修士論文

センサ IoT デバイスにおけるエミュレーション抽象化

1710171　広瀬太志

主指導教員　篠田 陽一 教授
審査委員主査　篠田 陽一 教授
審査委員　知念 賢一 特任准教授
丹 康雄 教授
Razvan Beuran 特任准教授

北陸先端科学技術大学院大学
先端科学技術研究科
（情報科学）

平成 31 年 2 月

概 要

IoT (Internet of Things) の登場により、広範囲の環境情報計測と収集が容易になる。このような計測のためのセンサを搭載する IoT デバイスは、市街地や圃場などでの使用が想定される。しかし、デバイスが広範囲に分散配置されると、設置後の再配置をともなう改修は困難である。そのため、設置前の検証が重要となるが、必要数のデバイスを用意して現地で検証作業を行うことは非常に非効率的である。したがって、コンピュータを使用したエミュレーションによる IoT デバイスシステムの検証が求められる。センサ IoT デバイスのエミュレーションでは、一度に大量のデバイスエミュレータを実行しなくてはならず、多くの計算リソースが必要となる。

そこで本研究では、検証するアプリケーションやセンサなどの種類に応じて、エミュレータの機能の一部を抽象化することを提案する。エミュレータ抽象化は、一般にエミュレーションの計算要求リソースの削減を可能にする。一方で、実機と比較して忠実度の低下を招くため、抽象化には、一種のトレードオフの関係がある。忠実度の変化が生じた場合、抽象化したエミュレーションではテスト要件を満たせなくなる可能性がある。また、適切な抽象度のエミュレータを実装しなければ、必要以上に計算リソースを使うエミュレータを開発することになる。したがって、本研究では、エミュレータを抽象化する際の忠実度とテスト要件のトレードオフ関係を明らかにし、エミュレータ抽象化の有効性の評価を行った。

エミュレータ抽象化間のトレードオフ関係を明らかにするため、エミュレーション実行が、4 つの層で大きく抽象化度が変わることに注目し、4 層の抽象化モデルを定義した。定義した抽象化モデルをもとに、エミュレータ抽象化の有効性の評価を行った。検証環境として、組込みデバイスで利用が想定される、ARM アーキテクチャのエミュレータを用いた。実験対象のセンサ IoT デバイスは、プロセッサとセンサを有し、これらの間を SPI (Serial Peripheral Interface) を用いて制御する。このインタフェースをエミュレータに実装しなければセンサ IoT デバイス用ソフトウェアの検証はできない。一方で、この様なインタフェースは必要であるが、プロトコルなどを忠実に実装する必要は無い。本研究では SPI 機能の一部を抽象化し、2 種類のエミュレータを設計した。1 つ目は、忠実な SPI 通信処理を行う SPI レジスタモデルである。SPI ハードウェアレジスタを定義し、これを内包するエミュレータを実装した。2 つ目は SPI 通信機能を忠実に再現しない Exodus モデルである。SPI 機能の抽象化のために、SPI ライブラリの書き換えを行い、ライブラリ中の通信に関する関数を呼び出すと、SPI による通信処理をエミュレータ外で代替する実装をした。

実験評価では、エミュレータの要求リソース削減に関して複数の実験を行った。始めに、エミュレータの抽象化による計算要求リソース削減について、エミュレー

タ実行時間の計測によって定量的に確認した。Exodus モデルのエミュレータは、忠実度の高い SPI レジスタモデルと比較して 9.95%実行時間が短く、計算リソース要求が削減された。次に、SPI 通信処理の時間を測定することで、抽象化が SPI 通信処理に与える影響を確認した。SPI 通信の抽象化により、忠実度を下げた Exodus モデルのエミュレータの実行時間が短くなる傾向を示した。最後に、エミュレーション実行時の要求メモリ量を計測した。IoT デバイスエミュレーションでは、同時に大量のデバイスを実行する必要がある。実験で使用したエミュレータ群では、現代のコンピュータにおいてメモリ使用量が十分に小さいため、問題にならないと結論づけた。これらの実験結果から、エミュレータ実行における計算時間が削減され、エミュレーションにおける要求リソースの削減が可能であることを示した。

本研究では、開発プロセスにおける IoT デバイスエミュレーションの影響について整理し、エミュレーションの抽象化間のトレードオフを明らかにした。これらとともにエミュレータの抽象化が IoT デバイスエミュレーションにおける要求計算リソースの削減に改善に貢献することを示した。

目 次

第 1 章　はじめに **1**
1.1　研究の背景 1
1.2　本研究における目的 2
1.2.1　研究における動機 2
1.2.2　研究における想定 2
1.2.3　検証手法の検討 2
1.2.4　既存手法の課題 3
1.2.5　研究における目的 3
1.3　本論文の構成 4

第 2 章　IoT の関連技術と用語 **5**
2.1　IoT のハードウェア 5
2.1.1　省電力なシステム 5
2.1.2　センサとアクチュエータ 5
2.1.3　その他のハードウェア 6
2.1.4　近距離機器間通信 6
2.2　IoT のネットワーク 6
2.2.1　ネットワークモデル 7
2.2.2　IoT 向け省電力ネットワーク 7
2.2.3　キャリアネットワーク 9
2.3　検証に関する技術 10
2.3.1　実機による検証 10
2.3.2　ソフトウェアエミュレータやシミュレータによる検証 10
2.4　検証基盤 11
2.4.1　物理検証環境 11
2.4.2　クラウド環境 13

第 3 章　ソフトウェアテストのための IoT デバイスエミュレーション **14**
3.1　IT システムと IoT システムの共通点と相違点 14
3.1.1　共通点 14
3.1.2　相違点 14
3.2　IoT デバイスエミュレーションの要件 15

3.3 エミュレータを用いる利点 15
3.3.1 規模性 15
3.3.2 バイナリ互換性 15
3.3.3 意図したシナリオの実験 16
3.3.4 実地検証における時間的コスト削減 16
3.4 エミュレータを用いる欠点 16
3.4.1 開発量増加 16
3.4.2 実物と比較した忠実度の低下 17
3.5 忠実度と抽象化間のトレードオフ 17

第 4 章 関連研究 **18**
4.1 IoT エミュレーションに関する研究 18
4.1.1 エミュレーション環境に関する研究 18
4.1.2 検証環境の構築とスケーラブルなエミュレーション環境展開に関する研究 19
4.2 組込みソフトウェアのテストに関する研究 19
4.3 無線通信シミュレーションに関する研究 21
4.4 本研究における取り組み 23

第 5 章 エミュレーションにおける抽象化モデル **24**
5.1 抽象化する対象 24
5.2 概念モデル 24
5.2.1 抽象化レベル 24
5.2.2 抽象化間のトレードオフ 26
5.3 抽象化するエミュレーションモデル 27
5.3.1 SPI レジスタレベルエミュレーション 28
5.3.2 SPI ライブラリレベルエミュレーション 28
5.4 抽象化モデル 29
5.4.1 SPI レジスタモデル 29
5.4.2 Exodus モデル 29
5.5 設計と実装 30
5.5.1 エミュレータ上の SPI インタフェース実装方式 30
5.5.2 Cortex-M0+エミュレータの拡張 30
5.5.3 対向側エミュレータ ADT7310 の実装 35

第 6 章 実験と評価 **37**
6.1 実験環境 37
6.2 実験のシナリオ 38
6.3 凡例 38
6.4 実行時間の測定 38

6.4.1 実験方法 . . . 39
6.4.2 結果 . . . 40
6.5 エミュレータ初期化時間の測定 . . . 41
6.5.1 実験方法 . . . 41
6.5.2 結果 . . . 42
6.6 SPI 通信における I/O 処理にかかる時間測定 . . . 42
6.6.1 実験方法 . . . 44
6.6.2 結果 . . . 45
6.7 使用メモリの計測 . . . 49
6.7.1 実験方法 . . . 49
6.7.2 結果 . . . 49
6.8 全体の評価 . . . 52

第 7 章 議論 **53**
7.1 エミュレーションの要求リソース削減の効果と応用 . . . 53
7.1.1 CPU あたりのエミュレータ展開数のスケーラビリティに関する考察 . . . 53
7.1.2 プログラム中に占める I/O 処理の量についての考察 . . . 53
7.1.3 I/O 処理時間がプログラムの動作に影響する場合の考察 . . . 54
7.1.4 クロックと実時間実行性 . . . 54
7.1.5 エミュレータの抽象化手法の適用性 . . . 55
7.2 忠実度の検討 . . . 55
7.3 IoT デバイスエミュレーションの検証開始までにおける開発規模と開発増加量についての考察 . . . 57
7.3.1 テストベッドによる開発サイクル . . . 57
7.3.2 IoT デバイス開発における開発プロセス . . . 59
7.3.3 エミュレーション開発を含む IoT 開発プロセスにおける各フェーズについて . . . 61

第 8 章 おわりに **66**
8.1 まとめ . . . 66
8.2 今後の課題と展望 . . . 66
8.2.1 本研究における課題 . . . 66
8.2.2 今後の展望 . . . 66

謝辞 **68**

本研究に関する対外発表 **69**

参考文献 **70**

付録 **75**
A.1 実験 6.4 のエミュレータ時間計測に関して 75
B.2 実験に使用したソースコード 77

図 目 次

2.1 IoT のネットワーク接続モデル 7

4.1 Justitia のソフトウェアテストのためのレイヤモデル 20
4.2 AOBAKO のデータフロー 22

5.1 エミュレーションの構成イメージ 25
5.2 抽象化レベル 25
5.3 SPI の通信フロー 28
5.4 ライブラリレベルの抽象化実装例 29
5.5 ハードウェア視点のエミュレータ実装形式例 31
5.6 疑似 SPI I/O モデル 32
5.7 SPI レジスタモデル 33
5.8 Exodus モデル 34
5.9 Exodus モデルの構築フロー 35
5.10 ADT7310 の振る舞いの例 36

6.1 実験のシナリオ 39
6.2 アプリケーションの処理フロー 40
6.3 エミュレータの起動から終了までの実行時間 41
6.4 エミュレータの処理フロー 42
6.5 アプリケーションの初期化処理にかかる時間実行時間 43
6.6 アプリケーションの初期化処理が 10M 命令の実行時間に占める割合 43
6.7 SPI 通信の処理時間の測定対象 44
6.8 SPI 通信に関する処理時間の計測方法 45
6.9 SPI 通信に関する処理時間 46
6.10 SPI 通信に関する処理時間 (合計) 47
6.11 Cortex-M0+エミュレータのメモリ使用量の収束 (32 エミュレーション/60 秒間) 50
6.12 ADT7310 エミュレータのメモリ使用量の収束 (32 エミュレーション/60 秒間) 51
6.13 Emulation flow of basic interpretation and binary translation 52

7.1 テストベッドにおける R&D サイクル 59

7.2 IPA SEC の V モデル 60
7.3 IoT システムの構成物イメージ 62

A.1 エミュレータの起動から終了までの実行時間計測 76
B.2 ADT7310 が送出するデータフォーマット 83

表 目 次

1.1 広域分散配置されるIoTデバイスの特徴 2

5.1 抽象化レベル図における抽象化間の性質 27
5.2 Cortex-M0+の詳細 (抜粋) 31
5.3 SPIレジスタのモデル 32
5.4 exd命令の仕様 34
5.5 エミュレーションモデル 35

6.1 StarBED Group Pのハードウェア諸元 37
6.2 実験のソフトウェア環境 38
6.3 開発環境 38
6.4 グラフラベルと凡例一覧 39
6.5 timeコマンド出力の例 39
6.6 オブジェクトコード中に含まれるSPI通信処理の割合 48
6.7 SPI通信処理がエミュレーション全体に含まれる時間の詳細 48
6.8 psコマンドのメモリ指標の一覧 49
6.9 Cortex-M0+のメモリ使用量 (32エミュレーション/60秒間) 50
6.10 ADT7310のメモリ使用量 (32エミュレーション/60秒間) 51

A.1 エミュレータの起動から終了までの実行時間計測のまとめ 76

第1章　はじめに

1.1　研究の背景

日本が超高齢化社会となる中で、将来に渡る人口の縮小とそれにともなう生産年齢人口の減少は、産業構造に大きな影響を与えることが予想されている。そこで、人口問題から生じる働き手不足を補うために、IT の利用促進によるコネクティビティ、自動化、システム化、ノウハウの蓄積やデータ共有による業務効率化が行われている。代表的な例として、農業分野では、広大な農地を少人数で管理可能にするスマート農業の導入が検討されている。また地方の過疎化が首都への人口一極集中に拍車をかける中で、人口動態などの情報から新たな都市計画の立案などに役立てるため、IT の活用が期待されている。他にもドイツでは、工場の知能化と言われる Industorie4.0 を政府主導で進めている。このような産業構造の変革を世界経済フォーラム (WEF) の定義で第 4 次産業革命と呼ぶ。こうした動向のもとで、新たな価値創造をデータ利活用によって生み出す Society5.0 に向けた政府レベル取り組みがなされている。

技術的な背景として、IT の進展とともに IoT (Internet of Things) の概念が広まっている。IoT とはネットワークにつながるコンピュータのみならず、様々な「モノ」がインターネットにつながる概念である。身の回りの様々なモノや場所にコンピュータを埋め込み、それらを相互に通信・協調させることでデータ収集を可能とし、データの利活用を行う。IoT の進展は、センシングデバイスが高性能化しスマートフォンなどのモバイル多機能端末の普及にともなう大量生産により低価格になったことや、津々浦々で利用可能な無線データ通信網の技術的発展と普及が影響していると考えられる。こうした IoT デバイスは、MNO (Mobile Network Enabler) 等が加入する業界団体の GSMA[1] によると、2022 年までに 50 億台にまで増えると予測されている。

以上の社会的背景と技術的背景により、IoT の普及が望まれている。そのような中で、IoT 機器の利用をより促進するための研究が重要となる。

表 1.1: 広域分散配置される IoT デバイスの特徴

要素	特徴
通信帯域	狭帯域 (数 bps から数百 kbps 程度)
電力要求	低消費電力

1.2 本研究における目的

1.2.1 研究における動機

センサを搭載した計測用 IoT デバイスを使用することによって、広範囲なセンシングが可能になる。その様なセンサの中でも特に気温や照度などの環境情報を測定するものは、広い市街地や圃場などでの利用が想定される。センシングを主とする IoT デバイスは、送信するデータ量が数 Byte から数 KByte とごく少量であることが多く、一定の期間ごとにデータ取得と送信を繰り返す。このような特徴から、センサ IoT デバイスでは長距離通信可能でかつ省電力であることが望ましい。また継続的なデータ収集が必要な計測では、データを収集するサーバ等に正しく届かなければならない。正しくデータが受信されない場合、分析時に一部が欠落した状態となる。無線通信では、データフレームの衝突や無線干渉、減衰など様々な考慮が必要となるため、期待したデータの収集が可能な IoT のシステムを構築するため、事前の検証が重要である。

1.2.2 研究における想定

1.2.1 節で述べた環境センシングでは、測定頻度も一時間に数度程度であることから、デバイスの省電力化のため、ほとんどの時間をスリープ状態にすることで、小型電池による動作を可能にしているものがある。このような IoT デバイスのモデルは小型電池の使用により、電力インフラの構築が不要になるなどの利点があり、圃場や市街地に広域分散配置されるデバイスにとって理想的である。一方で、前述の IoT デバイスは表 1.1 にまとめる特徴を持つため、OTA (On The Air : 無線を利用した) によるアップデートが困難であるという問題を抱えている。またこれらの特徴を持つ IoT デバイスは、広範囲に分散して配置するため、IoT デバイスの設置後の改修が難しく、事前の検証テストが重要である。

1.2.3 検証手法の検討

NICT スマート IoT 推進フォーラム技術戦略検討部会のテストベッド分科会 [12] では、キャラバンテストベッドが提案されている [13]。これは IoT デバイスのラストワンマイルの検証のための移動車式テストベッドの提供により、屋外での IoT シ

ステムの検証を補助するための取り組みである。しかし、この様なテストベッドによる検証であっても、広範分散して配置されるIoTデバイスでは、地理的、距離的に離れた現地に直接赴き、配置して調査する必要があるため、時間的コストがかかる。またハードウェアの調達、ソフトウェアやハードウェアの並行開発にかかるスケジュールなどの問題があるため、現地での検証は最小限にとどめるべきである。

以上のことから、現地で検証テストを行うことは非常に非効率的であるため、コンピュータを使用したIoTデバイスシステムの検証を行うべきである。コンピュータ上で検証する方法として、シミュレーションとエミュレーションを行う方法があるが、より本物のシステムに近い検証が期待できるエミュレーションによる手法について検討する。

1.2.4 既存手法の課題

センシングで使用されるIoTデバイスでは、組込みデバイスを使用したものが多い。組込みシステムで使用するソフトウェアの検証にはCPUエミュレータやマイコン(MCU)エミュレータを用いる。エミュレータを開発する際は、CPUの動作を忠実に実装する必要がある。さらにI/Oインタフェースの実装や外部のデバイスを実装しなくてはならない。この様な組込みシステムのエミュレータは実行コストが高い。また市街地や圃場で使用される環境センシングでは、一度に大量のIoTデバイスをエミュレーションしなくてはならない。そのため検証対象に対して、必要以上に忠実なエミュレータは計算要求リソースが膨大となる。

1.2.5 研究における目的

本研究では計算要求リソースの削減のため、エミュレータを忠実に実装せず、機能の一部を抽象化することを提案する。検証したいセンサやアクチュエータの種類、検証項目によってエミュレータに実装するべき忠実度は異なる。これに加えて、エミュレーションに要求される諸条件によっては、処理時間に対して制限がある場合に悪影響が生じる。したがって部分的に機能を抽象化したエミュレータの実装をするべきである。

エミュレータの抽象化によって、一般にエミュレーションの要求リソースは削減可能であるため、IoTデバイスエミュレーションの様な大量の計算リソースを使用するエミュレータのスケーラビリティ改善を期待できる。この様な利点がある一方で、忠実度を犠牲にするため、抽象化には一種のトレードオフの関係がある。忠実度の変化によって、抽象化したエミュレーションではテストできない、あるいはテスト要件を満たさなくなる可能性がある。したがって、本研究の目的は、エミュレータの機能を抽象化する際のトレードオフの関係を明らかにし、エミュ

レーション抽象化の有効性の評価を行う。また、IoT システム開発における IoT デバイスエミュレーションの導入による影響を調査する。

1.3 本論文の構成

本論文は、本章を含めて 8 章から構成される。2 章の「IoT の関連技術と用語」では、本研究で用いる IoT と検証環境に関する基礎知識を紹介する。3 章「ソフトウェアテストのための IoT デバイスエミュレーション」では、IoT デバイスをエミュレーションによる検証を行う際の要点をまとめる。4 章「関連研究」では、IoT や組込みシステムにおける検証や、ネットワーク検証技術等の関連研究の紹介と、本研究における取り組みを述べる。5 章「エミュレーション抽象化モデル」では、本研究で取り組んだエミュレータの抽象化に用いる抽象化レベルを定義し、エミュレーションモデルの定義と実験のためのエミュレータ設計を述べる。6 章「実験と評価」では、実装したエミュレータについて、種々の方法から抽象化の有効性を検証し、評価する。7 章「議論」では、抽象化と忠実度や、IoT システム開発プロセスにおけるエミュレータによる検証の役割についてなどを議論する。8 章「おわりに」では、本研究のまとめと今後の課題についてまとめる。

第2章　IoTの関連技術と用語

本章では、本研究に関連する技術、概念と用語についてまとめる。

2.1　IoTのハードウェア

IoT (Internet of Things：モノのインターネット) は、従来から存在するM2M (Machine-to-Machine) や組込みネットワークと類似する概念である。IoTの先駆けとなる概念は、カリフォルニア大学バークレイ校のKristofer S. J. Pister教授らがSmart Dustを発表している [55]。ここでは温度センサや照度センサなどのセンサ類と無線通信装置を搭載した超小型のデバイスをmote (モート) と呼んでいる。

IoTデバイスのモノに当たるハードウェアは多岐にわたるが、この節ではIoTに使用されるハードウェアが一般に有する特徴について述べる。

2.1.1　省電力なシステム

IoTはセンサやアクチュエータと通信装置を備えた組込みシステムである。処理の中心はCPUやMCU (Micro Control Unit) プロセッサであり、省電力で非力であるものが多い。省電力のため、小型の電池を使用することが可能であり、電力インフラから開放されたセンシングデバイスを構築可能である。

2.1.2　センサとアクチュエータ

環境情報を取得するセンサを始めとして、モーション検知に使用する音波センサやGPS情報を取得するセンサ、加速度センサ、近距離でデータを送受信するRFIDなどもIoTで使用されるセンサの一種である。

アクチュエータは具体的な動作を実現するデバイスである。エアコンや窓の開閉システムもアクチュエータの一種である。

2.1.3　その他のハードウェア

スマートフォンの様な高性能モバイルデバイスも IoT におけるモノに分類される場合がある。このようなスマートデバイスは高機能であることを利用して IoT デバイスの集約点として振る舞うことがある。

自動車も IoT の一種と考えられる。自動車には自動運転や安全制御のために多数のセンサが積載されており、コネクテッドカーとしてインターネットに繋がる。つまり自動車はセンサとアクチュエータを有し、通信機能を持つ IoT デバイスと言える。

2.1.4　近距離機器間通信

IoT デバイスは制御の中心となるプロセッサ、センサ、アクチュエータや通信機器などの周辺機器で構成されている。それらはプロセッサが持つ I/O インタフェースや汎用ポート (General-Purpose Input/Output：GPIO) を経由して使用する。Harris らの Digital Design and Computer Architecture の「9 章 I/O Systems」 [33] を参考に代表的な近距離機器通信用の通信インタフェースの一部を紹介する。

- **SPI**
  Serial Peripheral Interface の略。送信元となるマスタデバイス (プロセッサ) と受信するスレーブデバイス (センサなど) があり、マスタが 1 つであることに対してスレーブの数に制限はないが、物理結線上の制限がある。マスタからスレーブに送信するデータ線 (Master Out Slave In：MOSI)、スレーブからマスタに送信するデータ線 (Master In Slave Out：MISO)、クロック線、通信対象のスレーブを選択するチップセレクト線の 4 種類の信号線で通信する。双方向同時通信可能であるが、複数のスレーブデバイスと同時に通信することはできない。

- **I2C**
  Inter-Integrated Circuit の略。I-squared-C と発音する。使用する信号線はシリアルデータとシリアルクロックの 2 本からなる。同じ通信バス上で複数のデバイスの通信を行い、デバイスセレクトはアドレスによって行う。アドレスは 7bit のアドレス空間から予約アドレスを除いた 112 あり、理論上それらすべてのデバイスにアクセスが可能である。

2.2　IoT のネットワーク

IoT がインターネットと接続するための形態は多岐にわたる。その代表的な例を以下にまとめる。

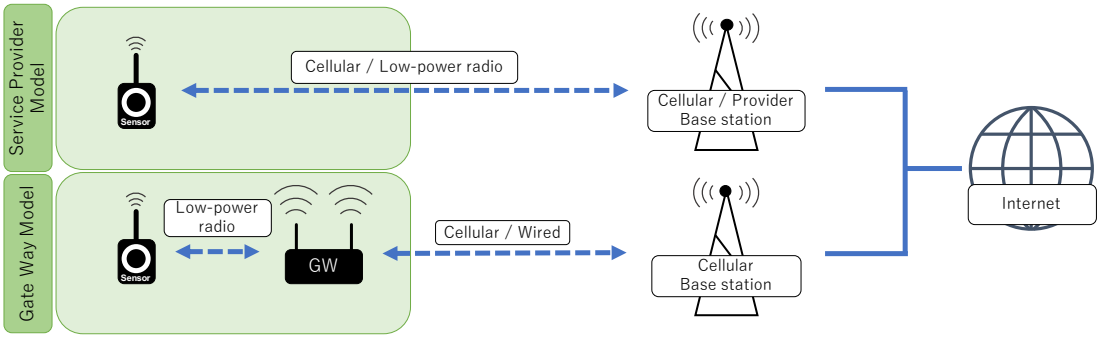

図 2.1: IoT のネットワーク接続モデル

2.2.1 ネットワークモデル

- **サービスプロバイダモデル**
  インターネットと直接接続する方式。セルラー通信は、基地局の設置が通信事業者によって行われており、携帯電話等と同様に広いエリアでの通信が可能。スマートフォンで利用する 3G や 4G などの高速データ通信回線を利用する。セルラー通信用の無線回路は複雑であり、消費電力が大きい傾向がある。IoT サービスプロバイダが敷設した IoT 用特殊無線網を利用する接続形態も同様の構成をとり、非常に低価格で提供されている。

- **Gate Way モデル**
  センサノードとインターネットの間に Gate Way と呼ばれる中間ノードがある接続方式。Gate Way をエッジと呼ぶことがある。IoT 向けの特殊な省電力無線通信規格を使用してセンサノードと Gate Way 間を通信し、データを集約、処理してインターネットと通信する。インターネットを直接経由すると往復の遅延が大きいため、レスポンス改善を目的に用いられ、インターネット上には必要なデータのみを送出する方式である。Gate Way とインターネット間の接続方法はセルラー通信を利用することがあるが、サービスプロバイダモデルと比較してセルラー通信の契約数を減らせる。

2.2.2 IoT 向け省電力ネットワーク

2.2.1 節で示したネットワークモデルにおいて、IoT 向けの特殊無線網ではサブギガ帯 (900MHz 帯) の電波を利用した LPWA (Low Power Wide Area) と呼ばれる規格群がある。また従来からある規格であっても、IoT 向けに省電力化した通信規格がある。これらのラストワンマイルに用いられる規格を挙げる [8]。

LPWA

LPWA/LPWAN (LPWA Network) とは、低消費電力・長距離通信を目的とした無線規格の総称である。LPWA は、免許を必要としない周波数帯域 (アンライセンスバンド) を使用するものと、免許が必要な周波数帯域 (ライセンスバンド) を使用するものに大別される。代表的なものを次に述べる。

- **LoRa**
  LoRa (LOng RAnge) は省電力で広範囲のエリアをカバーする無線通信方式である。また LoRa 規格を使用したネットワークを LoRaWAN と呼ぶ。LoRa とは単に変調方式を指すこともある。LoRa はサブギガ帯を使用し、日本国内においては、免許不要の特定小電力無線局が利用できる周波数帯の 920〜928MHz がこれに相当する。最大伝送速度は 250kbps 程度で、伝送距離は最大 10km 程度である。LoRa 規格はオープンな通信規格であり、普及を目的とする非営利団体の LoRa Alliance[2] が組織されている。

