Open SDV Initiative ディスカッションメモ

ウィンドウ制御の実装例

作成者: 高田広章（名古屋大学）

最終更新: 2026年3月27日

このドキュメントは，Open SDV API仕様に基づいたウィンドウ制御を実現するシステムの実装例について検討するものである。Open SDV APIを，アプリケーションおよびビークルサービスを実行するビークルコンピュータと，それとCANで接続されたウィンドウ制御ECUで構成される分散システム構成で実現できることを示すことを主目的としている。CANをシグナルベースのネットワークと捉えると，サービスベースのAPIを，シグナルベースの通信路上で実現する方法を示すものと位置付けることができる。

この検討結果は，ウィンドウ以外の制御にも応用できるものと考えている。ただし，ウィンドウ制御はリアルタイム制約が緩やかであり，それを前提に設計しているため，車両の運動制御のように高いリアルタイム性が必要な制御には，そのままでは適用できない。

なお，このドキュメントは，Open SDV API仕様（バージョン: 202603α）に関する知識を前提に記述している。

目次

1. 前提と制限事項 . . . 2
   1.1. 制御対象のウィンドウに関する前提 . . . 2
   1.2. センサーとアクチュエータに関する前提 . . . 2
   1.3. 制限事項 . . . 3
2. 開閉スイッチによる制御の仕様 . . . 4
3. 1つのECUでの実装 . . . 5
   3.1. 内部状態 . . . 5
   3.2. 動作開始/停止条件の判定 . . . 6
   3.3. 動作ロジック . . . 7
4. ハードウェア構成と設計方針 . . . 15
   4.1. ハードウェア構成 . . . 15
   4.2. 用語の定義 . . . 16
   4.3. 要求と基本方針 . . . 16
   4.4. 開閉スイッチによる制御と無効化 . . . 17
   4.5. リスク制御の実現 . . . 17
   4.6. 移動コマンドのシーケンス番号 . . . 18
   4.7. ロックコマンドの調停 . . . 19
   4.8. 移動コマンドによる開閉スイッチの無効化解除 . . . 21
   4.9. 状態の更新とイベントの生成 . . . 22
   4.10. 異常状態での振る舞い . . . 22
   4.11. 制限事項 . . . 23
5. CANメッセージの構成 . . . 23
   5.1. CANメッセージ1の構成と送信タイミング . . . 23

5.2. enableSwitchOnSuccessの振る舞い . . . 24
5.3. CANメッセージ2の構成と送信タイミング . . . 25
5.4. 目標位置の扱い . . . 27
6. ウィンドウ制御ECUの動作ロジック . . . 27
6.1. 内部状態 . . . 27
6.2. 動作ロジック . . . 28
7. ビークルコンピュータの動作ロジック . . . 33
7.1. サービスコールからリターンするタイミング . . . 33
7.2. コマンドの伝え方とコマンドキュー . . . 34
7.3. 状態を変化させないサービスコール . . . 35
7.4. 現在状態の取り込み時のロックの強制解除 . . . 35
7.5. CANメッセージ2受信処理の流れ . . . 36
7.6. コマンドとコマンドキューの操作 . . . 37
7.7. 内部状態 . . . 38
7.8. 動作ロジック . . . 39
8. ビークルコンピュータの動作ロジックの最適化 . . . 49
8.1. 最適化の必要性 . . . 49
8.2. オーバライドされる移動コマンドのスキップ . . . 50
8.3. 開閉スイッチの無効化状態を変更しないロックコマンド . . . 51
8.4. ロックコマンドの一括処理 . . . 52
8.5. コマンドの同時/独立発行 . . . 53
8.6. 内部状態 . . . 55
8.7. 動作ロジック . . . 55
付録 A: バージョン履歴 . . . 65

1. 前提と制限事項

1.1. 制御対象のウィンドウに関する前提

制御対象のウィンドウは，Movable（APIにより移動できる）かつClosedDetectable（全閉状態を検知できる）であるが，HasMove2（第二次元方向に移動できる）でない，つまり一次元方向のみに移動できるものとする。また，挟み込み検知機能を持つものとするが，RiskPinch（挟み込みのリスク）とRiskOpen（開動作に伴う諸々のリスク）に対策できていないものとする。

なお，挟み込み検知機能を持っていれば，RiskPinchに対策できていることが一般的であるが，実装例に盛り込む要素を増やすために，不自然な前提を置いている。

1.2. センサーとアクチュエータに関する前提

ウィンドウの制御に用いるセンサーとしては，ウィンドウの現在位置を検知する位置センサーと挟み込みを検知するセンサーが，アクチュエータとしては，ウィンドウを開閉するためのモーターがあるものとする。また，ウィンドウの開閉スイッチがあるものとする。これらの入出力仕様は以下の通り。

開閉スイッチ

readSwitchStatus()によって，開閉スイッチの現在の状態を取得できるものとする。開閉スイッチの状態は，以下のいずれかで表す。

- NoAction（操作なし）
- ManualOpen（開1段階目）
- AutoOpen（開2段階目）
- ManualClose（閉1段階目）
- AutoClose（閉2段階目）
- Fault（故障中）

位置センサー

readPosition()によって，ウィンドウの現在の位置を取得できるものとする。ウィンドウの位置は，PositionTypeで表す。すなわち，0〜100の値で表す開閉度（0：全閉，100：全開）または位置不明（UNKNOWN）または全閉でない（NONZERO）のいずれかの値を取る。

開動作中に全開になった場合には100，閉動作中に全閉になった場合には0になる（言い換えると，両端は確実に検知できる）ものとする。

挟み込み検知センサー

挟み込みを検知すると，「挟み込み検知時の処理」が1回呼び出されるものとする。

モーター

controlMotor(movement)によって，モーターを制御できるものとする。movementには，以下のいずれかの指示を指定する。

- Stop（停止指示）
- Open（開動作指示）
- Close（閉動作指示）

開閉動作中に停止指示を出しても，開閉動作が停止するまでに時間がかかるものとする。停止指示を出した後，開閉動作が停止した時点で，「開閉停止時の処理」が1回呼び出されるものとする。

また，停止指示を出した後，停止するまでの移動量（パーセンテージ）をSTOP_OVERRUNと表記し，目標位置のSTOP_OVERRUN手前になったら，停止指示を出すものとする。

故障の検知と回復

センサーやアクチュエータなどに何らかの故障が検知された場合には「故障検知時の処理」が，故障から回復した場合には「故障からの回復時の処理」が，それぞれ1回呼び出されるものとする。

1.3. 制限事項

この実装例では，APIの実現ロジックの本質部分にフォーカスするため，次の処理は単純化している。

- アクチュエータ（モーター）の制御
- 故障時と挟み込み検知時の振る舞い

また，次の処理は省略している。

- マルチインスタンスの扱い
- 移動プロファイルの扱い
- サービスコールのパラメータが範囲内かのエラーチェック
- アプリケーションが使用できない操作優先度に関するエラーチェック
- サービスコールのpriorityパラメータの指定を省略した場合の処理
- 複数の処理が並行に実行される際に必要な排他制御
- システム起動直後のウィンドウ制御ECUとビークルコンピュータの同期

