修 士 論 文

組込みリアルタイムデータベースライブラリの実装に関する研究

北陸先端科学技術大学院大学
情報科学研究科情報システム学専攻

林　晃平

2008 年 3 月

修 士 論 文

組込みリアルタイムデータベースライブラリの実装に関する研究

指導教官　田中清史 准教授

審査委員主査　田中清史 准教授
審査委員　日比野靖 教授
審査委員　井口寧 准教授

北陸先端科学技術大学院大学
情報科学研究科情報システム学専攻

0610072 林　晃平

提出年月: 2008 年 2 月

概 要

近年の組込みシステムの中には，大容量かつ複雑なデータ管理を要求するシステムが増加している．このようなシステムは，RTOS 上で大量のデータをデータベース化して効率良く管理し，検索・更新できる仕組みを備えている．このようなデータベース管理システム (DBMS) は，一般的にソフトウェア部品としてシステムに組込んで使用される．本論文では，DBMS をソフトウェア部品として実装する．そして，タスクの静的優先度に従って，システムに最適な実行バイナリコードを構築する手法を提案し，シミュレーションにより評価を行う．

目 次

図 目 次

表 目 次

第1章　はじめに

1.1　研究の背景

近年の組込みシステムは，高機能化に伴って，アプリケーションプログラムとそれを管理するプログラムが複雑化，大規模化してきている．一般的に，このようなシステムは，オペレーティングシステム (OS) という階層を用意してアプリケーションプログラムとそれを管理するプログラムに分割することで，プログラムの生産性や保守性を向上させている．このようなプログラムのモジュール化の考え方は組込みシステム開発でも一般的に用いられている．また，搭載される OS は，組込みシステムの特性であるリアルタイム性を向上させるために，リアルタイム OS(RTOS) である場合が多い．

一方，データ管理技法については，アプリケーションプログラムがデータ管理の機能までを含んで処理をしている場合が一般的である．しかし，システムの高機能化が進むにつれ，複数のアプリケーション間で大量のデータを共有する等，高度な要求が出てくると，大容量かつ複雑なデータ管理となるため，データ構造が複雑となる．このような複雑なデータ構造は，アプリケーションプログラムの実装や検証を難しくする要因となる．

本研究では，このような組込みシステムにおけるアプリケーションプログラムのデータ管理の問題点に着目する．アプリケーションプログラムがデータ管理機能を含むのではなく，データのデータベース化を行い，データベースを管理するプログラムを，RTOS にソフトウェア部品として提供する．これにより，データをカプセル化してプログラムから分離することが可能となり，データ管理の高効率化やシステムの開発・検証時間の削減に寄与する．このような，データをデータベース化して効率良く管理し，検索・更新できる仕組みを Database Management System(DBMS) という．実装する組込み DBMS は，制限されたメモリ空間内で，効率の良いアルゴリズムを用いて高速に処理できるソフトウェアであることが望まれる．本研究では，制限されたメモリ空間を効率利用する方法として，実行バイナリコードの最適化手法を提案し，実装する DBMS に適用する．実装する DBMS を実行するターゲットプロセッサは ARM[1] とする．ARM は組込み向けプロセッサとして近年に広く採用されており，ISA として 32 ビット ARM 命令とそのサブセットである 16 ビットの Thumb 命令を持つ．システム開発者は，実行バイナリ作成時に ARM と Thumb のコードを混在させることで，性能やコードサイズをルーチン毎に最適化できる．また，組込みシステム用の RTOS である μITRON 仕様 [2] では，タスク毎に静的優先度を持たせている．したがって，これらを利用することで，タスクの優先度を考慮しつつ，性能とコードサイズのバランスがとれたシステム構築が可能となる．

1.2 研究の目的

本研究では，ARM 上で動作する μITRON 仕様の RTOS をターゲットとした DBMS をソフトウェア部品として実装する．その際，クエリー関数は，実行速度に最適化された ARM 版と，少メモリ容量に最適化された Thumb 版を用意する．そして実行バイナリコード作成時のリンクの段階で，タスクの静的優先度に従ってリンクするクエリー関数を自動選択するツールを作成する．これにより，システム開発者のアプリケーション開発時における負担を軽減しつつ，実行バイナリコードサイズを削減することが可能である．これらのアプローチにより，メモリ使用量を削減しつつリアルタイム性を向上させることにより，システムの高効率化を達成することを目的とする．本論文では，実装する DBMS について述べ，その実行バイナリコードの最適化手法を提案し，システムの高効率化を達成することをシミュレーションで示す．

1.3 本論文の構成

本論文の構成を以下に示す．
第 2 章では，データベースを実装するターゲットシステムについて述べる．
第 3 章では，データベースライブラリの実装とその最適化手法について述べる．
第 4 章では，評価環境について述べた後，提案手法の性能評価を行う．
第 5 章では，まとめと今後の課題について述べる．

第2章　組込みデータベースと ターゲットシステム

組込みシステムにおいては，CPU 性能やメモリ容量といった資源の制約やリアルタイム性の確保等が問題となる．本章では，それらの問題をふまえ，組込みシステムに特化したデータベースの構築手法を述べる．また，組込みデータベースの実装ターゲットとなるリアルタイムシステムの定義とそれに組込まれるプロセッサと OS の具体的な仕様を決定する．

2.1　組込みデータベース

2.1.1　組込みデータベースの必要性

アプリケーション開発においてデータの管理を行う際，システム開発者は，アプリケーションで使用する論理的なデータが物理的にどこに格納されているかを把握したうえで，データ処理を定義するという手法をとる（図 2.1）．

組込みシステムの小規模なアプリケーション開発においてもこの手法を採用する場合が多い．しかし，この手法はデータ管理をアプリケーション内に内包するため，データが変わるたびにアプリケーションの再設計が必要となる．また，大容量かつ複雑なデータ管理となるとデータ構造が複雑になり，アプリケーションを検証する時間の増大を招く．それに対して，データ管理にデータベースの手法を適用した場合，アプリケーションのデータ管理をプログラム本体と分離することが可能となる（図 2.2）．このことをデータ管理のカプセル化という．この場合，データベースがデータの一括管理を行うため，システム開発者は，データが物理的にどの位置にあるかということを把握しなくともデータを扱うことが可能となる．また，システム開発人員をデータベース管理プログラムの開発者とアプリケーション開発者にわけることができ，システム開発の効率向上と工数削減に寄与する．

近年の組込みシステムは，扱うデータの大規模化や複雑な管理を要求する傾向にあり，システム開発効率向上の観点からも，後者の手法を適用することは有用である [3].

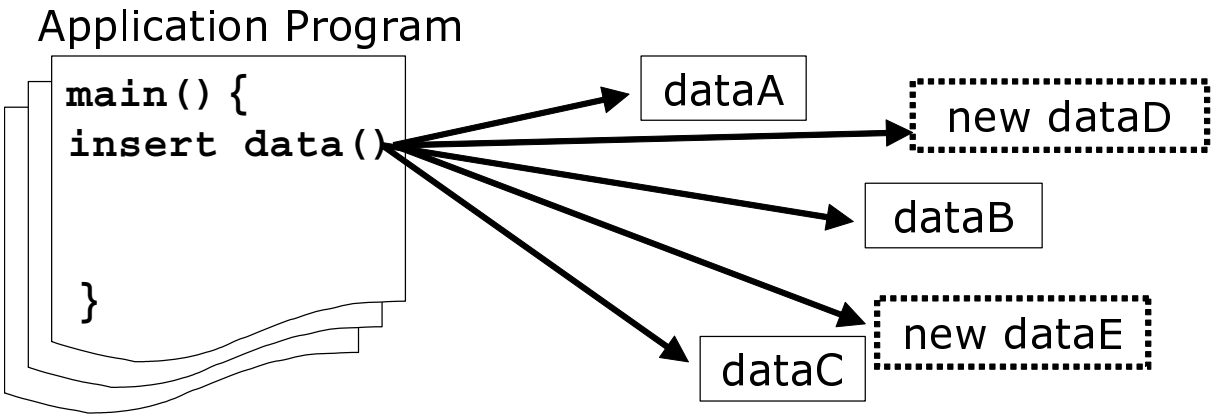

図 2.1: アプリケーションによるデータ管理

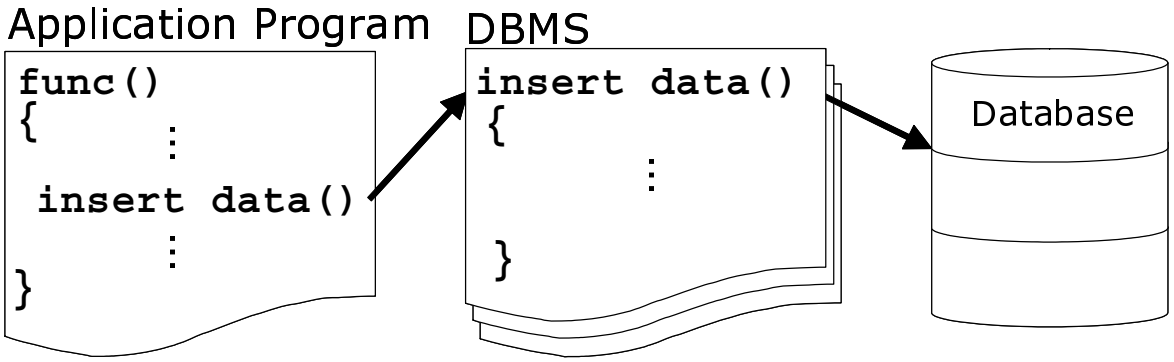

図 2.2: データベースによるデータ管理

2.1.2 組込みデータベースの機能

データベースは，データとデータ処理プログラムから構成される．データ処理プログラムは，データファイルの作成，レコード構造の定義，データの CRUD(Create(生成), Read(読出し)，Update(更新)，Delete(削除)) 操作をするプログラムである．このほかにデータの整合性を保障するプログラムやデータのロック処理を行うプログラム等がある．これらを総称してデータベース管理システム (DBMS: Database Management System) という．組込みデータベースの DBMS は以下の要件を満たす必要がある．

- CRUD 操作のリアルタイム性

  組込みシステムにおいて，リアルタイム動作の必要がある場合，データベースの CRUD 操作にもリアルタイム性が要求される．

- 高速で少メモリ容量

  組込みシステムには強いメモリ資源制約があるため，DBMS 本体のバイナリサイズは少メモリ容量が要求される．コンパクトなバイナリサイズの DBMS で大量のデータを高速に処理できるかどうかが問題となる．

- 応答が速いインターフェース

  組込みデータベースのインターフェースは，システムが必要としているもののみ取り込まれるようにするため，粒度が細かく，オーバーヘッドが少ない C 言語のカーネル・インターフェースを提供する必要がある．

2.1.3 組込みデータベースの構成

組込みデータベースは，アプリケーション開発者が必要とする機能を機能毎に関数化したライブラリとして提供される．このデータベース構成方法をアプリケーション内蔵型という．アプリケーション内にあるデータベースを操作する関数は，システムのコンパイル時に，DBMS ライブラリからリンクされる．この方法は，アプリケーションに必要となるデータベースの機能や要求される性能に応じて実装方法を変更することが可能であることに特徴がある．

2.2 ARM プロセッサ

ARM は，SoC(System on a Chip) 組込み向けに広く採用されている RISC プロセッサである．基本となるレジスタ数は 16 個と RISC としては少なく，遅延分岐やレジスタウィンドウ無しという構造は，低消費電力性や小さいチップ面積を実現するためである．また，すでに存在する機能ブロックの効率的な利用を追及しており，命令セットを高密度化し，効率的な命令セットとしている．その命令セットは，32 ビットの ARM 命令と，16 ビットの Thumb 命令からなる．Thumb 命令は，ARM 命令を 16 ビットに圧縮したサブセットであり，デコード時に ARM 命令に伸張される．これにより，メモリ制約の厳しいシステムにおいて性能をあまり落とさずにコードサイズを圧縮可能としている．

2.2.1 ARM モードと Thumb モード間の状態遷移

ARM モードと Thumb モード間で状態遷移するには，専用の分岐命令を用いる．例えば，ARM の Version 4T アーキテクチャでは bx という命令がそれであり，レジスタジャンプを行いつつ，そのレジスタの下位 0 ビット目の値で遷移先のモードを決定する．フラグが'0' なら，ARM モードへ，'1' ならば Thumb モードへ遷移する．ARM モード関数と Thumb モード関数間の状態遷移の例を以下に述べる．

- ARM モード関数 から Thumb モード関数を呼ぶ場合

```
address(Hex)  instruction          MODE  Comment
      38      bl  58               ARM   ; Jump and Link
      58      ldr r12, [pc, #0]    ARM   ; 0x60 から 4 バイトロード
      5c      bx  r12              ARM   ; 2003f7c(Thumb) へ分岐
      60      (02003f7d)                 ; 2003f7d & 0x1 == 1
```

- Thumb モード関数 から ARM モード関数を呼ぶ場合

```
address(Hex)  instruction  MODE   Comment
      3e      bl  6c       Thumb  ; Jump and Link
      6c      bx  pc       Thumb  ; pc[0] == 0 0x70 に分岐
      6e      nop          Thumb  ; word alignment
      70      b   0c       ARM    ; 0c 番地に分岐
```

どちらの場合も，関数へ分岐する場合は分岐元のアドレスをリンクレジスタ (r14) へ保存するため，bl(Branch and link) 命令を実行する．この命令を実行したのち，bx 命令を使用して状態遷移しつつ，関数に分岐する．

なお，ARM Version 4T プロセッサは，3 段パイプラインでプログラムカウンタ (pc) をレジスタとして持つ関係上，命令実行段階で pc を読み出すと，ARM モードの場合はその命令の pc + 8 となり，Thumb モードの場合はその命令の pc + 4 となる．

