修士学位論文

題目

データ競合を検出するための割込み自動生成

指導教員
井上 克郎 教授

報告者
東 誠

平成 22 年 2 月 8 日

大阪大学 大学院情報科学研究科
コンピュータサイエンス専攻 ソフトウェア工学講座

平成 21 年度 修士学位論文

データ競合を検出するための割込み自動生成

東 誠

内容梗概

発生タイミングが判らないイベントを処理するために，割込みを利用するソフトウェアがある．このようなソフトウェアは割込みが発生した時，割込みハンドラという関数を呼び出して処理を行う．この時，割込みハンドラ以外の処理が利用している途中のメモリの値を割込みハンドラが変更すると，予期しないメモリの値によってソフトウェアに障害が発生することがある．このような障害の検出を目的として，本研究では割込みの自動生成とメモリの値の設定を行う手法を提案する．この手法では，割込みハンドラとそれ以外の処理がメモリの値を共有する可能性のある全てのタイミングを網羅するために，ソフトウェアがメモリにアクセスした直後に割込みを自動的に生成する．また，障害を発生させる割込みハンドラの動作に必要なメモリの値があれば，その値がユーザを指定することによって，指定されたタイミングでメモリの値を設定することができる．提案手法を実際のオペレーティングシステムに適用したところ，データ競合の発生に必要な事前条件さえプログラムの実行前に設定すれば，割込みの生成タイミングを手動で調整することなしに障害が再現できることが判明した．

主な用語

データ競合
割込み
フォールトインジェクション

目 次

1　はじめに

タイマ処理やI/O処理など，発生タイミングが判らないイベントを処理するために，割込みをタイマやI/Oデバイスから受け付けるプログラムがある．割込みとは，プログラムに現在実行中の処理を中断させ，より優先度の高い処理を実行する要求である．割込みによって実行される処理やそれを行うための関数は割込みハンドラと呼ばれる．また，本研究では割込みによって中断される処理やそれを行うための関数を通常処理と呼ぶことにする．割込みは通常処理の動作に関係なくいつでも発生する可能性があり，発生すると割込みハンドラを実行してから，通常処理で割込みが発生した箇所に戻る．

この割込みによって発生する欠陥の１つとして，割込みハンドラと通常処理の間で存在するデータ競合がある．データ競合とは，同じメモリに対する複数のアクセスが適切に管理されていないために，ある処理が利用しているメモリの値が，意図しないタイミングで他の処理によって変更されてしまうことである [14]．データ競合によって変更されたメモリの値はプログラムで予期されていないため，その値によってプログラムが誤った動作をすることがある．割込みハンドラと通常処理の間に存在するデータ競合の例としては，条件分岐で配列の添字の値を確認した後で配列を利用する処理を行うプログラムで，条件分岐の直後で割込みが起こり，添字の値が割込みハンドラによって書き換えられた場合が挙げられる．この時，条件式の評価時に利用された添字の値と異なる値が配列の利用時に添字に格納されているため，バッファオーバーランなどを起こす可能性がある．本論文では，単にデータ競合と書いた時には，割込みハンドラと通常処理の間で存在するデータ競合を指すものとする．データ競合は割込みを利用するプログラムにしばしば存在し，深刻な障害を引き起こすことがある．例えば，データ競合による障害の例としては放射線医療機器の事故 [11] が挙げられる．そのため，割込みは注意深く利用され，割込みを用いるプログラムは充分にテストされなければならない．

しかし，このテストを行う際にはプログラムへの入力について問題がある．割込みを利用しないプログラムをテストする場合，ユーザは障害が起こる可能性の高い入力として関数の引数やグローバル変数の値を系統的に与えることによって，それらに対して正しい出力をプログラムが返すことを確認する．しかし，割込みを利用するプログラムをテストする場合，さらに２つの問題が発生する．１つは，データ競合を検出するには，割込みハンドラと通常処理の組をテストする必要があることである．もう１つは，通常処理の中で割込みが発生するタイミングが膨大なことである．そのため，このような組やタイミングの中から，障害を起こす可能性が高いものを系統的に選んで入力とする必要がある．これらのうち，本研究では割込みが発生するタイミングの問題に対処する．

このような割込みの発生タイミングを入力として与えるテスト手法の１つとして，既存研

究としてランダムなタイミングで割込みを発生させる Regeh の手法 [15] が挙げられる．この手法では割込みを発生させる頻度を調整することが可能なため，ユーザはテストしたい状況に応じて割込みを発生させることができる．また，実際に割込みを発生させて障害を探すので，モデル検査に基づく静的な手法 [5, 8, 13] に比べ，実際の製品では発生しない障害を検出する可能性が低いという利点もある．

しかし，データ競合の検出を目的とした場合，この Regeh の手法には 2 つの問題点がある．1 つは，割込みの発生タイミングがランダムであるため，データ競合を検出するのに必要なタイミング全てで割込みを発生させている保証がないことである．データ競合による障害が発生するのは，通常処理がメモリの値を 2 回アクセスしている間で割込みが発生する場合である．このうち，1 回目のアクセスはメモリの値の読み込みか書き出しであり，2 回目のアクセスはメモリの値の読み込みである．このような状況でデータ競合による障害が発生する理由は，このようにメモリの値を利用する場合，それらの間で同じ値が利用できることが想定される場合があるからである．その値を割込みハンドラが書き換えると，想定されていない値を通常処理が利用するという障害が発生する．しかし，割込みをランダムに発生させると，この条件を満たす箇所全てで割込みが発生している保証がないため，発見できないデータ競合が存在する可能性がある．もう 1 つは，単に割込みハンドラを呼び出しただけでは，データ競合による障害を発生させるのに必要なメモリの書き換えを割込みハンドラが行う保証がないことである．例えば，割込みハンドラにプログラムで使われる変数の値やデバイスから受け取る入力値を用いた条件分岐がある場合，変数の値や入力値を操作することによって適切な分岐を行うことがデータ競合の検出には必要である．また，配列の範囲外にアクセスする障害を検出したい場合，適切な配列の添字が必要になる．

そこで本研究では，データ競合の検出を目的として，割込みハンドラを制御する手法を提案する．この手法は，割込みハンドラを呼び出すタイミングを制御する割込み発生機構と，割込みハンドラの処理内容を制御する値すり替え機構から成り立つ．これらの機構のうち，割込み発生機構は，割込みハンドラと通常処理がメモリの値を共有する可能性のある全てのタイミングを網羅するために，ソフトウェアがメモリにアクセスした直後に割込みを発生させる．そうすることにより，任意の 2 つのメモリアクセス間で割込みを発生させることができる．また，値すり替え機構は，メモリの値やデバイスからの入力値をプログラムの実行前にユーザが指定した値とすり替えるため，データ競合による障害の発生に必要なメモリの値やデバイスからの入力値を割込みハンドラに処理させることができる．

本研究では，これらの機構を CPU エミュレータ上に実装する．その理由は，CPU エミュレータはソフトウェアのテストで広く用いられているからである．例えば，組み込みシステムの開発では，しばしばハードウェアと同時にソフトウェアを開発するため，ハードウェアが存在しない時点での開発には，CPU エミュレータが用られる．

以降，2 節では本研究で対象とするデータ競合をサンプルプログラムを用いて説明する．3 節ではデータ競合の検出を目的として割込みハンドラを制御する手法を提案し，4 節では提案手法を実装した CPU エミュレータについて説明する．5 節では提案手法によってデータ競合を検出するのに必要な作業を調べるために行った実験について説明し，その実験結果について議論を行う．6 節では関連研究を紹介する．7 節で本研究の結論と今後の課題について述べる．

```
 1: #define MAX 16
 2: unsigned int index = 0;
 3: int array[MAX];
 4: void array_print(void);
 5: {
 6:   if(index < MAX)
 7:     printf(“%d¥n”, array[index]);
 8: }
 9:
10: void interrupt_handler(DEVICE *dev){
11:     index+=get_int(dev);
12: }
```

図 1: データ競合が存在するサンプルプログラム

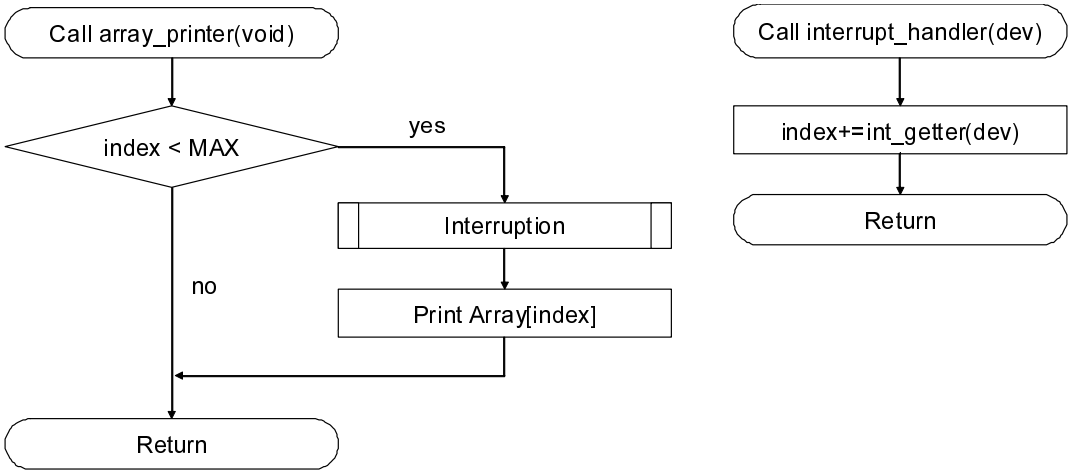

図 2: 図 1 で条件分岐の直後に割込みが発生した場合のフローチャート

2 データ競合

本節では，本研究が検出の対象とする，割込みハンドラと通常処理の間に存在するデータ競合を，例を用いて説明する．

図 1 にデータ競合が存在するソースコードの例を示す．このソースコードのうち，array_printer 関数は，添字 index の範囲が上限以下であること確認した後に，配列 array の要素を出力する手続きである．また，interrupt_handler 関数は割込みによって起動される手続きであり，配列 array の添字をデバイスからの入力値分だけ増加させるこの時，interrupt_handler 関数はデバイスからの入力値を取得するために，int_getter 関数を利用する．int_getter 関数は，dev という構造体に対応するデバイスから入力値を取得し，整数型の値

