frequency of ADCS cycle is much less than the frequency of I2C (100kHz) which brings the probability of starvation to occur, low enough to be excluded from the error handling algorithm. However, a simple re-transmission strategy has been implemented on MSP430 to go about the starvation problem in the worst case scenario.

Apart from its primary task of establishing the virtual linear connection as a mediator, the STM32L4 also records feedback values from the motor drivers. In case of an anomaly in the feedback values, the rotors are checked for rotor lock condition, and the motor driver Over Current Protection Circuits (OCPCs) are checked for a fault condition which if detected, is informed to the STM32F4 for invoking the required failure mode. The flow of control on the tertiary processor is shown in Figure 4.

Process Scheduling

An independent task handles each process that needs to be performed onboard the satellite. The System makes use of a real-time operating system to schedule the various processes. Each task is allotted a dedicated stack space and the information related to each task is stored in a Task Control Block (TCB).

The Scheduling Algorithm used by the Micrium Kernel makes use of a modification of the pre-emptive priority scheduling algorithm. Micrium maintains a Ready List (a double linked list) which maintains links to the TCB's of the processes that are ready to run, arranged in the order of decreasing priorities. Each time the scheduling algorithm executes it selects the process with the highest priority that is ready to run and executes it. After the task has completed its execution, it suspends itself (either for a fixed period or till an external event occurs) and is added to a waiting list and the scheduling algorithm executes again. When the event for which the process was waiting occurs (or the time for which it was waiting expires), the process is removed

Contexts

Name	Type	Stack Information	Run Count	Frequency	Last Run Time	Min Run Time	Max Run Time	Total Run Time	Run Time/s
Scheduler			6402	12 Hz	0.0117 ms	0.0044 ms (#9186)	495.8186 ms (#9417)	731953.5096 ms	1329.2606 ms
App Task Start	@0	500 @ 0x20012628	1	0 Hz	0.0 ms			0.0 ms	0.0 ms
ADCS	@0	7000 @ 0x20012938	1600	3 Hz	19.0612 ms	18.9333 ms (#737)	19.1217 ms (#38493)	30457.1951 ms	57.1747 ms
Fetch_GPS	@0	500 @ 0x200129fc	2	0 Hz	0.0093 ms	0.0093 ms (#76)	0.0093 ms (#76)	0.0093 ms	0.0 ms
CAPTURE 1	@0	500 @ 0x200127b0	2	0 Hz	0.0162 ms	0.0162 ms (#72)	0.0162 ms (#72)	0.0162 ms	0.0 ms
TRANSMISSION 2	@0	500 @ 0x200126ec	1286	3 Hz	0.0299 ms	0.0098 ms (#87)	4470.5979 ms (#428...	8966.2486 ms	0.0601 ms
COMPRESS 3	@0	500 @ 0x20012874	322	0 Hz	290.4781 ms	0.0075 ms (#905)	495.7452 ms (#126)	988.9658 ms	0.0 ms
check_Decommission	@0	500 @ 0x20012c48	2	0 Hz	0.0 ms			0.0 ms	0.0 ms
EPS_Housekeeping	@0	500 @ 0x20012ac0	1601	3 Hz	4.0777 ms	4.0526 ms (#31078)	4.1306 ms (#40359)	6534.3484 ms	12.2610 ms
Beacon_Formation	@0	500 @ 0x20012b84	1601	3 Hz	0.0734 ms	0.0544 ms (#255)	0.0759 ms (#33592)	107.0607 ms	0.2204 ms

Figure 5. **Context View of Tasks**

from the waiting list and added back to the Ready List. The Scheduling Algorithm executes each time an executing process is suspended, an external event occurs or the waiting time of a process expires.

To quantify the scheduling on STM32F2, we use segger system view[6]. Segger SytemView is a toolkit for the visual analysis of an embedded system. The SystemView host application provides real-time analysis and profiling through the use of Real-Time Transfer (RTT) technology. SystemView makes it possible to analyze which interrupts, tasks and software timers have executed, how often, and how much time they have used. SystemView is used to verify that the embedded system behaves as expected and can be used to find problems and inefficiencies, such as excessive and spurious interrupts, and unexpected task changes. It can be used with any RTOS which is instrumented to call SystemView event functions, but also in systems without an instrumented RTOS or without any RTOS at all, to analyze interrupt execution.

