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
VLD2021-52,CPSY2021-21,RECONF2021-60(2022-01)

RTOS 利用システムのフルハードウェア化における通信機能の実装

篠原由季乃† 石浦菜岐佐†

† 関西学院大学 〒669–1337 兵庫県三田市学園 2–1

あらまし リアルタイムシステムの応答性能を向上させる手法として, 大迫・六車らはタスク/ハンドラ等のカーネルオブジェクトと RTOS カーネルの機能全てをハードウェアで実装する手法を提案している. 本稿では, 六車が提案したハードウェア構成において RTOS の通信機能であるデータキューとメッセージバッファをハードウェア実装する. データキューは 1 モジュールで複数のデータキューを管理し, データの送信/受信だけでなく, データキューが空/満杯時のタスク待ち/待ち解除の処理も, タスクの実行管理を行うハードウェアと連携して高速に実行する. メッセージバッファに対する可変長データの授受は, タスクとサービスモジュールの間でバイトストリームを授受するレジスタを設けることにより可能にする. 本手法に基づくデータキューとメッセージバッファをハードウェア実装した結果, データキューに対するデータの授受は 3 サイクル, メッセージバッファに対する n バイトデータの授受は $n+8$ サイクル以内に実行できた.

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, TOPPERS/ASP3, FreeRTOS, 高位合成

Design of Inter-Task Communication Modules for Full Hardware Implementation of RTOS-Based Systems

Yukino SHINOHARA† and Nagisa ISHIURA†

† Kwansei Gakuin University, Gakuen 2–1, Sanda, Hyogo, 669–1337, Japan

Abstract This paper presents hardware implementation of inter-task communication functions of RTOS, in the scheme where all the tasks/handlers along with all the RTOS services are implemented as hardware. Hardware modules for the data queue and the message buffer for Muguruma's architecture are designed. The proposed data queue design maintains multiple data queues in a single module and processes send/receive operations including handling of task waiting and timeouts efficiently in cooperation with the hardware module to manage task execution. The message buffer module also manages multiple message buffers and arranges the transfer of variable length message data via a dedicated register between the message buffer module and each task module. The designed data queue takes only 3 cycles for a send/receive operation, and the message buffer processes send/receive of a message of n bytes within $n+8$ cycles.

Key words Real-Time Systems, RTOS, System Synthesis, Hardware Accelerator, TOPPERS/ASP3, FreeRTOS, High-Level Synthesis

1. はじめに

近年の情報通信技術の発展により, 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, 無人飛行機, ロボットの制御には, 高い機能と同時に高い応答性能が必要となる. このようなリアルタイムシステムの開発には, タスクやハンドラが制約時間内に実行を完了するように実装するための機能を提供するリアルタイム OS (RTOS) が用いられる. しかし, システムの高機能化が進むにつれ, RTOS を利用したシステムの応答性能の確保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は, RTOS 機能の一部または全てをハードウェア実装する方法がある. 文献 [1] [2] [3] では RTOS のスケジューラを, 文献 [4] [5] [6] では RTOS の機能の大部分をハードウェア化することにより高速化を図っている. しかし, タスク/ハンドラはソフトウェアのままであり, CPU 待ちやコンテキストスイッチによる遅延は解決されない. 一方, 指定したタスクをハードウェアに合成し高速化するシステムレベル設計手法 [7] [8] が提案されている. しかし,

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のままである．

この課題を解決する一手法として，文献 [9] はタスクおよび RTOS のサービス機能全てをハードウェア化する手法を提案している．この手法は，高位合成を用いてタスクをハードウェア化するとともに RTOS の機能もハードウェアで提供する．しかし，TOPPERS/ASP3 および FreeRTOS を対象としたプロトタイプ実装 [10] [11] [12] では，回路規模が大きくなることが課題となっていた．文献 [13] では，文献 [9] のハードウェア構成において，RTOS のサービス処理機能を集約することによって回路規模を削減する手法を提案している．

文献 [14] [15] では，文献 [13] のハードウェア構成で RTOS の同期機能であるミューテックス，イベントフラグのハードウェアモジュールを実装している．しかし，RTOS のタスク間における通信機能は実装されていない．

