一般社団法人　電子情報通信学会
THE INSTITUTE OF ELECTRONICS,
INFORMATION AND COMMUNICATION ENGINEERS

信学技報
IEICE Technical Report
VLD2019-70,CPSY2019-68,RECONF2019-60(2020-01)

FreeRTOS を用いたシステムのフルハードウェア合成

中野和香子† 石浦菜岐佐† 冨山 宏之†† 神原 弘之†††

† 関西学院大学 理工学部 〒 669–1337 兵庫県三田市学園 2–1
†† 立命館大学 理工学部 〒 525–8577 滋賀県草津市野路東 1 丁目 1–1
††† 京都高度技術研究所 〒 600–8813 京都市下京区中堂寺南町 134 番地

あらまし 組込みシステムには益々高い機能が実装される一方で厳しい応答性能が要求される．これを解決する一手法として , 大迫らは RTOS 上で動作するプログラムを入力として, RTOS の機能とカーネルオブジェクト全てをハードウェアに合成する手法を提案している．この手法では RTOS として TOPPERS/ASP3 を想定していたのに対し, 本稿では同じ手法を FreeRTOS に対応させる手法を提案する．FreeRTOS カーネルではタスクは動的に生成され, タスクの制御用データは線形リストで管理されるが, 本手法ではスケジューラ起動前にタスク生成を行なっているプログラムに対象を限定し, タスク制御用データは配列で管理するようにシステムコールを修正する．ソフトウェアタイマー機能はデーモンタスクではなく, 各タスクにタイマーを持たせる形に変換する．本手法に基づく合成システムを実装し, FreeRTOS 付属のデモプログラム main_full.c を縮小したテストプログラムで, タスク機能のテストと , ソフトウェアタイマーの評価部分をハードウェア実装した結果, タスクの起動を 19 サイクル, 割込みハンドラの起動を 1 サイクルで行えることが確認できた .

キーワード リアルタイムシステム, RTOS, システム合成, ハードウェアアクセラレータ, FreeRTOS, 高位合成

Full Hardware Synthesis of FreeRTOS-Based Systems

Wakako NAKANO†, Nagisa ISHIURA†, Hiroyuki TOMIYAMA††, and Hiroyuki KANBARA†††

† Kwansei Gakuin University, Gakuen 2–1, Sanda, Hyogo, 669–1337, Japan
†† Ritsumeikan University, 1–1–1 NojiHigashi Kusatsu, Shiga, 525–8577, Japan
††† ASTEM RI/KYOTO, 134 Chudoji Minamimachi, Shimogyo–ku, Kyoto, 600–8813, Japan

Abstract As higher and higher functionalities are being implemented in embedded systems, it is becoming a difficult task to ensure their real-time performance. As one approach to enhancing response performance of RTOS-based systems, Oosako proposed a method of implementing both kernel objects and RTOS functionalities as hardware utilizing high-level synthesis, where TOPPERS/ASP3 was assumed as an RTOS. This article extends this method to deal with systems based on FreeRTOS. In FreeRTOS, tasks are generated dynamically, whose control data are managed in the form of linear lists. We limit ourselves with systems where tasks are generated before scheduler starts, and keep the task control data in an array. Software timers are dealt with as tasks that have their own timers. We have implemented a prototype synthesis system and synthesized a reduced version of a demo program main_full.c bundled with FreeRTOS. Resulting hardware took 19 cycles and 1 cycles for activation of a task and a handler, respectively.

Key words Real-Time Systems, RTOS, System Synthesis, Hardware Accelerator, FreeRTOS, High-Level Synthesis

1. はじめに

近年の情報通信技術の発展によって, 新しいサービス, 機器が提供されるようになっており, 組込みシステムには益々高い機能が要求されるようになっている．特に, 車載機器, 自律飛行機, ロボットの制御には, 高い機能と同時に高い応答性能が要求される．

このようなリアルタイムシステムには, 制限時間内にタスク処理の実行を完了するリアルタイム性が必要であるが, そのためにはリアルタイム OS (RTOS) を用いた開発が不可欠である．RTOS はタスクの実行時間を制御するための機能を備えているが, システムの複雑化に伴って, このリアルタイム性の実現

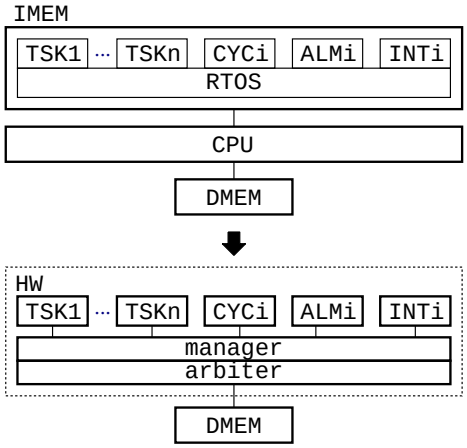

図 1: RTOS を用いたシステムのフルハードウェア実装

は難しくなってきている.

