卒 業 研 究 報 告 書

題　目　　エミュレーションと状態遷移の規則による
組み込みデバイスドライバーの
バグ検出手法の提案

指導教員　　水野　修　教授

崔　恩瀞　助教

京都工芸繊維大学 工芸科学部 情報工学課程

学生番号　　20122003

氏　　名　　有川　康幸

令和2年2月13日提出

目　次

エミュレーションと状態遷移の規則による
組み込みデバイスドライバーのバグ検出手法の提案

令和 2 年 2 月 13 日　　20122003　有川　康幸

概　要

IoT 機器をはじめとする組み込みデバイスは，処理を担うマイクロコントローラだけではなく，周辺機器を用いて機能を実現していることが多い．ここで，周辺機器を使用するためのデバイスドライバは，開発者が対象とするマイクロコントローラ向けに提供されていない場合がある．デバイスドライバが提供されていない場合，開発者がデバイスドライバを独自に開発することが多く見られ，バグが生じやすくなっている．デバイスドライバのデバッグには，まず周辺機器に対する問題のある操作 (バグ) を 検出する必要がある．この作業は，実機を用いて行われることが多く，開発の効率化における課題の 1 つとなっている．

組み込みデバイスのデバッグの手法として，エミュレーションがある．エミュレーションは，実機を用いることなく，汎用コンピュータシステム上でマイクロコントローラの動作を再現することでバグ検出などのデバッグを行う．しかし，先行研究ではデバイスドライバ特有のバグを検出することが難しく，デバイスドライバのデバッグには使用できない．

ここで，周辺機器は状態遷移を記述することである程度のモデル化が可能である．また，状態遷移が生じる前提条件 (制約) を定義することで，周辺機器の操作に関するバグ検出が可能になる．

そこで本研究では，周辺機器を含むマイクロコントローラの動作をエミュレーションし，状態遷移の制約を元にバグを検出する手法を提案する．本論文では，提案手法について意図的にバグを埋め込んだファームウェアを用いて．バグ検出ができることを確認した．

1. 緒言

組み込みデバイスは，Internet of Things(以降，IoT) 機器をはじめとした各種機器に組み込まれる制御用のコンピュータシステムのことを指す．組み込みデバイスは，CPU にあたるマイクロコントローラと，周辺機器で構成されている．

組み込みデバイス向けのソフトウェアであるファームウェアの開発において，デバッグは実機を用いて行われることが多い．しかし，実機を用いるデバッグは，実機の数やデバッグのための機能が限られているなどの理由により，効率的に行うことが難しい．また，実機を用いたデバッグでは，デバッグ機能が限られるため，バグを再現することや原因を特定することが難しくなっている [1].

実機を用いるデバッグ手法の困難を解消するために，エミュレーションを用いた手法が提案されている [2]．これは，実機の代わりにエミュレータを用いてファームウェアを動作させ，バグ検出やフォルトローカライゼーションなどのデバッグを行う手法である．この手法では実機が不要であり，さらに詳細な実行のトレース取得が可能になるなどデバッグに利用できる機能も多くなる．これにより，効率的なデバッグ作業が可能となる．このように，エミュレーションを用いたデバッグ手法は，ファームウェアの開発コストの削減に寄与できる．

組み込みデバイスのエミュレーションについての先行研究は複数存在する [3, 4, 5]. 先行研究においてはセキュリティ上の脆弱性検出やクラッシュする操作の検出などの一般的なバグを対象としたバグ検出手法が提案されている．

しかし，実際の組み込みデバイスを見ると，マイクロコントローラだけではなく，周辺機器を用いて機能を実現していることが多い．周辺機器を使用するための組み込みデバイスドライバ（以降，デバイスドライバ）は，自作されることも多く，バグが生じやすい．この種類のバグに着目している研究は少なく，例えば Zhou らの研究では，周辺機器の動作を詳細にエミュレーションしないため，デバイスドライバのデバッグに活用することは難しい．また，デバイスドライバのデバッグは，バグによって生じる周辺機器のエラーが直接クラッシュなどを引き起こさないという点からもデバッグが難しい．そのため，デバイスドライバのデバッグには，まず周辺機器に対する問題のある操作の存在を検出する必要がある．それにより，バグが存在すると判断し，デバッグを進めることができる．

本論文では，周辺機器を含むマイクロコントローラの動作をエミュレーションし，デバイスドライバのバグを検出する手法を提案する．エミュレーションは周辺機器の動作を開発者が記述した状態遷移のモデルを用いて行う．また，デバイスドライバのバグの検出は，状態遷移のモデルに基づいて行う．

提案した手法が有効であることを確認するために，意図的にバグを埋め込んだファームウェアを用いて実験した．その結果，提案した手法により，バグを検出できることを確認した．

本報告書の構成は以下の通りである．第 2 章では，本論文に関連する先行研究と用語について述べる．第 3 章では，提案手法について説明する．第 4 章では，提案手法の有効性を確認するための実験について述べる．第 5 章では，実験の結果について述べる．第 6 章では，提案手法の有効性と今後の課題について考察する．

2. 背景

2.1 組み込みデバイス

組み込みデバイスは，各種機器に組み込まれる制御用のコンピュータシステムのことを指す．特定の機器に最適化された機能を持つこと，リソースが限られていること，リアルタイム性の要求が特徴として挙げられ，汎用コンピュータシステムであるPCとは大きく異なる構成を取ることが多い[6].

