Arduino 互換ミッション OBC 用のソフトウェア開発
ーー抽象化とリプログラミングーー

堀口 淳史[*1]・橋本 論[*2]・中澤 賢人[*3]・久保田 晃弘[*4]

Development of Software for the Arduino-compatible Mission OBC
Abstraction and Re-programming

Junshi HORIGUCHI [*1], Ron HASHIMOTO [*2], Kent NAKAZAWA [*3], Akihiro KUBOTA [*4]

ABSTRACT

This paper describes the design philosophy and implementation details of the software of “Morikawa” which is a mission OBC of 1U CubeSat Art Satellite “INVADER”. On February 28 2014 (JST) INVADER was launched as a piggyback payload of the H-IIA launch vehicle No.23 and entered a circular non-sunsynchronous orbit at an altitude of 378 km and an inclination of 65 degrees. Morikawa is a open-source hardware Arduino compatible mission OBC. Developers can use the base software, extension libraries and the development environment that were cultivated in the Arduino community. Morikawa's hardware modules are abstracted consistently as much as possible, hence the coders can write programs to use them very easily and freely. In addition, a virtual machine is implemented by defining a original machine language on Morikawa using its strictly designed interface. The VM enables us to re-program Morikawa efficiently by sending only small bytecode from ground station. Last of all, some results of missions executed by Morikawa on orbit are reported.

Keywords: Art Satellite, CubeSat, Arduino, Programming, Software, C++, Virtual Machine

概要

1U CubeSat の芸術衛星「INVADER」に搭載されたミッション OBC「Morikawa」のソフトウェアの設計思想と実装の詳細について述べる. INVADER は 2014 年 2 月 28 日に H-IIA 23 号機の相乗り衛星として, 高度 378 km, 傾斜角 65 度の太陽非同期軌道に投入された. Morikawa はオープンソースハードウェアの Arduino 互換であり Arduino のコミュニティーで培われた基盤ソフトウェア, 拡張ライブラリや開発環境をほぼそのまま利用することができる. ハードウェアの実装をできる限り抽象化することで各種記憶素子をほぼ同一の手順で利用できるよう配慮した. さらにインターフェースを厳密に定義することで Morikawa 上に独自のマシン語を定義し, Virtual Machine (VM) を実装することが可能になった. この VM を使って少ないデータ転送量で効率的に軌道上でリプログラミングを行うことができる. 最後に Morikawa を用いて宇宙空間で実行したミッションの成果について報告する.

[*1] 多摩美術大学 × 東京大学 ARTSAT PROJECT (Tama Art University x Tokyo University ARTSAT PROJECT)

[*2] 多摩美術大学 (Tama Art University)

[*3] 多摩美術大学 × 東京大学 ARTSAT PROJECT (Tama Art University x Tokyo University ARTSAT PROJECT)

[*4] 多摩美術大学 (Tama Art University)

1 はじめに

超小型衛星の打ち上げ機会の増加にあたり, 大型衛星の専門家だけでなく大学の研究室やアマチュアグループによる小型衛星の開発が増えつつある. このような小型衛星の 1 つに約 10 cm 角の 1U CubeSat という衛星規格が存在する. 1U CubeSat は物理的な実験機器や動作機構, 推進機を搭載するには容積に制約があるが, 近年の電子部品の小型化, 軽量化により, センサの情報を収集したり宇宙空間でソフトウェアの動作実験を行うには十分な大きさがある. しかしながら衛星の開発を行うためには, ソフトウェアのみならず, 構造設計・熱設計・放射線耐性の検討・電子部品の選定など広範囲な分野の専門知識が必要であり, これらを総合して設計開発を行う必要がある.

近年は衛星の設計をモジュール化し初期導入コストを軽減する試み[5]が行われているが, それは主に独自ハードウェア分野での取り組みである. それに対して, 芸術活用を目的とした芸術衛星「INVADER」[6]では, ユーザに開かれた柔軟性のあるソフトウェアをベースとしたミッションを設定した. ソフトウェアの動作実験を主なミッションとする衛星にとっては, ハードウェアはオープンソースで多くの分野の非専門家に用いられているものが良い. そのため, ソフトウェアの動作環境としては芸術デザインの分野でも広く用いらている Arduino[7] を選択する. さらにプラットフォームのモジュール化を行うとともにソフトウェアのライブラリ化を実施し, ミッション関連機器を再利用可能なものとした.

芸術衛星 INVADER はパワーモジュール・メインモジュール・ミッションモジュールという 3 つの大きなサブシステムにより構成される. パワーモジュールは衛星全ての電力管理を行っており最も権限が強く, メインモジュールは地上局との通信やミッションモジュールの起動など衛星全体のオペレーションを行う. ミッションモジュールはメインモジュールに従属する形で動作する Arduino 互換の実験用ソフトウェア実行環境である.

本論文では, 芸術衛星 INVADER に搭載されている宇宙空間で動作する Arduino 互換ミッションモジュールの設計とリプログラミング可能なソフトウェアの実装方法について述べる.

2 芸術衛星のミッションモジュール

2.1 芸術衛星としての機能

芸術衛星である INVADER 衛星を設計するにあたり 1U CubeSat に搭載するべき必要最低限の機能を検討するとともに芸術目的の利用を行うために何が必要かを検討した. 情報芸術・メディアアート分野の作家や学生の意見を交えながら現実的に衛星に搭載できる機能を選定し, 情報の可視化やジェネラティブアートのパラメータとして利用できるようにセンサデータを取得する機能が搭載された.

「みんなの衛星・感じる衛星・美しい衛星」というコンセプトをもとに FM パケット通信だけではなく音声信号による地上へのメッセージ送信機能も搭載された. 音声信号には PWM で生成した音階を演奏する機能と音声合成チップ[8]を利用した日本語発音機能が含まれている. センサとしては, 太陽電池パネルの発電電力・太陽電池パネルの温度・リチウム蓄電池の充放電電力・リチウム蓄電池の温度・各種バス系統の電流電圧や温度・磁気・ジャイロの値を取得することができる. これらのセンサは衛星の健康状態の監視にも利用することができ衛星の少ないスペースを有効的に活用できる. さらに最大 640 * 480 ピクセルの CMOS カメラを搭載しており解像度は低いながらもカラー写真を撮影することができる.

これらの機能は過去の衛星開発でも採用されており目新しいものではないが芸術活用のためのパーソナル・メディアとしての INVADER 衛星では理論ではなく実際に自分の耳で聴き, 自分の手でシャッターを切り, 自分の目で地球を見るというプロセスが大切であると考え, 確実に動作する機器を採用しソフトウェアの力によって可能性を広げる手法を選択した.

2.2 Arduino の採用

2.2.1 Arduino の概要

情報芸術分野においてハードウェアでは Arduino, ソフトウェアでは Processing[9] や openFrameworks[10] の利用者が増えつつある. これらは高度な技術を持った技術者だけではなく技術に興味のある子供や DIY 者などにも利用できるように設計された初めての本格的なモジュールである. これらを利用した開発はラピッドプロトタイピングと呼ばれ, 実現したいアイデアを少ない知識と時間で可能にする.

Arduino では回路図は公開されソフトウェアは GPLv3 や LGPL ライセンスで提供されている. openFrameworks でもソースコードは MIT ライセンスで提供されておりオープンソースなプロジェクトである. 基盤部分がオープンソースであることによりこれらの環境をサポートしたサードパーティー製の製品や有志によ

[5] ほどよし SDK や SH4-BoCCHAN-1 OBC 等

[6] http://artsat.jp/project/invader

[7] http://arduino.cc

[8] 株式会社アクエストの AquesTalk pico

[9] http://processing.org

[10] http://openframeworks.cc

るライブラリも数多く存在し再利用可能なエコシステムが成立している.

これらの文化を踏まえ INVADER 衛星ではミッションモジュールを完全に独自開発するのではなく Arduino の資産を利用する. Arduino を採用することにより Arduino を利用したことのある者であれば過去に作成したプログラムをそのまま宇宙で動作させられる機会を提供することができる.

