第二編
タスク分割のケーススタディ

なぜタスク分割か?

- タスクは、組込みリアルタイムOSを利用したときの、実行モジュールの基本単位
- タスク分割問題≒モジュール分割問題
- モジュール分割には、基本的な原則があるが(次ページ参照)、組込みシステムの特徴や、リアルタイム性を重視したり、開発現場の状況を考慮すると、より高度なタスク分割の考え方が重要
- それが最終的なシステムの機能や性能を大きく左右する

「モジュール化の原則」
＝ 何を根拠としてタスクに分けるか?

- **機能的結合**
  - 一つの機能を実行するために関連し合っている要素を結合
- **情報的結合**
  - 一つの情報(データ構造)を扱う複数の機能を結合(情報の局所化、オブジェクト指向)
- **連絡的結合**
  - 同じデータを参照したり、データの受け渡しがある要素を結合
- **手順的結合**
  - 問題を処理するために関係している複数の機能のいくつかを結合
    - フローチャートの一部を結合する、など
- **時間的結合**
  - 実行される時期が同じ要素を結合
    - 初期設定モジュール、終了モジュール、など
- **論理的結合**
  - 論理的・抽象的に関係ある要素の結合
    - 入出力用モジュール、編集モジュール、など
- **暗合的結合**
  - 特別な関係のない要素を結合
    - たまたま重複したコードを結合する、など
    - チューニングする際などでみられる

タスク分割の考え方

- **結合原則に沿わない関係である機能同士が、分割するポイントとなる**
  - それだけではないところが、タスク分割の難しい点

- **タスク分割方法は、設計者によって様々な考え方がある**
- **ケーススタディでは、悪い例とともに良い例も示すが、それが唯一の方法ではない**
- **とにかく柔軟に考え、いろいろな分割候補の中から最適なものを選ぶようにすることが必要**
- **どれが最適かは、設計者が判断しなくてはならない**

ケーススタディ(1)
検査装置

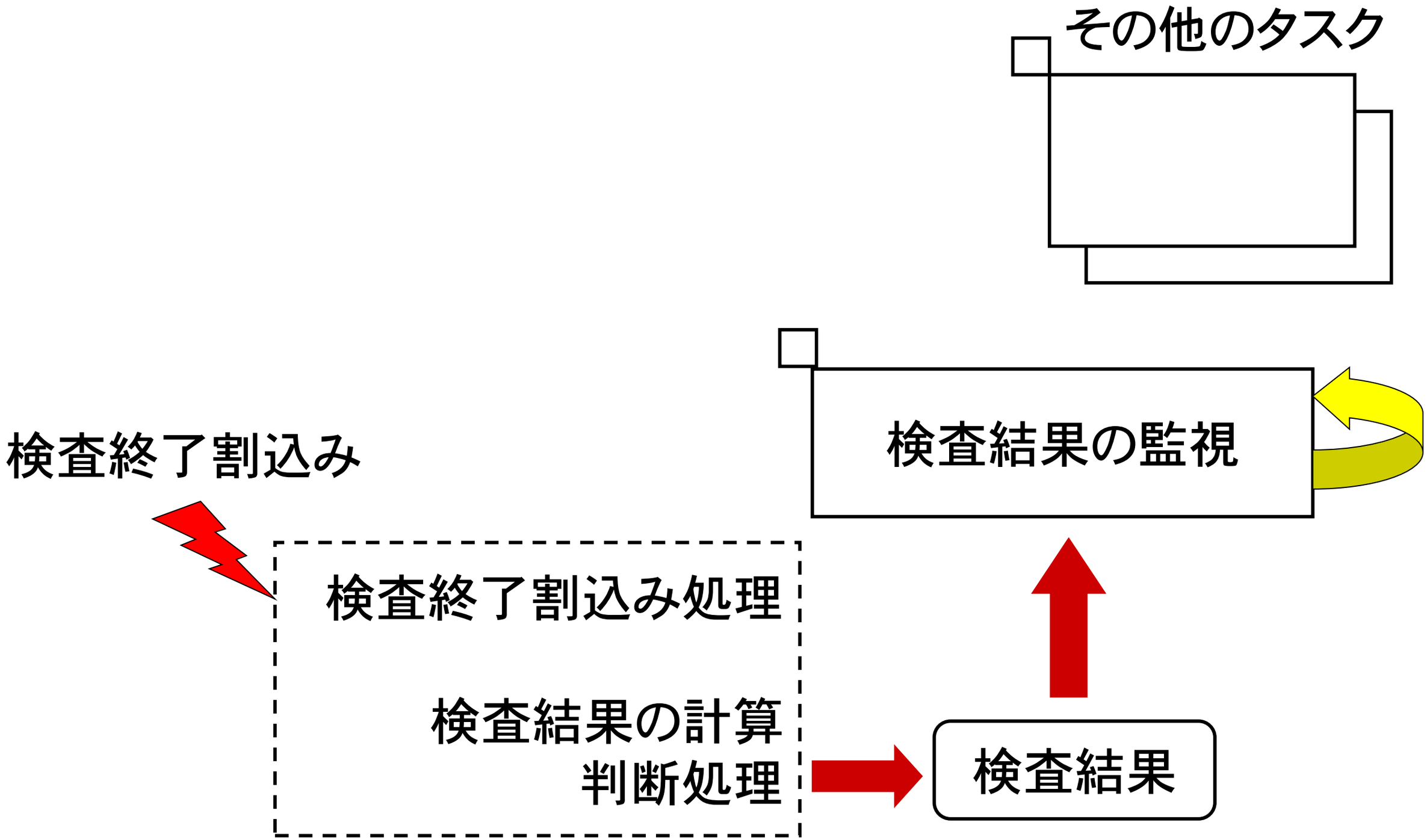
①処理の目的と内容

- 工場の製造ラインで、製造物の検査を行う機器
- 検査結果によって、製造物の合否を判断する
- 検査結果全体のばらつき(標準偏差)が大きい時は、製造ラインを停止する
- 試作を行った後、別途実機用のプログラムを開発する形で、プロジェクトを進める

②タスク分割の悪い例

- 標準偏差の計算や判断は、検査終了割込みがきっかけになるので、すべて割込み処理で行った
- メインタスクは、計算結果および判断段結果を受け取るのみ
- 試作段階では、これでも問題はなかった

②タスク分割の悪い例



③問題点は?

