信頼性向上を目的とした組込みソフトウェア向けプログラミング言語の開発

岡山 直樹[1)]・片山 徹郎[2)]

Development of a Programming Language to Improve Reliability of Embedded Software

Naoki OKAYAMA, Tetsuro KATAYAMA

Abstract

In recent years, embedded systems has become high-performance larger scale, therefore embedded software has become more complex. On the other hand, embedded software is required high-quality in development. So, this paper develops a programming language for embedded software development to improve reliability of embedded software. Specifically, the developed language has a function which can check a model at the time of compile and run-time, and has the relation of 1 to 1 between the state transition table and state chart diagram. Hence, it is expected to improve the reliability of embedded software.

Key Words:
Embedded software, Programming language, Compiler, State chart diagram

1. はじめに

近年，組込みシステムの高機能化や大規模化に伴い，それに搭載される組込みソフトウェアも複雑化している。しかし，複雑にも関わらず，組込みソフトウェアの多くは，いったん起動されると止まることなく期待された動作を続けることを求められる。また，汎用コンピュータのアプリケーションの場合は，修正プログラムを配布して不具合の修正を行うことができるが，組込みソフトウェアの場合はそういった手段は使えないことが多い[1)]。そのため，組込みソフトウェアは開発の段階で高い品質が求められる。

現在，組込みソフトウェアの多くは，一般的にC言語で開発される[1)]。C言語は開発者にとって使い易い開発言語であり，実際に多くの開発者に使用されている。しかし，C言語はコンパイル時や実行時の型のチェック機能が弱く，機能安全においては推奨できない言語，という評価がなされている[2)]。

1) 情報システム工学専攻大学院生
2) 情報システム工学科准教授

また，大規模な組込みソフトウェア開発において，従来のようなソースコードを中心とした開発では，品質，コスト，納期のいずれかを満たせなくなる可能性がある。そのため，現在のプログラミング言語から，UML(Unified Modeling Language)や状態遷移図表モデルといった抽象度の高いモデリング言語を用いたモデルベース開発を行う必要がある[3)]。

そこで本稿では，組込みソフトウェアの信頼性向上を目的とし，組込みソフトウェア開発向けプログラミング言語の開発を行う。具体的には，コンパイル時や実行時の型のチェック機能を強くし，さらに，状態遷移表や状態遷移図との対応関係が一対一となる言語を作成する。

2. 開発言語の特徴

2.1 型サフィックス

組込みソフトウェア開発に一般的に使われているC言語は，暗黙的な型変換を許している[1)]。このため，C言語でコーディングを行うと，誤った型の値を代入してしまう場合がある。このようなコードでバグが発

```
例:
  byte8 hogeB8 = 0B8;   //初期化
  hogeB8 = 10B8;
```

図 1: 型サフィックスの例

```
例:
  typedef byte3 B3 unsigned bit(3)
  typedef byte8 B8 unsigned bit(8)
  typedef long32 L32 signed bit(32)
```

図 2: 型の宣言の例

生した場合，バグの発生箇所が見つけづらいため，デバッグは困難な作業となる。

そこで，開発する言語では暗黙的な型変換を禁止する。しかし，暗黙的な型変換を禁止しただけではコード上で変数がどの型を取っているのか，また定数がどの型を取っているのか判断ができない。このため，開発する言語では変数や定数に，どの型かを表すサフィックスを付ける。図 1 に，型サフィックスの例を示す。B8 とは 8 ビットの範囲を取る byte8 型を表す型サフィックスである。

この型サフィックスを用いることによって，コード上で変数や値がどの型を取っているのかが明示的に分かる。そのため，開発者が変数の型を間違うといったバグを防ぐことができる。

2.2 型の宣言

C 言語では，変数の型は予め用意されている。しかし，処理系によって型のサイズが違う場合がある。そのため，開発者は予め開発に用いる処理系の型のサイズを調べる必要がある。また，既存のソースコードを別の処理系でコンパイルした場合，開発者の予期せぬバグが発生する可能性がある。

そこで開発する言語では，予め型を用意せず，コード中で型を宣言できるようにする。

型の宣言の例を，図 2 に示す。型の宣言は typedef 文を用いて行う。typedef に続けて，型名，型サフィックス，signed なのか unsigned なのか，型の取るビット数を指定する。