- **SIGFOX**
  フランスの SIGFOX 社が提供する無線通信規格である。各国につき一社が SIGFOX オペレータとして登録することで SIGFOX の無線基地局を設置し、サービスを提供する。日本では京セラコミュニケーションシステム [4] が事業者となって 2017 年からサービスを提供している。免許不要の 920MHz 帯を利用し、最大通信速度は 100bps で 2019 年現在は上り通信のみ提供しており、伝送距離は最大数十 km になる。主にセンサーをターゲットとしたサービス展開がなされている。

- **NB-IoT**
  Narrow Band IoT の略。高速データ通信を必要としない IoT 向け LTE 通信の規格である。3GPP で IoT 向けに Category 0 (Cat-0)、Category M1 (Cat-M1) といった仕様が策定されている。送受信するデータ量が少なく、移動しないような IoT 端末向けに Release 13 によって策定された。既存の LTE の基地局を使用することが可能で、ファームウェアの変更などによって NB-IoT の使用が可能になる。使用する周波数帯域は、通常の LTE 回線と同じ周波数帯域幅は 200kHz で、送信時はそれよりもさらに細い帯域を用いる。最大通信速度は受信時、送信時とも 100kbps 程度と低速で、建物の中や地下でも通信可能である。他の無線規格と比較して回路が複雑である LTE モジュールの価格を抑えて実装可能にしている。

省電力無線通信

LPWA 以外の規格で、IoT 向けの省電力な無線通信について挙げる。

- **Wi-SUN**
  Wi-SUN (Wireless Smart Utility Network) は情報通信研究機構 (NICT)、Elster、Itron、Landis+Gyr、Silver Spring Networks などが創設した業界団体、Wi-SUN アライアンス [6] が標準化、普及促進活動を行っている。スマートメータ等に使用される無線通信規格で、Smart Utility Network は、ガス、電気や水道のメータに端末を搭載して無線通信を用い、検針データを効率的に収集する無線通信システムである。物理層の仕様は IEEE にて標準化された国際規格の IEEE 802.15.4g である。通信速度は数百 kbps 程度だが、複数の端末がバケツリレー式にデータを中継することで遠隔地まで届けるマルチホップ通信に対応している。2.4GHz 帯やサブギガ帯で通信するため、日本国内では 920MHz 帯のアンライセンスバンドで割り当てられている。

- **BLE**
  BLE (Bluetooth Low Energy)[5] は Bluetooth 規格の省電力通信規格である。Bluetooth 4.0 以降に搭載される規格で、超省電力のためボタン電池 1 つで数年稼働する。2.4GHz 帯を利用して、最大 1Mbps の通信可能なことが特徴である。従来の Bluetooth と互換性がなく、最大 7 台であったが BLE に同時接続数の制約は無い。通信距離は 2.5m から 100m 程度である。

- **ZigBee**
  LAN や PAN (Personal Area Network) の規模で利用が想定されている。物理層と MAC 層は IEEE802.15.4 を使用することを想定されており、プロトコルの仕様は ZigBee Alliance[7] によって策定されている。通信速度は最大 250kbps、通信距離は製品などにもよるが、最大数十 m から数 Km 程度で、省電力や価格コストが低い。センサーネットワークの構築、家庭内では家電のリモコンや家電同士の機器間通信などでの利用が見込まれている。ZigBee 機器は中継局としての機能も有するため、1 つの基地局に各機器が繋がるスター型ネットワークだけでなく、すべての機器が通信範囲内の各機器と相互に繋がるメッシュ型のネットワークを構成できることが特徴である。

2.2.3 キャリアネットワーク

モバイル通信として広域で利用が可能な無線通信網も IoT で利用される。

- **LTE/4G (4th Generation)**
  スマートフォンなどで使用する高速大容量通信が可能な無線通信網である。LPWA 等の通信規格と比較して回路規模が大きく、消費電力が大きい。エッジノードとインターネットの間で使用が想定される。

- **5G (5th Generation)**
  5G は 3GPP によって仕様が企画されたセルラー通信である [53, 54]。高速・

広帯域・低遅延が特徴である。仕様策定の段階からIoTをサポートする動きがある。

2.3 検証に関する技術

IoTシステムのエンドノードの検証では、デバイスのアプリケーションの検証、センサやアクチュエータの検証、通信に関する検証などが考えられる。検証方法は検証するべき対象や範囲に対して適切なツールや設備がある。

以下では様々な検証ツールや検証環境について述べる。

2.3.1 実機による検証

IoTデバイスやシステムに採用する実機のデバイスを用いることが最も短適な検証方法である。近年は評価ボードにArduinoの様な組込みボードやLinuxが動作するRaspberry Piの様なシングルボードコンピュータが登場した。

2.3.2 ソフトウェアエミュレータやシミュレータによる検証

実機のデバイスを用意せず、コンピュータ内でソフトウェアを用いて再現する検証方法である。IoT以外にネットワークの検証のために使われるソフトウェアシミュレータがあり、ns-3[17]やQualNet[26]が代表的である。これらはプロトコルやアルゴリズム検証のために用いられることが多く、研究開発の初期段階で使用されることが見込まれる。またレイトレーシングの様な無線の伝搬をシミュレートするものもある。

一般にエミュレータはシミュレータと比較して忠実に機能を再現するため、正確な検証が可能である。エミュレータは機械やコンピュータのハードウェアを模倣しており、例として、モバイル端末用のクロス開発環境、ゲームやプログラムなどを別の環境で動作させるものがある。プロセッサエミュレータの代表例として、QEMUを紹介する。

QEMU

QEMU[42, 43]は、オープンソースのプロセッサエミュレータである。マシンエミュレータとバーチャライザの2通りの利用が可能である。マシンエミュレータは、あるシステム用に作られたOSやプログラムを別のマシン上で動作させる目的で利用する。バーチャライザは、ターゲットCPUのためのアプリケーションをQEMUが動作するCPU上で直接実行することを可能にする。QEMUを利用することで、OSを用いるIoTデバイス、あるいはOSを用いない組込みデバイスのア

プリケーションを検証が可能になる。一方で、QEMU はセンサ IoT などで利用されるセンサモジュールや通信モジュールが十分に開発されていない。

2.4 検証基盤

ここで述べる検証基盤とは、システム検証基盤を指す。検証基盤は開発中のシステムをテストするシミュレータである。検証基盤はシステムを検証するための基盤 (インフラストラクチャ) であり、その枠組みの中にはテストに必要なソフトウェア・ミドルウェア、計算機資源、ネットワーク資源やそれらを管理するための機構を持つ。ある大規模なシステムを開発する際に実際の運用環境と近い試験環境を構築できる。

テストベッドのトレンドに関する Gao らの研究 [11] を参考に、検証基盤の種類と特徴を紹介する。

2.4.1 物理検証環境

IoT やネットワーク、IT システムの検証を実機のハードウェアを用いる方法である。サーバやネットワーク機器を用意し、それらを検証対象のシステムとして組み上げることで検証する。

これらのような形態をとる物理検証環境を紹介する。

インターネット上のテストベッド

インターネット上で動作するシステムの検証を行うために、実際のインターネット上にトラヒックを送信するテストベッドがある。この様なテストベッドに PlanetLab[23] や GENI[24] がある。インターネット上の振る舞いやスループット等を検証するために用いるが、実際のインターネット環境は時々刻々と変化するため、システムの問題究明などの本質的な性能や振る舞いを調べるには向かない。

- **PlanetLab**
  世界中の様々な研究機関等が検証用のサーバをホストしており、そのすべてがインターネットに接続している。Linux ベースの OS を含む一般的なソフトウェアパッケージを実行することが可能である。ノードの起動やソフトウェアのアップデート、健全性のチェックなどの管理制御する機構を備えている。

- **GENI**
  GENI はアメリカ合衆国 National Science Foundation によって支持されている大規模な実験インフラストラクチャである。ネットワークおよび分散システムの研究と教育のための仮想的な研究環境を提供している。GENI は仮想

マシンやベアマシンなどのコンピューティングリソース、リンク、スイッチ、WiMax ベースステーションなどのネットワークリソースを含んでおり、広域に分散しているリソースにアクセスすることが可能である。

インターネットから独立したテストベッド

インターネットから独立したテストベッドは、外乱となるネットワークトラヒックを遮断し、研究、実装段階の秘匿したいシステムを検証できる。
この様なテストベッドの例を示す。

- **StarBED**
  NICT 北陸 StarBED 技術センター内に設置されている総合ネットワークテストベッドである。StarBED[15, 52] は数万 CPU のコンピューティングリソースとネットワークスイッチによって構成されたハードウェア、OS の配布を可能にする支援ソフトウェアを有している。これらによって実際のネットワークで起きる事象をリアルなスケール感でエミュレートすることを可能にしている。

IoT の実証実験のためのテストベッド

- **NICT キャラバンテストベッド**
  NICT が提供する IoT 環境を構築できる可搬式システムのキャラバン型テストベッド [13] である。多様なセンサデバイス、通信デバイス (Wi-Fi、LPWA、LTE、衛星通信)、可搬式サーバ・エッジノードやバッテリを有する。ラストワンマイルの検証に用いることが可能で、現地での実証実験に用いられる。

- **町を利用した広域実証基盤**
  NICT と神奈川県横須賀市が主体となって提供する、町そのものをテストベッドとして活用する取り組みがある [14]。IoT で利用が期待されている LoRa、SIGFOX、Wi-SUN の 3 種の無線規格を使用したアプリケーション検証のためのテストベッドである。低コスト・短期間で LPWA を用いた実証実験を行うことができる。
  他にも SmartSantander プロジェクトでは、ベオグラード、ギルフォード、リューベック、サンタンデールで 2 万個のセンサーを展開し、さまざまな技術を活用する町を利用した実証実験研究施設を構築している [40]。
  一方でこれらは、特定の都市を用いた実験であるため、サービス事業者が真に実装を望む環境ではない。したがって、サービス事業者は類似した地形を探す、あるいは設置したい土地で検証する必要がある。

2.4.2　クラウド環境

Amazon Web Service や Google Cloud Platform、Microsoft Azure のようなパブリッククラウドに代表される IaaS、PaaS のサービス展開がなされている。これらのクラウド環境は実機でコンピューティングリソースを用意する必要がなく、Web ブラウザ上のコントローラで必要なリソースを利用できる。サービスによっては時間単位の課金システムを持っており、テスト環境としても使用できる。しかし、仮想化されたコンピューティングリソースは、ハードウェア資源を共有しており、ノード中の他の仮想マシンの負荷がテスト対象のシステムに影響を与えることが考えられる。また、再度利用申請した仮想マシンと同じハードウェアを利用できるとは限らず、これに付随するネットワーク機器も同様のものとは限らない。したがって検証の再現性を持たないことが想定される。

第3章　ソフトウェアテストのための IoTデバイスエミュレーション

この章ではIoTデバイスで使用するソフトウェアのテストとIoTデバイスエミュレーションの特徴に関して述べる。

3.1　ITシステムとIoTシステムの共通点と相違点

3.1.1　共通点

サーバ・クライアントモデルに代表される様に、ITシステムは光回線や無線通信を介して計算処理やデータを送受信する形態を採用していることが多い。IoTシステムも末端のノードがネットワーク回線を通じてデータを送受信してサーバ等で処理をする点が共通している。

インターネット越しにサーバを利用すると往復の遅延が大きいため、レスポンス改善を目的にエッジコンピューティングを行うシステムもある。2.2節で示したGate Wayモデルのように、IoTのシステムでもエッジを用意してレスポンスを改善を目指す場合がある。

3.1.2　相違点

IoTシステムは末端のノードが組込みシステムに類似しているため、通常のITシステムと大きく異なる。この様な組込みシステムはエンドノードデバイスが非常に非力な計算リソースを有しており、センサやアクチュエータが接続されている。またセンサIoTデバイスでは、非常に小さなデータを高頻度に、あるいは定期的にデータを送信する。例として圃場の温度センサを考えると、一定間隔に多くのセンシング用IoTデバイスからの温度情報が一斉に送信される。

以上のような共通点と相違点の特徴があるため、従来のITシステムの要素と組込みシステムの特徴の融合的な視点が必要である。またIoTではインターネット

を通じて遠隔地をセンシングする、あるいはアクチュエータを使って動作することが期待されている。

3.2 IoTデバイスエミュレーションの要件

3.1節で示した特徴を持つIoTシステムを検証するために、IoTデバイスやそれに付随するITシステムをコンピュータ内に構築する必要がある。IoTデバイスは組込みシステムの要素を持つため、汎用コンピュータのアーキテクチャではなく組込み用プロセッサのアーキテクチャを有するエミュレーション環境が必要になる。例として、IoTエンドノードでアプリケーションが動作するプロセッサにはARMアーキテクチャのプロセッサやAVRマイコンなどがある。

IoTデバイスはデータの転送のために、省電力無線通信や高速インターネット接続が用いる。無線を通信に利用する際、様々な問題が生じる。例として、無線は複数の端末が同時に利用すると衝突を起こし、衝突を検知すると再送処理を行う。これは小型電池で駆動するIoTデバイスでは問題となる。また他にも無線装置が基地局と通信する際に接続が不安定になる場合がある。このような問題は、時系列にデータ取得する計測においてデータの欠落を発生させる。これらの問題を解決するためには、無線通信に関する事前検証が重要である。

IoTシステムはデータの蓄積や分析のためにITシステムを持つ。したがってIoTシステム全体の検証のためには、サーバ、アプリケーション、ミドルウェアやネットワーク機器が必要となる。

3.3 エミュレータを用いる利点

IoTデバイスで使用するソフトウェアの検証のために、エミュレータを用いる利点を以下にまとめる。

3.3.1 規模性

コンピュータ内でプログラムとして実行するエミュレータは、ソフトウェアによる大規模展開のスケーラビリティが得られやすい。ハードウェアをテストするために必要な個数を用意する必要がある。またソフトウェアとハードウェアの開発を分離して行う並行開発が可能である。

3.3.2 バイナリ互換性

検証するIoTデバイスのプロセッサアーキテクチャをもつエミュレータを用意することで、検証用のアプリケーションと実機に搭載するアプリケーションプロ

グラムに同じものが使用可能である。つまりプログラムのソースコードから生成されたバイナリファイルは検証環境と実機環境でバイナリレベルの互換性を持つことが可能である。シミュレータを使用した検証環境では必ずしもバイナリが互換性を持たないため、エミュレータを用いることでこの問題点が解決される。

3.3.3 意図したシナリオの実験

コンピュータ上で再現するエミュレーションでは、実機の IoT デバイスでは難しい検証を行うことができる。例として、通信を意図的に切断したときの振る舞い、IoT デバイスの電源が喪失したときの振る舞いを実験することが可能である。これらのようにシナリオを設定して任意のタイミングで意図した振る舞いを実験可能である。

3.3.4 実地検証における時間的コスト削減

エミュレータを用いることで実地における検証作業を減らし、時間的コストを削減可能である。市街地や圃場に広域分散配置される IoT デバイスは、設置のための時間的コストがかかる。IoT デバイスの通信は省電力な特殊無線規格が使用され、通信帯域がソフトウェアのアップデートに十分でなく、規格やサービスの仕様上の理由からアップデートの配布が困難な場合がある。また小型電池を利用してデバイスを駆動する場合、アップデートのために長時間通信することは連続稼働時間を縮小してしまうため、避ける必要がある。以上の理由から、IoT デバイスのアプリケーションをアップデートするためには、一度回収して再設置する必要がある。したがって、エミュレーションによる検証を採用することで、実地検証の回数を最小限に留め、再設置を減らすことが可能となるため、時間的コストを削減することが可能である。

3.4 エミュレータを用いる欠点

エミュレータを用いることによる種々の問題点を以下で述べる。

3.4.1 開発量増加

IoT デバイスをエミュレーションするために、対象のプロセッサアーキテクチャを有するエミュレータを用意する必要がある。既存のエミュレータが無い場合、IoT デバイス開発者が新たにエミュレータを開発する必要がある。

3.4.2　実物と比較した忠実度の低下

実物をモデル化して動作を模倣するエミュレータは、実機と比較して完全に忠実とは言えない。これはプログラムとして実装する以上避けられない問題であるが、その差異を明らかにしたうえで許容して使う必要がある。またここで述べる忠実とは、実機の機構とより近い実装がなされていることを言う。忠実度については 7.2 節で議論する。

3.5　忠実度と抽象化間のトレードオフ

忠実度と抽象度の間にはトレードオフの関係があり、抽象度を高めると忠実度が低下する。特にこれらのトレードオフは、特定の処理の処理時間やエミュレータ実行における計算リソースの増減に影響する。本研究で述べる高い忠実度とは、実機と同じアプリケーションがエミュレータ上で動作可能であることを指す。エミュレータによるアプリケーションの実行にはオーバーヘッドがあり、高い忠実度を有するエミュレータは多くの計算リソースを必要とする。一方でエミュレータの実行を抽象化することにより、実行時の要求計算リソースは削減される。エミュレータの機能を正確に実装しないでアプリケーションを実行することにより、オーバーヘッドの大きいエミュレータの処理を経ずに処理の実行が可能となるためである。これらにより、エミュレータの忠実度を適切に保ちつつ、機能の一部を抽象化して実装することで、必要な機能を模倣しながら同時に実行できるエミュレータ数を増やすことが可能である。

第4章　関連研究

IoT に関わる研究、ネットワーク検証技術やシミュレーションやエミュレーションに関する関連研究についてまとめる。

4.1　IoT エミュレーションに関する研究

IoT のような大量のデバイスをコンピュータでエミュレーションあるいはシミュレーションするための研究を紹介する。

4.1.1　エミュレーション環境に関する研究

RUNE

中田は StarBED の様な PC クラスタ環境において、ユビキタスネットワークのシミュレーションの実行を支援するプラットフォームである RUNE を開発した [19]。ユビキタスネットワークシステムのシミュレーションは、シミュレーション固有のロジック、シミュレーション一般に共通のロジック、シミュレーションの制御を司る部分から構成されると述べられている。RUNE はこの中で、シミュレーション一般に共通のロジックとシミュレーションの制御に関する部分を提供し、利用者が用意したシミュレーション固有のロジックと協調して動作可能である。クラスタ環境で多くのコンポーネントを利用して動作するためには、その各コンポーネントが独立して動作し、それらとの通信するための手段が必要である。クラスタ環境ではこの様な要件を適切に隠蔽する必要があるが、RUNE ではこの機構を提供することで分散シミュレーションを容易にすることを可能にしている。

ユビキタスコンピューティングにおけるシミュレーションやエミュレーション環境の要件

Reynolds らはユビキタスコンピューティングにおけるシミュレーションやエミュレーション環境に必要な要件について検討し、エミュレーションとシミュレーションの融合的な環境を構築した [36]。

ユビキタスコンピューティングの典型的なシナリオには多くの問題がある。地理的に広い空間に物理的に分散している多数のデバイス、さまざまな種類のセン

サやアクチュエータを含むハードウェアの問題、およびそのようなデバイスを使用する多くのアプリケーションフレームワークなどの問題がある。この研究では、ユビキタスコンピューティングのためのシミュレータの設計には、センサやアクチュエータをモデリングできること、アプリケーションフレームワークが利用可能であることや環境をモデル化して利用することが必要であると述べている。

また、このようなシミュレーションでは様々なモデルシミュレータを利用して、コンポーネントを複合的に使用するための機構が必要となる。これらを実現するために、センサパイプラインとアクチュエータパイプラインモデル、そしてレイヤスタックを定義し、様々なコンポーネントのモデリングにおいて、より大きな柔軟性を持つことを可能にしている。

4.1.2 検証環境の構築とスケーラブルなエミュレーション環境展開に関する研究

GUAN

岩橋は StarBED の様なコンピュータクラスタ環境上で IoT デバイスを対象とした実証実験用フレームワークを提案している [20]。汎用コンピュータのアーキテクチャと異なるプロセッサアーキテクチャを持つ IoT デバイスを汎用 PC ノード上で仮想的な IoT デバイスとして動作させる。また GUAN (Generic Utilization of Assorted Networking) フレームワークを構築し、ユーザが任意に選択したネットワーク技術を利用可能にしている。これらによって、IoT デバイスエミュレーションにおける大量のデバイスエミュレーションを行う際のデバイス展開や、ネットワーク利用について管理する機構を開発した。

4.2 組込みソフトウェアのテストに関する研究

Justitia

Seo らは組込みソフトウェアテストのためのツールである Justitia を提案している [34]。この研究において、組込みシステムは図 4.1 に示す Hardware、HAL、Kernel、アプリケーションの層で構築されており、これらの層はインタフェースを通じてやり取りをすると述べられている。このインタフェースに対してブレークポイントやテストケースを自動で生成するエンジンを作成することで、組込みソフトウェアのテスト支援を行う。

この研究で定義されている層モデルは、Jerraya らの組込みシステムのハードウェアとソフトウェア間のインタフェースデザインに関する研究 [35] をもとにしている。

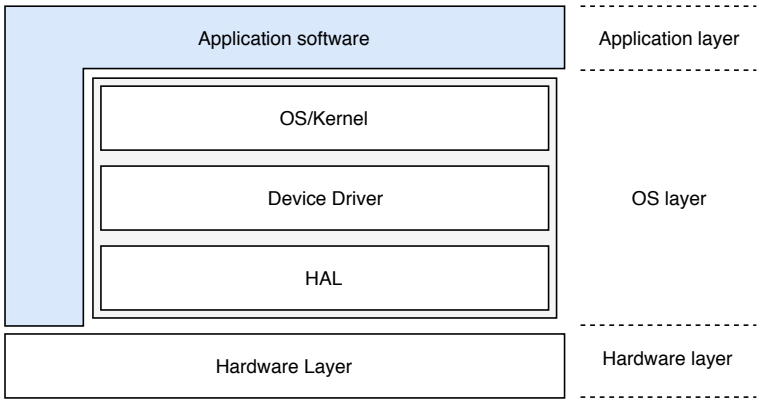

図 4.1: Justitia のソフトウェアテストのためのレイヤモデル

テストケースを生成するために、組込みシステムの層間のインタフェーステスト手法と組込みソフトウェアのインタフェースパターンを定義している。そのパターンを EmITM (Embedded system Interface Test Model) のテスト機能にマッピングする。また、テストケースを実行するためのエミュレーションテスト手法を提案している。

QEMU と仮想 GPIO を用いた組込みソフトウェア検証環境の構築

Platt は QEMU [42, 43] 環境下で、仮想 GPIO (General-purpose input/output : 汎用入出力) エミュレータを実装しており、Raspberry Pi と周辺機器のエミュレーション環境を構築した [44]。QEMU は ARM 等のプロセッサをエミュレーション可能なため、Raspberry Pi を模倣して OS を動作させることが可能である。しかし、QEMU では GPIO に相当する機能が無いため、UART や SPI などの通信インタフェースを用いる周辺機器を含んだテストが難しい問題があった。そこで、GPIO 相当の機能を模倣するレジスタを定義し、QEMU 内に内包した。またレジスタへのアクセスは、QEMU が動作するホストコンピュータ上の共有メモリにアクセスすることで実現している。クロスプラットフォームアプリケーションフレームワークの Qt [45] を用いることで、LED モジュールとボタンモジュールを実装し、この実装方法の有効性を検証している。

ARMISS

Lv らは ARM の命令セットシミュレータ (以下 ISS) である ARMISS の実装と評価をした [46]。ISS の実行方式はインタプリタ方式とバイナリ変換方式があるが、ARMISS ではインタプリタ方式を採用している。これは動作させるアプリケーションを ARM アーキテクチャの命令セットに逐次変換して実行する方式である。一方で、インタプリタ方式はバイナリ変換方式と比較して動作が遅い。解決策として

キャッシュ機構を実装し、キャッシュの有無がどれだけ高速化に貢献したか示している。

組込みシステム開発において、ソフトウェアとハードウェアの並行開発が行われるが、ソフトウェアの検証はハードウェアのプロトタイプの完成、あるいは製品の製造が完了しなくてはならない。ARMISS はハードウェア無しにソフトウェアの検証を可能にすると述べている。

4.3 無線通信シミュレーションに関する研究

IoT ではほとんどの場合、無線通信を利用した通信を行う。2 章では IoT で利用されるネットワーク接続方法や通信方式について述べたが、様々な無線規格の利用が想定されている。通信における関連研究についてまとめる。

AOBAKO

AOBAKO は湯村らによる研究成果 [22] で、BLE のエミュレーションフレームワーク BluMoon [18] を用いた BLE アプリケーション検証システムを構築した。BluMoon は BLE (Bluetooth Low Energy) をコンピュータ上でエミュレーションするためのフレームワークである。AOBAKO は BLE を使用したビーコンシステムのエミュレーションを実行するが、従来のエミュレーションシステムと大きく異なるのは、エミュレーションのパラメータの注入方法と結果の表示方法である。物理空間上のコンテキスト (位置情報) を仮想空間 (エミュレーションを行うコンピュータ内部) にパラメータとして取り込み、エミュレーション結果を実機のハードウェアから発信すると同時に現実空間のディスプレイでビジュアライズ可能である。

図 4.2 は発表資料 [41] より引用して示す。物理空間上のコンテキストは AOBAKO DESK 上の紙でできたビーコンチップの位置を Web カメラから取り込み、BluMoon に送信する。位置情報を考慮して BluMoon 上で計算した電波強度等の情報を BLE フレームにのせて送信する。AOBAKO SCOPE は BLE の Beacon の送信状況を映像にして表示する。またエミュレーションされた結果を実機のスマートデバイスで受信するために、AOBAKO BOX を用いて、BluMoon の生成した BLE フレームを現実の空間へ送信する。送信されたフレームはスマート端末などで受信することで、BLE アプリケーションの検証が可能となる。これらの機構により、わずか数十 cm 四方の AOBAKO DESK 上で、数十 m ある建物を再現し、BLE Beacon の配置を動的に変化させることで、スマート端末が受け取ることができる BLE フレームをインタラクティブにエミュレートすることを可能にする。