- 計算処理に時間がかかるため、ほかの割込みを取りこぼす
- 優先度が高いタスクに切替わるタイミングが遅い

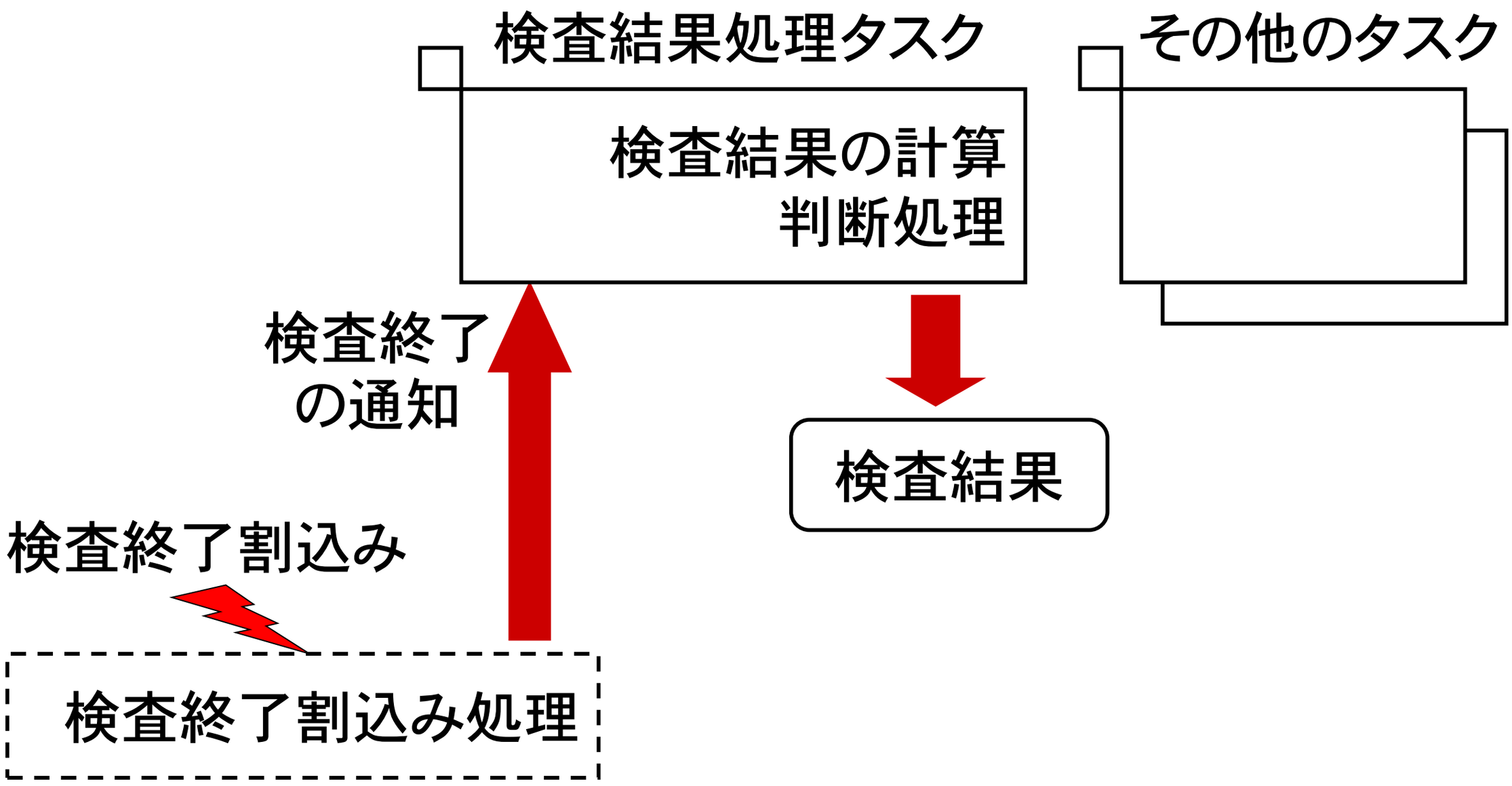
④改善のポイント

- 割込み処理では、複雑な計算や操作を行わない
- 時間がかかる計算を行うタスクは、優先度を下げる

⑤タスク分割の良い例

- 割込み処理は、検査終了をメインタスクに伝えるだけにする
- 判断および計算処理を検査結果処理タスクで行い、優先度を下げる

⑤タスク分割の良い例


検査結果処理タスク
検査結果の計算
判断処理
その他のタスク
検査終了
の通知
検査結果
検査終了割込み
検査終了割込み処理

⑥まとめ

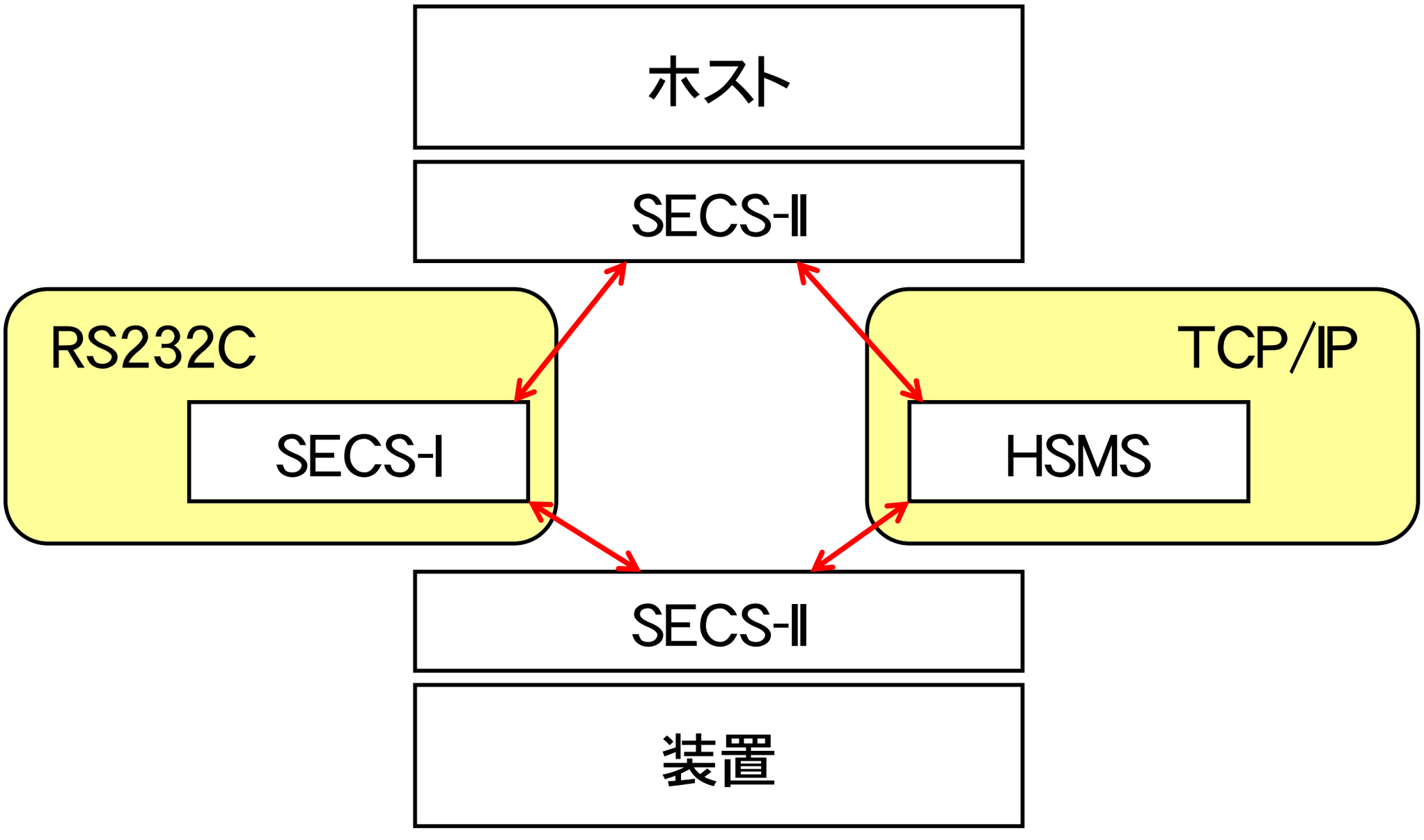
- **プログラムを試作しながらプロジェクトを進める場合(プロトタイピング手法)では、試作後のプログラムの扱いを最初から考慮しておく**
  - 骨格として使用するのか
  - 部分的に採用するのか
  - 使い捨てるのか
- **試作プログラムは、最終的な動作環境も考慮する**
  - 他に動作するタスクなども考慮する

ケーススタディ(2)
SECS通信

①処理の目的と内容

- **ある半導体製造装置にSECS通信機能を実装する**
- **SECS(エス・イー ・シー ・エス)とは、半導体製造装置など で使用される、標準的な通信プロトコル**
  - 実際の通信はSECS-I(RS232C)またはHSMS(TCP/IP)
  - 通信データは上位層のSECS-IIで意味づけされている

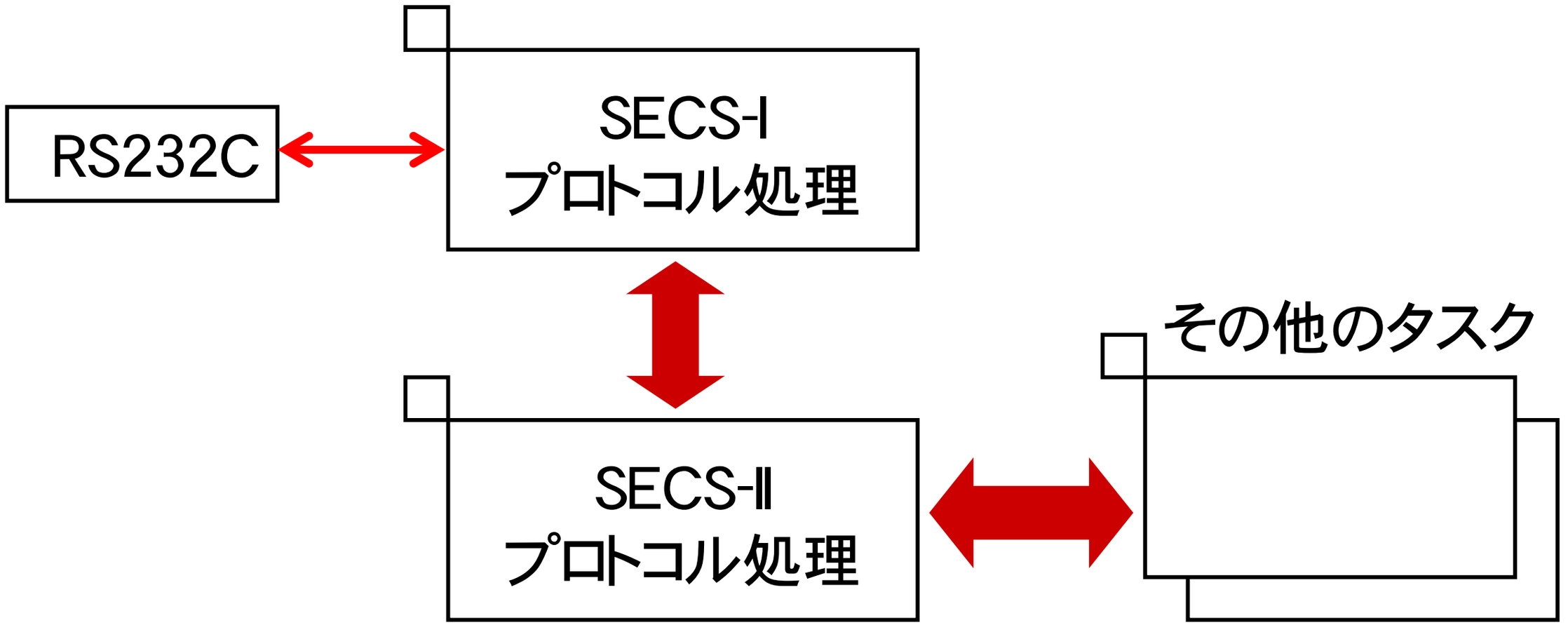
(資料)SECSプロトコル概要



②タスク分割の悪い例

- SECS-Iプロトコルに一つのタスク
- SECS-IIプロトコルに一つのタスク
- タスク内部で扱うデータを分離・独立させるため、別々のメモリ領域を用意して、コピーして使用した

②タスク分割の悪い例



③問題点は?

- タスク間で通信データのコピーが必要だったため、通信応答性能が悪い
- より高速な通信路(TCP/IP)を使用するHSMSに切替えても、通信速度の向上は見込めない

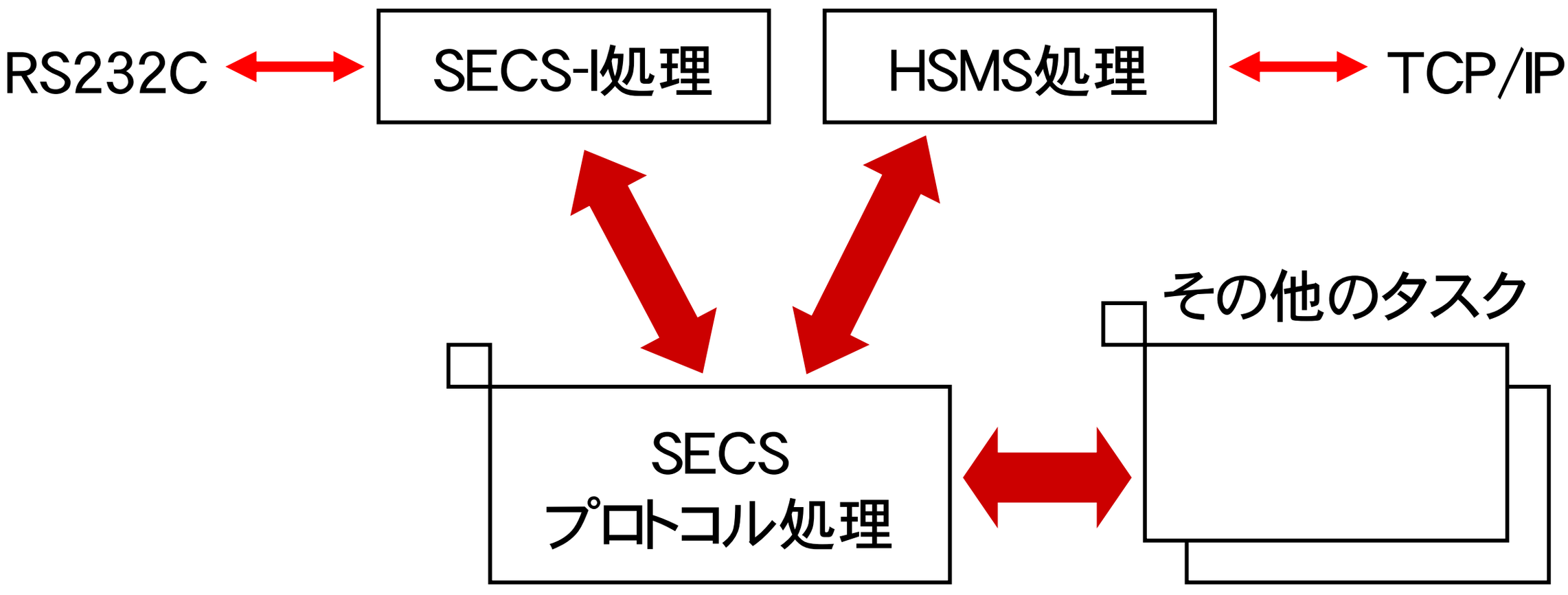
④改善のポイント

- SECSプロトコル処理は一つのタスクで行っても問題ないため、タスク内部を階層化することでSECS-IとSECS-IIに対応させる
- コピー処理の回数を減らす

⑤タスク分割の良い例

- SECSプロトコル処理タスクとして、通信処理を一つにまとめる
- 実際の通信を行う処理(SECS-IまたはHSMS)は関数ライブラリとした
- 一つのタスクにまとめたことで、コピー処理は不要になった