コード中で型を宣言することによって，開発者が型のサイズをコードのみで把握することができ，また，移植時に処理系の型のサイズの違いによるバグを防ぐことができる。

2.3 変数の値チェック

C 言語では，変数がオーバーフロー，もしくはアンダーフローをした場合でも，実行が継続される。そのため，プログラマが間違ってオーバーフロー，もしくはアンダーフローを起こすようなプログラムを作成してしまった場合，予期せぬ動作をしたり，無限ループに陥ることがある。

```
例:
  typedef byte8 B8 unsigned bit(8)

  byte8 hogeB8 = 0B8;   //初期化

  //byte8型の値の範囲を超えるのでエラー
  hogeB8 = 128B8 + 196B8;
```

図 3: 変数の値チェックの例

そこで開発する言語では，演算終了後に変数の値をチェックし，オーバーフロー，もしくはアンダーフローをしていた場合，プログラムはエラー終了するようにする。

変数の値チェックの例を，図 3 に示す。byte8 型は符号無し 8 ビットの整数型として定義している。つまり，byte8 型の変数 hogeB8 は 0〜255 の範囲しか扱えないため，それを超える範囲の値を代入しようとする場合には，エラーとなりプログラムはエラー終了する。

2.4 状態遷移構文

組込みシステムは，多くの状態を持つ。そのため，組込みソフトウェアでは，状態遷移表や状態遷移図を用いて設計やテストを行う場合ことが重要となる [3)]。しかし，C 言語には，状態遷移表や状態遷移図と一対一となるような構文はなく，状態を遷移させるためには開発者が独自に状態を遷移させるためのコードを記述しなければならない。これは状態遷移の数を多く持つシステムの場合，バグが増える要因となる。

そこで開発する言語では，状態遷移表や，状態遷移図と一対一の関係を持つ状態遷移構文を持つ。状態遷移構文を用いた例を，図 4 に示す。このコードで記述した状態遷移は，図 5 の状態遷移図と同じである。

状態遷移構文は，システムの各状態を表す状態ブロックの集まりから構成する。状態ブロックは，状態遷移図の各状態と対応関係を持ち，action ブロックと event ブロックを持つ。action ブロックは，状態遷移図の各状態の動作と対応関係を持ち，その状態で行いたい動作をするタスクの起動処理を行う。event ブロックは，態遷移図の各状態間の遷移と対応関係を持ち，次の状態へ遷移するためのイベントを記述する。event ブロックで使用するイベントは，event 構文で定義する。event 構文では，イベント名とイベントが発生する条件を指定する。

状態遷移構文を用いることによって，状態遷移表や

```
例:
  task task1(){
    while(1B8){
      event;
    }
  }

  task task2(){
    while(1B8){
      event;
    }
  }

  //event構文
  event{
    b1_down[(io.p5drB8 & 0x01B8) == 0x00B8];
    b2_down[(io.p5drB8 & 0x04B8) == 0x00B8];
  }

  //状態遷移構文
  state{
    //状態ブロック
    s1{
      //actionブロック
      action{
        t1.start;
      }
      //eventブロック
      event{
        b1_down{
          next(s2);
        }
      }
    }

    s2{
      action{
        t1.start;
      }
      event{
        b2_down{
          t2.start;
        }
      }
    }
  }
```

図 4: 状態遷移構文の例

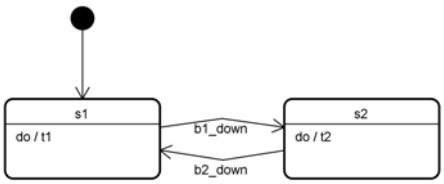

図 5: UML で記述した状態遷移図（ステートチャート図）

状態遷移図から容易に状態を遷移するプログラムが記述できる。また，プログラムから容易に状態遷移表や状態遷移図を作成できる。

3. 実装方法

開発した言語の特徴である，型の宣言，変数の値チェック，状態遷移構文の実装方法について述べる。

3.1 型の宣言

型は，型テーブルによって管理する。型テーブルは，型に必要な情報をまとめたものである。コンパイラは typedef 文を解析し，型テーブル構造体に値をセットし，型テーブルに登録する。

```
typedef struct _type{
  char label[128];
  char suffix[64];
  int size; //大きさ

  //ターゲットマシン上で実際に確保する変数サイズ
  int base_size;

  //最大値
  signed_unsigned_long max_range;

  //最小値
  signed_unsigned_long min_range;

  //signedかunsignedか is_signed==true
  bool is_signed;
} type;
```