RTOS を用いたシステムの高速化手法として, RTOS 機能のハードウェア実装がある. 文献 [1] [2] [3] では, ハードウェアにより RTOS のスケジューラの高速化を行っている. また, 文献 [4] [5] [6] では RTOS のほとんどの機能をハードウェア実装している. しかし, これらの手法では, タスクやハンドラなどはソフトウェアのままであり, プロセッサのタスク切り替えのオーバーヘッドが発生する. また, タスクの処理の負荷が大きい場合は他タスクの実行待ち遅延により, 応答時間が改善されない場合もある.

一方, 高位合成技術 [7] を用いることにより, タスクを専用ハードウェアに合成し, 高速化する手法がある. 指定したタスクをハードウェアに合成するためのシステムレベル設計手法 [8] [9] が提案されているが, これらの手法では, RTOS および一部のタスクはソフトウェアとして実行される. 文献 [10] [11] は割込みハンドラを含むシステム全体のハードウェア化を行っているが, 合成対象はベアメタルシステムのみである.

この課題を解決する一手法として, 文献 [12] はタスクおよび RTOS のサービスコールの機能全てをハードウェアに合成する手法を提案している. しかし現時点でこの手法が適応可能な RTOS は TOPPERS/ASP3 に限定されている.

本稿では, FreeRTOS と TOPPERS/ASP3 の仕様の違いを明らかにし, 文献 [12] の手法を, FreeRTOS に適応させる手法を提案する. 本手法ではスケジューラ起動前にタスク生成を行なっているプログラムに対象を限定し, タスク制御用データは配列で管理するようにシステムコールを修正する. ソフトウェアタイマー機能はデーモンタスクではなく, 各タスクにタイマーを持たせる形に変換する.

本手法に基づき, FreeRTOS のサービスコールを利用して記述された C 言語のプログラムから, Verilog HDL で記述されたハードウェア動作記述を生成するための合成システムを実装した. 予備実験では, タスクの起動が 19 サイクル, 割込みハンドラの起動が 1 サイクルで行えることを確認した.

2. RTOS を用いたシステムのフルハードウェア実装

2.1 TOPPRES/RSP3 を用いたシステムのフルハードウェア実装

RTOS は複数の逐次処理を並列に動作させるが, この実行単位をタスクと呼ぶ. TOPPERS/RSP3 は時間を管理する機能として, 特定の周期で実行される周期ハンドラと, 指定時間の経過後に実行されるアラームハンドラを持つ. 割込みハンドラは割込み信号もしくはイベントにより起動される.

文献 [12] では, TOPPERS/ASP3 のサービスコールを用いたプログラムを入力とし, これを実行するプロセッサと機能等価なハードウェアの合成を提案している. その概念を図 1 に示す. 上方の TSKi, CYCi, ALMi, INTi はそれぞれのタスク, 周期ハンドラ, アラームハンドラ, 割込みハンドラであり, RTOS の管理下で実行される. この手法では, 各タスク/ハンドラは独立したハードウェアモジュールに合成され, RTOS 機能は manager と arbiter というハードウェアが実現する.

タスク/ハンドラは, 実行可能になれば全て並列に実行する. この制御は manager が各タスク/ハンドラの状態変数の値からタスク/ハンドラの実行/停止を制御する信号を生成することにより行う. データメモリ (DMEM) の同じバンクに同時にアクセスがあった場合には, 優先度に応じて arbiter が調停を行う. スケジューリングやコンテキストスイッチ, および CPU 待ちのオーバーヘッドがなくせる上, タスク/ハンドラをハードウェア化して高速実行できるため, システムの応答性能を飛躍的に向上させることができる.