⑤タスク分割の良い例



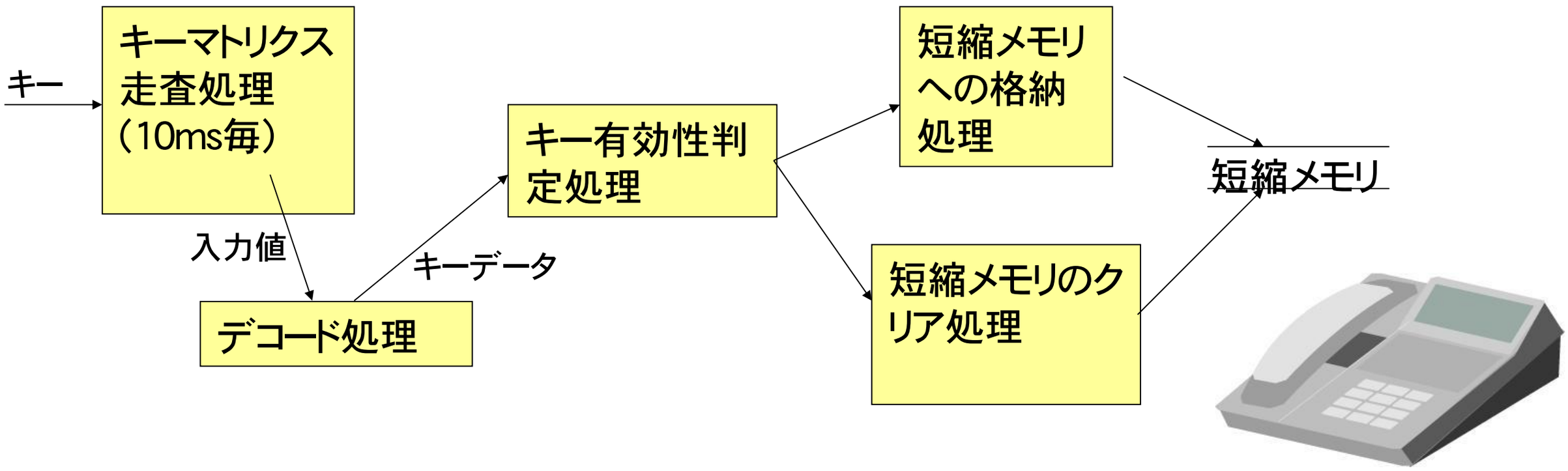
⑥まとめ

- **通信レイヤ(機能)=タスクと、限定して考えない**
- **タスク構成は性能なども考慮し、柔軟に検討すべき**
  - 複数の機能(通信レイヤ)を一つのタスクで実現
  - 一つの機能を複数のタスクで構成する場合も

ケーススタディ(3)
キー入力処理

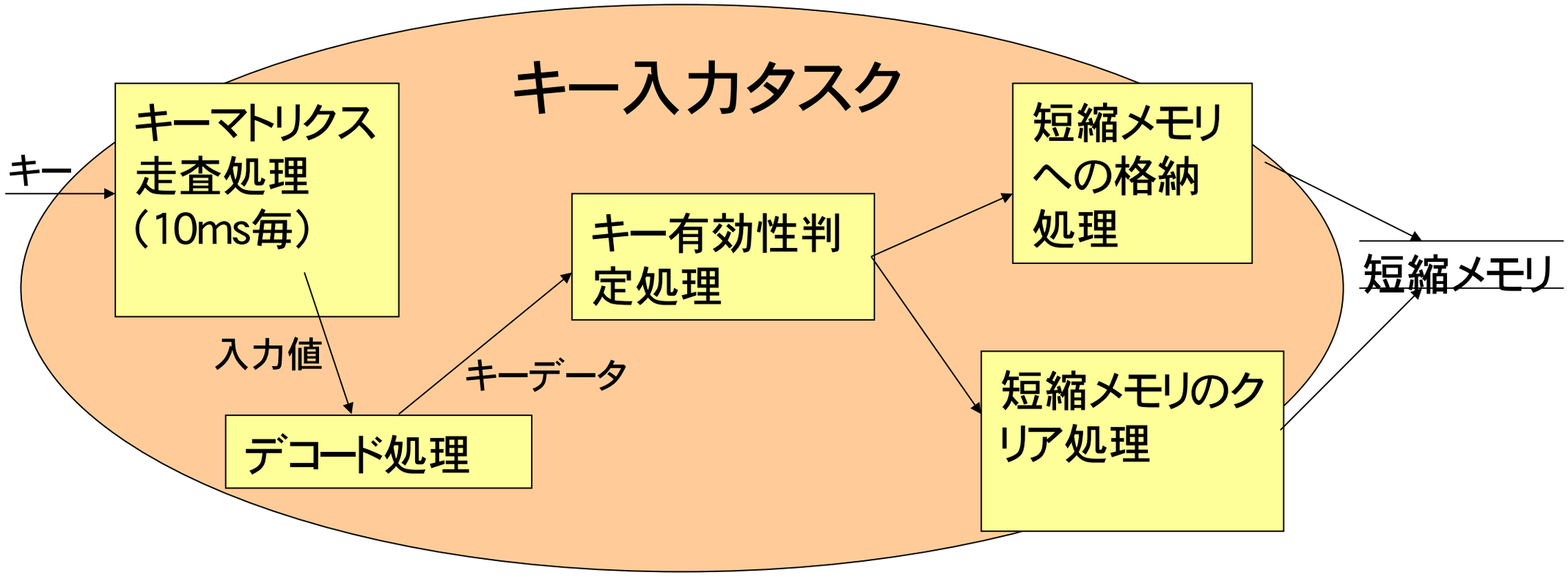
①処理の目的と内容

- 留守番電話機のキー入力処理にて
- キーマトリクス走査処理(10ms毎)
- 短縮メモリへ格納処理(30ms)



②タスク分割の悪い例

- キー関連処理(順次処理)をグルーピングし、1つの(キー入力)タスクとした
- その結果、キーマトリクス走査ができなくなるというトラブルが発生した



③問題点は

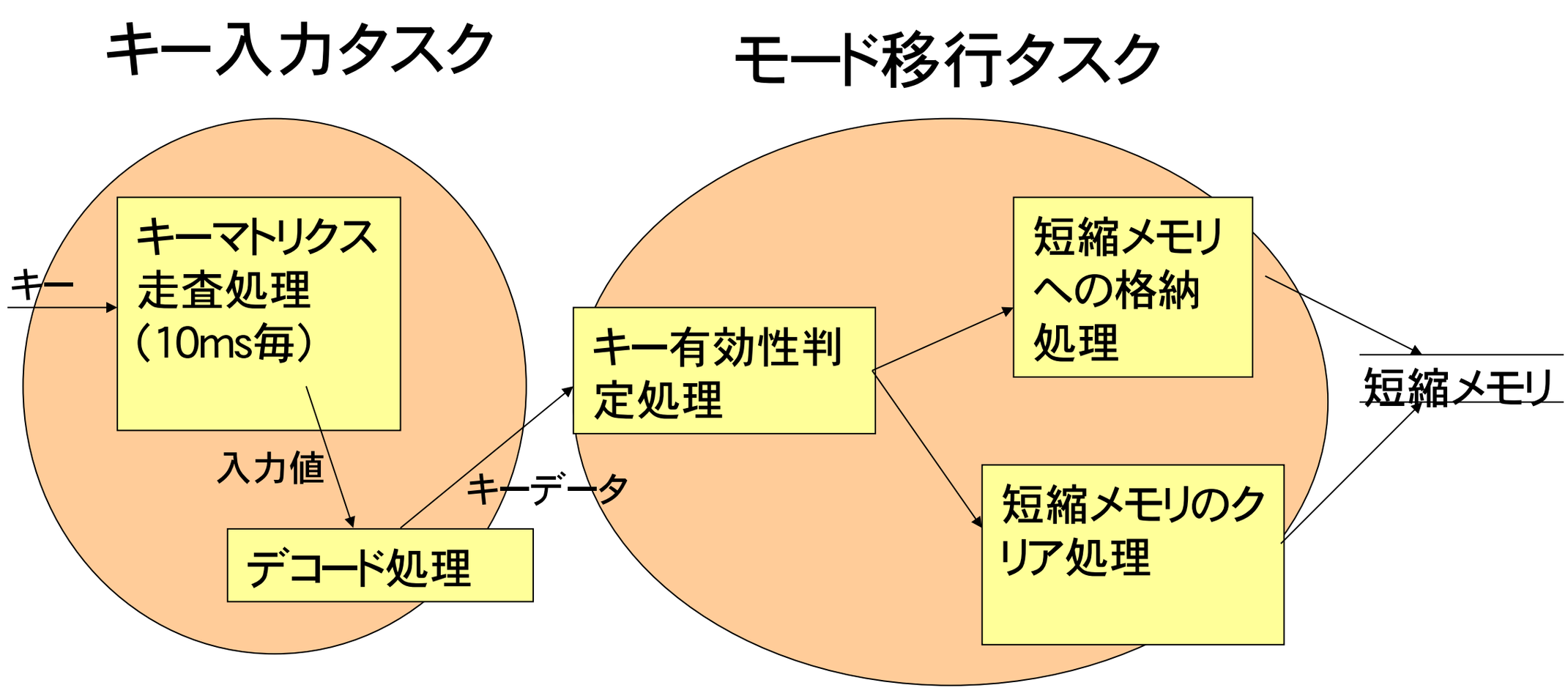
- **キーマトリクス走査は10ms毎に行う必要があるが、短縮メモリへの格納に30ms掛かるため、この間、キーマトリクス走査ができなくなってしまった**

④改善のポイント

- 順次処理とはいえ、処理時間の異なる処理は別タスクにする

⑤タスク分割の良い例

- キーマトリクス走査処理とデコード処理で1つ、キー有効性判定処理と短縮メモリ関連処理で1つ、それぞれタスク分割した



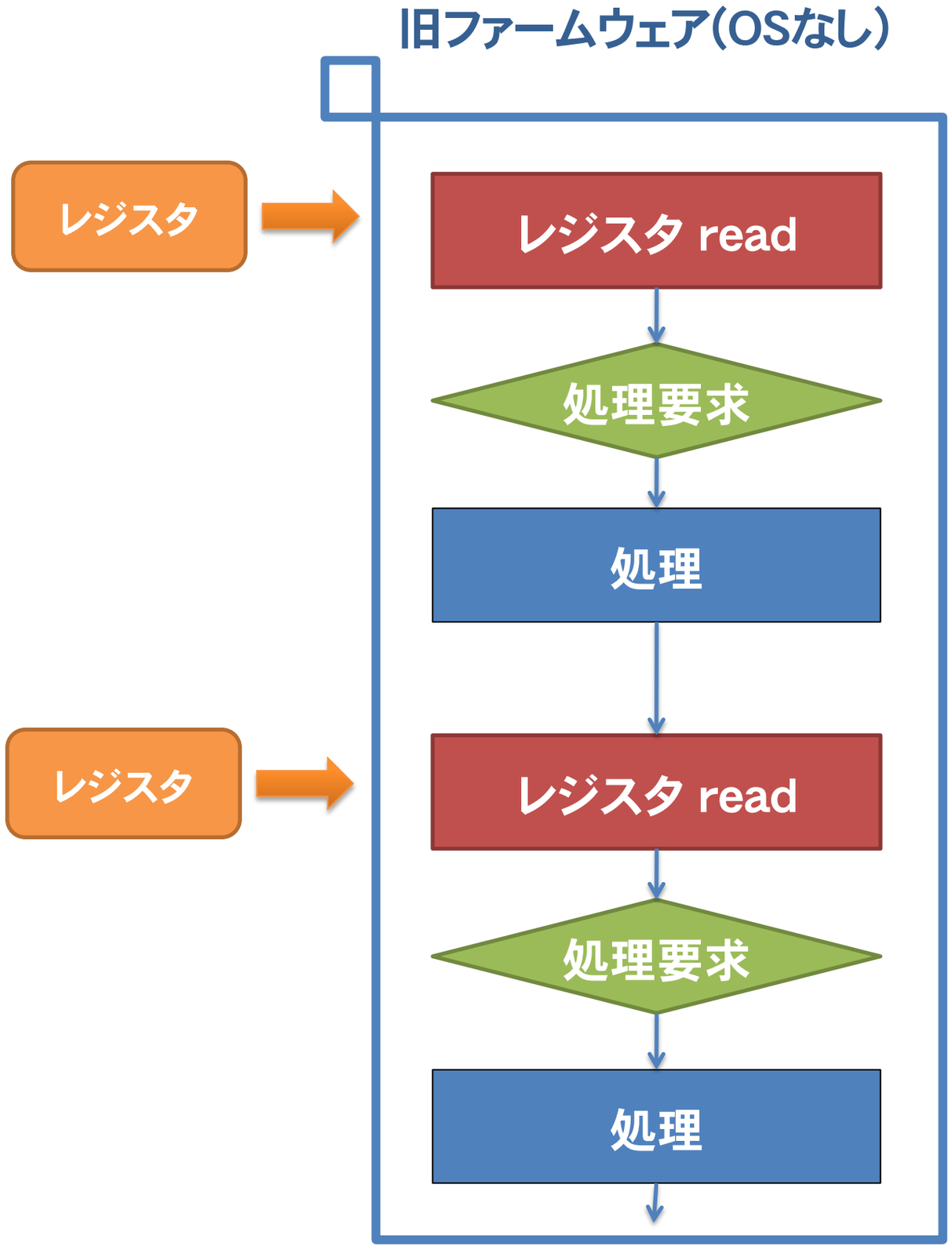
⑥まとめ

- 異なる時間(タイミング)で行われる処理は別タスクにし、リアルタイム性が損なわれないよう設計する必要がある
- 順次処理をグルーピングすると、またがるデータを少なくできるなどのメリットがある
- しかし、実は並行処理が潜んでいることは良くあり、注意深くタスク分割する必要がある

