18th International Conference on Space Operations, Montreal, Canada, 26 - 30 May 2025.

SpaceOps-2025, ID # 183

Creating a Command-and-Control Software Tailored to Operations for Low-Budget CubeSat Missions

Tom Cundict[a]*, Jake Fernhout[b] , Majd Eissa[c]

[a] *Mechanical Engineering, University of Alberta, Canada, tcundict@ualberta.ca*
[b] *Mechanical Engineering, University of Alberta, Canada, jakeferny@gmail.com*
[c] *Computing Science, University of Alberta, Canada, majd.eissa859@gmail.com*
* Corresponding Author

Abstract

Command-and-control (C2) software is an essential part of all satellite missions, and as the barrier to space continues to lower, more opportunities are emerging for low-cost missions, enabling organizations with smaller budgets—such as university CubeSat teams—to conduct satellite operations. However, licenses for existing C2 software can be expensive for small-scale missions. AlbertaSat Control Software (ABSCS) is a free and open-source C2 software, developed by volunteer students from AlbertaSat at the University of Alberta. Additionally, ABSCS is developed with guidance from Calian Advanced Technologies, leveraging over 30 years of operations and engineering expertise in satellite engineering and software development. It provides a flexible and accessible solution by defining simple telemetry input and command output formats, allowing any team to integrate it into their mission architecture. The software features configurable real-time telemetry displays, automated anomaly detection based on user-defined alarms, and leveraging user-defined command sequences and health checks to automate nominal passes. A simple text file-based command interface facilitates integration with external mission planning systems. These features aim to improve efficiency of commanding time and reduce the need for around-the-clock human involvement during passes, offering key advantages for university teams with limited staffing and resources. Future development of ABSCS is focused on supporting more input and output formats, and supporting existing standards for telemetry and commands. ABSCS is planned to be used in operations for AlbertaSat's 5th satellite, Ex-Alta 3, launching in 2026.

Keywords: C2, CubeSat operations, Anomaly detection, Pass automation, Student groups

Acronyms/Abbreviations
ABSCS: AlbertaSat Control Software
C2: Command and control
CSA: Canadian Space Agency
GMOC: Generic Mission Operation Center
IO: Input/Output
LEOP: Launch and Early Operations Phase

1. Introduction

AlbertaSat is a volunteer student group at the University of Alberta, with the primary goal of training highly qualified personnel by allowing student access to space missions. Currently AlbertaSat is designing and building a 3U CubeSat as part of the Canadian Space Agency's CUBICS program, an initiative to enable students at Canadian universities to design, build, and operate CubeSats. As part of preparation for operations several students from AlbertaSat have created ABSCS. ABSCS is a free, open-source, web-based C2 software tailored specifically for simple integration with low-budget satellite missions. Source code, user and developer documentation can be found at the GitHub repository in [1]. Volunteer student groups are plagued by issues that are not typical of traditional organizations, notably vastly high member turnover, limited knowledge retention, and unpredictable personnel resources. For operations which may last years, these issues are worrisome. For AlbertaSat's previous satellite missions, bespoke and barebones operations software has been created close to launch, often as an afterthought to the design of the satellite. This is disadvantageous for several reasons; creating bespoke software for each mission means a recurring stressor on the group's resources. Additionally, due to high member turnover, lessons learned from previous mission's operations are not captured by the time the next mission's software is created. Thus, for those reasons and the authors' time interning with Calian's Satellite Operations team at the Canadian Space Agency (CSA) motivated the creation of a

general C2 software that could be used for future missions, like the GMOC architecture used at CSA. While open-source options like OpenC3 COSMOS already exist; the desire to incorporate features from the lessons learned at CSA motivated the creation of a new software from scratch. In particular, anomaly signature detection; the integration of response to anomalies procedures into the C2 interface itself, and nominal pass automation. These features are described further in section 2, and align to address the operational difficulties inherent to student groups that were stated above. Finally, one of AlbertaSat's key objectives is to develop open-source technologies, which motivated the publicization of the project under the Apache 2.0 license.