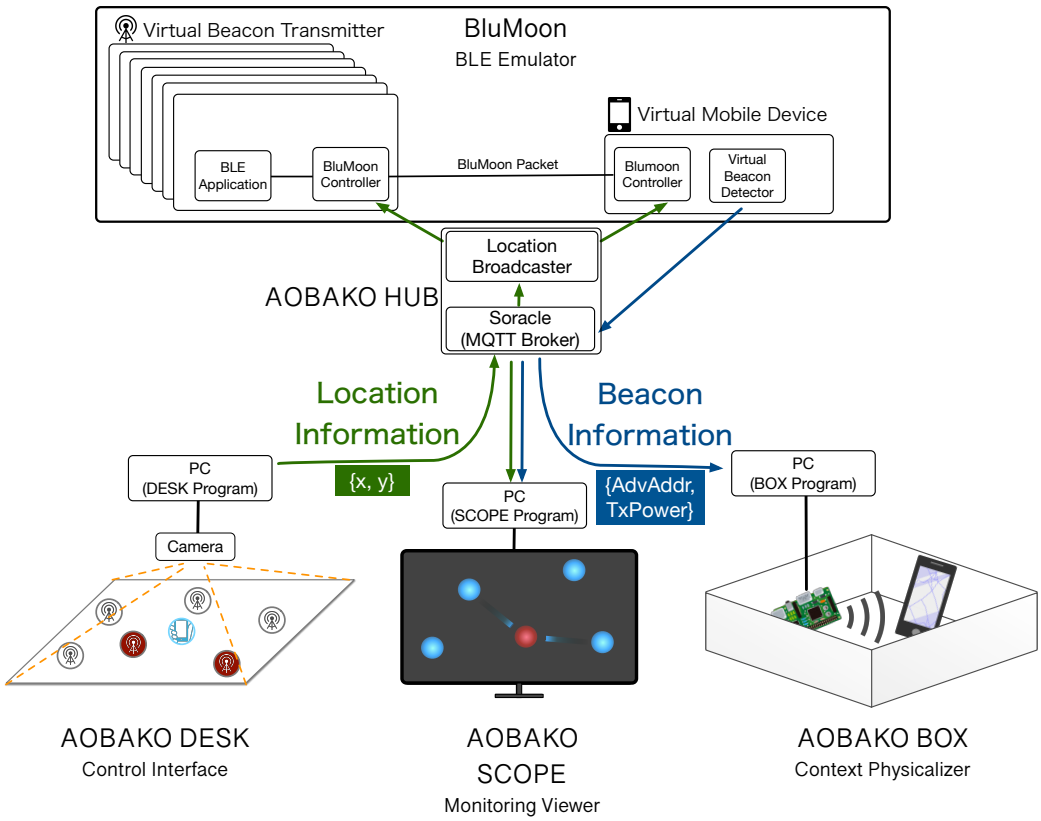

図 4.2: AOBAKO のデータフロー

NETorium

明石は有線ネットワーク環境下でワイヤレスネットワークエミュレーションに関する研究 [21] を行っている。NETorium はリンクエミュレータの Meteor、仮想無線ネットワークエミュレータの Asteroid から構成される。Asteroid は様々な通信メディアの模倣を可能にし、規模追従性を得るために有線ネットワーク上に仮想的な無線ネットワークを構築する。NETorium は特定のネットワークプロトコルに依存しないエミュレーション機構を持つ。これによりネットワークの様々なレイヤで動作するソフトウェアの検証が可能であり、無線規格で定義されている認証などの技術を使用することが可能である。

本論文中では小型端末のエミュレーションに関して述べられており、以下で述べる ContikiOS の Cooja と Asteroid の連携により、ネットワークテストベッド上に展開可能であるとしている。

ContikiOS

ContikiOS[28, 49] は、IoT 向けのソープンソース OS である。低電力無線規格である 6lowpan[9]、RPL[10]、CoAP[50] や IPv4、IPv6 をサポートしている。組込み用の OS として利用することが可能で、モニタリングや街灯の ON/OFF 制御等に利用ができる。ContikiOS には Cooja[51] と呼ばれるネットワークシミュレータを持ち、IoT (モート) をシミュレーションするためのツールキットが用意されている。API を用いてモートの動作コードを記述することで、GUI 上でモートのシミュレーションを可能にしている。

4.4 本研究における取り組み

市街地や圃場等での大規模分散配置されるセンシングデバイスは、無線通信を使用することが想定される。無線を使うシステムでは、無線特有の問題として無線フレームの衝突やバックオフなどの問題があり、信頼性のある End-to-End の通信を行うためにはアプリケーションでの担保が必要となる。また、IoT デバイスの省電力のための仕組みや、LPWA に代表される省電力通信は帯域が狭く、ベンダによる通信制限がかかることがあるため、搭載されるアプリケーションソフトウェアに欠陥があった場合、その修正をすることが困難である。ソフトウェアの検証では、実機と検証環境で同じプログラムを使用可能であることが理想である。そこで、コンピュータエミュレーションを使用する IoT のソフトウェア検証について考える。エミュレーションでは、プロセッサが IoT のエンドノードが有するアーキテクチャと同じにすることで、同じプログラムを利用できる。しかしプロセッサのエミュレータはプログラム実行のオーバーヘッドが大きい。そこで、アプリケーションソフトウェア検証のためのプロセッサエミュレータの機能を抽象化することを考える。プロセッサエミュレータの実行を独自定義した層モデルを適用し、モデル分類したレベル別でのエミュレータの抽象化を実装する。

以上から本研究では、適切な忠実度を保ちながらエミュレータの機能の一部を再現せずに抽象化して実装することで、エミュレーションにおける要求計算リソースの削減をはかり、その提案手法の有効性について検証する。同時にエミュレーションの抽象化がもたらす IoT デバイスエミュレーションの抽象化モデルの忠実度とテスト可能範囲のトレードオフ関係について考察する。また、IoT デバイスエミュレーションが IoT システム開発にどの様な影響をもたらすかについて考察する。

第5章　エミュレーションにおける抽象化モデル

本章では、本研究におけるエミュレータの抽象化のために用いる抽象化モデルについて述べる。

5.1　抽象化する対象

環境センシングのためのIoTデバイスは、一般に図5.1で示すプロセッサ、センサと通信モジュールで構成される。プロセッサとセンサの間を通信するためには様々な通信プロトコルやインタフェースがある。代表的な通信インタフェースとしては、SPIやI2Cなどがある。これらの通信インタフェースや機構をエミュレータに実装しなくては、センサを有するIoTデバイスのソフトウェアの検証ができない。このインタフェースを通じてセンサや通信モジュールのエミュレータを制御する。実機のインタフェースは電気信号を用いて、ビット列のパルスを通信プロトコルにしたがって送受信している。しかし、この様な処理をコンピュータ内で再現する必要はない。テストに必要なインタフェースの役割は、制御のコマンドやデータの送受信が可能にすることである。したがって、IoTデバイスエミュレーションでは必ずしもこれらの通信プロトコルやインタフェースを忠実に実装する必要がない。つまりソフトウェアの動作を検証する際は、外部のセンサや通信モジュールとの間の通信は抽象化できる。

5.2　概念モデル

5.2.1　抽象化レベル

エミュレータの実装にあたり、図5.2に示す層型のモデルを定義する。エミュレーションの実行は大きく4つのレイヤで抽象化度が変わる。このモデルでは、以下の資料を参考にして定義している。1つは、NICTの次世代ユビキタスエミュレーション技術開発プロジェクトの報告書[39]において、中田らによって開発されたRUNEによるマルチレベルエミュレーションの複数のエミュレーション形態の概念を用いた。他に4章で言及したSeoらのJustitia、Jerrayaらのインタフェー

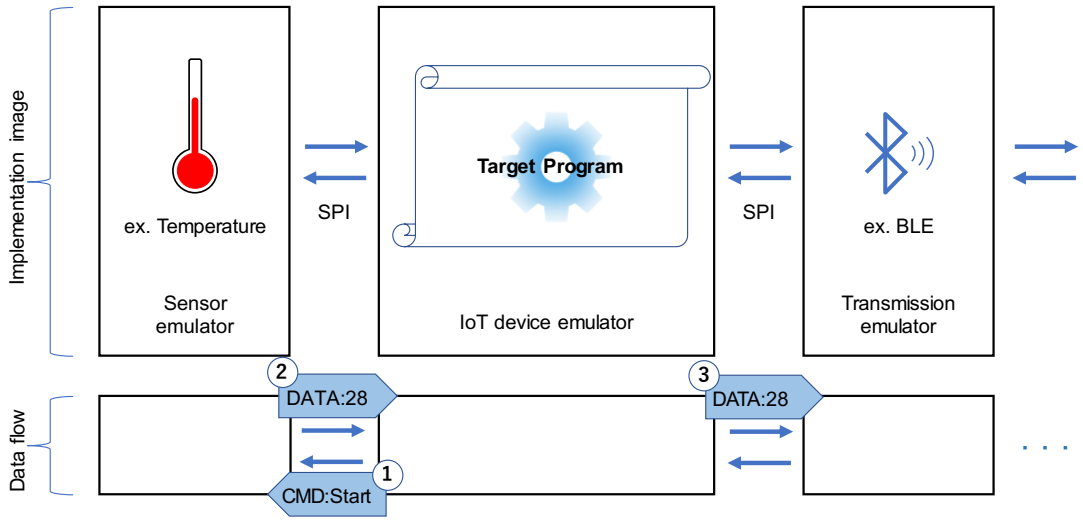

図 5.1: エミュレーションの構成イメージ

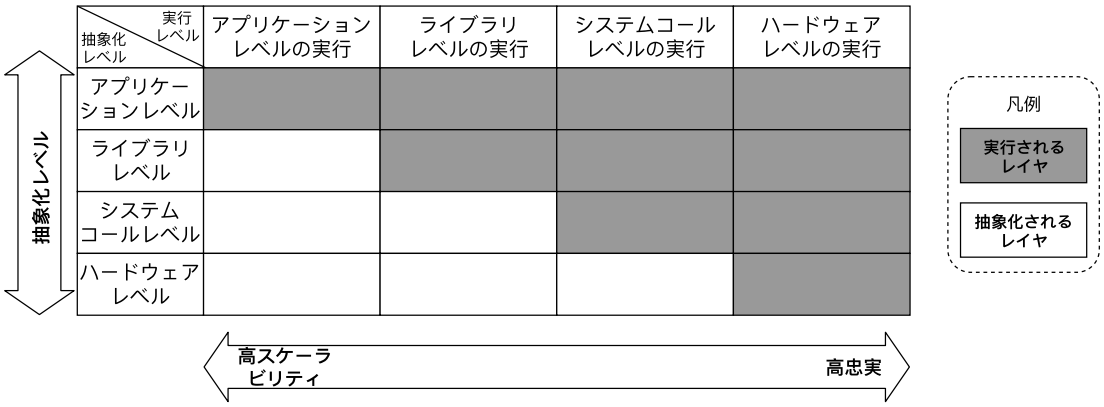

図 5.2: 抽象化レベル

スデザインに関する研究のコンセプトなどを参考にしている。これらの研究で示されているハードウェアは、本研究中ではハードウェアあるいはレジスタと定義している。また組込みソフトウェアは、より詳細に分類するとアプリケーションと関連ライブラリであるから、ライブラリの層を定義している。OS/kernel について、システムコールレベルを定義した。組込みシステムは OS を搭載するとは限らない。組込みデバイスにおいて、OS に期待する機能の 1 つとして、ハードウェア (レジスタ等) を操作する機能がある。これはシステムコールによってハードウェアにアクセスするからである。以上にしたがい、抽象化レベルを定義した。また図中に示した抽象化レベルについて、以下に詳細を述べる。

- **アプリケーションレベル**
  もっとも抽象度の高い実装である。アプリケーションは次に示すライブラリを含むプログラムソフトウェアである。

- ライブラリレベル
  ハードウェアなどを使用する場合、それらを操作する関数を定義したライブラリを利用する。

- システムコールレベル
  ハードウェア操作をする、あるいはOSが提供するシステムコールのレベルがある。IoTのエンドデバイスはOSを搭載しているものが少なく、組込みのプロセッサを利用していることが多い。一方で、エンドデバイスで使用されるプロセッサも高性能化していることもあり、RTOS[27]やContikiに見られるようなIoT向けのOSを搭載しているものもある。

- ハードウェアレベル
  アプリケーションを実行する際にはプロセッサの命令セットをエミュレーションする必要がある。プロセッサが使用するレジスタに限らず、外部周辺機器との通信に使用するハードウェアもレジスタを有している。

5.2.2 抽象化間のトレードオフ

エミュレーションの抽象化は3.5節で言及した様に、抽象化レベルと忠実度との間にはある種のトレードオフを持つ。

5.2.1節で論じた抽象化レベルに対して、抽象化によるトレードオフについて表5.1にまとめている。表中ではハードウェアをH/Wと表記している。

アプリケーションレベルの抽象化は、この4種類の中でもっとも高い抽象化レベルのエミュレーションである。高い抽象度を持つことは、検証範囲を狭めることと同義である。ライブラリ以下を抽象化するため、ライブラリ以下の機能を使用する模倣をしない。そのためエミュレーションが軽量である反面、表中の検証可能項目で列挙したように、アルゴリズムやシステムの起動/終了のような各種機能の抽象的あるいは俯瞰的な検証項目のみテストできる。

ライブラリレベルの抽象化はシステムコール以下を抽象化している。ハードウェアを用いないIoTアプリケーションのテストを行うことが可能である。ハードウェアの抽象化によって、一般に処理が軽くなる。

システムコールレベルの抽象化はハードウェアを抽象化する。システムコールは、ユーザプログラムがハードウェアを操作する際、ハードウェアを抽象化するための関数群で、通常はOSが提供する。OSを用いたテストやドライバプログラムを用いたテストが可能である。

ハードウェアレベルの抽象化は最も低い抽象度の実装である。低い抽象度はより実物に近いエミュレーションを行う。プロセッサやその他のアプリケーション実行に関する諸要素を忠実にエミュレーションするため、エミュレーションの実行が重くなる一方で、詳細な検証が可能である。

表 5.1: 抽象化レベル図における抽象化間の性質

	アプリケーション レベルの実行	ライブラリ レベルの実行	システムコール レベルの実行	ハードウェア レベルの実行
抽象化 する層	ライブラリ 以下	システムコール 以下	レジスタ 以下	-
I/F	ライブラリの 関数	ライブラリが使う システムコール	レジスタを 操作する関数	プロセッサの 命令セット
検証可 能項目	アルゴリズムや 各種機能テスト	H/W を用いない プログラムテスト	H/W 無し、OS 機能ありのテスト	実機並みの 詳細なテスト
利点	実装により軽量	中程度軽量	ドライバプログラム 等をテスト可	実機に近い テスト環境
欠点	実装によりテスト 範囲が狭い	H/W 機能を用い たテストは不可	H/W 機能を用い たテストは不可	忠実ゆえに 動作が重い

これらに示したように、この 4 種のレベル (層) で抽象化する場合、抽象化した機能が隠蔽され検証できなくなる。

5.3 抽象化するエミュレーションモデル

5.2.1 節で示した抽象化レベルに則って、エミュレータの抽象化モデルを設計する。

本研究では、プロセッサとセンサの間を通信する SPI 通信に着目し、このインタフェース間の処理を抽象化する。SPI 通信の仕組みは図 5.3 の (1) に示す様に行われる。4 種類の信号線があり、CLK は送受信のためのクロック信号 (線)、CS は通信対象を選択するチップセレクト信号 (線)、MISO は通信対象 (スレーブ) からマスタ (CPU/MCU プロセッサ) にデータを送信するデータ信号 (線)、MOSI はマスタからスレーブにデータを送信するためのデータ信号 (線) である。クロックがあり、CS 信号が HIGH になる間のみ、データの送受信がなされる。データは通信は同時かつ双方向に可能である。

(1) で示した形式の SPI 通信は電気パルスやプロトコルを実装しているが、この様なエミュレータはほとんど無い。今回使用するエミュレータは (2) で示すような、SPI のインタフェースの入口と出口のみ持ち、データの送受信を行うより簡易な実装となっており、SPI のハードウェアに該当する実装はない。

そこで本研究では、SPI 通信の抽象化を考慮するために、2 つの抽象化レベルでの実装をする。1 つ目はハードウェアレベルの抽象化をした SPI 通信を行うエミュレータを実装する。2 つ目にライブラリレベルで抽象化した SPI 通信を実装する。

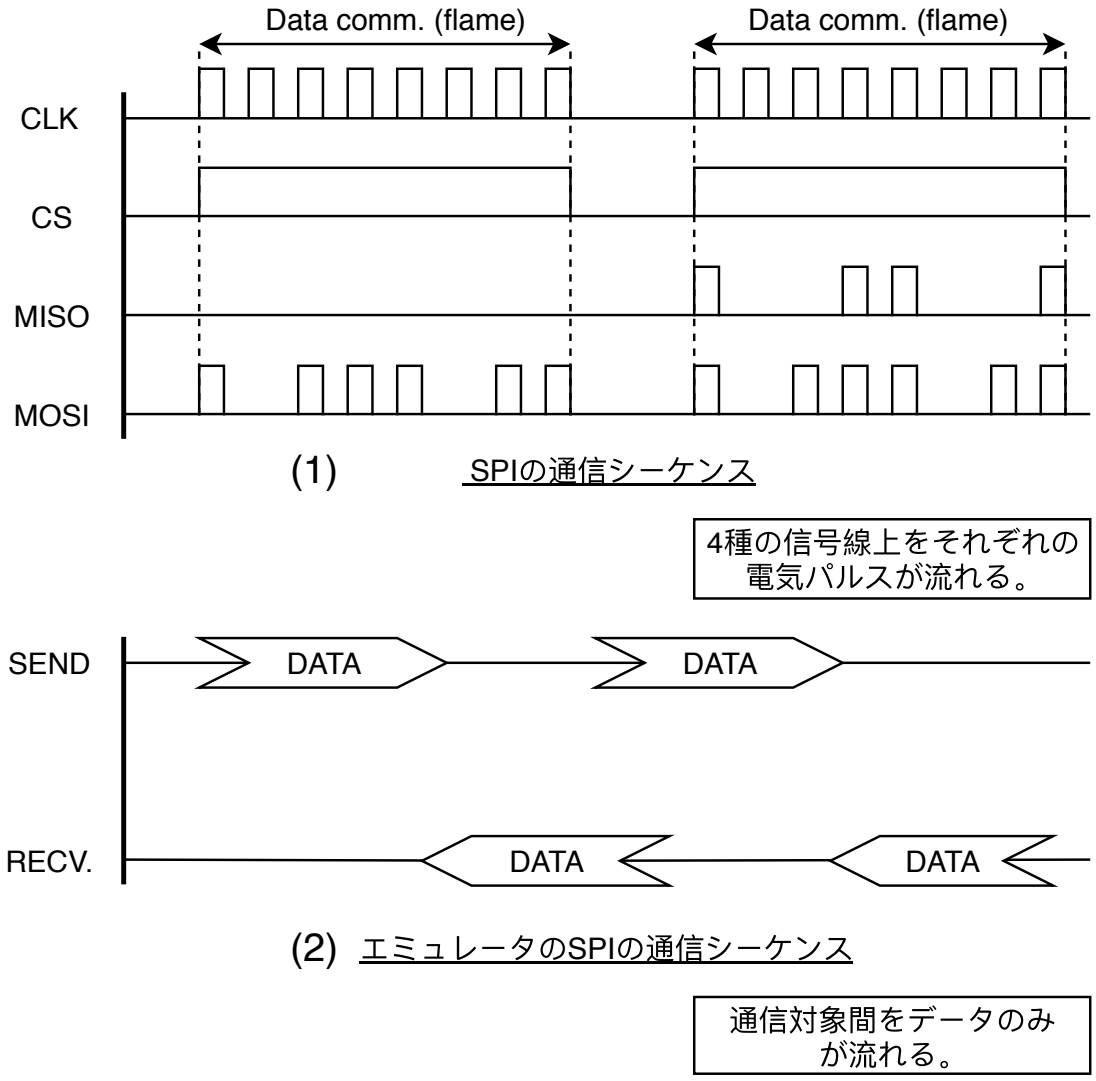

図 5.3: SPI の通信フロー

5.3.1 SPI レジスタレベルエミュレーション

ハードウェアレベルの SPI 通信では、図 5.4 の左側のフローで示すように、テスト対象のアプリケーションソフトウェアはライブラリやシステムコールの機能を使い、SPI 通信を行う。送信したいデータを送信用レジスタに書き込み、データを受信用レジスタから読み取ることで、SPI 通信によるデータの送受信が成り立つ。

5.3.2 SPI ライブラリレベルエミュレーション

図 5.4 の右側のフローでは SPI 通信をライブラリのレベルで抽象化している。システムコールやレジスタを通して通信せず、ライブラリの SPI 通信に関するメソッドがコールされたタイミングで、送信したいデータをエミュレータの外部に出力

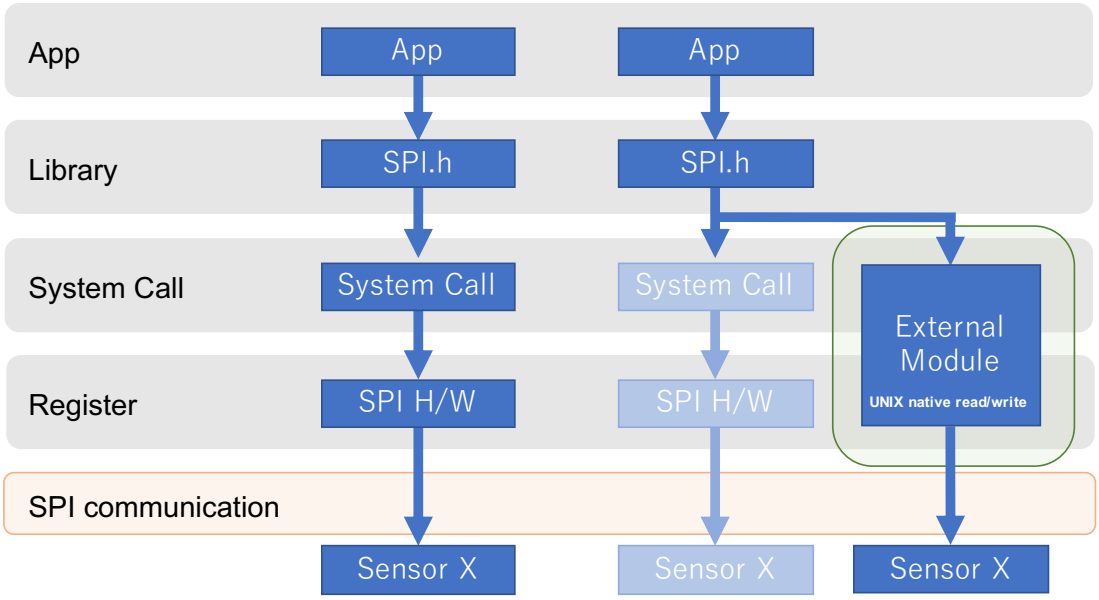

図 5.4: ライブラリレベルの抽象化実装例

する。図中の External Module は、プロセッサエミュレータの read/write 処理を用いず、UNIX/Linux 上で動作するプログラム等を表している。これにより、エミュレータの内部の処理を離れたデータ処理となる。以上により、SPI 通信がライブラリレベルで抽象化される。

5.4 抽象化モデル

5.3 節で示した 2 種類のエミュレーションモデルより、それぞれに抽象化モデルを定義する。

5.4.1 SPI レジスタモデル

SPI ハードウェアレベルのエミュレーションを実現する実装モデルを SPI レジスタモデルと呼ぶ。SPI 通信を行う SPI レジスタを模倣するインタフェースを設計・作成し、エミュレータの内部に実装する。アプリケーションプログラムからの視点では作成前と特段変化はないが、エミュレータの視点で見ると、より正確に SPI 通信をエミュレーションすることが可能になる。

5.4.2 Exodus モデル

SPI ライブラリレベルのエミュレーションを実現する実装モデルを Exodus モデルと呼ぶ。ライブラリレベルで SPI 通信を抽象化するために、プロセッサエミュ

レータと SPI ライブラリを拡張する。テスト対象のアプリケーションは SPI 通信のためのライブラリを読み込み、SPI による送受信をする transfer メソッドを使用する。抽象化の実現のために、transfer メソッドが呼び出されたとき、エミュレータを実行する OS の read/write に切り替える必要がある。

5.5 設計と実装

5.5.1 エミュレータ上の SPI インタフェース実装方式

プロセッサエミュレータの SPI ハードウェアの実装方法は、図 5.5 に示す (a)〜(c) に分類できる。(a) が最も忠実度が高く、(c) が最も忠実度が低い。

(a) は高い忠実度で実装し、通信インタフェースを実装する形式である。この実装では SPI のインタフェース同士が通信する。これは抽象化モデルのハードウェアレベルの抽象化に該当するが、コンピュータ内にこの形式で実装することは難しい。一般に SPI インタフェースは次の (b) の方式で実装される。

(b) はインタフェースを抽象化して、忠実度を低くすることで通信の機能を IPC (Inter Process Communication) やソケットインタフェースを通じて通信している。この実装方法は、いずれの抽象化レベルに置いても用いることができる。アプリケーションレベルの抽象化ではアルゴリズム、デバイスの起動や終了などの各種機能トリガ関数の動作を検証するが、この抽象化を実装するためにはプロセッサの機能の抽象化だけでなく、ライブラリの抽象化など実装範囲が広くなる。

(c) は (b) と同様の通信インタフェースを持っているが、検証するアプリケーションがエミュレータを動かすコンピュータ上で動作している。検証したい機能が限定的であるため、必ずしもターゲットデバイスのアーキテクチャで動作させる必要がなく、汎用コンピュータのアーキテクチャで実装することが考えられる。

SPI ハードウェアの抽象化において、同じ実装方式で抽象化できる。本研究では 2 種の抽象化モデルの実装において、ソケットインタフェースを用いた。

5.5.2 Cortex-M0+エミュレータの拡張

本研究で使用するプロセッサエミュレータは、ARM 社の ARMv6-M アーキテクチャ [48] を搭載する e4arduino を用いる。これは Atmel SAMD21 ATSAMD21G18A [25] を模倣するエミュレータである。以下、本論文では Cortex-M0+とする。このエミュレータは Ubuntu Linux 上で動作する。ARM 社は組込み用などの多目的に利用可能なプロセッサの設計を行っており、IoT の組込み分野やモバイルの分野で広く利用されているため、概念実装の対象とした。Cortex-M0+の諸元の一部を表 5.2 に記す。

