組込み CPU 向け S/W 機能分離アーキテクチャの提案

出原 章雄† 山本 整†
東山 知彦† 落合 真一†

近年の組込み H/W の高性能化に伴い，従来は別 H/W プラットフォームで実現していた複数システムをマルチコアの単一 H/W プラットフォームで実現可能になりつつある．本構成の問題点として，一方のシステムの S/W 障害が，他システムの S/W に影響し，動作が停止する可能性がある．この課題に対し，安全 S/W と，非安全 S/W を分離し，非安全 S/W の障害により問題が起こった場合でも，安全 S/W に影響することなく，復旧可能な基盤 S/W を提案する．

本基盤 S/W では，空間的パーティショニングにより，H/W の共有リソースを安全 S/W と非安全 S/W に分離し，非安全側の S/W 障害が安全 S/W に影響しない S/W 構成を実現する．さらに安全 S/W に影響することなく，非安全 S/W のみを再起動可能とする．

本基盤 S/W を実装した結果，コードの変更量は 100 行程度となり，小規模な変更となった．また，S/W の再起動時間は通常起動時と同様の起動時間となり，今回提案した手法が有効であることを確認した．

A proposal of S/W architecture based on functional separation for embedded processors

Akio Idehara†, Hitoshi Yamamoto†,
Tomohiko Higashiyama† and Shinichi Ochiai†

Since recent embedded H/W performance is increasing, one H/W platform with multi-core processors is able to integrate multiple systems. However, the H/W platform has the issue that a S/W fault of one system causes failure of another system. To solve this issue, we propose a S/W architecture which separates safe S/W and unsafe S/W, and it reboots only unsafe S/W whenever any unsafe S/W fault occur.

This S/W architecture doesn't make any influence to safe S/W when an unsafe S/W fault occurs, by partitioning safe S/W and unsafe S/W.

As the results of the S/W implementation, it was cleared that an amount of changed code is quite small (about 100 lines of code) and a reboot time is almost same as initial boot time. These show that the proposed S/W architecture is useful for this purpose.

1. はじめに

近年の組込みハードウェア(以下，H/W)の高性能化に伴い，従来は複数 H/W プラットフォーム(以下，PF)で実現していたシステムが，マルチコアの単一 H/W PF で実現可能になりつつある．例えば，リアルタイム性の必要な制御処理を行うソフトウェア(以下，S/W)と，情報系処理を行う S/W といった，複数 S/W をマルチコアの単一 H/W PF で実現可能となっている．

† 三菱電機株式会社 情報技術総合研究所
Information Technology R&D Center, Mitsubishi Electric Corporation

こうしたシステムでは，H/W PF を共有しているため，非安全 S/W の障害の影響により，安全 S/W の処理が停止する可能性がある．ここで，安全 S/W とはシステムの実現に際して，満たすべき安全レベル(安全目標を満たすことで実現できる安全性の度合い)が要求されている S/W と定義する．また，非安全 S/W は安全レベルが要求されていない S/W と定義する．

複数 H/W PF で実現している場合，障害が発生した非安全 S/W が動作している H/W PF のみを再起動させることにより，安全 S/W に影響を与えることなく継続処理が可能である．しかし単一 H/W PF の場合，H/W によ

る再起動は，正常動作中の安全 S/W にまで影響を与えてしまう．

そこで今回，非安全S/Wに障害が発生した際に，安全 S/W に影響を与えることなく，非安全 S/W が再起動可能となる基盤 S/W を検討し，実際に開発を行った．本稿ではこの内容を述べる．次節で関連技術，3 節で課題，4 節で設計を実装，5 節で評価と考察について述べる．6 節で今後の展望を述べ，最後に本論文をまとめる．

2. 関連技術

関連技術の内，システムの信頼性と安全性について述べる．信頼性は，JIS X 0014:1999 にて「機能単位が，要求された機能を与えられた条件のもとで与えられた期間実行する能力」と定義されている．また，安全性は JIS X 0134:1999 にて「システムが規定された条件の下で，人の生命，健康，財産またはその環境を危険にさらす状態に移行しない期待度合い」と定義されている．以上のことから，文献[1]にもあるとおり，信頼性と安全性は異なる概念である．

表 1 分離手法の例 (文献[3] pp.126 表 2 より転載)