関数からのリターンは，リンクレジスタの値を使用して bx 命令を用いることにより行う．

このような ARM モード関数から Thumb モード関数への状態遷移をするプログラムをベニア関数という．ARM モード関数と Thumb モード関数が混在したプログラムを作成する際は，このようなベニア関数が必要となる．このベニア関数をリンク時に自動生成する手法として，インターワーキングがある．

2.2.2 インターワーキング [4]

インターワーキングは，ARM モードや Thumb モードの関数が混在するオブジェクトをリンクする段階で，状態遷移に必要となるベニア関数を適宜自動的に挿入する手法である（図 2.3）．ARM 社や GCC のコンパイラはこの手法を正式にサポートしているため，コンパイル時にオプションとして指定することで適用できる．ただし，GCC のコンパイラでは，Thumb 命令でコンパイルする関数を個別に設定することができないため，ARM モードの関数と Thumb モードの関数を別ファイルにまとめる必要がある．

インターワーキングを用いることで，実行バイナリをシステムに最適化できる．しかし，コードサイズが小さくなる反面，命令数が多くなるため，実行速度とのトレードオフといえる．

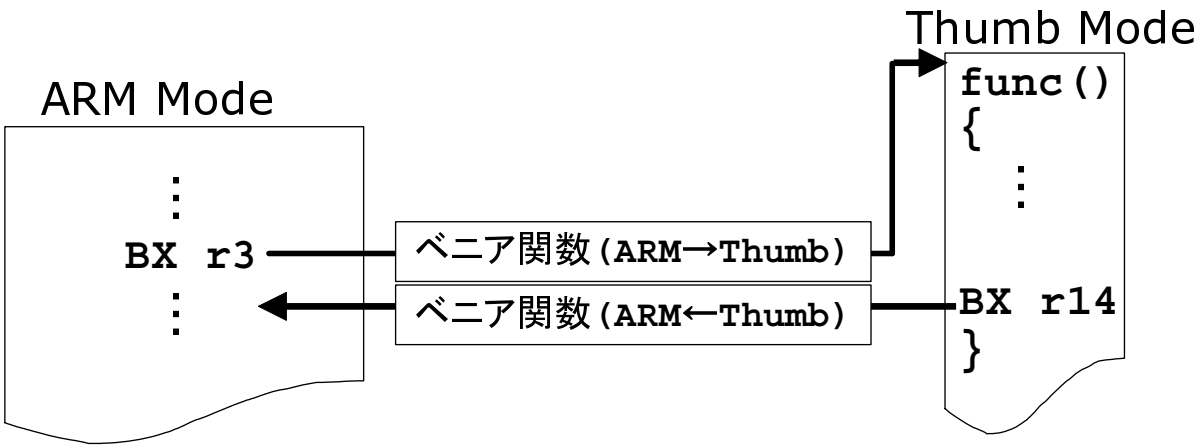

図 2.3: インターワーキング

2.3 リアルタイムシステム

本節では, リアルタイム性を評価するためのパラメータを μITRON 仕様に基づき定義した後，本研究で用いる μITRON 仕様 OS の機能とソフトウェア部品について述べる．

2.3.1 用語の定義

プログラムの並列実行の単位をタスクという．タスク中のプログラムは，逐次的に実行され，タスク同士は並行して実行される．このタスク同士が並行して実行される状態は，それぞれのタスクを時分割で実行することで実現している．

実行するタスクを切り替えることをディスパッチといい，ディスパッチを行うカーネル内の機構をディスパッチャという．

タスク実行の順序を決める順序関係を優先順位という．優先順位の高いタスクで実行できる状態にあるものは，優先順位の低いタスクを実行中でも優先的に実行することができる．このように，次に実行するタスクを決定する処理をスケジューリングという．スケジューリングはスケジューラが行い，一般的にディスパッチャに含まれて実装される．

2.3.2 タスクの時間情報 [5]

本研究におけるタスクの時間情報を以下に定義する（図 2.4）.

- 起動要求時刻 (activation time)[a]

  タスクが起動要求を受けて休止状態から実行可能状態へ移行した時刻.

- 開始時刻 (start time)[s]

  タスクの実行が開始された時刻.

- 完了時刻 (end time)[f]

  タスクの実行が完了した時刻.

- 実行時間 (execution time)[C]

  タスクが実行が開始されてから完了するまでに要する時間.

- 応答時間 (response time)[R]

  タスクが起動要求を受けてから実行を完了するまでに要する時間．$f - a$ で求まる.

- 相対デッドライン (relative deadline)[D]

  タスクの起動要求時刻から絶対デッドラインまでの時間

- 絶対デッドライン (absolute deadline)[d]

  タスクの実行が終了しなければならない締切り時刻．$a + D$ で求まる．

- 余裕時間 (laxity time)[L]

  絶対デッドラインを越えることなく実行を完了するタスクの実行開始時の最大遅延時間．$d - f$ で求まる．

- 周期 (period)[T]

  周期的なタスクの起動要求の周期

一般に周期的な起動要求があるタスクを周期タスク，そうでないタスクを非周期タスクという．

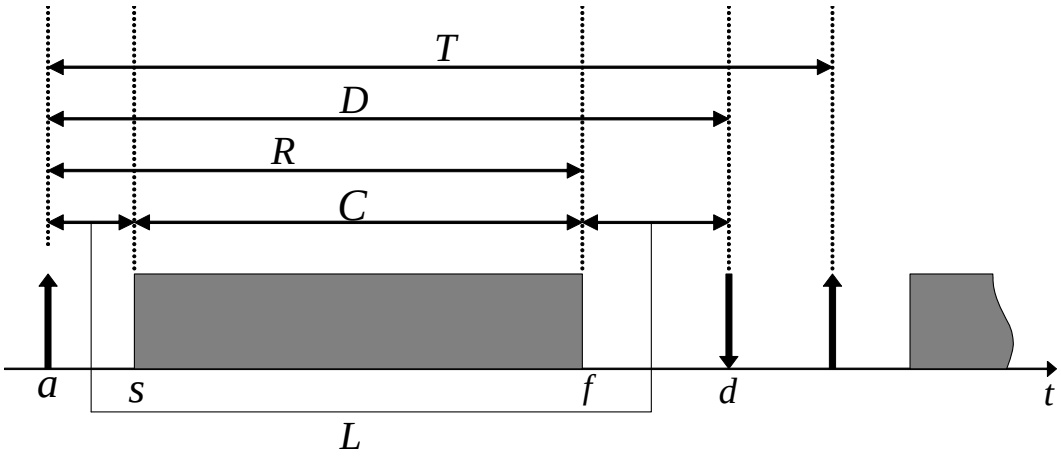

図 2.4: タスクの時間情報

2.3.3 μITRON 仕様 OS

μITRON は，リアルタイムの組込み機器制御システム用の OS であり，TRON プロジェクトのサブプロジェクトである．μITRON の機能を以下に挙げる．なお，各々の機能の詳細については，[2] を参照されたい．

- タスク管理機能

  タスクを管理するための情報として，タスク ID(タスクの名前)，タスクの状態，タスクの優先度などがある．これらの管理情報はタスクコントロールブロック (TCB) に格納される．タスクの管理はすべて TCB の情報に基づいて行われる．

  タスクの状態は，実行状態 (RUNNING)，実行可能状態 (READY)，待ち状態 (WAITING)，強制待ち状態 (SUSPENDED)，二重待ち状態 (WAITING-SUSPENDED)，休止状態 (DORMANT)，未登録状態 (NON-EXISTENT) という 7 つの状態をとりうる（図 2.5）．

  タスクのスケジューリングは，システム開発者が静的に決定したタスクの優先度に基づいて次のように行われる．タスクが複数ある場合は，タスクの中で優先度が最も高いものを実行状態 (RUNNING) とし，他のタスクは実行可能状態とする．優先度が同じタスクが複数ある場合は，FirstCome FirstServed 方式の原則により，それらの中で先に実行できる状態になったタスクから実行する．優先度が低いタスクを実行中に優先度が高いタスクが実行可能状態となると，優先度が高いタスクによって優先度が低いタスクの実行は中断される．このことをプリエンプト (Preempt) という．

- タスク付属同期機能

  タスクを起床待ち状態にしたり，待ち状態から起床させたりする等，タスクの状態を直接的に操作することで，同期を行うための機能が含まれる．

- 同期・通信機能

  タスク間の同期・通信機能として，セマフォ，イベントフラグ，メールボックスなどの機能が含まれる．

- メモリプール管理機能

  メモリプール管理機能は，アプリケーション内で動的なメモリ確保を行う場合に使用される．

- 時間管理機能

  時間管理機能は，時間に依存した処理を行うための機能である．システム時刻を設定/参照/更新する機能が含まれる．

- システム状態管理機能

  システム状態管理機能は，システム状態を変更/参照するための機能である．具体的にはタスク優先度変更，CPU ロック状態への移行/解除，タスクディスパッチを禁止/解除機能が含まれる．

- 割込み管理機能

  割込み管理機能は，外部割込みによって起動される割込みハンドラの定義や割込みサービスルーチンを生成/削除する機能が含まれる．割込みサービスルーチンは，割込みハンドラから起動される．

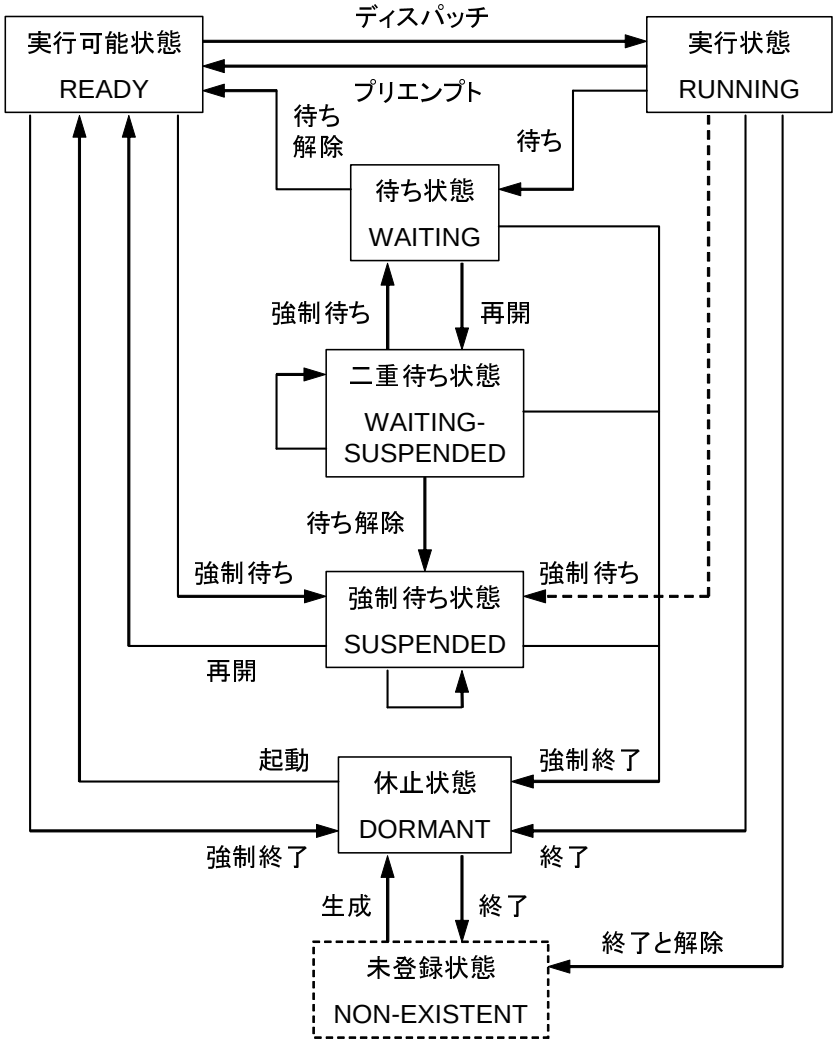

図 2.5: タスクの状態遷移

μITRON 仕様では，仕様全体としては「弱い標準化」の方針をとりつつ，ソフトウェアの移植性を向上させるために標準的な機能セットとその仕様を「スタンダードプロファイル」として強く規定している．

2.3.4 μITRON 仕様ソフトウェア部品

汎用システム開発において，ソフトウェア部品の利用が一般的である．同様に，近年の組込みシステム開発の大規模化・複合化の流れの中で組込みシステムのためのソフトウェア部品の重要性が広く認識されつつある．これに伴い，μITRON4.0 仕様では，静的 API という機構がある．静的 API は，これまで実装ごとに定められていた静的オブジェクト情報の記述方法を一本化し，ソフトウェア部品の流用を容易にすることを目的として導入された．

静的 API を用いて，静的に作成されるオブジェクトの初期化情報または初期化コードを生成する過程をコンフィギュレーションという．また，コンフィギュレーションを行うためのツールをコンフィギュレータという．コンフィギュレータは，カーネル，ソフトウェア部品毎に別々に用意する．カーネルやソフトウェア部品の静的 API 等をシステムコンフィギュレーションファイルに記述する．

その処理の流れを図 2.6 に示す．システムコンフィギュレーションファイルは，まず，C 言語のプリプロセッサに通される．次にソフトウェア部品のコンフィギュレータによって順に処理され，最後にカーネルのコンフィギュレータによって処理される．