2. 開閉スイッチによる制御の仕様

ウィンドウの開閉スイッチによりウィンドウの開閉を行う制御を「開閉スイッチによる制御」と呼び，標準制御（OEMが自動車の出荷時に組み込んでいる制御ロジック）と位置付ける。開閉スイッチによる制御は，1つのアプリケーションと扱い，アプリケーションID（開閉スイッチによる制御のID）を割り付ける。また，開閉スイッチによる制御がウィンドウを操作する時の操作優先度を「開閉スイッチの操作優先度」と呼ぶ。開閉スイッチの操作優先度は，「一般ユーザ」の操作優先度とすることを想定している。

開閉スイッチによる制御の振る舞いを，擬似コードで示す。

- 移動に関するイベントの通知を要求しておく。
- autoOpenFlag/autoCloseFlagは，開閉スイッチが開2段階目/閉2段階目まで操作されたことを示す変数である。いずれもfalseに初期化しておく。
- ウィンドウが異常状態（故障中，挟み込み回避中）でなく，開閉スイッチが無効化されていない場合，開閉スイッチの状態を周期的に読み，それに応じて以下の処理を行う。

```
    switch (readSwitchStatus()) {
    case NoAction:
        if (ウィンドウの主状態がStoppedでない
                && 前回の開閉スイッチの状態がNoActionまたはFaultでない
                && autoOpenFlag == false && autoCloseFlag == false) {
            stopMove();
        }
        break;
    case ManualOpen:
        startMove(100);
        autoCloseFlag = false;
        break;
    case AutoOpen:
        startMove(100);
        autoCloseFlag = false;
        autoOpenFlag = true;
        break;
```

```
    case ManualClose:
        startMove(0);
        autoOpenFlag = false;
        break;
    case AutoClose:
        startMove(0);
        autoOpenFlag = false;
        autoCloseFlag = true;
        break;
    case Fault:
        if (ウィンドウの主状態がStoppedでない
                && 前回の開閉スイッチの状態がFaultでない) {
            stopMove();
            autoOpenFlag = false;
            autoCloseFlag = false;
        }
        break;
    }
```

- イベントを受け取ったら，それに応じて以下の処理を行う。

```
    switch (イベント種別) {
    case ControlledByOther:
    case TargetReached:
    case FaultDetected:
    case PinchDetected:
        autoOpenFlag = false;
        autoCloseFlag = false;
        break;
    }
```

3. 1つのECUでの実装

各サービスコールの振る舞いを明確にするために，ここでは，1つのECU（ここではビークルコンピュータと呼ぶ）で実装する場合の動作ロジックを示す。

ウィンドウを開閉するモーター，ウィンドウの現在位置を検知する位置センサー，挟み込みを検知するセンサー，ウィンドウの開閉スイッチは，すべてビークルコンピュータに接続されているものとする。

3.1. 内部状態

ビークルコンピュータは，Open SDV API仕様で規定されたウィンドウの状態を内部状態に持つ。getStatusは，この内部状態をそのまま返す（ただし，ロック状態の3つのフィールドは，以下の動作ロジックではフラットな内部状態と扱っているが，getStatusでは1つの構造体に入れて返す必要がある）。

mainStatus　　主状態

lockStatus	ロック状態	
	locked	ロックされているか否か
	lockApplication	ロックしたアプリケーションID
	lockPriority	ロックを取得した操作優先度
position	現在位置	
targetPosition	目標位置	

これに加えて，TargetReachedの関連情報を生成するために，以下の内部状態を持つ。

sourceApplication	最後に正常実行されたstartMove/stopMoveを呼び出したアプリケーション

また，開閉スイッチによる制御のために，以下の内部状態を持つ。

switchStatus	前回の開閉スイッチの状態
autoOpenFlag	開閉スイッチが開2段階目まで操作された
autoCloseFlag	開閉スイッチが閉2段階目まで操作された

開閉動作中にcontrolMotor(Stop)でモーターに停止指示を出した後，停止が完了するまでの間は，mainStatusをOpeningまたはClosingとする。開閉が停止した時点で呼び出される「開閉停止時の処理」で，mainStatusをStoppedに変化させる。

3.2. 動作開始/停止条件の判定

startMoveが呼び出された時に，開動作を開始すべきかを判定する関数としてcheckStartOpenを，閉動作を開始すべきかを判定する関数としてcheckStartCloseを用意する。

```
// 開動作開始条件のチェック
checkStartOpen(targetPosition)
{
    // 目標位置が100であれば，現在位置によらずに開動作を開始。
    // そうでない場合，現在位置がわかっており，それが目標位置よりも小
    // さい場合に開動作を開始。
    return targetPosition == 100
            || (内部状態.position != UNKNOWN
                && 内部状態.position != NONZERO
                && targetPosition > 内部状態.position);
}

// 閉動作開始条件のチェック
checkStartClose(targetPosition)
{
    // 目標位置が0であれば，現在位置によらずに閉動作を開始。
    // そうでない場合，現在位置がわかっており，それが目標位置よりも大
```

```
    // きい場合に閉動作を開始。
    return targetPosition == 0
            || (内部状態.position != UNKNOWN
                && 内部状態.position != NONZERO
                && targetPosition < 内部状態.position);
}
```

また，位置センサーから新しい現在位置を読み込んだ時に，開動作を停止すべきかを判定する関数としてcheckStopOpenを，閉動作を停止すべきかを判定する関数としてcheckStopCloseを用意する。

```
// 開動作停止条件のチェック
checkStopOpen(position)
{
    if (position != UNKNOWN && position != NONZERO) {
        // 現在位置がわかっている場合，それが目標位置からSTOP_OVERRUN
        // を減じた値以上であれば停止。
        return position >= 内部状態.targetPosition - STOP_OVERRUN;
    }
    else {
        // そうでない場合，目標位置が100でなければ停止。
        return 内部状態.targetPosition != 100;
    }
}

// 閉動作停止条件のチェック
checkStopClose(position)
{
    if (position != UNKNOWN && position != NONZERO) {
        // 現在位置がわかっている場合，それが目標位置にSTOP_OVERRUNを
        // 加えた値以下であれば停止。
        return position <= 内部状態.targetPosition + STOP_OVERRUN;
    }
    else {
        // そうでない場合，目標位置が0でなければ停止。
        return 内部状態.targetPosition != 0;
    }
}
```

3.3. 動作ロジック

以下では，startMove，stopMove，lock，unlockの各サービスコールの処理ルーチン，開閉スイッチの読み込み処理，位置センサーの読み込み処理，開閉停止時の処理，故障検知時の処理，故障からの回復時の処理，挟み込み検知時の処理の動作ロジックを示す。これらの処理は，並行に呼び出されることはないと想定している（並行して呼び出される場合には，排他制御の追加が必要である）。

また，これらから呼び出される関数として，revokeLock，targetPositionWhenStopped，

clearAutoFlags，checkMoveの動作ロジックを示す。

以下の動作ロジック中で，「〜〜イベント（〜〜）を生成」という記述の（）内は，イベントの送り先と関連情報を決めるためのパラメータを示す。

具体的には，ControlledBySelfとOwnLockRevokedは，パラメータで示されるアプリケーションのみに送る。逆に，ControlledByOtherとOtherLockReleasedは，パラメータで示されるアプリケーション以外に送る。OtherLockReleasedのパラメータがNULLの場合には，すべてのアプリケーションに送る。また，TargetReachedは，すべてのアプリケーションに送る。なお，開閉スイッチによる制御の処理は，以下の動作ロジック内で記述しているため，開閉スイッチによる制御にはイベントを送る必要がない。