共有リソース	時間的な分離	空間的な分離
プロセッサ	・ タスクスケジューリング+タスク実行時間の保護(実行時間と到着頻度の監視) ・ パーティションのスケジューリング+パーティションの時間保護	・ マルチコアプロセッサ ・ ハードウェアマルチスレッディング ・ レジスタバンク
メモリ	・ 共有メモリへのアクススケジューリング+アクセス時間の上限を保護	・ 専用メモリへの配置 ・ MMU/MPU を用いた他パーティションのメモリ領域へのアクセス制限
周辺デバイス	・ 共有デバイスへのアクススケジューリング+アクセス時間の上限を保護	・ パーティション毎の専用デバイス ・ 他パーティションの専用デバイスレジスタへのアクセス制御
バス	・ 共有バスへのアクススケジューリング+アクセス時間の上限を保護	・ パーティション毎の専用バス ・ 他パーティションの専用バスへのアクセス制御

安全性を確保するための手法の一つである，S/W 分離に関しては，機能安全規格 IEC61508 を自動車分野向けに適用した規格である ISO26262 の中で，パーティショニングという概念が定義されている．また，航空機分野における機能安全の文献[2]にも同様の概念が定義されている．こうした中で，文献[3]では，パーティショニングを「リソース X を共有するコンポーネント間で，リソース X を介して故障が伝播しないようリソース X を分離すること」(文献[3] pp.122)と定義しており，さらに，S/W を分離する手法として表 1の例を示している．本節では文献[3]を参考に，各共有リソースで適用可能な分離手法を述べる．

2.1. プロセッサの分離

プロセッサの分離の内，時間的な分離はタスクスケジューリングとタスク実行時間の保護の組合せにより実現する手法がある．

空間的な分離の手法として，マルチコアプロセッサや，ハードウェアマルチスレッディングに対応した H/W を用いて，S/W に直接コア，スレッドを割り当てる手法がある．また，レジスタバンクを用いる手法もある．

プロセッサの分離を実現する方法として，ハイブリッド OS 技術がある．ハイブリッド OS 技術は単一 H/W 上で複数 OS を分離動作させる技術である．1 コアで実現する場合には，割込みを調停し，OS をスケジューリングすることにより，OS を時間的に分離して動作させる[4]．また，マルチコアを使用し，各コア上で OS を空間的に分離して動作させる方法もある[5]．

2.2. メモリの分離

メモリの分離の内，時間的な分離の手法について，共有メモリへのアクセスをスケジューリングし，さらに，アクセス時間の上限を保護する手法がある．

空間的に分離する手法としては，パーティション毎に専用のメモリ領域を割り当てる方式や，MMU(Memory Management Unit)，MPU(Memory Protection Unit)により，他パーティションのメモリアクセスを制限する方式がある．

メモリアクセスを制限する手法として，仮想化技術がある．仮想化技術は H/W リソースを仮想化することにより，複数の OS を動作可能とするものである．これにより，単一 H/W 上で複数 OS が動作する環境において，他の OS に影響を与えずに OS の再起動が可能となる．

CPU 負荷の観点で仮想化技術を分類すると，S/W エミュレーションにより H/W を模擬する方式(S/W エミュレーション方式)と，H/W の仮想化支援機能を利用して実現する方式(H/W 仮想化支援方式)に分けられる．

組込みシステムにおいて，S/W エミュレーション方

式を適用した場合，割込み応答時間が増加することが報告されている[6]．一般的に，S/W エミュレーション方式はオーバヘッドが増加することが想定されるため，想定するシステムに適用可能か評価が必要となる．H/W 仮想化支援方式に関しては，近年，H/W 仮想化支援機能に対応したハイパーバイザが提案されているが[7]，まだ一般的とはなっていない．

2.3. 周辺デバイスの分離

周辺デバイスの分離の内，時間的な分離の手法について，共有デバイスへのアクセス時間をスケジューリングし，さらに，アクセス時間の上限を保護する手法がある．

空間的に分離する手法ついて，パーティション毎の専用デバイスを用意する方法，あるいは，他パーティションからデバイスのアクセス制限を行う手法がある．

2.4. バスの分離

周辺バスの分離の内，時間的な分離の手法について，共有バスへのアクセス時間をスケジューリングし，さらに，アクセス時間の上限を保護する手法がある．

空間的に分離する手法ついて，パーティション毎の専用バスを用意する方法，あるいは，他パーティションからバスのアクセス制限を行う手法がある．ただし，一般的な組込み機器では，バスを空間的に分離する H/W は用意されておらず，問題になる可能性がある．