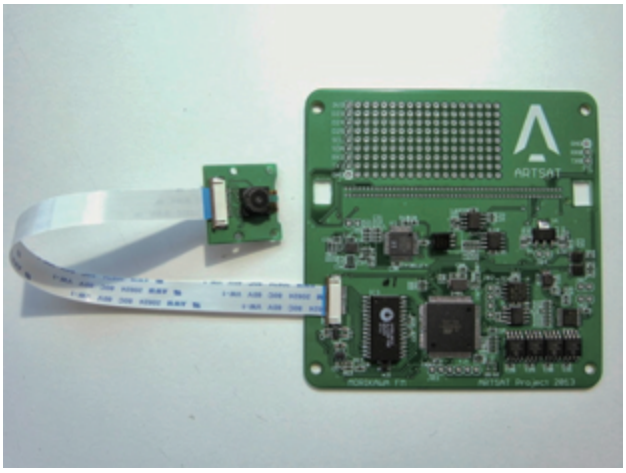


図 1 ミッションモジュール Morikawa

INVADER 衛星には【図1】に示す Arduino 互換のミッションモジュールが搭載され, これを Morikawa と呼ぶ. メイン CPU として ATMEL の ATmega 2560 を採用しており, Arduino Mega 2560[11] に相当し Arduino の開発環境[12]を用いて Morikawa のソフトウェアを開発することが可能である.

2.2.2 Arduino の特徴

Morikawa は Arduino 互換であるために一般的な Arduino の特徴を継承する.

以下に Arduino の標準的な特徴をまとめる.

- リアルタイム OS やその他のオペレーティングシステムを利用しないためにリアルタイム処理を記述することができる
- シングルタスクであり割り込みを利用する
- 外部メモリを搭載せずに CPU 内部の SRAM や EEPROM を利用する
- Arduino Mega 2560 の 5 V 系は 16 MHz, 3.3 V 系は 8 MHz で動作する
- ハードウェアやソフトウェアは冗長性を持たない

2.2.3 Arduino の開発環境

Arduino のソフトウェアは Arduino IDE を用いて開発される. Arduino IDE はソースコードの記述からコンパイル・リンク・実行バイナリのハードウェアへの書き込みまで全ての工程を簡単に行うことができる. Arduino IDE はコンパイルとリンクに内部で avr-gcc を利用しており, avr-gcc を直接利用した Arduino ソフトウェアの開発も可能であるが, Morikawa では Arduino IDE を用いて全ての開発を行う.

Morikawa は Arduino 1.0.5 を対象として開発された. Arduino 1.0.5 は内部では次のような機能を利用している.

- コンパイラは avr-g++ (GCC) 4.3.2 である
- リンカは avr-ld (GNU Binutils) 2.19 である
- アーカイバは avr-ar (GNU Binutils) 2.19 である
- ABI は -mmcu=atmega2560 を指定する

2.3 Morikawa ハードウェアの設計

2.3.1 設計の概要

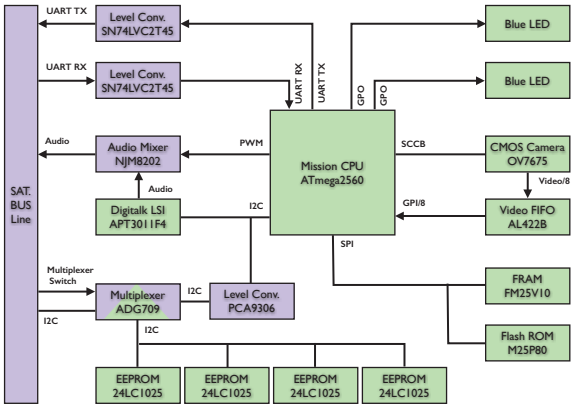


図 2 Morikawa のシステムダイアグラム

Morikawa の設計の中心は【図2】に示すようにメイン CPU である ATmega 2560 である. Morikawa は地上で単独動作する Arduino とは違い, 衛星全体を管制するメインモジュールや電力システムを管理するパワーモジュールと協調して動作しなければならない. 衛星の電池残量が少なくなれば計画的に強制終了される可能性もありデータの読み書きの安全性なども検討項目となる.

これらを解決するために Arduino の基盤ソフトウェア[13]に改造を加える方法が検討されたが, 標準的な Arduino の開発環境を利用できなくなることや標準環境を前提に構築されたエコシステムの恩恵を受けることが難しくなる. そこでハードウェアを構成する部品は一般的に入手可能な民生品を利用し, ソフトウェアにおいても可能な限り基盤ソフトウェアには手を加えない手法が取られた.

---

[11] http://arduino.cc/en/Main/ArduinoBoardMega2560

[12] Arduino IDE（執筆時のバージョンは 1.0.5）

[13] <Arduino>/hardware/arduino/cores/arduino/ フォルダ以下のファイル群

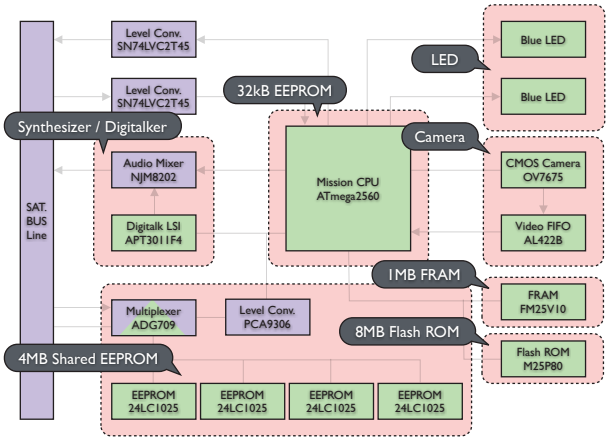


図 3　Morikawa のペリフェラル機器

メイン CPU である ATmega 2560 を中心に配置し，センサ，データ格納用メモリや音声合成 LSI などをペリフェラル機器として接続した【図3】．さらに，Morikawa は能動的にメインモジュールやパワーモジュールと連携するのではなく，メインモジュールに従属する受動的な設計とした．Morikawa もまたメインモジュールのペリフェラル機器の 1 つとなる．このような設計ではペリフェラル機器ごとにハードウェアとソフトウェアをモジュール化することが可能となり，機器の差し替えや追加を容易に行うことができ INVADER 衛星の開発だけではなく後の衛星の開発においても資産を生かすことが可能となる．

【表1】と以下に Morikawa の特徴をまとめる．

- Arduino Mega 2560 にペリフェラル機器を接続したものと同等である
- メインモジュールに従属し電源管理はメインモジュールが行う
- 秋葉原や digikey などを通して購入できる部品で構成され MIL 規格の部品を利用しない
- 標準的な Arduino の基盤ソフトウェアと開発環境を利用する

表 1　Morikawa ハードウェア

分類	詳細
CPU	ATmega 2560
電源電圧	3.3 V
動作周波数	8 MHz
部品規格	民生品
揮発性メモリ	内臓 SRAM
不揮発性メモリ	内臓 EEPROM, 外部 FRAM, 外部 FlashROM, 外部 EEPROM
OS	利用しない
タスク処理	シングルタスク
割り込み	利用する
開発環境	Arduino IDE 1.0.5

- タイマー・SPI・I2C 通信などのソフトウェアは標準 Arduino 用に提供されているライブラリ[14]を可能な限り利用する

2.3.2 メインモジュールとの協調

Morikawa はメインモジュールに従属する設計であるがメインモジュールとの間でテレメトリデータの伝達や状態の通知といった通信を行う．Morikawa とメインモジュールの間はシリアル通信 (UART) によって接続されており，具体的には Arduino Mega 2560 の Serial1 を利用する．

標準的な Arduino のシリアル接続を扱うクラスでは開発者は割り込み処理を記述せずに受信データのポーリングを実施する．Morikawa の設計では標準的な機能を積極的に利用するためにシリアル接続に関しても受信データのポーリングを行う．

通常受信データは loop() 関数内で取得されるが，メインモジュールとの通信ではデータの取りこぼしを避ける必要があるために，loop() 関数内ではなくタイマーによる一定間隔の割り込み処理の中で確実なポーリングを行う．ポーリング用のタイマーには Arduino Mega 2560 の Timer1 を利用する．

Morikawa とメインモジュールの間では【表2】のようなコマンドが送受信される．通信の開始権限は双方が保持しており通信に関しては対等の関係性である．

表 2　シリアル通信で伝達されるコマンド

コマンド	方向	役割
c-c-m-eco	メイン→Morikawa	存在確認
r-m-c-eco	Morikawa→メイン	上記への返答
c-c-m-smm	メイン→Morikawa	テレメトリデータの伝達
c-c-m-asd	メイン→Morikawa	強制終了予告の伝達
c-m-c-nsd	Morikawa→メイン	正常終了の伝達
c-m-c-don	Morikawa→メイン	オーディオ回路の電源オン要求
r-c-m-don	メイン→Morikawa	上記への返答
c-m-c-dof	Morikawa→メイン	オーディオ回路の電源オフ要求
r-c-m-dof	メイン→Morikawa	上記への返答

2.3.3 共有メモリを利用した協調

Morikawa とメインモジュールの間でシリアル通信を介して伝達されるデータは限られた種類のデータのみである．Morikawa に搭載されたアプリケーションが生成した個別のデータは共有メモリに書き込まれメインモジュールに引き渡される．また，メインモジュールが Morikawa を起動する際に起動パラメータを伝達する目的にも利用され，共有メモリの一部が特別に割り当てられている．

共有メモリは Morikawa が起動されている間は Morikawa が読み書きの権限を保持しており，Morikawa

[14] TimerOne ライブラリ https://code.google.com/p/arduino-timerone

とメインモジュールは排他的に共有メモリにアクセスする.