を返す手続きである．

array_printer 関数は添字の範囲を確認した上で配列にアクセスしているので，この手続きだけを見るとバッファオーバーランの危険はないように見える．しかし実際には，条件分岐の直後に割込みハンドラが呼出された場合，index が array の添字の上限を越えてしまう可能性がある．具体的には，図 2 で index の値が配列 array の添字の上限未満であることを確認した直後に割込みが発生した場合，interrupt_handler 関数によって index に int_getter 関数の返り値が加算されることで，index の値が配列 array の添字の上限を越えてしまう．

このように，割込みを考慮していないプログラムの場合，ある変数から値を読み出したり，変数に値を書き込んだ後に，その変数から値を読み込む時に，最初に用いた値と同じ値が得られることを暗黙のうちに仮定している場合がある．しかし，割込みハンドラが非同期に動作することによって変数の値が書き換えられると，この仮定が成り立たなくなるため，プログラムが正しく動作しない可能性がある．

そこで，本研究では，以下の条件が全て成立する時に，メモリの値が予期しないタイミングで変更されるという欠陥をデータ競合と定義し，これを検出することを目標とする．

- 同じメモリアドレスに対する 2 つのアクセスがある．それらのうち 1 回目はメモリの値を読み込むか書き出しており，2 回目はメモリの値を読み込んでいる
- それらのアクセスでは同じメモリの値が利用できることを仮定している
- それらのアクセスの間で割込みが発生し，それらのアクセスが利用するのと同じメモリの値を割込みハンドラが変更する

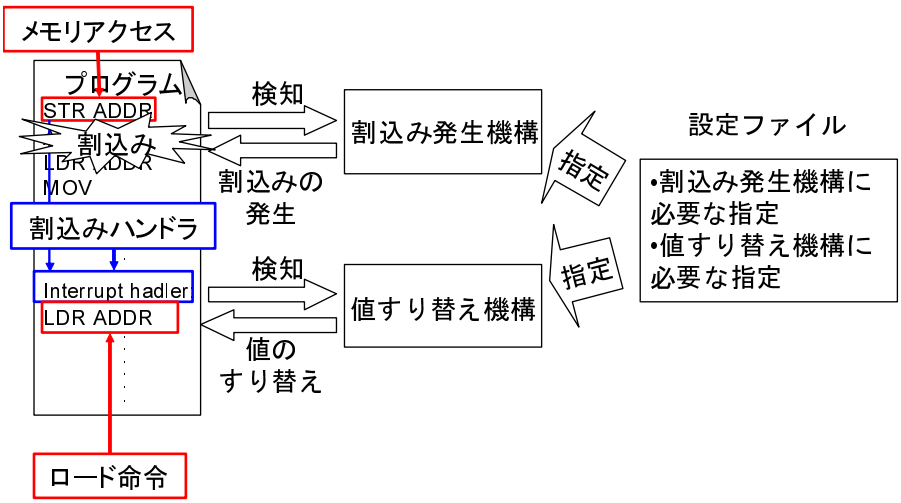

図 3: 割込みハンドラ制御手法によってデータ競合を検出するまでの過程

3　割込みハンドラ制御手法

本節では，CPU エミュレータに割込み発生機構と値すり替え機構を加えることで割込みハンドラを制御し，組込みソフトウェアのデータ競合を効率的に検出する手法について説明する．

2 節で説明したデータ競合が存在する条件から，データ競合を検出するにはプログラムがメモリに対する読み出しや変数への書き込みを行った後で割込みを発生させることと，データ競合による障害の発生に必要なメモリの値を割込みハンドラに書き換えさせることという 2 つの課題を解決する必要がある．

そこで，これらの課題のうち，前者を割込み発生機構，後者を値すり替え機構がそれぞれ対処する．これらの機構はそれぞれ独立して動作して異なる作業を行うが，データ競合の効率的な検出という共通の問題の解決に貢献する．

これらの機構を用いてデータ競合を検出するまでの過程を図 3 に示す．各機構の入力はプログラムの状態と，各機構の利用に必要な指定をユーザが記述した設定ファイルである．これらの機構のうち，割込み発生機構がプログラムのメモリに対するアクセスを検知し，設定ファイルの内容に基づいて割込みを発生させる．そして，その割込みによって呼び出された割込みハンドラがメモリから値を読み出す命令を実行した際に，それを値すり替え機構が検知し，読み出す値を設定ファイルによって指定された値とすり替える．こうすることによって，割込みハンドラから元の処理に戻った時にデータ競合による障害をを発生させることができる．

以降，各機構の詳細について説明する．

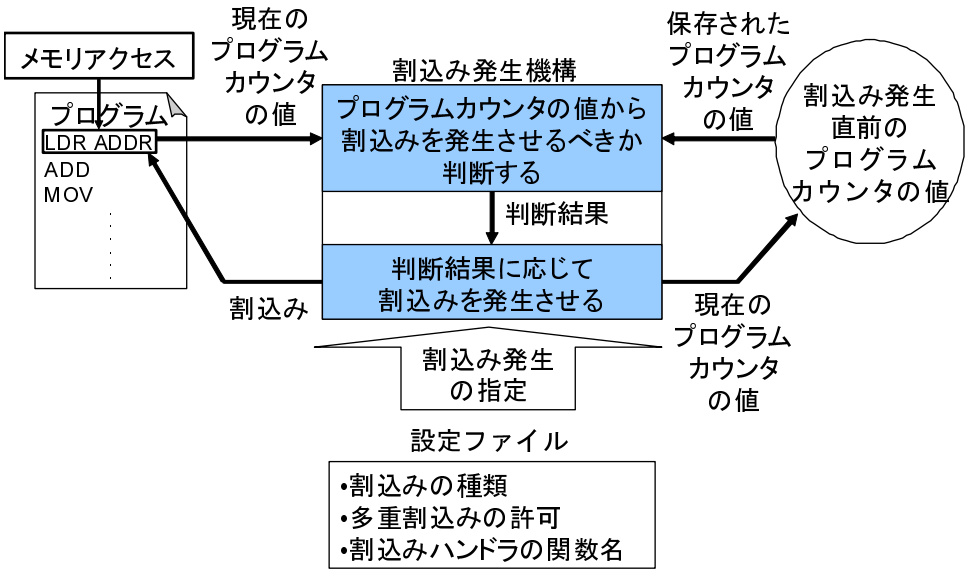

図 4: メモリアクセスの直後で割込みを生成する手順

3.1 割込み発生機構

本項では，割込み発生機構について説明する．この機構は，プログラムがメモリに対する読み出しや変数への書き込みを行ったことを検知し，その直後に割込みを発生させる．

この機構を実装する箇所は CPU エミュレータ上でメモリにアクセスする命令が実装されている箇所である．その理由は，この機構はデータ競合による障害が発生する可能性のある全てのタイミングで割込みを発生させるために，メモリアクセスの直後に割込みを発生させるからである．

割込み発生機構の要件は，以下の通りである．

- メモリアクセスの直後に割込みを発生させる必要がある．
- プログラムが割込みハンドラから戻ってきた直後に割込みを発生させてはならない．このような割込みを発生させた場合，プログラム中の同じ箇所で何度も割込みハンドラが呼び出されるため，プログラムの実行が進まなくなってしまう．
- 割込みハンドラを再帰的に呼び出すかをユーザに選択させる．このような呼び出しを行った場合，プログラムの実行が進まなくなってしまう．ただし，ユーザが行いたいテスト内容によってはこのような呼び出しも必要になるため，オプションとして利用できるようにする必要がある．

これらの要件に従って，割込み発生機構が割込みを発生させる手順の概要を図 4 に示す．

図 4 のように，割込み発生機構の入力は CPU エミュレータで利用されるプログラムカウンタの値と，CPU エミュレータの実行前にユーザが記述した設定ファイルである．これらのうち，プログラムカウンタの値は，プログラムが割込みハンドラから戻ってきた直後であるかを判断するために必要になる．また，設定ファイルには，以下の項目が記述されている．

割込みの種類 割込み発生機構によって発生させる割込みの種類．この種類の例としては，通常処理との間にデータ競合が存在するかをテストしたい割込みハンドラに対応する I/O デバイスの番号が挙げられる．

多重割込みの許可 割込みハンドラが実行されている間に割込みを発生させるかを選ぶオプション．

割込みハンドラの関数名 CPU エミュレータ上で動作するプログラムで呼び出される割込みハンドラの関数名．この関数名から，割込みハンドラの呼び出し中にプログラムカウンタが取りうる値の範囲を調べる．なお，この指定は多重割込みを禁止するために利用するので，多重割込みを許可するオプションを利用する際には必要ではない．

これらの入力を用いて，割込み発生機構が以下の手順で処理を行うことにより，全てのメモリアクセスの直後で割込みを自動的に発生させることが可能になる．

1. 命令がメモリアクセスかを確認する
2. 命令がメモリアクセスなら，現在のプログラムカウンタの値から，割込みを発生させるか判断する
3. 判断結果に従って，割込みを発生させる．なお，この時に割込み発生直前のプログラムカウンタの値を保存しておき，それをステップ 2 での判断に利用する

この時，ステップ 2 で割込みを発生させるか判断する基準を表 1 に示す．表 1 で示す判断基準の根拠は，以下のようになっている．

まず，割込みハンドラを呼び出していない時には，現在のプログラムカウンタの値が，割込み発生直前のプログラムカウンタの値と一致する場合には，割込みを発生させない．その理由は，このような状況はプログラムが割込みハンドラから戻ってきた直後だと考えられるからである．

次に，割込みハンドラを呼び出している時には，多重割込みを許可するオプションがあるかどうかで判断基準が異なる．多重割込みが許可されているなら，他の条件に関係なく割込みを発生させる．また，多重割込みが禁止されており，現在のプログラムカウンタの値が割

