Date: 7 Oct 2023
Authors: Chris Thompson, G0KLA / AC2CZ / VE2TCP, Burns Fisher WB1FJ
Version: 1.2

PACSAT Software Architecture

1 Introduction

This gives a high level overview of software architecture for PACSAT. The architecture is derived from the software used for the FOX and GOLF AMSAT satellites.

This document is organized into the following sections:

- **Hardware** – High level description of the processor and memory describing the background needed for software development.
- **Operating System** – High level description of the Real Time Operating System (RTOS).
- **Software components** – Overview of the key software tasks, queues, timers and data stores.

References are given at the end.

1.1 Key Decisions

This is a list of key decisions that need to be made:

1. Will commanding re-use the GOLF format, which would require a dedicated AX5043, or will it use AX25 framing and code to separate commands from the PACSAT protocol, so that any of the receivers can be used? Current assumption is that commands will be wrapped in AX25 so that we don't need a dedicated AX5043 receiver. The content of the command will be the same as GOLF commands.
2. Will it be possible to update any of the code post launch? The current assumption is no.
3. How much MRAM memory is needed for the file store? See reference [1] for an initial memory utilization model.
4. Should key configuration values, such as the frequencies, be stored twice in MRAM? Currently they are, but we have not seen an error between the two in orbit and it may

be a waste of space. We could instead write parity bits or a Forward Error Correction code.

5. Min/Max may not be worthwhile unless it clears automatically.

2 Hardware

Pin connections are not discussed in this document because they are subject to change. Currently connections are defined in terms of the TMS570 Launchpad board and are listed in the LaunchPad Headers spreadsheet[5].

2.1 Processor

The processor is a TMS570 at 160MHz or slower. There are two types of memory in the TMS570, Flash and RAM, with a third type of memory available over SPI. These are described below.

2.2 Memory

There is 1280K of Flash memory, which is non-volatile memory that holds the program and some constants. All code lives in the flash memory and is executed there. The FOX and GOLF satellites do not write to the flash memory in orbit since it uses a charge pump which (we are told) is often the first thing to fail in a high radiation environment. It also has a limited number of write cycles (a few hundred or thousand?). Flash memory has an address and can be operated directly by the processor and code is generally executed from it.

There is 192k of RAM which is where volatile data is stored. About 80k of RAM is currently allocated in the initial build, with 64k allocated as heap storage and 32k of the heap is used by the OS.

Separately there is a bank of Magneto-resistive RAM (MRAM). MRAM has no write cycle limit and seems to have a very good record in radiation. However MRAM is only accessible as a peripheral via a Serial Peripheral Interface (SPI) bus --it has no address and code can't be executed directly out of it.

MRAM will be used to store the following:

- PACSAT BBS messages
- Whole orbit telemetry data
- Min/Max telemetry data
- Writable non-volatile storage for such things as "what was the last mode we were in", reset count, etc.

The MRAM memory map is in Interfaces/MRAMmap.h

MRAM is composed of 2 sections. One section stores the File system and another stores paramaters.

When you init the MRAM, the version gets written at the start and end. Each time you boot, the written values get checked against MRAM_VERSION. If either is wrong it warns you to re-init the MRAM. Of course the intent is that if we change any part of the structure between the the two version cells, then you will be reminded that the MRAM needs to be re-initialized.

AuthenticateKey is used for the command key. The only way to write it is with the "load key" command. Otherwise it falls back to the "non-secret" default key used for testing.

A "state" structure has all the various pieces that are saved in MRAM, including the frequencies. They are not all used in the Pacsat code. They each have double the amount of space they need. Read reads them both and compares them. We have never seen them wrong, so this may be a waste of space, or maybe just a single parity bit for each byte or word would be sufficient.

2.3 Power on Reset

A reset or a power cycle does the same thing. There is an address that the processor goes to in flash that is hardwired (at least it was in the STM). We can place a jump there if we don't want the code to start there.

2.4 Code Updates in Orbit

We have never had the capability to change code on orbit. For all these processors the code has to be in flash to be executed. We felt we should not write flash after launch. We froze the code months before delivery of the satellite and started doing testing. For each problem, we carefully considered whether changing code or keeping a bug was more risky.

While writing flash is a risk, perhaps storing some code or scripts in MRAM is possible so it can be loaded after boot. This is something to research.

3 Operating System

FreeRTOS is the “operating system” for PACSAT. It has these main functions:

- Task Switching
- Timers
- Semaphores, Mutexes and queues
- Memory and IO (sort of)

3.1 Task Switching

RTOS is configured to use co-operative scheduling. Preemption is not enabled.

Tasks can be created and the OS will switch between them to give the impression that they are operating simultaneously. This switching always runs the highest priority task and a task is switched away from only when that task blocks (i.e. it makes a call to wait for something).

You do not load tasks, they are all compiled together. You can start and stop tasks, but there is little sense in doing that. It does not save any memory except a bit in the heap.

3.2 Timers

We can create timers which will call some code when it expires. This code can then be used to release a blocked task. In other words one of the things that a task can wait for is for the timer to trigger.