また，ControlledByOtherとTargetReachedの関連情報のsourceApplicationには，パラメータで指定されたアプリケーションIDを設定する。

```
// ロックの強制解除
//
// OtherLockReleasedイベントの生成が必要な場合には，lockReleasedをtrue
// にする。
revokeLock(lockReleased)
{
    OwnLockRevokedイベント（内部状態.lockApplication）を生成
    if (lockReleased)
        OtherLockReleasedイベント（内部状態.lockApplication）を生成
    }
    内部状態.locked = false;
}

// 主状態がStoppedに遷移した時の目標位置
targetPositionWhenStopped(position)
{
    if (position != UNKNOWN && position != NONZERO) {
        return position;
    }
    else {
        return UNKNOWN;
    }
}

// autoOpenFlagとautoCloseFlagのクリア
clearAutoFlags()
{
    内部状態.autoOpenFlag = false;
    内部状態.autoCloseFlag = false;
}

// startMove/stopMoveのエラーチェック
//
// ロックの強制解除が必要な場合に，*p_lockRevokeをtrueにする。
checkMove(application, priority, p_lockRevoke)
```

```
{
    *p_lockRevoke = false;
    if (内部状態.locked && 内部状態.lockApplication != application) {
        // 他のアプリケーションがロックしている
        if (priority >= 内部状態.lockPriority) {
            return E_OBJECT_LOCKED;
        }
        else {
            // ロックの強制解除が必要
            *p_lockRevoke = true;
        }
    }
    switch (内部状態.mainStatus) {
    case Fault:
        return E_OBJECT_FAULT;
    case PinchAvoiding:
        return E_OBJECT_STATUS;
    }
    return E_OK;
}

// リスク制御によるエラーのチェック
//
// riskClassによって制御を拒否する必要があるか判定し，拒否する必要が
// ない場合にはE_OK，拒否する必要がある場合には，返すべきエラーコー
// ドを返す。
checkRisk(riskClass)
{
    if (車両構成情報の「対策できていないリスククラスの集合」が,
                                                    riskClassを含む) {
        if (アプリケーションのリスク制御情報の
                「対応しているリスククラスの集合」が，riskClassを含まない) {
            return E_RISK_APPLICATION;
        }
        else if (ユーザプリファレンスデータ中のアプリケーションに対して
                「受け入れているリスククラスの集合」が，riskClassを含まない) {
            return E_RISK_USER;
        }
    }
    return E_OK;
}

// startMoveサービスコール
startMove(position, priority)
{
    // retvalとlockRevokeはローカル変数
    retval = checkMove(呼び出したアプリケーションID, priority, &lockRevoke);
    if (retval != E_OK) {
        return retval;
    }
```

```
    if (checkStartOpen(position)) {
        // 開動作
        retval = checkRisk(RiskOpen);
        if (retval == E_OK) {
            if (lockRevoke) {
                revokeLock(true);
            }
            ControlledBySelfイベント（呼び出したアプリケーションID）を生成
            ControlledByOtherイベント（呼び出したアプリケーションID）を生成
            内部状態.mainStatus = Opening;
            内部状態.targetPosition = position;
            内部状態.sourceApplication = 呼び出したアプリケーションID;
            controlMotor(Open);
            clearAutoFlags();
        }
    }
    else if (checkStartClose(position)) {
        // 閉動作
        retval = checkRisk(RiskPinch);
        if (retval == E_OK) {
            if (lockRevoke) {
                revokeLock(true);
            }
            ControlledBySelfイベント（呼び出したアプリケーションID）を生成
            ControlledByOtherイベント（呼び出したアプリケーションID）を生成
            内部状態.mainStatus = Closing;
            内部状態.targetPosition = position;
            内部状態.sourceApplication = 呼び出したアプリケーションID;
            controlMotor(Close);
            clearAutoFlags();
        }
    }
    else {
        // すでに目標位置にある場合
        if (lockRevoke) {
            revokeLock(true);
        }
        ControlledBySelfイベント（呼び出したアプリケーションID）を生成
        ControlledByOtherイベント（呼び出したアプリケーションID）を生成
        内部状態.targetPosition = position;
        内部状態.sourceApplication = 呼び出したアプリケーションID;
        if (内部状態.mainStatus != Stopped) {
            controlMotor(Stop);
        }
        else {
            TargetReachedイベント（呼び出したアプリケーションID）を生成
        }
        clearAutoFlags();
    }
    return retval;
```

```
}

// stopMoveサービスコール
stopMove(priority)
{
    // retvalとlockRevokeはローカル変数
    retval = checkMove(呼び出したアプリケーションID, priority, &lockRevoke);
    if (retval != E_OK) {
        return retval;
    }
    // 開閉停止
    if (lockRevoke) {
        revokeLock(true);
    }
    ControlledBySelfイベント（呼び出したアプリケーションID）を生成
    ControlledByOtherイベント（呼び出したアプリケーションID）を生成
    内部状態.targetPosition = targetPositionWhenStopped(内部状態.position);
    内部状態.sourceApplication = 呼び出したアプリケーションID;
    if (内部状態.mainStatus != Stopped) {
        controlMotor(Stop);
    }
    else {
        TargetReachedイベント（呼び出したアプリケーションID）を生成
    }
    clearAutoFlags();
    return E_OK;
}

// lockサービスコール
lock(priority)
{
    switch (内部状態.mainStatus) {
    case Fault:
        return E_OBJECT_FAULT;
    case PinchAvoiding:
        return E_OBJECT_STATUS;
    }
    if (内部状態.locked) {
        if (priority >= 内部状態.lockPriority) {
            return E_OBJECT_LOCKED;
        }
        else if (内部状態.lockApplication != 呼び出したアプリケーションID) {
            // ロックを強制解除する
            OwnLockRevokedイベント（内部状態.lockApplication）を生成
        }
    }
    内部状態.locked = true;
    内部状態.lockApplication = 呼び出したアプリケーションID;
    内部状態.lockPriority = priority;
    return E_OK;
```

```
}

// unlockサービスコール
unlock()
{
    if (!(内部状態.locked)
            || 内部状態.lockApplication != 呼び出したアプリケーションID) {
        return E_OBJECT_STATUS;
    }
    else {
        OtherLockReleasedイベント（呼び出したアプリケーションID）を生成
        内部状態.locked = false;
    }
    return E_OK;
}

開閉スイッチの読み込み処理() // 周期処理
{
    // switchStatusはローカル変数
    switchStatus = readSwitchStatus();

    if (内部状態.mainStatus != Fault
            && 内部状態.mainStatus != PinchAvoiding
            && !(内部状態.locked
                    && 開閉スイッチの操作優先度 >= 内部状態.lockPriority) {
        switch (switchStatus) {
        case NoAction:
            if (内部状態.mainStatus != Stopped
                        && 内部状態.switchStatus != NoAction
                        && 内部状態.switchStatus != Fault
                        && !(内部状態.autoOpenFlag)
                        && !(内部状態.autoCloseFlag)) {
                if (内部状態.locked) {
                    revokeLock(true);
                }
                ControlledByOtherイベント（開閉スイッチによる制御のID）を生成
                内部状態.sourceApplication = 開閉スイッチによる制御のID;
                controlMotor(Stop);
            }
            break;
        case ManualOpen:
        case AutoOpen:
            if (内部状態.locked) {
                revokeLock(true);
            }
            ControlledByOtherイベント（開閉スイッチによる制御のID）を生成
            内部状態.mainStatus = Opening;
            内部状態.targetPosition = 100;
            内部状態.sourceApplication = 開閉スイッチによる制御のID;
            controlMotor(Open);
```

```
            内部状態.autoCloseFlag = false;
            if (switchStatus == AutoOpen) {
                内部状態.autoOpenFlag = true;
            }
            break;
        case ManualClose:
        case AutoClose:
            if (内部状態.locked) {
                revokeLock(true);
            }
            ControlledByOtherイベント（開閉スイッチによる制御のID）を生成
            内部状態.mainStatus = Closing;
            内部状態.targetPosition = 0;
            内部状態.sourceApplication = 開閉スイッチによる制御のID;
            controlMotor(Close);
            内部状態.autoOpenFlag = false;
            if (switchStatus == AutoClose) {
                内部状態.autoCloseFlag = true;
            }
            break;
        case Fault:
            if (内部状態.mainStatus != Stopped
                        && 内部状態.switchStatus != Fault) {
                if (内部状態.locked) {
                    revokeLock(true);
                }
                ControlledByOtherイベント（開閉スイッチによる制御のID）を生成
                内部状態.sourceApplication = 開閉スイッチによる制御のID;
                controlMotor(Stop);
                clearAutoFlags();
            }
            break;
        }
    }
    else {
        // 異常状態（故障中，挟み込み回避中）と，開閉スイッチが無効化
        // されている場合には，何もしない
    }
    内部状態.switchStatus = switchStatus;
}

位置センサーの読み込み処理() // 周期処理
{
    // positionはローカル変数
    position = readPosition();
    switch (内部状態.mainStatus) {
    case Stopped:
        if (position != 内部状態.position
                    && 内部状態.position != UNKNOWN
                    && 内部状態.position != NONZERO) {
```

```
            // 停止中に現在位置が動いた場合は，一瞬故障したものと扱う
            if (内部状態.locked) {
                revokeLock(false);
            }
            FaultDetectedイベントを生成
            clearAutoFlags();
            OtherLockReleasedイベント（NULL）を生成
            FaultRecoveredイベントを生成
            内部状態.targetPosition = targetPositionWhenStopped(position);
        }
        break;
    case Opening:
        if (checkStopOpen(position)) {
            // 開動作中に目標位置に到達
            controlMotor(Stop);
        }
        break;
    case Closing:
        if (checkStopClose(position)) {
            // 閉動作中に目標位置に到達
            controlMotor(Stop);
        }
        break;
    case Fault:
        // 何もしない
        break;
    case PinchAvoiding:
        if (position != UNKNOWN && position != NONZERO
                    && position >= 内部状態.targetPosition - STOP_OVERRUN) {
            // 挟み込み回避中に目標位置に到達
            controlMotor(Stop);
        }
        break;
    }
    内部状態.position = position;
}

開閉停止時の処理()
{
    switch (内部状態.mainStatus) {
    case Opening:
    case Closing:
        TargetReachedイベント（内部状態.sourceApplication）を生成
        内部状態.mainStatus = Stopped;
        内部状態.targetPosition = targetPositionWhenStopped(内部状態.position);
        clearAutoFlags();
        break;
    case PinchAvoiding:
        OtherLockReleasedイベント（NULL）を生成
        PinchRecoveredイベントを生成
```

```
        内部状態.mainStatus = Stopped;
        内部状態.targetPosition = targetPositionWhenStopped(内部状態.position);
        break;
    }
}

故障検知時の処理()
{
    if (内部状態.locked) {
        revokeLock(false);
    }
    FaultDetectedイベントを生成
    内部状態.mainStatus = Fault;
    内部状態.targetPosition = UNKNOWN;
    clearAutoFlags();
    controlMotor(Stop);             // 可能なら停止させる
}

故障からの回復時の処理()
{
    OtherLockReleasedイベント（NULL）を生成
    FaultRecoveredイベントを生成
    内部状態.mainStatus = Stopped;
    内部状態.targetPosition = targetPositionWhenStopped(内部状態.position);
    controlMotor(Stop);             // 念のため停止させる
}

挟み込み検知時の処理()
{
    if (内部状態.locked) {
        revokeLock(false);
    }
    PinchDetectedイベントを生成
    内部状態.mainStatus = PinchAvoiding;
    内部状態.targetPosition = 挟み込み回避のための開動作の目標位置;
    clearAutoFlags();
    controlMotor(Open);
}
```