2.4 Morikawa ソフトウェアの考察

Morikawa の基盤ソフトウェアには標準 Arduino 用のソフトウェアを利用することができるが，Morikawa は 3.3 V 系で動作しており 5 V 系 Arduino Mega 2560 用の基盤ソフトウェア[15]をそのまま利用することはできない. そこで SparkFun Electronics[16] が公開し提供している Arduino Mega Pro 2560V 3.3 V 用のハードウェア定義ファイル[17]を Arduino の開発環境に追加[18]する.

Morikawa のペリフェラル機器を操作するソフトウェアはペリフェラル機器 1 種類につき 1 つの C++ 言語クラスとして提供する. Morikawa の基本機能を操作するインターフェースとペリフェラル機器を操作するインターフェースを統合した Morikawa クラスをアプリケーションプログラミングインターフェース (API) として開発者に提供する.

Arduino のプログラミングでは setup(), loop() という 2 つの関数がとても重要な意味を持っており，Arduino の開発者は慣れ親しんだ標準的な流儀に従ってプログラミングできることを期待している．そこでユーザインターフェースとなる Morikawa クラスも標準の流儀に従う形で定義し実装する必要がある.

さらに Morikawa はメインモジュールとテレメトリデータの受け渡しを行っているためテレメトリデータの送受信機能の実装とテレメトリデータを取得するインターフェースの提供が必要である．テレメトリデータの送受信機能の実装は Morikawa クラスを利用する開発者からは隠蔽し抽象化する. Morikawa ソフトウェアは具体的なアプリケーションの中身については実装せず，可能な限り抽象的な形でハードウェアのすべての機能を操作できるように提供し，これらをまとめて MorikawaSDK[19] として提供する.

3 MorikawaSDK の実装

3.1 MorikawaSDK の概要

これまでの考察に基づき MorikawaSDK には【表3】のクラスが含まれる. TSTMorikawa クラスは最も重要なクラスであり Morikawa アプリケーションの開発では主にこのクラスを利用する.

TSTFRAM, TSTFlashROM, TSTSharedMemory, TSTLED, TSTTone, TSTDigitalker, TSTCamera クラスはそれぞれ ATmega 2560 に接続されたペリフェラル機器を表しており読み書きやその他の操作を行うための機器固有の実装を含んでいる【図4】. ペリフェラル機器のクラスは TSTMorikawa クラスを通して利用する設計でありアプリケーション開発者が直接利用することはできない. これによりハードウェアの部品が入れ替わったり実装の詳細が変更された場合でもアプリケーションレベルではソフトウェアの変更を行うことなく対応することができる.

TSTCriticalSection クラスは ATmega 2560 のハードウェア割り込みを一時的に禁止するためのクラスであり C++ 言語を用いて実装されているのでデストラクタで自動的に割り込みの状態を復元することができソフトウェアの安全性が向上する.

TSTTrinity クラスはプリミティブ変数の 3 冗長化を行うクラスであり C++ 言語のテンプレート機能を用いて実装されている. 例えば int i; という整数型を

表 3 MorikawaSDK に含まれるクラス

クラス名	役割	直接利用可能
TSTMorikawa	Morikawa を操作する基本クラス	はい
TSTFRAM	1M bits FRAM を読み書きするクラス	いいえ
TSTFlashROM	8M bits Flash ROM を読み書きするクラス	いいえ
TSTSharedMemory	4M bits Shared EEPROM を読み書きするクラス	いいえ
TSTLED	Morikawa 基板上の LED を操作するクラス	いいえ
TSTTone	ATmega の PWM を利用して音階を演奏するクラス	いいえ
TSTDigitalker	AquesTalk 音声合成 LSI を操作するクラス	いいえ
TSTCamera	OV7675 CMOS カメラを操作するクラス	いいえ
TSTCriticalSection	割り込み禁止領域を管理するクラス	はい
TSTTrinity	プリミティブ変数の 3 冗長化を行うクラス	はい
TSTSCCB	SCCB プロトコル通信を行うクラス	はい

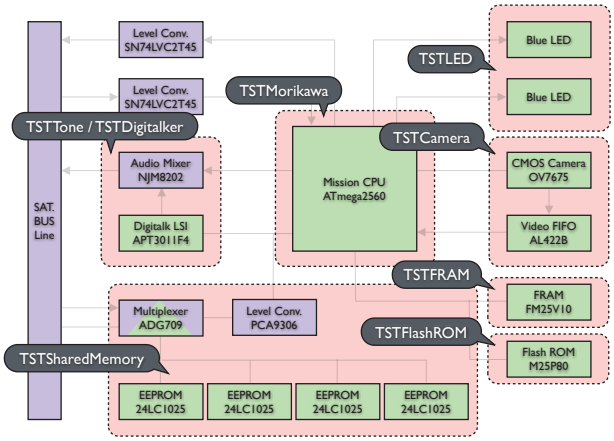


図 4 ペリフェラル機器を担当するクラス

[15] Arduino IDE から選択できる Arduino Mega 2560 or Mega ADK

[16] http://www.sparkfun.com

[17] http://dlnmh9ip6v2uc.cloudfront.net/datasheets/Dev/Arduino/Boards/mega-pro-3.3V-v11.zip

[18] <User Document>/Arduino/hardware フォルダ以下に追加する

[19] http://github.com/ARTSAT/MorikawaSDK

TSTTrinity<int> i; と変更すると 3 冗長化された変数にアクセスすることが可能となる.

TSTSCCB クラスは OV7675 と通信するための SCCB プロトコル[20]を実装した[1]クラスである.

MorikawaSDK は次のような仕様に基づき実装を行い高度に抽象化されたアプリケーションプログラミング環境を提供し, ソフトウェアが確実に安全に宇宙空間で実行されることを目指した. これらの仕様と抽象化は同時に Virtual Machine の実装を容易にすることにも寄与している.

- MorikawaSDK は C++ 言語を用いて実装する
- 完全なカプセル化を実施するためにクラス変数は private 変数のみとする
- コンストラクタ / デストラクタ・set / get・read / write・setup / cleanup / isValid 系関数などの仕様を統一化する
- 実行時エラーは可能な限り補足する
- 関数が内部でエラーを発生させた場合には, 呼び出し時の引数の内容を維持する
- ソフトウェアフローの中で重要な判定を行う変数は 3 冗長化する
- FRAM, Flash ROM, EEPROM などの各種記憶素子の読み書きインターフェースを抽象化する
- メインモジュールからの強制的な終了に備えて時間のかかる処理には中断機能を実装する
- エラー発生時にエラーの内容を地上に伝える手段を提供する

3.2 C++ 言語を用いた実装

MorikawaSDK は C++ 言語を用いて実装されているが, これは Arduino の基盤ソフトウェアが C++ 言語を用いて実装されていることに起因する. C, C++ 言語は実行時オーバヘッドが少なく低性能の環境でも効率的にソフトウェアを実行することが可能[2]である. またアセンブラよりも人間に理解しやすい記述方法なので不本意な不具合を組み込む可能性も軽減される.

C++ 言語には仮想継承, 仮想関数, typeid, dynamic_cast やテンプレートなどのオブジェクト指向プログラミングのための機能が備わっているが仮想継承や仮想関数はメモリ使用量の増加と実行時オーバーヘッドの増大を生じさせる. typeid や dynamic_cast などの実行時型情報をサポートするにはコンパイラの設定を変更[3]する必要がある. 同様に例外を利用する場合にもコンパイラの設定の変更[4]と ABI のリターゲットが必要になる. これらの機能は ATmega 2560 のような 8 ビットマイコンではできるだけ使用しない方が確実な動作が期待できるため MorikawaSDK では利用しない.