未拡張のエミュレータモデルと実装する 2 つのエミュレータのモデルについて述べる。

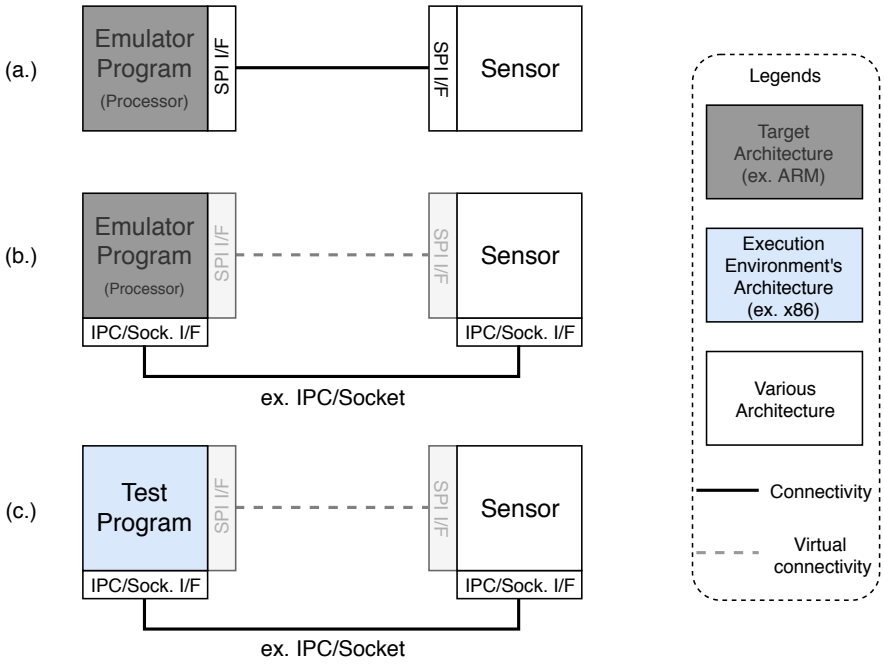

図 5.5: ハードウェア視点のエミュレータ実装形式例

表 5.2: Cortex-M0+の詳細 (抜粋)

項目	詳細
命令セット	ARMv6-M
アーキテクチャ	32bits
ISA サポート	Thumb/Thumb-2 subset

未拡張のエミュレータモデル

本研究で使用する Cortex-M0+エミュレータは、SPI 通信のためのハードウェアを模倣したレジスタに相当する機構を備えていない。SPI 通信に関わる通信をする際、特定のメモリアドレスを指定して read/write することから、メモリマップド I/O 相当の機構を有している。これを模したデータフローを図 5.6 に示す。しかし、プロセッサの命令セットを忠実に再現しており、擬似的な SPI 通信の I/O を提供しているため、レジスタレベルのエミュレーションの中でも、抽象度の高いエミュレーションに分類する。この未拡張のエミュレータモデルを本論文では疑似 SPI I/O モデルと呼ぶ。

SPI レジスタモデルの実装

SPI レジスタモデルを実装するために、SPI レジスタを定義した。レジスタの設計イメージを表 5.3 に示す。設計した SPI レジスタモデルは大きく 3 つのレジスタ

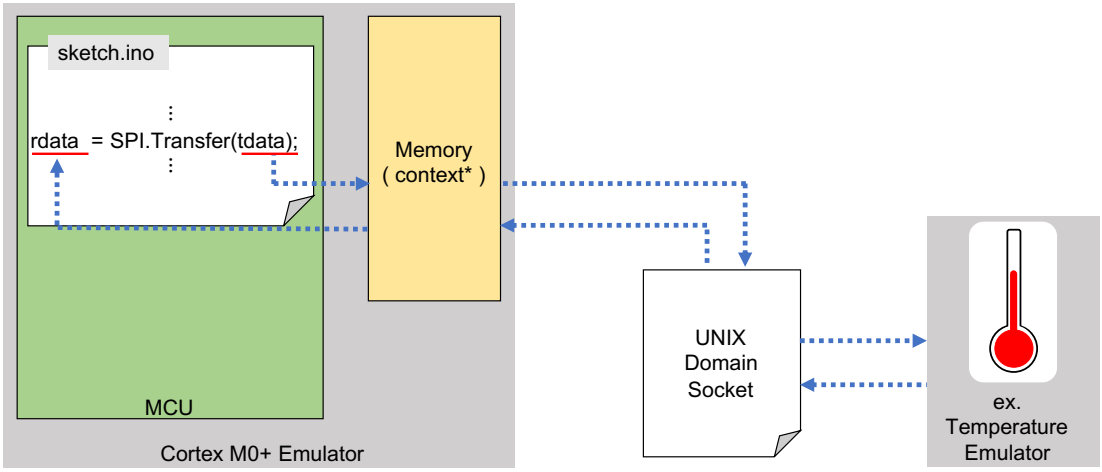

図 5.6: 疑似 SPI I/O モデル

表 5.3: SPI レジスタのモデル

レジスタ名	説明
status	r/w read status
rx_data	received data memory
tx_data	transmitted data memory

を持つ。

1. status レジスタ
   SPI の通信を行う際に、read と write のステータスを管理する。1byte のレジスタで、ステータスをフラグビットで管理している。外部からのデータを受信し終えたとき、rx_data にデータを保存し、read ready フラグがセットされる。アプリケーションによってデータが読み取られたときに read ready フラグが消去される。アプリケーションから SPI 通信を用いてデータを書き込むとき、tx_data にデータが保存されると write ready のフラグがセットされる。エミュレータ上でアプリケーションの送信命令が発行された後、送信される。送信後フラグは消去される。

2. rx_data レジスタ
   外部から受信したデータを格納するレジスタ。キューにより管理される。一種のバッファと考えられる。

3. tx_data レジスタ
   外部に送信したいデータを格納するレジスタ。キューにより管理される。同様に一種のバッファと考えられる。

図 5.7 で示すように、SPI レジスタのモデルを Cortex-M0+エミュレータに搭載

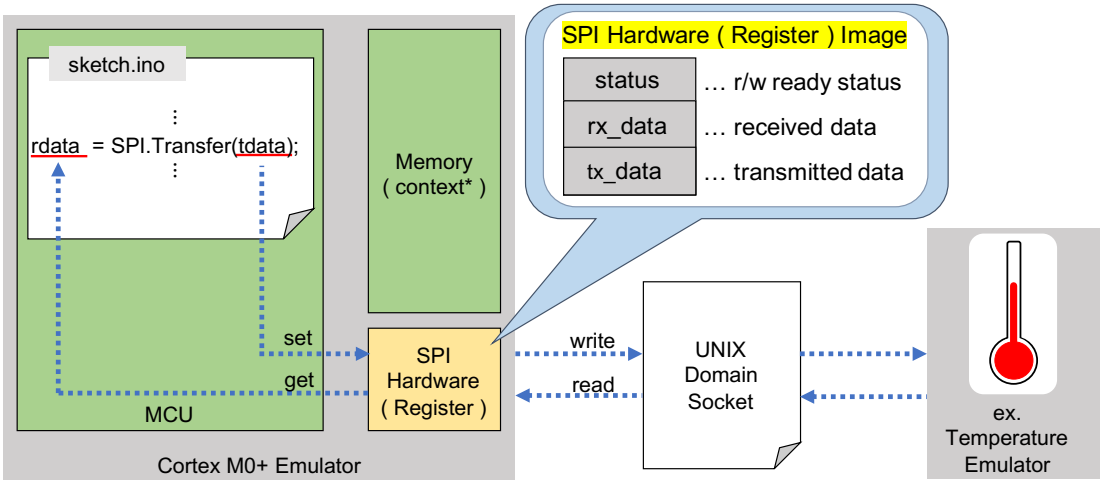

図 5.7: SPI レジスタモデル

する。図中の「sketck.ino」はこのエミュレータ上で動作するプログラムのソースファイルである。SPI を使用したデータ通信の流れを破線で示している。送信したいデータの tdata は SPI レジスタの tx_data にセットされる。Cortex-M0+のプロセッサ命令に照らし合わせると、str (store register) 命令に該当する。送信するデータが SPI レジスタにセットされた後、status レジスタの write ready ビットが 1 になる。str 命令の処理を終了したとき、エミュレータが動作する OS の write システムコールにより、UNIX Domain Socket を通じて対向側のエミュレータ (あるいはシミュレータ) にファイルディスクリプタを経由してデータを送信する。その後、write ready ビットが 0 となって次のプログラムカウンタにセットされている命令が実行される。

受信したいデータの rdata は SPI レジスタの rx_data から取得する。Cortex-M0+のプロセッサ命令に照らし合わせると、ldr (load register) 命令に該当する。データはファイルディスクリプタ経由で受信される。プロセッサエミュレータ上では POLL し、受信するべきデータが有る場合、read システムコールによって、SPI レジスタ上にセットされる。受信が終了した場合、status レジスタの read ready ビットが 1 となる。ldr 命令が実行されたとき、SPI レジスタ上からデータを読み取り、read ready ビットは 0 になる。その後、次の命令が実行される。

以上が SPI レジスタモデルによる SPI 通信の送受信の流れである。これらの実装により、メモリマップド I/O 相当であった SPI 通信の機構が、ポートマップド I/O へ切り替わったことになる。

Exodus モデルの実装

Exodus モデルを実装するため、プロセッサエミュレータに新しい命令 exd (表 5.4) を搭載した。exd 命令はアプリケーション上で SPI 通信の transfer メソッドが呼び

表 5.4: exd 命令の仕様

構文	exd {#imm,} rX
役割	プロセッサが使用するファイルディスクリプタより値を読み書きする
項目	説明
#imm	exd 命令が呼び出すシステムコールの種類を指定 (通常指定なし)
rX	送信データが格納されるレジスタの指定 受信データはこのレジスタに格納される

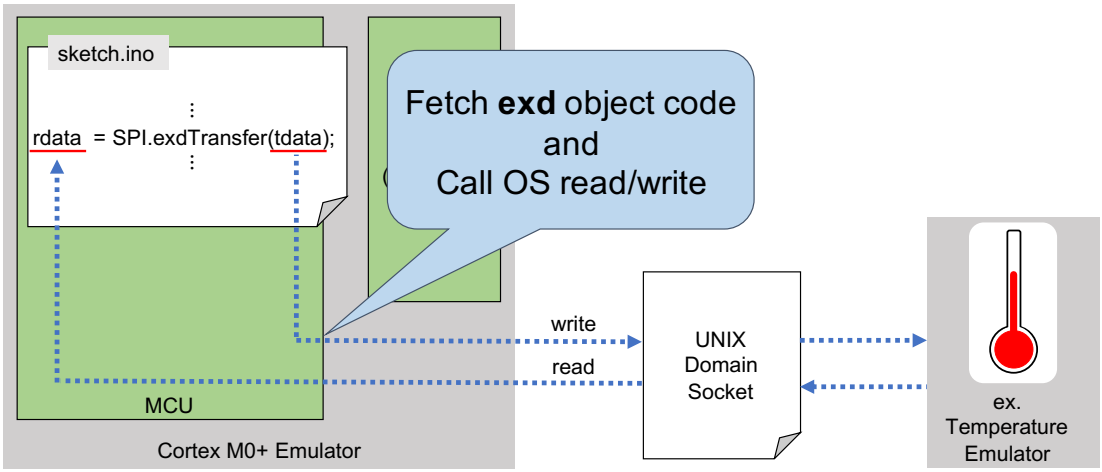

図 5.8: Exodus モデル

出されたとき、送受信したいデータを直接 OS の read/write で処理する。これにより、プロセッサエミュレータ内では SPI 通信をシステムコールやレジスタを使用せず一足飛びに実行することが可能となる。

図 5.9 では Exodus モデルの構築のフローを示している。送信したいデータを tdata、受信するデータ (メソッドの戻り値) は rdata に格納される。sketch.ino はアプリケーションのソースコードである。SPI.h では exd 命令を実行して SPI 通信を行う exdTransfer の関数プロトタイプと、実際にデータを送受信する writeTransfer と readTransfer メソッドの関数プロトタイプの宣言を SPI クラスに記述している。この実装では通常の SPI 通信を行う transfer メソッドと、exd 命令を使用するメソッドを明確に分離するため、exdTransfer を定義している。SPI.cpp では SPI.h で定義されたクラスの処理を記述している。図中の readTransfer を例にすると、exd 命令を使用する部分に対して C 言語の asm 関数を用いて任意のアセンブラ命令を記述する。

これらの sketch.ino や SPI.h などのライブラリを含めてコンパイルを実行すると中間生成物として elf 形式のファイルが得られる。これに対して、asm 関数で記述した該当箇所に対して、バイナリエディタを用いて exd 命令を記述する。

その後、ARM 用のバイナリコンバータである arm-none-eabi-objcopy を用いる

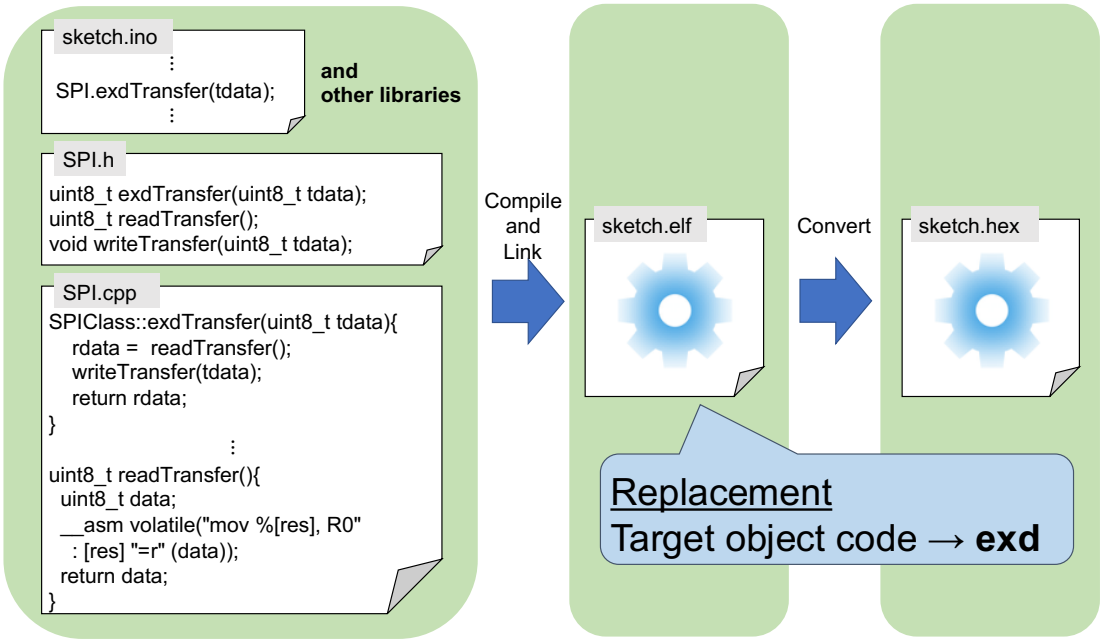

図 5.9: Exodus モデルの構築フロー

表 5.5: エミュレーションモデル

モデル名	抽象化レベル	説明
疑似 I/O	ハードウェア	未拡張 Cortex-M0+エミュレータ SPI の I/O 機能についてレジスタモデル をともなわない疑似 I/O を持つ
SPI レジスタ	ハードウェア	SPI を模倣するレジスタモデルを持つ
Exodus	ライブラリ	SPI の通信をライブラリの関数から エミュレータ外に処理を切り替える

ことで、Cortex-M0+が直接解釈可能な実行形式に変換する。

以上の手続きを経ることで、検証用のアプリケーションプログラムを作成する。また、このままでは Cortex-M0+エミュレータが exd 命令のニーモニックをデコードできないため、エミュレータのソースコードに exd 命令を処理する記述をする。

本節で定義・設計したエミュレーションのモデルについて、表 5.5 でまとめる。

5.5.3 対向側エミュレータ ADT7310 の実装

ADT7310 エミュレータ

Analog Devices 社製の温度センサである ADT7310 を本実験のために使用した。ADT7310 は SPI 通信を使用して温度データの送受信をする。このセンサのエミュ

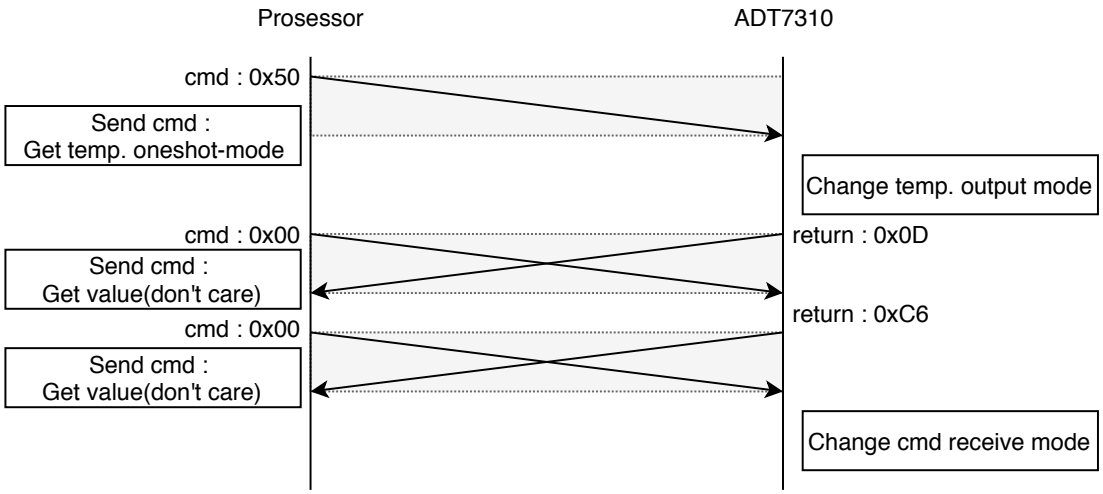

図 5.10: ADT7310 の振る舞いの例

レータを開発した。ソースコードは Github の adt7310_emulator[29] を参照されたい。Cortex-M0+エミュレータから利用できるエミュレータを設計したため、UNIX Domain Socket を経由して温度情報の通信をする。

このセンサから温度情報を取得するフローを図 5.10 に示す。図中の「cmd」は 16 進数のコマンド、「temp.」は Temperature を意味する。SPI 通信は送信と受信の双方向同時に実行する。網掛け部分は SPI 通信の送受信の 1 セットを表している。Cortex-M0+のエミュレータから温度取得に関するコマンド 0x50 (oneshot-mode : 一度だけ温度を取得するコマンド) を受け取ると、データ送出モードに切り替わり、プロセッサエミュレータから送信にかかわるコマンド 0x00 (解釈されないコマンド) を受信することで温度情報を送出する。温度データは 2byte に分かれており、プロセッサは ADT7310 から温度データを受信し、プログラム処理によって温度値を取得する。

コマンド体系については公式レファレンス [30] を参照されたい。

第6章　実験と評価

本章では、実験と評価について述べる。実験では未拡張のエミュレータと、実装した 2 種類のエミュレータの実証実験と性能評価を行う。性能評価は、実装した機能が正しく機能しているか評価し、また IoT デバイスのエミュレーションにおけるスケーラビリティに関わる諸項目について評価する。

6.1　実験環境

実験環境には、NICT の StarBED Group P のノードを使用した。性能等の諸元は表 6.1 に記す。また使用したソフトウェア環境に関しては、表 6.2 に記す。

組込み開発ボードの Arduino[16] シリーズには、Cortex-M0+のプロセッサを使用した Arduino M0 Pro が販売されている。6.6 節の実験において、比較のために Arduino M0 Pro を用いる。また Arduino M0 Pro は Cortex-M0+の ATSAMD21G18A を採用している。M0 Pro は Arduino の開発環境に対応しており、各種ライブラリをインストールすることで利用できる。よって、Cortex-M0+エミュレータで動かすプログラムを Arduino 開発環境で作成する。表 6.3 では開発環境について記す。

表 6.1: StarBED Group P のハードウェア諸元

ハードウェア構成項目	詳細
MODEL	DELL PowerEdge R430
CHIPSET	Intel C610
CPU	Intel Xeon E5-2683 v4 (2.1GHz/16core) x 2
MEMORY	32GB RDIMM (DDR4-2400/Multi-bit ECC) x 12 = 384GB
HDD	1.2TB 2.5in HDD (SAS/10K RPM) x 1
SSD	1.6TB 2.5in SSD (SATA/Read Intensive) x 1

表 6.2: 実験のソフトウェア環境

ソフトウェア項目	詳細
OS	Ubuntu16.04 LTS 64bits
Compiler	gcc version 5.4.0 20160609 (Ubuntu 5.4.0-6ubuntu1 16.04.10)

表 6.3: 開発環境

項目	詳細
PC	Microsoft Surface
OS	Windows10 64bits
IDE	Arduino IDE 1.8.1
Library	Arduino SAMD Boards (32-bits ARM Cortex-M0+) Version 1.6.11

6.2 実験のシナリオ

本実験のシナリオとして、図 6.1 の構成を構築した。Cortex-M0+エミュレータから ADT7310 エミュレータに対して温度情報の取得コマンドを送信し、温度データを受信する。この操作は 2 度繰り返される。その後、正しい値の送受信ができているか確認するために、ADT7310 の内部に格納されるセンサの T_{HIGH} Setpoint (実行中に不変の値) を取得し、アプリケーション中で比較することで状態チェックをする。これらの操作を 1 秒毎に繰り返すアプリケーションを用意する。アプリケーションの処理フローについて図 6.2 に示す。

開発したアプリケーションについては付録 B.2 を参照されたい。

6.3 凡例

グラフラベルの凡例を表 6.4 に示す。特に断りがない限り、以下の凡例にしたがう。単位の秒は s、ミリは m、マイクロは μ として、図中に示している。

pspio (Pseudo SPI I/O : 疑似 SPI I/O モデル) は拡張前の Cortex-M0+エミュレータである。spireg は SPI レジスタモデル、exodus は Exodus モデルを示している。actual は実機の Arduino ボードによる実験結果を示している。

6.4 実行時間の測定

エミュレータの実行時間を測定する実験を行う。

5 章では、2 種類のエミュレータの実装を設計した。エミュレータの抽象化により、SPI 通信によるデータ送受信の抽象化を実現し、エミュレータの計算要求リ

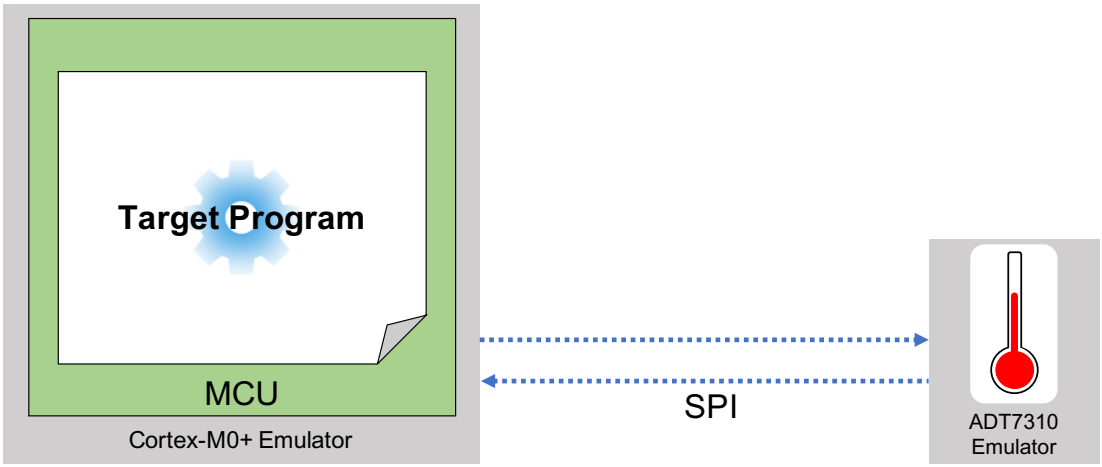

図 6.1: 実験のシナリオ

表 6.4: グラフラベルと凡例一覧

凡例	実行の種別	説明
pspio	エミュレータ	疑似 SPI I/O モデル
spireg	エミュレータ	SPI レジスタモデル
exodus	エミュレータ	Exodus モデル
actual	実機ボード	Arduino M0 Pro

ソースが削減されることが目的であった。そこで、エミュレータの起動、エミュレーションの実行、終了までの一連の時間を測る。エミュレータで動作するプログラムの命令数を一定の量に固定することで、要求リソースの削減を実行時間を指標にして表す。

6.4.1 実験方法

エミュレータの起動から終了までの時間を Linux の time コマンドを使用して測定する。また、エミュレータが実行する命令数を 1000 万個ステップに制限している。

time コマンドから取得できる情報を表 6.5 に示している。この実験では 100 回繰り返し、その平均値を出している。

表 6.5: time コマンド出力の例

パラメータ	説明
real	プログラムの呼び出しから終了までにかかった実時間
user	プログラム自体の処理時間 (カーネルの処理時間を含まない)
sys	プログラムを処理するためにカーネルが処理をした時間

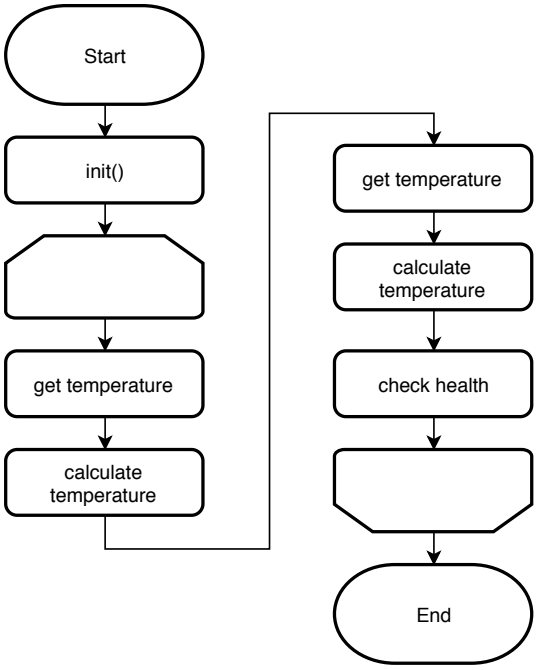

図 6.2: アプリケーションの処理フロー

6.4.2 結果

実行結果を図 6.3 に示す。縦軸は実行時間、横軸は time コマンドの指標順にエミュレータモデルを列挙した。real 指標に注目すると、pspio と spireg との計算時間の差は、pspio モデルに SPI レジスタを模倣するレジスタモデルを実装しているため処理が増える。したがって順当な結果であると言える。

spireg と exsodus では実行時間に 1.30 秒の差が出た。実行時間ベースで 9.95% の差が出ている。これは Exodus モデルのエミュレーション要求リソースが SPI レジスタモデルの要求リソースよりも少なかったことを示している。

pspio と exodus の結果は 0.27 秒 (2.3%) の差であり、わずかに exodus モデルが大きかった。これらについて、他の指標から理由を考察する。pspio と exodus の数値を比較すると、user 指標では exodus が 0.25 秒 (2.8%) 遅く、sys 指標では exodus が 0.02 秒 (0.007%) 遅かった。user 指標の結果は、カーネルの処理時間を含まないため、exodus のエミュレータの処理が大きかったことを示している。sys 指標の結果は、SPI の通信処理をする OS の read/write のシステムコールの処理がほとんど同等の速度であったことを示している。ここで、pspio と exodus のエミュレータで動作するアプリケーションプログラムは同じ処理をするプログラムである。通常の SPI 通信をする transfer メソッドを exd 命令を使用する exdTransfer メソッドに置き換えたものであった。つまり、プログラム中で SPI 通信をする回数は同じであるため、エミュレータの最もプリミティブな通信処理である read/write を行っている時間は同等であるため矛盾はない。そして、exodus ではエミュレータに独