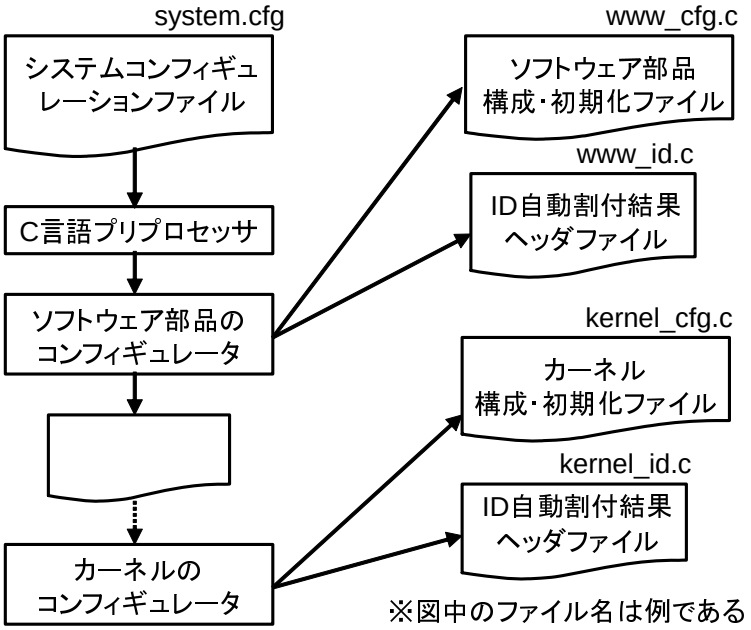

図 2.6: コンフィギュレーション処理手順

第3章　DBMSライブラリの実装

本研究では，ARMプロセッサ上で動作するμITRON4.0スタンダードプロファイル仕様のRTOSをターゲットとしたDBMSをソフトウェア部品として実装する．本章では，DBMSライブラリの具体的な実装方法を述べた後，DBMSライブラリのリアルタイム性を保ちつつ，実行バイナリコードを削減する手法について述べる．

3.1　DBMSライブラリの設計方針

本研究で実装するDBMSライブラリは，前述の組込みデータベースの要求事項を実現することを目指したものである．

各関数は，組込みシステムにおける適応化の概念を取り入れたものとし，関数毎にプリミティブな機能を提供する．また，ITRON仕様OS上での利用を想定し，ITRON仕様を踏襲してライブラリ関数を構成する．なお，DBMSライブラリを作成するにあたり，Empress Software社[6]のEmpress EmbeddedのAPIを参考にした．

3.2　μITRON4.0仕様DBMSライブラリの共通規定

3.2.1　用語の定義

- API(Application Program Interface)

  アプリケーションプログラムがカーネルやソフトウェア部品を使用する場合に用いるインターフェースのことである．DBMSライブラリ関数は，アプリケーションプログラムから呼び出されるサービスコールであるため，APIに含まれる．

- データベース各部の名称

  データベースは，複数のテーブルから構成される．テーブルは複数のレコードで構成され，レコードの属性をフィールドという（図3.1）．

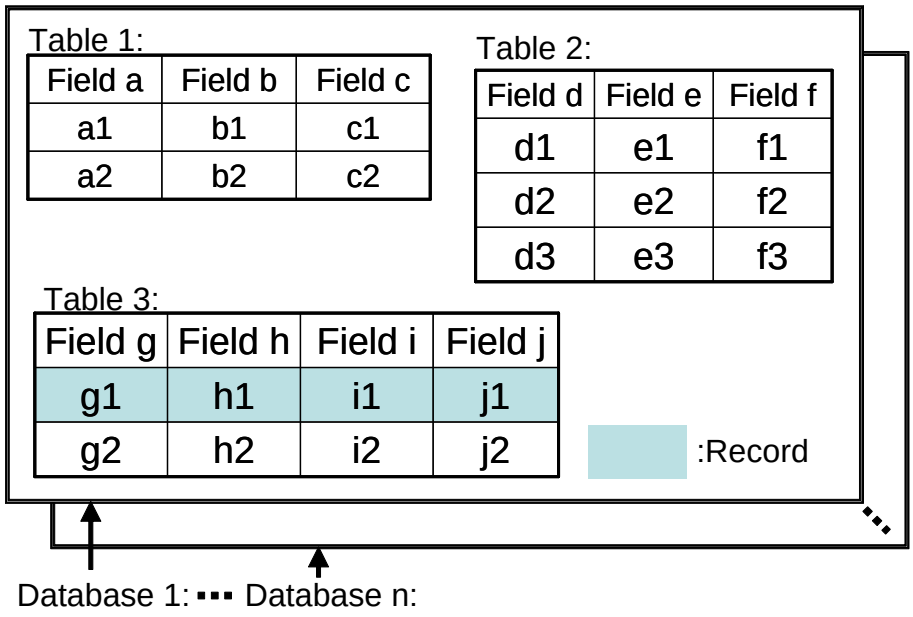

図 3.1: データベース各部の名称

3.2.2 API の名称に関する原則

- ソフトウェア部品識別名は，標準化されたソフトウェア部品の名称の衝突を避けるために用いられる．本研究で実装する DBMS ライブラリの識別名は，大文字で「SHOULDER」，小文字で「shoulder」とする．

- カーネルのサービスコールの名称は，xxx で操作の方法，yyy で操作の対象を表し，xxx_yyy の形を基本とする．それに従い，DBMS ライブラリのサービスコールの名称は，shoulder_xxx_yyy の形とする．

- 静的 API は，コンフィギュレーションファイルに記述された情報を元に，システム初期化時にデータベースのデータ構造とデータ格納領域を自動的に生成する．静的 API は，大文字で記述することで，通常の API(動的 API) と区別する．

- DBMS ライブラリの API を実装する際に使用するデータ型は，ITRON 仕様 OS 上での利用を想定し，ITRON 仕様で定義されたものを踏襲する．独自のデータ型の名称については，大文字で以下の原則を設ける．

  T_SHOULDER_〜 DBMS ライブラリで用いる構造体

  T_SHOULDER_CYYY〜 shoulder_cre_yyy に渡すパケットのデータ型

  その他定数やマクロの名称に関しても，ITRON 仕様の規定に従う．

- DBMS ライブラリの仕様で定められるサービスコールの宣言などを含むヘッダファイルの名称と DBMS ライブラリのコンフィギュレータが生成する自動割付け結果ヘッダファイルの名称は，ソフトウェア部品識別名で始まる名称とする．

- DBMS ライブラリの内部に閉じて使われるルーチンやメモリ領域などの識別子の内，オブジェクトファイルのシンボル表に登録され外部から参照できるものを内部識別子という．DBMS ライブラリは，アプリケーションとリンクされるため，アプリケーションプログラムとの名称の衝突が発生しかねない．これを避けるために，DBMS ライブラリの内部識別子は，C 言語レベルで，_shoulder_，または_SHOULDER_で始まる名称とする．

3.2.3 API の返値とエラーコード

各 API の返値は，ITRON 仕様に準拠し，エラーが発生した場合には負の値のエラーコード，正常に実行された場合には 0 または正の値とする．正常実行された場合の返値の意味は関数毎に定義される．

エラーコードのニーモニックと値および意味は，ITRON カーネル仕様のエラーコードと同じになるように標準化する．ただし，ITRON カーネル仕様で足りないエラーコードは追加定義する．

3.2.4 データ構造

静的 API によってメモリに確保されるデータベース用領域を以下に挙げる

- データベース管理ブロック (DBS_CB)
  データベース属性，データベース名，テーブルの個数，テーブル管理ブロックへのポインタを保持

- テーブル管理ブロック (TBL_CB)
  テーブル属性，テーブル名，フィールド管理ブロックへのポインタを保持

- フィールド管理ブロック (FLD_CB)
  フィールド属性，フィールド名，データタイプ，データオフセットを保持

- レコード管理ブロック (REC_CB)
  レコードの有効フラグを保持

API で，特定のデータへアクセスする際は，ポインタ演算を行う．以下の 5 つのデータへのポインタを定義する．

- テーブル記述子 (table_descriptors)
  オープンされているテーブルを指す
- フィールド記述子 (attribute_descriptors)
  指定されたのフィールドを指す
- レコード記述子 (record_descriptors)
  指定されたレコードを指す
- 条件記述子 (qualification_descriptors)
  テーブルやレコードと，条件を関連付ける
- 検索記述子 (retrieval_descriptors)
  テーブルと条件を関連付ける

これらが一つのデータベースで必要である．データベース用領域のメモリ配置を示す（図 3.2）．

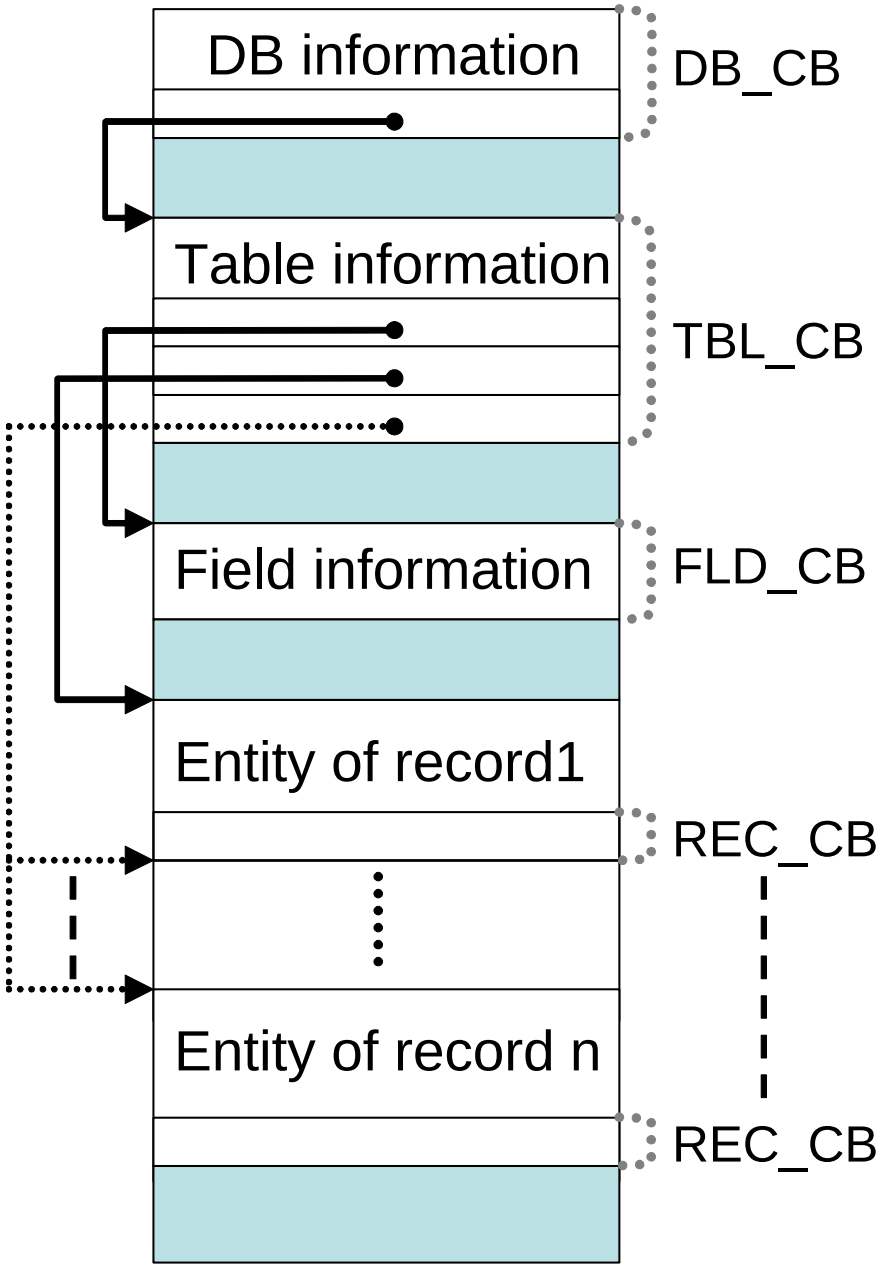

図 3.2: データベース領域のメモリ配置

3.3 DBMS ライブラリの機能

データベースで扱えるデータタイプは，4 バイト整数型 (INTEGER)，倍精度浮動小数点型 (DOUBLE)，可変長文字列型 (CHAR) である．

実装した DBMS ライブラリ API の一覧を示す（表 3.1）．

表 3.1: DBMS ライブラリ API 一覧

API 名	API の種類	機能
SHOULDER_CRE_DBS	静的 API	静的オブジェクトを生成
shoulder_open_dbs	動的 API	テーブルをオープンする
shoulder_ngeta_dbs	動的 API	フィールド名からフィールド記述子を取得
shoulder_mkrec_dbs	動的 API	レコードを保存するためのスペースを割当てる
shoulder_putvs_dbs	動的 API	レコード内のフィールドに文字を格納する
shoulder_add_dbs	動的 API	テーブルにレコードを挿入する
shoulder_addend_dbs	動的 API	レコードの挿入を確定する
shoulder_frrec_dbs	動的 API	shoulder_mkrec_dbs で確保したスペースを開放する
shoulder_close_dbs	動的 API	テーブルをクローズする
shoulder_igeta_dbs	動的 API	フィールド番号に対応するフィールド記述子を取得
shoulder_ganame_dbs	動的 API	フィールド記述子からフィールド名を取得する
shoulder_spv_dbs	動的 API	フィールドの値を保存するスペースを割当てる
shoulder_free_dbs	動的 API	shoulder_spv_dbs が割当てたスペースを開放する
shoulder_getbegin_dbs	動的 API	検索条件をレコードと関連付ける
shoulder_get_dbs	動的 API	テーブルからレコードを検索する
shoulder_copyv_dbs	動的 API	文字列型データを割当てられたスペースにコピーする
shoulder_copyi_dbs	動的 API	整数型データを変数にコピーする
shoulder_getend_dbs	動的 API	shoulder_getbegin_dbs が割当てた検索用スペースを開放する
shoulder_srtbegin_dbs	動的 API	検索条件をレコードと関連付け，ソートする
shoulder_strcpy	動的 API	文字列コピーをする内部関数
shoulder_strcmp	動的 API	文字列比較をする内部関数
shoulder_strlen	動的 API	文字列の長さを返す内部関数
shoulder_memcpy	動的 API	メモリの内容をコピーする内部関数
shoulder_atoi	動的 API	文字列型から整数型へ変換する内部関数
shoulder_malloc	動的 API	メモリを確保する内部関数