3.3 完全なカプセル化

アセンブラや C 言語を用いた 8 ビットマイコンのプログラミングでは変数や関数のカプセル化を実施することは稀であり, 機能がオブジェクト化されず再利用性の低いソフトウェアとなる傾向がある.

Arduino 互換の環境では C++ 言語が利用できるため C++ 言語のカプセル化機能を積極的に利用する. MorikawaSDK ではメンバ変数はすべて private 変数とし大域変数は一部の変数を除いてオブジェクトファイルをスコープの範囲とする. メンバ関数についても private・protected・public の順に優先的に利用し積極的に非公開メンバ関数とする.

3.4 関数の仕様の統一化

3.4.1 コンストラクタ / デストラクタ

MorikawaSDK では例外を使用しないのでコンストラクタやデストラクタの中で発生したエラーを外部に搬出する手段が存在しない. これらの内部では状態管理用の変数の初期化などエラーを発生させない処理のみを記述する.

3.4.2 set / get 系関数

set / get 系関数は主として private なメンバ変数へのアクセッサを提供する. MorikawaSDK ではペリフェラル機器の状態を操作する関数も含まれるためそのような関数ではエラーを適切に返す必要がある.

単純なアクセッサ

- void setXXX(typename const& param);
- typename const& getXXX(void) const;

エラーを返すアクセッサ

- TSTError setXXX(typename const& param);
- TSTError getXXX(typename* result) const;

3.4.3 read / write 系関数

read / write 系関数は主としてデータ格納用メモリなどの記憶素子への読み書き機能を提供する. ペリフェラル機器との通信ではエラーを発生させる可能性があるためにこれらの関数はエラーを返す必要がある.

- TSTError writeXXX(…);
- TSTError readXXX(…);

3.4.4 setup / cleanup / isValid 系関数

コンストラクタやデストラクタでは変数の初期化など単純な操作のみ可能であるためエラーを発生させる可能性のある操作は setup / cleanup 系関数で行う.

cleanup 系関数はインスタンスの解放時やアプリケーションの終了時に呼び出されるためエラーを返してもエラー状態から回復する有意義な方法が存在しない. そこで

[20] OmniVision Serial Camera Control Bus

cleanup 系関数は常に成功するものと仮定しエラーを返さない仕様とする.

isValid 系関数は setup / cleanup 系関数の呼び出し状態を判定するための関数であり setup 系関数の成功から cleanup 系関数の呼び出しまでのあいだ true を返し, それ以外の場合は false を返す.

- TSTError setup(…);
- void cleanup(void);
- bool isValid(void) const;

3.4.5 関数のコーディング規約

MorikawaSDK では上記の特別な関数以外においても次のような規約を採用している.

- 関数名の先頭は小文字で始まり 2 単語目からは先頭が大文字である（getData / getAngleX）
- R.A.M. や R.O.M. などの略語はすべて大文字とし関数名の先頭には用いない（getFRAMSize）
- position を pos などのように 1 単語を略さない
- インスタンスに対して操作を行う関数は動詞の原型で始まる（findItem / update）
- インスタンスの状態を取得する関数は is + 形容詞や動詞の三人称単数形 + 名詞の単数形とし bool 型を返す（isValid / hasUpdate）
- ハードウェア割り込みから呼び出される関数は on で始まる（onReceive / onTimer）
- インスタンスの内容を変更しない関数は const 関数とする
- 関数の引数にプリミティブ型以外のインスタンスを渡し, 関数が引数の内容を変更しない場合は const 参照渡しとする[21]
- 関数が引数の内容を変更する場合はポインタ渡しとする[22]
- 関数がエラーを返す場合には成功失敗を表現する bool 型ではなく TSTError 型を返す

3.5 実行時エラーの補足

例外を使用しないのでエラーは関数の戻り値として搬出する. errno のようなエラーを保持する大域変数は呼び出しシーケンスによって解釈が変更される可能性があるために抽象的なプログラミングには適さない.

【コード1】の例に示すようにエラーが発生した場合, 続く正常なシーケンスの処理を中止しエラーを呼び出し元に返却する. エラーを回復できる可能性がある場合には回復処理を実行する. この手法を再帰的に適用する.

コード 1 再帰的なエラーの補足

```
void setup(void)
{
    if (parent_func() == TSTERROR_OK) {
        if (another_func() == TSTERROR_OK) {
            // normal task
        }
        else {
            // error recovery task
        }
    }
    else {
        // error recovery task
    }
    return;
}

TSTError parent_func(void)
{
    TSTError error(TSTERROR_OK);

    if ((error = child_func()) == TSTERROR_OK) {
        // normal task
    }
    else {
        // error recovery task
    }
    return error;
}

TSTError child_func(void)
{
    TSTError error(TSTERROR_OK);

    return error;
}
```

3.6 エラー発生時の引数の内容の保持

関数の内部でエラーが生じた場合でも, 引数として渡されたインスタンスの中身は呼び出し前と同じであることが保証される. このような規約を設定することにより【コード2】に示すように同じ引数を代替関数に渡して安全に処理を続行したりエラーから容易に復帰することが可能となる.

コード 2 引数の内容の保持

```
int value = 123;

if (func(&value) == TSTERROR_OK) {
    // value will be changed successfully
}
else {
    // value = 123
}
```

3.7 ハードウェアの停止

Morikawa はシングルタスクで動作しておりオペレーティングシステムなどを利用せずメモリ保護機能も搭載していない. そのため, 不正なメモリアクセスや 0 除算などを行った場合の動作は不定である.

また, Arduino 用に開発されたライブラリの実装詳細には極力関与しない設計方針であり Watch dog timer

[21] インスタンスのコピーが行われる値渡しよりも効率的

[22] 関数呼び出し時に引数に & 演算子を記述することで内容が変更されることが明示的になる

をクリアするコードを既存のライブラリ内に挿入することができないので, Morikawa では Watch dog timer を利用しない. これらの動作は Arduino の実行環境に依存しており Morikawa では明示的な設定を行っていない.

メインモジュールは Morikawa を起動すると設定された時間の後に強制的に電源を停止する. これにより Morikawa の不意な停止時にも電力の消費を抑え Morikawa を再起動することが可能となる. Morikawa の電源が切断されると CPU 内部の各種レジスタや揮発性メモリの状態は保持されず破棄される. 不意な停止時にレジスタの状態を外部からコアダンプのような形式で取得することはできない.

3.8 変数の 3 冗長化

3.8.1 Morikawa における冗長化

Morikawa は導入コストの低い Arduino 互換の環境を目指しているためハードウェアの 3 冗長化を行わず, ソフトウェアでの冗長化に頼っている. 宇宙空間では放射線の影響で CPU やメモリの素子が破壊されたりデータが書き変わるというシングルイベントが発生するため実行時に予期しない動作を引き起こす可能性がある. ハードウェアを 3 冗長化し実行結果を評価することによって予期しないシングルイベントの影響を回避することができるがハードウェアが複雑化する.

3.8.2 ソフトウェアの冗長化

ハードウェアを冗長化せずにソフトウェアを冗長化する場合, プログラム本体のデータと実行時メモリ内のデータ[23]を冗長化する必要がある. プログラム本体のデータを冗長化して同時に実行することは ATmega 2560 では不可能であり, それは複数の ATmega 2560 を使用して実行することと同じである. プログラムメモリ内に同じプログラムを複数個複製して書き込んでおき, 動作開始時に 1 つを選択して実行することは可能であるが 1 つを選択する動作を行うプログラム自身は冗長化できない.

実行時メモリの冗長化についてもコンパイラが自動的に割り当てるレジスタやスタックの内容については冗長化できないため冗長化できるのは開発者が明示的に記述できる箇所のみである.

Morikawa ではプログラムメモリの冗長化は行わず, 実行時メモリ内の重要な変数のみを冗長化し少しでもシングルイベントの影響を軽減することを目指す. 具体的にはメインモジュールとの通信処理を管理する変数やペリフェラル機器の状態を判定する変数などである.

3.8.3 C++ 言語を用いた冗長化クラス

MorikawaSDK では C++ 言語のテンプレート機能を利用して変数の 3 冗長化を行う. TSTTrinity クラスを利用するとプリミティブ変数を【コード3】と【コード4】のような書き換えのみで冗長化することが可能となる. TSTTrinity クラスはポインタ変数にも適応することができる.