合成されるハードウェアの構成を図 2 に示す. 各タスク/ハンドラの状態は, manager モジュール内の status レジスタに, カーネルの状態は global status レジスタに格納される. manager モジュールは, status レジスタの値に基づき, そのタスク/ハンドラが実行可能な場合には stall 信号を 0 に, 実行を停止させる場合には stall 信号を 1 にすることにより, タスク/ハンドラの実行を制御する. 周期ハンドラとアラームハンドラは, それぞれが status レジスタ内部にタイマー (カウンタ) を持ち, 実行条件が成立すると manager が stall 信号を 0 にする. 割込みハンドラは, manager への IRQ 信号によって起動される. IRQ 信号が送られた時, 割込みが禁止されていなければ, 割込みハンドラへの stall が解除され, ハンドラが起動する.

RTOS のサービスコールの大部分は status レジスタと global status レジスタを参照/更新する処理として実装できる. status レジスタと global status レジスタは, メモリ空間にマップし, タスク/ハンドラからはメモリアクセスによって参照/更新できる. ロックの取得/解放などの処理は, ハードウェアにより実装する.

2.2 FreeRTOS と TOPPERS/ASP3

TOPPERS/ASP3 と FreeRTOS はいずれもプリエンプティブなスケジューリングを行う RTOS であるが, 仕様には相違点がある.

TOPPERS/ASP3 では, 全てのタスクやハンドラの生成は静的であり, コンパイル時に決定している. これに対し FreeRTOS

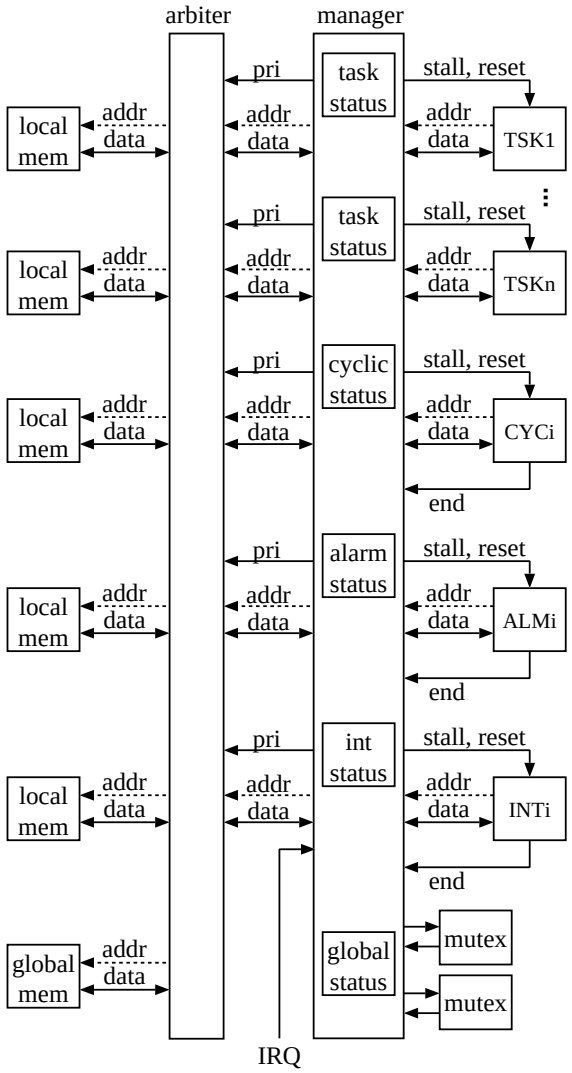

図 2: 合成されるハードウェアの構成

ではカーネルオブジェクトは静的にも動的にも生成可能であり, タスク, ソフトウェアタイマーなどはスケジューラ起動前だけでなく, 起動後にも生成が可能である. このためタスク管理のデータ構造は TOPPERS/ASP3 が配列であるのに対し, FreeRTOS は線形リストである.

時間管理機能は, TOPPERS/ASP3 は周期ハンドラ, アラームハンドラがその機能を担うが, FreeRTOS はソフトウェアタイマーが同様の機能を担う. 周期ハンドラ, アラームハンドラは非タスクコンテキストで実行されるのに対し, ソフトウェアタイマーは, デーモンタスクと呼ばれるタスクコンテキストで実行される. また, ソフトウェアタイマーには将来の設定した時刻に関数を実行するワンショットタイマー, また一定周期での関数の実行を管理する自動再ロードタイマーの 2 つのモードがあり, それぞれアラームハンドラ, 周期ハンドラの機能に対応している. しかし, ソフトウェアタイマーは起動後にモードの切り替えが可能である.