これらの API を用いてデータを挿入する場合とデータを検索する場合の例をフローチャートで示す（図 3.3，図 3.4）．

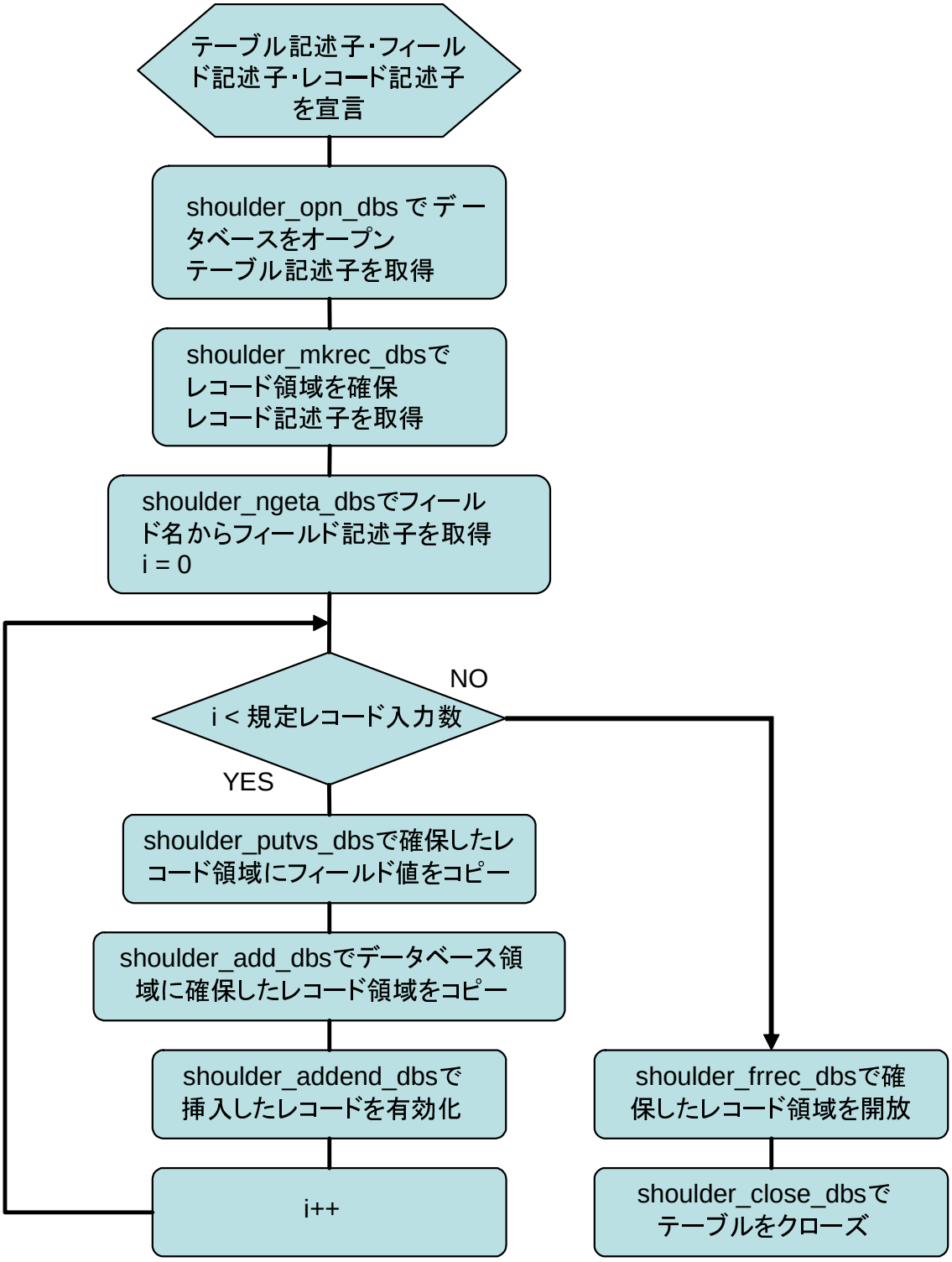

図 3.3: 挿入操作のフローチャート

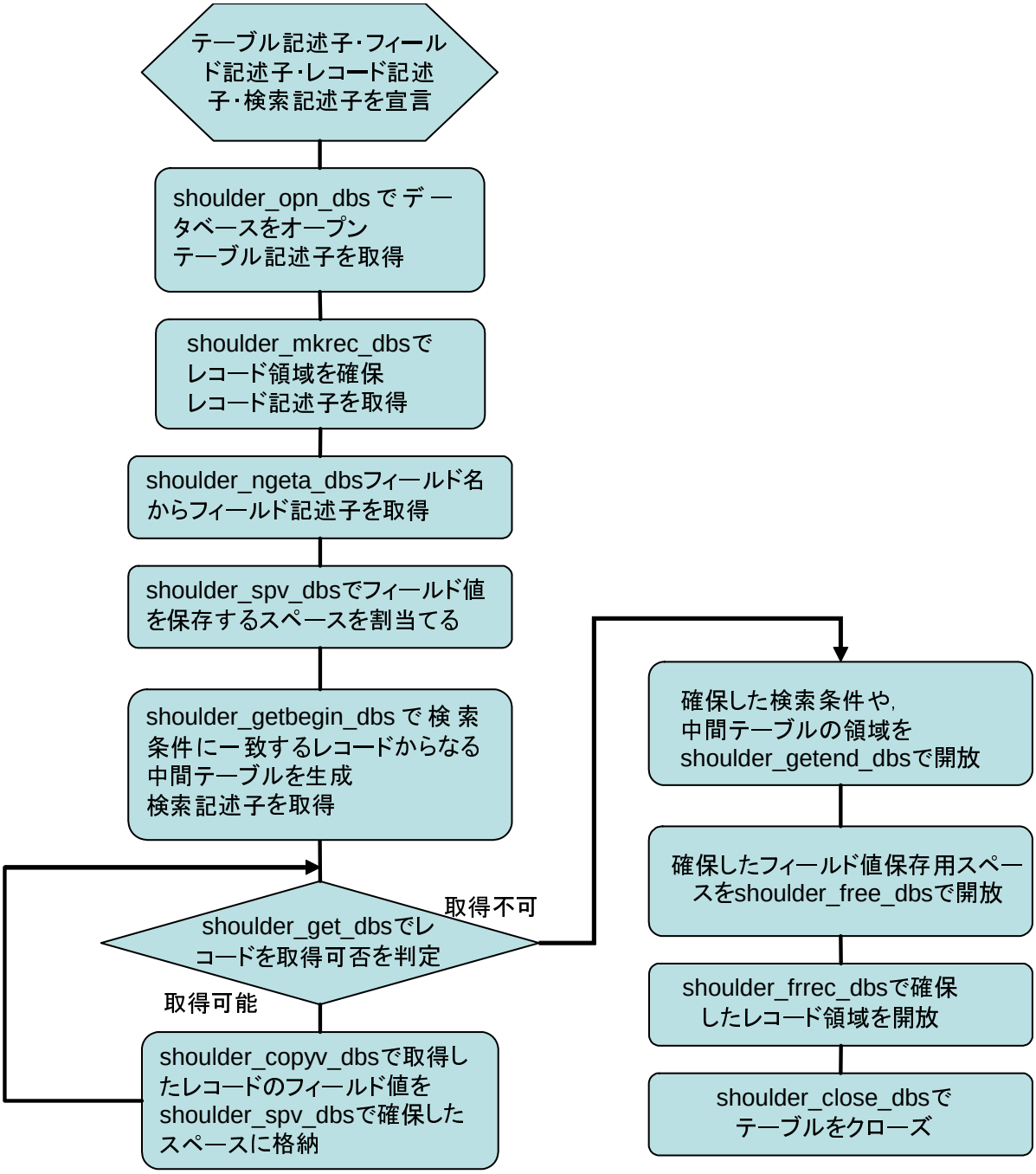

図 3.4: 検索操作のフローチャート

3.4 DBMS ライブラリの最適化

第 2 章でも述べたが，ARM プロセッサは，32 ビット ARM 命令を実行する ARM モードと，16 ビット Thumb 命令を実行する Thumb モードという 2 つのモードを持つ．また，μITRON 仕様 OS はシステム開発者が静的に決定したタスクの優先度を基にスケジューリングを行う．この際，優先度の値が小さいタスクほど優先して処理される．

本手法はこれらのターゲットシステムが持つ機能を利用することで実現できる．本研究では，ARM モード用と Thumb モード用の DBMS ライブラリをそれぞれ提供する．これらのライブラリは，両方とも同様な機能の関数を備えているが，ARM モード用 DBMS 関数は実行速度が速くなるように，Thumb モード用 DBMS 関数はコードサイズが小さくなるように，それぞれ最適化がなされている．両方のライブラリから同一の関数がリンクされると，実行バイナリサイズが増加してしまうため，メモリ制約の厳しい組込みシステムには適さない．したがって，どちらか一方のライブラリ関数を選択する必要がある．本手法では，ライブラリ関数を利用するタスクの優先度を指標とし，どちらを選択するか判断する．具体的な選択アルゴリズムを以下に述べる．

タスクの優先度にある閾値を与える．この閾値を最適化レベルという．最適化レベル以上のタスクから呼ばれる DBMS のライブラリ関数は，Thumb モードのライブラリから選択されるようにする．それ以外のタスクから呼ばれるものは，ARM モードのライブラリから選択される．この中で，複数のタスクから呼ばれるものについては，優先度が一番高いタスクの優先度と最適化レベルを比較し，どちらを選択するか判断する．最適化レベルの決定はシステム開発者が行うものとする．

本手法により，最適化レベル以上のタスクから呼ばれる DBMS ライブラリ関数が Thumb モードになるため，全て ARM モードのものを使用した場合と比べ，システムの実行バイナリに占める DBMS ライブラリ関数のコードサイズを削減することができる．また，Thumb モードではデッドラインミスしてしまうような余裕時間が少ないタスクを ARM モードとすることで，リアルタイム性を保つこともできる．

3.5 DBMS ライブラリ関数の自動選択

本研究では，前述の提案手法を自動的に実行するリンクツールを実装する．このリンクツールでシステムの高効率化を自動的に行うことで，システム開発者の負担を軽減することができる．リンクツールの実装方法について以下に述べる．

DBMS ライブラリ関数の選択に必要な情報は，最適化レベル，タスク毎の優先度，DBMS ライブラリ関数の利用情報である．最適化レベルは，システム開発者が決定する値であるため，引数として渡される．タスク毎の優先度は，システムコンフィギュレーションファイルに記述されている静的 API の情報を用いる．DBMS ライブラリ関数の利用情報は，システム開発者が記述する．リンクツールはこれらの情報を基に，ARM モードと Thumb モードのライブラリ関数のオブジェクトファイルのどちらかを指すシンボリックリンクを自動生成する．本手法の適用例を次に示す．

提案手法の適用例

優先度が 1 の task_a と優先度が 7 の task_b があるとする（図 3.5）．この例では，task_a の優先度の値のほうが task_b のそれよりも小さいため，task_a のほうが task_b よりも優先的に実行される．

ここで，最適化レベルを 7 とする．リンクツールは，システムコンフィギュレーションファイル (system.cfg) から得るタスクの優先度と DBMS ライブラリ利用情報 (shoulder_optimize_target.txt) から，必要な分だけの DBMS ライブラリ関数へのシンボリックリンクを以下のように生成する．task_a で呼ばれている shoulder_getbegin() という DBMS ライブラリ関数は ARM モード用にリンクされる．task_b で呼ばれている shoulder_srtbegin() という DBMS ライブラリ関数は Thumb モード用にリンクされる．そして，システムコンパイル時にこれらの実体がリンクされることにより，DBMS ライブラリが最適化される．Thumb モード用 DBMS ライブラリ関数は ARM モード用のそれに比べてコードサイズが小さくなるため，タスクセットのコードサイズを削減することができる．