コード 3 冗長化されていない変数

```
int i;

i = 123;
if (i == 123) {
    // yes
}
```

コード 4 冗長化された変数

```
TSTTrinity<int> i;

i = 123;
if (i == 123) {
    // yes
}
```

TSTTrinity クラスは内部に 3 つの変数を保持し, 代入・比較や演算などで常に 3 つの変数に対して演算を行う. 2 つの変数が同一で 1 つの変数が異なる場合, 即時に 3 つの変数が同一になるように多数決によって修復される. 3 つの変数が同時に異なる場合にはランダムに 1 つが選択され 3 つの変数を一致させる.

TSTTrinity クラスの実装では 3 つの変数が配列として確保されておりコンパイラはこれらの変数をメモリ上に順番に確保するため隣り合うメモリセルにも破壊が及ぶようなシングルイベントの場合には対応できない.

3.9 記憶素子の抽象化

3.9.1 記憶素子による特性の違い

Morikawa はデータ格納用の記憶素子として不揮発性メモリである FRAM, Flash ROM, Shared EEPROM, ATmega EEPROM を内蔵している. 【表4】に示すようにこれらの記憶素子はそれぞれの特性に応じた読み書きシーケンスを持っている.

例えば FRAM (FM25V10) ではページの概念がなくランダムにアドレスを指定して任意のバイトを書き換えるこ

表 4 記憶素子の特性

	FRAM	Flash ROM	Shared EEPROM	ATmega EEPROM
型番	FM25V10	M25P80	24LC1025	ATmega 2560
バイト書き込み	可能	ページ毎に 1 回	非効率だが可能	可能
バイト読み込み	可能	可能	可能	可能
ページの概念	なし	あり	あり	なし
ページ書き込み	不可	可能	可能	不可
セクタの概念	なし	あり	なし	なし
書き込み前の消去	なし	必要	なし	なし

[23] ATmega 2560 はハーバードアーキテクチャなのでプログラムメモリとワーキングメモリは分かれている

とができる[5]. Flash ROM (M25P80) にはページの概念がありページ毎に 1 度だけ書き込むことができる. 再度書き換えるにはセクタ単位での消去を実行してから書き込む必要がある[6].

Shared EEPROM (24LC1025) では消去せずに上書き可能であるがページの概念があり, ページ単位で書き込むと高速に書き込むことができ, 同時に素子の寿命を延ばすことができる[7]. ATmega EEPROM は ATmega 2560 に内蔵された EEPROM であり AVR のアセンブリ言語や C 言語からは任意のバイト数で読み書きすることができる[8].

このように記憶素子によってランダムアクセスの可否, ページの概念の有無, 消去の必要性などに違いがある.

3.9.2 インターフェースの抽象化の必要性

アプリケーション開発者は記憶素子の種類を区別することなく「データを保存したい」「データを読み出したい」という目的を重視する. そこで記憶素子の種類に関わらず同じ使い方でデータの読み書きができると学習コストを軽減しソフトウェアの再利用性が向上する.

読み書きの方法を統一化することは記憶素子を抽象化していることと同等であり C++ 言語を用いたクラスの継承やテンプレートクラスとも相性が良く Virtual Machine の実装においても素子による条件分けを必要とせず効率的なマシン語の定義を可能にする.

3.9.3 抽象化されたインターフェース

そこで MorikawaSDK では【表5】の設計仕様に基づき, TSTMorikawa クラスが次のような仕様の読み書きインターフェースを提供する（XXX には FRAM, FlashROM, SharedMemory, EEPROM が該当する）.

表 5 記憶素子を抽象化した設計仕様

操作内容	設計仕様
バイト書き込み	受け付けるが書き込めないときはエラーを返す
バイト読み込み	受け付けるが読み込めないときはエラーを返す
ページの概念	ページサイズを取得できるようにする, 存在しないときは 0
ページ書き込み	バイト書き込みと共通とし内部で自動判別する
セクタの概念	セクタサイズを取得できるようにする, 存在しないときは 0
書き込み前の消去	内部で自動的に処理する

- unsigned long getSizeXXX(void);
  解説：全体サイズを取得する関数
  引数：なし
  戻値：全体のサイズ, 素子が存在しない場合は 0

- unsigned int getPageSizeXXX(void);
  解説：ページサイズを取得する関数
  引数：なし
  戻値：ページのサイズ, 概念が存在しない場合は 0

- unsigned long getSectorSizeXXX(void);
  解説：セクタサイズを取得する関数
  引数：なし
  戻値：セクタのサイズ, 概念が存在しない場合は 0

- bool isValidXXX(void) const;
  解説：利用可能かどうかを判定する関数
  引数：なし
  戻値：利用可能な時は true, そうでない場合は false

- TSTError writeXXX(unsigned long address, void const* data, unsigned int size, unsigned int* result = NULL);
  解説：RAM 領域上の任意のバイト列のデータを書き込む関数
  引数：address 書き込みを開始する記憶素子上のアドレス
  data 書き込む RAM 領域のデータへのポインタ
  size 書き込むデータのサイズ
  result 実際に書き込んだサイズ
  戻値：バイト数が適合しない時や消去が必要な場合などにはエラー

- TSTError writeXXXPGM(unsigned long address, void const PROGMEM* data, unsigned int size, unsigned int* result = NULL);
  解説：ROM 領域上の任意のバイト列のデータを書き込む関数
  引数：address 書き込みを開始する記憶素子上のアドレス
  data 書き込む ROM 領域のデータへのポインタ
  size 書き込むデータのサイズ
  result 実際に書き込んだサイズ
  戻値：バイト数が適合しない時や消去が必要な場合などにはエラー

- TSTError readXXX(unsigned long address, void* data, unsigned int size, unsigned int* result = NULL);
  解説：任意のバイト列のデータを読み込む関数
  引数：address 読み込みを開始する記憶素子上のアドレス
  data データを読み込む RAM 領域へのポインタ
  size データを読み込む RAM 領域のサイズ
  result 実際に読み込んだサイズ
  戻値：バイト数が適合しない時や問題が発生した場合などにはエラー

- TSTError formatXXX(void);
  解説：全体を 0xFF で初期化する関数
  引数：なし
  戻値：問題が発生した場合などにはエラー

これらの関数を使用した記憶素子へのデータの読み書きのサンプルコードを【コード5】と【コード6】に示す.

コード 5 FRAM への読み書き

```
// 読み書きに使うバッファ
char data[600] = “write to FRAM!";

// FRAM が利用可能かどうか判定
if (Morikawa.isValidFRAM()) {

    // FRAM の全領域を 0xFF で初期化
    Morikawa.formatFRAM();

    // FRAM の 100 バイト目から 600 バイトを書き込み
    Morikawa.writeFRAM(100, data, sizeof(data));

    // FRAM の 80 バイト目から 600 バイトを読み込み
    Morikawa.readFRAM(80, data, sizeof(data));
}
```

コード 6 Flash ROM への読み書き

```
// 読み書きに使うバッファ
char data[600] = “write to Flash ROM!";
```

```
// Flash ROM が利用可能かどうか判定
if (Morikawa.isValidFlashROM()) {

    // Flash ROM の全領域を 0xFF で初期化
    Morikawa.formatFlashROM();

    // Flash ROM の 100 バイト目から 600 バイトを書き込み
    Morikawa.writeFlashROM(100, data, sizeof(data));

    // Flash ROM の 80 バイト目から 600 バイトを読み込み
    Morikawa.readFlashROM(80, data, sizeof(data));
}
```

3.9.4 FRAM の実装

FRAM (FM25V10) は任意のアドレスのデータにランダムにアクセスすることができ，ページやセクタの概念が存在せずバイト単位で読み書きを行うことができる【図5】.

SPI での接続となりデータ列に先立って読み書きの開始アドレスを送信する必要がある．アドレスの送信は 1 度限りの方が効率がよいため始めにアドレスを送信し次に任意のバイト数を送受信する.

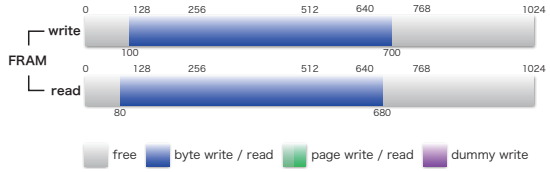


図 5 FRAM の読み書き手順

3.9.5 Flash ROM の実装

Flash ROM (M25P80) はページ毎に 1 度だけ書き込むことができ，再度書き込むにはセクタ単位での消去を必要とする．読み出しはランダムなアドレスから任意のバイト数で可能である【図6】.