The setup consists of a target system, the host system, and a J Link debugger module. A software module, containing SystemView and RTT is included on the target system.The target system calls SystemView functions to monitor events. The SystemView module collects and formats the monitor data and passes it to RTT. The RTT module stores the data in the target buffer, which allows continuous recording with J-Link on supported systems.

The results obtained from segger allows us to verify and quantify the functioning of the RTOS. From the context view (Figure 5) given by segger, we get the complete runtime statistics of the system. It can be seen that ADCS task runs for the longest amount of time whereas the task responsible for Transmission has the most substantial CPU time for one run.

From the Timeline view window (Figure 6), we can see that the ADCS task has the highest priority whereas the task responsible for beacon data formation task has the lowest priority. We also see the priority based preemption of the task responsible for compression by the task responsible for fetching GPS values. From the CPU load window (Figure 7), we see that each of the tasks have their own needs and resources and the load they put on the CPU is varied.

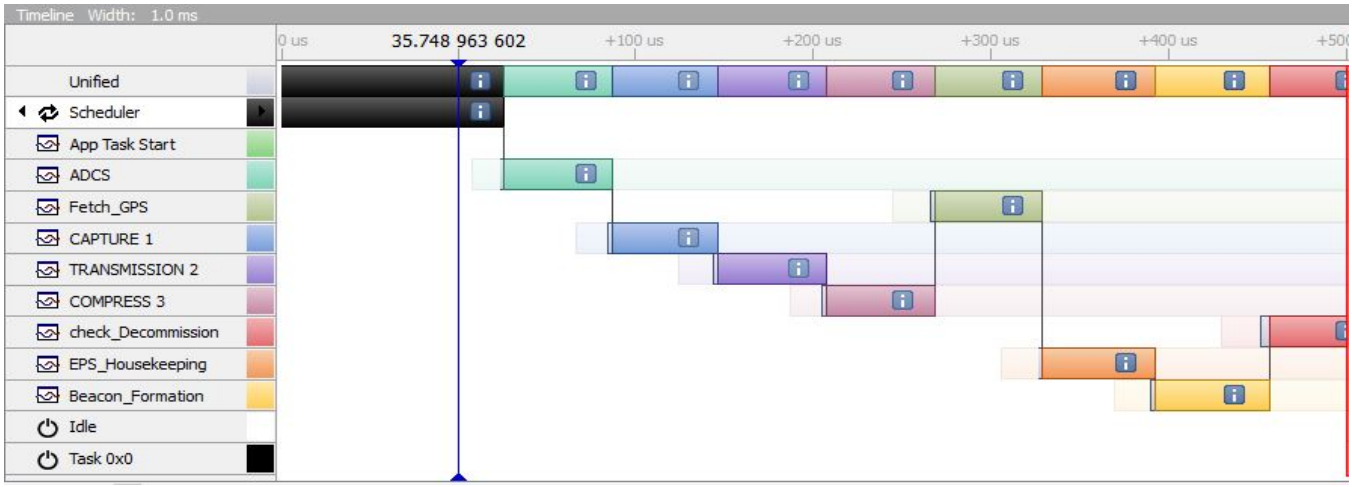

Figure 6. **Timeline View of Tasks**

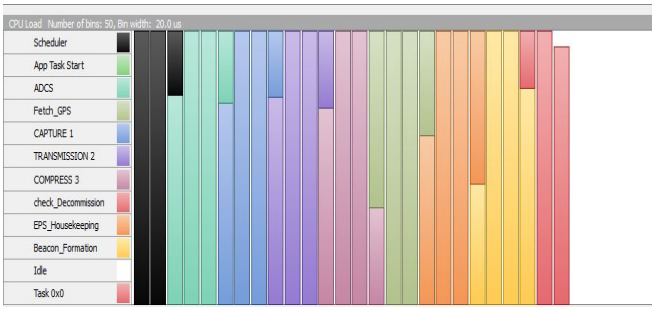

Figure 7. Load on CPU

The task scheduling on MSP430 was modeled into a Gantt chart. As can be seen in Figure 8, the MSP switches between two major tasks after the antenna deployment is complete. The telecommand reception task runs for sixty seconds after the transmission of every beacon packet. While power constraints might force the system into switching off beacon transmission, telecommand reception is never disabled throughout the mission life.