組み込みデバイスの例として，IoT機器や車載機器，産業用ロボットなどが挙げられる．この中でもインターネット接続するセンサなどのIoT機器は，近年急速に普及しており，開発が盛んに行われている[7].

組み込みデバイスは，図2.1に示すようにCPUにあたるマイクロコントローラと，センサやアクチュエータなどの周辺機器で構成されている．マイクロコントローラには，通常のPC向けプロセッサーと異なり，汎用入出力（General-purpose input output: GPIO）や，Serial Peripheral Interface（SPI）・Inter-Integrated Circuit（I2C）のような各種通信用のインターフェースが設けられており，周辺機器を利用する際に用いられる．マイクロコントローラ単体で成立する機能は少なく，多くの組み込みデバイスは周辺機器を用いて機能を実現している．

ファームウェアとは，マイクロコントローラ上で動作するソフトウェアを指す．ファームウェアは，GPIOや通信用のインターフェースを操作し，組み込みデバイスを制御する．一般に，ファームウェアは組み込みデバイスの構成や用途ごとに開発されるため，多様なファームウェアが存在する．

ファームウェアはいくつかの種類に分類できる[8]．汎用OSを利用するファームウェアは，Linuxなどの OSにより動作する．組み込みOSを利用するファームウェアは，VxWorks(注 1) やZephyr(注 2)などの組み込み特化のOSにより動作する．OSによる抽象化を行わないファームウェアは，mbed OS(注 3) などのライブラリを用いており，単一の制御ループと割り込みによって動作する．

汎用OSを利用するファームウェアは，PCと同様に，プロセス管理やネットワー

(注 1)：https://www.windriver.com/japan/products/vxworks
(注 2)：https://www.zephyrproject.org/
(注 3)：https://os.mbed.com/mbed-os/

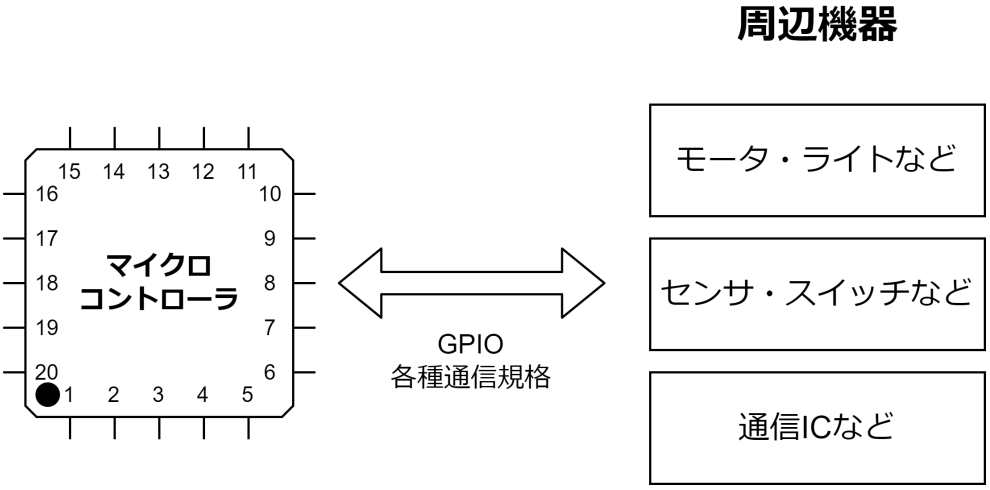

図 2.1　組み込みデバイスの構成

ク機能を利用でき，一般的なソフトウェア開発手法を用いることができる．しかし，それ以外のファームウェアは，その構成の違いから，開発手法について大きく違いが見られる [1].

まず，結合テストのような自動テスト手法は用いられないことが多い．これには，テスト環境に必要なモックを用意することは困難であることが理由として挙げられる．そのため，開発者は実機を用いた手動でのテストやデバッグを行っている．

ファームウェアのデバッグには，実機における通信や実行のトレース等の詳細なログを用いることが望ましい．しかし，マイクロコントローラの性能制約により，ログを取得するための機能は実装されていないか，限定的であることが多い．そのため開発者は，ログの取得を汎用的なシリアル通信と print 命令を用いて行うことが多い．これにより得られるログは限定的であり，通常の PC におけるソフトウェアのデバッグと比べてマイクロコントローラのデバッグは手間を要する．

2.1.1 周辺機器

組み込みデバイスは，マイクロコントローラと周辺機器で構成されている．そのうち，周辺機器は，センサやアクチュエータなどの機能を提供する．

周辺機器は，GPIO や SPI，I2C などの通信インターフェースを用いてマイクロコントローラと通信するものが多い．このうち，I2C や SPI を用いる周辺機器は，専用のコマンドを用いて内部に存在するレジスタを読み書きすることで，周辺機器を操作する．

周辺機器のレジスタの詳細は，データシートに記載されている．データシートは，レジスタに書き込むことで周辺機器がどのような動作をするかを記述している．周辺機器の動作の大部分は，単純な状態遷移として捉えることが可能であり，データシートによっては状態遷移図が記載されていることもある．状態遷移は，遷移を発生させる直接の要因となる操作と，遷移に必要な前提条件の制約の 2 つを条件として行われる．本論文では，この前提条件の制約を状態遷移の制約と呼ぶ．