込み発生直前のプログラムカウンタの値と異なる場合は，多重割込みが禁止されているので割込みを発生させない．

しかし，多重割込みが禁止されており，現在のプログラムカウンタの値が割込み発生直前のプログラムカウンタの値と一致する場合は，この状況が発生した回数に依存する．というのも，多重割込みが禁止されているにもかかわらず，以前に割込みハンドラの呼び出し中に割込みを発生させたということは，割込み以外の方法で割込みハンドラを呼び出していると考えられるからである．そのため，現在のプログラムカウンタの値と割込み発生直前のプログラムカウンタの値が一致する状況のうち，1 回目は割込みによって呼び出された割込みハンドラの実行中，2 回目はその割込みハンドラから戻ってきた直後だと考えられるため，割込みを発生させない．そして，3 回目なら通常の処理でメモリアクセスが行われた直後と同様に割込みを発生させる．以後，$3n-23n-1, 3n$(n は 2 以上の自然数) 回同じ状況が発生するごとに，同様の処理を行う．

3.2 値すり替え機構

本項では，値すり替え機構について説明する．この機構は，プログラムがメモリから読み出す値を，データ競合による障害の発生に必要なメモリの値とすり替える．

この機構を実装する箇所は，CPU エミュレータでメモリから値を読み取る命令が実装されている箇所である．その理由は，この機構は割込みハンドラの処理内容を制御するために，メモリの値やデバイスからの入力値をユーザが指定した値にすり替えるからである．この時，割込みハンドラでデバイスと入出力を行う組込みソフトウェアはメモリマップド I/O を用いることが多いので，デバイスからの入力値はメモリの値と同じように扱うことができる．そのため，メモリから値を読みとる命令の実装箇所に値すり替え機構を実装すれば，デバイスからの入力値をすり替えることができる．

表 1: 割込み発生の判断基準

割込みハンドラ	プログラムカウンタの値	多重割込み	判断結果
呼び出していない	割込み発生直前と異なる	どちらでも	発生
呼び出していない	割込み発生直前と一致	どちらでも	禁止
呼び出している	どちらでも	許可	発生
呼び出している	割込み発生直前と異なる	禁止	禁止
呼び出している	割込み発生直前と一致	禁止	$3n-2$，$3n-1$ 回目は禁止，$3n$ 回目は許可 (n は自然数)

値すり替え機構の要件は，以下の通りである．

- データ競合の検出のために値を変更する必要があるとユーザが判断したメモリのアドレスを指定できるようにする．
- データ競合の検出に必要だとユーザが判断した値を注入できるようにする．この値は必ずしも定数ではなく，プログラムで利用される変数の値である場合もある．というのも，プログラムに条件分岐があり，それを行う際に評価される条件式に変数の値どうしの比較が含まれていた場合，それらの変数の値を動的に決定する必要があるからである．
- 上記のような値の注入を行う条件をユーザが指定できるようにする．このような条件は，特定の変数の値を一定以上にしたい場合などに利用する必要がある．例えば，配列の添字が上限を越えるかテストしたい場合，添字が上限を越えたにもかかわらず，その添字が値すり替え機構によって変更されてしまうと，存在するはずのデータ競合による障害が発生しなくなってしまう．

これらの要件に従って，値すり替え機構がメモリの値やデバイスからの入力値をすり替える手順の概要を図 5 に示す．

図 5 のように，値すり替え機構の入力は命令が値を読み出すメモリのアドレスと，CPU エミュレータの実行前にユーザが記述した設定ファイルである．設定ファイルには，以下の項目が記述されている．また，これらの項目の対応関係を IDEF1X 表記の ER 図で表すと，図 6 のようになる．

メモリアドレス 値をすり替えるメモリのアドレス．

算出式 すり替えるための新しいメモリの値を計算するための数式．

条件式 利用する算出式を選択する数式．各条件式が成立する際，その条件式に対応する算出式を用いて，新しいメモリの値が計算される．

これらの入力を用いて，値すり替え機構が以下の手順で処理を行うことにより，メモリの値やデバイスからの入力値を予めユーザが指定した値とすり替えることができるようになる．

1. 命令がメモリから値を読み出す命令かを確認する．
2. もしメモリから値を読み出す命令であれば，CPU エミュレータの実行前にユーザが記述した設定ファイル中の状況と，現在の実行状況が，一致するか確認する

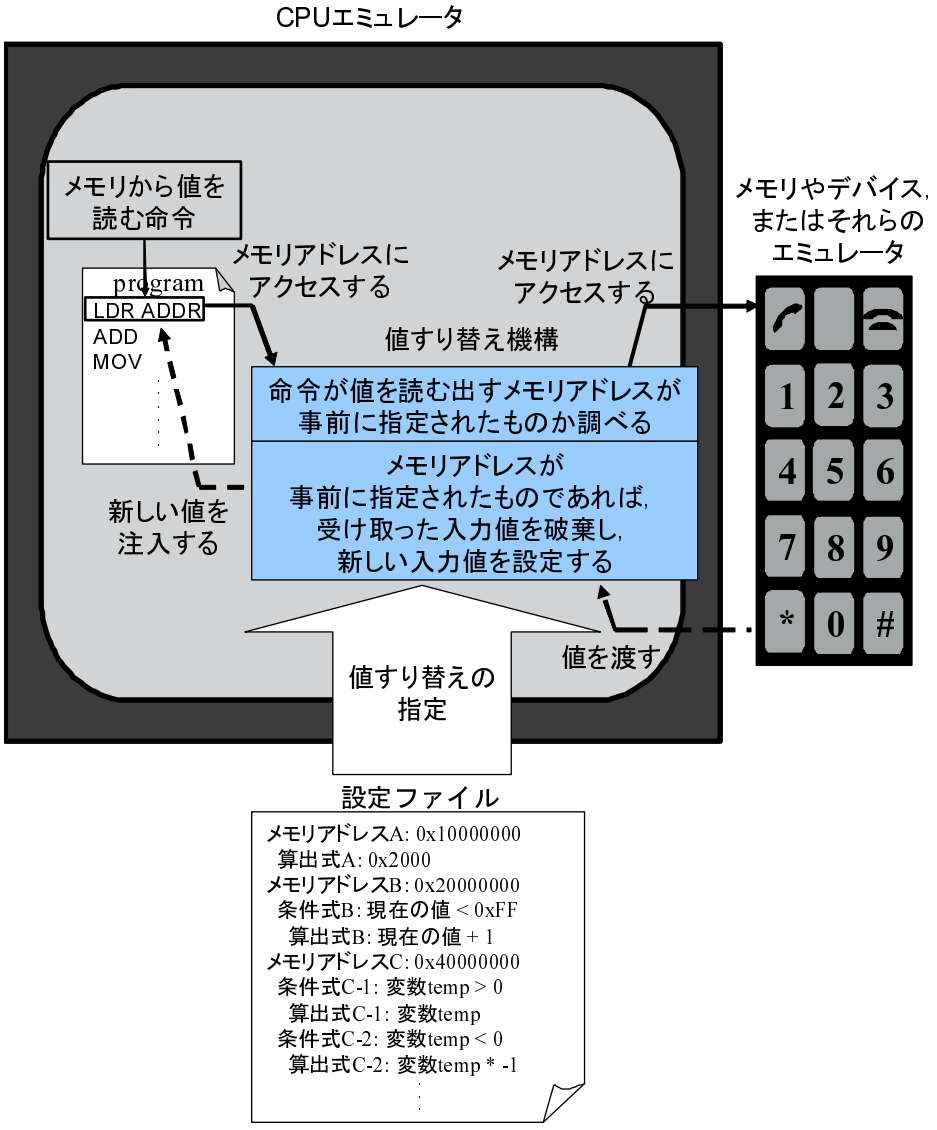

図 5: メモリの値やデバイスからの入力値をすり替える手順

3. 設定ファイルに記述された状況と一致する状況であれば，設定ファイルに従ってメモリの値を書き換える．

この時，算出式や条件式のオペランドには，定数や現在のメモリの値，プログラムで利用されているグローバル変数の値が利用可能である．このような式によって，割込みハンドラは様々なデータ処理を容易に行うことが可能になる．

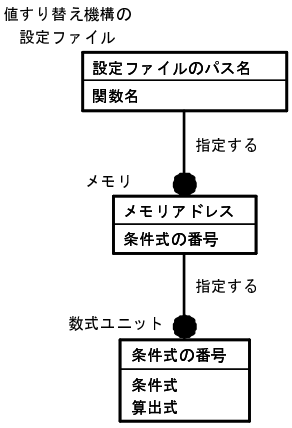

図 6: 値すり替え機構の設定ファイルに記述される項目の対応関係

4 実装

提案手法を実際のソフトウェアに適用するために，設定ファイルの解析機構と割込み発生機構，値すり替え機構を ARM アーキテクチャーの CPU エミュレータである skyeye[3] に加えた．これらの機構に必要な各機能は，以下の方法で実装した．

4.1 設定ファイルの解析機構

今回の実装では，割込み発生機構の指定と値すり替え機構の指定を同じ設定ファイルに記述した．この設定ファイルを解析する機構は，設定ファイルを木構造のデータに変換する機能と木構造のデータから指定内容を抽出する機能からなる．これらの機能が利用されるのは，CPU エミュレータの起動後かつ，その CPU エミュレータ上でプログラムを実行する前である．そして，これらの機能によって抽出した指定内容が利用されるのは，CPU エミュレータ上でプログラムを実行している間に，割込み発生機構と値すり替え機構を利用する時である．

以下，各機能を説明する．

なお，設定ファイルの書式については付録にて説明する．

設定ファイルを木構造のデータに変換する機能 設定ファイル中の算出式や条件式を値すり替え機構に利用し易くするために，設定ファイルの内容を木構造のデータに変換する関数を，yacc と lex で作成した．

木構造のデータから指定内容を抽出する機能 木構造のデータから，割込み発生機構と値すり替え機構の利用に必要な指定を抽出する関数を，C 言語で作成した．これらの指定

のうち，割込み発生機構の指定である割込みハンドラの関数名から，dwarf2[1] を用いて，割込みハンドラの呼び出し中にプログラムカウンタが取りうる値の範囲を取得した．dwarf2 を用いた理由は，dwarf2 はプログラムカウンタの範囲を取得できるだけではなく，様々なデバッグ情報を取得でき，今後の機能拡張の際に有益であるからである．また，値すり替え機構の指定は，必要な指定内容を利用し易くするために，ハッシュテーブルに格納された．このようなハッシュテーブルは 2 種類ある．1 つのハッシュテーブルでは，メモリアドレスの値をキーとして，そのメモリアドレスをロード元とするロード命令の実行時に利用される算出式と条件式の組の集合を取り出せる．もう 1 つのハッシュテーブルは，グローバル変数の名前をキーとして，その変数の値が格納されているメモリアドレスを取り出せる．このようなハッシュテーブルを用いることで，必要な指定内容を効率的に利用できるようにした．