本稿は，文献 [13] のハードウェア構成において RTOS の通信機能であるデータキューとメッセージバッファをハードウェア実装する．データキューは 1 つのモジュールで複数のデータキューを管理し，待ち/待ち解除処理をタスクの実行管理を行うハードウェアと連携して実行する．また，メッセージバッファに対する可変長データの授受は，タスクとサービスモジュールの間でバイトストリームを授受するレジスタを設けることにより可能にする．

本手法に基づいてデータキュー/メッセージバッファモジュールを実装した結果，データの送信/受信はデータキューで 3 サイクル，メッセージバッファでは n バイトのデータを $n+8$ サイクル以内に実行することができた．

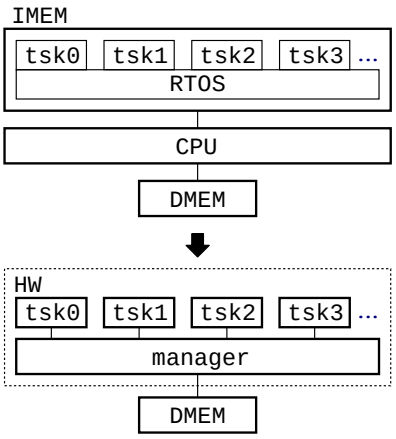

図 1: RTOS 利用システムのフルハードウェア実装

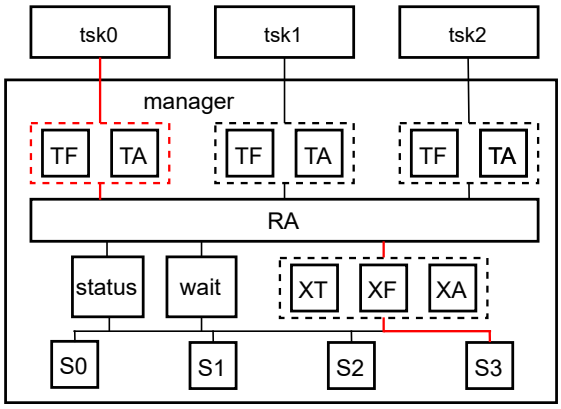

図 2: 文献 [13] のハードウェア構成

2. RTOS 利用システムのフルハードウェア実装

2.1 RTOS 利用システムのフルハードウェア実装

文献 [9] では，RTOS 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を合成する手法を提案している．その概念を図 1 に示す．上側の図で，tsk$i$ はタスク (およびハンドラ等のカーネルオブジェクト) であり，RTOS の管理下で実行される．この手法では，下側の図のように各タスクは高位合成によって独立したハードウェアモジュールに合成され，RTOS 機能はマネージャ (manager) というハードウェアが提供する．

タスクを実行するハードウェアは，実行可能になれば全て並列に実行される．この制御はマネージャが各タスクの状態変数の値からタスクの実行/停止を制御する信号を生成することにより行う．この手法では，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．

文献 [13] では文献 [9] の手法において，タスクに分散している RTOS のサービス処理機能をマネージャ側に集約することによる回路規模の削減と，基本的なサービス処理を RTL で実装することによる実行の高速化を実現している．

2.2 RTOS サービス処理ハードウェアのアーキテクチャ

文献 [13] の手法で実装されるハードウェアの構成を図 2 に示す．

tsk0, tsk1, tsk2 はタスクを実行するハードウェアであり，S0〜S3 は RTOS が提供するサービス処理を実行するハードウェアである．本稿で実装するデータキューとメッセージバッファもこのサービスモジュールの一つである．サービス間の競合を避けるため，サービスは一度に一つしか処理しない．RA (Request Arbiter) は，複数のタスクからのサービス要求を調停する回路である．

タスク tsk$i$ がサービスを要求する場合には，レジスタ TFi に所望のサービスの番号を，TAi に引数を書き込む．RA が実行を許可する要求を選択し，XT にタスクの番号，XF に TFi，XA に TAi を書き込むと，そのサービスを担当するサービスモジュールが処理を開始する．サービスモジュールは処理が完了すると XA に返り値を書き込み，マネージャは TAi に XA を転記してタスクに返す．

タスクとマネージャのやり取りは，図 3 のように，サービスコールをレジスタ TFi, TAi の読み書きとして実装したインタフェース関数 (タスクと共に高位合成する) により行う [16]．