Zhou らの研究では，周辺機器ではなくマイクロコントローラを状態遷移するモデルとして扱い，そのモデルを用いてファームウェアのバグを検出する手法を提案している [3]． Zhou らは状態遷移の制約を満たすかどうかを確認することで，ファームウェアのバグを検出する手法も同時に提案している．

2.2 デバイスドライバ特有のバグ

ファームウェアのうち，周辺機器を直接操作する部分を組み込みデバイスドライバ（以降，デバイスドライバ）と呼ぶ．デバイスドライバは，一般に I2C や SPI を用いて周辺機器内部のレジスタを読み書きすることで周辺機器を操作している．

組み込みデバイスにおいては，多様なデバイスの存在により，デバイスドライバが独自に開発されることがある．この場合，デバイスドライバのバグは容易に発生する．バグが存在すると，周辺機器を正しく操作できなくなり，不正な値が読み取られる，意図しない動作が生じる．そのため，デバイスドライバのバグを発見し修正することは重要であり，組み込みデバイスの信頼性向上に繋がる．

デバイスドライバのバグの発見は，ファームウェア自体のバグの発見よりも難しい．これは，周辺機器での操作が直接ファームウェアをクラッシュさせることが少ないためである．

デバイスドライバのバグの事例として，TCS34725 というセンサを用いた際のバグが挙げられる．当該センサは，割り込みが発生する度に割り込みの状態をリセットしなければ，次の割り込みが発生しない．しかし，デバイスドライバが割り込みの状態をリセットしていなかったため，割り込みが発生しなくなるというバグが発生した．

デバイスドライバのバグを減らす手法として，デバイスドライバの自動生成等が研究されている [9]．また，静的解析によりデバイスドライバのバグを検出する手法が研究されている [10]．しかし，実機またはエミュレーションを用いた動的解析によるデバイスドライバのバグ検出は，ファジングを除き，組み込みデバイスの分野においてあまり研究されていない．

デバイスドライバのバグが生じる原因は以下のようなものが挙げられる．

- 周辺機器の使用方法を誤って把握している．
- 割り込みを考慮していない．割り込みの対応が不十分である．

周辺機器の使用方法を誤って把握したことによるバグは，周辺機器の操作が誤っていて，意図しない動作が起きることを指す．

割り込みは，周辺機器の利用においては GPIO を変化させることにより，状態変化を素早く伝達する手法である．割り込みはその目的から，発生した直後に対応す

ることが望ましい．そのため，割り込みが発生した後に対応する処理を行われていない場合，不要な割り込み機能が有効化されていたり，割り込みの対応が不十分であると見なすことができる．これは，バグの可能性がある挙動である．

これらのバグは，デバイスドライバで行われている操作を単純な制約により評価することで検出できる．状態遷移の制約は，検出に利用できる制約の 1 つである．本研究では，この制約を利用してデバイスドライバのバグを検出する．

2.3 マイクロコントローラのエミュレーション

マイクロコントローラのエミュレーション（以降，エミュレーション）は，PC 上でマイクロコントローラを再現することで，ファームウェアの開発を支援する手法である．QEMU (注 4)を用いたエミュレーションフレームワークや，Web ブラウザ上で動作する Wokwi (注 5)などがある．

エミュレーションの用途として，セキュリティ上のバグ (脆弱性) 検出が挙げられる．これを目的として，複数の研究が行われている [3, 4, 5].

実機を用いることに対するエミュレーションの利点は，実機のリソース的制約の影響を受けないことである．これによりデバッグの規模を大きくでき，短時間でのテスト駆動開発，継続的インテグレーションなどを適用できるようになる．同時に，実機の場合では取得できない詳細なトレースなどのデバッグ情報を得ることも可能になり，デバッグの効率化にもつながる．

エミュレーションを用いてバグを検出する研究は，デバイスドライバに対してはあまり行われていない．これは，周辺機器のエミュレーションが，大きな課題であるためである [2]．先行研究として，マイクロコントローラ以外の機器は実機で行う Hardware-in-the-loop のエミュレーションや，機械学習により周辺機器の挙動を予測する手法がある．しかし先行研究は，自動生成された大量の入力をバグが見つかるまで与えるファジングを用いる手法が多い．また，対象がファームウェアのクラッシュなど周辺機器の関係しないバグに限定されている．加えて，ファジングを用いる手法は一般にバグが検出されるまでに要する時間が長い手法であり，開発者にとって手軽な手法ではない．そのため，デバイスドライバのバグ検出を支援する手法と

(注 4)：https://www.qemu.org/
(注 5)：https://wokwi.com/

しては実用性が低い．

エミュレーションを行っている研究の 1 つに，HALucinator[5] がある．HALucinator は，QEMU を用いたエミュレーションフレームワークであり，ハードウェア抽象化レイヤー (Hardware abstraction layer: HAL) を置き換えることにより汎用性の高いエミュレーションを実現している．その一環として，HALucinator は，マイクロコントローラの入出力機能に接続される周辺機器の動作を Python を用いて記述する仕組みを用意している．本研究では，HALucinator を用いてエミュレータを作成する．