2. Capabilities

2.1. Defining Mission Configurations

At a high-level, setting up ABSCS involves the creating of different software configurations called "Missions". Once a Mission has been created; further configuration data is defined relative to that Mission and stored in a mission-specific configuration database. Thus, if an organization is operating multiple satellites concurrently, only one instance of ABSCS would be needed, allowing for reusability of the rest of the ground segment between satellites. After creating a mission, the first step is to create its telemetry mnemonics. Alarm limits associated with specific mnemonics can then be created; these will be checked when a telemetry is received and will generate alarms for any mnemonic that is out of limits. Commands are also user-defined and are composed of a name, along with any number of parameters. Parameters may be configured as required or optional and can have associated input validators with them. A series of default validators are given, but user-defined validators written in Python may also be used.

2.2. IO

Satellite input/output (IO) is currently done through a TCP/IP connection. Connection Profiles representing a satellite IO source can be added to a Mission and selected for use on the homepage. One Connection Profile can be used at a time, and it is shared across all operators connected to the system. Presently, the only input to the system is telemetry. Incoming telemetry must adhere to a JSON format that is given in the documentation. Likewise, commands are outputted in a given JSON format. Since most low-budget CubeSat teams do not use standards like CCSDS, it was decided that defining simple, general formats for IO would permit the easiest uptake of ABSCS for teams wishing to use it. A translation module to format the incoming telemetry and outgoing commands is typically needed, but implementation is left flexibly to the user. An option is given to write one operating internally in the ABSCS webserver or it can be done externally at some point in the ground segment. IO of other natures, such as integrating ABSCS with other parts of the ground segment is typically done through HTTP API endpoints. Much of future work is centered around expanding satellite IO capabilities, both in increasing supported connection types, and providing default translation modules to support plug and play for missions that do adhere to preexisting standards. See Section 4 for more details.

2.3. Basic operation using Operation Pages

Operation Pages are the primary displays used by operators during passes. They can be created and deleted at will. They consist of a configurable selection of live-updating telemetry mnemonics, incoming alarms, and a commanding section. Owing to the web-based approach, multiple pages can be open on any number of computers connected to the ABSCS webserver, thus for intensive operations like Launch and Early Operations Phase (LEOP), the number of active operators can be scaled easily. Operation Pages should be configured prior to a pass to show desired telemetry.

2.4. Anomaly Detection

Anomalies are defined as a specific combination of alarms and conditions. When an alarm is generated or when requested, the anomalies database will be checked against the current state of telemetry, alerting the operator of any matches. Response to Alarms is an operational structure in use for operations at the CSA, whereby specific alarm states are correlated with known anomalies and have a recovery procedure associated with them. The alarm state and associated recovery for an anomaly are currently found by an operator manually searching a binder. While this is functional, it could be automated as part of the C2 software. Thus, in a structure based on the Response to Alarms methodology in use at the CSA, documents and command sequences can be associated with anomalies, enabling operators to quickly resolve known anomalies by confirming anomaly presence and running associated recovery command sequences, reducing the time wasted during a real-time pass. Additionally, new or unknown anomalies can be quickly identified as not matching a known signature in the Response to Alarms, helping to trigger root-cause and recovery investigations at an operational level.

2.5. Commanding and Pass Automation

Executing commands can be done in two ways: single commands and command sequences. Single commands can be sent via operator input on Operations Pages. Command sequences are text-files adhering to simple syntax rules allowing for efficient use of commanding time. Basic logic flow exists and will send out commands according to the rules in the file. Routine pass automation a significant advantage to the other software currently available. A series of user-defined conditions is checked at the beginning of a pass to validate satellite health. Pending good health, command sequences can then be scheduled to run during the pass, with configurable checks on satellite health between sent commands.

3. Material and Methods

3.1. Programming Languages and Frameworks

As stated above a goal of the project is to be web-based and open-source. As a result, ABSCS is built using HTML, CSS and JavaScript for the frontend while the backend was constructed with Python using the Django web framework. Git was utilized to version control the development of the application and publish the code onto GitHub.