4.2 割込み発生機構

割込み発生機構として実装した機能の利用手順は，以下の通りである．

1. メモリアクセスが実行されると，割込みの発生を判断する機能を実行する．
2. ステップ 1 で割込みを発生すると判断した場合，割込みを起こす機能を実行する．
   - ステップ 2 で割込みを起こしたにもかかわらず割込みハンドラが呼び出されなかった場合，割込みを解除する機能を実行し，ステップ 1 に戻る．
   - ステップ 2 で起こした割込みによって割込みハンドラが呼び出される場合，割込み発生直前のプログラムカウンタの値を保存する機能を実行する．そして，機械語命令が実行される度に，その割込みハンドラから戻ってきたことを確認する機能を実行する．戻ってきたことが確認できれば，ステップ 1 に戻る．

各機能の詳細は，以下のようになっている．

割込みの発生を判断する機能 表 1 に従って割込みを発生させるべきかを判断する関数を，C 言語で作成した．この機能への入力は，設定ファイルに記述された多重割込みを許可するオプションと，割込みハンドラから戻ってきたことを確認する機能の実行結果である．これらの入力を元に，表 1 に従って割込みの発生に対する判断結果を出力する．なお，プログラムカウンタの値を用いた判断は，この関数では行わずに割込みハンドラから戻ってきたことを確認する機能で行っている．このようにモジュール化することで，各機能の内容を簡略化することができた．

そして，その関数を 6 種類のロード命令と 6 種類のストア命令を実装している関数から呼び出すことで，メモリアクセスの直後に割込みを発生させることを実現した．この時，割込みの発生を判断する関数を呼び出す箇所は，その数が最も少なくて済むと同時に，従来の skyeye が行う処理を妨げない箇所を選んだ．skyeye では実行する CPU 命令を解析した後，ロード命令用関数やストア命令関数を呼び出す箇所がそれぞれ 32 箇所ある．これらの箇所の直後で割込みの発生を判断する関数を呼び出すと手間がかかるため，ロード命令用関数やストア命令関数の内部で割込みの発生を判断する関数を呼び出すようにした．

割込みを起こす機能 割込みの発生を判断する機能の出力結果を元に割込みを発生させる関数を，C 言語で作成した．この時，割込みの発生方法は CPU エミュレータで模倣している CPU 毎に異なるため，CPU の差異を意識せずに共通のインターフェイスで割込みを発生できるのが望ましい．そこで，各 CPU に対応して割込みを発生させる関数を作成した後，それらの関数から適切なものを選んで呼び出す関数を作成した．また，前者の関数はいずれも，設定ファイルに記述された割込みの種類を引数とし，CPU エミュレータで割込みの発生を管理するグローバル変数を書き換えることで割込みを発生させるようにした．

割込みを解除する機能 割込みを起こす機能によって割込みを発生させても，CPU エミュレータ上で動作しているプログラムが割込みを禁止している場合，割込みハンドラは呼び出されない．この時，発生した割込みを放置していると，プログラムが割込みを許可した瞬間に割込みを呼び出してしまう．これを防ぐため，割込み発生後に割込みハンドラが呼び出されなかった場合に割込みを解除する関数を，C 言語で作成した．この関数の作成時にも，割込みを起こす機能と同様に，各 CPU に対応して割込みを発生させる関数を作成した．この時，発生させた種類の割込みを確実に解除するために，CPU によっては，関数の引数に割込みの発生時に利用した割込みの種類を利用した．

割込み発生直前のプログラムカウンタの値を保存する機能 割込みハンドラが呼び出されるのが確定した時にプログラムカウンタの値を保存する機能を，C 言語で作成した．この機能は，正確なプログラムカウンタの値を保存する必要があるので，割込みハンドラを呼び出すためにプログラムカウンタの値が変更される直前の箇所に実装した．

割込みハンドラから戻ってきたことを確認する機能 割込みハンドラを呼び出した後に，その割込みハンドラから戻ってきたことを確認する機能を，C 言語で作成した．この確認を行う基準は，表 1 に従って，割込み発生直前のプログラムカウンタの値と現在のプログラムカウンタの値が一致している上に，以下のいずれかの条件が成立した時に

割込みハンドラから戻ってきたと判断した．

- 割込み発生直前のプログラムカウンタの値が，割込みハンドラの呼び出し中にプログラムカウンタの値が取りうる範囲に含まれていない
- 多重割込みが許可されている
- 上記以外のケースで，割込み発生直前のプログラムカウンタの値と現在のプログラムカウンタの値が一致したことが既に 1 回確認されている

4.3 値すり替え機構

値すり替え機構として実装した機能と，これらの機能の利用手順は，以下の通りである．

1. ロード命令が実行されると，ロード元を調べ，そのメモリアドレスに対応した算出式と条件式の組の集合を取り出す機能を実行する．対応した算出式と条件式の組の集合があればステップ 2 へ，なければ値すり替え機構を終了する．
2. ステップ 2 で取り出した集合中の各条件式を評価し，真になった条件式に対応する算出式を計算する機能を実行する．そして，その実行結果を新しいロード値とする．

各機能の詳細は，以下のようになっている．

メモリアドレスに対応した算出式と条件式の組の集合を取り出す機能 メモリの値をロードしたり，メモリマップド I/O によってデバイスから入力値を受け取る時に，そのロード元であるメモリアドレスに対応した算出式と条件式の組の集合を取り出す関数を，C 言語で作成した．この関数では 4.1 節で作成したハッシュテーブルを利用し，ロード元のメモリアドレスをキーとして必要な算出式と条件式の組の集合を取り出した．このようなハッシュテーブルを用いることにより，必要な情報を高速に検索できるようにした．この関数を呼び出す箇所は 6 種類のロード命令を実装している関数の内部である．これを選んだ理由は，メモリアクセスの直後に割込みを発生させる機能と同様に，呼び出す箇所の数が最も少なくて済むと同時に，従来の skyeye が行う処理を妨げない箇所だからである．

条件式を評価し，対応する算出式を計算する機能 集合に含まれる各条件式を評価し，真になった条件式に対応する算出式を計算する関数を，yacc と lex で作成した．この関数の引数は，ロード命令でロードした値である．これを用いることにより，式のオペランドに現在のロード値を利用することができる．また，式のオペランドにグローバル変数が含まれていた場合は，4.1 節で作成したハッシュテーブルを利用し，グローバル

変数の変数名をキーとして変数に対応するメモリアドレスを取得した．そして，そのメモリアドレスを引数として CPU エミュレータのロード命令を実行することにより，グローバル変数の値を取得した．そして，この関数の返り値は，算出式の計算結果である．この返り値をロード命令側で新しいロード値とした．なお，全ての条件式の評価結果が偽であり，算出式を利用しなかった場合，引数であるロード命令でロードした値をそのまま返す．こうすることで，元のロード命令でロードした値を壊さないようにした．

5 実験

提案手法を用いて実際のソフトウェアに存在するデータ競合が発見できることを確認し，その発見に必要な手間を調べるため，実験を行った．本節では，その実験内容や結果，それに対する議論について述べる．

5.1 実験対象

本実験の検出の対象として，組込みソフトウェア向け OS である uClinux 2.4.x と TinyOS 1.x に実際に存在したデータ競合を選んだ．これらのデータ競合が存在するソースコードの一部を，図 7 に示す．これらのうち，uClinux のデータ競合は/drivers/char/68328serial.c のリビジョン 1.12 で修正された．[1] また，TinyOS のデータ競合は/apps/Oscilloscope/OscilloscopeM-.nc のリビジョン 1.1.2.1 で修正された．[2]

5.1.1 uClinux のデータ競合

uClinux でデータ競合が存在したソースコードの概要を，図 7(a) に示す．図 7(a) の関数のうち，`rs_interrupt` 関数は割込みハンドラであり，`rs_flush_chars` 関数は通常処理である．これらの関数を UART ポートのドライバは，ポートを通じて文字データを送信するために利用する．どちらの関数も，以下の条件を共に満たす時，文字データを格納しているキューである `info->xmit_buf` から 1 個のデータを取り出して送信する．

445-448 行目および 782 行目 UART ポートに対してデータが書き込み可能である．すなわち，`utx.w` と UTX_TX_AVAIL(0x2000) の論理積が真である．

400 行目および 768 行目 キューに文字が 1 つ以上格納されている．すなわち，`info->xmit_cnt` が 1 以上である．

これらの関数でデータ競合が顕在化するのは，`rs_flush_chars` 関数が 768 行目の条件分岐で else 側に分岐した後でキューに格納されている文字が 1 つの時に，`rs_interrupt` 関数によってキューが変更された場合である．すなわち，768 行目の条件判定が失敗した直後に，`info->xmit_cnt` が 1 かつ `utx.w & UTX_TX_AVAIL` が真である時に割込みが発生することが，データ競合が顕在化する条件である．このとき，`rs_interrupt` 関数によってキューに格納されている最後の 1 文字が送信されることで，キューの残りデータ数が 0 になる．しか

[1] http://cvs.uclinux.org/cgi-bin/cvsweb.cgi/uClinux-2.4.x/drivers/char/68328serial.c.diff?r2=1.12&r1=1.11&f=h