3.3 Semaphores, Mutexes and queues

These are really the same thing---a task can wait on a semaphore, a mutex (which is a binary semaphore) or queue, and a timer or another task or interrupt routine can release it by putting something in the queue.

Note that some semaphores can be arranged so that if they block, the task that currently holds the semaphore can have its priority raised to be the same as the blocked task. This allows the blocked task to run because the lower priority task is able to process and release the semaphore. (This is a Great idea---Mars Pathfinder had this problem and they were able to upgrade the semaphores *ON MARS* to fix it using this algorithm")

3.4 Memory and IO

Notice that we did not say anything yet about address spaces. All tasks share the same address space. If you are used to programming in Linux, think of the tasks as more like threads in a single process.

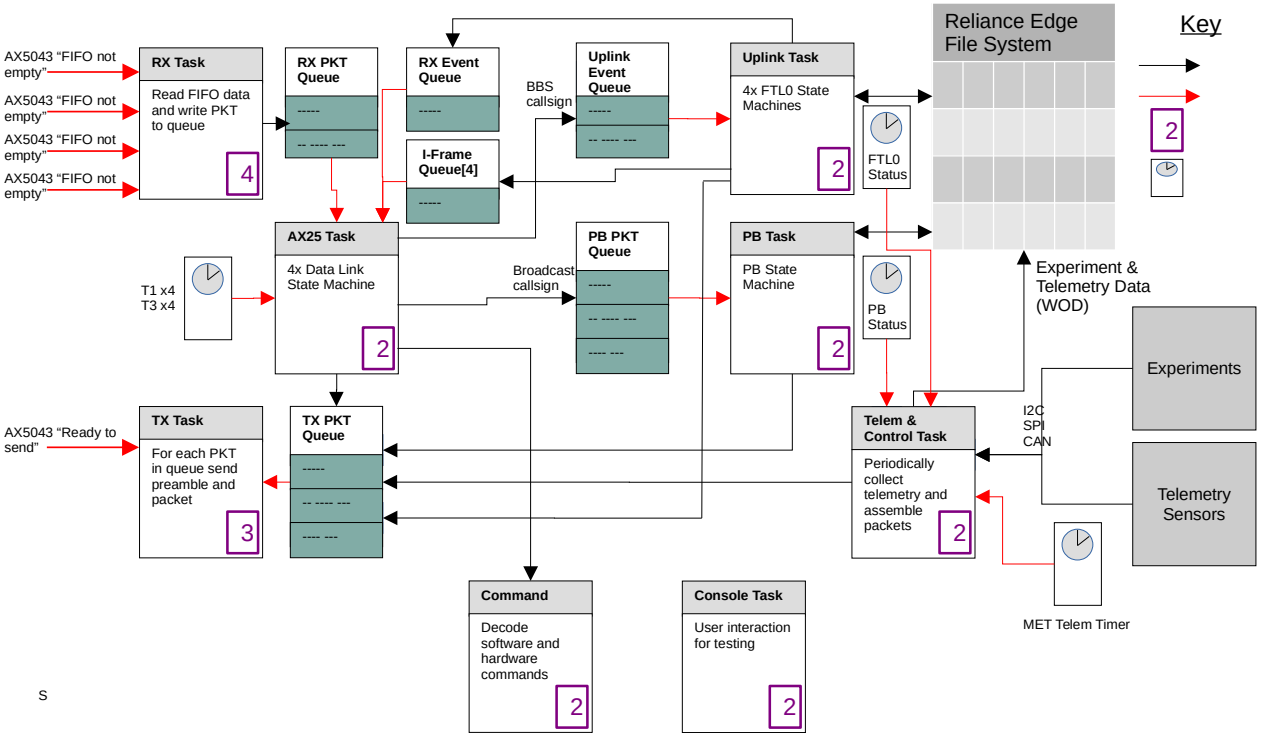
We also did not mention I/O. It is up to us to create all the I/O functions and to ensure that a task waiting for an I/O to complete does not block the rest of the tasks (i.e. it can't just loop). We have previously done this for all the peripherals used by Fox and Golf.

We also did not mention memory management. This is not strictly true. FreeRTOS does have a heap that we can use to alloc and free from. The OS itself uses it to allocate task, queue, and timer data structures. However, we recommend never using this ourselves for two reasons. 1) We generally don't create and then destroy data structures. It is better to allocate them statically. 2) Deallocating data tends to leave holes in the heap so it is easy to get to a point where there is enough memory for an allocation you might want, but you can't allocate it because it is splintered. This is especially true with the relatively small amount of memory in our processor.

Also note that there are drivers for i2c, SPI and CAN, as well as device support code for a number of devices, including MRAM, temp sensors, serial port etc

4 Software components

There will be several RTOS tasks, queues and other items as shown in the diagram and described below:



4.1 RX Task, PKT Queue

This task will wait for the FIFO interrupt from the AX5043 receivers. If an interrupt is received then data is available in the FIFO for one of the receivers. This data is read and if it results in a valid AX25 packet, then it is written to the RX PKT QUEUE.