図 6: 型テーブル構造体

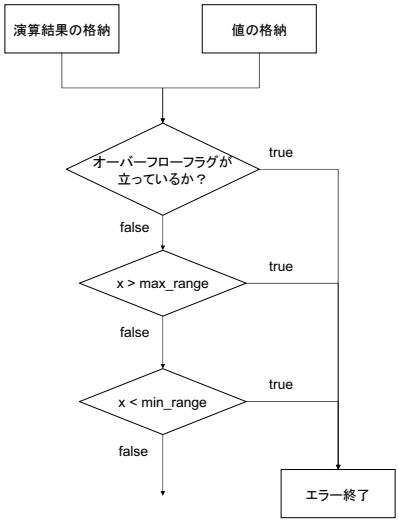

図 7: 変数の値チェックの流れ

型テーブル構造体を，図 6 に示す。型テーブル構造体は，型名，型サフィックス，型の大きさ，ターゲットマシン上で実際に確保する変数サイズ，変数の取り得る最大値，最小値，符号有りか無しかを表すフラグを持つ。ターゲットマシン上で実際に確保する変数サイズは，型を表すために必要であり，かつ，ターゲットマシン上で扱える最小の演算サイズを示す。ターゲットマシン上で実際に確保する変数サイズ，変数の取り得る最大値，最小値は，型テーブル構造体に値をセットする際にコンパイラが計算をして求める。

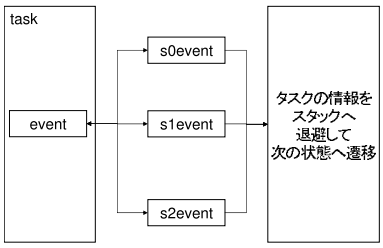

図 8: イベントによるディスパッチの流れ

3.2 変数の値チェック

変数の値チェックの流れを，図 7 に示す。演算した結果をレジスタへ格納，または値をレジスタへ格納した後，型テーブル構造体に格納されている最大値と，最小値の値について比較する。この時，もし型の表せる値の範囲を超えていた場合は，エラー終了する。

3.3 状態遷移構文

コンパイラは状態遷移構文を解析し，状態ごとに，状態の開始位置を表すラベルを付け，そのラベルにジャンプすることで状態を切り替える。event ブロックは状態ごとに関数化する。タスク中の event 文は，event ブロックを解析して作成した関数を呼び出す。イベントが発生していた場合は，ディスパッチャに処理が移り，タスクの状態を保持した後，次の状態へジャンプする。イベントによるディスパッチの流れを，図 8 に示す。

4. 実行例

開発した言語のコンパイラを実装し，実際に開発した言語を使用して，プログラムが作成できることを検証した。コンパイラの開発環境を，表 1 に示す。また，作成したプログラムの動作を確認するために，テスト基盤を作成した。テスト基盤の回路図を，図 9 に示す。

4.1 型の宣言と変数の値チェック

型の宣言と，変数の値のチェックが正しく動作することを確認するために，今回開発した言語で記述したサンプルプログラム 1 を作成した。図 10 に，サンプルプログラム 1 のソースコードを示す。

サンプルプログラム 1 は，符号なし 8 ビット整数型である byte8 型を宣言し，その型を用いて LED を点灯させるプログラムである。テスト基板上に LED は赤，紫，緑の 3 つがある。その LED を赤，紫，緑の順

表 1: 開発環境

開発 OS	WindowsXP + Cygwin
開発言語	C++
ターゲットマシン	AKI-H8/3069F
アセンブラ	h8300-hms-asm
リンカ	h8300-hms-ld
モトローラ S フォーマット 形式への変換	h8300-hms-objcopy
書き込みソフト	h8write
モニタ	TOPPERS プロジェクト H8 用簡易モニタ