ケーススタディ(4)
システムファームウェア

①処理の目的と内容

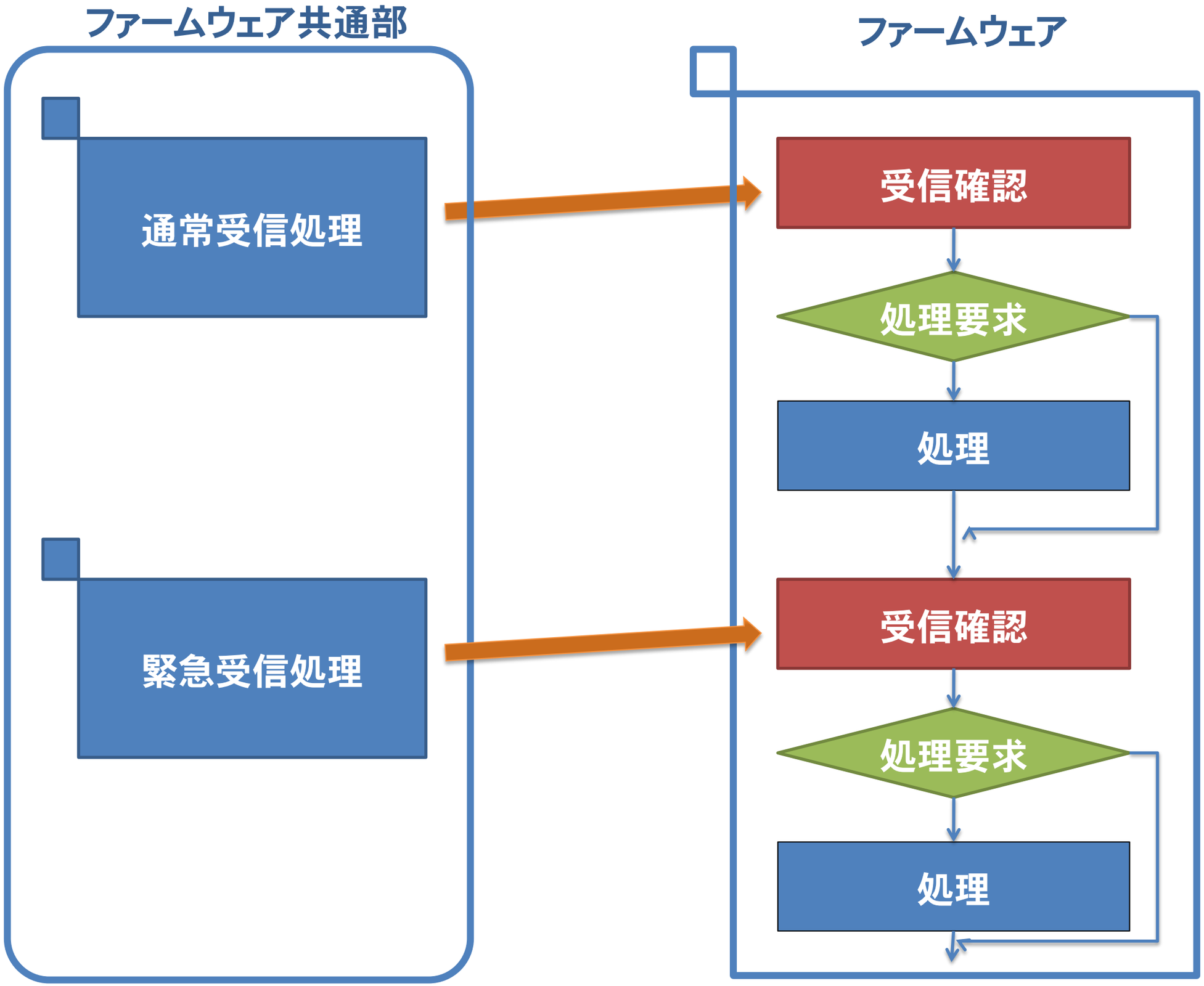
- **通信関連システムのファームウェアの例**
  - ベースとなるシステムでは、OS がないファームウェアにて制御していた
  - マルチタスク/リアルタイムという概念はなく、ポーリング処理により、処理振り分けを行う設計となっていた
  - 次期システムへの移行にて、上位システムとの通信機能を、他のファームウェアと共通化して供給されることとなった
  - また、同時にマルチタスク/リアルタイムOS を使用することとなった



②タスク分割の悪い例

- **このときに、上位からのコマンドを2 種類に分類し、優先度をつけた**
  - 通常入力:上位システムからの入力処理
    - DMA を使用した大量データの転送が可能である
  - 緊急入力:通常入力の処理を割り込んで処理が可能
    - 例えば、通常入力処理のキャンセル
- **この共通部を使用して、従来のファームウェアのフローをそのまま実装した**

②タスク分割の悪い例



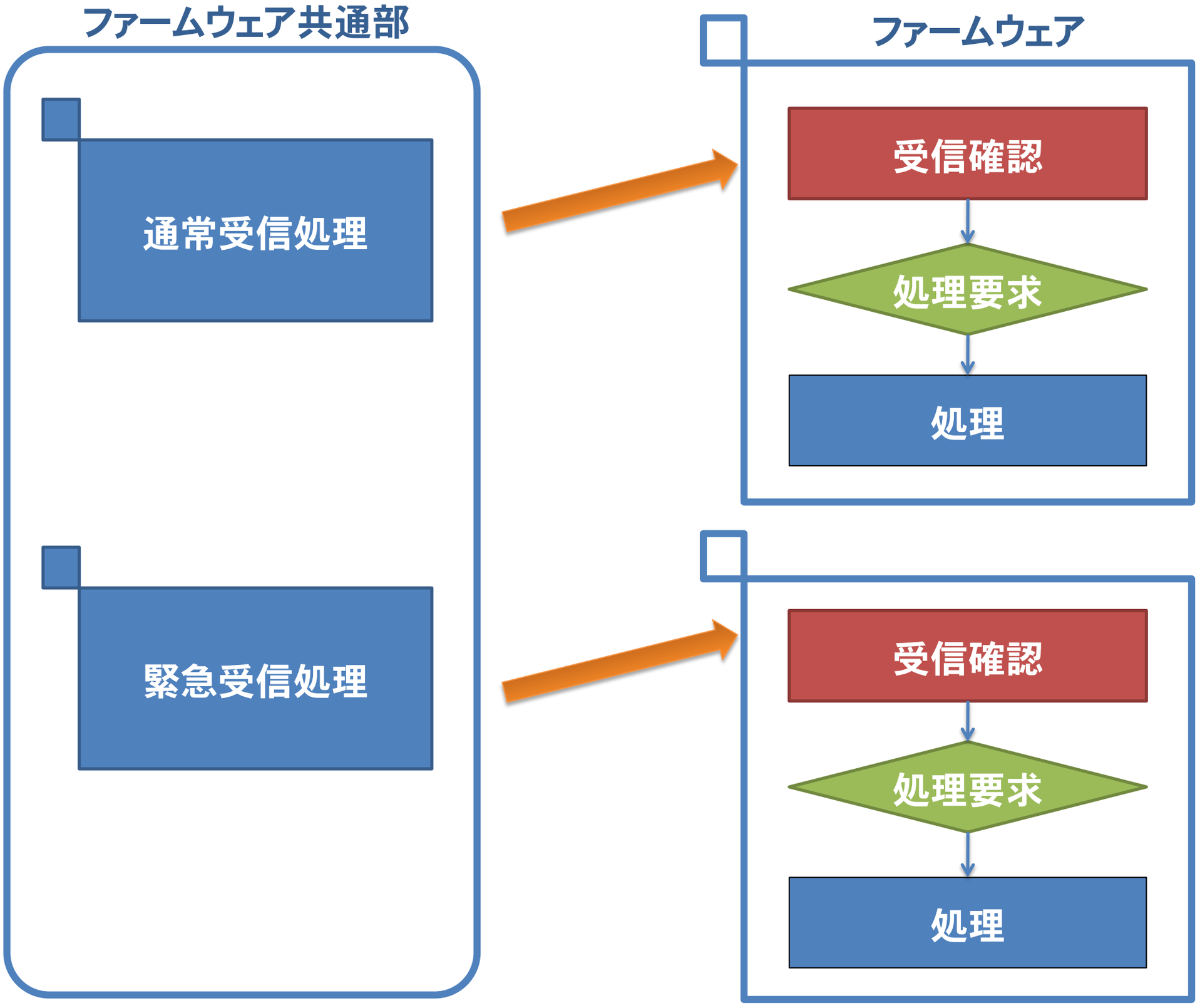
③問題点は

- 通常受信の個別ファームウェア処理が完了しない間、緊急受信処理を共通部が受信しても、個別ファームウェアは処理ができない
- したがって、通常処理の中断処理が行うことができない

④改善のポイント

- 個別ファームウェア部分のタスクを分割し、緊急受信側のタスク優先度を高く設定する

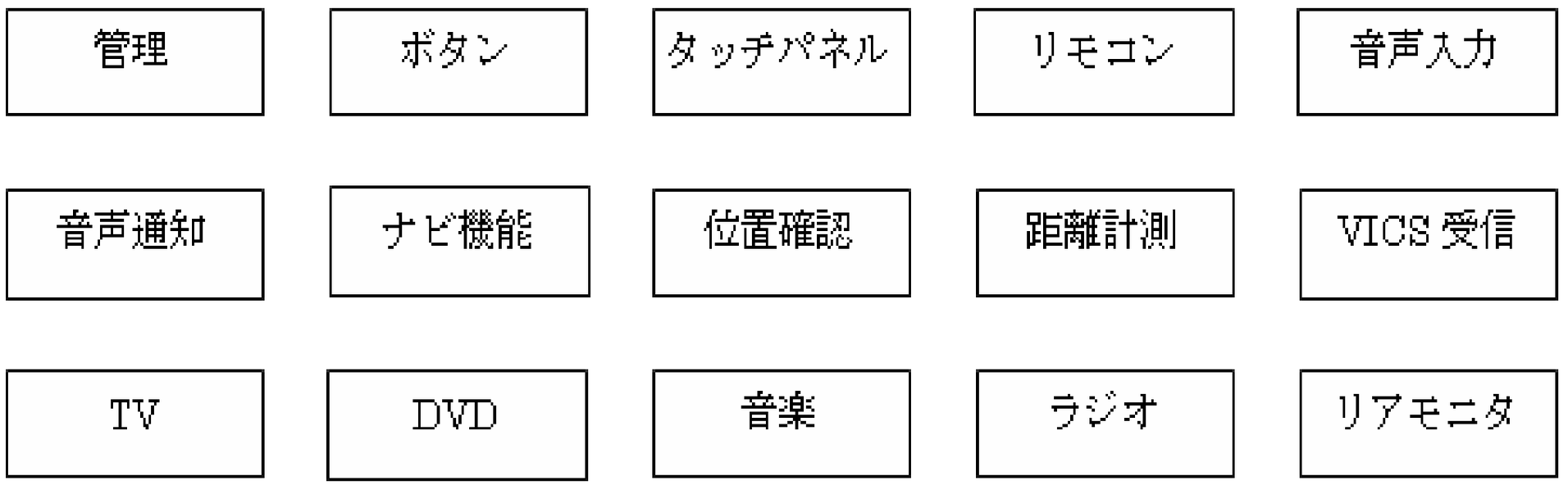
⑤ タスク分割の良い例



ケーススタディ(5)
システムアーキテクチャ
(カーナビ)