[2] http://tinyos.cvs.sourceforge.net/viewvc/tinyos/tinyos-1.x/apps/Oscilloscope/OscilloscopeM.nc?r1=1.1&r2=1.1.2.1

```
389: static _INLINE_ void transmit_chars(struct m68k_serial *info)
390: {

400:  if((info->xmit_cnt <= 0) || info->tty->stopped) {
401:       /* That's peculiar... TX ints off */
402:       uart->ustcnt &= ~USTCNT_TX_INTR_MASK;
403:       goto clear_and_return;
404:  }
405:
406:  /* Send char */
407:  uart->utx.b.txdata = info->xmit_buf[info->xmit_tail++];
408:  info->xmit_tail = info->xmit_tail & (SERIAL_XMIT_SIZE-1);
409:  info->xmit_cnt--;

423: }

428: void rs_interrupt(int irq, void *dev_id, struct pt_regs * regs)
429: {

445:  tx = uart->utx.w;

448:  if (tx & UTX_TX_AVAIL)   transmit_chars(info);

453: }

757: static void rs_flush_chars(struct tty_struct *tty)
758: {

768:  if (info->xmit_cnt <= 0 || tty->stopped || tty->hw_stopped ||
769:      !info->xmit_buf)
770:       return;

782:  if (uart->utx.w & UTX_TX_AVAIL) {

786:       /* Send char */
787:       uart->utx.b.txdata = info->xmit_buf[info->xmit_tail++];
788:       info->xmit_tail = info->xmit_tail & (SERIAL_XMIT_SIZE-1);
789:       info->xmit_cnt--;
790:  }
```

(a) uClinux-2.4.x/drivers/char/68328serial.c のリビジョン 1.11

```
145: event result_t ADC.dataReady(uint16_t data){

149:  pack->data[packetReadingNumber++] = data;

153:  if (packetReadingNumber == BUFFER_SIZE)
154:  {
155:       packetReadingNumber = 0;

170:  }

178: }
```

(b) tinyos-1.x/apps/Oscilloscope/OscilloscopeM.nc のリビジョン 1.1

図 7: 実験対象のデータ競合が存在するソースコード

し， rs_flush_chars 関数では 786-789 行目の処理が実行されるため，空のキューからデータを読み出してしまい，誤った文字データが送信されるという障害が発生する．

なお，今回の実験では，提案手法を実装した CPU エミュレータ上で該当の機能を動作させるために，以下の改変を uClinux に行なった．

- rs_interrupt 関数と rs_flush_chars 関数を ARM アーキテクチャから呼び出せるように，rs_interrupt 関数を割込みハンドラとして登録する箇所や，rs_flush_chars 関数を呼び出す箇所を改変した．その理由は，これらの関数は本来 m68k アーキテクチャでしか呼び出せないからである．なお，関数の内容そのものは改変していない．

- __delay 関数を呼び出さないように uClinux を改変した．その理由は，この関数は大量の実行時間を消費する上に，rs_interrupt 関数や rs_flush_chars 関数の動作と無関係だからである．なお，この実験の評価には厳密な実行時間を必要としないため，この改変による実験結果への影響は低いと考えられる．

5.1.2 TinyOS のデータ競合

TinyOS でデータ競合が存在したソースコードの概要を，図 7(b) に示す．図 7(b) に記載されている ADC.dataReady 関数は割込みハンドラである．この関数は A/D 変換器から受け取った入力値 data を配列 pack->data に格納し，その添字 packetReadingNumber を加算する．その後，packetReadingNumber が添字の上限を越えれば，packetReadingNumber に 0 が代入される．

この関数でデータ競合が顕在化するのは，packetReadingNumber が添字の上限を越えた時に，packetReadingNumber に 0 が代入される前に割込みが発生し，ADC.dataReady 関数が再び呼び出された場合である．この時，添字の上限を越えた値を持つ packetReadingNumber によって，配列 pack->data の範囲外にアクセスするという障害が発生する．

今回の実験では，提案手法を実装した CPU エミュレータ上で該当の機能を動作させるために，TinyOS から実験に必要な関数のみを抽出し，ARM アーキテクチャの CPU である LPC2388 上で動作できるように改変した．その理由は，TinyOS は本来 AVR アーキテクチャ向けに開発されているからである．具体的には，割込みハンドラの実行に必要なソースコードを抽出し，それらに以下の改変を行なった．

- 割込みを受け付けた時に，その割込みに対応した割込みハンドラを呼び出すアセンブラコードを記述した．

- 割込みの許可や禁止を行う処理を LPC2388 向けに改変した．

5.2 実験方法

提案手法によってデータ競合を発見するのに必要な作業を調べるために，データ競合が存在する割込みハンドラと通常処理の組をデータ競合の修正ログから調べ，それを基に提案手法によってデータ競合による故障を再現する作業を行った．これらの情報はソースコードから容易に特定できるため，ユーザがテスト前に把握している可能性は高いと考えられる．特に，割込みによるデータ競合が存在する可能性を疑っている場合，割込みハンドラと，その割込みハンドラが書き換える変数を 2 回利用する通常処理に着目するのは自然であると考えられる．

データ競合による故障を再現する作業は，以下の通りである．

1. 割込みハンドラに対応する割込みの種類を特定する．これを特定するには，CPU の仕様書と，ソースコード中で関数を割込みハンドラとして登録する箇所を調べた．

2. データ競合による障害が発生する原因となる変数を特定する．これを特定するには，割込みハンドラが書き換えている変数から，通常処理で 2 回利用されている変数を調べた．

3. データ競合による障害の再現に必要な事前条件を調査した．このような事前条件としては，ステップ 2 で特定した変数を割込みハンドラに書き換えさせるのに必要な事前条件や，割込みハンドラが書き換える直前での変数の値である．

4. ステップ 3 の調査結果から，割込み生成機構や値すり替え機構に必要な指定内容を特定し，それを設定ファイルに記述した．この設定ファイルに記述する際には，値をすり替える変数のメモリアドレスを，nm コマンドや CPU エミュレータの実行によって調査した．

5.3 実験結果

本項では，5.2 節の手順によってデータ競合による故障を再現するのに行った作業について説明する．

5.3.1 uClinux のデータ競合

uClinux のデータ競合による故障を再現するのに記述した設定ファイルは図 8 のようになった．この設定ファイルを記述する時，3.1 で説明した多重割込みを許可するオプションは選択しなかった．その理由は，今回の実験対象の発生に多重割込みが不要だからである．

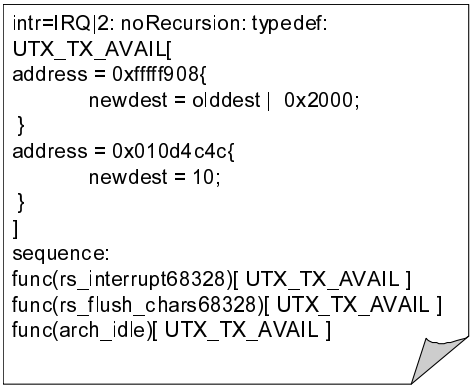

図 8: uClinux の実験に用いた設定ファイル

また，uClinux では膨大な量の多重割込みを受け付けることが可能なため，多重割込みを許可すると処理が進まなくなるからである．

また，データ競合による障害を発生させるために，以下の処理を行う関数を作成した．

1. キューにデータを格納する関数を呼び出し，5 個のデータを格納する．
2. rs_flush_chars 関数を呼び出す．

このような設定ファイルや関数を作成するのに行った作業は，以下の通りである．

割込みハンドラに対応する割込みの種類の特定 rs_interrput 関数を呼び出す割込みの種類が IRQ と FIQ のいずれであるかと，rs_interrput 関数を呼び出す割込みのデバイスに対応する番号を，割込み発生機構の設定ファイルに記述する．これらは，ソースコード中で rs_interrput 関数を登録している箇所と，skyeye によってエミュレートしている CPU である Atmel の仕様書を見ることで特定した．

データ競合による障害が発生する原因となる変数の特定 データ競合による障害が発生する原因となる変数として info->xmit_cnt を特定し，データ競合による障害が発生する時にこの変数が取る値を調査した．その結果，図 7(a) の 768 行目が実行される直前で info->xmit_cnt の値が 1 であり，図 7(a) の 768 行目が実行された後で割込みが発生することによってデータ競合が起こることが判明した．ただし，info->xmit_cnt はキューのデータ数を管理する変数なので，この値を設定するには rs_flush_chars 関数を呼び出す前にキューにデータを格納する必要がある．また，rs_flush_chars 関数が呼び出されてから図 7(a) の 768 行目が実行されるまでに，データ競合に無関係なメモリアクセスが 4 回実行されており，その直後で提案手法によって余分な割込みが 4 回生成される．その結果，図 7(a) の 768 行目が実行されるまでにキューのデータが 4 個送信され，info->xmit_cnt の値が 4 減算されてしまう．そのため，データ競合による障害を再現するには，rs_flush_chars 関数を呼び出す前にキューにデータを 5 個格納する必要があった．

変数の値を変更する処理を割込みハンドラに行わせるのに必要な事前条件の特定 データ競合による障害の再現に必要な事前条件として，info->xmit_cnt の値を変更する処理を割込みハンドラに行わせるのに必要な事前条件を調査した．その結果，utx.w & UTX_TX_AVAIL が真であることが事前条件として必要なことが判明した．そのため，utx.w の値とすり替える値と，utx.w のメモリアドレスを値すり替え機構の設定ファイルに記述した．この時，UTX_TX_AVAIL の値は uClinux のソースコードから特定した．また，utx.w に対応するメモリアドレスは，CPU エミュレータで一度実行して調

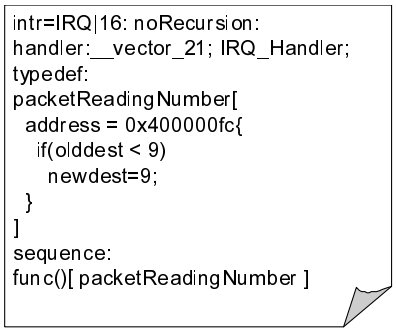

図 9: TinyOS の実験に用いた設定ファイル

べることによって特定した．その際，`utx.w` の値を読み出すロード命令を特定する方法の 1 つとして，dwarf2[1] を利用し，ソースファイル中で `utx.w` の値を読み出すロード命令を実行している箇所の行に対応するプログラムカウンタの値を特定した．ただし，この作業は `utx.w` で入力値を取得できるデバイスの仕様書があり，そこにデバイスの入力に対応するメモリアドレスが記述されていれば行う必要はない．このようなメモリアドレスが書かれた仕様書は大半のデバイスに対して存在している．