RA は TFi, TAi, status レジスタ，wait レジスタの値からサービスを実行するタスクを決定する．RA は実行可能な状態のタスクからの要求のみを受け付け，そのようなタスクが複数ある場合には優先度が最も高いタスクを 1 つ選択する．status レジスタは，各タスクの状態，優先度，タイマー等を格納している．wait レジスタは，どのタスクがどのサービス待ちをしているかという情報を保持するフラグ配列であり，wait[S][T] はタスク T がサービス S を待って待ち状態になっていることを表す．サービスモジュール S が，wait[S][T] をセットすると，RA は T から S への要求を保留する．また S は wait[S][*] 要素を

```
#define chg_pri(tskid, tskpri) \
        _chg_pri(tskid, tskpri, _F, _A0, _A1)

ER _chg_pri(ID tskid,
            PRI tskpri,
            volatile int* const _F,
            volatile int* const _A0,
            volatile int* const _A1){
  *_A0 = tskid;
  *_A1 = tskpri;
  ap_wait(); // 次サイクルに以下を実行
  *_F = SERV_CTRL_TSK | METHOD_CHG_PRI;
  ap_wait(); // 次サイクルに以下を実行
  return *_A0;
}
```

図 3: サービスのインタフェース関数の記述例 [16]

一斉にクリアして複数タスクの待ち解除を行うことができる.

文献 [14] [15] では, このハードウェア構成においてそれぞれミューテックス, イベントフラグのサービスを処理するハードウェアを実装している.

3. データキュー

3.1 データキューの仕様

データキューは, タスク間で固定長のデータ受授を可能にする機構である. 基本的な操作には send, receive があり, send によってデータキューの末尾にデータを追加し, receive によってデータキューの先頭からデータを取り出すことができる. 一つのデータキューに対して send/receive の操作はいずれも複数のタスクから行うことができる. send/receive のサービスコールは, TOPPERS においては snd_dtq/rcv_dtq, FreeRTOS においては xQueueSend/xQueueReceive として実装されている.

データキューに格納できる最大のデータ数はあらかじめ定められている. タスクが満杯のデータキューに対して send 操作を行った場合には, データキューに空きができるまでそのタスクは待ち状態となる. このため, 満杯のキューに対してあるタスクが receive 操作を行った際には, send を待っているタスクの待ちを解除する必要がある. 複数のタスクが send の実行を待っている場合には優先度が最も高いタスクの待ちが解除される. send 操作には待ち時間の上限が設定でき, 指定された時間内に send を実行できなければ, タイムアウトとなりエラーが返る.

空のデータキューに対する receive や send の処理も同様である.

3.2 データキューモジュール

本稿ではデータキューに対する send/receive およびタイムアウトの機能を全てハードウェアで実装する. データキューの主要な機能は図 2 のサービスモジュール (S0〜S3) の 1 つとして実装する. 1 つのデータキューモジュールが複数のデータキューのデータを保持し, それに対する処理を行う.

データキューモジュールはマネージャ内の中間レジスタ XF にデータキューのサービス番号が書き込まれると処理を開始する. 複数あるキューのうちどのキューを対象にするか (データキューの番号 qid) は XF 中に符号化されている. send の場合には, XA[0] にタイマー値, XA[1] にデータキューに格納するデータが与えられ, 処理完了時に XA[0] に返り値を返す. receive の場合には, XA[0] にタイマー値が与えられ, 処理完了時に XA[0] に返り値, XA[1] にデータキューから取り出したデータを返す.