4. ハードウェア構成と設計方針

以下では，ウィンドウ制御を，ビークルコンピュータとウィンドウ制御ECUで構成される分散システム構成で実装する場合の設計について検討する。

4.1. ハードウェア構成

アプリケーションおよびビークルサービスを実行するビークルコンピュータと，ウィンドウを制御するウィンドウ制御ECUがあり，それらはCANで接続されているものとする（図　1）。

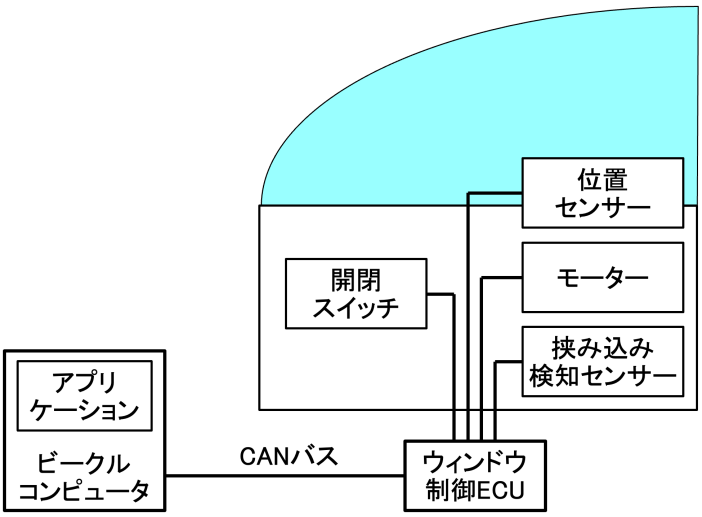

図 1. ウィンドウ制御のハードウェア構成

ウィンドウを開閉するモーター，ウィンドウの現在位置を検知する位置センサー，挟み込みを検知するセンサー，ウィンドウの開閉スイッチ（ウィンドウと同じドアに設置されているものを想定。運転席横のドアに設置されているものではない）は，ウィンドウ制御ECUに接続されているものとする。

CANによる通信では，繰り返し，送信時点の状態がメッセージに入れて送信される。状態に変化がない場合には，同じメッセージが再送される。CANによる通信では，メッセージの遅延や欠落が起こると想定する。

4.2. 用語の定義

コマンド，移動コマンド，ロックコマンド

startMove/stopMoveサービスコールによる移動に関する指令を移動コマンド，lock/unlockサービスコールによるロックに関する指令をロックコマンド，両者を総称してコマンドと呼ぶ。

コマンドの発行

ビークルコンピュータからウィンドウ制御ECUにコマンドを伝え始めることを，コマンドの発行と呼ぶ。

異常状態

ウィンドウが故障中と挟み込み回避中の状態を総称して，異常状態と呼ぶ。

4.3. 要求と基本方針

開閉スイッチと挟み込み検知センサーは，ウィンドウ制御ECUに直接接続されているため，ビークルコンピュータがダウンしたり，CANで通信できなくなった場合でも，開閉スイッチによるウィンドウの開閉や挟み込み防止はできるようにする。