なお，データ競合の発見が実時間以内で終わることを確認するために，実行時間を計測した．その結果，提案手法を実装した CPU エミュレータ上でのプログラムの実行開始から，データ競合を発見するまでに掛かった時間は，CPU エミュレータのクロックにして 72417488 サイクルだった．一方，提案手法を利用しなかった場合に，本来ならデータ競合によって障害が発生するはずの箇所を実行するまでにかかった時間は 4836078 サイクルだった．つまり，提案手法を利用することによって実行時間は約 15 倍になった．なお，発生した割込みの数などは以下のようになっていた．

- 割込み発生機構が発生させた割込み数:1409375
- 割込み発生機構が発生させた割込みによって割込みハンドラが呼び出された回数:390722
- 割込みハンドラの処理時間の合計:69952632 (実行時間全体の 96%)

5.3.2 TinyOS のデータ競合

TinyOS のデータ競合による故障を再現するのに記述した設定ファイルは図 9 のようになった．この設定ファイルを記述する時，3.1 で説明した多重割込みを許可するオプションは選択しなかった．その代わりに，main 関数から割込みハンドラである `ADC.dataReady` 関数を通常処理として呼び出し，その中で提案手法によって割込みを発生させるようにした．こ

のような方法を取った理由は，main 関数内でメモリアクセスが実行される保証がなく，従って main 関数内で提案手法によって割込みハンドラが呼び出されるとは限らなかったからである．

また，ADC.dataReady 関数を呼び出す関数を割込みハンドラとして作成した．ADC.dataReady 関数自身を割込みハンドラとして呼び出さない理由は，ADC.dataReady 関数を呼び出すにはデバイスから受け取る入力値を引数とする必要があるからである．この引数には，定数として 100 を指定した．

このような設定ファイルや関数を作成するのに行った作業は，以下の通りである．

割込みハンドラに対応する割込みの種類の特定 ADC.dataReady 関数を呼び出す割込みの種類が IRQ と FIQ のいずれであるかと，ADC.dataReady 関数を呼び出す割込みのデバイスに対応する番号を，割込み発生機構の設定ファイルに記述する．今回は自力で作成したサンプルプログラムから ADC.dataReady 関数を呼び出す関数を割込みハンドラとして呼び出すようにしたため，これらの情報は自明であった．

データ競合による障害が発生する原因となる変数の特定 データ競合による障害が発生する原因となる変数として packetReadingNumber を特定し，データ競合による障害が発生する時にこの変数が取る値を調査した．その結果，図 7(b) の 149 行目の処理が実行されてから，153 行目の処理が実行されるまでの間に packetReadingNumber の値が配列 pack->data の添字の上限である BUFFER_SIZE である時に，割込みが発生することによってデータ競合が起こることが判明した．そのため，設定ファイルで packetReadingNumber の値を BUFFER_SIZE にすり替えるために，BUFFER_SIZE の値と packetReadingNumber に対応するメモリアドレスを調査した．このうち，BUFFER_SIZE の値は TinyOS のソースコードから特定した．また，packetReadingNumber に対応するメモリアドレスは，実験対象の実行ファイルに対して nm コマンドを実行することで特定した．ただし，このように値をすり替える指定を設定ファイルに記述する際には，packetReadingNumber の値が添字の上限を越えた後で，その値と添字の上限がすり替えられるのを防ぐため，すり替えを行う条件式として “packetReadingNumber の値 < 添字の上限″ を記述した．

なお，データ競合の発見が実時間以内で終わることを確認するために，実行時間を計測した．その結果，提案手法を実装した CPU エミュレータ上でのプログラムの実行開始から，データ競合を発見するまでに掛かった時間は，CPU エミュレータのクロックにして 1565 サイクルだった．一方，提案手法を利用しなかった場合に，本来ならデータ競合によって障害が発生するはずの箇所を実行するまでにかかった時間は 494 サイクルだった．つまり，提案

手法を利用することによって実行時間は約 3 倍になった．なお，発生した割込みの数などは以下のようになっていた．

- 割込み発生機構が発生させた割込み数:216
- 割込み発生機構が発生させた割込みによって割込みハンドラが呼び出された回数:16
- 割込みハンドラの処理時間の合計:1068(実行時間全体の 68%)

5.4 考察

本項では，5 節の実験内容や実験結果を元に，以下の観点から提案手法について考察する．

- ランダムテストとの違い
- 変数の値をすり替える際の問題点
- 未知の障害を検出する手順
- テストに必要なテスト対象のプログラムについての知識

5.4.1 ランダムテストとの違い

1 節で説明したように，本研究の提案手法は Regehr の手法 [15] を改善することを目標とした．その改善点とは，割込みを発生させるタイミングと，割込みハンドラの動作である．これらについて，5.3 項を元に，どれほど改善されているかを考察する．

まず，割込みを発生させるタイミングについて議論する．提案手法ではメモリアクセスの直後で網羅的に割込みを発生させているのに対し，Regehr の手法は割込みをランダムに発生させている．そのため，両手法によって割込みが発生するタイミングは必ずしも一致しない．しかし，Regehr の手法では割込みを発生させる頻度を，ユーザからの入力によって調整できる．従って，Regehr の手法で割込みを発生させる頻度が高ければ，提案手法と同様のタイミングを Regehr の手法で網羅することが可能になる．

しかし実際には，本研究の提案手法が割込みを発生させるタイミングを，Regehr の手法で網羅するのは困難だと考えられる．その理由は，プログラムが割込みハンドラを既に呼び出していることを，本研究の提案手法では検知できるのに対し，Regehr の手法では考慮できていないからである．3.1 項で説明したように，本研究の提案手法では割込みハンドラが呼び出されている間に多重割込みを発生させないことをオプションとして選べる．このオプションによって，多重割込みを受け付けないプログラムで余分な割込みを発生させることを

防ぐことができる．しかし，Regehr の手法にはこのような機能がないため，多重割込みを受け付けないプログラムの実行時には無意味な割込みを発生させることになる．

次に，割込みハンドラの動作について議論する．Regehr の手法では割込みを発生させるだけだったのに対し，本研究の提案手法では割込みハンドラがメモリから読み出す値やデバイスからの入力値をすり替えることを可能にした．この機能を用いて，デバイスからの入力値を用いた条件分岐や，メモリから読み出す値の改変を割込みハンドラに行わせることによって，データ競合による障害を発生させる割込みハンドラの命令がより実行し易くなった．

このような実行パスの変更がデータ競合の検出に役立つソフトウェアは多いと考えられる．というのも，ソフトウェアはデバイスから入力値や割込みを受け取って動作するために作成されている上に，デバイスから受け取った入力値やメモリの値を割込みハンドラが処理した結果は通常処理にも影響するからである．

ただし，この機能はユーザが事前に割込みの種類やすり替える値を指定する必要がある．従って，ユーザが全くこの指定を行わずに提案手法を利用した場合は，Regehr の手法との差異は小さくなる．しかも，このような指定を行うには，デバイスの仕様書やソースコードを読む手間を費やす必要がある．従って，今後の課題としては，ユーザが行わなければ成らない指定を極力減らしつつ，割込みハンドラを呼び出すのに必要な割込みや値を自動的に特定する機能の実装が挙げられる．

5.4.2 変数の値をすり替える際の問題点

5.3 項のように，データ競合の検出には，デバイスからの割込みや入力値以外にも，変数の値などの事前条件が必要になることがある．そこで，このような事前条件を用意する方法について考察する．

5.3 項の実験では，データ競合による障害を発生させる事前条件を用意するために，値すり替え機構によって変数の値を変更した．この時，変数のメモリアドレスを指定するために，nm コマンドを実行した．この時，実験対象のソフトウェアを再ビルドする度に nm コマンドを実行する手間がかかってしまう．また，グローバル変数のメモリアドレスは nm コマンドで特定できるが，ローカル変数のメモリアドレスは nm コマンドで特定できないという欠点もある．

そこで，このような変数の用意を支援する方法として，本研究の値すり替え機構の拡張を提案する．本研究の値すり替え機構はデバイスから受け取る入力値を指定するために特定のメモリアドレスから受け取る値をすり替える．この時，デバイスではなくプログラム中の変数に対応するメモリアドレスを自動的に特定できれば，ユーザがメモリアドレスを調べることなくプログラム中の変数の値をすり替えることが可能になる．

この機能を実現するには，4.3 項で実装した，算出式や条件式のオペランドとしてグロー

バル変数の値を取得する機能を利用することが考えられる．

5.4.3 未知の障害を検出する手順

5.2 項では既知のデータ競合による障害を再現した．ここでは，未知のデータ競合を提案手法によって検出する手順について考察する．なお，ここでの検出とはデータ競合による障害を発生させることであり，その障害となったデータ競合が存在する箇所をユーザが確認することについては本研究で対象としていない．

まず，提案手法によって検出可能なデータ競合が存在する可能性のある箇所をソースコードから特定する必要がある．このような箇所は，2 節で説明したデータ競合が存在する条件から，以下の方法で特定できる．

1. 割込みハンドラで書き換えられている変数を特定する．
2. その変数の値を読んでいる箇所や，書き換えている箇所を特定する．
3. それらの箇所のうち，同じ値を 2 回利用することを仮定している箇所を特定する．このような箇所とは，2 回同じ変数を利用している箇所のうち，2 回目の利用では変数の値を読み出している箇所である．

次に，提案手法を利用するのに必要な指定を行う．この指定として，どのデータ競合を検出する際にも必要になるのは，テスト対象の割込みハンドラを呼び出すために必要な割込みの種類である．また，割込みハンドラが変数を書き換える処理を行うのに必要な条件分岐があれば，その条件式が真になるような値をすり替える指定も必要になる．

そして，その割込みハンドラで書き換えられる変数の種類によって，すり替える値の指定が異なってくる．具体的には，各変数に対して以下の指定が必要になる．

- 配列の添字などのデータ数を管理する変数なら，その上限や下限とすり替えるように指定する．こうすることにより，変数への不適切な加減算によるバッファオーバーフローが検出できる．なお，5.3 項のように，データ競合による障害が発生する状況になった後で変数の値をすり替えないようにするため，適切な条件式を値すり替え機構の設定ファイルに記述する必要がある．
- 1 回目の利用時に条件分岐の条件式で評価される変数なら，その条件式で境界値分析を行い，その境界値とすり替えるように指定する．こうすることにより，割込みハンドラが変数の値を書き換えることによって，条件分岐が正常に機能しなくなる状況を発見することができる．ただし，5.3 項のように，余分な割込みが発生するために境界値分析では適切なデータ数が特定できないこともある．その場合には，条件分岐直前