SPI での接続となりデータ列に先立って読み書きの開始アドレスを送信する必要がある.

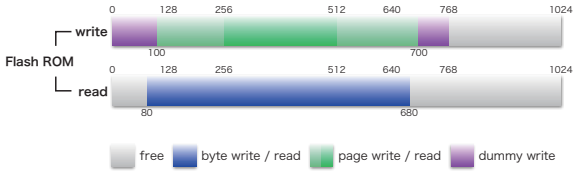


図 6 Flash ROM の読み書き手順

ページの概念が存在しページ毎に 1 度だけ書き込むことができるためデータの書き込みをページ単位で行う必要がある．任意の開始アドレスから書き込む場合には次のページ境界までと最後のページ境界から書き込みの最終アドレスまでを特別に処理し，それぞれのページで不足しているバイトを 0x00 のダミーデータで補完する．中間部分のデータについてはページ単位で書き込む.

一度書き込んだページには再度書き込むことができないため，書き込みたいバイト数に該当する連続したページ領域が消去された状態でなければいけない．連続したページを確保できない場合はエラーを返すなど処理を中止する．MorikawaSDK では Flash ROM のページ毎の状態を管理するために ATmega EEPROM の一部をページマップとして利用している．ページマップはページ毎に 2 ビットを使用し 1 バイトで 4 ページを管理する．1 つ目のビットはページの正常性を表し，2 つ目のビットはページが書き込み済みかを表す.

セクタ単位での消去時に始めにセクタを 0x00 で上書きし全ビットが 0 に変更されたことを検証し，次にセクタ消去を行い全ビットが 1 に変更されたことを検証することによってセクタ内のページの正常性が検証される．検証に失敗したページは破壊されていると認識され利用できなくなる.

3.9.6 Shared EEPROM の実装

Shared EEPROM (24LC1025) にはページの概念が存在しページ毎に処理を行うことで高速に書き込むことができ同時に素子の寿命を延ばすことができる．また，ページを利用せずに任意のアドレスからバイト単位で書き込むことも可能である．読み出しはランダムなアドレスから任意のバイト数で可能である【図7】.

I2C での接続となり，バイト単位での書き込みではアドレスを送信し次に 1 バイトのデータを送信する．ページ単位での書き込みではアドレスを送信し次に 128 バイトを送信する．読み込みではアドレスを送信し任意のバイト数を受信する.

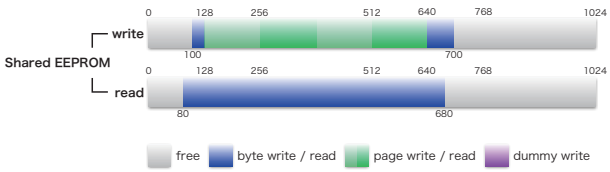


図 7 Shared EEPROM の読み書き手順

MorikawaSDK では任意のアドレスから書き込む場合，ページ境界に属さないデータはバイト単位で書き込まれる．該当するページのデータを作業用メモリに一時的に読み出し，更新するバイトを新しいデータで置き換えてページ書き込みをすると素子の寿命を延ばせるが 1 ページ分の作業用メモリが必要となる．ATmega 2560 は実行時メモリの資源が少ないために素子の寿命よりも実行時メモリの節約を優先しバイト書き込みを採用した.

3.9.7 ATmega EEPROM の実装

ATmega EEPROM は ATmega 2560 に内蔵された EEPROM であり AVR の命令セットや C 言語ライブラリ[24]から読み書きすることができる．ATmega EEPROM

[24] avr/eeprom.h ファイルに含まれる関数群

ではページの概念を意識せずランダムなアドレスからバイト単位で読み書きを行うことが可能である【図8】.

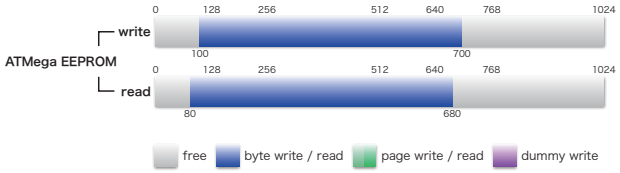


図 8 ATmega EEPROM の読み書き手順

ATmage 2560 に内蔵された EEPROM であるため読み書きにおいてエラーが発生する可能性は低く，外部に接続されたペリフェラル機器の状態の管理や実行時状態の保存などに利用することができる.

3.9.8 その他の種類のメモリ

Morikawa では全て不揮発性メモリを採用しているがアクセス速度や素子の耐久性の観点から SRAM や DRAM などの揮発性メモリを採用することも考えられる. そのような場合においても Morikawa で定義しているインターフェースを踏襲することができ FRAM での実装に近いものになると考えられる.

3.10 処理の中断

Morikawa はメインモジュールに従属して管理されているため予期しないタイミングで電源を切断される可能性がある. データ格納用のメモリにデータを書き込んでいる状態で電源を切断されると予期しない不具合を引き起こす可能性があるため，電源の切断前にメインモジュールから Morikawa に強制終了のメッセージが通知される.

MorikawaSDK は受信割り込み処理内で強制終了の通知を受信すると強制終了フラグを設定し，ペリフェラル機器への読み書きなど時間のかかる処理を行う関数は強制終了フラグが設定されているかどうかを随時確認する. 強制終了フラグが設定されている場合には安全な時点で速やかに処理を終了し強制終了エラーを返却する.

3.11 地上へのエラーの伝達

実行中に発生したエラーを地上に伝達する手段が用意されていると実行時の不具合の原因特定の手助けとなる. MorikawaSDK ではすべてのエラーを関数の戻り値として返しエラーを確実に補足する記述方法を採用しており，アプリケーション開発者も返されたエラーを適切に処理することによりプログラムの内部で発生したエラーを外部に伝達することが可能である.

補足されたエラーは Shared EEPROM のデバッグ用テキストエリア[25]に書き込むことができ，メインモジュールからデバッグ用テキストエリアを参照することが可能である. ただし Shared EEPROM の操作においてもエラーを発生させる可能性があるため，エラーの内容をモールス信号で地上に送信する[26]など代替手段の検討も必要である. MorikawaSDK は地上へのエラー送信の具体的な実装には関与しないのでアプリケーション開発者の設計に任されている.

4 リプログラミングの実現

4.1 軌道上でのリプログラミング

地球を周回している衛星のソフトウェアを書き換えるにはいくつかの手法があるが，Arduino 互換である Morikawa では Arduino のブートローダを利用したソフトウェアの書き換えが検討された.

この方法は標準的な Arduino と Arduino の開発環境を用いて地上でプログラムを実行する場合と同等である. 予め Arduino のブートローダを書き込んだ ATmega 2560 を Morikawa に搭載しておき，地上から新しいソフトウェアを送信し，メインモジュールを介してリプログラミングを行う. 標準的な Arduino のリプログラミングの手法をそのまま転用することが可能となるが衛星に送信するソフトウェアのサイズが大きくなりアマチュア無線帯の転送帯域を利用する場合には現実的ではない.

そこで INVADER 衛星では MorikawaSDK の抽象化された API の利点を生かし独自の言語を用いたスクリプティングが考えられた. MorikawaSDK では Morikawa に搭載されているすべての機能を操作できるように最小の組み合わせの API 関数が用意されている. これらの関数をどのように呼び出すかという手順のみをリプログラミングすることで，個別の機器の具体的な操作手順を再送信することなくリプログラミングと同じ効果を得ることが可能となる.

4.2 InvaderVM の概要

InvaderVM は InvaderVM マシン語を実行する Morikawa アプリケーションの 1 つである【付録A】. InvaderVM は【表6】に示す 16 個のレジスタ，128 バイトのヒープ，128 バイトのプログラムメモリを持っている.