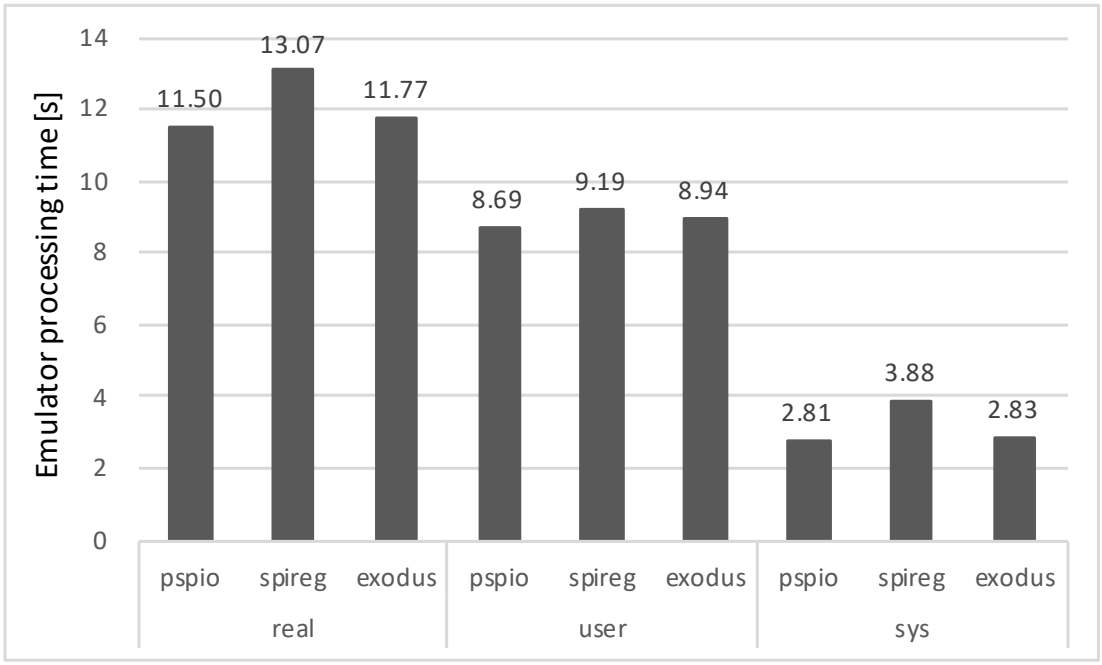

図 6.3: エミュレータの起動から終了までの実行時間

自設計の exd 命令処理を追加したため、エミュレータの処理が増加しことが考えられる。したがって、exodus は pipsio より read/write の処理を除く exd 命令の処理のオーバーヘッドが大きいと考えられる。

以上から、この実験はより抽象度を低くした spireg より高い抽象度で実装した exodus の計算時間が短くなる結果となり、期待通りの結果を得ることができた。

6.5 エミュレータ初期化時間の測定

組込みデバイスにおける初期化処理は負荷が大きい処理である。ROM からプログラムをロードし、メモリ内に転送する処理は、プロセッサの処理の中でも特に遅いためである。エミュレーションモデル間の実装の差異が初期化に与える影響を計測する。また、6.4 節のエミュレーション実行時間の中で、どれだけの時間が初期化処理に当てられていたか検証する。

6.5.1 実験方法

エミュレータの初期化時間を調べるために、初期化完了後にエミュレーションを強制的に終了し、その間の実行時間を time コマンドによって計測する。今回開発した Cortex-M0+用のアプリケーションプログラムは図 6.4 で示す処理を行う。setup 関数の実行後、loop 関数が実行される。したがって setup 関数実行後、loop

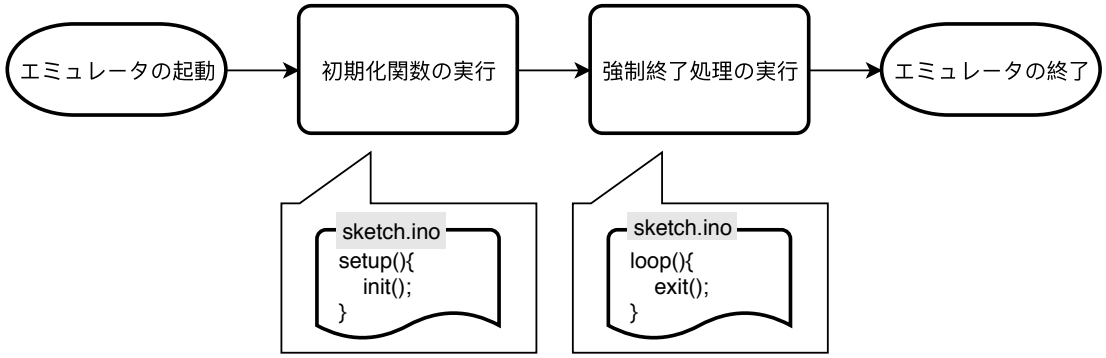

図 6.4: エミュレータの処理フロー

関数の開始直後でアプリケーションを終了する処理を行うことで実現する。アプリケーションの実行バイナリを RAM 上にコピーする時間をなるべく同一に保つため、同じアプリケーションサイズにした。

Cortex-M0+の ARMv6-M 命令セットや Arduino の機能として、アプリケーションを強制終了する、あるいはプロセッサのシャットダウンをする関数や命令は存在しない。したがって、エミュレータ上でのみ実行可能な強制終了命令 ext を実装した。この実験は 10 回行い、その平均を算出している。

6.5.2 結果

実行結果を図 6.5 に示す。縦軸をエミュレータの実行時間 (ms)、横軸にはエミュレータのモデルを列挙した。

いずれも 6ms 強の初期化時間であり、エミュレーションモデル間の差は小さいことがわかる。

図 6.6 では 1000 万命令の実行時間中、初期化にかかる時間がどの程度の割合を占めるかを示したグラフである。実行時間が 11 秒強であり、初期化にかかる時間が 6ms 程度と非常に小さな値であったため、初期化による時間差がエミュレータ間の全体の実行時間差を生み出す原因とは言えない。したがって、6.4 節で実験したエミュレーション実行時間で現れた時間差は、メインの loop の実行によって生み出された処理の時間差であったと言える。

6.6 SPI 通信における I/O 処理にかかる時間測定

本研究では SPI 通信の抽象化を行ったため、SPI 通信における I/O 処理にかかる時間を測定する。Cortex-M0+の実機ボード上に搭載されている SPI ハードウェアの処理時間と実装したエミュレータの SPI 通信の処理時間を計測し・評価する。

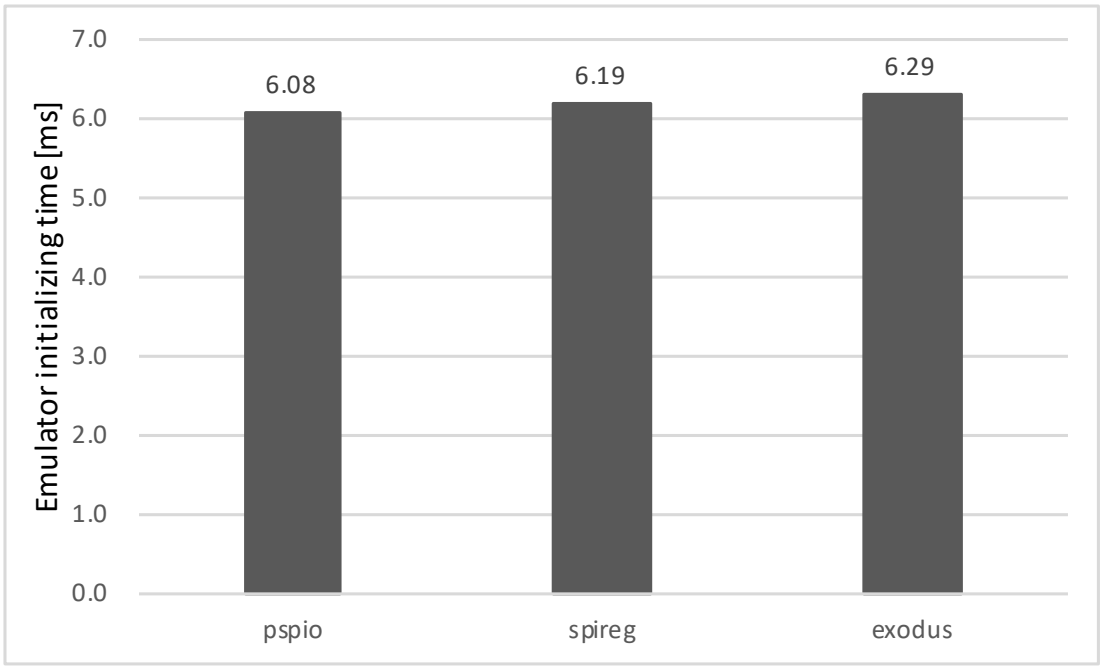

図 6.5: アプリケーションの初期化処理にかかる時間実行時間

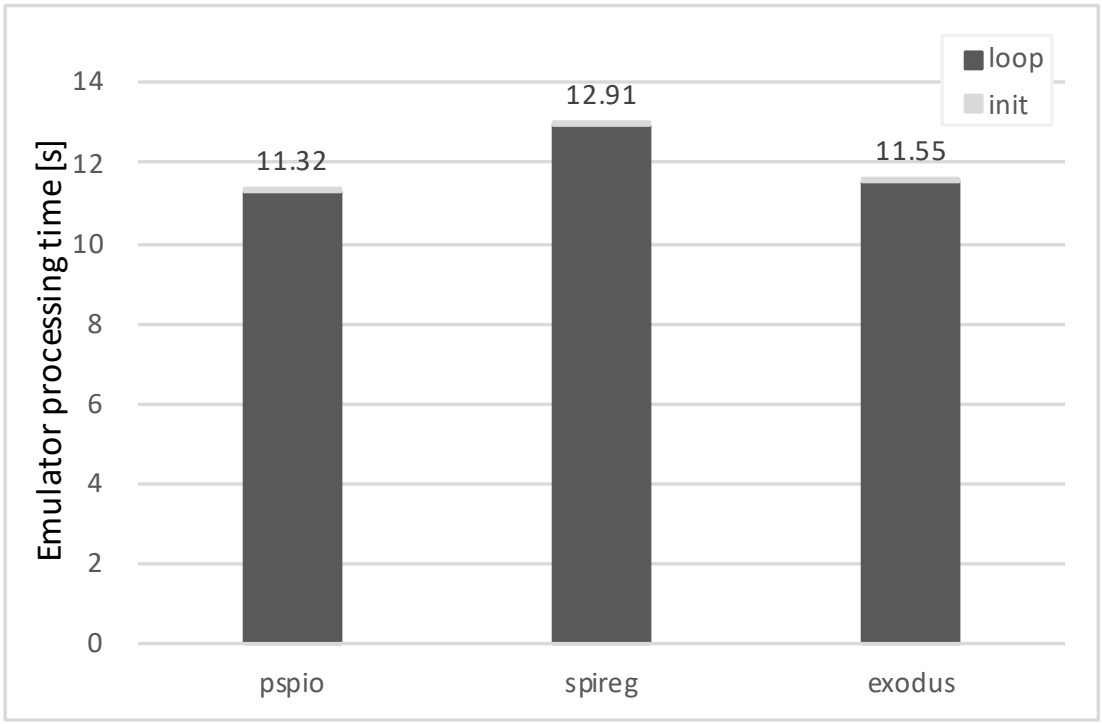

図 6.6: アプリケーションの初期化処理が 10M 命令の実行時間に占める割合

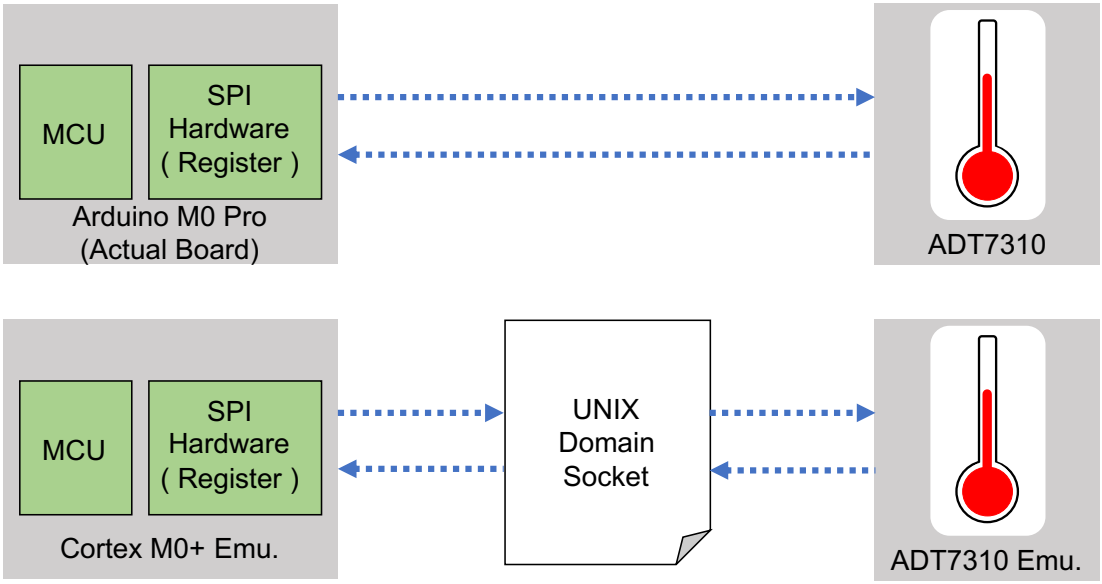

図 6.7: SPI 通信の処理時間の測定対象

6.6.1 実験方法

実機の実験方法

本実験では実機の Arduino M0 Pro (以下 Arduino) を使用して SPI 通信の処理時間を計測し、エミュレータと比較する。SPI 通信の処理時間とは、図 6.7 の上で示す様に、SPI 通信命令の開始から対向側の通信相手にデータを送信し、通信相手が結果を返信するまでの往復時間であり、read/write 処理時間の合計値である。

Arduino の SPI 通信の計測方法は、Arduino の開発環境の Arduino IDE に SAMD ライブラリをインストールして行う。計測のために、SAMD ライブラリの SPI クラスに対し、処理時間を計測する関数を追記する。Arduino の SPI 通信は SPI ライブラリの transfer メソッドを利用する。このとき、SAMD のライブラリでは SPI 通信処理を transferDataSPI メソッドを通して使用する。このメソッドは Arduino の SPI ハードウェアである SEARCOM 社製のチップに対してデータを読み書きするメソッドである。これは Arduino の SPI 通信を行うプログラムインタフェースであるため、この処理を対象に、前後で時間を計測して出力する。計測は、Arduino の標準ライブラリに定義されている micros 関数を用いる。micros 関数は Arduino ボードがアプリケーションプログラムの実行開始から現在までの時間をマイクロ秒単位で取得する関数である。計測の分解能は、Arduino の動作周波数を元に、micros 関数のソースコード [32] を参考にして、1 マイクロ秒単位である。この実験では SPI 通信 300 回分の平均を算出した。

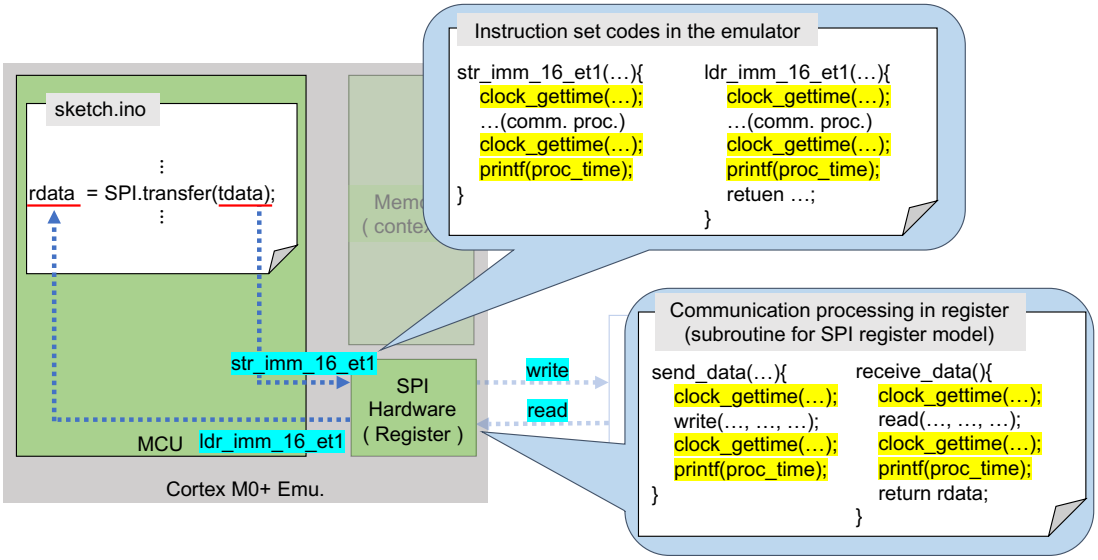

図 6.8: SPI 通信に関する処理時間の計測方法

エミュレータの実験方法

エミュレータにおける SPI 通信処理時間の計測は、SPI 通信を行うために呼び出される Cortex-M0+エミュレータ内の関数の処理時間の計測によって行う。Cortex-M0+エミュレータは、アプリケーションから SPI 通信を行う際に特定のメモリアドレスへのアクセスをともなう ldr 命令と str 命令を使用する (図 6.8)。また SPI レジスタモデルにはこれらの命令に加えて実際の通信を行うサブルーチンがある。これらの命令に対して、エミュレータを実行するコンピュータにおいて、C 言語の clock_gettime 関数を用いて計測を行う。

これに対して、1000 万ステップの命令を実行したときに含まれている SPI 通信のうち、1 回あたりの往復時間を平均して算出している。

6.6.2 結果

計測結果を図 6.9 に示す。縦軸を SPI 通信の処理時間 (マイクロ秒) として、横軸に左から Arduino (actual)、エミュレータ各種 (pspio/spireg/exodus) と列挙した。

エミュレータで行われた SPI 通信 1000 万回のうち、SPI に関する read (ldr 命令) は 106260 回、SPI に関する write (str 命令) は 53130 回の通信処理があった。これは、1 回の SPI 通信の処理では ldr 命令が 2 回、str 命令が 1 回行われるからである。

actual では 2 マイクロ秒程度の処理時間で read と write の処理時間の差は少なかった。

pspio では、read 処理 0.72 マイクロ秒、write 処理が 1.48 マイクロ秒であった。

spireg では命令レベル (ldr/str) の処理は Instruction level time で示す処理時間であった。read 処理が 1.95 マイクロ秒、write 処理が 1.58 マイクロ秒であった。

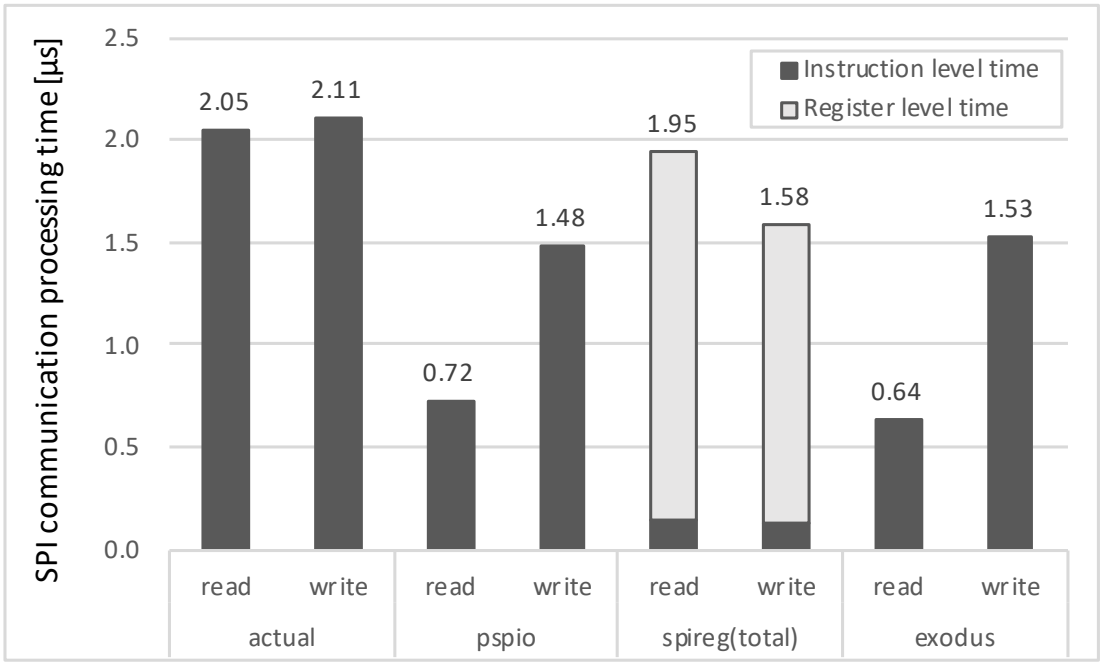

図 6.9: SPI 通信に関する処理時間

SPI 通信は対向側の通信相手がいるため、処理時間が通信相手に依存する。しかし、SPI レジスタモデルではエミュレータの内部に搭載される SPI レジスタの実装したモデルの実態に対してデータを読み書きするため、非常に高速に処理される。実際の SPI 通信処理にかかる時間を測定するためには、SPI レジスタモデルの実装エンティティが通信処理を行う部分を合計して得られる。その際の処理時間はグラフの Register level time で示す時間である。Instruction level time と Register level time の合計時間は read で 1.95 マイクロ秒、write で 1.58 マイクロ秒であった。exodus では、read 処理 0.64 マイクロ秒、write 処理が 1.53 マイクロ秒であった。

図 6.9 で read/write で示した値を合計した値が実際の SPI 通信の処理時間である。これを図 6.10 に示す。軸のスケールとラベルは図 6.9 と同様である。

actual の SPI 通信時間は 4.16 マイクロ秒であった。これはエミュレータの SPI 通信実行時間と比較して大きかった。

pspio のエミュレータの処理時間は 2.20 マイクロ秒、spireg モデルは 3.53 マイクロ秒、exodus は 2.17 マイクロ秒であった。この処理時間の傾向は、実験 6.4 のエミュレータの処理実行時間を計測した際の傾向と同じであるが、pspio と exodus の処理時間が実験 6.4 と比較して実行時間の傾向が逆転している。これは計測誤差の範囲であると考えられる。

以上より、抽象化する手法により、SPI 通信処理の計算時間を削減をしている。本実験では時間を指標として評価したが、エミュレーションの計算要求リソースに関しても同じく削減可能であると考えられる。

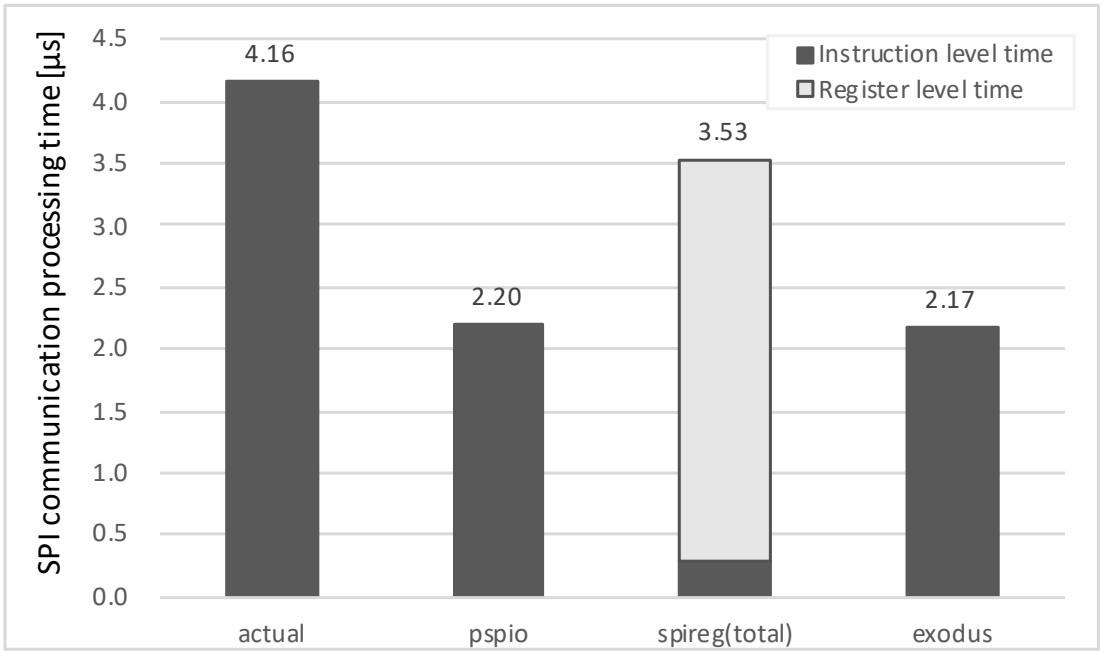

図 6.10: SPI 通信に関する処理時間 (合計)

アプリケーション中に含まれる SPI 通信処理量に関する考察

アプリケーション中における SPI 通信の割合とエミュレータの抽象化の関係を考察する。SPI 通信量の割合を求める方法として 3 種類ある。1 つ目はコードレベルで記述される SPI 通信の割合である。プログラム中に記述される関数などの行数から SPI 通信の割合を示す方法であるが、それぞれの関数中に含まれる処理にかかる時間が行数と直接的に相関を持たないため適切に影響を考察できない。