①処理の目的と内容

- カーナビの多様な機能を、複数のタスクに分割して実現したい
- (以下)カーナビの機能一覧



②タスク分割の悪い例

- 機能一つを一つのタスクとして実現した

③問題点は

- **タスク数が多すぎでタスク切替のオーバヘッドが大きく、システムが重くなった**

④改善のポイント

- 並行処理される単位、優先度が異なる処理単位に応じてだけで分割
- それによって、タスクの分割数を適切なところにする

⑤タスク分割の良い例


管理
ボタン
タッチパネル
リモコン
音声入力
入力
音声通知
ナビ機能
位置確認
距離計測
VICS 受信
TV
DVD
音楽
ラジオ
リアモニタ
エンターテイメント

⑥まとめ

- **並列実行できる単位でタスクを分割する**
- **分割数はバランスが大事**
  - 分割数が多いと、タスク切替の回数が増え、オーバヘッドが増える
  - 分割数が少ないと、効率よく並列実行することが難しい

第三編
タスク分割のガイドライン

タスク分割の考え方

- タスク分割方法は、設計者によっていろいろ考え方がある
- 考え方の例を示すが、これが唯一の方法ではない
- とにかく柔軟に考え、いろいろな分割候補の中から最適なものを選ぶようにすることが必要
  - どれが最適かは、設計者が判断しなくてはならない

設計の前に：作るのか、購入するのか

- **実際の開発では、RTOSやミドルウェアなどは、購入して組合わせる場合もある**
- **どの機能を作成して、どの機能を購入するのかは、最初に検討する必要がある**
  - 購入する場合は、そのミドルウェアなどが、どのようなタスク構成で、どのような機能を持っているかを知っておく必要がある
  - 「**自分が設計して作る部分はどこか?**」を明確にしておく

モジュール分割の理論

モジュール分割の目的と方法

- ソフトウェアを意味的、機能的な単位に分割することにより、可読性や再利用性、あるいは複数人開発での分担効率を上げるため
- ソフトウェア設計における最も基本的な部分であり、最終的な成果物の品質を左右する重要な設計要素
- モジュール分割には、いくつかの方法がある

データの流れに着目

- STS分割
  - データの流れを「S(Source)源泉」「T(Transform)変換」「S(Sink)吸収」に分割して、モジュールの階層構造を設計する技法
  - それぞれ、「入力モジュール」「処理モジュール」「出力モジュール」となる
- TR分割
  - データ(トランザクション)の処理単位で分割する技法
  - データの流れが分岐する場合に適用し、入力されたトランザクションにより、異なる処理を実行する場合に分割する
- 共通機能分割
  - STS分割・TR分割で分割されたモジュール中に共通する機能を持ったモジュールがある時、これを共通モジュールとして独立させる方法

データ構造に着目

ジャクソン法(JSP)

- 入出力のデータ構造に着目してモジュールの階層構造を導く技法
- 「JSP木構造図」(「基本」「連続」「選択」「繰返し」の構成要素からなる)を使用する

ワーニエ法

- 入出力データの関係からモジュールの階層構造を導く技法
- 「ワーニエ図」を使用
- 入力データを集合論によって分析し、選択・繰り返しと前処理・後処理の構造に整理する

STS分割

- 1. **機能の分類**
  - システム全体の中から、対象となる機能を分割
- 2. **最大抽象点を見つけ出す**
  - 「最大抽象入力点」これ以降は入力データが発生しない区切りの場所
  - 「最大抽象出力点」これ以降は出力データが発生しない区切りの場所
- 3. **最大抽象点を区切りとして、各モジュールに分割する**

TR(Transaction:トランザクション)分割

- 1. **機能の分類**
  - システム全体の中から、対象となる機能を分割
- 2. **データの流れが分岐する点を見つけ出す**
- 3.**入力されたトランザクションにより、異なる処理を実行する部分をモジュールに分割する**

RTOS上のタスク分割

- **基本的には静的なモジュール分割と同じ考え方で、各機能ごとに分割する**
  - 分割技法は「プロジェクトに合っているもの」もしくは「使いやすいもの」で問題ない
- **RTOS上のタスクとして分割するために、さらに以下の点に着目する**
  - 並列処理
    - 互いに独立した処理の並行実行
    - 処理時間の差の吸収
  - リアルタイム性の高い処理の優先実行

タスク分割作業の実際

タスク分割の前に

- **最優先で考えるべきことは、システム全体に求められている機能を実現すること!**
- **そのためには、まずはこれから作るシステム全体の機能を把握する**
  - 特定の機能にのみ注目していると、見落としてしまう機能が出てくる可能性がある

必要な情報の収集と整理

- **RTOSを使用する場合は、RTOSの機能を知って理解していないと、設計はできない**
  - 何ができて何ができないのか?
  - どの機能を使って、何を実現するのか

- **固有のテクニックも知っておくべき**
  - RTOS
  - TCP/IP
  - その他

- **次に各機能を、機能の種類ごとに分類する**
  - 着目すべき種類:通信、入力、出力、計算処理、その他

機能の整理と分割手順

- **最初に着目すべき点は「どのような入力があって、どのような出力が要求されているのか」**

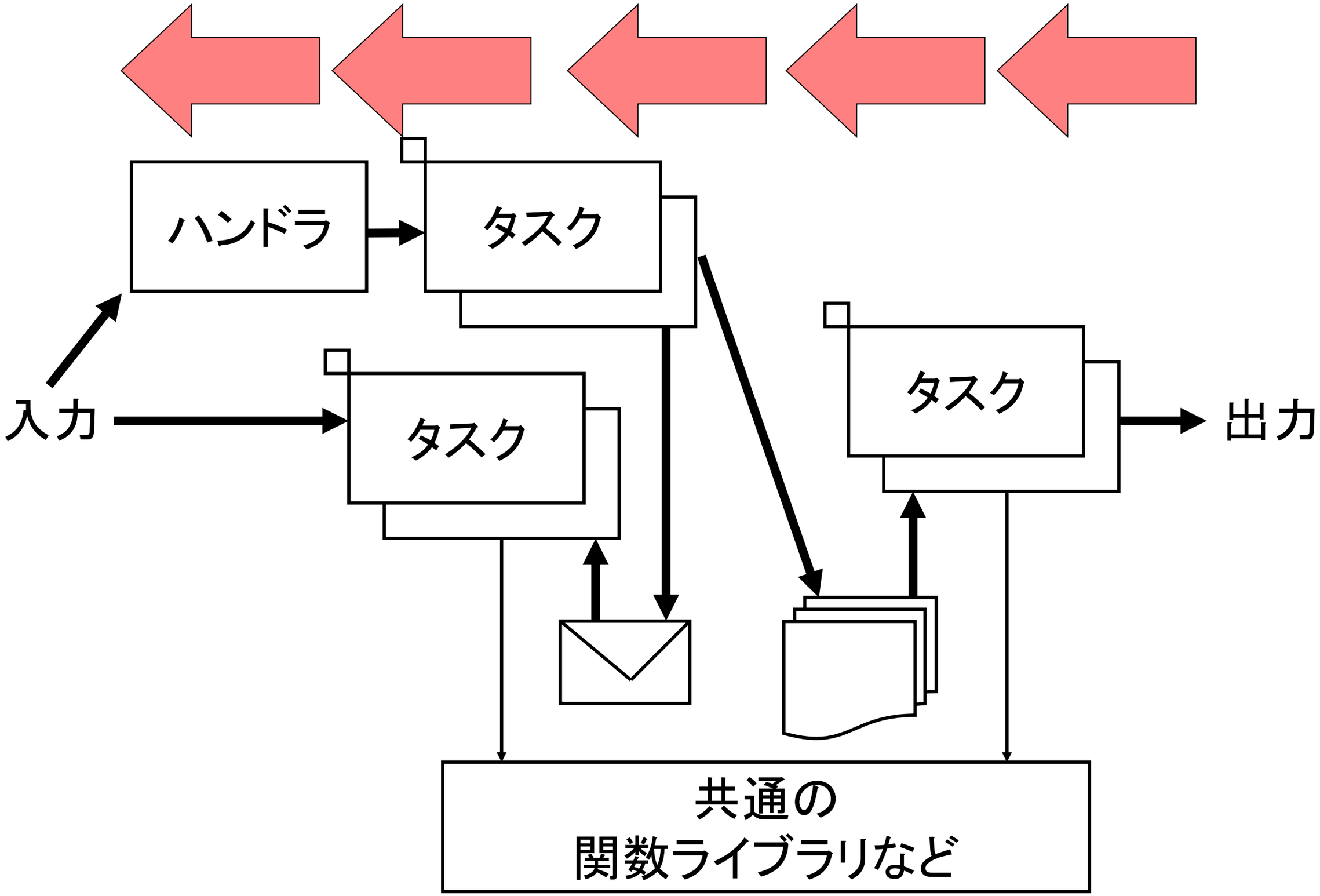
- **1.出力を行うために、どのような情報が必要か**
  - すべての情報がそろわないと出力できない
  - 出力側から考えると、整理しやすい
  - 矛盾する情報はないかも確認しておく
- **2.そのためにはどのような機能が必要か**
- **3.その機能のために必要な入力は問題ないか**