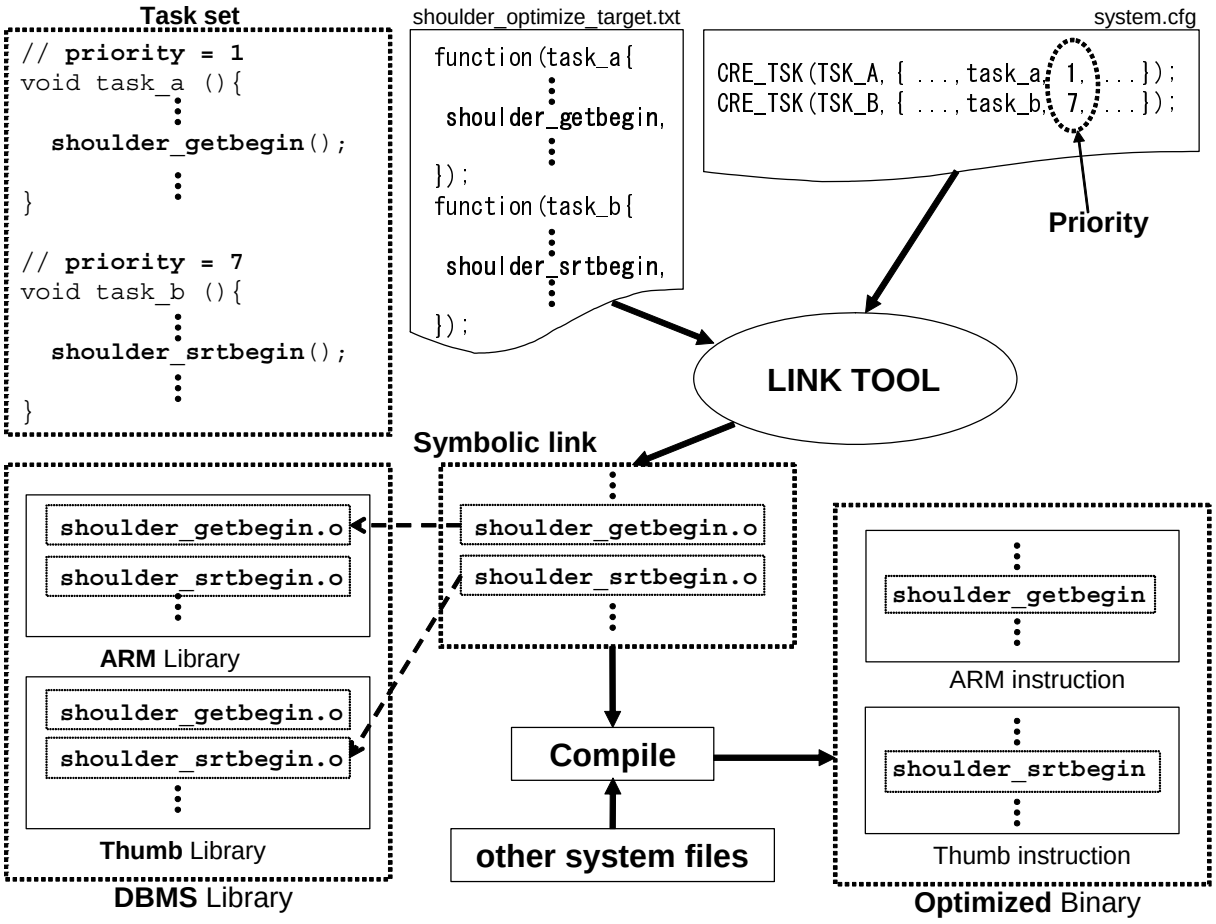

図 3.5: 実行バイナリコードの最適化

3.6 DBMS ライブラリのコンフィギュレーション

実装する DBMS ライブラリは，静的なオブジェクトを生成する静的 API を導入しているため，それを処理する独自のコンフィギュレーションが必要である．本研究で実装するコンフィギュレータには，基本的なコンフィギュレーション機能と前述のリンクツールの機能を搭載する．本節では，システムコンフィギュレーションファイルに DBMS ライブラリの静的 API を記述する方法と，システム全体のコンフィギュレーションの過程について述べる．

3.6.1 静的 API の記述方法

作成したいデータベースの定義情報は，システム開発者がシステムコンフィギュレーションファイルに静的 API として記述する．コンフィギュレータは，その静的 API を処理してデータベース初期化プログラムを生成する．そして，システム初期化時に生成されたデータベース初期化プログラムが実行されることにより，データベースの管理領域とデータ領域が生成される．

データベースを定義する静的 API の記述方法を以下に述べる．整数型の *number*，文字列型の *name* と *phone* という 3 つの属性を持つ *student* というテーブルを，データベース *school* に登録するという例を挙げる．SHOULDER_CRE_DBS の () 内の最初には，データベースの ID，属性，名前，テーブル数を記述する．テーブル数の次に “{” “}” を記述し，その中にテーブルの属性，名前，レコード数，フィールド数を列挙する．これらをテーブル管理情報という．テーブルが複数ある場合はカンマで区切って以降同様にテーブル管理情報を記述する．テーブルの定義を終えたら，フィールド数の次に “{” “}” を記述し，その中にフィールドの名前，データ型，データのバイト数を列挙する．これらをフィールド管理情報という．フィールドが複数ある場合は，カンマで区切って以降同様にフィールド管理情報を記述する．以下に記述例を示す．

```
SHOULDER_CRE_DBS(DBS_A,ATR_D,school,1,{ATR_DS,student,10,3,{
{number,INTEGER,4},{name,CHAR,25},{phone,CHAR,15}}}
```

3.6.2 コンフィギュレーションの過程

DBMS ライブラリをシステムに組込む際の全体のコンフィギュレーションの過程を図 3.6 に示す．システムコンフィギュレーションファイル (system.cfg) は，最初に C 言語プリプロセッサにかけられる．次に，DBMS ライブラリのコンフィギュレータは，渡されたシステムコンフィギュレーションファイルの中から自分自身の静的 API を解釈する．この時，自分自身の静的 API の記述にエラーがある場合は，そのエラーを報告する．そして，データベースの構成や初期化に必要なファイルを C 言語のソースファイル (shoulder_cfg.c) の形で，ID 自動割付結果とデータ構造情報のヘッダファイルを C 言語のヘッダファイル (shoulder_id.h) の形で生成する．また，システムコンフィギュレーションファイルから自分自身の静的 API を削除し，以降のコンフィギュレータに対する静的 API を追加し，次

のコンフィギュレータに渡す．この際に追加する静的 API は，データベースのデータ構造の作成や初期化を実行するプログラムを登録する API と，データベースの管理やデータ領域で使用する固定長メモリプールを登録する API である．

また，このコンフィギュレータは，本研究の提案手法であるリンクツールの機能を備えている．ゆえに最適化レベル，システムコンフィギュレーションファイル，DBMS ライブラリ利用情報 (shoulder_optimize_target.txt) を解釈してこの段階で DBMS ライブラリ API のオブジェクトファイルを最適化することができる．最適化された DBMS ライブラリ関数のシンボリックリンクの作成情報は，shoulder_optimizer.csh というスクリプトファイルにまとめられる．このスクリプトファイルは，実行バイナリを make する段階で実行される．これらの生成されたファイル群はコンパイラに渡され，一つの実行バイナリとなる．

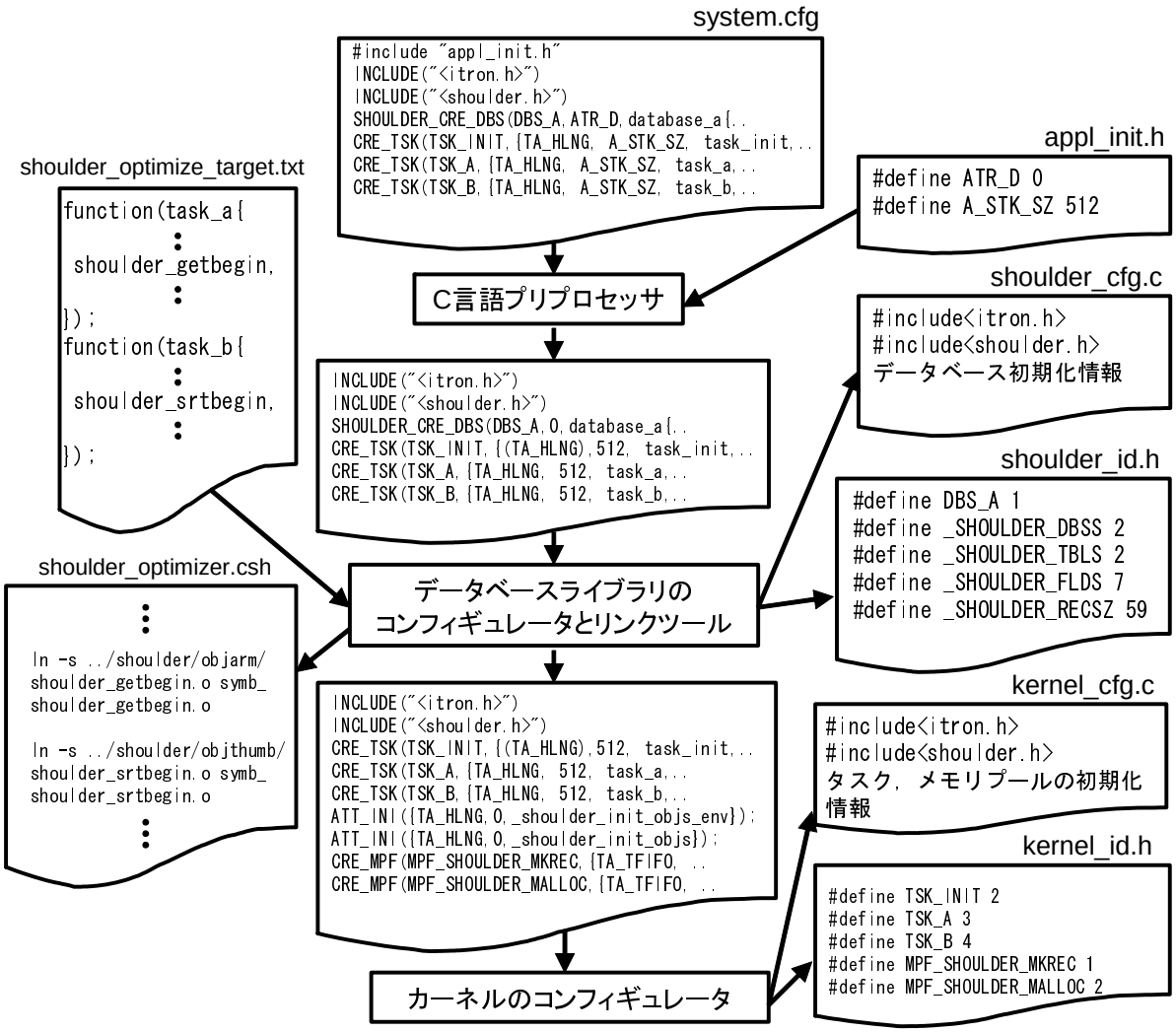

図 3.6: システムコンフィギュレーションファイルの処理

第4章　評価

本章では，シミュレーション環境について述べた後，提案手法をシミュレーションで評価した結果とその考察について述べる．

4.1　シミュレーション環境

本節では，評価に用いる ARM シミュレータと μITRON 仕様 OS について述べる．

4.1.1　ARM シミュレータ

ARM7TDMI の Version4T[1] に準拠したシミュレータを作成した．このシミュレータは，ARM モード，Thumb モードを備えており，クロックサイクルレベルシミュレーションが可能である．組込み向けのプロセッサのため，キャッシュは搭載していない．命令は基本的に 1 命令で 1 クロックサイクルで実行するが，以下のペナルティがかかるものがある．

- 乗算命令：+2 クロックサイクル
- 複数ロード/ストア命令：+ レジスタ数 − 1 クロックサイクル

タスクセットの実行を完了すると，CPU 使用率，ARM と Thumb の実行比率，各タスクの平均応答時間，デッドラインミス数などを出力する．また，14 本の外部割込み（以下，単に割込みと呼ぶ）チャネルを持ち，設定ファイルに任意のクロックを記述することで割込みを周期的，非周期的に発生させることができる．その内の 4 本はタイマー機能としており，4 つの 16 ビットカウンタを内部に持つ．カウンタ値がオーバーフローすると割込みが発生する．カウンタの初期値は memory mapped I/O を介して設定することができる．通常の割込みは，非周期タスクの生成に用いる．また，タイマー機能は OS の時間を管理する際に用いる．

4.1.2 μITRON 仕様 OS

本研究では，μITRON4.0 スタンダードプロファイル仕様のカーネルライブラリ [7] を用いて ARM プロセッサで動作する OS を実装する．このカーネルライブラリは，機能毎に C 言語で記述されており，移植性に優れていることに特徴がある．しかし，ソフトウェア割込みを発生させる処理や割込みハンドラ等，ターゲットシステムによって固有の部分が存在する．そのため，本研究ではソフトウェア割込みハンドラと割込みハンドラを ARM アセンブリコードで実装した．以下にその実装方法を述べる．

ソフトウェア割込みハンドラ

ARM プロセッサでソフトウェア割込みを発生させるには，ソフトウェア割込み命令（SWI 命令）を使用する．SWI 命令を実行すると，SWI 命令の次の命令を指すアドレスとプロセッサ状態レジスタ (CPSR) が特権モード（SVC モード）のリンクレジスタ (R14_svc) とプロセッサ状態保存レジスタ (SPSR_svc) に保存される．そしてユーザモード（USR モード）から強制的に ARM モードの SVC モードになり，割込みを禁止して例外ベクタアドレス 0x8 番地にジャンプする．0x8 番地にソフトウェア割込みハンドラへ分岐するように記述すれば，ソフトウェア割込みハンドラが実行できる．ソフトウェア割込みの種類を判別するには，ソフトウェア割込みハンドラで SWI 命令の 24 ビット即値部分をロードし，値の比較を行うことで決定できる．SWI 命令の 24 ビット即値部分を SWI 番号という．

実装したソフトウェア割込みハンドラのフローチャートを図 4.1 に示す．プログラムの最初と最後にコンテキストの *save* と *restore* を行う．一般にソフトウェア割込みは，ユーザモードで動作するプログラムから特権モードを必要とするシステムコールを呼出す際に使用する．本研究では，SWI 番号の 1 番を CPU のロック（割込み禁止），2 番を CPU のアンロック（割込み許可），3 番をディスパッチに割当てた．SWI 番号 1，2 のシステムコールは，CPSR の割込み許可ビットを変更することで実現できる．

ディスパッチは，TCB の中の DYN_FLG の値によって動作が異なる．DYN_FLG が “0” のタスクは初期状態，“1” のタスクは途中状態を表す．現在実行中のタスクが途中状態の場合にプログラムカウンタ，スタックポインタを TCB に保存する．初期状態のタスクは保存操作を行わない．そして，スケジューラが次に実行すると決定したタスクに実行が切替った後，そのタスクの DYN_FLG を判断する．現在実行中のタスクが初期状態の場合，そのタスクを途中状態にする．また，この後 *restore* する要素が存在しないため，あらかじめスタック領域に初期値をストアしておく．こうすることにより，DYN_FLG の状態に関係なく，*restore* のパスが共通となる．