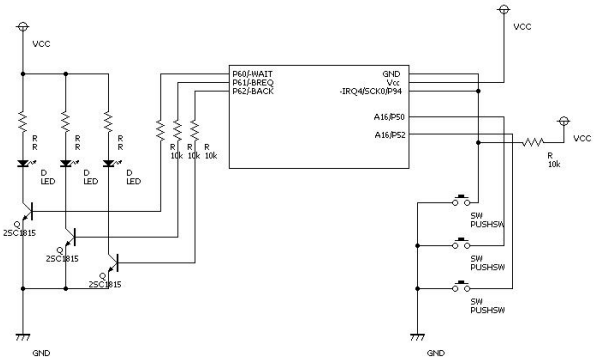

図 9: テスト基盤回路図

番に点灯させる。LED を点灯させるたびに byte8 型の変数 hogeB8 に 128 を足していく。byte8 型の変数は 0〜255 の範囲の値しか扱えないため，12 行目でオーバーフローとなりエラー終了し，LED は紫が点灯した状態でプログラムは終了することになる。

サンプルプログラム 1 をコンパイルし，実行した結果，紫の LED が点灯した状態で停止した。このことから，型の宣言ができ，実行時に型の範囲を超える演算を行った場合エラー終了することが分かる。よって，型の宣言と変数の値チェックが正しく動作していると言える。

4.2 状態遷移構文

状態遷移構文が正しく動作するかを確認するために，図 11 に示すような状態遷移を持つ，サンプルプログラム 2 を作成した。サンプルプログラム 2 のソースコードの一部を，図 12 に示す。

サンプルプログラム 2 は，状態を 4 つ持つ。状態 1 は LED 赤を点灯，状態 2 は LED 紫を点灯，状態 3 は

```
typedef byte8 B8 unsigned bit(8)

main{
  byte8 p6ddrB8@0xfee005A = 0xffB8;
  byte8 ledB8@0xffffd5A = 0x00B8;
  byte8 hogeB8 = 0B8;

  ledB8 = 1B8;
  hogeB8 = 128B8 + hogeB8;
  ledB8 = 2B8;
  //エラーになり以下の処理は実行されない
  hogeB8 = 128B8 + hogeB8;
  ledB8 = 4B8;
}
```

図 10: サンプルプログラム 1

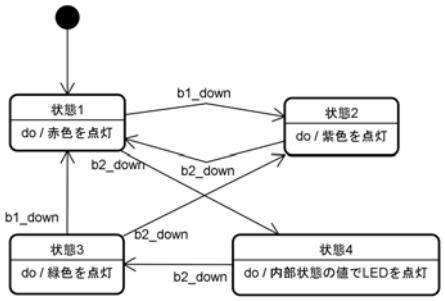

図 11: サンプルプログラム 2 の状態遷移図

LED 緑を点灯，状態 4 はタスク内に保持されている変数の値に対応した LED を点灯する。タクトスイッチ 1 が押されると変数に値を 1 加える。イベント b1_down はタクトスイッチ 1 が押されたことを表し，イベント b2_down はタクトスイッチ 2 が押されたことを表す。

サンプルプログラム 2 をコンパイルし，実行した結果，図 11 の状態遷移図で表した通りに状態が遷移することを確認した。このことから，状態遷移構文が正しく動作していると言える。

5. 考察

今回開発した言語と，組込みソフトウェア開発で一般的に使用されている C 言語とを比較し，開発した言語の特徴である，型の宣言，変数の値チェック，状態遷移構文により信頼性が向上したかどうかを考察する。

5.1 型の宣言，変数の値のチェック

C 言語では型のサイズは処理系によって違う場合があり，移植時にコンパイルが通ったとしても，型のサイズの違いにより予期せぬバグが発生する可能性がある。また，C 言語では実行時に変数の値に対しては何のチェックも行われないため，オーバーフローやアンダーフローを起こした場合でも気が付かない場合がある。

これに対して，今回開発した言語では，コード中で

```
event{
  b1_down[(io.p5drB8 & 0x01B8) == 0x00B8];
  b2_down[(io.p5drB8 & 0x04B8) == 0x00B8];
}

state{
  s1{
    action{
      t1.start;
    }

    event{
      b1_down{
        next(s2);
      }

      b2_down{
        next(s4);
      }
    }
  }

  s2{
    action{
      t2.start;
    }

    event{
      b2_down{
        next(s1);
      }
    }
  }

  s3{
    action{
      t3.start;
    }

    event{
      b1_down{
        next(s1);
      }

      b2_down{
        next(s2);
      }
    }
  }

  s4{
    action{
      t4.start;
    }

    event{
      b2_down{
        next(s3);
      }
    }
  }
}
```

図 12: サンプルプログラム 2

開発者が独自に型を宣言でき，また，実行時に変数の値が型の範囲を超えないかをチェックできる。これにより，実行時にオーバーフローやアンダーフローを起こし，ソフトウェアが予期せぬ状態に移行したり，無限ループに陥ることがない。