3. 手法

第 2.3 節で示した通り，デバイスドライバ特有のバグを検出できる実用的なエミュレータは存在しない．

デバイスドライバ特有のバグを検出するためには，まず周辺機器の動作をエミュレーションするエミュレータが必要である．また，デバイスドライバ特有のバグを検出するためには，クラッシュなどマイクロコントローラで生じるエラーだけでなく，周辺機器の使用方法が誤っていることによるバグも検出する必要がある．本研究では，周辺機器の状態遷移の制約を定義し，その制約を満たさない操作を検出することで，デバイスドライバ特有のバグを検出する手法を提案する．

提案手法の概要を図 3.1 に示す．図 3.1 において，細い矢印は入力・出力であり，太い矢印は通信を示している．

なお，本論文において対象とするファームウェアは，第 2.1 節で示した組み込み OS を利用するファームウェアと OS による抽象化を行わないファームウェアである．

3.1 周辺機器をエミュレーションするソフトウェアの作成

周辺機器のエミュレータを第 2.3 節で紹介した HALucinator を拡張する形で開発した．組み込みデバイスの部品とエミュレータの関係を図 3.2 に示す．本研究で取り扱う周辺機器は，内部にメモリ空間を持ち，そこに I2C や SPI を用いてアクセスする周辺機器を想定している．

エミュレータは周辺機器のモデルを再現する部分と，HALucinator とやり取りする部分により成立している．

モデルを再現する部分は，周辺機器について事前に定義したメモリ空間を確保し，レジスタの挙動を再現するコンポーネントである．このコンポーネントは，同時に特定のレジスタへの書き込みに反応しイベントを発生させる機能と，レジスタの読み書きのログを出力する機能を持つ．メモリ空間は以下に示す内容を YAML 形式[(注 6)]で定義する．

- レジスタのビット数・名前・アドレス

(注 6)：Yet-another Markup Language

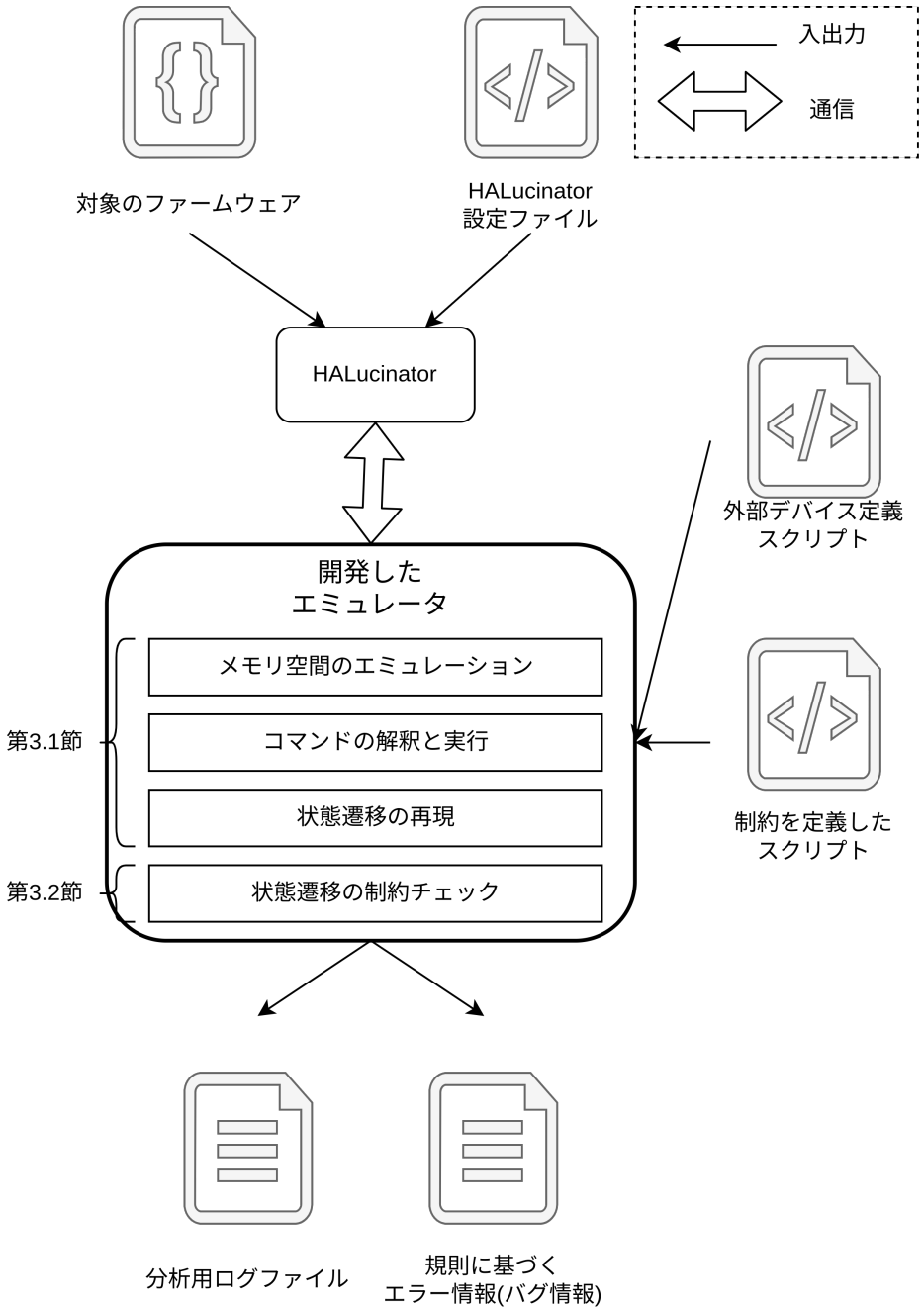
入出力
通信
対象のファームウェア
HALucinator
設定ファイル
HALucinator
外部デバイス定義
スクリプト
開発した
エミュレータ
メモリ空間のエミュレーション
第3.1節
コマンドの解釈と実行
状態遷移の再現
第3.2節
状態遷移の制約チェック
制約を定義した
スクリプト
分析用ログファイル
規則に基づく
エラー情報(バグ情報)