割込み処理

ARM プロセッサで割込みを発生させるには，CPSR の割込み禁止ビットが “0” であることが前提条件である．割込みが発生すると，割込みが発生した命令の次の命令を指すアドレスとプロセッサ状態レジスタ (CPSR) が特権モード（IRQ モード）のリンクレジスタ (R14_irq) とプロセッサ状態保存レジスタ (SPSR_irq) に保存される．そしてユーザモード（USR モード）から強制的に ARM モードの IRQ モードになり，割込みを禁止して例外ベクタアドレス 0x18 番地にジャンプする．0x18 番地に割込みハンドラへ分岐するように記述すれば，割込みハンドラが実行できる．

μITRON4.0 仕様では，割込みハンドラは，プロセッサの機能のみに依存して起動することを基本としている．したがって，割込みコントローラの操作はカーネルではなく割込みハンドラで行う．

実装した割込みハンドラのフローチャートを図 4.2 に示す．プログラムの最初と最後にコンテキストの *save* と *restore* を行う．割込みハンドラ内で C 言語で記述された割込みハンドラを呼び出している．この割込みハンドラは多重割込みをサポートしている．すなわち，既存の割込みを処理中に割込みを再許可し，さらなる割込みを発生させることができる．この際，処理中の割込みと同じまたはそれより低いデバイス番号のデバイスの割込みを無視するように割込み許可レジスタを変更し，デバイス番号の値が小さいデバイスほど優先して処理されるようにしている．

割込みハンドラ内でタスクが起動される場合，割込みハンドラが実行されている間はディスパッチ保留状態であり，ディスパッチは起こらない．多重割込みが全て処理されてからディスパッチャが実行され，タスクのディスパッチが行われる．

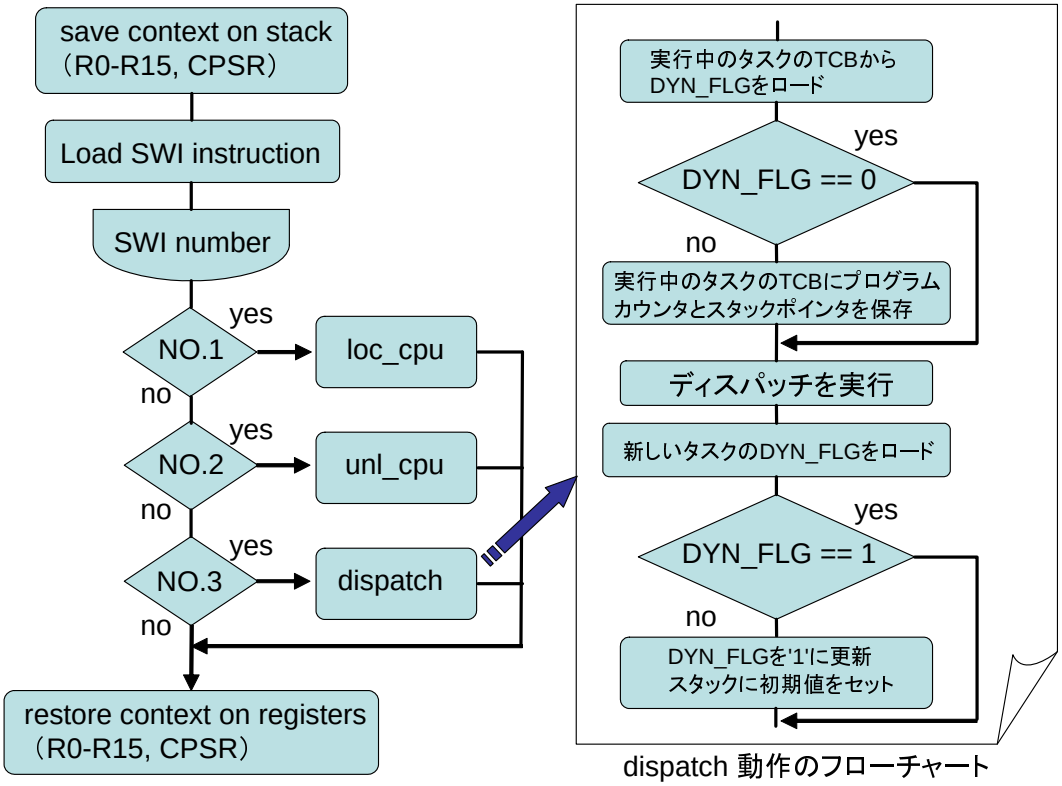

図 4.1: ソフトウェア割込みハンドラのフローチャート

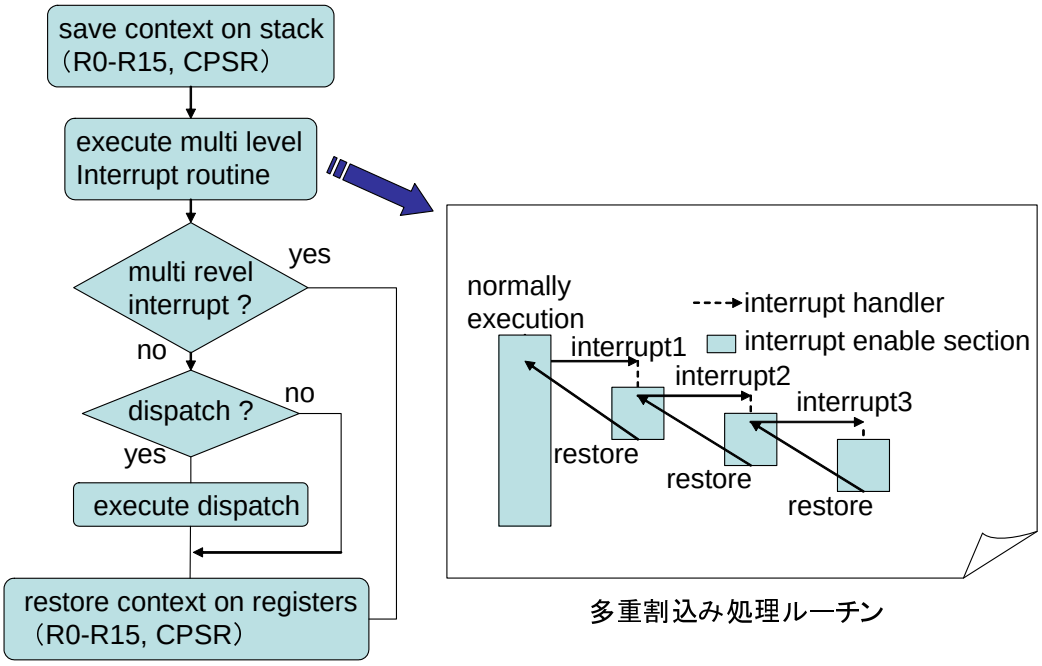

図 4.2: 割込みハンドラのフローチャート

4.1.3 入力バイナリコードの作成

シミュレータに入力するバイナリコードの生成過程を図 4.3 に示す．タスク，OS のカーネルライブラリ，ソフトウェア割込みハンドラ，割込みハンドラ，DBMS ライブラリの API は各々の処理過程を経て，最終的に一つのバイナリとなる．なお，本研究で使用する C 言語クロスコンパイラは GCC Ver3.3.2 である．

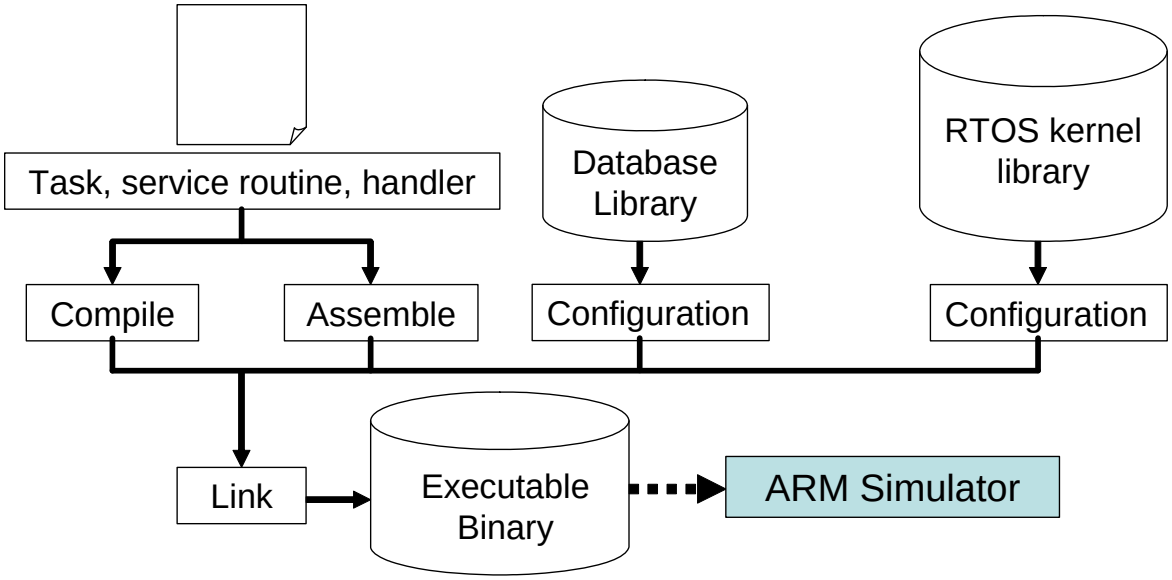

図 4.3: 入力バイナリコードの生成過程

4.2 評価関数の定義

提案手法のリアルタイム性を評価する際，デッドラインミス数と平均応答時間を指標とする．これらの指標を評価関数として以下に定義する．

- デッドラインミス数

  n 個のタスクが実行完了したとき，デッドラインミス数 N_{miss} を以下に定義する．

$$N_{miss} = \sum_{i=1}^{n} miss(f_i)$$

  ただし，$miss(f_i)$ は以下の定義とする．

$$miss(f_i) = \begin{cases} 0 & (f_i \leq d_i \text{のとき}) \\ 1 & (\text{それ以外のとき}) \end{cases}$$

- 平均応答時間
  n 個のタスクが実行完了したとき，平均応答時間 R_{avg} を以下に定義する．

$$R_{avg} = \frac{1}{n} \sum_{i=1}^{n} (f_i - a_i)$$

4.3 DBMS ライブラリの実行バイナリサイズ

DBMS ライブラリを ARM モードと Thumb モードでそれぞれコンパイルし，各 API の実行バイナリサイズと命令数を求めた．その結果のグラフを図 4.4，図 4.5 に示す．

ARM モードを基準としたときの Thumb モードの増加率を表 4.1 に示す．

表 4.1: Thumb モードの増加率

	増加率
命令数	1.28
コードサイズ	0.64

Thumb モードは ARM モードに比べて命令数が 28%増加し，コードサイズが 36%減少している．Thumb モードは ARM モードに比べて命令長が半分であるためコードサイズが減少する傾向がある．しかし，ARM は 3 オペランド方式に対して，Thumb モードは 2 オペランド方式であり，使用可能レジスタ数が ARM の半分になる等，1 命令あたりの演算が単純である．その結果，命令数が増加してしまい，コードサイズを半分にするまでには至っていない．