3.2. Database Design

ABSCS uses SQLite for storing mission configuration data and long-term telemetry storage. The software uses a separate database for each separate mission. As a result, when a new mission is created a new SQLite database is constructed and all data relating to that specific mission will be stored in that new database. In terms of database structure, ABSCS models follow an entity-relationship data approach to store the software's information. The following is the relationship schema (Figure 3.1):

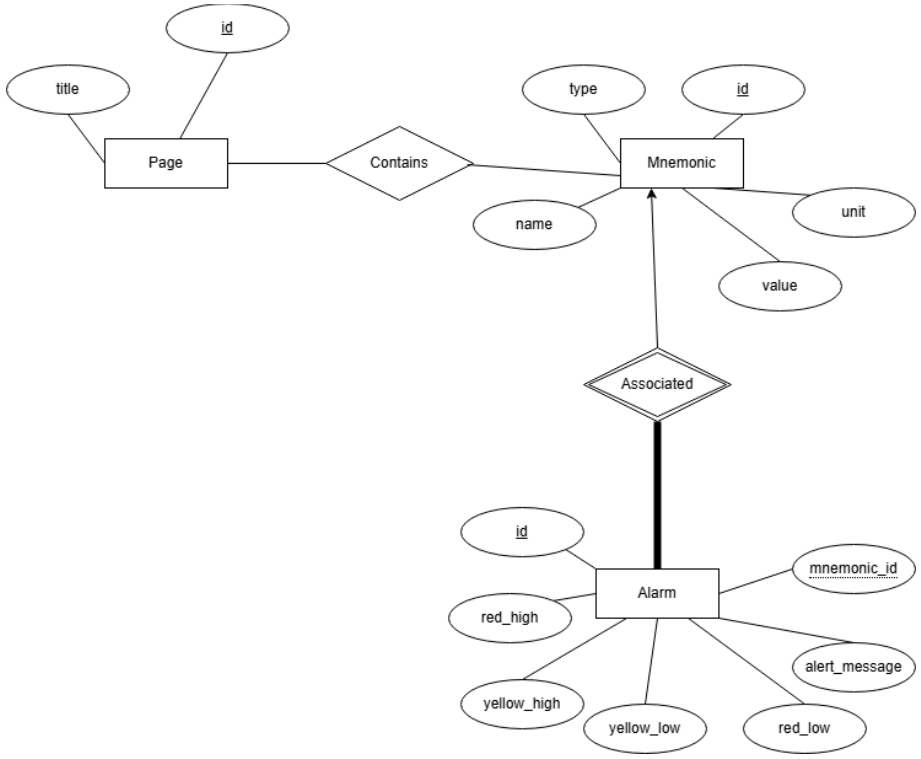

Figure 3.1*: Entity-Relationship (ER) diagram of a mission for ABSCS

*This will change in the future as more features are added onto the software.

3.3. Software Architecture

For the client side of the application, ABSCS makes use of Django's templates to dynamically construct user interfaces written in HTML, CSS and JavaScript. These interfaces will send HTTP requests to the local server and receive incoming data from the server to:

- Update mnemonic/page data
- Establish a WebSocket connection with the local server
- Utilize WebSockets to receive and display real-time data
- Send commands and set up automated commands
- Request server to swap active Mission/mission database

For the server side of the application, ABSCS utilizes Python with Django create RESTful API interfaces for the client to send requests. These interfaces will expect specified requests, as documented on the GitHub repository, and operate to:

- Update mnemonic/page entities within database
- Handle converting and transmitting user commands over TCP
- Receiving and sending telemetry to the frontend via WebSocket
- Setting up WebSocket with the frontend
- Sending commands and setting up automated commands
- Swapping mission database

3.4. Continuous Integration and Testing

While developing the project, continuous integration was a high priority as to identify and minimize the number of software bugs, as well as reduce the testing, review, and deployment time for new updates. This was accomplished through creating a GitHub workflow that would execute a series of automated tests when a developer pushes any changes to the main branch. Manual testing is used to cover areas of the software that are not tested by the automated tests. This ensures previously committed code functions as expected while adding/changing features and potentially isolate sources for which a bug can occur. These tests include:

- API calls and ensuring that the proper response is created for different scenarios.
- Unit tests of crucial functions produce outputs we expect when given a specified input. Tests include checking "regular" inputs and edge cases.