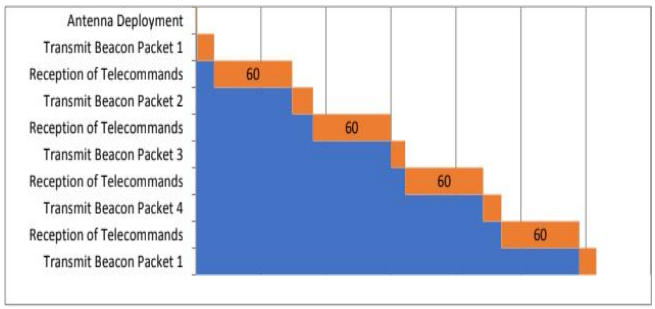

Figure 8. MSP Task Flow

Inter Processor Communication

Flow of Control— I2C links for data transfer connect the microcontrollers on board. In this environment the STM32L4 acts as a slave for both STM32F2 and MSP430 which are linked to different I2C peripherals of the STM32L4 to avoid a direct multi-master environment. The data flow between the processors is show in Figure 9.

The flow of control between STM32F2 and STM32L4 is given below:
1) The STM32F2 being the I2C master initiates the communication. It sends commands for actuator control. These commands are generated by the ADCS algorithms running on the Primary PCB.
2) STM32F2 follows this by sending the PMA (Power Management Algorithm) mode flags to the Tertiary PCB. The Tertiary PCB later relays these flags to the Secondary PCB.
3) The Primary PCB then sends the Housekeeping data. Housekeeping data includes health data about each peripheral on the PCB.
4) The STM32F2 then queries the Tertiary PCB for any telecommands that it might have received from MSP430.

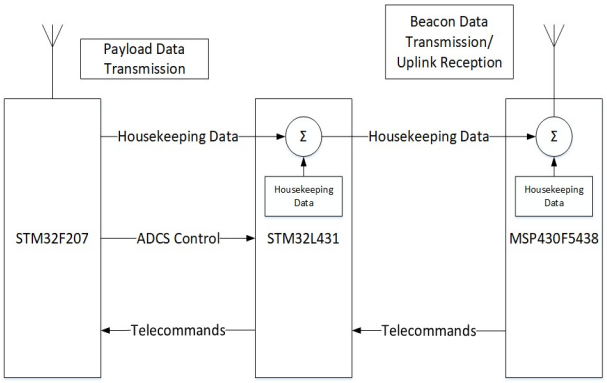

Figure 9. Data Flow

The flow of control between MSP430and STM32L4 is given below:
A communication block between these two microcontrollers can be described as follows:
1) The MSP430 being the I2C master initiates the communication by requesting for data.
2) The STM32L4 complies by sending the PMA mode flags that it had received from the Primary PCB.
3) The Tertiary PCB then sends the Housekeeping information for both the Primary and Tertiary PCB.
4) On receipt of the Housekeeping information, MSP430 sends any telecommands that it might have received that requires processing in the Primary or Tertiary PCB.

This sequence of data transfer is repeated every time the MSP430 transmits one complete block of beacon data and needs to form a new packet. To handle conflicts arising due to simultaneous communication requests by both the I2C masters the interrupts on the I2C slave (STM32L4) have been prioritized. A higher priority has been given to the Primary PCB to ensure that the Attitude Control of the satellite is not disturbed.

Justification Of A No-OS Model On STM32L4

The tasks on the STM32L4 have been scheduled using interrupts from the other two processors. This is in contrast to the other two processors which use a real-time operating system (RTOS) to schedule their tasks. This choice has been made based on the available processing power, the availability of a code library, and the nature of the tasks to be executed. The tasks on STM32L4 are short-lived, time-dependent and cannot be interrupted by other tasks in order for them to produce the expected results. Tasks on STM32L4 include I2C communication, buffer processing, input frequency compare and GPIO probing, all of which contain sub-tasks to be executed in a strict order and at specific intervals of time. Interleaving of these sub-tasks among different tasks will result in catastrophic failure.