図 3.1　提案手法の概要

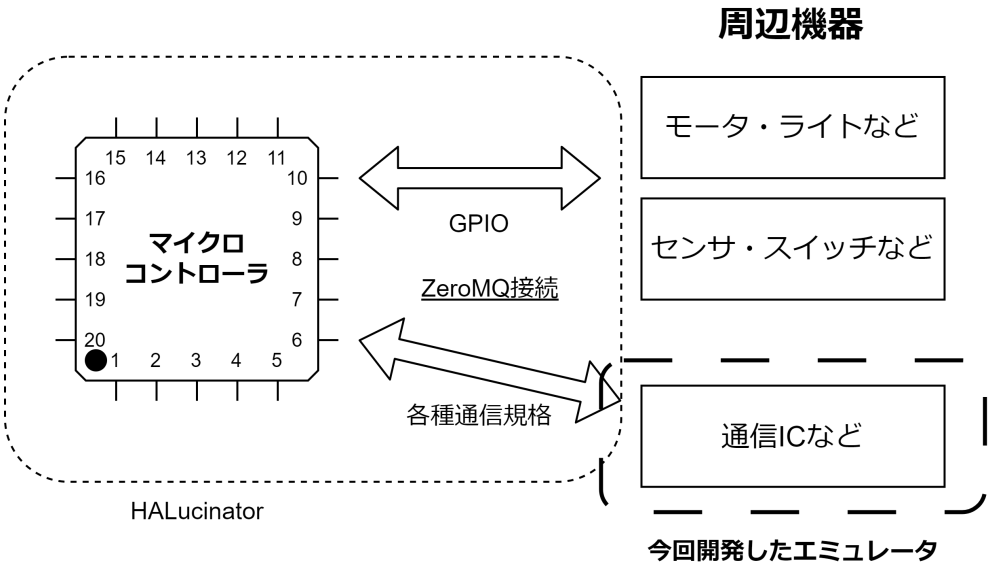

図 3.2　エミュレータの構成

- レジスタ内部のフィールドの名前・ビット幅・デフォルト値

イベントについても一部を YAML 形式で定義する．YAML 形式で定義するイベントは，単純な状態変化に限られ，**条件 (when)** と**操作 (what)** の組み合わせで定義される．**操作**は以下に示すものが利用でき，複数個定義されている場合は AND で評価される．

- 特定のレジスタの値が式を満たすようになった時
- 特定のレジスタの値が式を満たしている間

操作には，レジスタの値を特定の値に書き換える操作が利用できる．また，時間を要する初期化動作のエミュレーション等を目的に，**操作**の実行を一定時間遅延させることも可能である．

YAML 形式で定義されるイベントの例をコード 3.1 に示す．

コード 3.1　イベントの例

```
rules:
    # C1CON.REQOP を C1CON.OPMOD に反映する
    # 反映の時間差をエミュレートする
    - when:
        - left:
            register: C1CON.REQOP
            trigger: true # 書き換えられたときのみ反応する
          op: ne
          right:
            register: C1CON.OPMOD
      what:
        - set: C1CON.REQOP
          value: 0
          defer: 1
        - set: C1CON.OPMOD
          from: C1CON.REQOP
```

YAML 形式のイベントで対応できない割り込みや周辺機器固有の処理は，Python コードによって記述される．コードにおいては，特定レジスタの値の変更により定義した関数が呼び出される．また，一定時間ごとに定期的に呼び出される関数を定義できる．

YAML 形式のモデルと Python コードによるモデルは，周辺機器のデータシートを参照し，開発者が作成する．

HALucinator とのやり取りは，ZeroMQ [注 7]により行われる．HALucinator は，SPI や I2C の通信をエミュレータに転送し，エミュレータからの応答をファームウェアに対して返却する．エミュレータにおいては，SPI や I2C のコマンド列を解釈し，レジスタの読み書きと応答を作成する．

3.2 デバイスドライバのバグ検出

前節で開発したエミュレータを用いて，デバイスドライバのバグを検出する手法を提案する．ここで対象とするバグは 2 種類に分けられる．

まず，使用方法を誤って把握していることによるバグである．これには，以下のようなものが含まれる．

- 存在しないコマンドを送信する．
- メモリ空間範囲外のアドレスにアクセスする．
- ある条件のもとでのみ許可される操作を，その条件を満たさない状態で行う．
- 割り込みが生じているのに無視する．