2 つ目はアセンブラレベルで記述される SPI 通信関係の処理の割合を調べる方法である。CPU や MCU は高級言語などで記述されたソースコードをオブジェクトコード (バイナリ) に変換した後に一行ずつ実行する。したがってアプリケーションのアセンブラコード行数を分析することで、アプリケーション全体の処理に含まれる SPI 通信処理の量を 1 つ目と比較してより実態に即した方法で定量的に示すことができる。この手法による解析では、アセンブラコード中のメインループ 1 回において、SPI 通信処理行数の割合を求めることで算出し近似できる。このとき初期化関数などは、繰り返し実行されるメインループと比較して、繰り返し操作が行われない非常に小さな処理であるため無視する。またエミュレータで実行するアプリケーションのバイナリは pspio と spireg は共用、exodus は 5.5.2 節で示した方法で構築しているため命令行数が異なる。結果を表 6.6 に示す。

pspio/spireg 共用のバイナリでは、全アセンブラコード行数に占める SPI 通信処理の命令行数は 7.5%であった。exd 命令を用いた exodus 用バイナリでは 3.9%であった。6.4 節で示した spireg と exodus の処理時間差が 9.95%であったことと 6.6 節で示した処理時間差が 38.5%であったことに関して、これらの結果からは原因を

表 6.6: オブジェクトコード中に含まれる SPI 通信処理の割合

バイナリの種類	全体の命令行数	SPI 通信処理の命令行数	SPI 通信処理の割合
pspio/spireg 共用	1679	126	7.5%
exodus 用	1617	63	3.9%

表 6.7: SPI 通信処理がエミュレーション全体に含まれる時間の詳細

エミュレータ	エミュレータ全体の処理時間 (E)	SPI 通信の処理時間 (S)	エミュレータの処理時間 (Y)	SPI 通信処理の割合 ($\frac{S}{E} * 100$)
pspio	11.55s	0.16s	11.39s	1.39%
spireg	13.07s	0.16s	12.91s	1.22%
exodus	11.77s	0.11s	11.66s	0.93%

説明できない。spireg では SPI 通信の割合がアプリケーション中の 7.5%であったことに対して、このうちの SPI 通信の処理時間を 38.5%削減してもエミュレータ全体の実行時間が 9.95%削減できないためである。したがってエミュレータの処理時間の削減を示した方法と同じ実行時間をもとにして割合を算出するべきである。

3 つ目の手法として、6.4 節や 6.6 節の実験結果を用いて、実行時間をベースに SPI 通信処理の割合を算出する。6.4 節で実験した際に算出したエミュレータの処理時間から、図 6.10 の平均の SPI 処理時間を算出する前の SPI 処理時間の総和の差を求められる。S は SPI 通信処理時間の総和、R は 1 回の read 処理の時間、W は 1 回の write 処理の時間、Y は求める SPI 通信処理を除いたエミュレータの処理時間、E はそれぞれのエミュレータの起動から終了までの処理時間として、これらの関係式は式 6.1 と式 6.2 で与えられる。

$$S = \sum_{i=1}^{n} R_i + \sum_{j=1}^{m} W_j \tag{6.1}$$

$$Y = E - S \tag{6.2}$$

n は read 関数が呼び出された回数で 106260 回、m は write 関数が呼び出された回数で 53130 回である。それぞれの計算結果を表 6.7 に示す。

いずれの割合に対しても、1%前後の処理時間であったことがわかる。しかしこの結果からも 2 つ目の手法と同様に、6.4 節で示した spireg と exodus の処理時間差や 6.6 節で示した処理時間差の原因を説明できない。したがって、本実験で求めた SPI 通信の処理時間の測定した処理以外の部分によって全体の処理時間が削減されていると考えられる。またアプリケーション中における SPI 通信処理の割合が小さいとしても、エミュレータの機能の抽象化によって全体の処理時間を大きく削減することがわかる。

表 6.8: ps コマンドのメモリ指標の一覧

パラメータ名	パラメータ名非省略形	詳細
VSS	**Virtual Set Size**	仮想的なメモリ使用量
RSS	**Resident Set Size**	物理メモリの消費量
PSS	**Proportional Set Size**	プロセスが実質的に所有するメモリ
USS	**Unique Set Size**	ひとつのプロセスが占有するメモリ

6.7 使用メモリの計測

エミュレーション実行におけるメモリ使用量を調査する。IoT デバイスエミュレーションのスケーラビリティを阻害する要因に、メモリ使用量が問題になる可能性がある。一般に大量のエミュレータを実行するにはそれ相応のメモリ使用が考えられるため、エミュレーションの数に比例した量のメモリを消費する。したがって、メモリの使用量を計測することで、IoT デバイスエミュレーションを行うためのプラットフォームの選定に影響があるか調査する。

6.7.1 実験方法

メモリの使用量を調査するために、Linux の ps コマンドを使用した。ps コマンドは実行中のプロセスの一覧を取得するコマンドであるが、u オプションを使用することでメモリ使用量を調べられる。本実験で用いるエミュレータはプロセスとして実行される。このとき表示されるメモリ指標は RSS (Resident Set Size) であり、プロセスが使用中の物理メモリの量である。参考のため、表 6.8 にメモリの指標をまとめる。

またこの測定では 32 個のエミュレーションを同時に実行し、60 秒間のメモリ使用量の推移を取得した。

6.7.2 結果

図 6.11 に示すように、Cortex-M0+エミュレータは最初の 60 秒間にメモリの使用が上昇し、その後一定の値に収束した。これは、OS によるメモリのページング処理の結果であると考えられる。

1 エミュレーションあたりのメモリの最大使用量を収束時のメモリ使用量をもとに表 6.9 に示す。

pspio と spireg の差は、0.01MB(0.45%) であったため誤差と考えられる。pspio や spireg と exodus の差は、0.06MB(2.70%) であった。exodus ではアプリケーション中において SPI 通信に関する一部のライブラリのメソッドが不使用となったため、呼び出されなかったメソッド分の差がメモリ使用量に現れていると考えられ

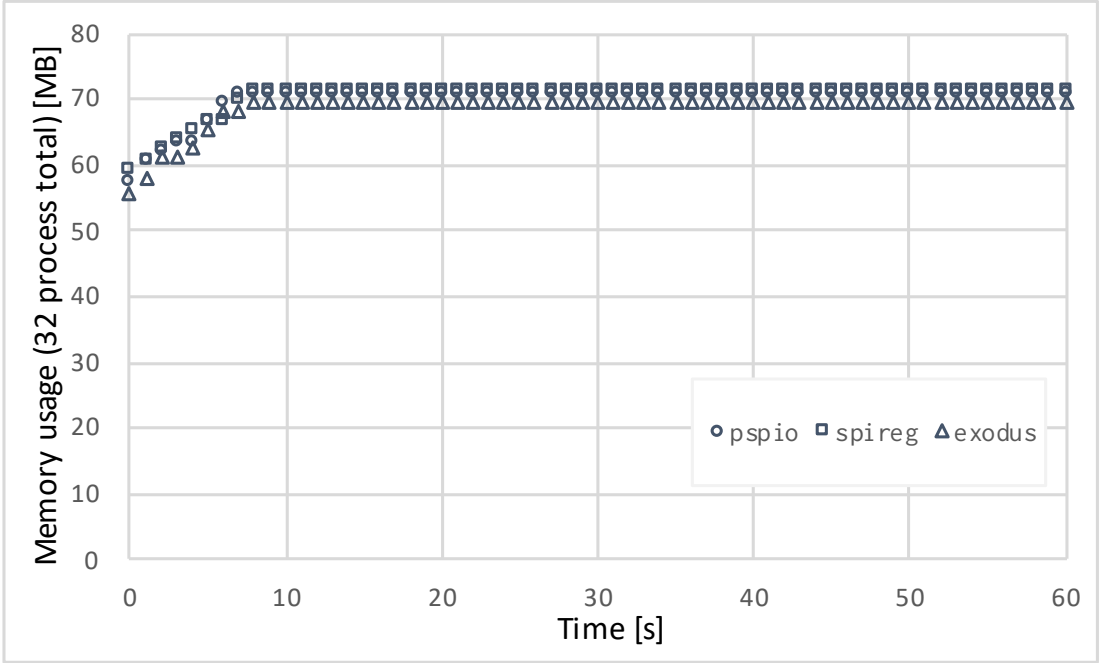

図 6.11: Cortex-M0+エミュレータのメモリ使用量の収束 (32 エミュレーション/60 秒間)

表 6.9: Cortex-M0+のメモリ使用量 (32 エミュレーション/60 秒間)

エミュレータ	メモリ平均使用量
pspio	2.22MB
spireg	2.23MB
exodus	2.17MB

る。これらの結果から、Cortex-M0+エミュレータで使用するメモリ使用量は 1 エミュレーションあたり 2.2MB 程度であった。これは 2019 年現在の計算機リソースとしては十分に小さい値であると考えられるため、IoT デバイスエミュレーション実行において、大量のデバイスの同時エミュレーションでも十分に展開可能である。

次に図 6.12 に示すように、対向側の ADT7310 エミュレータのメモリ使用量の時間変化をグラフ化した。60 秒の中で、徐々に上昇するが、ある一定のところでメモリの使用量が収束する。こちらもメモリの最大使用量が収束したときの値をもとに、表 6.10 に示す。

3 種類間のメモリ使用量にほとんど差は認められなかった。Cortex-M0+エミュレータは実装の違いがあるが ADT7310 エミュレータに実装の違いは無いため、妥当な結果である。以上から、ADT7310 エミュレータが使用するメモリ使用量は 1 エミュレーションあたり 1.4MB 程度である。こちらも十分に小さい値であると考

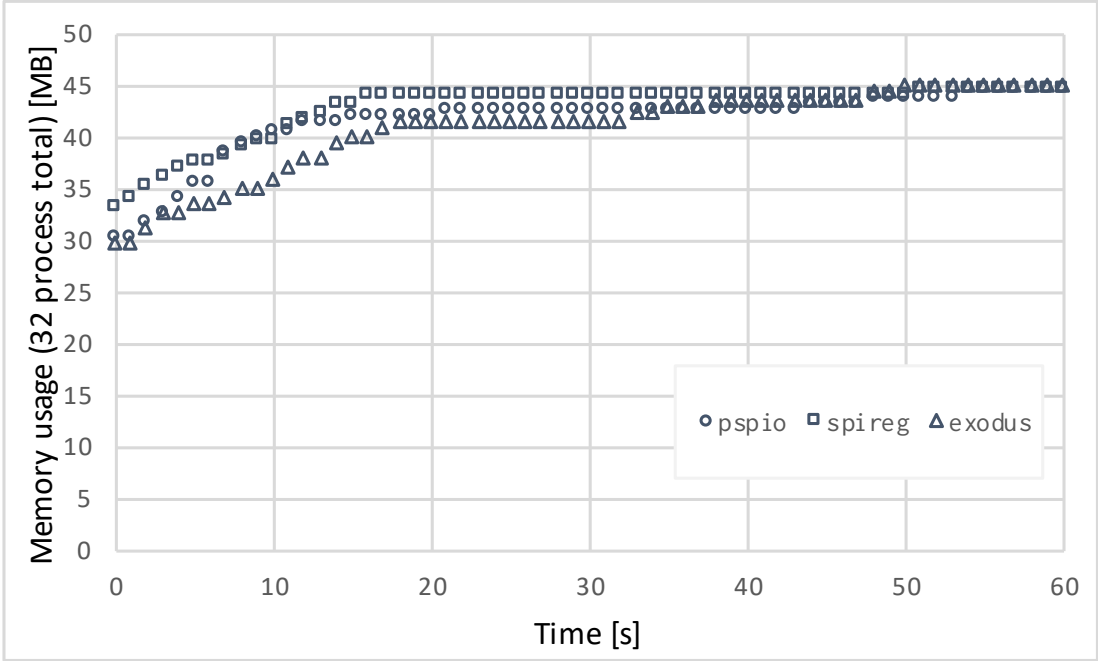

図 6.12: ADT7310 エミュレータのメモリ使用量の収束 (32 エミュレーション/60 秒間)

表 6.10: ADT7310 のメモリ使用量 (32 エミュレーション/60 秒間)

エミュレータ	メモリ平均使用量
pspio	1.40MB
spireg	1.40MB
exodus	1.41MB

えられるため、IoT デバイスエミュレーション実行において、大量のデバイスの同時エミュレーションする際において十分に展開可能である。

本研究で使用した Cortex-M0+エミュレータは、プログラムコードが対象デバイスのアーキテクチャ用に変換されている。つまり、実際のデバイスで動作可能なアプリケーションの実行可能形式をエミュレータが直接解釈する。この様なモデルを Lv らの論文 [46] ではバイナリ変換方式 (Binary Translation) としている。この方式の他に、インタプリタ方式 (Interpretation) が示されている。インタプリタ方式は、実行可能形式の命令列を逐次変換して、ターゲットデバイスのアーキテクチャで実行する。この方式を採用した、インタプリタ形式の命令セットシミュレータ (ISS : Instruction Set Simulator) [47] について、インタプリタ方式を (a)、バイナリ変換方式を (b) として、それぞれの処理フローを図 6.13 (文献 [46] より参照) に示す。

本研究で使用したバイナリ変換方式は、インタプリタ方式と比較してメモリ使

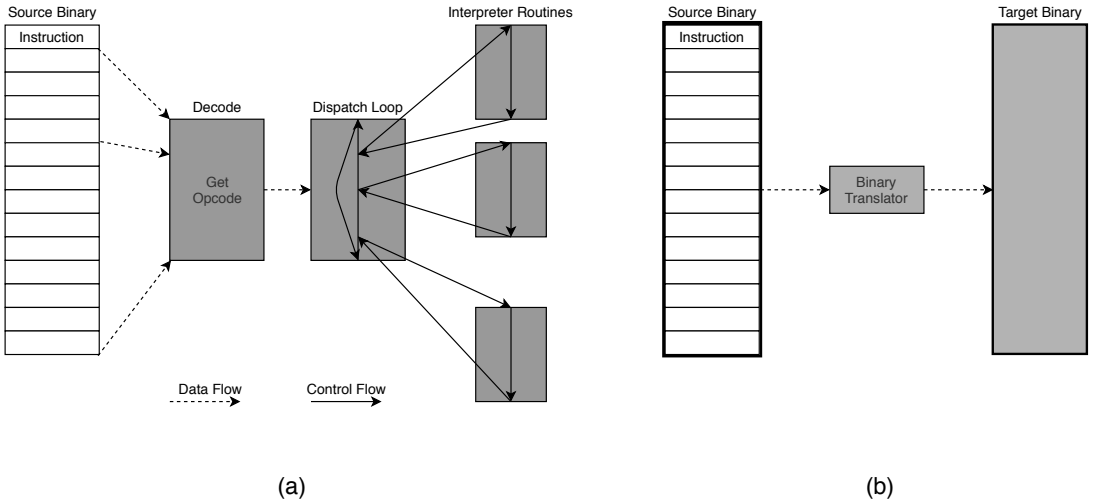

図 6.13: Emulation flow of basic interpretation and binary translation

用量が小さくなるとされている。したがって、本研究ではメモリ使用量が問題とならなかったが、エミュレータの命令変換・解釈方式によってはこの限りではない。

6.8 全体の評価

以上の実験結果より、IoT デバイスや組込み用途で使用される SPI 通信のエミュレーション抽象化を Cortex-M0+エミュレータを用いて評価した。1 つ目に Cortex-M0+エミュレータに SPI レジスタを定義し、より実際の SPI 通信に近いエミュレータを実装した。2 つ目に独自実装の命令を用いることで SPI 通信をライブラリレベルで抽象化した。抽象化した実装方式によって、エミュレータ実行における計算時間が削減され、エミュレーションにおける要求リソースの削減が可能であることを示した。

以上の実験結果より、エミュレータの抽象化が IoT デバイスエミュレーションにおけるスケーラビリティ改善に貢献することを示した。

第7章　議論

本章では、エミュレーションの要求リソース、忠実度、IoT デバイスエミュレーションがシステム検証においてどの様な影響を与えるかといった観点から議論する。

7.1　エミュレーションの要求リソース削減の効果と応用

7.1.1　CPU あたりのエミュレータ展開数のスケーラビリティに関する考察

3.3 節では、エミュレータがソフトウェアであることから、IoT デバイスエミュレーションに必要なスケーラビリティの確保がしやすいことに言及した。しかしプロセッサエミュレータを用いるエミュレーションでは実行オーバーヘッドが大きいため、スケーラビリティを改善するためには計算要求リソースの削減をする必要がある。本研究の実験では、SPI 通信機能の抽象化がエミュレーションの計算要求リソースを削減することを示した。また、複数の機能を抽象化すれば、エミュレータ全体の要求リソースがさらに削減可能であると考えられる。加えて、要求リソースの削減により、CPU あたりのエミュレータ実行数を増やすことが可能であると考えられる。市街地や圃場のセンシングで利用が想定されるセンサ IoT デバイスのエミュレーションでは、一度に大量のエミュレータを動作させなくてはならない。一台のコンピュータあたりに実行できるエミュレータの数が増えれば、より少ないコンピューティングリソースで IoT デバイスエミュレーションを実行可能である。

7.1.2　プログラム中に占める I/O 処理の量についての考察

今回の実験では、SPI インタフェースを用いて制御する温度センサを使用した。しかし、温度センサは時間あたりの測定頻度が低いため、一般的に一時間に数度程度である。組込みシステムはデータ通信や機器制御を含む I/O が主たる処理であるため、より現実に即した IoT デバイスを想定した実験では、抽象化しないエミュレータと抽象化したエミュレータによる計算要求リソースの変化の影響が大きく現れる。

7.1.3 I/O 処理時間がプログラムの動作に影響する場合の考察

対向側のセンサやアクチュエータが中央のプロセッサとのやり取りの中で、処理にタイムアウトを設定するものがある。この様なアプリケーションでは、I/O の処理にかかる時間やプロセッサの処理時間が長くなるとタイムアウトを起こす可能性がある。エミュレーションのオーバーヘッドによる処理時間の増大がタイムアウトを引き起こす場合、エミュレーションの抽象化によって計算時間が削減されればタイムアウトが解消される可能性がある。一方で、これらの時間に依存する処理は、エミュレータの実装による部分が大きいため、正しく検証するためには実装方式についてよく検討する必要がある。

7.1.4 クロックと実時間実行性

3.3.3 節では、エミュレーション手法を用いることで意図したシナリオを実験することが可能であると言及した。コンピュータ上で IoT デバイスノードや関係する IT システムまで再現することで、全体のシステム検証が可能となる。システム全体を検証するためには IoT デバイスが実時間の処理を行い、データを送出・収集・蓄積・分析する必要がある。これをエミュレーション環境を用いて実現するためには、プロセッサエミュレータにおける実時間実行性を実現しなくてはならない。

一方で、プロセッサエミュレータを使用することは、実機のアプリケーションをテストすることと比較してオーバーヘッドが大きい。これはハードウェアでプログラムの命令を処理する実機ハードウェアと、ソフトウェアによって命令を処理するエミュレータの構造的問題である。広くコンピュータで使用される CPU の動作クロック周波数は、GHz のオーダで動作している。また、IoT など組込み用途で使用されるプロセッサの動作周波数は KHz オーダから MHz オーダの動作クロックが主流である。IoT のエンドノードのプロセッサは今後高性能化が進むと考えられるが、センシングノードに限れば省資源の計算リソースであることは大きく変わらないと想定される。これらのことから、汎用サーバやワークステーションに搭載される CPU 上で動作する IoT デバイスエミュレータは、IoT デバイス用のプロセッサと比較して大きくクロック周波数が異なる。実機と同じクロックを持つ IoT デバイスエミュレーションを実現するために、エミュレータの実装方法を変更することにより、エミュレータを擬似的に IoT のプロセッサと同等の周波数で動作させることが可能である。しかし、実時間で実行するためには、次のクロックサイクルまでに前の命令が実行終了になってなければいけないため、オーバーヘッドの大きいエミュレータでは問題になりうる。オーバーヘッドの削減について、エミュレーションの抽象化が貢献し得ると考えるが、本研究では検証していない。

7.1.5　エミュレータの抽象化手法の適用性

実験ではプロセッサとセンサの間で用いるSPIを抽象化した。本研究で提案した抽象化手法によるエミュレータの要求計算リソースの削減はSPI以外に適用が可能である。特に通信インタフェースに関しては同じ手法を用いる事ができるため、例としてI2C、UART、USBやEthernetの抽象化が可能である。センサIoTデバイスのプロセッサは、データの取得と送信をするための処理が主であるためI/Oの処理が多い。したがって通信インタフェースの抽象化により、エミュレータ上のアプリケーション実行の処理を削減することができるため、エミュレータ実行時間における影響は大きいと考えられる。

一方でエミュレータの通信インタフェース以外の抽象化は、同じ手法を用いて実現することが難しい。本研究ではSPIの抽象化のために、通信したいデータの処理をエミュレータが動作するOSの処理に切り替えることで抽象化した。しかし、通信インタフェース以外のエミュレータの処理は、エミュレータの内部でデータを保持して処理するため同じ手法での抽象化はできない。したがってエミュレータの通信インタフェース以外の機能を抽象化するためには、本研究で使用した手法でない方式が必要となる。

7.2　忠実度の検討

コンピュータエミュレーションとは、システムやデバイスを忠実にソフトウェアで実装することにより、その機能を模倣し、コンピュータ内で再現することである。3.4.2節では、エミュレータの実装には少なからず忠実度の低下があることを言及した。エミュレータの忠実度に関してさまざまな観点があり、議論のために定義する必要がある。本研究では、IoTデバイスのためのソフトウェア検証を対象としている。IoTデバイスで利用するデバイスのアプリケーションと同様のアプリケーションがコンピュータ上で検証可能であることが重要であった。したがって本研究における忠実度の定義は、実機と同じアプリケーションが動作可能なエミュレータを忠実度が高いと定義する。より実機の機構と近い実装をなされたエミュレータがより忠実度の高いものとする。

5.2.1節で示した図5.2の抽象化モデルと表5.1の抽象化モデル間の性質をもとに、エミュレータの機能を抽象化することによって忠実度がどの様に下がるか検討する。本研究ではIoTデバイスのアプリケーション開発のためのエミュレーションに関して言及する。

- ハードウェア(レジスタ)レベル抽象化
  プロセッサの命令セットを模倣して実装するため、実機デバイスと同じアプリケーションを使用できる。したがって、最も忠実度が高い抽象化レベルで

ある。しかし、プロセッサの動作を模倣する一方で、物理的な現象を模倣していない。コンピュータはデジタルデータを扱うため、アナログの特徴を持つデータやインタフェースの再現は難しい。データのやり取りをするインタフェースやバスは電気信号で行われているが、その意味でデバイスを模倣していない。もっとも、アプリケーションテストの目的において、この様な再現を必要とすることは少ないと考えられる。しかし、周辺機器やI/Oについて実装方法により忠実度が変わる。ハードウェアレベルの抽象化では、実行速度も重要な観点である。プロセッサエミュレータが実機のクロック周波数で実行されることは、IoTシステム全体をエミュレーションすることにおいて重要な要素である。実機のプロセッサと同様の処理時間やタイミングで動作してデータを送受信することにより、IoTシステムの各構成要素との連携を検証することが可能になるためである。しかしプロセッサエミュレータでは処理のオーバーヘッドが発生するため、処理時間が許容範囲に収まるためには機能の抽象化等によって処理が短縮されなければならない。

- システムコール (OS/Kernel) レベル抽象化
  ハードウェアが抽象化されるため、ハードウェアやプロセッサのアーキテクチャに依存しなくなる境界である。このレベルの抽象化では、実装方法によっては実行するアプリケーションは実機と互換性があり、同じものを使用することが可能である。通常、一度作成したプログラムが、別のハードウェア環境で動作を可能にするため、システムコールによってハードウェアを抽象化している。一般に、システムコールによる抽象化はOSが提供している。一方で、組込み用アプリケーションは様々なハードウェア環境で動作することを念頭に置かないで作成されることがある。これは組込みデバイスではOSを用いないものも多く、ハードウェアの抽象化が行われず、デバイス専用のアプリケーションを開発するためである。このようなOSを持たないデバイスのアプリケーション開発は、システムコールレベルで抽象化するのではなく、抽象度を高くするべきである。

- ライブラリレベル抽象化
  システムコール以下で抽象化するため、ハードウェアを操作しない。実装上の概念的なデータの流れは5章の図5.4で示したが、ライブラリレベルの抽象化ではエミュレータの処理が、実行するコンピュータ上での処理に置き換わる。このレベルの抽象化でも、実機とアプリケーションを同じものにしてテスト可能である。本研究ではSPIのインタフェースを抽象化したが、インタフェースとそれに付随するプロトコルが必要となるテストの場合は適していない。

- アプリケーションレベル抽象化
  アプリケーションレベルの抽象化は、ライブラリ以下を抽象化する。ライブ

ラリはセンサモジュールや通信モジュールを利用するためのメソッドや関数を内包している。アルゴリズムやプロトコルの検証、IoT デバイスの起動、終了、データを送受信するなどの機能を持つ関数のインタフェースを用意することで、IoT デバイスのアプリケーションテストを行う。そのため通信モジュールやセンサモジュールと連携するために、インタフェースの実装や、エミュレータの拡張が必要になる。また、これらの機能のテストのためにエミュレータを用いない検証方法を選択可能で、x86 などのコンピュータのアーキテクチャを用いることで、テスト対象アーキテクチャを用いずにアプリケーション検証が可能である。したがって、実装方法によって実機とエミュレータの間で同じアプリケーションを用いることができない。対象のアプリケーションは、テストに期待するアプリケーションとしてエミュレーションされない可能性があり、エミュレーションの性質がシミュレーションに近くなる。

以上のことから、抽象化はエミュレータの一部あるいは全てに対して行われる。抽象化のレベルを高くするにつれて、エミュレータで検証するアプリケーションの実行アーキテクチャはエミュレータのアーキテクチャから実行しているコンピュータのアーキテクチャに近づく。これはエミュレーションの性質がシミュレーションに近づいていくことと同義である。