までは割込みを受け付けない処理をテスト対象のソフトウェアに行わせることで，余分な割込みを防ぐことができる．

- 標準出力やファイルへの出力が行われる変数なら，元の値に加減算を行った値とすり替えるように指定する．その理由は，ソフトウェアの出力結果を見るだけでデータ競合による障害の発生が確認できるので，元の値と異なる値であれば，どんな値でも構わないからである．

このように，データ競合による障害が発生しうる全ての箇所を特定し，その箇所で変数が利用される方法に応じてすり替える値や割込みを発生させるデバイスを指定することで，データ競合を網羅的に検出することができる．

5.4.4 テストに必要なテスト対象のプログラムについての知識

5.2 項ではデータ競合が発生する割込みハンドラや通常処理が予め特定できている状態で，データ競合を再現した．本項では，ユーザが持つ知識や考えに応じた，提案手法を用いて行うテスト方法について議論する．

まず，ソフトウェアについての知識を殆ど持っていないが，デバイスについての知識があるユーザを想定する．この時，あるデバイスの割込みを一度または連続して発生させることだけで顕在化するデータ競合を検出することが可能になる．というのも，デバイスに対応する割込みの種類さえ特定できれば，全てのメモリアクセスの直後で割込みを網羅的に発生させることは可能だからである．

また，デバイスの仕様書に従って，デバイスから受け取る入力値を網羅的に割込みハンドラに与えれば，データ競合の原因となる割込みハンドラの命令を実行できる可能性が高いと考えられる．割込みハンドラの処理に必要な入力を厳密に特定するにはソースコードを見る必要があるが，実際はそのような入力値はデバイスの仕様書に従うだけでも作成できると考えられる．というのも，割込みハンドラの動作はデバイスの仕様に基づいて設計されることが多いからである．

ただし，全てのデータ競合を検出するためには，適切なテストケースをプログラムへの入力として作成する必要がある．その理由は 2 つある．1 つは，データ競合は割込みハンドラと通常処理の間に存在するため，全てのデータ競合を検出するには割込みハンドラを呼び出すだけではなく，プログラムの様々な関数を網羅的に実行する必要があるからである．もう 1 つは，5.3 項の実験結果のように，特殊なテストケースを作成しないと検出できないデータ競合も考えられる．ただし，5.3 項では 6 パターン程度のテストケースを作成すればデータ競合を検出できたため，データ競合が検出できるテストケースを作成することは必ずしも困難ではないと言える．

```
 1: unsigned int len = 0;
 2: void str_cpy(char *buf, char *str);
 3: {
 4:   len = strlen(str);
 5:   if((0 < len) && (len <= strlen(str)))
 6:     memcpy(buf,str,len+1);
 7: }
 8:
 9: void interrupt_handler(void){
10:     len++;
11: }
```

図 10: 特定のタイミングで割込みが発生するとデータ競合が検出できないサンプルプログラム

また，以下の条件で存在するデータ競合は提案手法によって検出することは不可能である．従って，下記のデータ競合を効率的に検出できる手法の提案が今後の課題として挙げられる．

- 特定のタイミングで割込みが発生すると発現しなくなるデータ競合．例えば，図 10 のソースコードでは，割込みが 4 行で発生せずに 5 行目で発生した時のみデータ競合による障害が発生する．
- 複数のデバイスから割込みを，特定の順序で受け付けた時にのみ発生する障害の原因となるデータ競合．例えば，タイマーの割込みを受け付けてから，UART ポートからの割込みを受け付けた場合に発生する障害の原因となるデータ競合は，一度に 1 つのデバイスからの割込みしか指定できない現在の提案手法では検出できない．

また，テストのためにソフトウェアを実行する回数は，割込みハンドラの数 × テストケースの数以上になる．これは，提案手法では一度に 1 つのデバイスに対応する割込みハンドラしか呼び出せない上に，割込みハンドラと通常処理の組み合わせを網羅するために複数のテストケースが必要になると考えられるからである．さらに，割込みハンドラによっては，デバイスから受け取る入力値によって分岐する処理の数だけ実行する必要がある．

6 関連研究

これまでの研究でも，割込みを用いたテストを行うための様々な手法が開発されてきた．というのも，割込みによるデータ競合が存在する実行パスは非常に多く，その全てをテストするのが困難だからである．

Regehr は割込みをランダムに発生させることで，割込み駆動型のソフトウェアをテストする手法を提案した [15]．これに対し，本研究の手法はデータ競合が存在する可能性のある実行パスで割込みを CPU エミュレータが発生させる．

本研究は割込み駆動型のソフトウェアを対象としているが，データ競合の検出はマルチスレッドプログラムでも重要な問題となっている．Entropy Injection[4] は，マルチスレッドプログラムでのデータ競合を検出するために，データを読む命令と書く命令の間に人工的な遅延を注入する．この手法はマルチスレッドプログラムでデータ競合による障害が発生する可能性を上げているが，本研究の手法はデータ競合による障害が発生する割込みを直接発生させる．これにより，ユーザはテスト中に特定の実行シナリオをテストやデバッグすることが可能になる．

これ以外にも，様々な動的検出手法が存在する [9, 12] が，これらの手法は割込み駆動型のソフトウェアに適用することは不可能である．これに対し，Regehr と Cooprider は割込み駆動型プログラムのソースコードをマルチスレッドプログラムに変換するという手法を提案している [16]．本研究では，データ競合による障害が発生する時に割込み駆動型プログラムが実際にどう振舞うかをテストしたかったため，この手法は利用しなかった．

本研究ではプログラムの実行を監視し，プログラムが特定のメモリアドレスにアクセスした時に，人工的に割込みを発生させたり，メモリの値やデバイスからの入力値を作成している．このような値を自動的に生成する手法は，フォールトインジェクションと呼ばれている．フォールトインジェクションの目的はシステムの障害が発生するような入力値を作成してプログラムを評価することであり [10]，本研究の目的とは異なっている．しかし，フォールトインジェクションはプログラム中で利用されるデータを動的にすり替えることができるため，本研究ではデータ競合を分析するためのデータを作成するためにフォールトインジェクションと同様の手法を提案した．

このようなフォールトインジェクションのうち，Xception[6] は，プロセッサがハードウェアを利用する時に発生する一時的な欠陥を模倣するツールである．例えば，Xception は特定のアドレスからロードされた値を，ビットフリップなどの方法で書き換えることができる．本研究でもロードされた値を書き換える機構を実装したが，その方法はロードされた値をユーザによって指定された値とすり替えることである．そのため，ユーザは特定の実行シナリオを意図的にテストすることが可能である．

また，Qinject[7] は本研究とは異なるフォールトインジェクション機能を実装した CPU エミュレータである．Qinject は欠陥を含むコードの実行を模倣するために，指定された命令が実行された際に，そのメモリの値を書き換える．これに対し，本研究ではデバイスから特定の入力値を受け取った状況を模倣するために，指定されたメモリアドレスにプログラムがアクセスした際に，そのメモリの値を書き換える．

プログラムのテストやデバッグを行う際，the GNU Project Debugger(以下，gdb) [2] を利用しても，指定されたメモリアドレスからプログラムが値を読む際に，そのメモリの値をテスト用の値とすり替えることができる．ただし，gdb は割込みを直接管理することはできないため，gdb と一般的な CPU エミュレータを組み合わせるよりも本研究で実装した CPU エミュレータの方が実装は容易である．

7　まとめと今後の課題

本研究では，割込みハンドラと通常処理の間に存在するデータ競合を効率的に検出することを目的として，メモリアクセスの直後で割込みを自動的に発生させる機構と，割込みハンドラで利されるメモリの値やデバイスからの入力値をすり替える機構を CPU エミュレータに実装した．

提案手法を適用した CPU エミュレータ上で実際のソフトウェアをテストしたところ，既存手法に比べ，データ競合を検出するのに必要十分なタイミングで割込みを発生させられる上に，メモリの値をすり替えることでデータ競合を検出できることが確認できた．また，提案手法を用いれば，割込みの生成タイミングを手動で調整することなしに，データ競合が検出できることが判明した．

今後の課題としては，プログラムで利用される変数の値をすり替える指定の支援や，割込みを発生させるタイミングの調整，複数のデバイスからの割込みをデータ競合による障害が発生する可能性のある全てのタイミングで発生させる機能の実装が挙げられる．

謝辞

本研究の全過程を通して，常に適切な御指導および御助言を賜りました大阪大学大学院情報科学研究科コンピュータサイエンス専攻 井上 克郎 教授に心より深く感謝いたします.

本研究を通して，随時適切な御指導および御助言を賜りました立命館大学情報理工学部情報システム学科 山本 哲男 准教授に心より深く感謝いたします.

本研究において，随時適切な御指導および御助言を頂きました大阪大学大学院情報科学研究科コンピュータサイエンス専攻 早瀬 康裕 特任助教に深く感謝いたします.

本研究において，適切な御指導および御助言を頂きました大阪大学大学院情報科学研究科コンピュータサイエンス専攻 石尾 隆 助教に深く感謝いたします.

本研究を通して，適切な御指導および御助言を頂きました大阪大学大学院情報科学研究科コンピュータサイエンス専攻 松下 誠 准教授に深く感謝いたします.

最後に，その他様々な御指導および御助言を頂きました大阪大学大学院情報科学研究科コンピュータサイエンス専攻ソフトウェア工学講座の皆様に深謝いたします.

参考文献

[1] Dwarf home. `http://dwarfstd.org/`.

[2] Gdb: The GNU project debugger. `http://www.gnu.org/software/gdb/`.

[3] SkyEye - open source simulator. `http://www.skyeye.org/`.

[4] L. Albertsson. Entropy injection. *SICS Technical Report*, T2007(02), February 2007.

[5] T. Ball, E. Bounimova, B. Cook, V. Levin, J. Lichtenberg, C. McGarvey, B. Ondrusek, S. K. Rajamani, and A. Ustuner. Thorough static analysis of device drivers. *SIGOPS Oper. Syst. Rev.*, 40(4):73–85, 2006.