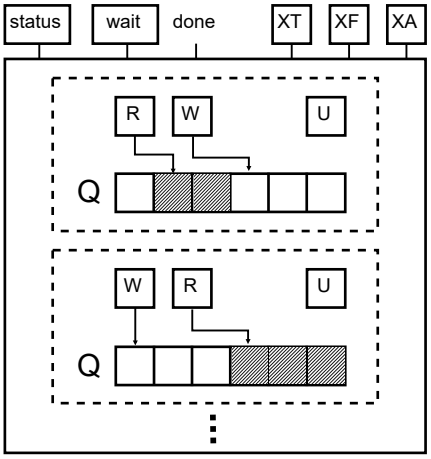

図 4: データキューのハードウェア構成

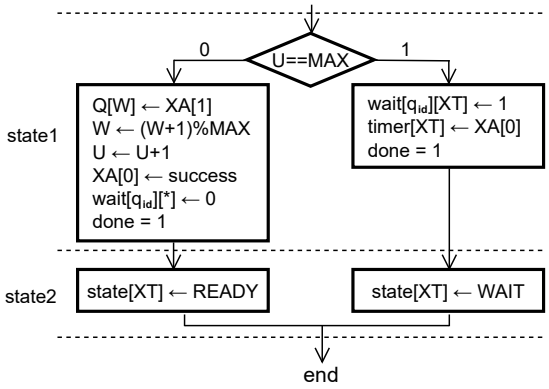

図 5: データキューにおける send 実行の流れ

本稿のデータキューモジュールの構成を図 4 に示す. モジュール内には各データキューに対し, データを格納するキュー (Q), リードポインタおよびライトポインタ (それぞれ R と W), キューに格納されているデータの個数を記憶するレジスタ (U) がある.

データキューモジュールの send 実行の流れを図 5 に示す.

send が実行されたデータキュー (番号 qid) が満杯 (U==MAX) であれば send を要求したタスクを待ち状態にする. これは, 1) wait レジスタの [qid][XT] 要素をセットしてタスク XT が qid のデータキューを待っていることを記録し, 2) タスクのタイマー timer[XT] をセットし, 3) タスクの状態 state[XT] を待ち状態にすることにより行える. timer と state はともに status レジスタ内にあるため同時に書き込むことができず, その処理には 2 サイクルを要する. ただし, state2 の動作はマネージャの処理と並列して実行できるので, state1 で完了信号 (done) を出力する.

データキューが満杯でなければ, データを Q に追加する. Q[W] に XA[1] を格納すると同時に XA[0] にリターンコードを渡し, マネージャに完了を通知する.

待ち解除の処理は receive 待ちタスクの有無に関わらず行う. これは wait レジスタの当該キュー (qid) に関する全タスクのビット (wait[qid][*]) をクリアすることにより行える. これによって receive 待ちのタスク全ての待ちが一旦解除される. 複

数のタスクが receive 待ちをしていた場合に待ちを解除するタスクの選択は RA が行う．すなわち，その中で優先度が最も高いタスクの receive 要求が RA により選択され，データキューに送られて処理される．

タイムアウト処理は，主としてマネージャが行う．マネージャは，各タスク T のタイマー (timer[T]) がセットされていれば毎クロックカウントダウンし，その値が 0 になれば T のサービス待ちがタイムアウトしたと判断する．タイムアウトしたサービス要求はキャンセルして返り値をタスクに返す必要がある．これは，マネージャが当該タスクのサービス要求の XF レジスタ中のキャンセルビットをセットすることにより行う．キャンセルビットがセットされているサービス要求は，RA が他のタスクよりも高い優先度でサービスモジュールに送る．サービスモジュールはキャンセルされたサービス要求を受け取ると，wait フラグのクリア等のキャンセル処理を行い，タイムアウトした旨の返り値を XA レジスタに返す．マネージャはそれを TA レジスタを介してタスクに返し，タイムアウト処理が完了する．

以上のように，タイムアウト処理や他タスクの待ち解除はマネージャが行うので，send の処理自体はデータキューモジュールでは 2 サイクル，TF レジスタに要求が書き込まれてから返り値が XA に書き込まれるまでは 3 サイクルで処理できる．

receive の処理は send と同様である．

4. メッセージバッファ

4.1 メッセージバッファの仕様

メッセージバッファは，データキューと同様に FIFO でデータの授受を可能にする機構だが，バッファ内に格納するデータが可変長である点が異なる．基本的な操作はデータキューと同じ send, receive であるが，いずれにおいてもデータのサイズを指定する．send ではデータ領域へのポインタとともに送信するデータのサイズを指定し，receive ではデータ領域へのポインタとともに受け取れるデータサイズの上限を指定する．FreeRTOS の仕様では，メッセージバッファ内にあるバッファに対してデータを格納するタスクは 1 つ，受け取るタスクは 1 つしかないことを前提にしているが，TOPPERS ではその制限はなく，本稿でも 1 つのメッセージバッファに対して複数のタスクが send/receive を実行できるものとする．send/receive のサービスコールは，TOPPERS においては snd_mbf/rcv_mbf, FreeRTOS において xMessageBufferSend/xMessageBufferReceive として実装されている．