存在しないコマンドの送信およびメモリ空間範囲外のアドレスへのアクセスは，明らかなバグである．これは，エミュレータでのコマンドの解釈時に検出する．

条件を満たさない状態での操作は，通信を開始できない状態での通信開始や，書き込みが禁止されているアドレスに書き込む動作を指す．こうした動作は，予期しない動作を引き起こす場合がある．これらは，レジスタの状態が遷移するときに，その遷移が許可されているかを確認することで検出する．遷移の許可は，イベントの条件と同等の記法で記述した規則を用いる．

割り込みが生じているのに無視する動作は，何らかのイベントが発生したという情報を無視する動作であり，バグに繋がる可能性が高い動作である．これらは，割り込みが生じていることがわかるレジスタの値を監視し，ある時間を超えて割り込みが生じているが変化していない場合に検出する．

(注 7)：https://zeromq.org/

4. 実験

本章では，第 3 章で説明した提案手法の有効性を評価するための実験について解説する．実験では，作成したエミュレータを用いて，ファームウェアをエミュレーションする．その際に提案したバグ検出手法を利用し，バグを検出できるかを確認する．正しくバグを検出できれば提案手法には有効性があると言える．

4.1 使用する環境とソフトウェア

本実験では，以下のような環境で実験した．

- Docker 上に構築された Ubuntu 20.04 LTS
- HALucinator 用に改変された QEMU (注 8)
- HALucinator 用に改変された Avatar2 (注 9)
- HALucinator[11]
- CAN 通信用拡張基板を装着した Raspberry Pi 4B 上に構築された Ubuntu 22.04 LTS (ファームウェア実機テストのみ)

ファームウェアは，京都工芸繊維大学 ロボコン挑戦プロジェクト (注 10)で利用されている DC モータドライバを対象として作成する．この組み込みデバイスは，STM32F4 シリーズ [12] と MCP2517FD CAN コントローラ [13] を搭載する．

4.1.1 STM32

STM32 は，STMicroelectronics 社から発売されている，民生用の ARM コアを搭載したマイクロコントローラである．民間の組み込みデバイスで用いられている．本実験で用いる組み込みデバイスには，STM32F407VGT6 が搭載されている．

ファームウェアの開発にはいくつかのフレームワークが用意されている．本実験で用いるファームウェアについてはすべて mbed OS 6 を RTOS なしの状態で用いる．このファームウェアは第 2.1 節で示した分類において OS による抽象化を行わないデバイスと分類される．

(注 8)：https://github.com/sandialabs/halucinator_avatar-qemu
(注 9)：https://github.com/sandialabs/halucinator_avatar2
(注 10)：https://www.fortefibre.net

4.1.2 MCP2517FD

MCP2517FD は，Microchip 社から発売されている CAN コントローラである．CAN は車載機器やロボットなどの組み込みデバイスで用いられる通信規格である．MCP2517FD は，CAN FD (注 11)と呼ばれる，従来の CAN よりも高速な通信に対応している．

MCP2517FD は約 3kB のメモリ空間及び 193 個のレジスタ (注 12)を持ち，SPI によりレジスタを操作する．レジスタの一覧についてはデータシート [14] を参照されたい．SPI による操作コマンドは以下に示すものがある．

1. RESET: デバイスをリセットし，初期状態 (設定モード) に戻す．
2. READ: レジスタの値を読み取る．
3. WRITE: レジスタに値を書き込む．
4. READ_CRC: レジスタの値を読み取る．同時に CRC が返却され，が化けていないか検証できる．
5. WRITE_CRC: レジスタに値を書き込む．同時に CRC を渡し，値が化けていれば割り込みが生じる．
6. WRITE_SAFE: WRITE_CRC と同様だが，CRC を確認後に値を書き込む．

MCP2517FD は通常，以下の順序で初期化される．

1. 電源投入か，コマンド送信によりリセットする．
2. CAN および送受信バッファ(FIFO) に関する設定値の書き込む．
3. 通常動作モードへ移行する．
4. 空き送信バッファへのメッセージの書き込み，埋まっている受信バッファからのメッセージの読み取りを行う．

MCP2517FD をエミュレーションするためのモデルは，組み込み機器開発の経験が 1 年以上ある開発者により，作成された．

(注 11): CAN with Flexible Data-Rate
(注 12): CAN メッセージ保持領域を除く

4.2 バグを含むデバイスドライバの用意

バグを含むデバイスドライバとして，MCP2517FD のリファレンス・デバイスドライバ [13] にバグを埋め込む．

リファレンス・デバイスドライバは，PIC32 シリーズを対象としており，利用している HAL が異なる．そのため，本実験では，リファレンス・デバイスドライバを，制御フローを変更しないように，STM32F4 シリーズ向けに移植する．移植したリファレンス・デバイスドライバは，実機を用いて CAN 通信の動作確認を行う．

リファレンス・デバイスドライバを用いて作成したファームウェアは，一定間隔で CAN メッセージを送信する．また，CAN メッセージを受信すると，メッセージの内容を UART で出力する．

埋め込むバグを以下に示す．

1. 不正なコマンドを送信する．
2. 書き込みが禁止されているアドレスに書き込む．
3. Configuration Mode のときに CAN の通信を試みる．
4. IRQ が有効であるときに IRQ 信号を無視する．