7.3 IoT デバイスエミュレーションの検証開始までにおける開発規模と開発増加量についての考察

3.4.1 節では、エミュレーションを使用した検証手法に関して、IoT デバイス開発者の開発量の増加が欠点となることに言及した。本節では、IoT デバイス開発にエミュレーションを用いた検証がもたらす影響について開発プロセスモデル等をもとに考察する。エミュレーションを用いるテストでは、開発者にエミュレータの開発が新たなタスクとして発生する。そこで、エミュレータ利用の検討や開発がどの段階で行われるべきか、IoT デバイスのソフトウェア開発プロセスがどうあるべきか、コード開発量や検証に関してソフトウェア開発プロセスを議論する。

7.3.1 テストベッドによる開発サイクル

センサ持つ IoT デバイスのテストのために、デバイスを設置し、ネットワークやデバイスなどを再現してテストを構築する。実環境を用いる方式とテストをコンピュータ内で実行する方式がある。前者は設置コストが高く、後者は低い。したがってコンピュータを用いることで広域配置、再設置等に関する様々なコストの削減が期待できる。この様な観点から、特にコンピュータを用いたテストベッドに関して議論する。

街を利用した広域実証基盤テストベッド

市街地や自然環境をテスト環境として用いる形式のテストベッドは国内外に存在するが、テストベッドのための専用の用地を用いるため、本来設置したい実地とは異なる場所である。したがって、真に様々な設置環境をテストするためには設置したい場所でテストする必要がある。加えて、実地テストはデバイス設置や修正後の再設置が大きな負担となる。このようなテストは、ソフトウェアやハードウェアの開発が終了し、ハードウェアの調達が完了した後、実地展開直前に行うべきでり、最終的なテストにとどめなくてはならない。

コンピュータベースのテストベッド

NICT で研究開発されている StarBED は、大規模なコンピュータクラスタを有したネットワークテストベッドである。StarBED は汎用のコンピュータを用いているため、Linux 等の OS を用いて検証環境を構築できる。NICT では、ネットワーク R&D のプロセスイノベーション [38] のために、テストベッドがどの様な役割を持つべきかを目的として掲げている。これによると、従来の開発プロセスが「研究」・「技術開発」・「製品開発」・「展開・運用」のライフサイクルを持つとしている。また、テストベッドはこのライフサイクルの各フェーズにおいて、検証の役割を持つべきだとしている。図 7.1 で示すように、テストベッドによる検証により、実験支援や製品化促進などをサポートすることが期待されている。

したがって本節では、IoT デバイスエミュレーションを用いたソフトウェア検証を、テストベッドの R&D サイクルに適用して、これらがどの様な役割を持つか考察する。

IoT デバイスの研究段階では技術要素の研究を行う。その研究対象として、デバイスの制御のためのアルゴリズム、制御や通信のためのプロトコルなどが挙げられる。また IoT を含むシステムとして考える場合、収集システムや分析方法、制御や結果の応用が考えられる。このようなフェーズでは、テストベッドを用いた大規模な検証に限らず、1 つのデバイスで検証する、あるいは論理検証のツールを使用することになる。

技術開発では、研究から生まれた技術シーズに対して、製品に必要な技術を開発・施策する段階である。IoT デバイスのハードウェアを試作する、またアプリケーションの開発を始めとして製品コンセプトモデルの実証実験や開発を行う。このフェーズでは、テスト検証のためにシミュレータの利用や、実装物をテストベッド上で動作させて検証する。

製品開発では、コンセプトモデルから商品化のためにより具体的な検討をする段階である。製品化するためには品質や意図した振る舞いが可能であるか検証する。このフェーズでは設置前の検証となり、設置後の後戻り工程を可能な限り無くすため、テストベッドによる検証をする。開発したハードウェアの検証は実機を用いて検証するが、開発したアプリケーションの検証はテストベッドを用いた

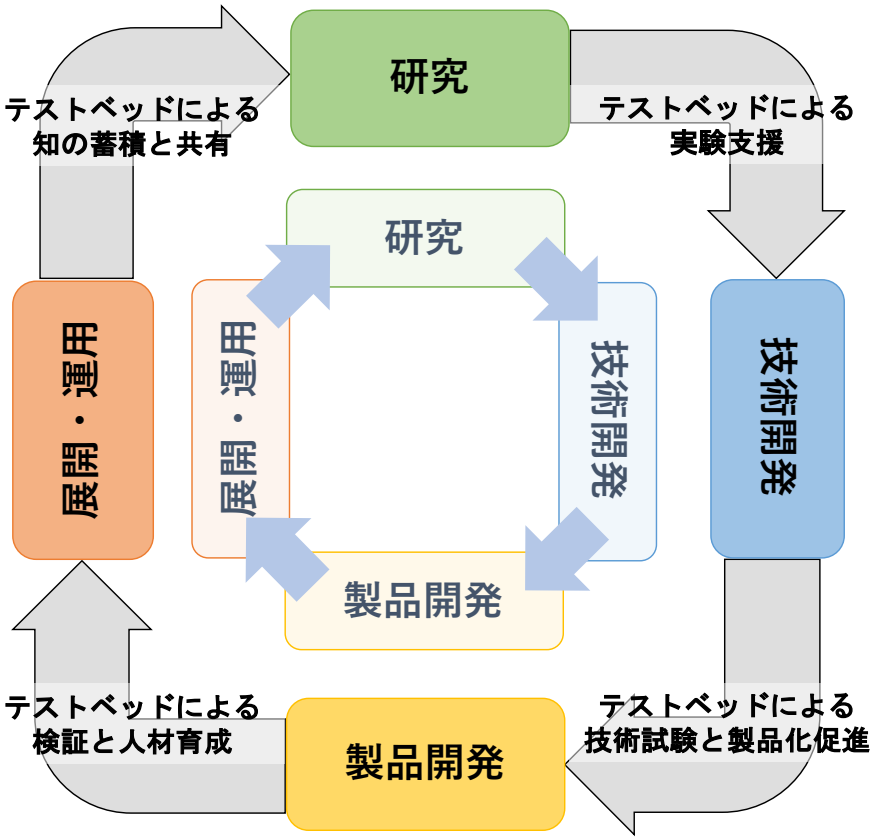

図 7.1: テストベッドにおける R&D サイクル

エミュレーションを利用することで、実機と互換性のある同じアプリケーションを検証することが可能である。またエミュレーションによる検証は、コンピュータ上でデバイスの展開がしやすいため、実際のデバイス数と同じスケール感で実証実験が可能となる。

展開・運用では、リリースしたシステムからのフィードバックより得られる改善点を収集し、情報の共有や次の研究につなげる段階である。当初の要件を満たせなかった要因や、問題点は課題となり、運用によってカバーされ、データを収集することで知見を蓄えることが可能となる。これらを研究によって改善し、次の製品に活かしていくことが可能となる。このフェーズでは、テストベッドが知識集約や新たなテストベッドの研究開発の場として機能する。

7.3.2 IoT デバイス開発における開発プロセス

センサやアクチュエータを搭載する IoT のソフトウェア開発はいわゆる組込みソフトウェア開発と同様である。7.3.1 節ではテストベッドを用いた IoT システムの R&D について議論したが、製品開発におけるプロセスの詳細には V モデル、スパイラルモデルやアジャイル開発などがある。

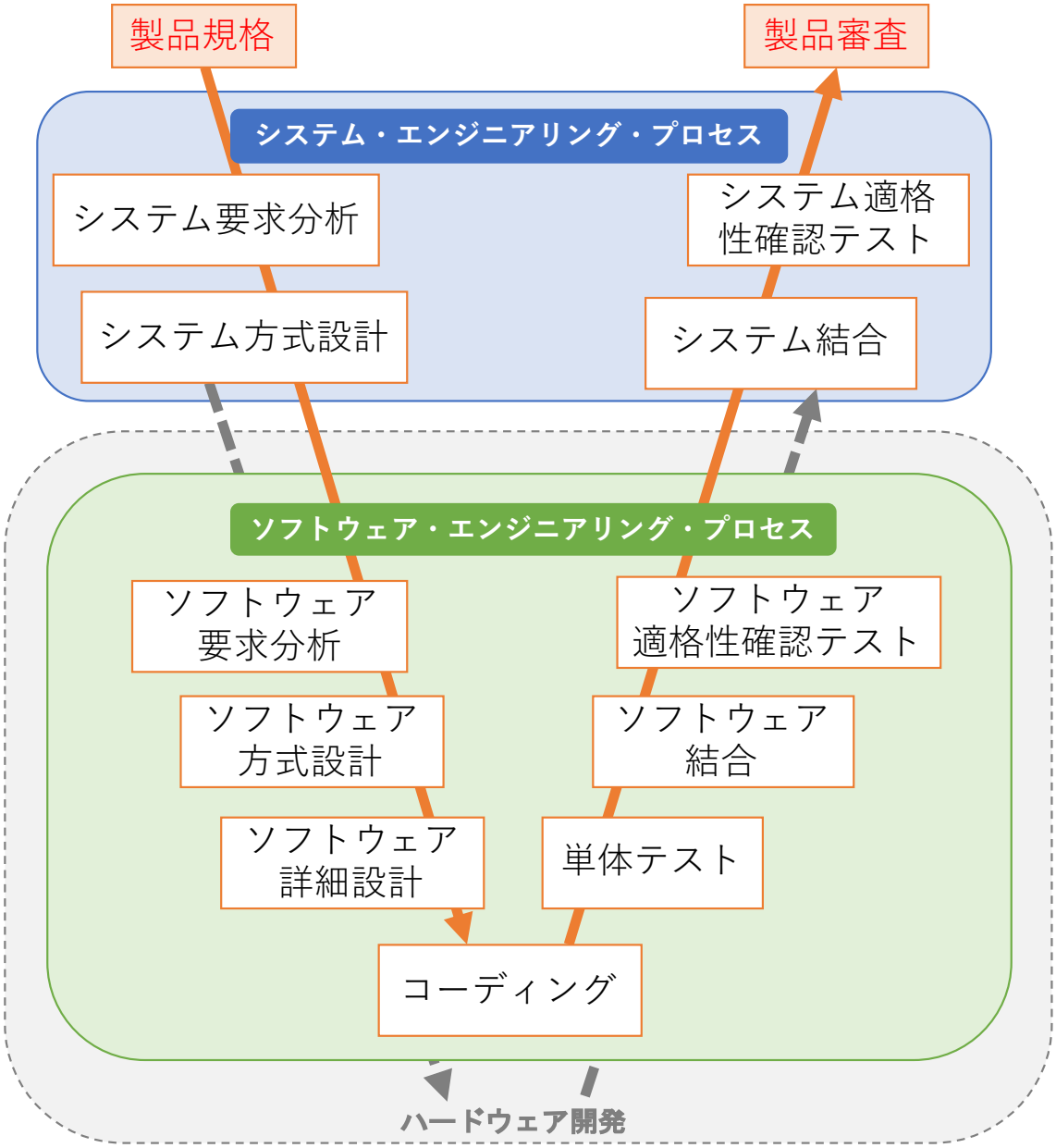

図 7.2: IPA SEC の V モデル

特に V モデルはシステム開発やソフトウェア開発の歴史のなかでも古くから用いられており、現在でも採用されることが多い。組込みソフトウェア開発プロセスでも V モデルを用いて設計開発が行われている。V モデルは使用される現場によって様々な定義が存在する。IPA(情報処理推進機構) ソフトウェア・エンジニアリング・センター (SEC) の組込みソフトウェア開発プロセスガイド (ESPR : Embedded System development Process Reference guide)[37] は、国際規格の ISO/IEC12207、15288 を元に、国内の組込みソフトウェア開発の現場で実践されてきたソフトウェア開発作業 (ソフトウェア開発プロセス) に関する知見を組み合わせて紹介されて

いる。IPA の V モデルにおける組込みシステムとは、1 つ以上の機能を持った単一の機器であり、組込みソフトウェアとは組込みシステムに組込まれる、あるまとまったソフトウェアの単位である。この文献をもとに、ソフトウェア開発モデルに関して V モデルを定義する。

V モデルは製品規格から開発、そしてテストと成果物の審査までの一連の開発プロセスを V 字になぞらえて時系列順に表した開発プロセスである。V 字の左の流れは分析や設計を表し、右の流れはテストを表す。縦のつながりは前のプロセスから次のプロセスを表し、左上から下に進むにつれてより詳細な設計に進む。水平間のつながりは分析・設計に対してテストプロセスと対応しており、下から右上に進むにつれて完成に近づく。

このモデルをもとに IoT デバイスエミュレーションが開発プロセスに与える影響を考察する。

7.3.3 エミュレーション開発を含む IoT 開発プロセスにおける各フェーズについて

センサ IoT デバイスを持つ IoT システムは、2.2.1 節で示したネットワークモデルを採っており、センサを搭載する IoT では一般に Gate Way モデルを採用していることが多い。この様なシステムは図 7.3 で示すような各コンポーネントによって構成される。末端のエンドノードでは、CPU や MCU などのプロセッサを中心に、センサやアクチュエータ、無線通信モジュールを持つ。エンドノードからデータを受け取る Gate Way(基地局) がある。基地局ではプロセッサを中心に、無線通信モジュールや有線通信モジュールを持つ。これらとインターネットを経由して、データを収集するためのデータストアやそのデータを活用して分析、活用をするためのサーバが存在する。

これをもとに、IoT システム開発全体における開発プロセスへの影響と、プログラムコード開発量に関する影響について議論する。

IoT デバイスソフトウェア開発

IoT デバイスのソフトウェア開発は、図 7.1 における、テストベッドによる R&D サイクルの製品開発に含まれる。製品開発は図 7.2 中に示す「システム・エンジニアリング・プロセス」と「ソフトウェア・エンジニアリング・プロセス」に大別できる。また、V モデルにおけるテスト工程は、V 字の右側フローを下から上へ順に行う。

ソフトウェア・エンジニアリング・プロセスでは単体テスト、ソフトウェア結合テストやソフトウェア総合テストを行う。単体テスト、ソフトウェア結合テストでは、開発対象プログラムを機能ごとに分割して開発し、その都度テストを行う。こ

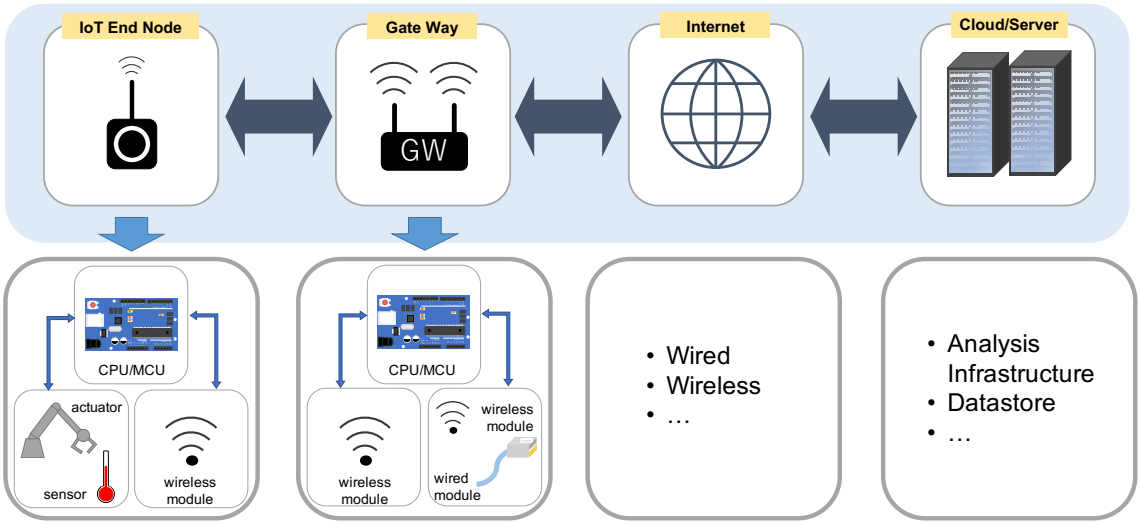

図 7.3: IoT システムの構成物イメージ

こでは実機ハードウェアを用いてテストを行うのではなく、スタブやテストドライバを並行して開発し、テストに望む。したがって実機ハードウェアと切り離した開発であるから、プロセッサエミュレータを利用したテストは適切でない。ソフトウェア総合テストでは、前述の分割開発したプログラム部品を統合し、ソフトウェアとして総合的にテストする。このときに使用するテスト環境は、テスト治具やシミュレータを用いるか、あるいは実機ハードウェアを用意して行う。これによって実装するソフトウェアの開発が完了する。

システム・エンジニアリング・プロセスは、組込みシステムにおいてハードウェアとソフトウェアを含む製品として、要求分析からテストまでを定義している。組込みシステムとは 1 つ以上の機能を持った単一の機器のことであるから、IoT システムにおけるエンドノードデバイスや Gate Way を指す。このテスト群では開発済みのソフトウェアとハードウェアを用いて、システムを構成するコンポーネントを組み合わせた際の個々の機能をテストする。本来のテストでは実機を用いたテストを行うが、IoT デバイスのプロセッサエミュレータと周辺コンポーネントのエミュレータあるいはシミュレータを用意することで、全てコンピュータ内で実行することが可能となる。テスト仕様の策定はそれぞれのテストフェーズにおいて事前に作成するものであるため、IoT デバイスエミュレーションがソフトウェア開発プロセスにおいて影響するのは、システム・エンジニアリング・プロセスのテストフェーズである。

以上の V モデルベースの開発プロセスでは、システム・エンジニアリング・プロセスにおいて、本研究で利用したプロセッサエミュレータを用いたテストを適用することが可能である。ソフトウェア開発の中でエミュレータを用いたテストをするべき利点として、ハードウェアの調達を並行する必要がないことが挙げられる。エミュレータは、後の IoT システム全体の実地展開前において、テストベッ

ド上のテストで利用するから、エミュレータ開発はソフトウェア開発が終わる前の段階で終了していなくてはならない。また本研究で論じているエミュレータの抽象化について、この様な抽象化に関する検討をするべきフェーズは、システム・エンジニアリング・プロセスにおけるシステム要求分析とシステム適格性確認テスト間で考慮する必要がある。これはシステムの要求に対して期待する結果を検証するテストフェーズであるから、機能の一部を抽象化しても必要な要求を満たしているか同時に検証し、システムの性質についてよく考察するべきである。

組込みソフトウェア開発に限らず、サーバシステムも含めたトータルな IoT のシステムも V モデルを用いて開発できる。V モデルは、IPA の ESPR で定義される各々の組込みシステムと、サーバシステムのそれぞれに対して適用できる。プロセッサエミュレータを用いた検証において、抽象化した IoT デバイスエミュレーションが最も有効であるのは、トータルな IoT システムを実地展開前に大規模テストするときである。これは図 7.1 のテストベッドによる R&D サイクルにおける、製品開発と展開・運用の間で行われるテストベッドによる検証が該当する。このとき、エミュレータを用いたテストによって、大量の実機デバイスを用意せずに、IoT デバイスを大規模展開する形式のテストが可能になる。

コードベースの開発量

IoT システムを構成する主要なコンポーネントは図 7.3 で示した。それらについて、コードベースのプログラム開発量についてまとめる。

- CPU/MCU などのプロセッサ
  IoT のエンドノードで利用されるプロセッサは多岐にわたる。制御用のマイコンは非常に種類が多く、市場に出回っている製品が多い。しかし、ARM のように同じアーキテクチャで動作し、互換性を持つものもある。したがって、IoT で利用されるプロセッサの代表的な製品を選択して、あらかじめ開発しておくことによって、エミュレーション開始までの期間を短くできる。また IoT デバイスエミュレーションのために、エミュレータの拡張が必要になる場面があるため、プログラムコードが入手可能なプロセッサエミュレータを利用する必要がある。これらを実現するためには、アーキテクチャ仕様が公開されているもので、自身で開発するあるいはオープンソースのエミュレータを利用する、これに付随して必要なライセンスの取得などが必要である。

- センサ/アクチュエータ
  センサエミュレータの開発では、どの様なデータを取得するかによって開発規模が変わる。温度センサや気圧センサなどの代表的なセンサはあらかじめ用意することが可能であるが、特殊な用途のセンサは都度開発が必要になる。また、データの忠実度や精度がテスト検証に影響を及ぼす可能性がある

ならば、データ生成機構の工夫が必要である。原理的にアナログデータはコンピュータ上で扱うことができず、疑似データの生成にはアルゴリズムの開発が必要となる。真にアナログ的なデータを生成すると、再現性のある実験を作り出しにくくなる。対応策として実機のセンサから取得したデータを蓄積し、再生する形式のモデルであれば忠実度や精度を保つことが可能である。アクチュエータのエミュレータ開発はセンサと比較して難しい。センサでは送出するデータを生成するだけであったが、アクチュエータでは物理的な動作とその結果が必要となる。動作に関するパラメータを入力すると機構が動作するという結果が生成され、さらに制御のためにフィードバックして次の制御をする。コンピュータ内では計算によって、物理環境に変化を与える動作をエミュレーションしなくてはならない。そのため、アクチュエータのエミュレーションでは、入力と出力の関係においてテストしたい範囲を明確に定義し、コンピュータ内で再現できないことがあることを了解しなくてはならない。

- 通信モジュール

  - 有線通信モジュール
    有線の通信モジュールを用いたIoTエミュレーションを想定する場合、Ethernetの様に普及している技術を用いる場合は導入がしやすい。エミュレーションに用いる汎用コンピュータの多くは、Ethernetの規格の通信モジュールを持つため、実行しているコンピュータのOSの機能を用いて仮想の通信デバイスを生成することで、これらの機能を使うことができる。したがってエミュレーションを行うコンピュータとIoTデバイスエミュレータとの間をブリッジする、あるいはバイパスするためのプログラムを開発する必要がある。一方で、Ethernetを用いていても、内部で利用するプロトコルが汎用コンピュータで用いられるものと異なる場合、どの様にプロトコルを実装するべきか検討しなくてはならない。例えば工業用通信プロトコルがLinuxシステム上で利用可能か、移植可能であるかなどを検討し無くてはならない。
  - 無線通信モジュール
    無線の通信モジュールはコンピュータ内のエミュレーションで実現することが難しい。無線は空気を伝搬する電磁波であり、外乱の影響を受けやすく、反射や減衰、回折などの物理現象を起こす。また同時に多数の機器が通信を行うことがあると衝突を起こし、正常に送受信できない。この様な無線通信特有の現象をコンピュータ内部で再現することが無線モジュールでは必要となる。関連研究の4.3節で述べたNETorium など、有線ネットワーク上で無線通信を模倣する研究があるが、IoTで利用が想定されているLPWAなどの規格に関しては今後の研究に期待す

るところである。

以上から、IoT デバイスの開発者は、IoT デバイスで利用が想定されるプロセッサエミュレータ、センサやアクチュエータ、通信モジュールをあらかじめ用意しておくことで並行して開発するコンポーネントを削減可能である。特に CPU や通信の無線に関する検討をどの程度調べるかによって実装負荷が変わる。また、広く利用されるプロセッサやモジュール類は既に開発されている可能性があるため、利用可能であるかを検討することが重要である。

第8章　おわりに

8.1　まとめ

本論文では、抽象化モデルによる抽象化エミュレータの設計・評価と抽象化による忠実度の低下の関係について研究した。

エミュレータの抽象化を実現するために4層の抽象化モデルを定義した。このモデルにもとづき、ハードウェアレベルエミュレータのSPIレジスタモデルとライブラリレベルエミュレータのExodusモデルを設計した。これらのモデルを実装して実験を行い、2種類のモデルと未拡張のエミュレータとの間で比較した。結果として、SPIレジスタモデルより抽象化したExodusモデルの方がエミュレータの実行時間が短くなり、計算要求リソースを削減することが可能であることがわかった。これにより、エミュレータの抽象化の有効性を示した。

また、エミュレータを抽象化することで、忠実度が低下する。忠実度が低くなると、実機と比較して正確な模倣が期待できない。この関係について、4層の抽象化モデルをもとに、層ごとに検証項目が変化することを示した。

この様なIoTデバイスエミュレーションが、IoTシステム開発プロセスのどのフェーズで検討するべきで、実行するべきか明らかにした。

8.2　今後の課題と展望

8.2.1　本研究における課題

本研究ではCortex-M0+エミュレータを利用したが、実験では対象プロセッサのクロックや実時間性について考慮していない。テストベッドなどを利用して大規模なIoTに関わるシステム全体を検証するためには、プロセッサの実時間実行などについて検証するべきである。

8.2.2　今後の展望

IoTデバイスエミュレーションがこの先、より進展するうえで重要な研究について以下にまとめる。

- 通信エミュレータ
  コンピュータを用いたエミュレーションでは、無線環境の再現が難しい。関連研究で紹介したように、コンピュータ内で様々な無線通信をエミュレーションすることが可能となれば、IoT デバイスをすべてコンピュータ内で検証することが可能となるだろう。

- その他のエミュレータおよびシミュレータ
  IoT デバイスエミュレーションは複雑なテスト条件であっても、複数のエミュレータやシミュレータを組み合わせて再現することが可能である。例として、センシングで想定される小型電池によって駆動する IoT デバイスでは、デバイスの処理の実行時間や気温などによる小型電池の状態変化によって持続時間が問題になる。この様な例においても、テストする項目のために様々なシミュレータを開発することでテスト可能となる。

謝辞

本研究を進めるにあたり、多大なご助言と議論をさせていただいた関係者の方々にはこの場をかりて謝意を述べたいと思います。

主指導教員の篠田陽一教授を始め、副指導教員として知念賢一特任准教授、インターンシップ指導教員として丹康雄教授、本研究室の宇多仁助教には示唆に富んだアドバイスを頂きました。

また、国立研究開発法人情報通信研究機構北陸 StarBED 技術センターの研究員の方々には研究におけるアドバイスをいただきました。多くの場面でご助力を頂いた湯村翼氏に感謝いたします。また活発な議論をしてくださった宮地利幸氏、井上朋哉氏、榎本真俊氏、明石邦夫氏に感謝いたします。

WIDE Project の方々にも研究にかかるアドバイスをいただき、非常に有意義な議論を展開していただきました。

本研究室の博士後期課程の太田悟氏、阿部博氏、三浦良介氏に感謝いたします。また先輩の阿波史和氏、可児友邦氏、三木晶司氏、押川侑樹氏、砂川真範氏、橋本光世氏、村上正樹氏に加え、同輩の淺葉祥吾氏、小松源氏、三島航氏、宮崎駿氏、山口礼央氏、また後輩の菅野洋信氏、北沢尭宏氏、廣中颯氏、渡邊司揮氏にはゼミ活動や輪講等において非常に有意義な議論をしていただいたことに感謝いたします。

最後にこれまでの学生生活や私生活を支えてくださった義一氏、祥江氏、智史氏をはじめ、春夫氏、博氏、スマ子氏、俊子氏に感謝いたします。

本研究に関する対外発表

- 広瀬太志, *"IoT* デバイスエミュレーションのための抽象化に関する研究*,"* *WIDE Project* ポスターセッション, Sep. 2018.
- Tsubasa Yumura, Masatoshi Enomoto, Kunio Akashi, Futoshi Hirose, Tomoya Inoue, Satoshi Uda, Toshiyuki Miyachi, Yasuo Tan, Yoichi Shinoda, *"AOBAKO: A Testbed for Context-Aware Applications with Physicalizing Virtual Beacons," The 2018 ACM International Joint Conference on pervasive and ubiquitous computing (UBICOMP2018)*, Oct. 2018. [22]
- 広瀬太志, 湯村翼, 篠田陽一, *"*センサ *IoT* デバイスのエミュレーションの抽象化に関する研究*,"* 知的環境とセンサネットワーク研究会（*ASN*）, 信学技報, vol.118, no.468, pp.49–54, Mar. 2019.