この要求を満たせる範囲で，なるべく多くの処理をビークルコンピュータ側で行い，ウィンドウ制御ECU側の役割を最小限にする。

ビークルコンピュータからウィンドウ制御ECUには，アプリケーションからのコマンドを伝える。CANメッセージは，状態を伝えるという使い方が一般的であるため，コマンドを送るという考え方ではなく，最新のコマンドが指示するウィンドウの目標位置を伝えるという考

え方をする。一方，ウィンドウ制御ECUからビークルコンピュータには，ウィンドウ制御の現在の状態と，最後に処理したコマンドの処理結果を伝える。

ウィンドウ制御ECUからビークルコンピュータには，すべての状態遷移を伝えるわけではないため，状態の変化が連続して発生した場合や，CANメッセージが欠落した場合には，ウィンドウ制御ECUにおける状態遷移がビークルコンピュータに伝わらない場合がある。このような状況が起こっても，Open SDV API仕様で規定されたイベントの抜けに関する仕様に合致していれば，問題ないものとする。

4.4. 開閉スイッチによる制御と無効化

開閉スイッチによる制御は，ウィンドウ制御ECU上に実装する（図 2）。

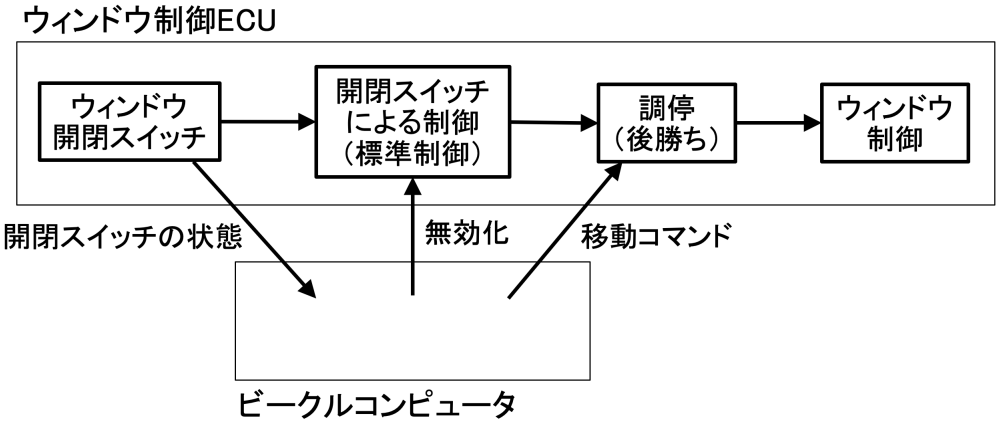

図 2. 開閉スイッチによる制御

ビークルコンピュータからの移動コマンドと，開閉スイッチによる制御からの移動コマンドは，ウィンドウ制御ECU内で，後勝ちの原則に従って調停する。開閉スイッチによる制御はロックを取得/解除しないが，移動コマンドの実行や異常状態への遷移によりロックが強制解除される場合があるため，ロックコマンドの調停も必要である。ロックコマンドの調停方法については， 4.7節で検討する。

開閉スイッチの操作優先度と同じかそれより高い操作優先度でウィンドウがロックされている間，開閉スイッチによる制御を無効化する必要がある。これを実現するために，ビークルコンピュータからウィンドウ制御ECUに，開閉スイッチの無効化を指示する機能を設ける。開閉スイッチを無効化またはそれを解除する指示を，開閉スイッチ無効化制御コマンドと呼ぶ。また，ウィンドウ制御ECUからビークルコンピュータに，無効化の指示が実行されたことを伝える機能を設ける。

開閉スイッチを汎用スイッチ（アプリケーションが任意の目的に使用できるスイッチ）として使用する場合のために，ウィンドウ制御ECUからビークルコンピュータに，開閉スイッチの状態を伝える。開閉スイッチの無効化の指示は，ウィンドウ開閉スイッチを汎用スイッチとして使用する場合にも用いることができる。

4.5. リスク制御の実現

ウィンドウに対するリスク制御は，RiskOpenはウィンドウの開動作を開始する時，RiskPinchは閉動作を開始する時に行う必要がある。

ビークルコンピュータが保持している現在位置は，通信遅延があるために，最新の値であるとは限らない。そのため，ウィンドウが動作開始する時に，開動作するか閉動作するかは

ウィンドウ制御ECU側でしかわからず，リスク制御の判定もウィンドウ制御ECU側で行う必要がある。

そこで，ビークルコンピュータからウィンドウ制御ECUに伝える移動コマンドに，リスク制御が必要かどうかの情報を付加する。また，ウィンドウ制御ECUからビークルコンピュータに伝える移動コマンドの処理結果に，リスク制御で移動コマンドを拒否したかどうかの情報を含める。

4.6. 移動コマンドのシーケンス番号

ビークルコンピュータからウィンドウ制御ECUには，アプリケーションからの最新の移動コマンドを伝えるが，この際に，移動コマンドのシーケンス番号を付けて伝える。

シーケンス番号は，ウィンドウ制御ECUが移動コマンドを処理したことをビークルコンピュータが知るためと，ウィンドウ制御ECUが移動コマンドの調停を行うために必要である。例えば，シーケンス番号を設けない場合，図　3のような状況で，どちらのstartMoveに対する目標位置に到達したのかを，ビークルコンピュータ側で区別することができない。また，図 4のような状況で，ウィンドウ制御ECUは，2回目に届いた目標位置50のコマンドが，最初のstartMoveに対する目標位置が再送されたもの（赤で記載）か，新たなstartMoveの呼び出しによるもの（青で記載）か，区別できない。後勝ちの原則に従うと，前者であれば2回目の目標位置50のコマンドは無視すべきであるが，後者であれば無視してはならない。

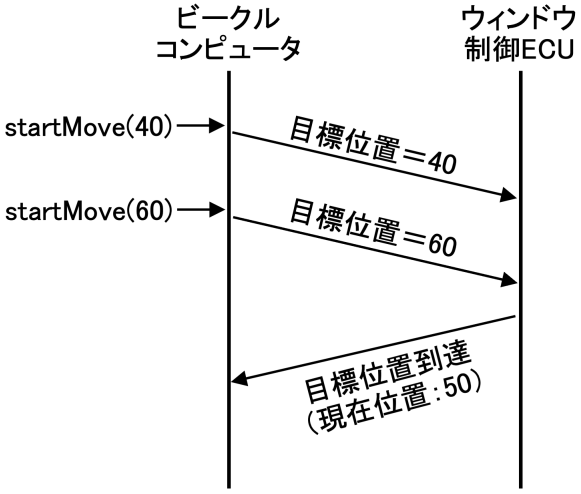

図 3. シーケンス番号の必要性(1)