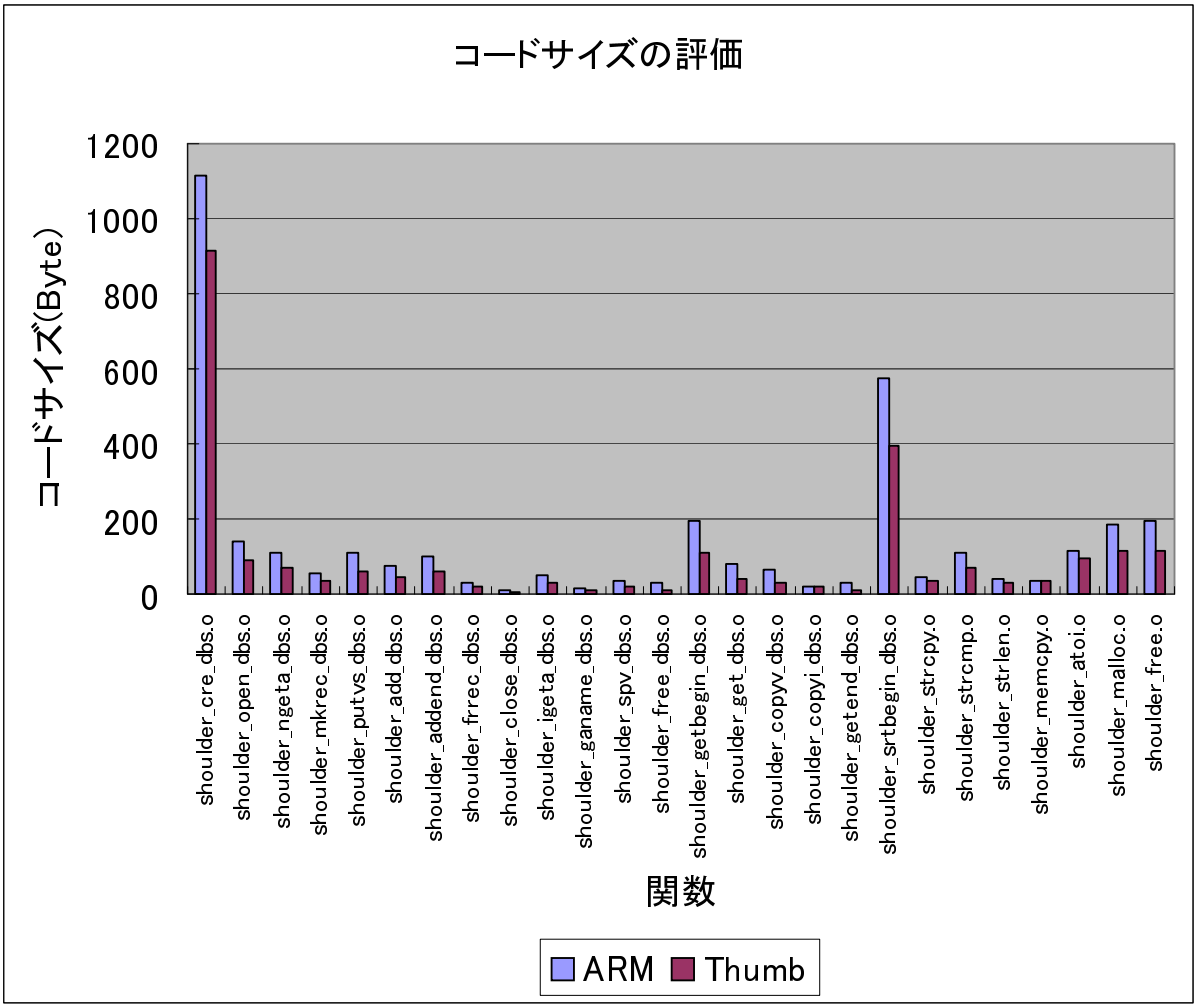

図 4.4: API のバイナリサイズ

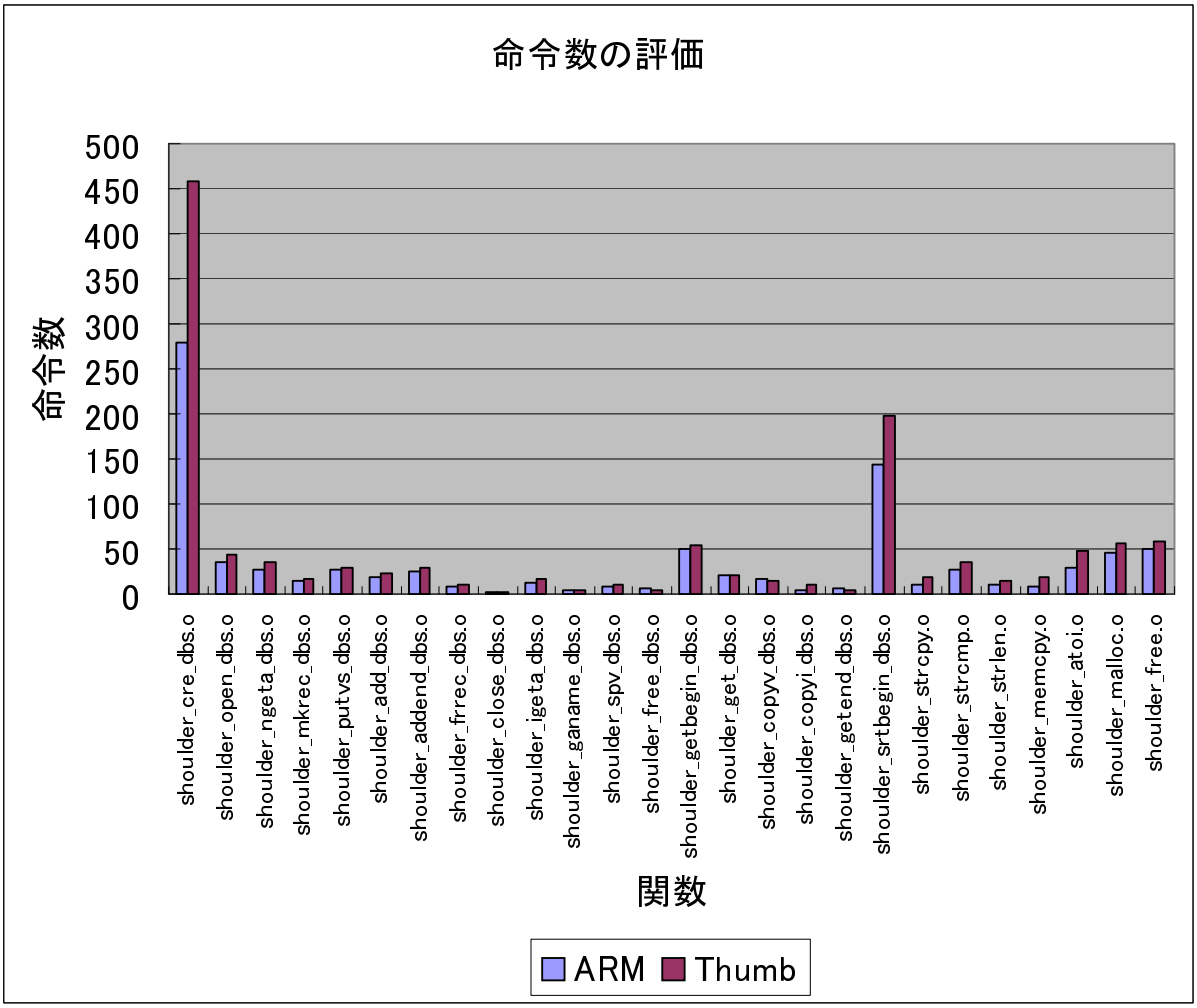
命令数の評価
命令数
500
450
400
350
300
250
200
150
100
50
0
shoulder_cre_dbs.o
shoulder_open_dbs.o
shoulder_ngeta_dbs.o
shoulder_mkrec_dbs.o
shoulder_putvs_dbs.o
shoulder_add_dbs.o
shoulder_addend_dbs.o
shoulder_frrec_dbs.o
shoulder_close_dbs.o
shoulder_igeta_dbs.o
shoulder_ganame_dbs.o
shoulder_spv_dbs.o
shoulder_free_dbs.o
shoulder_getbegin_dbs.o
shoulder_get_dbs.o
shoulder_copyv_dbs.o
shoulder_copyi_dbs.o
shoulder_getend_dbs.o
shoulder_srtbegin_dbs.o
shoulder_strcpy.o
shoulder_strcmp.o
shoulder_strlen.o
shoulder_memcpy.o
shoulder_atoi.o
shoulder_malloc.o
shoulder_free.o
関数
ARM
Thumb

図 4.5: API の命令数

4.4 タスクセット

評価に用いるタスクを表 4.4 に示す．表中における優先度は，値が小さいタスクほど優先順位が高くなることを表している．これらのタスクは，周期タスクと非周期タスクがある，以下のように定義する．

周期タスク (ID: 4〜20)

周期タスクの起動は，OS の周期ハンドラからタスクを起動させることで実現する．周期ハンドラから起動されるタスクは以下の規則性を持つ．

- ID: 4〜10

  周期ハンドラが呼ばれる度に起動.

- ID: 11〜15

  周期ハンドラが 3 回呼ばれるにつき 1 回起動.

- ID: 16〜20

  周期ハンドラが 4 回呼ばれるにつき 1 回起動.

周期ハンドラの基本周期は OS の時間管理の単位 $tick$ である．本研究のシミュレーション環境では，$tick = 8390$ クロックに設定されている．

この周期ハンドラの起動周期を変化させることで，CPU 使用率[1] が異なるタスクセットを作ることができる．

本研究では，周期ハンドラの起動周期を $30tick$ 〜 $200tick$ まで 1 ずつ変化させることにより，170 通りのタスクセットを作成した．このとき，周期ハンドラの起動周期 T_i は以下のように求まる．

$$T_i = tick * i \quad (30 \leq i \leq 200)$$

タスク ID が 4〜9 のタスクはリアルタイム性が要求される周期タスクである．それらについては，優先的に実行されるように優先度を高く設定し，相対デッドラインを周期より短い値とした．

それ以外の周期タスクの相対デッドラインは，周期タスクの起動周期とした．

[1] CPU 使用率 P_{usage} は，観測時間 t において実行時間 C のタスクが n 個完了した時，以下に定義される．

$$P_{usage} = \frac{1}{t} \sum_{i=1}^{n} C_i$$

表 4.2: 各タスクを構成するタスクの情報

タスク ID	優先度	周期/非周期	周期 (クロック)	相対デッドライン時間
2	2	非周期	-	8000
3	3	非周期	-	8725
4	4	周期	T_i	42361
5	4	周期	T_i	61062
6	4	周期	T_i	80403
7	4	周期	T_i	98948
8	5	周期	T_i	117414
9	5	周期	T_i	135995
10	6	周期	T_i	T_i
11	6	周期	$T_i * 3$	$T_i * 3$
12	6	周期	$T_i * 3$	$T_i * 3$
13	7	周期	$T_i * 3$	$T_i * 3$
14	7	周期	$T_i * 3$	$T_i * 3$
15	7	周期	$T_i * 3$	$T_i * 3$
16	7	周期	$T_i * 4$	$T_i * 4$
17	7	周期	$T_i * 4$	$T_i * 4$
18	7	周期	$T_i * 4$	$T_i * 4$
19	7	周期	$T_i * 4$	$T_i * 4$
20	7	周期	$T_i * 4$	$T_i * 4$
21	8	非周期	-	96167
22	8	非周期	-	110422
23	8	非周期	-	42587
24	8	非周期	-	146731
25	8	非周期	-	19162

非周期タスク (ID: 21〜25)

非周期タスクの起動要求は割込みにより発生させる．発生時刻を表 4.3 に示す．

表 4.3: 非周期タスクの起動要求発生時刻

タスク ID	起動要求発生時刻
21	900000
21	1800000
21	2520000
22	180000
22	1260000
22	2160000
23	540000
23	1620000
23	2700000
24	720000
24	1440000
24	1980000
25	360000
25	1080000
25	2340000

本タスクセットで使用する DBMS ライブラリ関数の集合を DBMS_API という．タスク中で，DBMS_API を使用しているタスクを以下に示す．

- ID3 は，データベースにデータを一括挿入する．優先度は 3 である．
- ID4 は，データベースからデータを検索し，検索条件に一致するデータを保存する．優先度は 4 である．
- ID10 は，データベースからデータを検索し，検索条件に一致するデータを昇順にソートして保存する．優先度は 6 である．

本タスクセットにおいて，DBMS ライブラリ最適化レベルに 3，4，6，7 を設定したとすると，実行バイナリ中の DBMS_API を以下のように場合分けすることができる．

- 最適化レベル 3 では，全て Thumb モード API となる．
- 最適化レベル 7 では，全て ARM モード API となる
- 最適化レベル 4，6 では，両モード API が混在するものとなる．

したがって，本研究では，これら 4 種類の実行バイナリを作成してシミュレーションを行い，これらを比較し評価を行う．

4.4.1 DBMS_API のコードサイズ

本タスクセットにおける DBMS_API の最適化レベルを 4，6 に設定し，実行バイナリを作成した場合に，DBMS_API 全体のコードサイズがどれほど変化するか，比較を行った．比較対象は，全て ARM の API(ARM_API)，全て Thumb の API(Thumb_API)，最適化レベル 4(OPT4_API)，最適化レベル 6(OPT6_API) である．比較結果を表 4.4 に示す．

表 4.4: DBMS ライブラリのコードサイズの比較

API の種類	コードサイズ (byte)	コードサイズ削減率 (%)
ARM_API	3480	0.00
最適化レベル 6	3298	5.22
最適化レベル 4	2940	15.51
Thumb_API	2412	30.69

表 4.4 より，Thumb_API は ARM_API よりコードサイズを約 30.7%削減している．OPT4_API は ARM_API よりコードサイズを約 15.5%削減している．OPT6_API は ARM_API よりコードサイズを約 5.2%削減している．

4.5 シミュレーション結果

各計測に対して全タスクの平均応答時間，システム全体のデッドラインミス数，デッドラインミスしたタスクの優先度の平均値，全体に占める Thumb モードの実行割合を観測する．観測時間は 3,300,000 クロックとする．シミュレーション結果を図 4.6，図 4.7，図 4.8，図 4.9 に示す．

図 4.6 は，周期ハンドラの起動周期に対するタスク全体の平均応答時間である．横軸は周期ハンドラの起動周期 ($tick$)，縦軸はタスク毎の平均応答時間をタスク全体で平均化したものである．

図 4.7 は，周期ハンドラの起動周期に対するデッドラインミス数である．横軸は周期ハンドラの起動周期 ($tick$)，縦軸はデッドラインミス数である．

図 4.8 は，周期ハンドラの起動周期に対するデッドラインミスしたタスクの平均優先度である．横軸は周期ハンドラの起動周期 ($tick$)，縦軸はデッドラインミスしたタスクの優先度の平均値である．

図 4.9 は，周期ハンドラの起動周期に対する全体に占める Thumb モードの実行割合である．横軸は周期ハンドラの起動周期 ($tick$)，縦軸は全体に占める Thumb モードの実行割合である．