3. 課題

安全 S/W と非安全 S/W が依存関係を持ったまま共存する場合，一般的に非安全 S/W の安全レベルを安全 S/W の安全レベルに引き上げる必要がある．ここで，今回想定している安全 S/W はリアルタイム制御処理であり，例えば，外部デバイスに対する I/O 制御を行う．また，非安全 S/W は情報系処理であり，例えば，ネットワークからダウンロードされたアプリケーション等の可能性がある．こうした場合に，非安全 S/W の安全レベルを安全 S/W が要求する安全レベルに引き上げるのは現実的ではない．

そこで，非安全 S/W と安全 S/W を分離し，非安全 S/W に問題が発生し，処理が停止した場合であっても，安全 S/W に影響を与えないようにする必要がある(S/W 分離動作)．さらにシステムの信頼性を向上させるため，問題が発生した S/W を再起動することにより，情報系処理の機能を復旧し，再度，システム全体として機能提供を可能とする(S/W 再起動)．この際，可能な限りシステムが停止している時間を短くするために，非安全 S/W の再起動を高速に実施する必要がある．本 S/W 構成を図 1に示す．

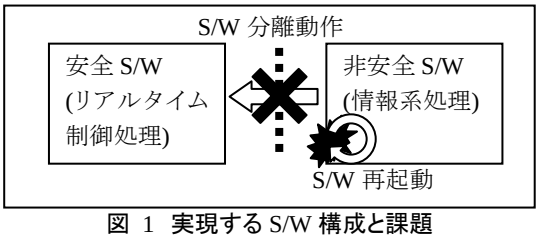

図 1 実現する S/W 構成と課題

S/W 分離に関しては，前節で示したように時間的な分離と空間的な分離がある．単一 OS 上で複数 S/W の時間的な分離を行う場合，開発ターゲットに応じて毎回タスク設計などの S/W 設計・実装が必要となり，開発コストが増大する．そこで本稿では，可能な限り空間的な分離の手法を用いて開発コストの低減を図る．S/W の空間的な分離に際して，単一 OS 上で複数 S/W の空間的な分離を実現することも可能と考える．しかし，安全 S/W ではリアルタイム制御処理を行うため，RTOS を用いる必要がある．この場合，RTOS を用いて汎用 OS が提供するような情報系処理を実現するのは現実的でない．

そこで，安全 S/W の動作する RTOS と非安全 S/W の動作する汎用 OS を空間的に分離し((1) OS 分離動作)，非安全 S/W により汎用 OS に問題が発生し，処理が停止した場合でも，安全 S/W が動作する RTOS に影響することなく，汎用 OS のみを再起動し((2) OS 再起動)，非安全 S/W を復旧可能となる基盤 S/W を提案する．以上をまとめると，以下の(1)，(2)の両方を実現する必要がある．

(1) 同一 H/W PF 上で RTOS と汎用 OS を動作させる(OS 分離動作)．
(2) 汎用 OS 側の非安全 S/W 障害で汎用 OS が停止した場合でも，RTOS 側の安全 S/W に影響がないように，汎用 OS のみを再起動する(OS 再起動)．

3.1. OS 分離動作

OS 分離動作を行うには，マルチコアによるハイブリッド OS 方式と仮想化技術を用いる方式が考えられる．近年，マルチコアを搭載した H/W PF は増加しており，マルチコアによるハイブリッド OS 方式が実現可能な H/W PF は多い．これに対して，仮想化技術のうち，S/W エミュレーション方式について，割込みオーバヘッドが増加する可能性が考えられ，システムに応じた評価，および性能チューニングが必要となる可能性がある．また，H/W 仮想化支援方式について，仮想化支援機能を搭載している組込み向け H/W PF はまだ少ない．

以上から，本稿では，マルチコアによるハイブリッド OS 方式を採用する．

マルチコアによるハイブリッド OS 方式を用いて複数 OS を分離して動作させる場合，特に以下の点が課題となる．

3.1.1. H/W リソース分離

OS を分離動作させる際，各 OS で同じ H/W リソースを使用しないように H/W を設定するという課題がある．この課題を解決することにより，ある OS が動作中に他の OS の挙動を阻害することがなくなる．なお OS 共通に使用される割込みコントローラなどの H/W リソースについても検討が必要である．