メッセージバッファに格納できる最大のデータサイズはあらかじめ定められており，データを格納するだけの空きがないメッセージバッファに対して send を行うとそのタスクは待ち状態となる．receive 操作を行った際に send を待っているタスクがあれば待ち解除処理が行われる．指定された時間内に実行できない send はタイムアウトとなりエラーが返る．

空のバッファに対する receive や send の処理も同様である．

4.2 メッセージバッファモジュール

本稿ではメッセージバッファに対する send/receive およびタイムアウトの機能をマネージャとサービスモジュールの組み合わせでハードウェア実装する．マネージャとメッセージバッファモジュールのハードウェア構成を図 6 に示す．タスクとメッセージバッファ間でバイトストリームを受け渡すためのレジスタ m を追加している．

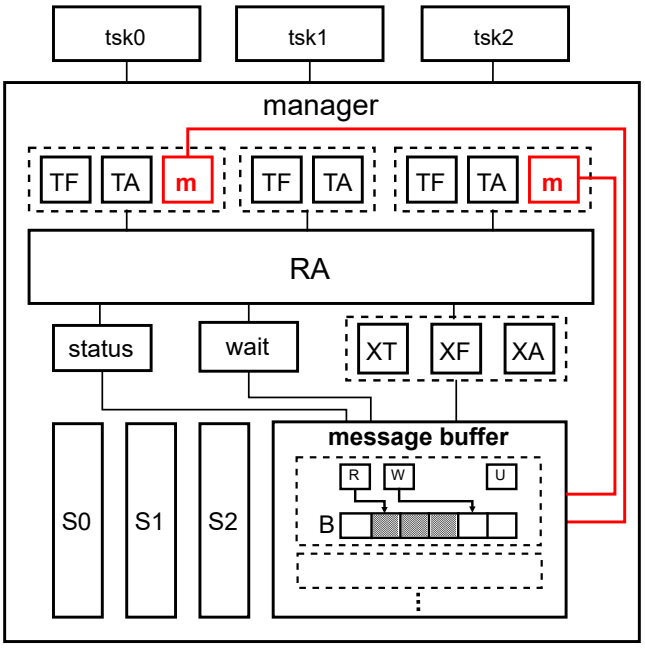

図 6: メッセージバッファのハードウェア構成

send の場合，XA[0] にタイマー値，XA[1] にバッファに格納するデータのバイト数が書き込まれると，m を介して受け取ったデータをバッファに格納し，処理完了時にバッファに格納したバイト数を XA[0] に返す．receive の場合，XA[0] にタイマー値，XA[1] にバッファから取り出せるデータのバイト数の上限が書き込まれると，データを m へ送出し，処理完了時に送出したデータのバイト数を XA[0] に返す．タスクのデータ領域と m レジスタ間のデータ転送はサービスコールのインタフェース関数で行う．

図 6 の右下にあるメッセージバッファモジュールはデータキューと同様の構成である．バッファ B はバイトデータの配列である．n バイトのメッセージは $n+4$ バイトで B に格納され，最初の 4 バイトがメッセージのバイト数を表す．新しいメッセージは W の指す位置に追加し，メッセージの取り出しは R の指す位置から行う．

send の実行の流れを図 7 に示す．B 内にメッセージとそのサイズを格納する空きがない (XA[1]+4>MAX−U) ならば，データキュー同様の処理で send を要求したタスクを待ち状態にする．バッファに空きがあれば，データ格納の処理を進める．state1 でデータのサイズを 4 サイクルで B に格納し，state2 では m を介して受け取ったメッセージをそのバイト数のサイクル数で B に格納する．データを格納し終えたら，XA[0] に受け取ったデータサイズのバイト数を渡し，マネージャに完了を通知する．その後，データキュー同様に待ち解除処理を行う．

メッセージバッファモジュールの receive の実行の流れを図 8 に示す．receive 実行時に，バッファが空 (U==0) であれば，データキュー同様の処理で待ち状態にする．バッファにデータがあればデータ受信の処理を進める．state1 で B[R:R+3] にはデータのサイズが格納されており(注1)，タスクが指定したデー