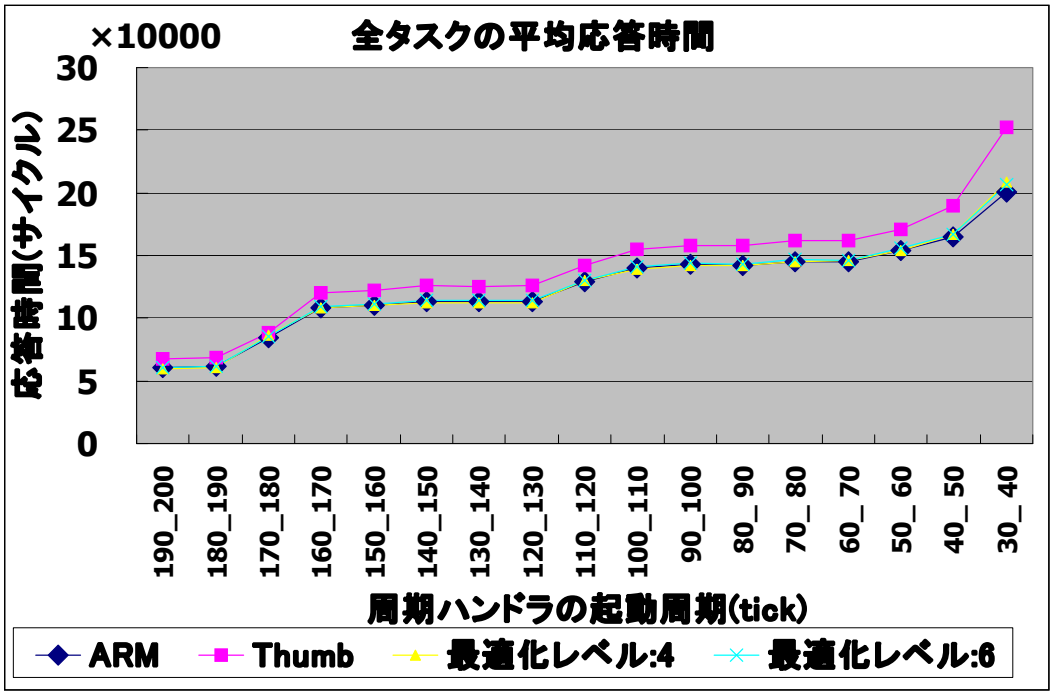

図 4.6: 全タスクの平均応答時間

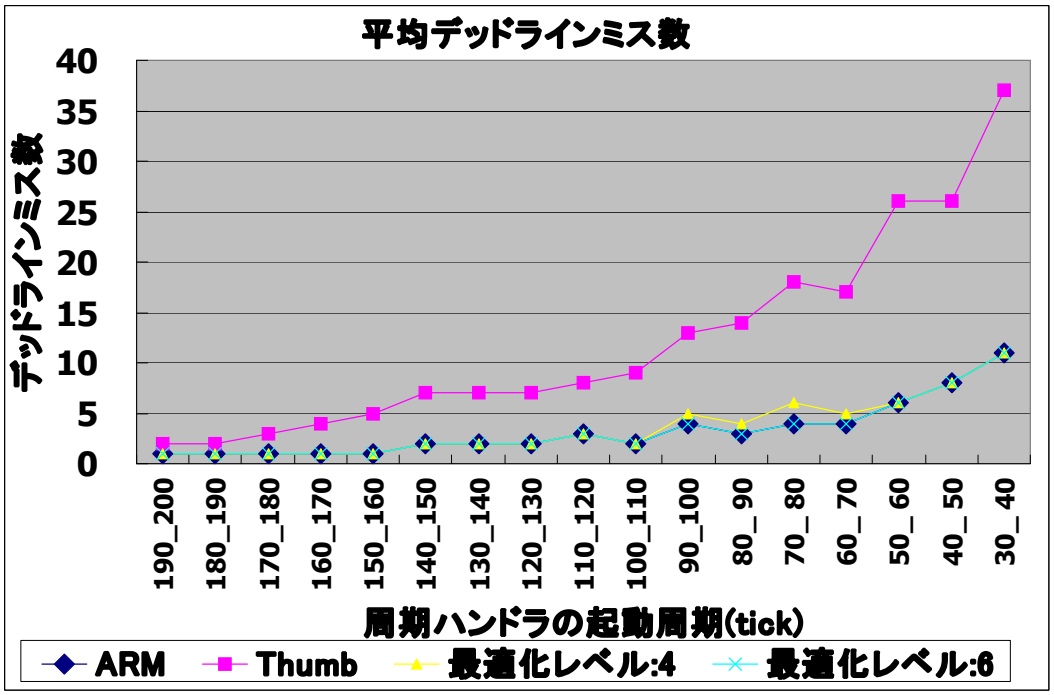

図 4.7: システム全体のデッドラインミス数

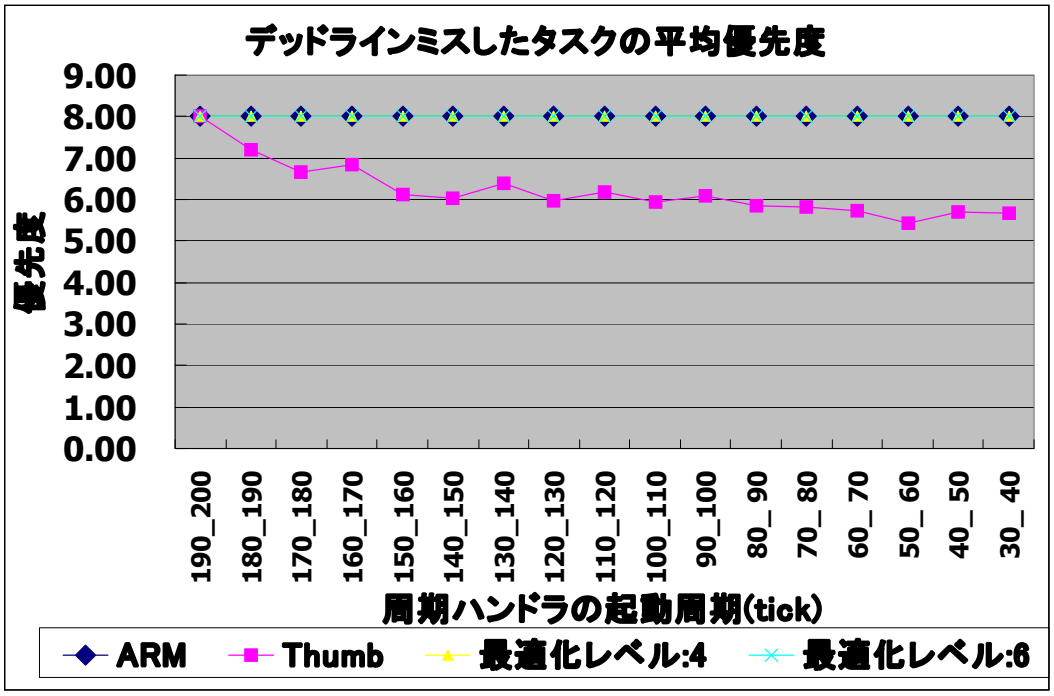

図 4.8: デッドラインミスしたタスクの平均優先度

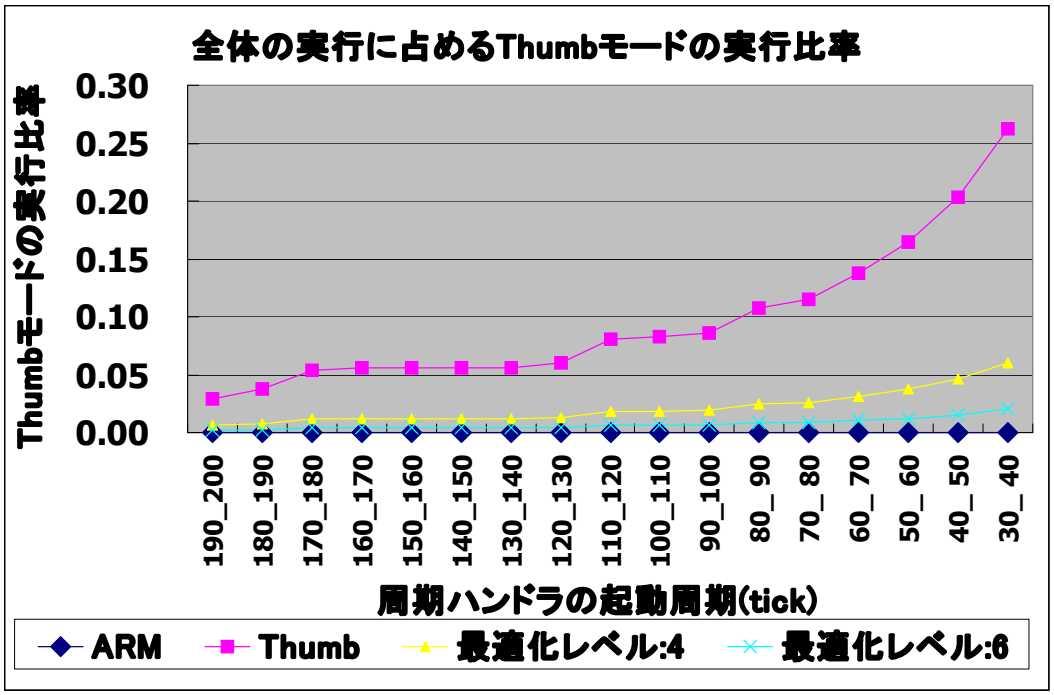

図 4.9: 全体実行に占める Thumb モードの実行割合

4.6　シミュレーションの考察

本節では，各計測結果に対して，全て ARM モードの API(ARM_API)，全て Thumb モードの API(Thumb_API)，提案手法の最適化レベル 4(OPT4_API)， 6(OPT6_API) の場合を比較し，考察を行う．

考察にあたり，組込みシステムでは CPU 使用率が 80%を超える使用状況は現実的ではないため，考慮しないことにする．

全タスクの平均応答時間

図 4.6 を考察する．Thumb_API は，ARM_API と比較して平均応答時間が増加している．平均 11.7%の増加であった．これは，Thumb_API のほうが 1API あたりの実行命令数が多く，タスクの実行時間が長くなったためである．

次に，OPT4_API，OPT6_API と ARM_API との比較では，平均応答時間の増加はみられなかった．これは，Thumb モードと ARM モードで実行時間に差が無い API が存在したためである．

平均デッドラインミス数

図 4.7 を考察する．Thumb_API は，ARM_API と比較してデッドラインミス数が増加している．平均 3.6 倍の増加であった．これは，Thumb_API の実行時間が長くなり，タスクに与えられた相対デッドラインとの余裕時間が小さくなったためである．

次に，OPT4_API，OPT6_API と ARM_API を比較する．OPT4_API では，平均 1.1 倍の増加であった．OPT6_API では，デッドラインミス数の増加はみられなかった．これは，前述の通り，Thumb モードと ARM モードで実行時間に差が無い API が存在したためである．

デッドラインミスしたタスクの平均優先度

図 4.8 を考察する．Thumb_API は，ARM_API，OPT4_API，OPT6_API と比較してデッドラインミスしたタスクの平均優先度が減少している．ARM_API，OPT4_API，OPT6_API は，平均 8.0 であった．Thumb_API は，平均 6.2 であった．これは，Thumb_API の実行時間が長くなり，優先度の高いタスクがデッドラインミスしたためである．

全体の実行に占める Thumb モードの実行割合

図 4.9 を考察する．Thumb_API は，平均 9.7%であった．OPT4_API は，平均 2.2%であった．OPT6_API は，平均 0.7%であった．これらことから，OPT4_API，OPT6_API では，観測時間における ARM モードの処理時間が支配的であることがわかる．

第5章　おわりに

5.1　まとめ

本研究では，ARM プロセッサ上で動作する μITRON4.0 スタンダードプロファイル仕様の RTOS をターゲットとした DBMS をソフトウェア部品として実装した．そして，タスクの静的優先度に従って，リアルタイム性を保ちつつ，実行バイナリコードを削減する手法を提案し，DBMS のコンフィギュレータに実装した．

提案手法を評価するために，ARM シミュレータを実装した．また，μITRON4.0 スタンダードプロファイル仕様のカーネルライブラリを用いて，ARM プロセッサで動作する RTOS を実装した．そして DBMS ライブラリを使用するタスクセットを実装し，それに提案手法を適用した場合，DBMS_API のコードサイズが減少することを確認した，また，提案手法を適用したタスクセットをシミュレーションにより評価し，ARM モードと同等な平均応答時間であることを確認した．これらのことから，提案手法を用いて実行バイナリコードが削減されつつリアルタイム性を保つことを確認した．

5.2　今後の課題

今後の課題として，以下の項目が挙げられる．

- 組込み DBMS は，メモリ制約の厳しいシステムにおいても高速に処理できるアルゴリズムであることが望まれている．本研究で実装した DBMS ライブラリには，DBMS 機能の多機能化やアルゴリズムの高速化の面で改良する余地が残されていると考える．
- 本提案手法は，リンクツールに渡す DBMS 利用情報ファイルの生成をシステム開発者側で行う仕様である．これを自動化するには，タスクのプログラムから DBMS_API を自動で収集するツールを実装することで実現できると考える．

謝辞

本研究を行うにあたり，熱心にかつ懇切丁寧にご指導を賜りました，田中 清史 准教授に心から深く感謝するとともに，ここにお礼を申し上げます.

適切な御助言をいただきました日比野 靖 教授，井口 寧 准教授，菅原 英子 助教に深く感謝いたします.

その他，貴重な御意見，御討論をいただきました田中・日比野研究室の皆様をはじめとする多くの方々の御助言に対して厚く御礼申し上げます.

最後に，日頃から暖かく支援してくださいました両親に感謝いたします.

参考文献

[1] ARM Limited.“ARM Architecture Reference Manual”,Pearson Education, 1996-2000.

[2] (社) トロン協会.“μITRON4.0 仕様 Ver.4.03.00”.

[3] インターフェース.“組み込みシステムにおけるデータ処理の効率化技法”,CQ 出版社 2006.

[4] Sloss, Symes, Wright, “ARM System Developer’s Guide”, MORGAN KAUFMANN PUBLISHERS, 2004.

[5] G. C. Buttazzo, “Hard Real-Time Computing Systems : Predictable Scheduling Algorithums And Applications,” Springer, 2004

[6] Empress Software 社.“http://www.empress-jp.com/”.

[7] Kiyofumi Tanaka,“Real-Time Adaptive Task Scheduling”, Proc. of International Conference on Embedded Systems and Applications (ESA’05), pp. 24 - 30, 2005