3. FreeRTOS を用いたシステムからの合成

本稿では文献 [12] の手法に基づき, FreeRTOS のサービスコールを用いて書かれた C プログラムを入力として, これを実行するプロセッサと機能等価なハードウェアの動作記述を自動合成する手法を提案する.

表 1: FreeRTOS における状態

(a) タスクにおける状態

状態名	意味
Ready	タスク自身は実行できる状態にあるが, それよりも優先順位の高いタスクが実行状態にあるために, そのタスクが実行されない状態
Running	タスクが実行されている状態. (タスクの実行中に発生した割込みまたは例外処理中かつ, タスクコンテキストに戻った後に, そのタスクの実行を再開する状態を含む)
Blocked	タスクが何らかの条件が揃うのを待つために, 自ら実行を止めている状態
Suspend	強制的に実行を止められている状態
PendingReady	スケジューラ停止時にタスク起動がかかり, タスクがスケジューラ起動を待っている状態.

(b) ソフトウェアタイマーにおける状態

状態名	意味
Dormant	ソフトウェアタイマーは休止中であり, 参照はできるが, そのソフトウェアタイマーは実行されない状態
Running	ソフトウェアタイマーが実行されている状態

表 2: 実行制御と排他制御のためのサービスコール

(a) タスク実行制御

サービスコール	機能
xTaskResume	指定したタスクを起床する
vTaskSuspend	指定したタスクを強制待ちにする.
vTaskDelay	自タスクを起床待ちさせる
vTaskSuspendAll	ディスパッチ禁止しスケジューラを停止する
xTaskResumeAll	ディスパッチ禁止を解除する
vTaskPrioritySet	指定したタスクの優先度を設定する

(b) ソフトウェアタイマー機能

サービスコール	機能
xTimerStart	指定した時間, 自タスクを待ち状態にし, 遅延させる
xTimerStop	自タスクを終了させる
xTimerReset	指定したタスクに終了要求を行う

本手法では一つのカーネルオブジェクトを一つのハードウェアモジュールに合成するため, カーネルオブジェクトの生成は静的なものに限定する. すなわち, 全てのカーネルオブジェクトは main 関数内でスケジューラが起動する前に生成されていることを前提とする. またタスク管理のデータ構造はリストから配列に変更し, サービスコールの中でこのデータ構造にアクセスしている部分は全て書き換える. ソフトウェアタイマーは文献 [12] における周期ハンドラ, アラームハンドラの機能を統合したハードウェアに合成する.

3.1 TCB の管理

FreeRTOS の各タスクは, 表 1 (a) の 5 つのうちいずれかの状態をとる. タスク管理を行うデータ構造はタスク制御ブロック (TCB) と呼ばれる. FreeRTOS は状態ごとに線形リストを持ち, タスク状態に対応したリストにそのタスクの TCB をつなぐことによりタスクを管理する. リストによるタスクの管理を図 3 に示す. Task1, Task2, Task5 が Ready 状態の場合, その TCB は, ReadyList につなぐ. 各 TCB はリストのヘッダへのポインタを持っており, これによって自分の状態を知ることができる. 状態の変更はリストのつなぎ替えにより行う. 図 3 において Task1 の状態を Ready から Suspend に変更する場合には, Task1 の TCB を Ready List から外し, Suspend List

表 3: 状態変数

(a) タスクの状態変数

変数	意味
xState	現在の実行状態
xStateValue	待ち時間
uxPriority	優先度
uxBasePriority	ベース優先度
ulNotifiedValue	タスク通知値
ucNotifyState	タスク通知状態
pcTaskName	タスク名
uxMutexesHeld	ミューテックス取得フラグ
uxCriticalNesting	クリティカルセクションのネスト数

(b) ソフトウェアタイマーの状態変数

変数	意味
ucStatus	タイマーのモード
xTimerState	実行状態
xTimerPeriodInTicks	通知時刻までの相対時間
xTimerBasePeriod	タイマー設定時間
flgReset	タイマリセットフラグ
*pcTimerName	タイマー名
*pvTimerID	タイマー ID