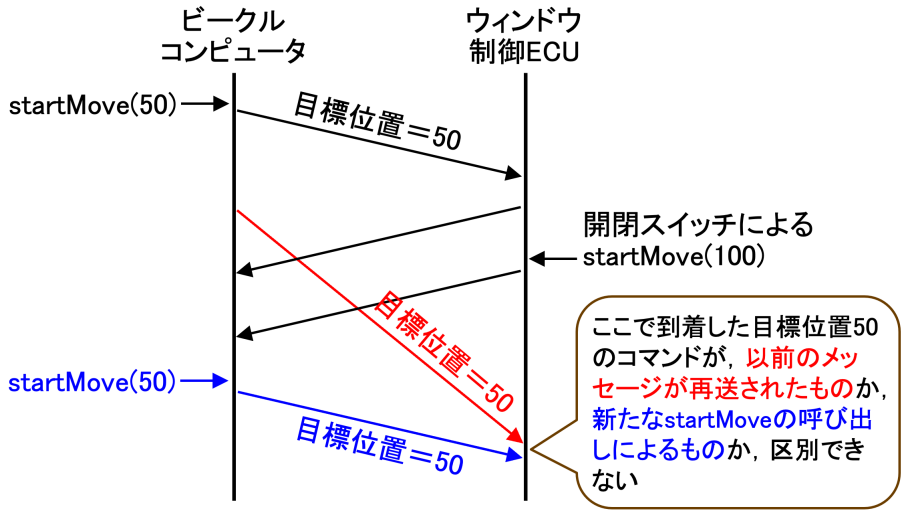

図 4. シーケンス番号の必要性(2)

ビークルコンピュータからウィンドウ制御ECUに送る移動コマンドに，シーケンス番号を付けて伝えるのに対して，ウィンドウ制御ECUからビークルコンピュータには，処理した移動コマンドのシーケンス番号を伝える。

シーケンス番号は，この実装例では1ビットとしている。そのため，ある移動コマンドが処理されるまで，次の移動コマンドを伝えることができず，発行を待たせる必要がある。具体的には，最後に発行した移動コマンドと処理した移動コマンドのシーケンス番号が一致している場合，最後に発行した移動コマンドは処理されており，次の移動コマンドを発行できる。一致していない場合，最後に発行した移動コマンドはまだ処理されておらず，次の移動コマンドを発行できない。

【補足説明】

複数の移動コマンドを同時に発行できないという制限が厳しい場合には，シーケンス番号を複数ビットとする拡張も可能であるが，この実装例の状況では，以下の理由で効果が限定的である。

シーケンス番号の複数ビットへの拡張が有効なのは，複数の移動コマンドの実行が要求された場合に，後勝ちの原則により，最後の移動コマンド以外の実行をスキップできるためである。逆に，途中の移動コマンドをスキップできない場合は，複数の移動コマンドを発行することができず，シーケンス番号を拡張しても意味がない。

この実装例の状況では，リスク制御により，発行した移動コマンドがエラーになる可能性がある。エラーになった場合，先に発行された移動コマンドをスキップすることができない。そのため，シーケンス番号を拡張しても，エラーになる可能性がある移動コマンドは，発行済みのコマンドがある場合には発行できないという制限がある。

途中の移動コマンドをスキップできない場合に，複数の移動コマンドを同時に発行するためには，1つのCANメッセージで複数の移動コマンドを伝えられるようにする必要がある。

4.7. ロックコマンドの調停

ウィンドウのロックは，lockにより取得，unlockにより解除されることに加えて，startMove/stopMoveの実行と異常状態への遷移によって強制解除される。これらの中で，異常状態への遷移については，故障と挟み込みを検知するのはウィンドウ制御ECUであるため，ビークルコンピュータの動作とは無関係に発生する。そのため，ビークルコンピュータ

とウィンドウ制御ECUの間で調停が必要である。

4.4節で述べた通り，ビークルコンピュータからウィンドウ制御ECUに開閉スイッチの無効化を指示する機能と，ウィンドウ制御ECUからビークルコンピュータに指示が実行されたことを伝える機能が必要である。これを実現する最もシンプルな方法として，開閉スイッチの無効化指示と，指示を実行した後の無効化状態を，それぞれ1ビットで伝える方法が考えられる。しかし，この方法には，図　5のような状況で，ウィンドウ制御ECUは，2回目に届いた「無効化指示＝1」のメッセージを無視すべきか否かを決定できないという問題がある。これは，図　4における移動コマンドと同様の状況であり，ウィンドウ制御ECUがロックコマンドの調停を行うことができない。

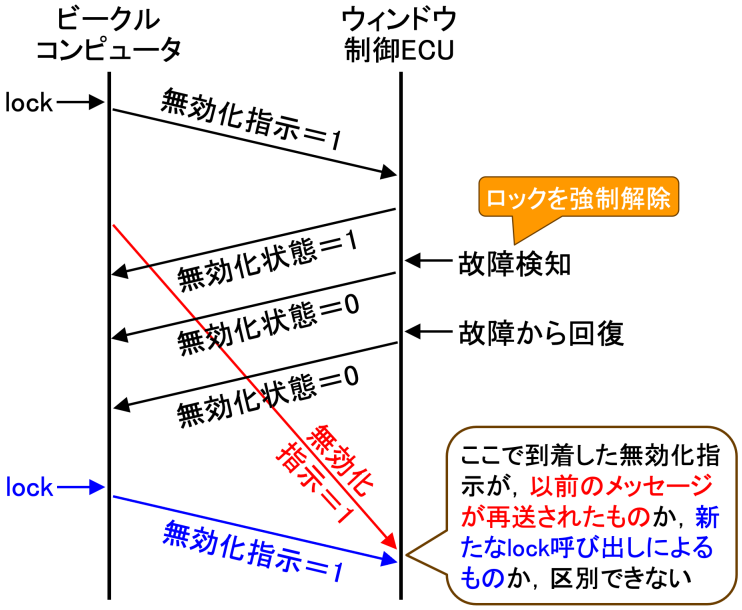

図 5. ロックコマンドの調停の必要性

これを解決するアプローチとして，次の2つ（細かく分けると3つ）が考えられる。

1. 開閉スイッチ無効化制御コマンドにも，シーケンス番号を付加する。さらに，この実現方法に，2つのアプローチがある。
   a. 開閉スイッチ無効化制御コマンドを移動コマンドと同様に扱い，1つのシーケンス番号で両方のコマンドを管理する。
   b. 移動コマンドのシーケンス番号とは別に，開閉スイッチ無効化制御コマンドに対するシーケンス番号を設ける。
2. 開閉スイッチの無効化に関する制御を，ビークルコンピュータのみで行う。

   図　5では，ウィンドウ制御ECUは，故障を検知すると，開閉スイッチの無効化状態を0（無効化を解除した状態）にしている。これは，ウィンドウ制御ECUが，ビークルコンピュータの動作とは無関係に，開閉スイッチの無効化を制御していることになる。これをやめて（つまり，ウィンドウ制御ECUは，故障を検知しても，開閉スイッチの無効化状態を変更しない），開閉スイッチの無効化に関する制御を，ビークルコンピュータのみで行うことで，この問題を解決できる。具体的には，ビークルコンピュータが，故障検知を伝えられた時点でロックを強制解除し，ウィンドウ制御ECUに開閉スイッチの無効化解除を指示する。

   このアプローチには，ウィンドウ制御ECUにおける開閉スイッチの無効化解除に，CANメッセージ1往復分の時間がかかる場合があるという欠点がある。ただし，異常状態はあ