However, if an OS based model is forced upon this pool of tasks, the code will ultimately end up using scheduler lock statements before every function, hence defeating the purpose of using the OS in the first place. The second functionality provided by an OS is a precise channel of timers that can run independently from the scheduler and can be used for error handling parallel to the task switching. Since the STM32L4 has multiple independent clock sources with available slots, specific timers can be dedicated to run error handling algorithms and pseudo-watchdogs without interrupting the mainline tasks, hence fulfilling the need of OS

timers. Operating systems require huge stacks of memory and a fair share of CPU cycles to execute system calls and scheduling algorithms which consumes memory and power. This can be conserved by using an interrupt based model. The HAL kernel is sufficient enough in providing the required functionalities to cater to the needs of this processor, especially when because STM32L4 is a slave to other processors and does not generate any data but only uses what it is provided with.

The whole point of a modularized multi-processor environment is to decrease the system complexity, therefore further decreasing the possibilities of an error which is simply achieved by implementing an interrupt based model on the STM32L4 and reducing one redundant layer between the kernel and the code.

3. CHALLENGES FACED AND SOLUTIONS

Working with Distributed Architectures

One of the biggest concerns on using a Real-Time system across multiple processors is the designing of message passing interfaces. Designing the interfaces is complicated due to different byte ordering and padding rules in different processors. When multiple processors and links are involved in message interactions distribution of load among the processors is of critical importance. If the system has an evenly balanced load, the capacity and efficiency of the system can be increased. However, in most cases a single processor or link becomes a bottleneck, leading to the costly redesign of the features to improve system efficiency. Another issue faced is the maintenance of data structure integrity across multiple processors.

One more critical issue is the recovery from processor failure. When the failed processor comes back up, it will have to recover all its lost context from other processors in the system. However, there is a chance of inconsistencies existing between different processors in the system thus putting the system in an inconsistent state when the failed processor is running again. This leads to the need for an efficient and complex error management scheme being implemented on the system which further increases the complexity of the system.

Race Conditions and Timing

A real-time system is described with the exact timing specification for each stage. Real-time systems deal with timing issues by using timers. Timers are started to monitor the progress of events. If the expected event takes place, the timer is stopped. If the expected event does not take place, the timer will timeout, and recovery action will be triggered. A race condition occurs when the state of a resource depends on timing factors that are not predictable. The major issue here lies in identifying the reason for the occurrence of the race condition. Most race conditions are subtle and can only be identified by careful examination of the design.

Error Handling

All real-time distributed systems are prone to hardware and software errors, requiring a robust and dynamic error handling algorithm which should itself be recursively safe from the errors it is trying to solve while still being a part of the same system. Although the problem sounds paradoxical, some hardware and software changes can solve the problem to a great extent and bring down the probability of an unsolvable error occurring to a minimum.

Independent watchdogs are great examples of such hardware changes which solve the problem of error detection and to some extent, error correction. In the event of the system hanging due to bad scheduling, asynchronization between hardware and software, hardware lock or power failure, the watchdog serves as a great tool to detect the error in its preliminary stage and make a first naive attempt to correct it. Complementing the watchdog with a permanent storage device such as an EEPROM helps in transforming this naive attempt into a dynamic algorithm which allows the system to not only reset by hit and trial but to record and analyze the conditions which triggered the reset in the first place. The system can then disable or re-initialize individual modules that caused the error and update all processors with this information to re-establish synchronization by invoking a predefined algorithm for a modeled failure in all the processors. A multi-processor environment reduces the chances of a complete system breakdown and provides a window for redundancies which different parts of the satellite to act as stand-alone modules. A dominant I2C architecture where two processors are the master of the same processor further reduces the chances of error as compared to a transitive I2C architecture where a processor is a master for one but the slave for another processor by reducing the number of possible link failure points.

Apart from the hardware implementations, some algorithmic changes can further enhance the system's error detection and correction capability. Backtracking is one such method, to start detecting the error from the layers farthest from the hardware layer and move down in the hierarchy by one step at a time. This is implemented by reading predefined bytes from the peripherals, called the peripheral ID, to ensure the absence of an error at that layer. The next layer involves re-initializing the underlying structures used for communication and configuration of the peripheral causing the error and invoking a different configuration if necessary. The next layer involves the communication protocol being used by the peripheral and to re-configure the registers used by the protocol in order to re-establish connection followed by the last layer, which involves the hardware and requires a hard reset of the peripheral.