- **基本的には1.～3.を必要な分だけ繰り返す**

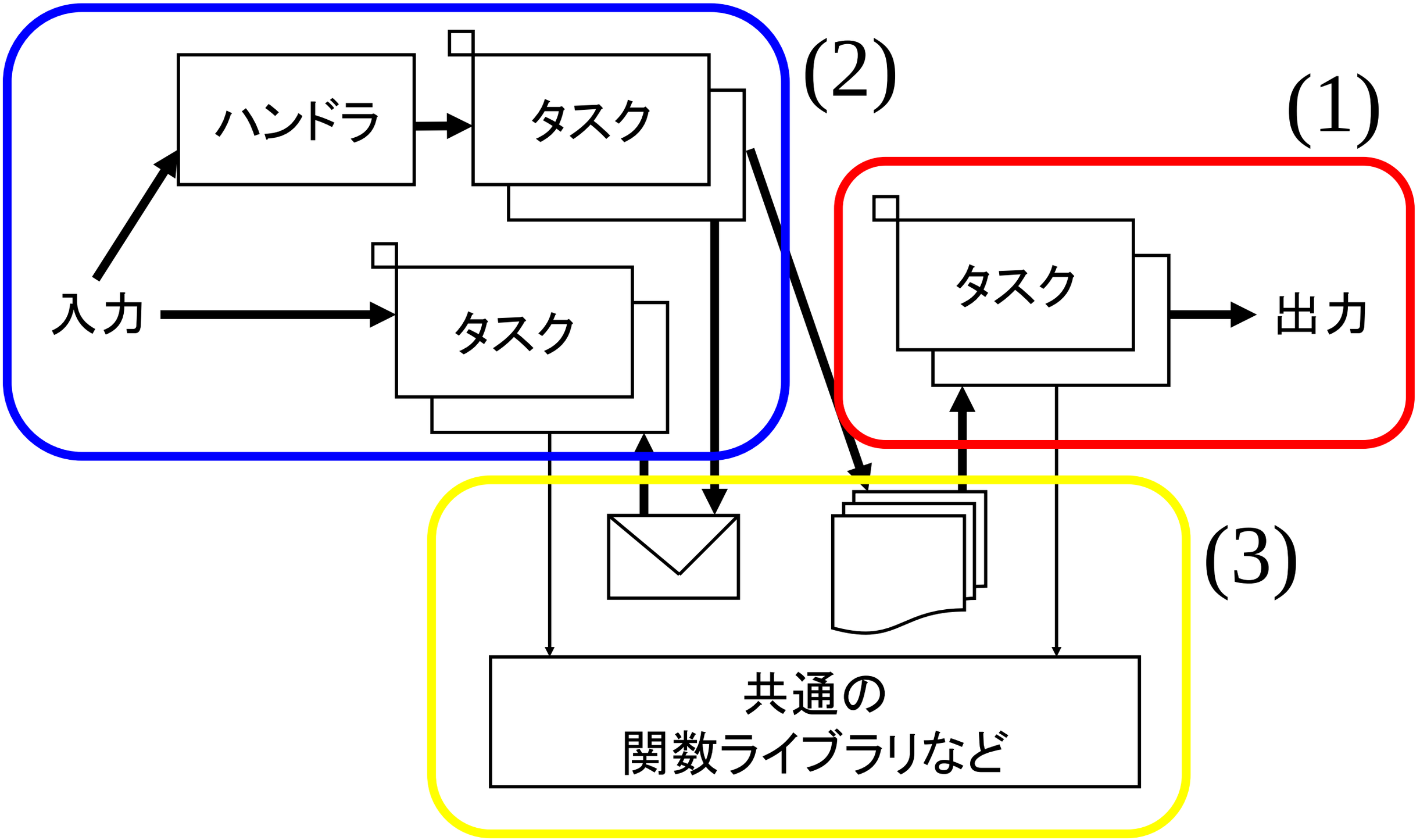
設計の順序(全体の流れ)

- **設計は、出力側から見ていくのが基本**
  - その次に入力側
  - 最後に共通の関数ライブラリなど
- **システム全体に求められている機能を常に意識する**
  - システムに要求される機能によって、どこが重要なのかは違ってくる

設計の順序(全体の流れ)



設計の順序(全体の流れ)



分割する機能の役割、ハンドラの役割および情報伝達方法

タスク分割は役割分割

- タスク分割作業(設計作業)は、どの機能をどの処理で行うかを決めるための作業
  - 必要な割込みハンドラは、ハードウェアの構成で決まってしまう
  - 単純に機能のみで判断するのではなく、動的な要求も考慮する必要がある
- どの機能をタスクで行い、どの機能をハンドラで行うのかも一緒に考える
  - 処理の構成や目的によって異なるが、ハンドラで処理を完結させたほうが良い場合もある
- それぞれの機能間の情報伝達方法を決めるのも、設計作業の一つ

タスク分割方法(並行動作に着目)

- **まずは並行して動作すべき機能に着目**
  - 並行して動作すべき機能が、まずはタスクの候補
- **機能の種類が違う場合には、並行して動作する必要がない場合もある**
  - 別のタスクにまとめられるかどうかを検討する
  - シーケンシャルに動作させた方が良いものもある
- **機能によっては、タスクで動作させずに、関数ライブラリでよい場合もある**

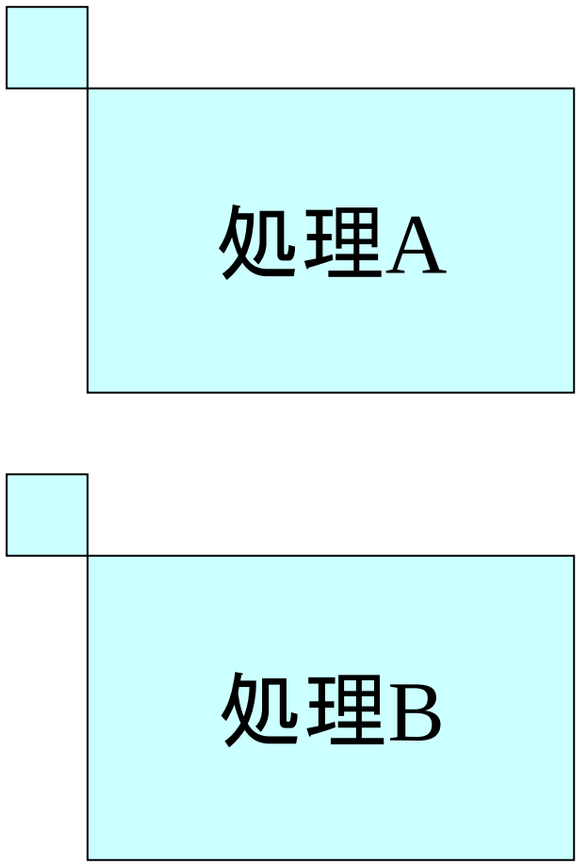
タスク分割方法(優先機能に着目)

- 次に優先的に処理すべき機能に着目
  - 非常停止や通信などで、システムに求められている機能をもとに検討する
- 優先すべき機能は、システムによって要求が違うので、注意が必要
  - たとえば同じネットワーク機器でも、ルータとネットワークプリンタでは、優先すべき処理がちがう

タスク間の情報伝達

- タスク間で、どのような機能を使って情報伝達(同期や通信)を行うのか検討する
  - RTOSの同期通信機能を使うのか?
  - グローバル領域を使うのか?
  - ハードウェアの機能を使うのか?
- 選択した情報伝達手段で、問題が無いかどうかを検討する
  - 複数の機能を同時に使えない場合もある

まずは並行動作



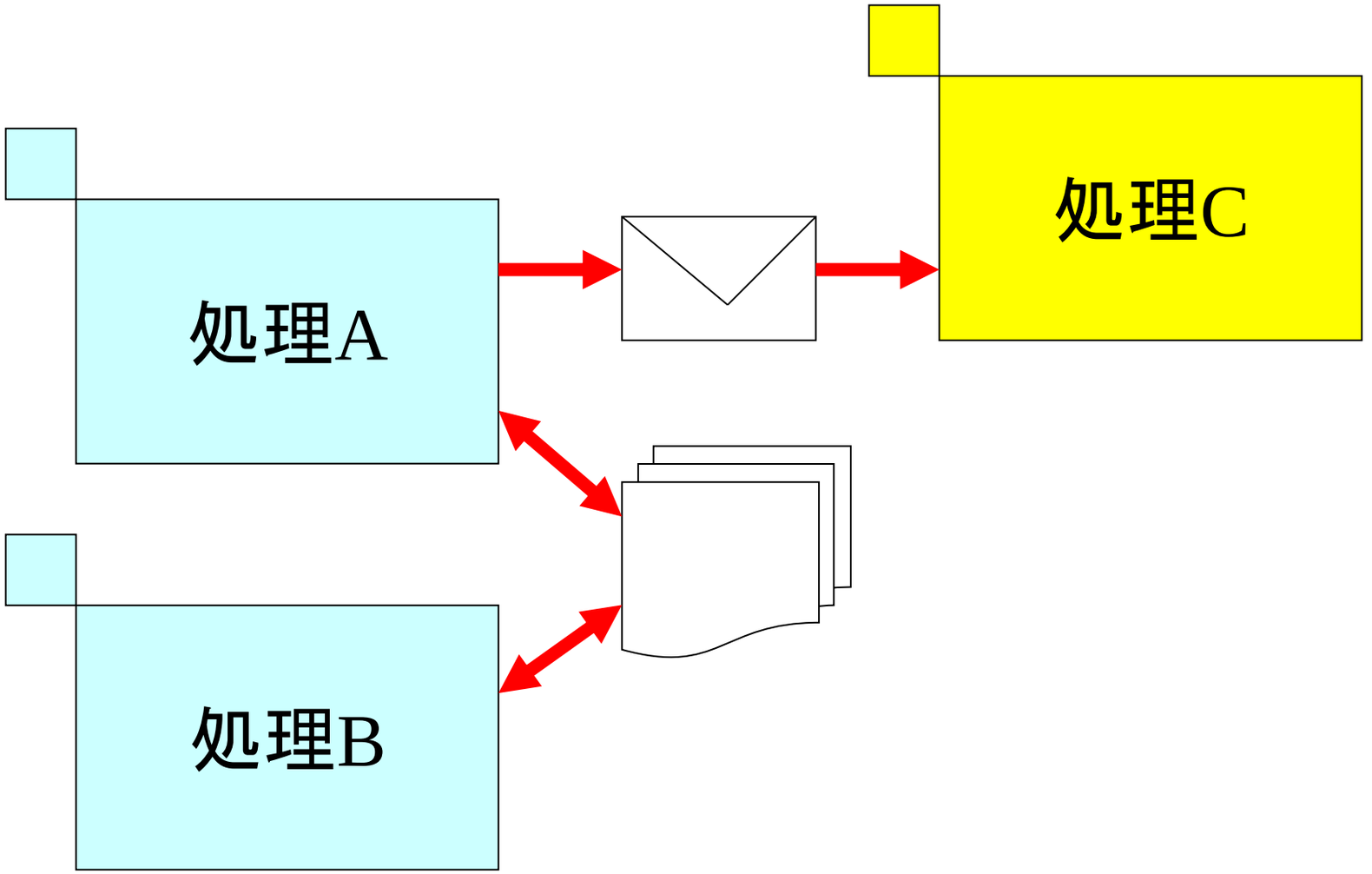
次は優先動作

処理A

処理B

処理C

そしてタスク間通信



優先度の設定はあとで

- **最初に、すべてのタスクを同一優先度とする**
  - 基準をそろえる
- **優先的に処理すべきタスクの優先度を高くする**
  - すべてのタスクが「優先的」という考えでは、うまくいかない
- **「犠牲」にしてよいタスクの優先度を低くする**
  - アイドルタスクやバックグラウンド処理タスクなど

タスクはイベントドリブンで

- タスクはできるかぎり、イベントが発生してから動作を開始するように設計する
  - イベントが無い間は待ち状態
- どのイベントが発生した時に、どのタスクが動作するのか、整理しておく
  - 同時にイベントが発生した時には、RTOSが優先度を使って判断する