それぞれについてデバイスドライバを作成し，正常に動作するファームウェアを含め 5 つのファームウェアを作成する．それぞれのバグと，その確認方法について解説する．

不正なコマンドを送信するとは，第 4.1.2 節で示したコマンド一覧に含まれないコマンドを送信することである．MCP2517FD は 8 ビットでコマンドを指定できるため，本実験では未使用のコマンドである 0x1F を送信し，それを検出できるか確認する．

書き込みが禁止されているアドレスに書き込むとは，メモリ空間の範囲外に書き込むか，何らかの理由で書き込みが禁止されているアドレスに書き込むことである．MCP2517FD は，一部のレジスタへの書き込みが禁止されている．本実験では，動作モードを示す読み取り専用のレジスタである OPMOD に書き込みを行い，それを検出できるか確認する．

Configuration Mode のときに CAN の通信を試みるとは，MCP2517FD の初期化手順において，通常動作モードへの移行前に CAN の通信を試みることである．こ

の段階では，CAN の通信が可能なことは保証されていないため，CAN の通信を試みることはバグの可能性がある挙動である．

IRQ が有効であるときに IRQ 信号を無視するとは，第 2.2 節で示したバグの可能性のある挙動である．本実験では，メッセージ受信時に生じる，バッファ非エンプティ割り込みを有効にした上で，割り込みが生じたときにメッセージ受信処理を行わない．割り込みが無視されたかを検出できるか確認する．

5. 実験結果

本章では，第 4 章で説明した実験の結果について解説する．

まず，移植したリファレンスソフトウェアを実機で動作させ，CAN 通信が正しく行われることを確認した．この確認においては，Raspberry Pi に CAN 通信用の拡張ボードを接続し，実機と接続，CAN 通信が正常に行われた．

次に，移植したリファレンスソフトウェアをエミュレータで動作させ，CAN 通信が正しく行われることを確認した．この確認においては，エミュレータの出力を確認し，実機と同様の動作が行われていることを確認した．

最後に，埋め込んだバグを検出できるか確認する．バグを検出できるかどうかは，エミュレータの出力を確認することで判断できる．出力として意図した制約の違反が確認できれば，バグが検出できたと判断する．結果を表 5.1 に示す．表 5.1 より，すべての埋め込んだバグは提案手法を用いて検出できたことがわかる．

表 5.1　バグ検出の結果

埋め込んだバグ	結果
不正なコマンドを送信する	検出
書き込みが禁止されているアドレスに書き込む	検出
Configuration Mode のときに CAN の通信を試みる	検出
IRQ が有効であるときに IRQ 信号を無視する	検出

6. 考察

6.1 議論

本論文で提案した手法は，周辺機器の動作を単純な状態遷移として取り扱っている．また，その状態遷移の状態はレジスタに反映されることを前提としている．この前提が満たされている場合，提案手法は有効である．

今回行った実験では，制約を定義する上で十分な情報がデータシートに記載されていることを確認している．十分な情報とは，状態遷移を把握できるレジスタの情報や，状態遷移の前提条件が記載されていることである．よって，実験ではこの 2 つの前提が満たされている周辺機器を選定している．実験結果においてすべてのバグを検出できたのは，この 2 つの前提が満たされていたためである．

しかし，この 2 つの前提が片方でも成り立たない場合，提案手法は利用できない．例えば，状態遷移が隠蔽されておりデータシートに記載されていない場合は利用できない．また，データシートの記述が不十分である場合や，周辺機器自体に問題（エラッタ）がある場合は特にバグ検出が不正確になる．

本研究においては，周辺機器の操作のみをバグ検出の対象として評価している．そのため，操作した結果の値をファームウェアどのように使用しているかという点については考慮していない．エンディアンの誤りや，データの不正な使用があった場合は，提案手法では検出できない．

また，周辺機器のモデル作成が手動で行われるため，多くの工数が必要である．そのため，その工数を用意できない場合は提案手法を利用することが難しい．工数を用意できたとしても，モデル作成に誤りが生じる可能性はある．そのため，モデルの正確性を確認するための手法が必要である．

6.2 今後の課題

開発したエミュレータにおいては，バグの検出が可能なことを確認できた．しかし，実際のソフトウェア開発において使用できるほどの完成度には至っていない．そのため，今後の課題としては，以下のようなものが挙げられる．

1. **周辺機器モデル作成の自動化**

現在，周辺機器モデルは，ユーザが手動で作成する必要がある．そのため，周辺機器モデルの作成には多くの時間を要し，またモデル自体に誤りが生じやすい．先行研究として，仕様書やデータシートを自然言語処理を用いて解析し，自動的にマイクロコントローラのモデルを作成する手法が提案されている [3]．これを応用することで，周辺機器モデルの作成を自動化できると考えられる．応用する上で，周辺機器を製造する企業の数は多く，データシートに多様性があることが課題となる．

2. Fault Injection

現在，周辺機器モデルはデバイスドライバからの要求に応じて，正常な場合の挙動を再現するのみである．このため，周辺機器でデバイスドライバが原因ではないエラーが発生した場合の挙動を再現できない．意図的にエラーが発生した場合を再現する (Fault Injection) ことにより，デバイスドライバのエラーハンドリングが正しく行われているかを確認できる．