3.1.2. 複数 OS 起動

初期起動時における各 OS の起動方法について，H/W の制約等に応じて，OS イメージのロード方法や OS の起動順等を決定するという課題がある．

3.2. OS 再起動

3.2.1. デバイス初期状態化

通常，OS が再起動する際に，H/W PF 全体にリセット(以下，H/W リセット)を行うが，この処理は動作中の他 OS の動作を阻害する．そこで OS 再起動時には，H/W リセットではなく，OS の初期化処理から実施する必要がある．ここで，OS の終了時，デバイスドライバによっては，H/W リセットを想定した処理となっており，デバイスの状態を OS 起動前の初期状態に戻していない．こうしたドライバが存在する場合，OS 再起動時のデバイスの初期化処理に失敗することがある．そのため，デバイスの状態を OS 起動前の初期状態に戻した上で，OS 起動処理を実行するという課題がある．

3.2.2. OS イメージ再利用

システムの信頼性を向上させるため，OS の再起動を高速に実施させる必要がある．つまり，OS 再起動時にはブートローダを使用せずに再起動を行う必要がある．そのため初期起動時に使用した OS イメージを OS 再起動時も利用できるようにする，という課題がある．

3.3. 課題まとめ

本稿での目標は，OS 分離動作によりシステムの安全性を向上させること，および，OS 再起動によりシステムの信頼性を向上させることである．以上の課題をまとめると図 2のとおりとなる．

すなわち，システムの安全性確保の観点で，空間的パーティショニングにより OS を分離し，汎用 OS のアプリケーションが RTOS を破壊しないようにする必要がある．さらに，システムの信頼性向上の観点で，汎用 OS に問題が発生し，処理が停止した場合に，汎用 OS の再起動を実現可能とする必要がある．この際，信頼性を高めるために可能な限り高速に再起動可能とする必要がある．

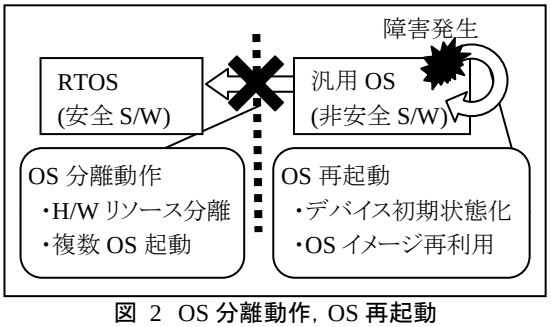

図 2 OS 分離動作，OS 再起動

4. 設計と実装

今回ターゲットとして想定する H/W および S/W を表 2に示す．

表 2 ターゲット H/W および S/W

H/W	PandaBoard CPU:OMAP™4430(Cortex™-A9(1GHz x 2)) MEMORY:LPDDR SDRAM 1GB
S/W	RTOS (安全 S/W) uITRON 4.0 (自製)
	汎用 OS (非安全 S/W) Linux 3.4 (OMAP™4 用 Ubuntu12.04)

今回選定した H/W について，マルチコアの組込みボードとして一般的な PandaBoard を使用する．選定した S/W について，リアルタイム制御処理(安全 S/W)に関しては，自製の uITRON 4.0 ベースの RTOS を用いる．情報系処理(非安全 S/W)に関しては，ユーザがダウンロードアプリ等を利用する可能性があるため，こうした用途で一般的な汎用 OS である Linux(R)を用いる．

4.1. OS 分離動作

通常，Linux は単一 OS で動作しているとして，H/W PF 上に搭載されているデバイスを可能な限り初期化する．そのため，同一 H/W PF 上で複数の OS を動作させるには，OS 動作に必要なデバイスを各 OS に割り当てるとともに，各 OS の起動についても検討する必要がある．本節ではこの内容について述べる．

4.1.1. H/W リソース分離

H/W リソース分離に際して，2 節の機能安全にて示されている S/W の空間的な分離手法を可能な限り用いて，H/W リソース分離を実現する．

4.1.1.1. プロセッサ

プロセッサに関して，今回のターゲット H/W はマル

チコアプロセッサを使用しており，各 OS に対しコアを固定的に割り当てることにより，プロセッサの空間的パーティショニングを実施した．

4.1.1.2. メモリ

今回は，1GB の SDRAM 上に，各 OS 専用のメモリ領域を割り当てることとした．また，汎用 OS については，OS 再起動用の領域も別途割り当てた．これにより，メモリの空間的パーティショニングを実施した(図 3).