(注1)：B は循環バッファになっているので R+3 が MAX 以上の場合にはラップラウンドする．

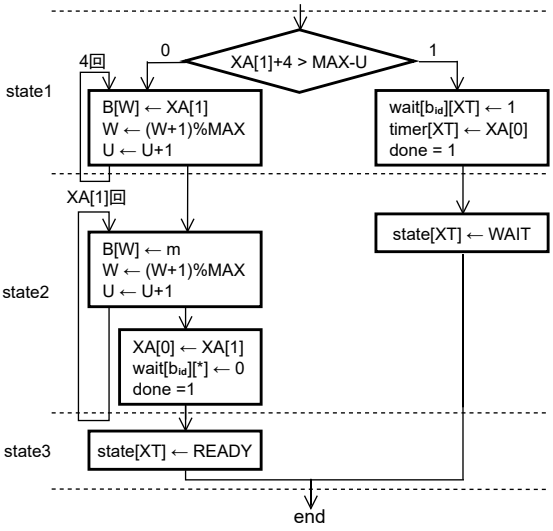

図 7: メッセージバッファにおける send 実行の流れ

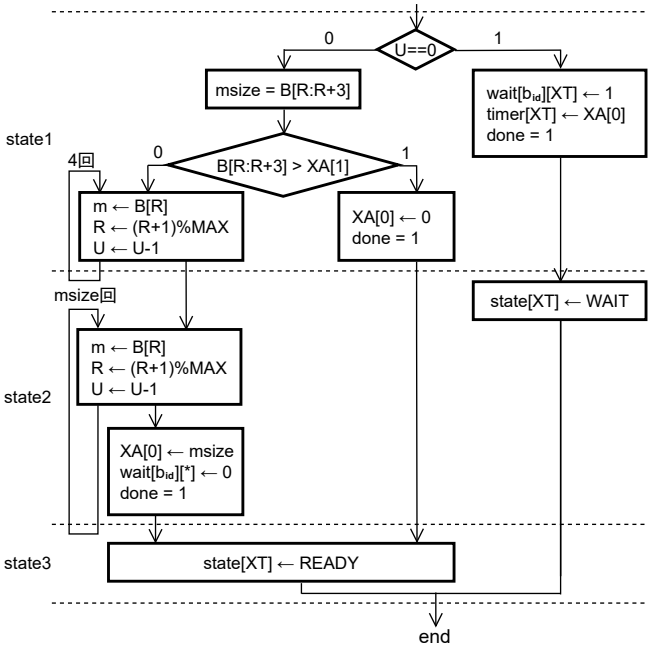

図 8: メッセージバッファにおける receive 実行の流れ

タ上限のバイト数より大きい (msize>XA[1]) ならば, データ送出は行わず, XA[0] に受信バイト数 0 を返し, マネージャに完了を通知する.

データのサイズが XA[1] 以下であれば, データのサイズを 4 サイクルで m に出力する. タスク側のサービスコールのインタフェース関数は, この 4 バイトを見てメッセージのサイズを知り, その回数だけ m を読んで得たデータをタスクの受信領域にコピーする. state2 では B からデータを m にデータのバイト数のサイクルで出力し, 送信し終えたら送信したデータのバイト数を XA[0] に渡してマネージャに完了を通知する. またデータキュー同様に, receive 待ちタスクの有無に関わらず待ち解除の処理を行う.

タイムアウトの処理もデータキューと同様である.

表 1: 実行サイクル数

DataQueue	#cycle
send (not full)	3
receive (not empty)	3
待ち状態になるまでのサイクル数	2
待ち解除からサービスを始めるサイクル数	5
タイムアウト処理	4

MessageBuffer	#cycle
send (not full)	$n+5$
receive (not empty)	$n+8$
待ち状態になるまでのサイクル数	2
待ち解除からサービスを始めるサイクル数	5
タイムアウト処理	4

XA から TAi に返り値を渡すまでのサイクル数

表 2: 論理合成結果

module	#LUT	#FF	delay [ns]
DataQueue	192	44	4.797
MessageBuffer	751	227	5.928

Logic synthesizer: Xilinx Vivado (2020.2)
Target: Xilinx Artix-7 (xc7a100tcsg324-3)

5. 実装と実験

提案手法に基づいて, データキュー/メッセージバッファモジュールを Verilog HDL で設計し, 動作確認を行った.

データキュー/メッセージバッファモジュールにおけるサービスコールの実行に要するサイクル数を表 1 に示す. このサイクル数は, マネージャにあるレジスタ XF, XA, XT に書き込まれてから, TAi に返り値が格納されるまでの各サービスのサイクル数である. データキューは, send および receive のサービスをいずれも 3 サイクルで実行できた. メッセージバッファは, n バイトのデータの send と receive をそれぞれ $n+5$, $n+8$ サイクルで実行できた.