表 6 InvaderVM のレジスタ

レジスタ名	役割
ERRN	API 関数のエラー
FUNC	呼び出す API 関数
RETV	API 関数の戻値
HCUR	ヒープの現在位置
HEAP	HCUR が指し示すヒープの内容
ARG0 ~ ARGA	汎用レジスタ 0 ~ A

[25] Morikawa.setText(TEXT_DEBUG, “something");

[26] Morikawa.playMorse(NOTE_C6, “something");

表 7 InvaderVM のアセンブリ言語

ニーモニック	オペランド	動作
END	なし	プログラムの終端
NOP	なし	何もしない
SET	char, long	long 値を設定
SETC	char, char	char 値を設定
SETI	char, int	int 値を設定
CLR	char	0 クリア
MOV	char, char	移動コピー
XCHG	char, char	入れ替え
CALL	なし	関数呼び出し
JMP	char	ジャンプ
JMPIF	char, char	条件ジャンプ
JMPNOT	char, char	否定条件ジャンプ
INC	char	1 加算
DEC	char	1 減算
NEG	char	符号入れ替え
ADD	char, char	加算
SUB	char, char	減算
MUL	char, char	乗算
DIV	char, char, char	除算
AND	char, char	論理積
OR	char, char	論理和
XOR	char, char	排他的論理和
NOT	char	否定
SHR	char, char	右シフト
SHL	char, char	左シフト
EQ	char, char, char	合同
LT	char, char, char	より小さい（未満）
LE	char, char, char	以下
GT	char, char, char	より大きい
GE	char, char, char	以上
WAIT	char	ミリ秒停止
TXT	long, char...	文字列読み出し
EXC	long	ヒープをプログラムとして実行
COMPRESSED	なし	圧縮されたプログラム

InvaderVM マシン語は少ないプログラムデータ量で MorikawaSDK に含まれる API 関数を効率的に呼び出すことができる設計となっており，【表7】に示すように，レジスタへの代入・数値演算・数値比較・条件分岐・関数呼び出し・文字列の読み出し・ヒープの利用に対応している．

InvaderVM マシン語を生成するには InvaderVM ニーモニックを用いてプログラムを記述し，InvaderVM アセンブラ[27]を使用してコンパイルする．

4.3 InvaderVM の実装

InvaderVM は独自のプログラムカウンタを持ち 128 バイトのプログラムメモリ内のマシン語を逐次実行する．各レジスタは 4 バイトあり 32 ビット値をそのまま格納することができる．

スタックの概念は存在せず API 関数の引数は ARG0 ~ ARGA のレジスタを介して渡される．API 関数の TSTError 型の戻値は ERRN レジスタに格納され，API 関数の結果を表す戻値や引数は RETV レジスタに格納される．API 関数を呼び出すには FUNC レジスタに呼び出したい API 関数のインデックスを設定し CALL 命令を実行する．

128 バイトのヒープメモリは HCUR レジスタと HEAP レジスタを介してアクセスする．HCUR レジスタはヒープ上の現在位置を表し，HEAP レジスタは現在位置の内容を表す．InvaderVM はヒープ上のバイト列をマシン語として実行する機能を持っており EXC 命令を実行することで動的に生成したマシン語を実行する機能も持っている．また，MorikawaSDK の FastLZ 圧縮 / 解凍エンジン[28]を利用して FastLZ により圧縮されたマシン語を実行することも可能である．

5 INVADER 衛星での実証と成果

5.1 MorikawaSDK を利用したアプリケーション

INVADER 衛星には MorikawaSDK を利用して実装された【表8】に示す 10 種類のアプリケーションと 5 種類のメンテナンス用ユーティリティーが搭載された．

これらのアプリケーションは Shared EEPROM の起動モードを指定するパラメータ[29]を利用して Morikawa の起動時に選択して実行される．

【コード7】と【コード8】に MorikawaSDK を利用して作成されたアプリケーションのソースコードを示す．

コード 7 HelloSpace

```
static char const hellospace_morse[] PROGMEM
    = "Hello, space!";
static char const hellospace_speak[] PROGMEM
    = "konnnichi'wa uchu-";

void HelloSpace_setup(void)
{
    __debug__(Morikawa.setTextPGM(
        TEXT_Y,
        hellospace_morse
    ), 1);
    return;
}

void HelloSpace_loop(void)
{
```

[27] http://github.com/ARTSAT/INVADER/tree/master/ground_station/mission/software/iva

[28] Morikawa.freezeFastLZ() 関数と Morikawa.meltFastLZ() 関数

[29] Morikawa.getBootMode() 関数で取得できる

```
    __debug__(Morikawa.playMorsePGM(
        NOTE_C6,
        hellospace_morse
    ), 101);
    delay10mTimes(300);

    __debug__(Morikawa.speakPhrasePGM(
        hellospace_speak
    ), 102);
    delay10mTimes(300);
    return;
}
```

コード 8　PlayMelody

```
static NoteParam playmelody_param;

void PlayMelody_setup(void)
{
    TSTError error;

    error = __debug__(Morikawa.getParamNote(
        &playmelody_param
    ), 1);
    if (error != TSTERROR_OK) {
        Morikawa.shutdown();
    }
    return;
}

void PlayMelody_loop(void)
{
    __debug__(Morikawa.playNote(
        reinterpret_cast<NoteSequence const*>(
            playmelody_param.data
        ),
        playmelody_param.size / sizeof(NoteSequence)
    ), 101);
    delay10mTimes(300);
    return;
}
```

5.2 Morikawa の稼働状況

日本時間 2014/02/28 の INVADER 衛星の打ち上げ後, 2014/03/08 の運用で Morikawa が初めて起動され HelloSpace アプリケーションが実行された. その後随時動作確認を行い, 搭載されているすべてのアプリケーションの起動確認を行った.

HelloSpace によるモールス符号の受信と日本語メッセージの受信, SpeechText による地上から設定したメッセージの発音, PlayMelody による地上から設定した楽譜の演奏, InvaderBot による INVADER 衛星との会話などが通常の運用である. SpeechText を利用し次のような宇宙詩の朗読実験も行った.

```
fuuuuuuuuuu;mu,;poppo
tsukikibowo,o;o;ooooochi_suchi_kuchi_suchi_kuchi;
    pogosama;pogosama+poppo+poggo,ochitamuuuuun?
ku;biee?
```

日本時間 2014/05/23 03:34 にはミッションモジュールを使った Morikawa のタイマー起動によりヨーロッパ上空での HelloSpace の実行に成功し, ヨーロッパ各地のアマチュア無線家から受信報告を得ることができた.

表 8　アプリケーションとユーティリティー

名称	ID	内容
SelfTest	0	起動後すぐに正常終了する
HelloSpace	1	Shared EEPROM の TEXT_Y パラメータに “Hello, space!” を設定し, モールス符号で FM 送信した後 「こんにちは, 宇宙」を音声合成し FM 送信する
SpeechText	2	Shared EEPROM の TEXT_X パラメータに 設定された内容を音声合成して FM 送信する
CodeText	3	Shared EEPROM の TEXT_X パラメータに 設定された内容をモールス符号で FM 送信する
PlayMelody	4	Shared EEPROM のノートパラメータに 設定された楽譜を演奏して FM 送信する
InvaderBot	5	Shared EEPROM の TEXT_X パラメータに 設定された問いかけに Space ELIZA が応答し TEXT_Y パラメータに返答を設定する
InvaderMusic	6	全センサのテレメトリデータを数式に従って 変換した音階を演奏し FM 送信する
InvaderCam	7	Morikawa に搭載された CMOS カメラを使って 写真を撮影し Shared EEPROM に書き込む
InvaderVM	8	Shared EEPROM の TEXT_Z パラメータに 設定されたマシン語を実行する
TelemetryDump	9	全センサのテレメトリデータを 順次 Shared EEPROM に書き込む
EraseSelfTestLog	128	自己診断テストログを初期化する
FormatEEPROM	129	ATmega EEPROM を 0xFF で初期化する
FormatSharedMemory	130	Shared EEPROM を 0xFF で初期化する
FormatFRAM	131	FRAM を 0xFF で初期化する
FormatFlashROM	132	Flash ROM を 0xFF で初期化する

5.3 InvaderVM を用いたリプログラミング

InvaderVM の動作検証として「こんにちは, 宇宙」を InvaderVM 上で動作させるプログラム【コード9】(【コード10】) を送信し実行することに成功した.

コード 9　InvaderVM のテストコード

```
CLR,   HCUR              # load string into the heap
TXT,   11,   <NUM VAL=3>
SETC,  FUNC, speakPhrase # initialize registers
SETC,  ARG0, 11          # speakPhrase() argument
SETC,  ARG1, 3           # counter
SETI,  ARG2, 1000        # 1000 millisec
begin:                   # while (ARG1 > 0)
CLR,   HCUR
CALL
WAIT,  ARG2
SETC,  HCUR, 9
DEC,   ARG1
DEC,   HEAP
JMPIF, ARG1, begin:
CLR,   HCUR              # end while
TXT,   18,   konnnichi'wa uchu-
SETC,  ARG0, 18
CALL
END
```

コード 10 コンパイルされたマシン語

```
05 03 1f 0b 00 00 00 3c 4e 55 4d 20 56 41 4c 3d
33 3e 03 01 4a 03 05 0b 03 06 03 04 07 e8 03 05
03 08 1e 07 03 03 09 0d 06 0d 04 0a 06 f1 05 03
1f 12 00 00 00 6b 6f 6e 6e 6e 69 63 68 69 27 77
61 20 75 63 68 75 2d 03 05 12 08 00
```

また, 音声合成 LSI の数値読み上げ機能を利用してテレメトリデータを声で地上に伝える InvaderVM プログラムの動作実験にも成功した.