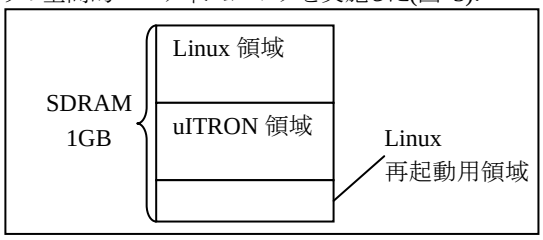

図 3 メモリ領域分割

4.1.1.3. 周辺デバイス

(1) I/O デバイス

周辺デバイスの内，I/O デバイスに関して，各 OS が占有可能なデバイスは各 OS に割り当てた．

uITRON は必要最低限の H/W リソースとして，タイマおよび UART を割り当てた．その他のデバイスは Linux に割り当てた(図 4). これにより，I/O デバイスの空間的パーティショニングを実施した．

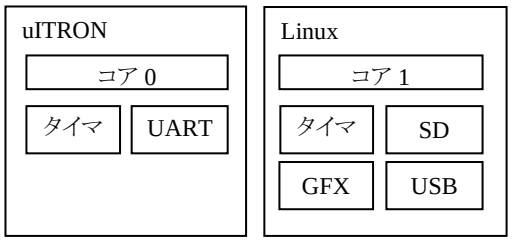

図 4 各 OS への周辺デバイスの割当て

この構成を実現するために，Linux に対し，以下の変更を実施した．なお，uITRON はタイマと UART のみを利用するため変更不要であった．

- 割込みコントローラのコア割り当て処理無効化
- シリアルデバイスの無効化
- uITRON 用タイマの無効化

(2) 内蔵デバイス

内蔵デバイスの内，割込みコントローラとリセット制御レジスタに対し，以下の設計を行うことにより uITRON に占有させた．これにより，内蔵デバイスの空間的パーティショニングを実施した．

・割込みコントローラ

割込みコントローラの中で，I/O デバイスを各コアに割当てる設定(コア割当て設定)はコア共有の H/W リソースであり，各 OS の処理に対し修正が必要であった．今回割込みコントローラのコア割り当て設定は uITRON 側で実施し，Linux では設定しないこととした．この理由について，Linux 側でコア割当て設定を実施した場合，Linux の起動時にコア割当て処理が行われる．ここで，コア割当て処理は，割込みを禁止した上で実施する必要があるが，この場合，uITRON の割込み応答に影響が出る可能性があるためである．

・リセット制御レジスタ

Linux は reboot コマンド等により再起動処理を行う際に，通常はリセット制御レジスタを操作し，H/W 全体のリセット処理を行う．今回のターゲット H/W の場合も H/W 全体のリセットを行うことが可能なデバイスが搭載されており，reboot コマンドにより，H/W 全体にリセットが可能となっていた．Linux の reboot コマンドにより H/W 全体のリセットを行った場合，正常に動作している uITRON にまで影響がでる．そこで，リセット制御レジスタにアクセスしている処理を修正し，OS の初期化処理から処理を開始することとした．

4.1.1.4. バス

今回のターゲット H/W はバスの空間的な分離が可能となるような特殊な H/W を搭載しておらず，バスの空間的な分離は実施できなかった．今回は，マルチコアの H/W を用いて，空間的な分離の手法を用いることによる開発コストの削減を目的としており，バスの時間的な分離は実装しないこととした．

4.1.2. 複数 OS 起動

複数 OS を起動する場合，OS イメージのロードの処理等を行うブートローダを拡張し，各 OS を起動可能にする方式と，OS を拡張し，他方の OS を起動する方式が考えられる．今回の構成では，安全性の高い uITRON が安全性の低い Linux を起動することとした．そのため，ブートローダは各 OS イメージのロードと uITRON の起動のみを実施し，uITRON が Linux を起動する(図 5).