3. **他の周辺機器での実験**

本実験では，STM32 と MCP2517FD の組み合わせのみを用いて実験しており他の組み合わせにおいても同様の結果が得られるかは不明である．そのため，他の組み合わせにおいても同様の結果が得られるかを確認する必要がある．また，エミュレーションに必要な工数について，他の組み合わせと比較することも今後の課題である．

7. 結言

本研究では，デバイスドライバのデバッグ支援手法の 1 つとしてバグ検出手法を提案した．バグ検出はエミュレーションと周辺機器の状態遷移の制約を用いて行った．

提案手法の実現のため，HALucinator を拡張することで，エミュレータを作成した．また，バグの検出は単純な状態遷移の規則により行った．

有効性を確認するため，マイクロコントローラである STM32F4 と周辺機器である MCP2517FD が搭載された組み込みデバイスを対象とし実験した．実験においては，手作業によるモデルの作成により，エミュレーションおよびバグの検出に成功した．バグの検出は，単純な規則による手法ではあるが，デバイスドライバのバグの検出という観点では有効であることがわかった．また，エミュレーションがデバイスドライバのバグの検出に有効であることも確認できた．

これらの結果から，本研究で提案した手法は，デバイスドライバのバグの検出に貢献でき，組み込み機器のデバッグを支援できると言える．

謝辞

本研究を行うにあたり，研究方針の設定や研究手法の助言，本論文の作成に至るまで丁寧なご指導を頂きました，本学情報工学・人間科学系　水野修教授，崔恩瀞助教に厚く御礼申し上げます．実験で用いた組み込みデバイスの提供を頂きました，西口拓実さん，城戸浩吏さんをはじめとするロボコン挑戦プロジェクト ForteFibre の皆さんに深く感謝致します．本学情報工学専攻・渡邉紘矢先輩をはじめとする，ソフトウェア工学研究室の皆さん，学生生活を通じて著者の支えとなった家族や友人に深く感謝致します．

参考文献

[1] A. Makhshari and A. Mesbah, “Iot bugs and development challenges,” In Proceedings of the 2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE), pp.460–472, 2021.

[2] C. Wright, W.A. Moeglein, S. Bagchi, M. Kulkarni, and A.A. Clements, “Challenges in firmware re-hosting, emulation, and analysis,” ACM Computing Surveys, vol.54, no.1, pp.1–36, Jan. 2022.

[3] W. Zhou, L. Zhang, L. Guan, P. Liu, and Y. Zhang, “What your firmware tells you is not how you should emulate it: A specification-guided approach for firmware emulation,” In Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security, p.3269–3283, 2022.

[4] E. Gustafson, M. Muench, C. Spensky, N. Redini, A. Machiry, Y. Fratantonio, D. Balzarotti, A. Francillon, Y.R. Choe, C. Kruegel, and G. Vigna, “Toward the analysis of embedded firmware through automated re-hosting,” In Proceedings of the 22nd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2019), pp.135–150, Sept. 2019.

[5] A.A. Clements, E. Gustafson, T. Scharnowski, P. Grosen, D. Fritz, C. Kruegel, G. Vigna, S. Bagchi, and M. Payer, “HALucinator: Firmware re-hosting through abstraction layer emulation,” In Proceedings of the 29th USENIX Security Symposium, pp.1201–1218, Aug. 2020.

[6] 高田広章，“組込みシステム開発技術の現状と展望,” 情報処理学会論文誌，vol.42，no.4，pp.930–938，apr 2001.

[7] M. Kavre, A. Gadekar, and Y. Gadhade, “Internet of things (iot): A survey,” In Proceedings of the 2019 IEEE Pune Section International Conference (PuneCon), pp.1–6, 2019.

[8] M. Muench, J. Stijohann, F. Kargl, A. Francillon, and D. Balzarotti, “What you corrupt is not what you crash: Challenges in fuzzing embedded devices,” In Proceedings of the 25th Network and Distributed System Security Symposium 2018, pp.1–15,

Feb. 2018.

[9] L. Ryzhyk, A. Walker, J. Keys, A. Legg, A. Raghunath, M. Stumm, and M. Vij, “User-Guided device driver synthesis,” In Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14), pp.661–676, Oct. 2014.

[10] D. Monniaux, “Verification of device drivers and intelligent controllers: a case study,” In Proceedings of the 7th ACM & IEEE International Conference on Embedded Software, p.30–36, Sept. 2007.

[11] S.N. Laboratories, HALucinator, （オンライン）, 入手先〈https://github.com/sandialabs/halucinator〉（参照 2023-01-18）.

[12] STMicroelectronics, STM32F4 Series, （オンライン）, 入手先〈https://www.st.com/en/microcontrollers-microprocessors/stm32f4-series.html〉（参照 2023-01-18）.

[13] M.T. Inc., MCP2517FD External CAN FD Controller with SPI Interface, （オンライン）, 入手先〈https://www.microchip.com/en-us/product/mcp2517fd〉（参照 2023-01-18）.

[14] M.T. Inc., MCP2517FD External CAN FD Controller with SPI Interface, （オンライン）, 入手先〈https://ww1.microchip.com/downloads/en/DeviceDoc/MCP2517FD-External-CAN-FD-Controller-with-SPI-Interface-20005688B.pdf〉（参照 2023-01-18）.