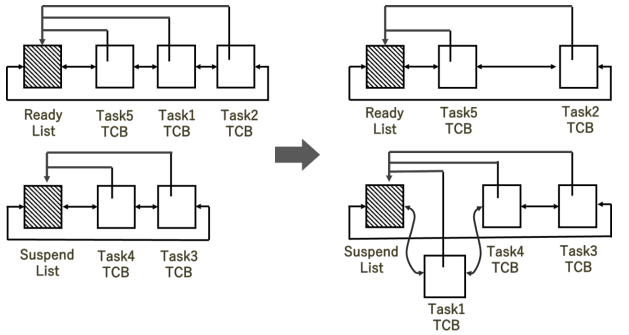

図 3: 線形リストによるタスク管理

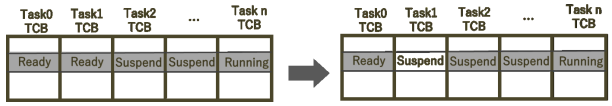

図 4: 配列によるタスク状態の変更

につなぐ.

本稿の手法ではこのデータ構造を変更し, 各タスクの TCB を配列で保持するようにする. また各 TCB のメンバにはリストヘッダへのポインタではなく, タスクの状態を直接持たせるようにする. 図 4 に本稿の TCB 保持法と状態変更の処理例を示す. Task1 の状態を Ready から Suspend に変更する場合には, 単に Task1 の TCB のタスク状態を Ready から Suspend に変更するだけである.

3.2 タスクの実行管理

タスクへの stall 信号は Running 状態の時のみ 0 とし, その他の状態 (Ready, Blocked, Suspend, PendingReady) の時は 1 とする. ただし, Running 状態であってもメモリアクセス要求が待たされている場合には stall 信号を 1 とする. この stall 信号の管理は manager モジュールが行う.

本稿の手法で合成するハードウェアでは, Ready になったタスクは全て実行される. これは, Ready になったタスクの状態を manager が強制的に Running に書き換えることにより実現する. すなわち, あるタスクが Ready 状態になると, manager は次のクロックでタスクの状態を Running 状態に更新し, stall 信号を解除する.

タスクの状態の管理と stall 信号の生成をハードウェアで行うために, 本手法では表 3 (a) に示す TCB の 9 個のメンバのうち manager の制御に必要な 6 個のメンバを manager 内の status レジスタに保持し. それ以外はメモリに置く. status レジスタはメモリ空間にマッピングし, タスクモジュールからはメモリのロード/ストア命令でアクセスできるようにする. また, システム全体の状態は manager 内部の global status レジスタに保持し, タスクモジュールの制御に使用する.

表 2 はタスク, ソフトウェアタイマーの状態制御に関する主要なサービスコールである. 例えば, `vTaskSuspend` は指定した TCB へのポインタに対応するタスクを Suspend 状態に遷移させる. FreeRTOS の殆どのサービスコールは status レジスタの参照/更新処理として実装できる.

TCB の管理を配列に変更したことに伴い, タスクの状態変更を変更する FreeRTOS のサービスコールのコードは全て書き換える. ユーザコードの変更は一切必要ない. 例えば, Suspend 状態にあるタスクを Ready 状態にするサービスコール `vTaskResume` は図 5 のように書き換える. 11 行目のコードにより status レジスタが Ready 状態に変更できる. 3 行目はエラーチェックであり, 7, 14 行目はサービスコールをシリアライズするのロックの取得/開放処理である.

3.3 ソフトウェアタイマーの実装

ソフトウェアタイマーは, FreeRTOS において設定した時刻における関数の実行, または一定周期での関数の実行を管理する機能であり, デーモンタスクと呼ばれる 1 個のタスクのコンテキストでその処理を行う. ソフトウェアタイマーは, 表 1 (b) の 2 つのうち, どちらかの状態をとる.

本稿の手法では, ソフトウェアタイマーが管理する各コールバック関数を高位合成で一つハードウェアモジュールに合成する. 各モジュールはタイマーの状態をタスク同様に manager モジュール内の status レジスタに格納し, manager は status レジスタの値を基に stall 信号でモジュールの実行/停止を制御する. manager のレジスタには, 表 3 (b) の 7 個のメンバからなるタスクの状態変数のうち manager の制御に必要な 5 個のメンバのみを保存する. ソフトウェアタイマーはワンショットタイマー, 自動再ロードタイマーの 2 つのモードの切り替えが可能であるため, manager はモードを保持するレジスタを持つ.