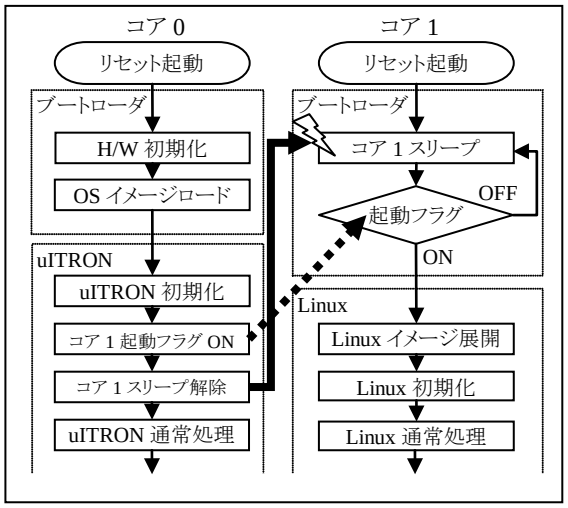

図 5 複数 OS 起動処理

4.2. OS 再起動

Linux では復旧不可能な障害が発生すると OS パニックとなり，汎用レジスタの状態などを出力した後，OS 機能を停止する．今回の構成では，一方の OS 障害が発生した際に，他方の OS に影響を与えることなく，OS 再起動を行う必要があるため，OS パニック処理に修正を行った．本節ではこの内容について述べる．

4.2.1. デバイス初期状態化

Linux が OS 機能を停止する際の処理について，デバイスドライバによっては H/W リセットを想定した S/W 処理となっている．そのため，各デバイスの状態を OS 起動前の状態に戻すかどうかは，各ドライバの実装レベルによる．そのため，各デバイスの OS 停止時の状態を調査し，必要に応じて OS 停止時のデバイス状態の修正が必要となる．そこで，I/O デバイス，および，内蔵デバイスについて検討を実施した．

4.2.1.1. I/O デバイス

Linux には Warm Boot(再起動時にブートローダ等の初期化処理をスキップする仕組み)を実現する機構(kexec)が存在しており，I/O デバイスのドライバではデバイス初期状態化対応が進んでいる．そのため，今回コア 1(Linux)に割り当てた I/O デバイス(タイマ，SD，GFX，USB)についてはデバイス初期状態化の対応は不要だった．

4.2.1.2. 内蔵デバイス

内蔵デバイスの中で，特に MMU (Memory Management Unit)は，デバイス初期状態化が必要となる．例えば，OS 終了時に，単に MMU の機能を無効にした場合，物理アドレスと仮想アドレスのマッピングがずれてしまうため，以降の処理が継続できなくなる(図 6)．

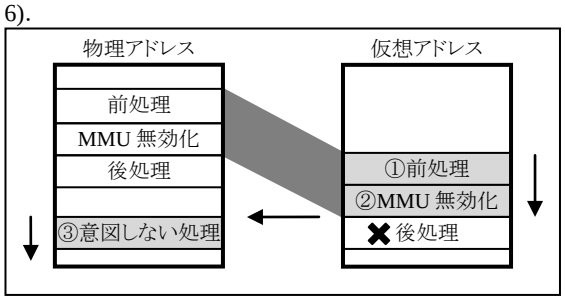

図 6 処理継続が不可となる例

この問題を解決するために，Linux 終了時に MMU の機能を無効にする処理については物理アドレスと仮想アドレスが等しいマッピングになるように変更した．これにより，MMU の機能を無効にした後も，OS 終了処理が継続可能となる(図 7)．

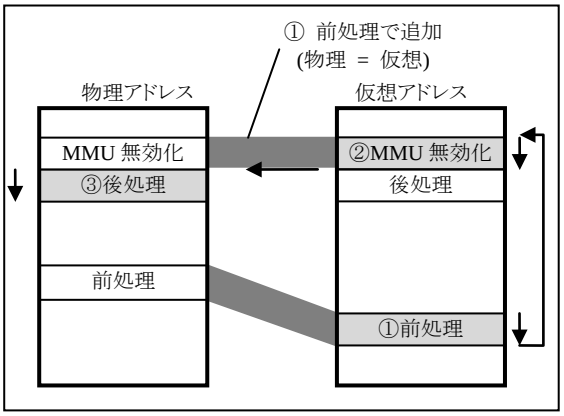

図 7 処理継続が可能となる例

4.2.2. OS イメージ再利用化

OS 再起動時について，再起動を実施するたびに同一の OS イメージを使用することとした．通常の ARM(R)用 Linux は，OS 起動時に OS イメージ内の GOT(Global Offset Table)領域を書き換えるため，再利用できない．そこで，OS イメージを ROM 化して使用可能となるオプション(CONFIG_ZBOOT_ROM)を有効にし，GOT 領域が変更されないようにした．