関数ライブラリなど

- タスク設計後、関数ライブラリなどの共通処理部分を整理する
- グローバル領域のデータなども含めて、全体の整理が必要

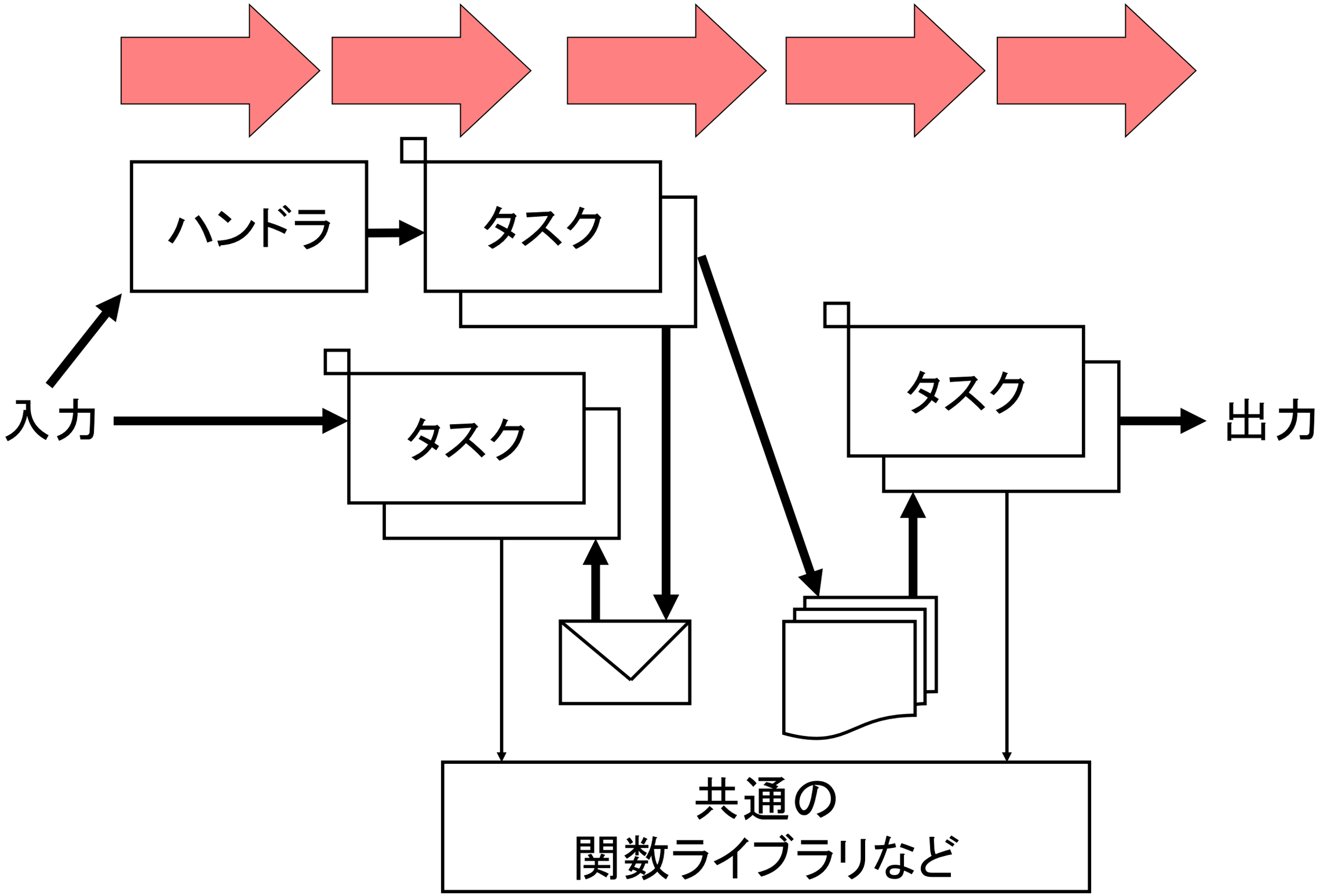
タスクとして明確化する

- **並行動作や優先動作の検討等をもとに、タスクとして明確化する**
  - つまりタスク分割を含めた設計作業のゴール
- **分割判断基準があいまいなままだと、その後の実装やテストがうまくいかない場合が出てくるため、文章化しておく**
- **意思疎通のためにもドキュメント化しておくことが重要**
  - 開発者が一人であったとしても、メンテナンスのために、資料が必要になる場合も多い
  - 人間の記憶はあいまいである

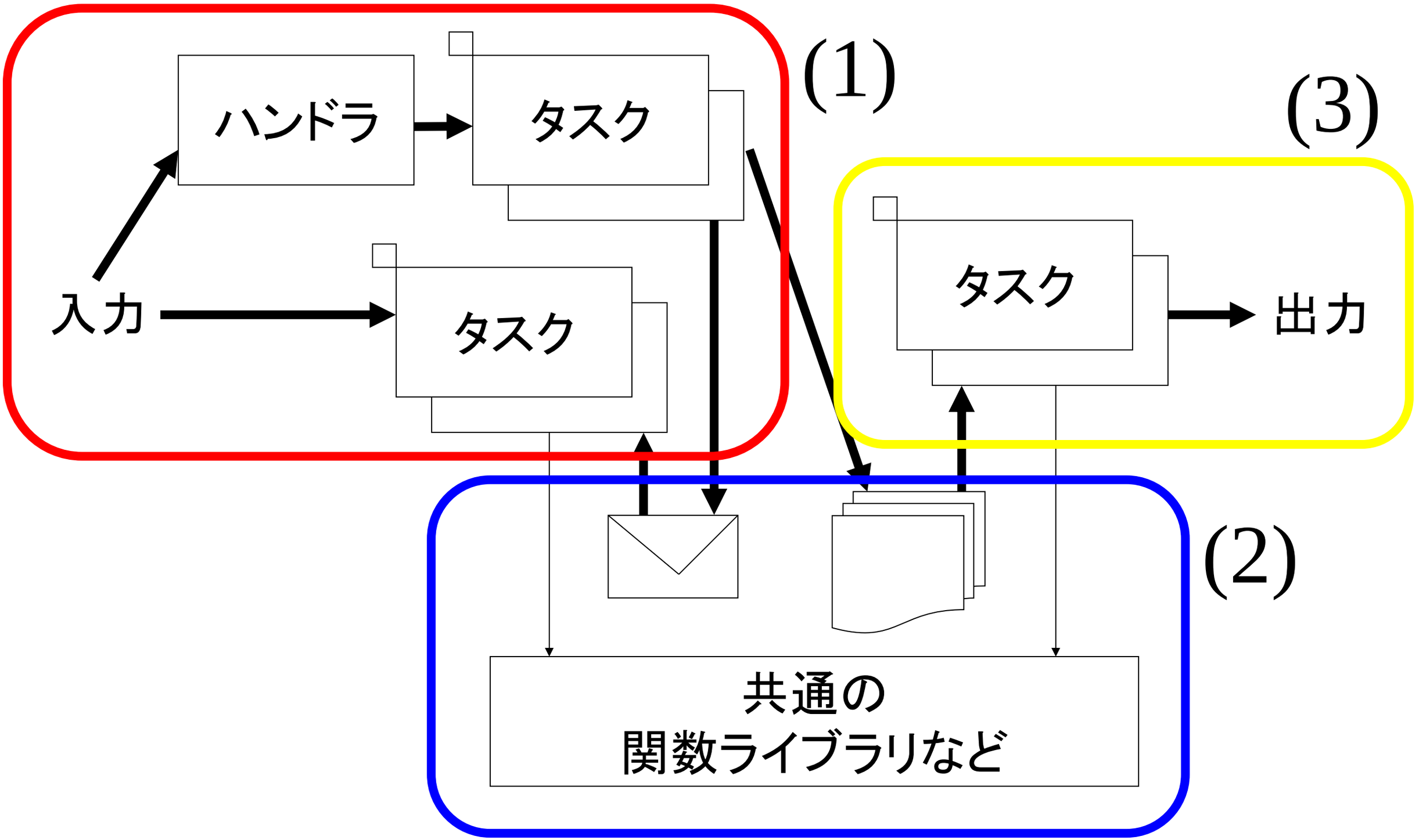
テストの順序

- **テスト内容や順序は、設計時に一緒に検討しておく方が良い**
- **テストは入口側から始め、出力側に向かって行う**
  - 入力が正しいことが保障できない状態で出力側のテストを行っても無意味
  - 正常な状態だけでなく、異常な場合のテストも必要
- **タスクが動作するタイミングについても確認する**
  - プリエンプションが発生した場合に問題ないか?など

テストの順序(全体的な流れ)



テストの順序



設計時に注意すべき点

考え方についての注意

- **考え方(着目点)の違いで、タスク分割方法も異なる**
  - いわゆる「正解」はない
- **OSやミドルウェアが持っている機能によっては、タスク分割の考え方が異なる場合もある**
  - 例えば、BSDソケットのタスク(プロセス)分割の考え方はUNIX向きだが、小さな組込みシステムにはあまり向かない

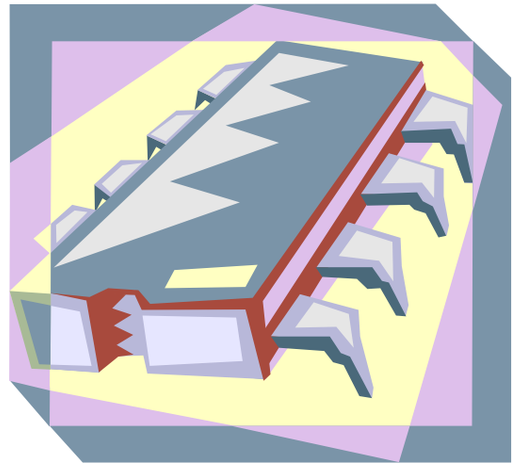
リソースや機能の制限

- **ROMおよびRAMの容量には限界がある**
  - 使用できるROMおよびRAM容量で、その機能が実現可能か?
  - メモリアクセス速度にも注意が必要
    - 期待した性能が出ない場合もある
- **使用するRTOSやミドルウェアの制限**
  - 直接的に実現したい機能がない場合、その機能を作る必要があるかどうかの判断も必要

適度な大きさと量に注意

- **あまり細かくタスク分割しない**
  - タスク切替えにも、時間がかかる
- **タスク規模(コード量や処理機能量)をそろえた方が見通しが良い**
  - 管理しやすくなる
- **一概にこの程度とは言えないので、難しい問題ではある**

プロセッサの性能



- **組込みシステムでは、性能の低いプロセッサを使用することが多い**
  - コストや実装面積などの問題で、選択の幅が限られる
  - 性能の範囲内で、機能が実現できることが重要
- **性能が高いプロセッサでも、無駄な動作が少なくなるように設計すべき**
  - 無駄な動作＝無駄に電力を消費する
  - 無駄な動作は、リアルタイム性能を低下させる原因になる可能性も

動的な生成と削除の問題

- **RTOSの動的な生成と削除は、必要なときに必要なものを用意する機能なので、便利ではあるが注意が必要**
  - 生成や削除にも時間がかかる
  - 削除したカーネルオブジェクトに連動したデータなどが、取り扱い不明になってしまう場合がある
    - メモリプールから取得したデータ
    - あるタスクがセマフォを獲得したまま削除された場合
- **アプリケーションが動作中は、なるべく動的な生成と削除を使用しないほうがよい**

なるべくリエントラントな処理に

- **リエントラント(再入可能)な関数にすることで、特定のタスクや処理に依存しにくい形式になる**
  - 再利用や汎用化が可能(保守性も向上)
- **リエントラントな関数にするために**
  - 作業用変数は関数内(Auto変数)で用意する
  - グローバル領域へのアクセスが必要な場合は、排他制御を行う
  - 情報を受渡しするためにポインタを使う場合は、使用上の注意が必要
    - 排他制御を関数内部で行うのか、排他制御をしなくてもよい領域を渡すのか、ルールを決めておく

スタックに対する配慮

- RTOSでは、各タスクごとにスタック領域が必要
  - サイズもタスクごとに決める
  - RTOSによっては、タスクで使用するスタック領域+割込みハンドラのスタック領域が必要
    - 多重割込みが発生する場合は、その分のスタック領域も考慮する
- 大きな配列データをなるべくスタックに取らない
  - 大きな領域が必要な場合は、領域の確保方法も検討する必要がある

デッドロックと優先度逆転

■ デッドロックは設計時に特に注意すべきである

- 状態によってデッドロックが発生しない場合もあり、必ず発生するとは限らない

■ 優先度逆転は、性能低下や動作不能も

- 設計時に見落としやすい
- 処理内容や状態によっては、システム動作不能にも

デバック時や機能追加時への配慮

- 初期設計時に注意しても、デバック時に安易にタスクを追加したりすると、デッドロックや優先度逆転が発生しやすい
- なるべく変更しやすい設計を心がける
  - 排他制御を行う基準を明確にする
  - 優先度は、多少の変更が可能なように、設定値には余裕を設ける
- その他…

リアルタイム性能を確保するために

本当にリアルタイム性が必要か?