タイマーの具体的な動作は以下の通りである.

1) (task) xTimerStart を呼ぶ.
   xTimerState, flgReset に 1 をセットする.
2) (manager) flgReset が 1 ならば xTimerPeriodInTicks に xTimerBasePrtiod の値をセットする.
   flgReset を 0 に戻す.
3) (mamager) xTimerPeriodInTicks をデクリメントする.
4) (manager) xTimerPeriodInTicks が 0 なら stall を 0 にする.
   コールバック関数の処理を実行する.
5) (timer) 実行が完了すれば, end 信号を 1 にする.
6) (manager) 信号が 1 ならば, reset を 1 にする.
   自動再ロードタイマーの場合は xTimerPeriodInTicks

```
1  void vTaskResume( TaskHandle_t xTaskToResume )
2  {
3    TCB_t * const pxTCB = xTaskToResume;
4    configASSERT( xTaskToResume );
5    if ( pxTCB->xState != eRunning &&  pxTCB != NULL )
6    {
7      _loc_service_call();
8      {
9        if( prvTaskIsTaskSuspended( pxTCB ) != pdFALSE )
10       {
11         pxTCB->xState = eReady;
12       }
13     }
14     _unl_service_call();
15   }
16 }
```

図 5: `vTaskResume`

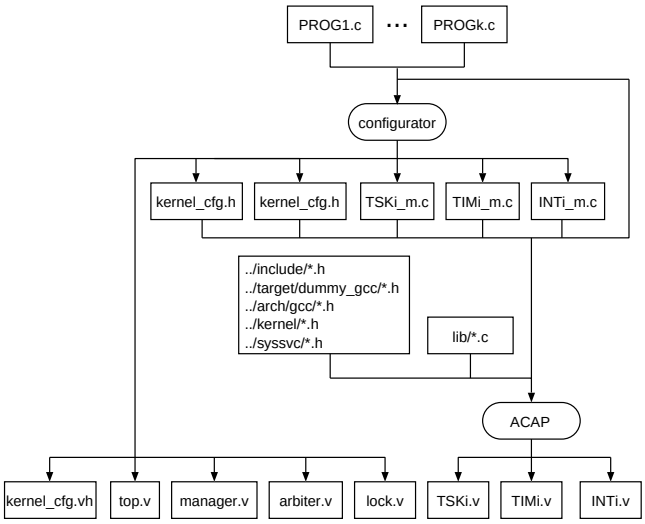

図 6: 合成の流れ

に xTimerBasePrtiod の値をセットする.

ワンショットタイマーなら タイマー状態を Dormant にする.

7) (manager) 3 に戻る

3.4 全体の合成

本手法で提案する合成の流れを図 6 に示す. 図の右上方の PROG1.c, ..., PROGk.c が入力となるシステムを構成するプログラムである. まず, システムの main 関数内で呼ばれるカーネルオブジェクト生成のサービスコールおよびその引数から各種カーネルオブジェクトの構成情報を得る. 例えば, `vTaskSuspend` というタスク生成のサービスコール引数から, タスクの関数名, タスク名, タスクのベース優先度, タスクハンドルの変数名を取得する. これらの情報から, C の定義ファイル kernel_cfg.c, kernel_cfg.h, およびレジスタの構成を記述した kernel_cfg.vh を作成する. また, 各タスク, ソフトウェアタイマーのトップ関数を含む C ファイル (TASK*i*_m.c) を生成する.

これらのソースコードおよびアプリケーションプログラムから, 高位合成システムを用いて必要なハードウェアを合成する. manager や arbiter などのシステム構成によってポート数が変わるモジュールの RTL 記述もシステム構成情報に基づいて生成する.