5. 評価と考察

5.1. uITRON と Linux の分離動作

uITRON と Linux の分離動作の評価として，Linux 側で動画再生処理を実施しながら，uITRON 側の制御処理が動作し続けることを確認した．

Linux と uITRON の分離動作について考察する．

Linux のユーザアプリケーションは Linux が管理する H/W リソース上で動作する. 今回 H/W リソースについては, 空間的な分離を実施したため Linux のアプリケーションから uITRON が破壊されることはない. ただし, Linux 上のドライバに関しては, uITRON 側の H/W リソースにアクセスすることが可能なため, uITRON 上の安全 S/W が破壊され, 安全性が保てない可能性がある. これに関しては, ユーザにはドライバをインストールさせないようにするなどの対処が必要となる.

また, uITRON から Linux の保護に関して, 通常の使用方法であれば uITRON タスクが Linux を破壊することはない. ただし uITRON から Linux の H/W リソースに対して, アクセス制限を書けているわけではないので, uITRON タスクから Linux 側を破壊することは可能である.

5.2. Linux 再起動

Linux の OS パニックを模擬する簡易的なドライバを用いて Linux に OS 障害を発生させた. この結果, uITRON 側制御タスクに影響することなく, Linux が再起動可能となることを確認した.

また, Linux の再起動時間評価として, 今回開発した Linux の再起動時間, および, 初回起動時間を評価した. 結果を表 3に示す.

表 3 OS 起動時間内訳

再起動	初回起動
4.5 [s]	4.7 [s]

今回開発した Linux の再起動にかかる時間は 5 秒程度であり, 初回起動と同程度の起動時間となった. これにより, 一般的な処理時間で OS 再起動を実現できた. なお, 初回起動と比較して再起動が 0.2 秒速い点について, 初回起動時は, コア 0 からコア 1 に対してコア起動処理を送信し, コア 1 のエントリポイントへ処理を移す処理が含まれており, この影響と考えられる.

以上のことから, Linux 側に障害が発生した場合, Linux のみを高速に再起動可能となり, 信頼性の高いシステムを構築することが可能となった.

5.3. ソースコード変更量

ソースコードの変更量を表 4に示す. 全体で 100 行程度の修正となり, 小規模な修正となった. このことから, 修正範囲は少なく, 今後の OS バージョンアップにも容易に追従可能と考えられる.

表 4 ソースコード変更量

	変更量	
	追加	削除
uITRON	57 行	9 行
Linux	30 行	10 行
計	87 行	19 行

6. 今後に向けて

本節では, 今回提案した基盤 S/W の改善点について検討する.

6.1. H/W リソース分離の容易化

今回, Linux のソースコードを直接修正することで H/W リソースの分離を行った. 開発容易性の観点から, ソースコードを直接修正するのではなく, 設定プロファイルなどの形で, OS に依存しない共通プロファイラなどにより, 各 OS 間で H/W リソース分離を行うべきである.

最近の組込み Linux では, H/W PF ごとの搭載デバイスの差異を吸収する目的で Device Tree[9]と呼ばれる仕組みでデバイスを管理することも可能となっている. Linux 起動時に, Linux カーネル起動イメージとは別に, Device Tree のフォーマットに沿ってデバイス搭載状況が記載されたファイルを使用することで, カーネルイメージは単一ながら, 異なる H/W PF に対応することが可能となる. Linux では, Device Tree 対応が進んでいるため, 今回使用した自製の uITRON 等で Device Tree のサポートを行うことにより H/W リソース分離の容易化が可能となると考える.

6.2. デバイス初期状態化の改善

デバイス初期状態化について, I/O デバイスに対しては修正不要であった. これは, 4.2.1節で述べたとおり, 今回使用した Linux には Warm Boot を実現する機構が搭載されていたためである. 同様に他の汎用 OS を用いて OS 再起動を行う場合, I/O デバイスのデバイス初期状態化が必要かどうかは, その OS が Warm Boot に対応しているかどうかで判断できる.

ここで, Warm Boot に対応していない汎用 OS について, 現状では汎用 OS が使用する個々の I/O デバイスが正しく再起動できるかどうか確認しなければならず, 今後, 改善が必要と考えられる.