- システムに要求される機能のうち、本当にリアルタイム性が要求される機能は、一部である場合が多い
  - 例えば、デジタルカメラの場合、シャッターを押す前の画像処理は、画像が多少粗くても、画像の更新が多少遅くても問題ない
    - シャッター押下時の処理は、リアルタイム性が要求される
    - ある程度の時間で処理しないと、操作性が悪いと感じられる

- 機能ごとに、要求されている時間を明確にする!

要求されている時間を明確にする

- **要求される時間は、機能によって違う**
  - 数μsから数十ms、あるいは数分の要求も
- **過剰な性能追求は、システム全体を不安定にさせたり、動作不能になる場合もある**
- **リアルタイム性が要求されない機能もあることを意識しておく**

処理時間を確定させる

■ ループはなるべく使用しない

- ループ回数が決まっていない場合は、処理時間が変動する
- 最大回数が決まっていても、処理時間の見通しが悪くなる

■ 再帰処理を使用しない

- 再帰回数によって、処理時間が変動する
- スタック(RAM)も余分に消費する

応答性能を良くする

- **できる限りイベントドリブンなタスクに**
  - すべて能動的な動作では、リアルタイム性能を確保するのは難しい
- **使うサービスコールを選ぶ**
  - サービスコールによって、処理時間が違う
  - 排他制御は、ディスパッチ禁止ではなく、できるかぎりセマフォやミューテックスを使用する
- **アイドル時間をなるべく多く**
  - アイドル時間が多い＝タスクが効率よく動作している証明

データの取り扱い

- **データは、その必要度によって取り扱い方が違う**
  - すべてのデータが必要な場合
  - 途中を間引いても良い場合
    - マウスの移動距離、音楽再生時のデータ、他
  - 最終データがあれば問題ない場合
    - デジタルカメラの画像データ、他
  - 後で処理しても良い場合
    - 測定器の統計データ、他
- **処理回数を少なくできるかどうかも検討する**
  - 処理回数を少なくして、応答性能を向上させる

割込み処理は短く

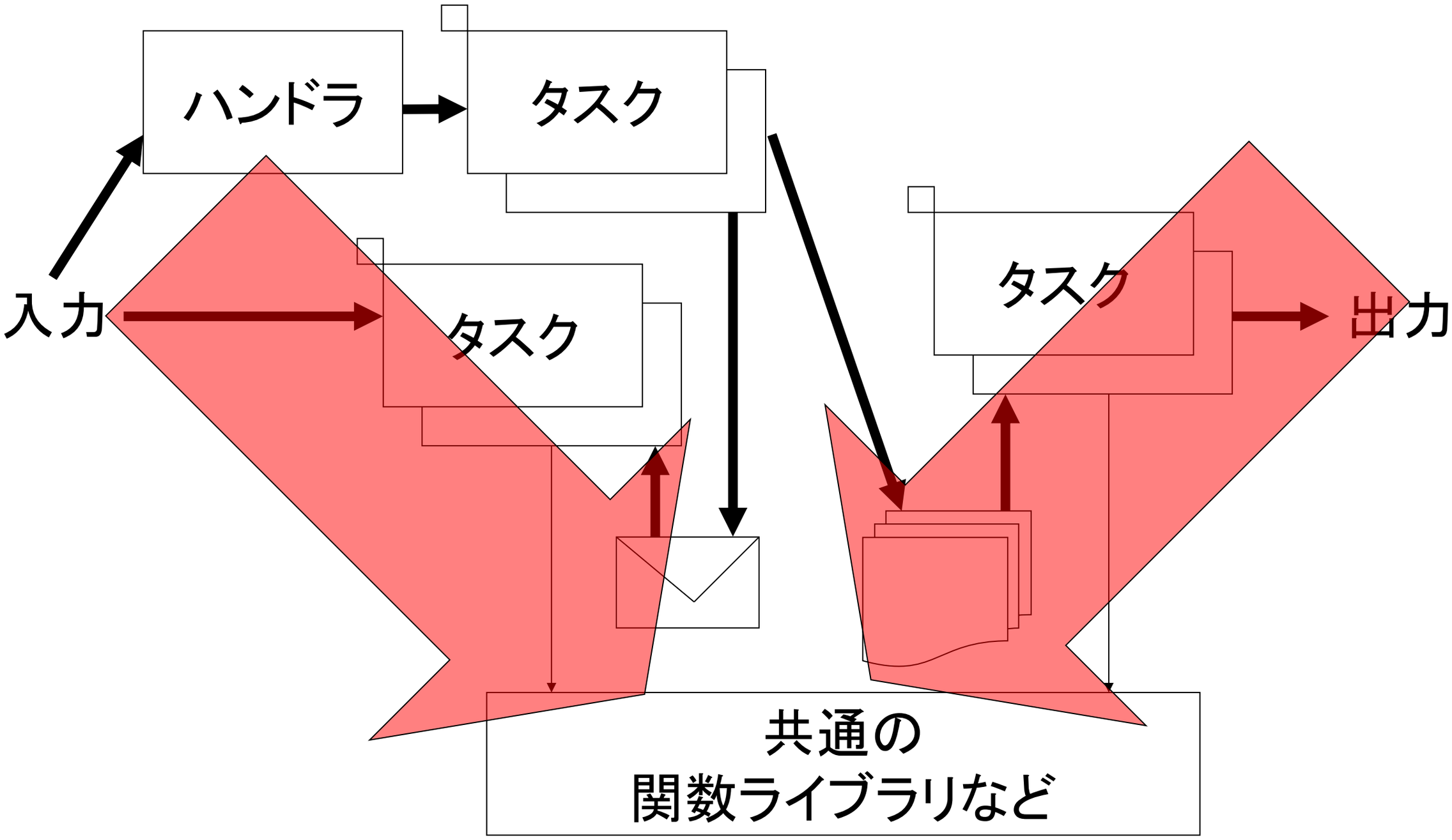
- **できるかぎり割込み処理の時間を短くすることにより、他の割込み処理も、効率よく動作する**
  - 最小限の必要な処理のみ行うことを心がける
- **割込み処理を短くすることで、優先度の高いタスクへの切替えも高速になる**
  - プログラムコードが短いことと、実際の処理時間が短いことは違う

より良いプログラムのために

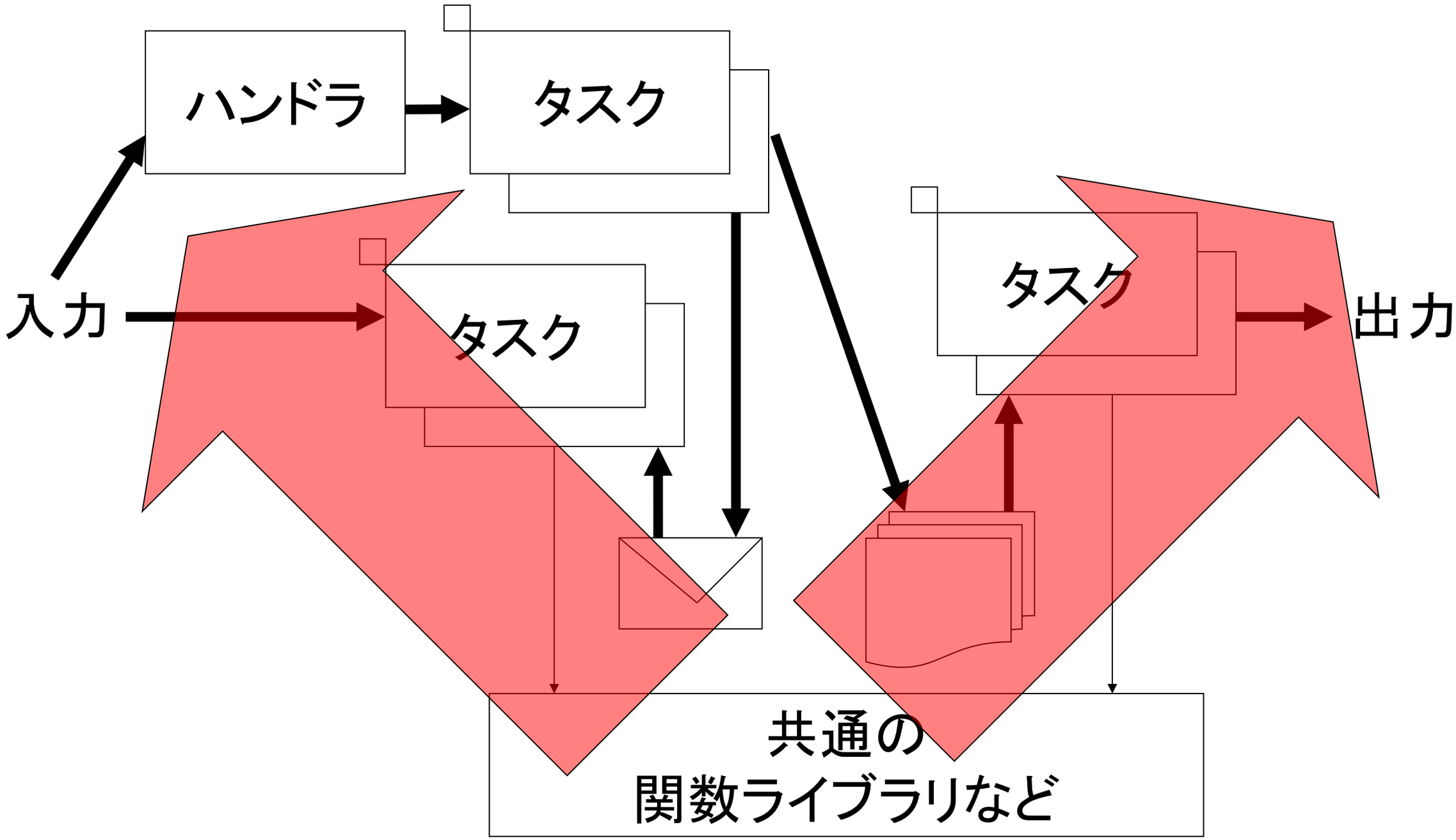
設計と実装の順序

- **設計はデータの入出力側から考える方が良い**(Top down)
  - 計算処理などの設計が先になると、どのようなデータを使用して計算し、その出力結果がどのように使用されるかわかりにくくなるため、見通しが悪くなる
- **実装は、共通処理や簡単な機能から用意していく**(Bottom up)
  - 間違いなく動作するものを増やしていく

設計の順序(Top down)



実装の順序(Bottom up)



機能の独立性を確保する

- **それぞれの機能(一つまたは複数のタスクで構成)が、できる限り独立して動作できるように考慮する**
  - なるべくグローバルデータは使用しない
  - 他のタスクの動作に影響しない・影響されないようにする
- **独立性が高いと、変更容易性や再利用性、保守性も向上する**

バランスが重要

- リアルタイム性能確保のための処理や高速な制御アルゴリズムのように、制御工学的技術が要求される処理と、リエントラントや変更容易性などが求められる処理は、結果が相反する場合も多い
- 組込みシステムではどちらも重要で、どちらを優先するか、バランスが必要になる

【実習】ITRON中級テキスト「 プログラミング演習(1)」

著者　TRON Forum