Other more naive algorithms include hit and trial strategies used for those modules only which are less prone to errors and are guaranteed to revive at some point in time. An OS based error handling is intrinsic to the scheduling algorithm implemented by the OS itself which prevents the system from freezing in a particular state and continues system operation until that error is detected and corrected.

4. LESSONS LEARNT AND FUTURE PROSPECTS

Performance Comparison Of OS-based and Interrupt-based Systems

The system organization revolves around OS based and interrupt based task scheduling. The STM32F2 and MSP430 implement task scheduling with the help of a real-time operating system (RTOS), and the STM32L4 uses interrupt based task management. The choice of architecture is based on the nature of tasks, the workload, and the available processing power. The tasks on the STM32F2 and MSP430 are interruptable and secure from the hazards of parallel processing. Most of these tasks are executed due to triggers caused by

specific flags which implies they can be absent from the active set of tasks for a long time. The sudden introduction of these tasks into the ready list, for example, transmission of an image when over the region of interest, can cause errors in execution of other housekeeping tasks and requires robust task switching to continue normal operation by providing the illusion of the availability of CPU time to all the tasks equally. Since this is a dynamic system, the data is always actively changing, and the results of various functions are only true for small periods of time thus requiring frequent updates to maintain the integrity of data. Such functions are called real-time tasks, with strict deadlines. The function needs to move on, in order to make a new attempt to generate reliable data instead of waiting for some data that was to be generated by a previously stuck state of the function, for example, a failed I2C communication. This is exactly what makes a real-time operating system (RTOS) a necessity in real-time distributed systems.

On the other hand, the tasks on STM32L4 are non-interruptable and prone to the hazards of parallel processing. STM32L4 receives preprocessed data ready for application which usually means initiating an I2C communication on the virtual linear bus or probing the incoming frequency from the motor drivers for a few consecutive pulses. These tasks are composed of less number of subtasks which cannot be interleaved in space and time, making the whole task intrinsically atomic, for example, capturing and comparing input pulses, which involves consecutive pulses to be captured with no time delay. Moreover, there are a less number of tasks on the STM32L4 as compared to the other two microcontrollers and the memory overhead generated by using an operating system model is huge. Since the STM32L4 processor is a mediator between its two masters and uses data generated by them, it works just fine on the HAL kernel without implementing any layer of operating system above it.

In conclusion, the use of an operating system seems unavoidable in cases where a large number of multiplexable real-time tasks are run on a single master processor. Still, it can be avoided in cases where a small number of short-lived tasks are running on a single slave processor, taking into consideration the discussed architecture.

Lessons Learnt

Throughout the development of the satellite, several problems were faced and resolved. While problem faced in the past and the subsequent decisions are mentioned in system evolution, the problems faced in the current model are resolved as explained in the paper. One of the significant challenges of our architecture was making sure that tasks on the system were executed in real time and no deadlocks were arising in the system. This was solved by the development of the packet protocol and assigning fixed priorities for the STM32F2 and the MSP43O I2C links in the scope of STM32L4. The decision of having a while-loop kernel on the L4 allowed for the faster development of the system software. Since each MCU has a dedicated task, our system will still be functional in the case of partial failure which is one of the most considerable advantages of having a distributed system. Even if the interboard links fail, the MSP430 will still transmit the beacon data collected from the secondary PCB; the F2 will continue operation of the payload and transmit the thermal images. The error handling present on the system is also designed to ensure that the satellite deploys the tether and decommissions itself even if the telecommand reception link fails.

Future Prospects

The real-time-interrupted distributed data handling system discussed throughout the paper is a scalable model that can be integrated with other software systems to produce software designs for handling various missions not just in the space industry but any dynamic institution. Autonomous systems can use the following model coupled with machine learning to transfer the control of task switching from space and time-dependent flags to the historical data generated by the software itself, allowing it to learn from previously made mistakes. For example, an autonomous drone can use the actuation information delivered by the ADCS algorithm to stabilize itself and record the degree of stabilization achieved and then tweak the input parameters of the ADCS algorithm based on this history of data in the upcoming ADCS cycle. Small to medium scale satellites and rovers can implement this distributed system on their sub-modules as independent entities such as communication hub or gyration monitoring.