このことから，型の宣言，変数の値のチェックの機能により信頼性が向上したと言える。

5.2 状態遷移構文

組込みシステムの多くは，多くの状態を持つ。これら多くの状態が複雑に遷移することによって組込みソフトウェアは動作している。

C言語には状態の遷移を直接表せるような構文はなく，開発者が独自に状態を遷移するためのコードを記述しなければならない。もし，状態遷移を行う処理にバグがあれば予期せぬ状態に移行し，ソフトウェアが正しく実行できなくなる。

これに対して，今回開発した言語では，状態遷移構文を用いることによって，状態遷移表や状態遷移図と一対一対応を持つコードを記述できる。そのため，状態遷移を行う処理を容易に記述することができる。これにより，ソフトウェアが予期せぬ状態に移行するようなバグを減少，もしくはバグの発見が容易となる。

このことから，状態遷移構文により信頼性が向上したと言える。

5.3 問題点

今回開発した言語の問題点を，以下に述べる。

- リアルタイム性の確保
  今回開発した言語は，信頼性向上のために，変数の値のチェックを行っている。このため，C言語と比べ実行時間が遅くなる。組込みソフトウェアではリアルタイム性を求められる。実行時間が遅くなると，リアルタイム性の確保が難しくなる。リアルタイム性を求められる部分に関しては，テストを念入りに行うことを前提とし，変数の値のチェックを行わないようにするといった対策をとることで，リアルタイム性を確保できるのではないかと考えられる。

- オーバーフロー，アンダーフロー時の例外処理
  開発した言語では，オーバーフローやアンダフローを起こした場合，エラー終了する。これにより，ソフトウェアが暴走することがなくなるので，信頼性は向上する。ここで，エラー終了するのではなく，適切な例外処理を行いソフトウェアを実行し続けるようにすることによって，さらに信頼性が向上すると考えられる。

- 排他制御
  開発した言語には排他制御の機能がない。このため，排他制御を行う場合は自分で排他制御の処理を書かなければならない。排他制御はマルチタスクプログラミングを行う場合，よく使用される技術であり，排他制御を言語仕様として実装することで，より利便性が向上すると考えられる。

- 状態遷移表，状態遷移図からコードの自動生成
  開発した言語は，状態遷移表や状態遷移図と一対一の対応を持つ状態遷移構文がある。そのため，状態遷移表や状態遷移図から状態遷移するコードを自動で生成することが容易であると考えられる。コードを自動で生成することで，人為的なミスによるバグをさらに防ぐことが期待できるので，より信頼性が向上すると考えられる。

6. 終わりに

本稿では，組込みソフトウェアの信頼性向上を目的とし，組込みソフトウェア開発向けプログラミング言語の開発を行った。今回開発した言語を使用して，実際にプログラムが作成できることを検証した。また，今回開発した言語の有効性を確認するために，C言語と比較した。

その結果，開発した言語では実行時に型を宣言，変数の値をチェックすることで，変数の値が型の範囲を超えた場合エラー終了するため，ソフトウェアが予期せぬ状態に移行したり，無限ループに陥ることがない。このため，信頼性が向上したと言える。また，状態遷移構文により，状態遷移表や状態遷移図と一対一の対応を持つコードを記述できる。そのため，状態遷移を行う処理を容易に記述することができる。これにより，人為的なバグを減少することができ，信頼性が向上したと言える。

今後の課題を以下に述べる。

- リアルタイム性の確保
- オーバーフロー，アンダーフロー時の例外処理
- 排他制御
- 状態遷移表，状態遷移図からコードの自動生成

参考文献

1) SESSAME『組込み現場の「C」プログラミング基礎からわかる徹底入門』技術評論社 (2007)

2) MISRA-C研究会『組込み開発者におくるMISRA-C 2004 C言語利用の高信頼化ガイド』 日本規格協会 (2006)

3) 渡辺 政彦『日経エレクトロニクス2005年10月10日号 NETs 組み込みソフトウエアの検証手法（1）モデル検査やレビューなど7つの手法を俯瞰』 日経BP社 (2005)

4) 秋月電子通商『AKI-H8/3069F フラッシュマイコンLANボード：組み立てキット』
⟨http://akizukidenshi.com/catalog/g/gK-00168/⟩