4. Discussion and Future Work

Through the development of the software, we highlight some interesting findings behind our implementations. First, establishing a new database per mission design with an entity-relationship data model has allowed for well compartmentalized data. With this being an open-source project, having well organized data lowers the barrier for users to modify ABSCS to their liking as information is clearly defined, does not overlap, and allows for other features to be implemented to the system by adding more entities. Furthermore, as mentioned before ABSCS's database of choice was SQLite as it comes packaged with Django. During our testing of the software SQLite held up well with active data manipulation with a small group (2 users). This is as expected as SQLite boasts its "emphasizes economy, efficiency, reliability, independence, and simplicity," [2] making it an ideal choice for ABSCS's simple and open-source nature. However, this does not come without its drawbacks. Mainly when running this software in extremely high stress environments with many users writing data to the database at once. SQLite states that "[their database engine] strives to provide local data storage for individual applications and devices ... SQLite does not compete with client/server databases." [2]. As a result, our current database solution could be improved in the future for better concurrency enabling increased writes and reads to the database. Solutions include switching to a more client/server focused database such as PostgreSQL, which is also open-source, or implementing a queue system that can restrict concurrent writes from occurring on the SQLite engine.

Another consideration around ABSCS is the overall security of the software. The ABSCS team leaves security considerations to the end user and does not claim to provide a secure-by-default system. While the software has been developed with reliability and usability in mind for trusted, small-team environments, it has not undergone formal

security audits and does not include features to protect against common attack vectors such as traffic interception or denial-of-service (DoS) attacks. Users intending to deploy ABSCS in networked or exposed environments are strongly encouraged to implement additional security measures—such as HTTPS, firewalls, authentication layers, or reverse proxies—appropriate to their mission requirements. As the developers are not security professionals, it would be irresponsible to guarantee the security of the system beyond the local, closed-network use cases it was designed for. We believe transparency in this regard is essential and hope future contributors may expand ABSCS to address these needs more comprehensively.

Recognizing the current capabilities of ABSCS allows us to clearly define priorities for future development. One major focus is expanding the range of supported communication protocols. At present, ABSCS supports only TCP/IP, but many CubeSat teams and hobbyists may use other communication methods. To improve compatibility and accessibility, we plan to add support for additional protocols, starting with HTTP, and extending to hardware protocols in future to enable easier integration of ABSCS for hardware-in-the-loop and FlatSat testing.

In addition to protocol support, we also intend to implement translation modules for common standards used within the space industry. Currently, users must manually define their command and telemetry formats. With pre-made translation modules, users will be able to leverage pre-defined or community-shared standards, reducing setup time and improving interoperability across missions.

5. Conclusions

With the cost of satellites reducing year over year; it is becoming more and more viable for smaller organizations to develop and launch space missions. For university teams where resources come from fluctuating volunteer involvement and are often second priority to schooling, running around-the-clock operations poses significant challenges. ABSCS serves to fill a niche of C2 software geared towards facilitating operations for these groups. Features like anomaly detection seek to allow mission knowledge to be consolidated into the software itself, helping with high personnel turnover and improving anomaly response times. Pass automation intends to alleviate the burden of staffing operations around the clock while allowing enough flexibility in automation conditions to ensure satellite health. ABSCS is available under the open-source Apache 2.0 license at a GitHub repository. [1]

Acknowledgements

The authors of the paper would like to acknowledge the Calian Advanced Technologies Satellite Operations team for sharing their wealth of experience. Additionally, the Canadian Space Agency for the creation and support of the CUBICS program allowing students to participate in hands-on space missions. The teams involved with both opportunities were integral to this project's formation and success and the authors would like to thank them for their aid.

References

[1] T. Cundict, M. Eissa, J. Fernhout, ABSCS GitHub Respository, 4 April 2025, https://github.com/tcundict/ABSCS, (accessed April 4th, 2025)

[2] SQLite Authors, Appropriate Uses For SQLite, https://sqlite.org/whentouse.html, (accessed April 2nd, 2025)