The protocols and compression algorithms will be re-visited to explore possibilities of improvement, to be later used for point to point encrypted communication and broadcasted communication in a constellation of nanosatellites. In the near future, the feasibility of implementing such a distributed system using a single master multi-core processor will also be a topic of research. Based on the performance of this nanosatellite system, the variability of use of this design on other real-time systems looks very much efficient and builds an area of interest.

5. SUMMARY

The proposed system design for nanosatellites was thoroughly tested by exposing its hardware and software modules to different use cases by using both dummy data and real-time data. The inter-processor communication was put through multiple erroneous instances to test the reliability of the underlying error handling algorithms, which successfully passed those tests. Dummy images were fed into the compression algorithm, and a complete payload-data processing and transmission cycle was simulated under varying temperature and pressure environments. The ADCS cycle was put in a software loop to simulate six months of sun-synchronous orbiting and was tested for various boundary conditions that might get realized in an actual orbit. The test for decommissioning and deorbiting of the satellite was conducted with other mechanisms in a zero-G environment, putting the system through the closest possible load to an actual space mission. In conclusion, the system went under rigorous testing and is ready for deployment in space missions for nanosatellites with the architecture discussed above.

APPENDIX: ABBREVIATIONS

ADCS: Attitude Determination and Control Subsystem
DCMI: Digital Camera Memory Interface
DMA: Direct Memory Access
EBI: External Bus Interface
EEPROM: Electrically Erasable Programmable Read Only Memory
EPS: Electrical Power Subsystem
FET: Flash Emulation Tool
FSMC: Flexible Static Memory Controller
GPS: Global Positioning System
I2C: Inter-Integrated Circuit
IGRF: International Geomagnetic Reference Field

IMU: Inertial Measurement Unit
JTAG: Joint Test Action Group
OCPC: Over Current Protection Circuit
ODHS: On-Board Data Handling Subsystem
OTP: One Time Programmable
PCB: Printed Circuit Board
PD: Proportional Derivative
PID: Proportional Integral Derivative
PMA: Power Management Algorithm
PWM: Pulse Width Modulation
ROI: Region Of Interest
RTC: Real Time Clock
RTOS: Real-Time Operating System
SDR: Software Defined Ratio
SOC: Status Of Charge
SPI: Serial Peripheral Interface
SRAM: Static Random Access Memory
TLE: Two Line Elements
UART: Universal Asynchronous Receiver Transmitter
USART: Universal Synchronous Asynchronous Receiver Transmitter
WDT: WatchDog Timer

REFERENCES

[1] www.st.com/resource/en/datasheet/cd00237391.pdf

[2] doc.Micrim.com/display/osiiidoc/uC-OS-III+Documentation+Home

[3] www.flir.com/globalassets/imported-assets/document/16-0423-oem-datasheet-update-tau-2.pdf

[4] http://www.ti.com/lit/ds/symlink/msp430f5438a-ep.pdf

[5] www.st.com/resource/en/datasheet/stm32l431cc.pdf

[6] https://www.segger.com/downloads/free-utilities/UM08027

BIOGRAPHY

Akshit Akhoury *is a third-year student in Manipal Institute of Technology pursuing a B.Tech in Computer Science. His areas of interests include embedded systems, space missions, and advancements in technology.*

Arun Ravi *is a third-year student at Manipal Institute of Technology pursuing a B.Tech degree in Computer Science and Engineering. His interests include Internet of Things, Sensor Networks, Embedded Computing, and Real Time Systems.*

Krishna Birla *is a third-year student at Manipal Institute of Technology pursuing a B.Tech degree in Computer Science and Engineering. His interests include physics, algorithms, and writing.*

Rohit Sarkar *is a third-year student at Manipal Institute of Technology pursuing a B.Tech degree in Computer Science and Engineering. His interests include embedded systems and their applications to a variety of problem domains.*

Shaleen Kalsi *is a fourth-year student at Manipal Institute of Technology, pursuing a B. Tech degree in Computer Science and Engineering. Her interests span the fields of Data analysis and algorithms, embedded systems, and Artificial Intelligence*

Subhojit Ghorai *is a second-year student at the Manipal Institute of Technology, pursuing a B. Tech degree in Computer and Communication Engineering. His areas of interest include embedded systems, application development, and algorithmic coding.*