表 4: 合成結果

(a) 実行サイクル数と遅延

service call	#cycle	latency [ns]
xTaskResume	19	272.6
vTaskSuspend	14	200.9
vTaskDelay	23	330.0
vTaskSuspendAll	13	186.5
xTaskResumeAll	15	215.2
vTaskPrioritySet	24	344.4
xTimerStart	13	186.5
xTimerStop	14	200.9
xTimerReset	14	200.9
interrupt	1	14.3

delay = 14.349 [ns]

実行開始から状態が変化するまでを測定

(b) 回路規模

module	#LUT	#FF
top	129	1
manager	3,566	2,440
arbiter	652	9
lock0	27	18
lock1	27	18
LIM_INC	5,319	367
CNT_INC	6,829	425
C_CTRL	7,060	556
Tmr Tst	9,133	619
AT_TMR1	7,321	426
AT_TMR2	7,009	426
ONE_TMR1	7,619	394
ISRONR_TMR1	2,471	10
ISR OS	3,566	10
total	62,006	5,717

High-level synthesizer: ACAP (2016.10)
Logic synthesizer: Xilinx Vivado (2018.3)
Target: Xilinx Artix-7 (xc7a100tcsg324-3)

4. 実装と実験

4.1 実　　装

本稿の提案手法に基づく合成システムを Perl5 で実装した. 合成システムには ACAP [14] を用いた. ただし現時点では, ヘッダーファイル kernel_cfg.c, kernel_cfg.h, kernel_cfg.vh は自動生成ではなく手で記述している.

ACAP は, C プログラムもしくは MIPS R3000 の機械語プログラムを入力として, これを 32bit の整数演算命令を並列に実行可能なハードウェアに変換する. まず, 入力となるプログラムは GCC を用いてコンパイル/リンクされる. 得られた MIPS の機械語プログラムを CDFG (control dataflow graph) に変換し, 最適化, スケジューリング, バインディングの処理を施して, Verilog HDL を生成する. これにより ACAP は, C プログラムの修正なしに, タスク, 時間管理機能を CPU と機能等価なハードウェアに合成できる. Vivado HLS(注1) などの高位合成システムの場合でも, ソースコードへ少しの修正を加えることにより同様に合成することが可能であると考えられる.

現時点で, FreeRTOS のサービスコール 132 個中 23 個を実装している.

(注1)：`https://www.xilinx.com/products/design-tools/vivado.html`

4.2 実験結果

本システムにより FreeRTOS 付属のデモプログラム “main_full.c” をハードウェア化した．このプログラムは, 機能ごとのテストを行うもので, 今回はタスクとソフトウェアタイマーの機能テストのみに縮小している．

タスク機能のテストは, 共有変数の値をインクリメントするタスク LIN_CNT, CNT_INC と, その 2 個のタスクを制御するタスク C_CTRL の 3 個のタスクからなる．ソフトウェアタイマー機能のテストはタイマーを作成するタスク Tmr Tst, 2 個のオートリロードタイマー FR Timer, ワンショットタイマー Oneshot Timer, 割込みサービスルーチン ISR OS, 割り込みサービスルーチンから起動される ISRONE_TMR1 からなる．

生成した Verilog HDL は, 論理合成ツール Xilinx Vivado 2018.3 によって, Xilinx FPGA Artix-7 (xc7a100tcsg324-3) をターゲットに論理合成した．

合成したハードウェアにおける応答性能を表 4 (a) に示す．#cycle, latency はそれぞれ, 各サービスコールが呼ばれてから状態が更新されるまでの実行サイクル数, 時間を表す．例えば, 1 行目の xTaskResume は, タスクを強制待ち状態から実行可能状態に遷移させるまでに 19 サイクル要したことを表す．(そして次のサイクルで実行状態に遷移する.) このうち, 状態の更新に要するのは 1 サイクルであり, それ以外の 18 サイクルはエラーチェックやシリアライズのためのロック取得/開放処理に費やしている．latency は実行サイクル数とクリティカルパス遅延 14.880 ns の積である．いずれのサービスコールも 357.12 ns 以内に実行することができた．また, 割込みハンドラの起動は 1 サイクルで行うことができた．

表 4 (b) は各ハードウェアモジュールの回路規模を示す．#LUT はルックアップテーブル数, #FF はフリップフロップ数である．manager モジュールの回路規模は文献 [12] に比べ小さくなっているが, これは manager モジュール内のカーネルオブジェクト情報を保持するレジスタを manager モジュールの制御に関するもののみに削減したためである．

MIPS R3000 互換のソフトコアプロセッサの回路規模が約 3200 LUT であるため, 高位合成により生成したタスク, 時間管理機能の回路規模はやや大きいといえる．これについては, ハードウェアの最適化の余地があると考えられる．

5. むすび

本稿では, FreeRTOS のサービスコールを用いたシステムをハードウェアに自動合成する手法を提案した．本手法により生成されたハードウェアは, 応答性能を大きく向上させていると考えられる．