4.2 AX25 Task, PKT Queues

The AX25 task wakes up whenever there are packets on the RX PKT QUEUE. This task will process the packet according to the Data Link State Machine definition in the AX25 Specification[3]. In summary it will:

1. Handle connection requests from ground stations and establish or disconnect the link.
2. Process connected mode packets and send the required confirmations or retries. These will be written directly to the TX PKT QUEUE

3. Store a queue of the last 7 connected mode packets to handle repeat requests from the other station.
4. Automatically forward or digipeat packets that are addressed with a VIA address, assuming digipeating is enabled. These packets would be written directly to the TX PKT QUEUE
5. Route Type I data packets addressed to the BBS callsign to the UPLINK PKT QUEUE
6. Route Type UI packets addressed to the Broadcast callsign to the PB PKT QUEUE
7. Notify the Command Task of any uplinked commands.
8. Ignore all other packets

4.3 Uplink Task

This task will implement the Uplink State Machine defined in PACSAT Protocol: File Transfer Level 0 [4] and discussed in the “File Uploads” section of the Pacsat BBS Operations[2] design document.

4.4 PB Task

This task implements the Pacsat Broadcast Protocol as discussed in the “PACSAT Broadcast” section of the Pacsat BBS Operations[2] design document.

4.5 TX Task

This task waits until data is available in the TX PKT QUEUE and the AX5043 transmitter is not busy. The data is then added to the AX5043 FIFO and transmitted on the downlink frequency.

Note that this queue should be very short. It should be just long enough to hold items that are waiting while the AX5043 is busy. If the queue is too long then high priority items, such as a connected mode acknowledgment, could be delayed too long. It is also not the same as the packet buffer defined in the AX25 Specification[3], which holds 7 items and handles repeat requests from the other station. That queue will be in the AX25 task and only holds connected mode packets.

If a task such as the PB Task has a large number of packets to send, then it must wait until space is available in the TX PKT QUEUE.

4.6 Telem & Control Task

This task will respond to an RTOS timer. Periodically it will collect health data and calculate max/min values. These will be stored in telemetry records. They will then be either saved into a Pacsat File as Whole Orbit Data or wrapped in an AX25 frame and added to the TX PKT QUEUE for downlink.

This task will also periodically query on board experiments according to their Interface Control Document (ICD). Typically their data will be saved in a Pacsat file for later retreival, but some data could also be wrapped in an AX25 frame and added to the TX PKT QUEUE.

The operation of this task will be dependent on the spacecraft mode. e.g., only critical health information will be collected and downlinked when in safe mode.

4.7 Console Task

This is the task that interacts with a user during debugging and testing phases. It has typed command parsing and can be expanded as needed for more commands. The serial line pins used for testing are defined in the LaunchPad Headers spreadsheet[5]. This is TTL levels at 38,400 baud.

4.8 CAN Task

This is not shown on the architecture diagram because there are currently no plans to use it. Because CAN messages can arrive asynchronously, the easiest way to implement them is with a separate task. It works together with canDriver.c to send and receive CAN messages and to dispatch the received messages. Often these received messages will be telemetry, so they are dispatched to a telemetry collection task. That is currently removed to simplify the PACSAT code. We don't know if we will use CAN at all, but the basic code is there to support potential experiment partners.

4.9 Command Task

Commanding will most likely follow the GOLF/FOX approach to authentication with the data uplinked in an AX25 packet. This task gets a message from the AX25 task when there is command data available. Its job is to decode and validate the command and then to execute it. Note that we often say "encrypt" or "decrypt" for commands. In fact, the commands themselves are not encrypted. However, they are validated by the standard method of transmitting an encrypted digest of the command and then decrypting the digest and comparing it against the digest of the received command. (This is pretty much the same way that downloading Windows upgrades works). The encryption/decryption requires a key.

There is a default key that will be used normally. When we are approaching flight time, we will generate a "real" key to match the "real" command software and load it into the MRAM where the software can read it, but no human can.

5 Guiding Principles

- Don’t use preemption.
- Don’t allocate memory from the heap. Allocate everything statically.

6 References

1. PACSAT Mermory Model, Chris Thompson G0KLA, 26 Nov 2022, https://github.com/AMSAT-NA/PacSatDocs/blob/main/requirements/pacsat-memory-model.ods
2. PACSAT BBS Operations, Chris Thompson G0KLA, 1 Nov 2022, https://github.com/AMSAT-NA/PacSatDocs/blob/main/requirements/Pacsat%20BBS%20Operations.odt
3. AX.25 Link Access Protocol for Amateur Packet Radio, Version 2.2, July 1998, William A. Beech, NJ7P, Douglas E. Nielsen, N7LEM, Jack Taylor, N7OO, https://www.tapr.org/pdf/AX25.2.2.pdf
4. PACSAT Protocol: File Transfer Level 0, Jeff Ward G0/K8KA, Harold E. Price NK6K, https://www.g0kla.com/pacsat/ftl0.txt and its update: https://www.g0kla.com/pacsat/ftl0u1.txt
5. Launch Pad Headers, Burns Fisher WB1FJ, 7 Feb 2023, https://github.com/AMSAT-NA/PacSatDocs/blob/main/requirements/LaunchPad_Headers.ods