る程度の時間継続するのが通常で，その間はいずれにしても開閉スイッチは働かないため，ほとんど問題にならないと考えられる。

アプローチ1は，状態を伝えるというCANメッセージの使い方からの逸脱を大きくすることになるため，この実装例では，アプローチ2で検討を進める。

4.8. 移動コマンドによる開閉スイッチの無効化解除

次に，startMove/stopMoveによるロックの強制解除の実現方法について検討する。具体的には，ウィンドウがロックされている時に，ロックを取得しているアプリケーション以外のアプリケーションにより，ロックを取得したよりも高い操作優先度で移動コマンドが実行されると，ロックは強制的に解除される。ロックにより開閉スイッチが無効化されていた場合には，ロックの強制解除により，開閉スイッチの無効化も解除する必要がある。

まず，開閉スイッチによる制御からの移動コマンドに関しては，ロックの強制解除は必要であるが，それにより，開閉スイッチの無効化の解除が必要になることはない。なぜなら，開閉スイッチによる制御が移動コマンドを出すのは，開閉スイッチが無効化されていない場合に限られるためである。

次に，ビークルコンピュータ上で呼び出されたstartMove/stopMoveによる移動コマンドに関して検討する。

ビークルコンピュータからの移動コマンドでロックが強制解除されるのは，移動コマンドが正常実行された場合のみであるが，startMoveが正常実行されるかエラーになるかは，リスク制御の判定に依存するため，ウィンドウ制御ECU側でしかわからない（stopMoveがリスク制御でエラーになることはない）。そのため，開閉スイッチの無効化を解除するかどうかも，ウィンドウ制御ECU側で決定する必要がある。ただし，異常状態への遷移と異なり，ビークルコンピュータの動作と無関係に発生するわけではない。

これを実現するアプローチとして，次の2つが考えられる。

1. ビークルコンピュータからウィンドウ制御ECUに，移動コマンドが正常実行された場合に開閉スイッチの無効化を解除することを指示する機能を設ける。
2. ビークルコンピュータは，移動コマンドが正常実行されたという処理結果を受け取った後に，開閉スイッチの無効化解除をウィンドウ制御ECUに指示する。

   このアプローチには，ウィンドウ制御ECUにおける開閉スイッチの無効化解除に，CANメッセージ1往復分の時間がかかるという欠点がある。また，CAN通信に故障が発生した場合に，サービスコールの一部の処理が実行できずに残るという気持ち悪さがあるが，いずれもウィンドウ制御では許容できると考えられる。

2つのアプローチは一長一短であるが，この実装例では，アプローチ1で検討を進める。

【補足説明】

アプローチ2は，CANメッセージ1のenableSwitchOnSuccessフィールドを削除し，ウィンドウ制御ECUとビークルコンピュータの動作ロジック中のそのフィールドを操作する部分を約20行削除することで実現できる。

4.9. 状態の更新とイベントの生成

ウィンドウ制御ECUからウィンドウ制御の現在の状態が伝えられると，ビークルコンピュータは，それを内部状態に取り込む。この時，イベントの生成と（必要な場合には）ロックの強制解除を行う。

ビークルコンピュータが必要なイベントを生成できるように，ウィンドウ制御ECUからビークルコンピュータには，現在のウィンドウの主状態に加えて，その主状態に遷移した理由に関する情報を伝える。具体的には，主状態の停止中を，遷移した理由により3つに分類する形で拡張した拡張状態と，最後に正常実行した移動コマンドを出したのがビークルコンピュータか開閉スイッチか（これをコマンド源と呼ぶ）を区別する情報を伝える。

ウィンドウ制御の状態変化が連続して発生すると，最後の状態のみが伝えられ，途中の状態が伝わらない場合がある。そのため，途中のイベントを発生させることができないが，MovableObjectの仕様では，連続して発生した移動に関するイベントの中で，最後のイベント以外は抜けても良いとしているため，問題ない。

ただし，状態変化が連続発生して最初の状態に戻り，最初の状態のみがビークルコンピュータに伝えられた場合，ビークルコンピュータは状態変化が発生したことを知ることができず，イベントを生成することができない。これは，最後の状態からイベントを再現する手法の本質的な限界である。ウィンドウ制御ECUからビークルコンピュータに状態遷移の回数を伝えるといった問題軽減策が考えられるが，回数を有限ビットで表現している限りは，完全に解決することはできない。このようなことが起こるのは，状態が一瞬だけ変化してすぐに元に戻った場合であり，そのような場合にイベントが抜けることは大きい問題ではないと考えられるため，許容することとする。

なお，拡張情報とコマンド源の情報を用いてロックの強制解除を行う方法は，7.4節で検討する。

4.10. 異常状態での振る舞い

異常状態でサービスコール（startMove，stopMove，lock）が呼ばれた場合，E_OBJECT_FAULT/E_OBJECT_STATUSを返すという仕様になっている。

ビークルコンピュータ側で，ウィンドウが異常状態であることがわかっている場合には，このエラー検出はビークルコンピュータ側で行えば良い。

問題は，ウィンドウ制御ECUが異常状態になったが，通信遅延によりそれがビークルコンピュータに伝わる前にこれらのサービスコールが呼び出されると，ビークルコンピュータ側でエラーを検出できず，ウィンドウ制御ECUにそのコマンドが伝えられてしまう状況である。

この状況に対応する方法の1つは，ウィンドウ制御ECUからビークルコンピュータに伝える移動コマンドの処理結果に，異常状態であるためにコマンドを実行しなかったという情報を含める方法である（リスク制御で移動コマンドを拒否したことを伝えるのと同じ方法）。

この実装例では，この方法ではなく，よりシンプルな方法をとっている。具体的には，ウィンドウ制御ECUは，異常状態の間に伝えられた移動コマンドを単に無視し，ビークルコンピュータには，移動コマンドを処理したことと，現在の状態が異常状態であることを伝える。これを受けたビークルコンピュータは，次のいずれの振る舞いをしても良い。

1. サービスコールを呼び出したアプリケーションにControlledBySelfとFaultDetected/PinchDetectedをこの順序で送り，サービスコールから正常リターンする。この振る舞いは，異常状態になる前にサービスコールが実行されたものと考えれば，仕様に合致している。
2. サービスコールを呼び出したアプリケーションにFaultDetected/PinchDetectedを送り，サービスコールからエラーリターンする。この振る舞いは，異常状態になった後にサービスコールが実行されたものと考えれば，仕様に合致している。

以降で示す動作ロジックは，1の振る舞いをするように実装してある。

4.11. 制限事項

以下の動作ロジックでは，ビークルコンピュータやウィンドウ制御ECUのダウン時や，CAN通信の故障時の例外処理は省略している。

ビークルコンピュータが開閉スイッチの無効化を指示したままダウンした場合，ウィンドウの開閉スイッチが効かなくなるという問題がある。これに対しては，タイムアウトを用いてビークルコンピュータのダウンを検出し，開閉スイッチの無効化を解除すべきである。CAN通信の故障時も同様である。

逆にビークルコンピュータ側では，ウィンドウ制御ECUがコマンドを処理した後にサービスコールからリターンするため，ウィンドウ制御ECUがダウンした場合やCAN通信の故障時には，サービスコールからリターンしなくなるという問題がある。これに対しても，タイムアウト等を用いて，サービスコールからエラーリターンさせる必要がある。

また，CANメッセージが欠落した場合に，サービスコールからのリターンが遅くなるという問題がある。CANメッセージの送信周期が長い場合には，タイムアウトを用いて，CANメッセージを再送するなどの対策を取ることが考えられる。