現在, 生成したハードウェアの回路規模がやや大きいため, 高位合成の入力となるソースコードの改善, 生成されるハードウェアの最適化を進めている．また, Vivado HLS などの汎用の高位合成システムを用いることも検討している．

他の課題として, 複数のタスクが同時に実行されないことを前提にした排他制御を行っているプログラムからの合成が挙げられる．解決手法としては, 最も高い優先度を持つタスクのみを動作させるモードを導入することを検討している．

また, イベントフラグ, データキュー, セマフォ等に関わる FreeRTOS の他のサービスコールの実装も今後の課題である．

謝辞

本研究に関して有益な御助言を頂いた元立命館大学の中谷嵩之氏, 元関西学院大学の田村真平氏に感謝致します．本研究は一部 JSPS 科研費 19H04081 および 15H02679 の助成による．

文献

[1] Y. Cho, S. Yoo, K. Choi, N-E Zergainoh, and A. A. Jerraya: “Scheduler implementation in MPSoC design,” in *Proc. ASP-DAC 2005*, pp. 151–156 (Jan. 2005)

[2] M. Vetromille, L. Ost, C. A. M. Marcon, C. Reif, and F. Hessel: “RTOS scheduler implementation in hardware and software for real time applications,” in *Proc. RSP ’06* pp. 163–168 (June 2006).

[3] P. Kohout, B. Ganesh, and B. Jacob: “Hardware support for real-time operating systems,” in *Proc. CODES+ISSS ’03*, pp. 45–51 (Oct. 2003). DOI:`http//doi.org/10.1145/944645.944656`

[4] T. Nakano, Y. Komatsudaira, A. Shiomi, and M. Imai: “Performance evaluation of STRON: A hardware implementation of a real-time OS,” in *IEICE Trans. Fundamentals*, vol. E82-A, no. 11 pp. 2375–2382 (Nov. 1999).

[5] N. Maruyama, T. Ishihara, and H. Yasuura: “An RTOS in hardware for energy efficient software-based TCP/IP processing,” in *Proc. SASP 2010*, pp. 58–63 (June 2010).

[6] C. Stenquist: “HW-RTOS—Improved RTOS performance by implementation in silicon,” White Paper—Renesas R-IN32M3 Industrial Network ASSP (May 2014). Available at available at `https://www.renesas.com/en-us/media/support/partners/r-in-consortium/technology/R-IN32_HWRTOS_Whitepaper_5_20_14.pdf` (accessed 2018-06-11).

[7] D. D. Gajski, N. D. Dutt, A. C-H Wu, and S. Y-L Lin: *High-Level Synthesis: Introduction to Chip and System Design*, Kluwer Academic Publishers (1992).

[8] S. Shibata, S. Honda, H. Tomiyama, and H. Takada: “Advanced system-builder: A tool set for multiprocessor design space exploration,” in *Proc. ISOCC 2010*, pp. 79–82 (Nov. 2010).

[9] Y. Ando, S. Honda, H. Takada, M. Edahiro: “System-level design method for control systems with hardware-implemented interrupt handler,” *IPSJ Journal of Information Processing*, vol. 23, no. 5, pp. 532–541 (Sept. 2015).

[10] N. Ito, N. Ishiura, H. Tomiyama, and H. Kanbara: “High-level synthesis from programs with external interrupt handling,” in *Proc. SASIMI 2015*, 10–15 (March 2015).

[11] N. Ito, Y. Oosako, N. Ishiura, and H. Tomiyama, and H. Kanbara: “Binary synthesis implementing external interrupt handler as independent module,” in *Proc. RSP 2017*, pp. 92–98 (Oct. 2017).

[12] Y. Oosako, N. Ishiura, H. Tomiyama, and H. Kanbara: “Synthesis of Full Hardware Implementation of RTOS-Based Systems,” in *Proc. RSP 2018*, pp. 1–7 (Oct. 2018).

[13] ITRON Committee, TRON association: *μITRON4.0 Specification* (1999, 2002). Available at `http://www.ertl.jp/ITRON/SPEC/FILE/mitron-400e.pdf` (accessed 2018-06-11).

[14] N. Ishiura, H. Kanbara, and H. Tomiyama: “ACAP: Binary Synthesizer Based on MIPS Object Codes,” in *Proc. ITC-CSCC 2014*, pp. 725–728 (July 2014).