データキュー/メッセージバッファの待ち状態にかかるサイクル数は, マネージャにあるレジスタ XF, XA, XT に書き込まれてから status レジスタのタイマー timer[XT] をセット, タスクの状態 state[XT] を wait に書き換えるまでを 2 サイクルで実行可能である. 待ち解除を行なった際, サービスモジュールとマネージャが処理を完了した後, 待ちタスクの中で優先度が最も高いタスクがサービス処理を開始するまでには, 5 サイクルを要した.

タイムアウト処理は, いずれにおいても, データキュー/メッセージバッファがタイマー値が 0 になってから TAi に返り値を返すまでに 4 サイクルを要した.

設計したモジュールを論理合成した結果を表 2 示す. 論理合成は, Xilinx Vivado (2020.2) により Artix-7 をターゲットに行なった. データキュー/メッセージバッファモジュール内には, それぞれデータキュー/メッセージバッファを 1 つだけ実装している. データキューの LUT 数は 192, メッセージバッファの LUT 数は 751 となり, モジュール単体では小規模な回路になった. クリティカルパス遅延は, データキューが 4.797ns, メッセージバッファが 5.928ns であり, 十分小さいと考えられる.

6. むすび

本稿では, 文献 [13] のハードウェア構成で RTOS のタスク間通信機能であるデータキューとメッセージバッファを実装した.

実装したデータキュー/メッセージバッファモジュールは比較的小規模であり, データ授受にかかるサイクル数はデータキューが 3 サイクル, メッセージバッファが n バイトデータに対して $n+8$ サイクル以内と非常に高速に実行できた.

本稿のハードウェア構成では, 待ち解除は全て優先度順に行っているが, RTOS によっては待ち解除を到着順としているものもあるので, それに対応した設計が必要である. また, 入力プログラム中のタスク数やその中で使われているサービスに応じてマネージャとサービスモジュールを自動生成するシステムの開発も今後の課題である.

謝　辞

本研究にあたり, ご助言を頂きました京都高度技術研究所の神原弘之氏, 立命館大学の冨山宏之教授, 元立命館大学の中谷嵩之氏に感謝いたします. また, 安堂拓也氏, 中原正樹氏, 南口比呂氏, 石井雄吾氏はじめ, 本研究にご協力, ご討議頂いた関西学院大学石浦研究室の諸氏に感謝致します. 本研究は一部 JSPS 科研費 19H04081 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).

[7] 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).

[8] Yuki Ando, Shinya Honda, Hiroaki Takada, Masato 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).

[9] 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).

[10] 大迫 裕樹, 石浦 菜岐佐, 神原 弘之, 冨山 宏之: "RTOS を用いたシステムのフルハードウェア実装とその自動化," 信学技報, VLD2018-122 (Mar. 2019).

[11] 中野 和香子, 石浦 菜岐佐, 神原 弘之, 冨山 宏之: "FreeRTOS を用いたシステムのフルハードウェア合成," 信学技報, VLD2019-70 (Jan. 2020).

[12] W. Nakano, Y. Shinohara, and N. Ishiura: "Full hardware implementation of FreeRTOS-based real-time systems," *in Proc. TENCON*, (Dec. 2021).

[13] 六車 伊織, 石浦 菜岐佐, 安堂 拓也, 神原 弘之, 冨山 宏之: "RTOS 利用システムのフルハードウェア化におけるサービス処理機能の集約," 信学技報 VLD2020-75 (Mar. 2021).

[14] 南口比呂, 石浦菜岐佐: "RTOS 利用システムのフルハードウェア化におけるミューテックスの実装," 信学ソ大 A-6-1 (Sept. 2021).

[15] 中原正樹, 石浦菜岐佐: "RTOS 利用システムのフルハードウェア化におけるイベントフラグの実装," 信学ソ大 A-6-2 (Sept. 2021).

[16] 安堂 拓也, 石井 雄吾, 石浦 菜岐佐, 冨山 宏之, 神原 弘之: "RTOS 利用システムの汎用高位合成系を用いたフルハードウェア化," 信学技報, VLD-2021 (Jan. 2022).