6.3. OS 再起動処理の保護

汎用 OS 側の障害が RTOS 側に影響しないという観点では, 今回提案した基盤 S/W により目的を達成できた. しかし, 汎用 OS 側の障害により, OS 再起動処理が正常に動作しない場合、OS 再起動できない可能性がある. この場合, 安全 S/W が動作し続けるという観点で,

システム全体の安全性は保たれるが，汎用 OS が動作しなくなるため，ユーザが期待する情報系処理を継続できず，システム全体の信頼性が低下してしまう．

そこで，汎用 OS の再起動処理が利用するメモリ領域を保護し，確実に OS を再起動することで，システムの信頼性を上げるといった改善が必要と考えられる．

6.4. バスの空間的な分離

今回の H/W はバスの空間的な分離を実現可能な H/W を搭載していなかった．今回ターゲットとした H/W に限らず，一般的な組込み H/W ではバスの空間的分離が可能となるような H/W 機構は用意されていない可能性が高い．

今後は，OS に依存せずにバスの時間的な分離を実現するフレームワークを検討する等，バスの時間的な分離の開発コストを下げる手法を検討する必要がある．

7. おわりに

近年の組込み H/W の高性能化に伴い，従来は複数 H/W PF で実現していたシステムが，マルチコアの単一 H/W PF で実現可能になりつつある．こうしたシステムにおいて，安全 S/W と非安全 S/W を動作させる場合，非安全 S/W の障害により，安全 S/W の処理が停止する可能性がある．

そこで本稿では，非安全 S/W 側の OS 障害が発生した場合であっても，安全 S/W 側の OS に影響することなく，さらに非安全 S/W 側の OS を再起動可能となる基盤 S/W を提案した．

本基盤 S/W を実現するためには，OS 分離動作，および，OS 再起動を実現するという課題があった．この課題を解決するために，OS 分離動作については S/W の空間的パーティショニング手法を用いて，H/W の共有リソースを安全 S/W 側の OS と非安全 S/W 側の OS に分離し，非安全 S/W 側の OS 障害が安全 S/W 側の OS に影響しないように設計と実装を行った．また OS 再起動については，非安全 S/W 側の OS のデバイス初期状態化等により，安全 S/W 側の OS に影響することなく，非安全 S/W 側の OS のみの再起動可能となるように設計と実装を行った．

本基盤 S/W を評価した結果，コードの変更量は 100 行程度となり，小規模な変更となった．また，OS の再起動時間は通常起動時と同様の起動時間となり，今回提案した手法が有効であることを確認した．

今後は，H/W リソース分離の容易化などの課題解決を図る．

参考文献

[1] 向殿政男：コンピュータ安全と機能安全，IEICE Fundamentals Review, Vol.4, No.2 pp.129-135 (2010).

[2] Certification Authorities Software Team (CAST): Guidelines for Assessing Software Partitioning/Protection Schemes, CAST Position Papers, FAA (2001).

[3] 松原豊，高田広章：組込みシステムにおけるソフトウェア障害の安全対策，日本ソフトウェア科学会学会誌 コンピュータソフトウェア，Vol.30, No.1 pp.119-129 (2013).

[4] 金田一勉：第 17 章 Linux on ITRON - ハイブリッド構造の実装,インタフェース増刊 TECHI vol.16,CQ 出版社(2003).

[5] 菅井尚人，遠藤幸典，山口義一，近藤弘郁：シングルチップマルチプロセッサ上のハイブリッド OS 環境の実現 -OS 間インタフェースの実装-，情報処理学会第 66 回全国大会(2004)

[6] Wataru Kanda, Yu Yumura, Yuki Kinebuchi et al.: SPUMONE: Lightweight CPU Virtualization Layer for Embedded Systems, IEEE/IFIP International Conference on Embedded and Ubiquitous Computing, pp.144-151(2008)

[7] 岡部亮：QorIQ P4080 プロセッサ向けハイパーバイザの設計，電子情報通信学会 2013 年総合大会(2013)

[8] 出原章雄，山本整，東山知彦，落合真一：組込み CPU 向け高信頼基盤ソフトウェアの開発，情報処理学会第 75 回全国大会(2013)

[9] Device Tree: <http://www.devicetree.org/Main_Page>, (accessed 2013-06-17)