[6] J. Carreira, H. Madeira, and J. G. Silva. Xception: Software fault injection and monitoring in processor functional units. In *in Processor Functional Units, DCCA-5, Conference on Dependable Computing for Critical Applications*, pages 135–149, 1995.

[7] F. M. David, E. M. Chan, J. C. Carlyle, and R. H. Campbell. Qinject: A virtual machine based fault injection framework. International Conference on Architectural Support for Programming Languages and Operating Systems, March 2008.

[8] D. R. Engler and K. Ashcraft. Racerx: Effective, static detection of race conditions and deadlocks. In Proceedings of SOSP'2003, March 2003.

[9] C. Flanagan and S. N. Freund. Fasttrack: efficient and precise dynamic race detection. In *PLDI '09: Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*, pages 121–133, 2009.

[10] M.-C. Hsueh, T. K. Tsai, and R. K. Iyer. Fault injection techniques and tools. *Computer*, 30:75–82, 1997.

[11] N. G. Leveson. An investigation of the therac-25 accidents. *IEEE Computer*, 26:18–41, 1993.

[12] D. Marino, M. Musuvathi, and S. Narayanasamy. Literace: effective sampling for lightweight data-race detection. In *PLDI '09: Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*, pages 134–143, 2009.

[13] E. G. Mercer and M. D. Jones. Model checking machine code with the GNU debugger. In *12th International SPIN Workshop*, volume 3639 of *Lecture Notes in Computer Science*, pages 251–265. Springer, August 2005.

[14] R. H. B. Netzer and B. P. Miller. What are race conditions?: Some issues and formalizations. *ACM Lett. Program. Lang. Syst.*, 1(1):74–88, 1992.

[15] J. Regehr. Random testing of interrupt-driven software. *Proc. of the 5th ACM international conference on Embedded software*, pages 290–298, 2005.

[16] J. Regehr and N. Cooprider. Interrupt verification via thread verification. *Electron. Notes Theor. Comput. Sci.*, 174(9):139–150, 2007.

付録

本節では，4.1 項で実装した機能の入力となる設定ファイルについて説明する．

実装した機構を利用するのに必要な設定ファイルの書式を，BNF 記法で表すと以下のようになる．

設定ファイル ::= 割込み発生用指定 値すり替えの定義 値すり替え指定列の集合
　　| 割込み発生用指定 割込みハンドラ指定 値すり替えの定義 値すり替え指定列の集合
割込み発生用指定 ::= "intr" "=" 割込みの種類 "|" デバイス番号 ":" オプション
割込みの種類 ::= "IRQ" | "FIQ"
デバイス番号 ::= 整数値
オプション ::= | "noDwarf" ":" | "noRecursion" ":"
　　| "noDwarf" ":" "noRecursion" ":"
割込みハンドラ指定 ::= "handler" ":" 割込みハンドラの定義
割込みハンドラの定義 ::= 割込みハンドラの関数名 行末
割込みハンドラの関数名 ::= 識別子
行末 ::= | ";"
値すり替えの定義 ::= "typedef" ":" 値すり替え指定の集合
値すり替え指定の集合 ::= 値すり替え指定 | 値すり替え指定の集合 値すり替え指定
値すり替え指定 ::= 値すり替え指定の識別子 "[" 指定ブロック "]"
値すり替え指定の識別子 ::= 識別子
指定ブロック ::= メモリアドレス指定ブロック | 指定ブロック メモリアドレス指定ブロック
メモリアドレス指定ブロック ::= 指定ブロックの識別子 ":" "address" "=" メモリアドレスの範囲 "{" 条件式と算出式の組の集合 "}"
指定ブロックの識別子 ::= 識別子
メモリアドレスの範囲 ::= 値の範囲
値の範囲 ::= 整数値 | 整数値 ":" 整数値
条件式と算出式の組の集合 ::= 条件式と算出式の組 | 条件式と算出式の組の集合 条件式と算出式の組
条件式と算出式の組 ::= 算出式 | 条件式
算出式 ::= "newdest" "=" 式 行末
条件式 ::= "if" "(" 式 ")" 条件式と算出式の組
　　| "if" "(" 式 ")" 条件式と算出式の組 "else" 条件式と算出式の組

式 ::= 式 演算子 式 | 負数考慮オペランド | "(" 式 ")"
演算子 ::= 四則演算子 | 比較演算子 | ビット演算子
四則演算子 ::= "+" | "-" | "*" | "/" | "%"
比較演算子 ::= "&" | "|" | "==" | "!="
| "<=" | ">=" | "<" | ">"
ビット演算子 ::= "||" | "&&"
負数考慮オペランド ::= "-" オペランド | オペランド
オペランド":" 整数値 | グローバル変数 | "olddest"
グローバル変数 ::= 識別子
値すり替え指定列の集合 ::= "sequence" ":" 関数別すり替え指定の集合
関数別すり替え指定の集合 ::= 関数別すり替え指定 | 関数別すり替え指定の
集合 関数別すり替え指定
関数別すり替え指定 ::= "func" "(" 関数名 ")" "[" 値すり替え指定列 "]"
| "func" "(" ")" "[" 値すり替え指定列 "]"
関数名 ::= 識別子
値すり替え指定列 ::= 値すり替え指定の識別子
| 値すり替え指定の識別子 "*" 整数値
| 値すり替え指定の識別子 ">" 値すり替え指定列
| 値すり替え指定の識別子 "*" 整数値 ">" 値すり替え指定列
整数値 ::= 10 進数 | 16 進数 | 8 進数
10 進数 ::= 数字 | 0 以外の数字 10 進数 | 0 以外の数字 数字 10 進数
数字 ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
0 以外の数字 ::= 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
16 進数 ::= 16 進接頭辞 15 以下の値
16 進数接頭辞 ::= 0x | 0X
15 以下の値 ::= 16 進数数字 | 16 進数数字 15 以下の値
16 進数数字 ::= 数字 | a | b | c | d | e | f
| A | B | C | D | E | F
8 進数 ::= 0 7 以下の値
7 以下の値 ::= 7 以下の数字 | 7 以下の数字 7 以下の値
7 以下の数字 ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7
識別子 ::= アルファベット 文字
アルファベット ::= a | b | c | d | e | f | g
| h | i | j | k | l | m | n

```
| o | p | q | r | s | t | u
| v | w | x | y | z
| A | B | C | D | E | F | G
| H | I | J | K | L | M | N
| O | P | Q | R | S | T | U
| V | W | X | Y | Z
```

文字 ::= アルファベット | 数字

これらの要素のうち，指定内容を記述するのに重要な要素の意味を，以下に示す．

割込みの種類 割込み発生機構が発生させる割込みの種類の指定．ARM アーキテクチャでは IRQ と FIQ という 2 種類の割込みを受け付ける．そこで，各割込みに対応する識別子を記述することで，割込み発生機構が発生させる割込みを指定する．

デバイス番号 割込み発生機構によって割込みを発生させるデバイスに対応する番号．skyeye では割込みの発生時にレジスタに書き込まれている値によって，その割込みを発生させたデバイスを特定する．そこで，割込みの発生前に設定ファイルに記述されたデバイス番号をレジスタに書き込むことによって，割込みを発生させるデバイスを指定する．このデバイス番号は，CPU の仕様書から容易に調べることができる．

オプション 割込み発生機構や値すり替え機構のオプション．各オプションの指定内容は，以下のようになっている．

まず，“noDwarf” と記述した場合，dwarf2[1] を利用せず，関数名に対応するプログラムカウンタの値や，や変数名に対応するメモリアドレスを取得しない．このオプションが必要な状況は，関数名や変数名を設定ファイルに記述しない上に，大規模なソフトウェアを実行する場合である．大規模なソフトウェアから dwarf2 によって情報を取得するには時間がかかるため，関数名や変数名を利用しない場合には dwarf2 による情報取得を省いた方が良いと考えられる．

また，“noRecursion” と記述した場合，割込み発生機構で割込みを発生させる時に，多重割込みを禁止する．このようなオプションにより，多重割込みによって不都合が生じるソフトウェアでもデータ競合を検出することが可能になる．この禁止を行った場合に割込みを発生させる条件は表 1 の通りである．

割込みハンドラの定義 割込みハンドラの関数名の指定．このように指定を独立させた理由として，様々な機能拡張が考えられることが挙げられる．機能拡張案として挙げられ

るのは，割込みハンドラとして複数の関数を指定したり，割込みハンドラ毎に多重割込みで呼び出す回数を設定したりすることである．

指定ブロック 値すり替え機構の指定内容．具体的には，値をすり替えるメモリのアドレスと，新しいメモリの値を計算するのに必要な算出式と条件式の組である．

メモリアドレスの範囲 値をすり替えるメモリアドレス．今回の実装では，値をすり替えたいメモリのアドレスのうち，最小値と最大値のみを記述すれば，その間の範囲にあるメモリアドレスも指定できるようにした．こうすることにより，複数のメモリアドレスを容易に指定できるようにした．

算出式 すり替える値を計算するための式．なお，C 言語と同じ記法で書けるように，比較演算子やビット演算子の利用を可能にし，行末に';' をつけられるようにした．

条件式 利用する算出式を選ぶための式．C 言語と同じ記法で書けるように，if 文と同じ構造で記述できるようにした上に，入れ子構造の指定も可能にした．

オペランド 算出式や条件式で用いるオペランド．このうち，olddest は現在のロード値を指す．これを用いることにより，現在のロード値を条件式で評価してから，算出式で適切な値にすり替えることを可能にした．

関数別すり替え指定 値のすり替えを，特定の関数が呼び出されている時のみに行うための指定．この関数名がない場合，どの関数が呼び出されている場合でも指定された値のすり替えを行うようにした．

値すり替え指定列 値のすり替えを，一定の列として実行する指定．例えば，指定ブロックとして valueA, valueB, valueA を定義し，値すり替え指定列として [value > valueB > valueC] を指定した場合，値のすり替えは，valueA,valueB,valueC,valueA, ... という順序で行われる．このような列による指定を可能にすることにより，特定のデバイスの動作を想定した入力値のすり替えが可能になる．