5.4 MorikawaSDK の既知の不具合

INVADER 衛星に搭載された MorikawaSDK には Shared EEPROM 上の特定のバイト列を読み込むと無限ループに陥る不具合が発見された. 不具合の原因は MorikawaSDK が採用している I2C ライブラリの利用方法であった.

このライブラリは標準 Arduino 付属の Wire ライブラリ[30]をもとにミッションモジュールと効率的に通信できるように改変されたものであるが, 0 バイトの受信を行うと無限ループに陥る欠陥がある. Wire ライブラリではこのような利用は想定されていないため unsigned 型変数により演算を行っている箇所でアンダーフローが発生し, 次にバッファオーバーランを引き起こし I2C 通信を制御している重要なフラグを上書きする.

コード 11 修正前

```
// I2Cm.cpp (twi.c)
static volatile uint8_t twi_masterBufferIndex;
static volatile uint8_t twi_masterBufferLength;
```

コード 12 修正後

```
// I2Cm.cpp (twi.c)
static volatile int16_t twi_masterBufferIndex;
static volatile int16_t twi_masterBufferLength;
```

【コード11】と【コード12】に示すようにこの不具合は最新版の MorikawaSDK では修正済みであるが, INVADER 衛星では不具合を避ける対処療法により運用が行われている.

5.5 Morikawa で撮影された地球

日本時間 2014/04/08 16:28:18 に InvaderCam の動作実験が行われた. 初回の起動であったため 160 * 120 サイズ RGB565 フォーマットの写真が撮影された.

始めに 60 行目を受信したところ地球らしき緑色から青色のピクセルが確認されたので引き続きデータを受信したところ地球と INVADER 衛星のアンテナが写っていることが判明した. そこでアマチュア無線家の協力を得てすべてのデータを 1 ヶ月あまりかけて受信した.

[30] <Arduino>/libraries/Wire/utility/twi.c ファイル

撮影された時点の INVADER 衛星の位置を【図9】に撮影された写真を【図10】に示す.

図 9 撮影時の位置[9)]

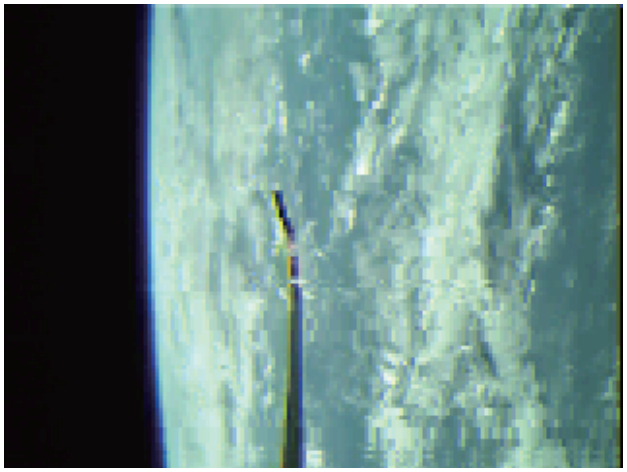


図 10 撮影された写真

6 謝辞

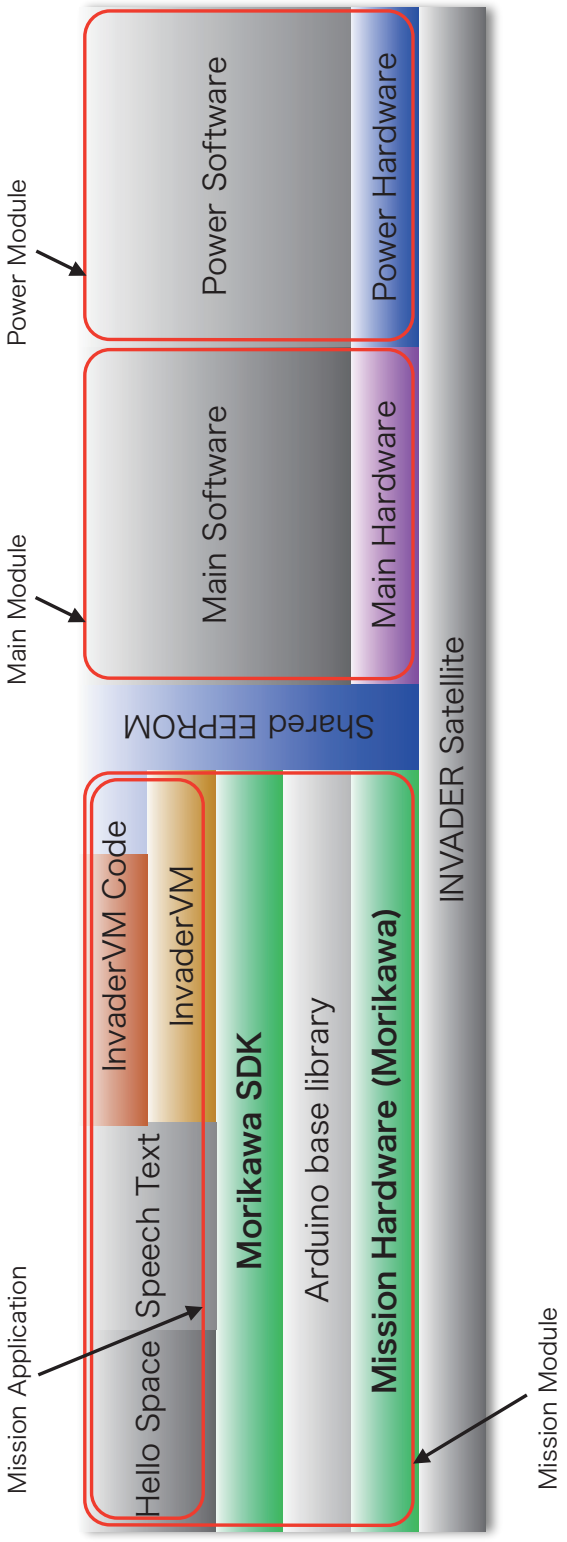
ARTSAT プロジェクトは, 2014 年度多摩美術大学共同研究費「超小型芸術衛星 INVADER の打ち上げと ARTSAT プロジェクトの展開」および 2014 年度科研費基盤研究 (C)「衛星芸術用ミッションモジュールの開発と遠隔創造の実践」の支援を受けて進められた.

田中利樹 INVADER 開発 PM を始めとする ARTSAT プロジェクトメンバーと関係各位の尽力に厚く御礼を申し上げる.

参考文献

1) OmniVision Technologies, Inc., OmniVision Serial Camera Control Bus (SCCB) Functional Specification, OmniVision Technologies, Inc., Document Version 2.2 (2007), pp.8-15
2) Bjarne Stroustrup（著）$\varepsilon\pi\iota\sigma\tau\eta\mu\eta$（監）岩谷宏（訳）, C++の設計と進化, ソフトバンククリエイティブ, 2005, pp. 31-154
3) Richard M. Stallman and the GCC Developer Community, Using the GNU Compiler Collection, GNU Press, For gcc version 4.3.6, p. 33
4) Richard M. Stallman and the GCC Developer Community, Using the GNU Compiler Collection, GNU Press, For gcc version 4.3.6, p. 222
5) Cypress Semiconductor Corporation, FM25V10 1Mb Serial 3V F-RAM Memory, Cypress Semiconductor Corporation, 001-84499 Rev. *B (2013), pp.1-3
6) Micron Technology, Inc., Micron M25P80 Serial Flash Embedded Memory, Micron Technology, Inc., Rev. G (2013), pp.6-15
7) Microchip Technology Inc., 24AA1025/24LC1025/24FC1025, Microchip Technology Inc., Revision E (2007), pp.8-12
8) Atmel Corporation, ATmega640/V-1280/V-1281/V-2560/V-2561/V [DATASHEET], Atmel Corporation, 2549Q–AVR–02/2014 (2014), pp.22-25
9) Sebastian Stoff, Orbitron - Satellite Tracking System, http://www.stoff.pl (2014)

付録 A



付録 B