参考文献

[1] “GSMA,” GSMA, https://www.gsma.com/, (accessed: 2018-09-28).

[2] “LoRa Alliance,” LoRa Alliance, https://lora-alliance.org/, (accessed: 2018-09-28).

[3] “Sigfox -The Global Communications Service Provider for the Internet of Things(IoT),” Sigfox, https://www.sigfox.com/en, (accessed: 2018-09-28).

[4] “Sigfox とは,” KCCS, https://www.kccs.co.jp/sigfox/, (accessed: 2018-09-28).

[5] “コア仕様 - Bluetooth Technology Website,” Bluetooth, https://www.bluetooth.com/ja-jp/specifications/bluetooth-core-specification, (accessed: 2018-09-28).

[6] “Wi-SUN Alliance,” Wi-SUM, https://www.wi-sun.org/index.php/ja/, (accessed: 2018-09-28).

[7] “ZigBee Specification. Document 053474r2,” ZigBee, http://www.zigbee.org/wp-content/uploads/2014/11/docs-05-3474-20-0csg-zigbee-specification.pdf, (accessed: 2018-09-28).

[8] 鄭 立, IoT ネットワーク LPWA の基礎 -SIGFOX、LoRa、NB-IoT-, リックテレコム, 2017.

[9] “N. Kushalnagar, *et al.*, “RFC4919 - IPv6 over Low-Power Wireless Personal Area Networks (6LoWPANs): Overview, Assumptions, Problem Statement, and Goals,” IETF, https://tools.ietf.org/html/rfc4919, Aug. 20707.

[10] “T. Winter, Ed., *et al.*, “RFC6550 - RPL: IPv6 Routing Protocol for Low-Power and Lossy Networks,” IETF, https://tools.ietf.org/html/rfc6550, Mar. 2012.

[11] Haihui Gao, *et al.*, “Techniques and Research Trends of Network Testbed,” 2014 Tenth International Conference on Intelligent Information Hiding and Multimedia Signal Processing, Aug. 2014.

[12] “テストベッド分科会／TOP ページ／,” NICT, https://testbed.nict.go.jp/bunkakai/, (accessed: 2018-12-22).

[13] “テストベッド分科会／総合テストベッド活用研究会／,” NICT, https://testbed.nict.go.jp/bunkakai/caravan-tb.html, (accessed: 2018-12-22).

[14] “横須賀市におけるハイブリッド LPWA テストベッドの構築と利用開始について,” NICT, http://www.nict.go.jp/info/topics/2018/03/180309-1.html, (accessed: 2018-12-27).

[15] “IoT 時代の総合的なエミュレーションを実現する StarBED4 プロジェクト,” NICT, http://starbed.nict.go.jp/, (accessed: 2018-09-28).

[16] “Arduino,” Arcuino.cc, https://www.arduino.cc/, (accessed: 2018-09-28).

[17] “ns-3 — a discrete-event network simulator for internet systems,” ns-3, https://www.nsnam.org/, (accessed: 2018-09-28).

[18] Tsubasa Yumura, Kunio Akashi, Tomoya Inoue, “BluMoon: Bluetooth Low Energy Emulation System with Software-Implemented Controller,” 2018 IEEE International Conference on Pervasive Computing and Communications Workshops (PerCom Workshops), Mar. 2018.

[19] 中田潤也, “ユビキタスネットワークシミュレーション環境の構築に関する研究,” 北陸先端科学技術大学院大学, 博士論文, Mar. 2009.

[20] 岩橋 紘司, 井上 朋哉, 篠田 陽一, “Internet of Things を対象とした大規模実証実験環境構築に関する研究,”マルチメディア，分散，協調とモバイル (DICOMO2014) シンポジウム, pp.1258-1263, Jul. 2014.

[21] Kunio Akashi, *et al.*, “NETorium: high-fidelity scalable wireless network emulator,” AINTEC ’16 Proceedings of the 12th Asian Internet Engineering Conference, pp.25–32, Bangkok, Thailand, 2016.

[22] Tsubasa Yumura, Masatoshi Enomoto, Kunio Akashi, Futoshi Hirose, Tomoya Inoue, Satoshi Uda, Toshiyuki Miyachi, Yasuo Tan, Yoichi Shinoda, “AOBAKO: A Testbed for Context-Aware Applications with Physicalizing Virtual Beacons,” The 2018 ACM International Joint Conference on pervasive and ubiquitous computing(UBICOMP2018), Oct. 2018.

[23] “PLANETLAB,” Planetlab, https://www.planet-lab.org/, (accessed: 2018-12-20).

[24] “geni,” Geni, https://www.geni.net/, (accessed: 2018-12-20).

[25] "Cortex™-M0 Revision: r0p0 Technical Reference Manual," ARM, http://infocenter.arm.com/help/topic/com.arm.doc.ddi0432c/DDI0432C_cortex_m0_r0p0_trm.pdf, (accessed: 2018-09-28).

[26] "QualNet," 構造計画研究所, https://network.kke.co.jp/products/qualnet/, (accessed: 2018-09-28).

[27] "The FreeRTOS™ Kernel.," FreeRTOS, https://www.freertos.org/, (accessed: 2018-09-28).

[28] "Contiki: The Open Source OS for the Internet of Things," Contiki, http://www.contiki-os.org/, (accessed: 2018-09-28).

[29] "adt7310_emulator," Github, https://github.com/rosev838/adt7310_emulator, (accessed: 2018-11-28).

[30] "Data Sheet ADT7310," Analog Devices, https://www.analog.com/media/en/technical-documentation/data-sheets/ADT7310.pdf, (accessed: 2018-12-18).

[31] "SAM D21 Family Data Sheet - SAMD21-Family-Data-Sheet-DS40001882D.pdf," Microchip, http://ww1.microchip.com/downloads/en/DeviceDoc/SAMD21-Family-DataSheet-DS40001882D.pdf, (accessed: 2019-02-01).

[32] "Arduino, "arduino/ArduinoCore-samd," Github, https://github.com/arduino/ArduinoCore-samd, (accessed: 2018-12-14).

[33] Sarah Harris, David Harris, "Digital Design and Computer Architecture: ARM Edition," Morgan Kaufmann, 2015.

[34] Jooyoung Seo, Ahyoung Sung, Byoungju Choi, Sungbong Kang, "Automating Embedded Software Testing on an Emulated Target Board," AST '07 Proceedings of the Second International Workshop on Automation of Software Test, p.9, 2007.

[35] A.A. Jerraya, W. Wolf, "Hardware/software interface codesign for embedded systems," IEEE, Computer, Volume: 38 Issue: 2 pp.63–69, Feb. 2005.

[36] Vinny Reynolds, Vinny Cahill, Aline Senart, "Requirements for an ubiquitous computing simulation and emulation environment," InterSense '06 Proceedings of the first international conference on Integrated internet ad hoc and sensor networks Article No. 1, 2006.

[37] 独立行政法人情報処理推進機構 ソフトウェア・エンジニアリング・センター, “【改訂版】 組込みソフトウェア向け開発プロセスガイド,” 翔泳社, Nov. 19. 2007.

[38] “世界最大規模のエミュレーション基盤 StarBED3 ウェブサイト,” NICT, http://starbed.nict.go.jp/archives/starbed3/aboutus/purpose.html, (accessed: 2019-1-8).

[39] “平成 18 年度 次世代ユビキタスネットワークシミュレーション技術研究開発プロジェクト 研究開発報告書,” NICT, Mar. 2007.

[40] “SmartSantander,” SmartSantander, http://www.smartsantander.eu/, (accessed: 2019-1-10).

[41] “StarBED4 プロジェクト ウェブサイト | 公開展示・発表事例ライブラリ,” NICT, http://starbed.nict.go.jp/research/pdf/20180908_UBICOMP2018/AOBAKO_poster_ubicomp2018.pdf, (accessed: 2019-1-17).

[42] “QEMU,” QEMU, https://www.qemu.org/, (accessed: 2019-1-25).

[43] Fabrice Bellard, “QEMU, a Fast and Portable Dynamic Translator,” USENIX, 2005.

[44] Evan Robert Platt, “Virtual Peripheral Interfaces in Emulated Embedded Computer Systems,” MASTER OF SCIENCE IN ENGINEERING Report, THE UNIVERSITY OF TEXAS AT AUSTIN, Dec. 2016.

[45] “Qt — Cross-platform software development for embedded & desktop,” Qt, https://www.qt.io/, (accessed: 2019-1-26).

[46] Mingsong Lv, Qingxu Deng, Nan Guan, Yaming Xie, Ge Yu, “ARMISS: An Instruction Set Simulator for the ARM Architecture,” 2008 International Conference on Embedded Software and Systems, 12 Aug. 2008.

[47] Harold W. Cain, Kevin M. Lepak, and Mikko H. Lipasti, “A Dynamic Binary Translation Approach to Architectural Simulation,” ACM, ACM SIGARCH Computer Architecture News, vol. 29, pp.27–36, 2001.

[48] “ARM コンパイラツールチェーンバージョン 4.1 ARM プロセッサをターゲットとしたソフトウェア開発,” ARM, http://infocenter.arm.com/help/topic/com.arm.doc.dui0471bj/DUI0471BJ_developing_for_arm_processors.pdf, (accessed: 2018-09-28).

[49] Adam Dunkels, Bjorn Gronvall, Thiemo Voigt, “Contiki-a lightweight and flexible operating system for tiny networked sensors,” IEEE, 29th Annual IEEE International Conference on Local Computer Networks, pp.455–462, 2004.

[50] Matthias Kovatsch, Simon Duquennoy, Adam Dunkels, “A low-power CoAP for Contiki,” IEEE, 2011 IEEE 8th International Conference on Mobile Adhoc and Sensor Systems (MASS), pp.855–860, 2011.

[51] Joakim Eriksson, Fredrik Österlind, Niclas Finne, Nicolas Tsiftes, Adam Dunkels, Thiemo Voigt, Robert Sauter, Pedro José Marrón, “COOJA/MSPSim: interoperability testing for wireless sensor networks,” Proceedings of the 2nd International Conference on Simulation Tools and Techniques, p.27, 2009.

[52] Toshiyuki Miyachi, Takeshi Nakagawa, Ken-ichi Chinen, Shinsuke Miwa, Yoichi Shinoda, “StarBED and SpringOS architectures and their performance,” International Conference on Testbeds and Research Infrastructures, pp.43–58, 2011.

[53] Jeffrey G. Andrews, Stefano Buzzi, Wan Choi, Stephen V. Hanly, Angel Lozano, Anthony C. K. Soong, J. C. Zhang, “What Will 5G Be?,” IEEE, IEEE Journal on Selected Areas in Communications (Volume: 32, Issue: 6 , June 2014), pp.1065–1082, Jun. 2014.

[54] “3GPP における 5G 標準化動向,” NTT DoCoMo, https://www.nttdocomo.co.jp/corporate/technology/rd/technical_journal/bn/vol25_3/003.html, (accessed: 2019-02-04).

[55] “Kris Pister, SMART DUST, https://people.eecs.berkeley.edu/~pister/SmartDust/, (accessed: 2019-02-04).

付録

A.1　実験 6.4 のエミュレータ時間計測に関して

実験に使用した time コマンドは、時刻計測に C 言語の gettimeofday 関数を使用している。これは、今後廃止予定の関数で、分解能も clock_gettime 関数がナノ秒オーダであるのに対して gettimeofday 関数はマイクロ秒オーダであり、10^3 倍悪い。

したがって、time コマンドを使用せず、C 言語の clock_gettime を使用した方法で計測した結果を図 A.1 に示す。図中「-2」のサフィックスは clock_gettime 関数を使用した計測で、無いものは time コマンドを使用した計測である。

また、表 A.1 に time コマンド使用時の結果と併記した。計測誤差とみられる程度の時間差は認められるが、spireg が最も高く、exodus が spireg より低くかった。これに加えて、pspio より exodus が少し高くなる傾向は変わらないため、本実験で検証したい spireg と exodus の計算時間の関係性は time コマンドでも十分示すことができている。

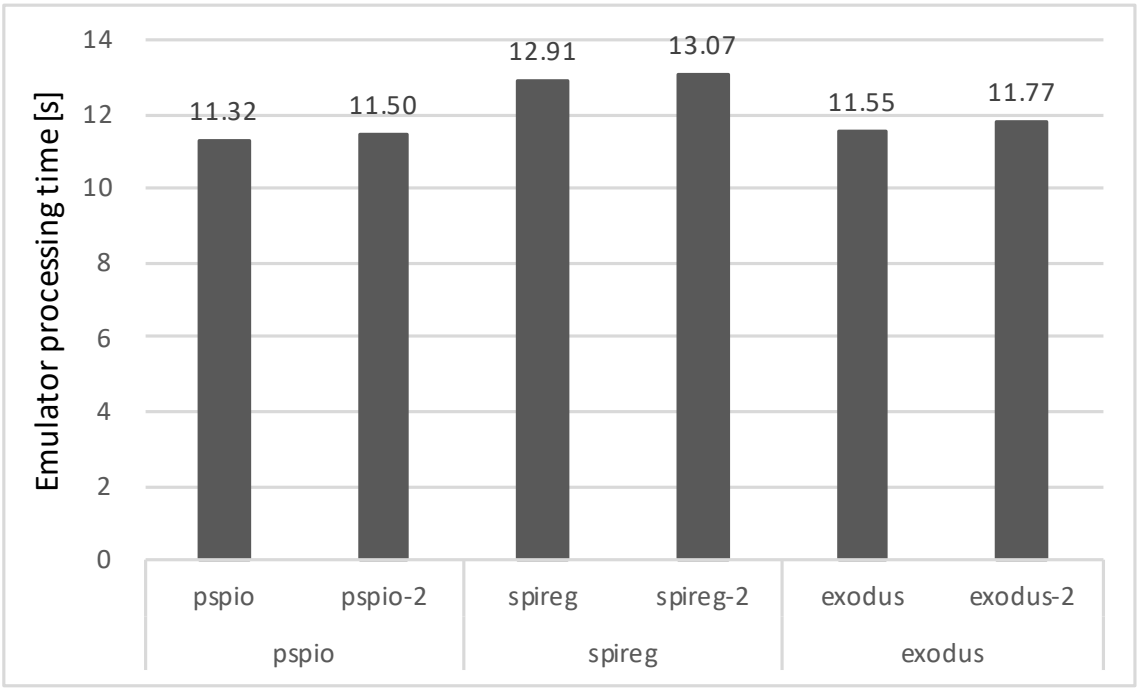

図 A.1: エミュレータの起動から終了までの実行時間計測

表 A.1: エミュレータの起動から終了までの実行時間計測のまとめ

エミュレータ	time コマンド	clock_gettime 関数
pspio	11.50	11.32
spireg	13.07	12.91
exodus	11.77	11.55

B.2　実験に使用したソースコード

本研究の実験ではエミュレータで動作するアプリケーションを開発した。Cortex-M0+エミュレータからSPIを通じてコマンドを送出し、温度データを受信する。使用プログラム言語はC++である。実験で使用したアプリケーションのソースコードをList 1から2に示す。

List 1: sketch.ino (疑似SPI I/Oモデル、SPIレジスタモデル用)

```
/*
  Analog Devices ADT7310 : A temperature meter
  SDN : pin 7
  CSN : pin 10
  SDI/MOSI : pin 11
  SDO/MISO : pin 12
  SCK : pin 13
*/

#include "adt7310.h"
#include <SPI.h>

#define chipSelectPin 10
#define shutDown 7
#define outpin 6

void setup()
{
  pinMode(outpin, OUTPUT);

  SPI.begin();

  pinMode(shutDown, OUTPUT);
  digitalWrite(shutDown, HIGH);
  pinMode(chipSelectPin, OUTPUT);
  digitalWrite(chipSelectPin, HIGH);
  delay (10);
}

void loop()
{
  if(!read_temp())
  {
    digitalWrite(outpin, HIGH);
  }else
  {
    digitalWrite(outpin, LOW);
  }
```

```
39   delay(1000);
40 }
41
42 int read_temp()
43   {
44   uint8_t a = 0;
45   uint8_t b = 0;
46   uint8_t data_proc;
47   float temperature;
48   unsigned int value;
49
50   digitalWrite(chipSelectPin, LOW);
51   delay(3);
52   SPI.transfer(0x50); // one-shot mode
53   a = SPI.transfer(0x00);
54   b = SPI.transfer(0x00);
55   value = ((a << 8) | b);
56   temperature = calc(value, 16);
57   delay(3);
58   digitalWrite(chipSelectPin, HIGH);
59
60   digitalWrite(chipSelectPin, LOW);
61   delay(3);
62   SPI.transfer(0x50); // one-shot mode
63   a = SPI.transfer(0x00);
64   b = SPI.transfer(0x00);
65   value = ((a << 8) | b);
66   temperature = calc(value, 16);
67   delay(3);
68   digitalWrite(chipSelectPin, HIGH);
69
70   digitalWrite(chipSelectPin, LOW);
71   delay(3);
72   SPI.transfer(0x60); // health check(read Thigh)
73   a = SPI.transfer(0x00);
74   b = SPI.transfer(0x00);
75   value = ((a << 8) | b);
76   temperature = calc(value, 16);
77   delay(3);
78   digitalWrite(chipSelectPin, HIGH);
79
80   return 0;
81 }
```

List 2: sketch.ino (Exodus モデル用)

```
1 /*
```

```
  Analog Devices ADT7310 : A temperature meter
  SDN : pin 7
  CSN : pin 10
  SDI/MOSI : pin 11
  SDO/MISO : pin 12
  SCK : pin 13
*/
#define chipSelectPin 10
#define shutDown 7
#define outpin 6

void setup()
{
  pinMode(outpin, OUTPUT);

  SPI.begin();

  pinMode(shutDown, OUTPUT);
  digitalWrite(shutDown, HIGH);
  pinMode(chipSelectPin, OUTPUT);
  digitalWrite(chipSelectPin, HIGH);
  delay (10);
}

void loop()
{
  if(!read_temp())
  {
    digitalWrite(outpin, HIGH);
  }else
  {
    digitalWrite(outpin, LOW);
  }
  delay(1000);
}

int read_temp()
{
  uint8_t a = 0;
  uint8_t b = 0;
  uint8_t data_proc;
  float temperature;
  unsigned int value;

  digitalWrite(chipSelectPin, LOW);
  delay(3);
```

```
48   SPI.exdTransfer(0x50); // one-shot mode
49   a = SPI.exdTransfer(0x00);
50   b = SPI.exdTransfer(0x00);
51   value = ((a << 8) | b);
52   temperature = calc(value, 16);
53   delay(3);
54   digitalWrite(chipSelectPin, HIGH);
55
56   digitalWrite(chipSelectPin, LOW);
57   delay(3);
58   SPI.exdTransfer(0x50); // one-shot mode
59   a = SPI.exdTransfer(0x00);
60   b = SPI.exdTransfer(0x00);
61   value = ((a << 8) | b);
62   temperature = calc(value, 16);
63   delay(3);
64   digitalWrite(chipSelectPin, HIGH);
65
66   digitalWrite(chipSelectPin, LOW);
67   delay(3);
68   SPI.exdTransfer(0x60); // health check(read Thigh)
69   a = SPI.exdTransfer(0x00);
70   b = SPI.exdTransfer(0x00);
71   value = ((a << 8) | b);
72   temperature = calc(value, 16);
73   delay(3);
74   digitalWrite(chipSelectPin, HIGH);
75
76   return 0;
77 }
```

実験でライブラリ用いた SPI ライブラリのソースコードは、Arduino のライブラリ [32] を参考にして拡張した。List 3 と List 4 に変更点のみ掲載する。

List 3: SPI.h の SPI Class (変更点のみ抜粋)

```
1  class SPIClass {
2    public:
3    SPIClass(SERCOM *p_sercom, uint8_t uc_pinMISO, uint8_t uc_pinSCK
         , uint8_t uc_pinMOSI, SercomSpiTXPad, SercomRXPad);
4
5    byte transfer(uint8_t data);
6    uint16_t transfer16(uint16_t data);
7    void transfer(void *buf, size_t count);
8
9    // Add methods for "exd".
10   uint8_t readTransfer();
```

```
11   void writeTransfer(uint8_t data);
12   uint8_t exdTransfer(uint8_t data) __attribute__((noinline));
13
14   // Transaction Functions
15   void usingInterrupt(int interruptNumber);
16   void beginTransaction(SPISettings settings);
17   void endTransaction(void);
18
19   // SPI Configuration methods
20   void attachInterrupt();
21   void detachInterrupt();
22
23   void begin();
24   void end();
25
26   void setBitOrder(BitOrder order);
27   void setDataMode(uint8_t uc_mode);
28   void setClockDivider(uint8_t uc_div);
29
30   private:
31   void init();
32   void config(SPISettings settings);
33
34   SERCOM *_p_sercom;
35   uint8_t _uc_pinMiso;
36   uint8_t _uc_pinMosi;
37   uint8_t _uc_pinSCK;
38
39   SercomSpiTXPad _padTx;
40   SercomRXPad _padRx;
41
42   bool initialized;
43   uint8_t interruptMode;
44   char interruptSave;
45   uint32_t interruptMask;
46 };
```

List 4: SPI.cpp (変更点のみ抜粋)

```
1  uint8_t SPIClass::exdTransfer(uint8_t tdata)
2  {
3    uint8_t rdata = 0;
4
5    rdata = readTransfer();
6    writeTransfer(tdata);
7    writeTransfer(tdata);
8
```

```
 9   return rdata;
10 }
11
12 void SPIClass::writeTransfer(uint8_t tdata)
13 {
14   __asm volatile("mov R0, %[input]"
15     :[input] "=r" (tdata)
16   );
17   __asm volatile("mov R2, R0");
18 }
19
20 uint8_t SPIClass::readTransfer()
21 {
22   uint8_t rdata;
23
24   __asm volatile("mov %[res], R0"
25     :[res] "=r" (rdata)
26   );
27   return rdata;
28 }
```

本実験で使用した Analog Devices 社の温度センサ ADT7310 は、1 つの温度データを送出する際に 2 バイトあるデータを 1 バイトずつ送出する。その際のデータフォーマットを図 B.2 に示す。これを受信側のアプリケーションで処理することによって、1 つの温度情報として扱うことができる。その際に使用するライブラリを List 5 と List 6 に示す。

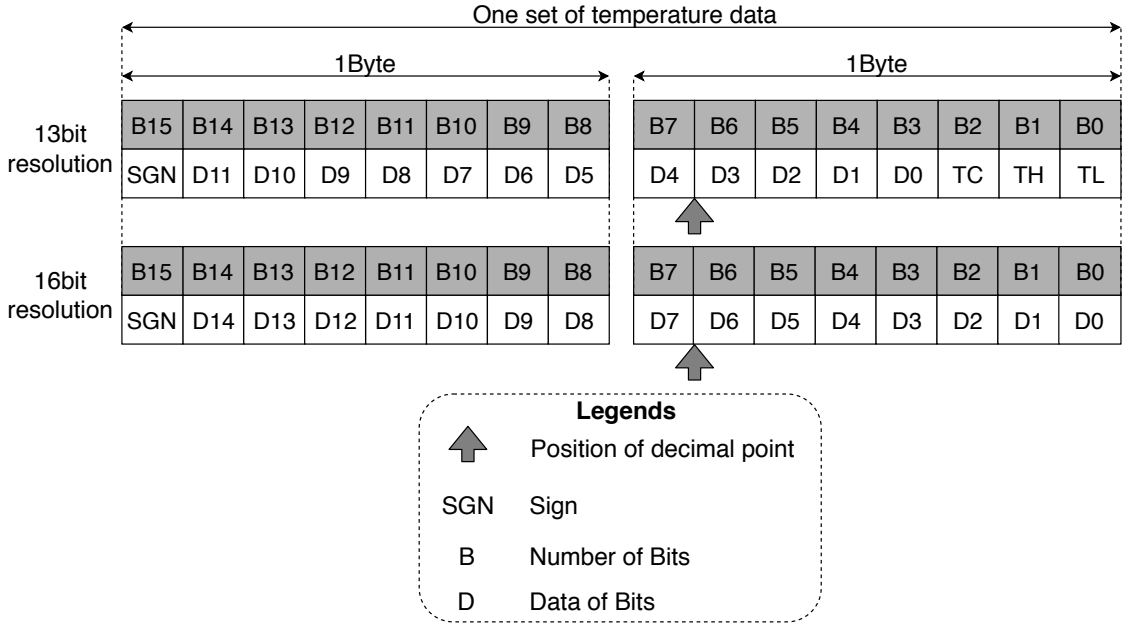

図 B.2: ADT7310 が送出するデータフォーマット

List 5: ”adt7310.h”

```
#include "Arduino.h"

float calc(unsigned int val, int bitlen);
```

List 6: ”adt7310.cpp”

```
#include "adt7310.h"
float calc(unsigned int val, int bitlen)
{
  float t;
  int flag = false;

  val = (val >> (16 - bitlen));
  if (bitRead(val, bitlen - 1) == 1)
  {
    val |= (0xFFFF << bitlen);
    val = (~val + 1);
    flag = true;
  }

  t == -9999.99;
  if (bitlen == 16)
  {
    t = val / 128.0;
  }
  else if (bitlen == 13)
  {
    t = val / 16.0;
  }
  else if (bitlen == 10)
  {
    t = val / 2.0;
  }
  else if (bitlen == 9)
  {
    t = val * 1.0;
  }

  if (flag) t = (-1) * t;

  return t;
}
```