5. CANメッセージの構成

ビークルコンピュータからウィンドウ制御ECUへのCANメッセージをCANメッセージ1，ウィンドウ制御ECUからビークルコンピュータへのCANメッセージをCANメッセージ2と呼ぶ。いずれのCANメッセージも，周期送信に加えて，メッセージの内容に変化があった場合（一部例外があるため，厳密な規定は後述する）にも送信する。

CANメッセージ1とCANメッセージ2は，ウィンドウのインスタンス毎に，異なるCAN IDを用いて送信する。

5.1. CANメッセージ1の構成と送信タイミング

CANメッセージ1は，移動コマンドを伝えるフィールド（targetPosition，sequenceNum，riskPinch，riskOpen，enableSwitchOnSuccess）と，開閉スイッチの無効化の指示を伝えるフィールド（disableSwitch）で構成する。

targetPosition（最大7ビット）

ビークルコンピュータからウィンドウ制御ECUに対する最新の移動コマンドが指示するウィンドウの目標位置（0〜100）。停止指示の場合は，COMMAND_STOP（数値では127）とする。

必要なビット数を減らすために，位置の粒度を大きくしても良い。例えば，位置を10%単

位で扱うこととして，4ビットで表現しても良い。

sequenceNum（1ビット）
targetPositionで伝えている移動コマンドのシーケンス番号。新しい移動コマンドを発行する毎に，0と1を反転させる。

riskPinch（1ビット）
targetPositionで伝えている移動コマンドが，RiskPinchに対応できている場合に0，対応できていない場合に1とする。このフィールドが1の場合，ウィンドウの閉動作を行う移動コマンドは拒否される。

riskOpen（1ビット）
targetPositionで伝えている移動コマンドが，RiskOpenに対応できている場合に0，対応できていない場合に1とする。このフィールドが1の場合，ウィンドウの開動作を行う移動コマンドは拒否される。

enableSwitchOnSuccess（1ビット）
targetPositionで伝えている移動コマンドが正常実行されたら，開閉スイッチの無効化を解除する場合に，1とする。

disableSwitch（1ビット）
開閉スイッチの無効化指示。無効化を指示する場合に1とする。

1つのCANメッセージ1内の移動コマンドと開閉スイッチの無効化の指示は，アトミックに（不可分に）処理するものとする。片方だけを処理した状態のCANメッセージ2を送信してはならない。

CANメッセージ1は，周期送信に加えて，メッセージの内容に変化があった場合にも送信する。

5.2. enableSwitchOnSuccessの振る舞い

enableSwitchOnSuccessは，disableSwitchに優先する。

具体的には，enableSwitchOnSuccessが1の時，ウィンドウ制御ECUは，targetPositionで伝えられている移動コマンドを正常実行した場合は，開閉スイッチの無効化を解除し，disableSwitchは参照しない。その他の場合は，disableSwitchに従って，開閉スイッチを無効化/解除する。

ウィンドウ制御ECUが，CANメッセージ1の受信時に，開閉スイッチの無効化状態をどのように更新するかを以下の表に示す。

enableSwitch OnSuccess	シーケンス番号の更新	移動コマンドの成否	disableSwitch	更新後の開閉スイッチの無効化状態
0	—	—	0	解除
0	—	—	1	無効化
1	あり	正常	—	解除
1	あり	エラー	0	解除
1	あり	エラー	1	無効化
1	なし	正常	—	解除
1	なし	エラー	0	解除
1	なし	エラー	1	無効化

ここで，シーケンス番号が更新されなかった場合（新しい移動コマンドが発行されていない場合）にも，移動コマンドの実行の成否により処理が変わる（表の6〜8行目）のは，図　6のように，enableSwitchOnSuccessが1の移動コマンドが再送されてきた場合に，前回と同じ処理結果を返す必要があるためである。そのため，ウィンドウ制御ECUは，最後に処理した移動コマンドの実行の成否（または，最後に処理した移動コマンドの処理後の開閉スイッチの無効化状態）を記憶しておく必要がある。

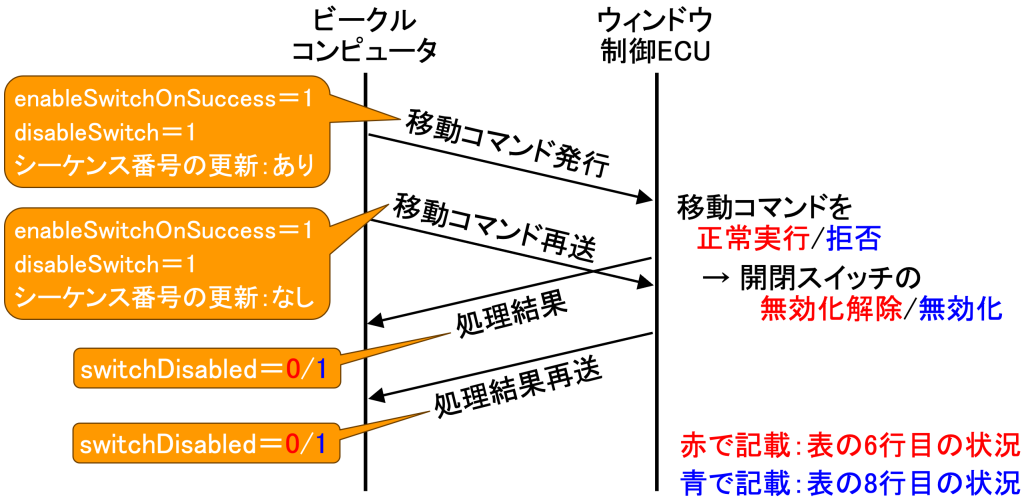

図 6. 移動コマンドの実行の成否の記憶

enableSwitchOnSuccessはdisableSwitchに優先するため，ビークルコンピュータは，enableSwitchOnSuccessを1にした移動コマンドが処理されたら，その後のdisableSwitchによる開閉スイッチの無効化に関する指示が受け付けられるように，enableSwitchOnSuccessを0に戻す必要がある。

5.3. CANメッセージ2の構成と送信タイミング

CANメッセージ2は，ウィンドウ制御の現在の状態を伝えるフィールド（extendedStatus，commandSource，position，switchDisabled），最後に処理したビークルコンピュータからの移動コマンドの処理結果を伝えるフィールド（sequenceNum，denyRiskPinch，denyRiskOpen），開閉スイッチの現在状態を伝えるフィールド（switchStatus）で構成する。

ウィンドウ制御の現在の状態を伝えるフィールドは，最後に処理した移動コマンドを実行し

た直後の状態とは限らない。最後の移動コマンドを処理した後に，開閉スイッチの操作，故障の検知/回復，挟み込み検知/回復により，現在の状態は変化している場合がある。

extendedStatus（3ビット）

ウィンドウの主状態のStopped（停止中）を，遷移した理由により3つに分類する形で拡張したウィンドウの拡張状態。以下のいずれかで表す。

- TargetReached（目標位置に到達して停止中）
- FaultRecovered（故障から回復して停止中）
- PinchRecovered（挟み込みから回復して停止中）
- Opening（開動作中）
- Closing（閉動作中）
- Fault（故障中）
- PinchAvoiding（挟み込み回避中）

図 7に，ウィンドウの拡張状態の状態遷移を示す。

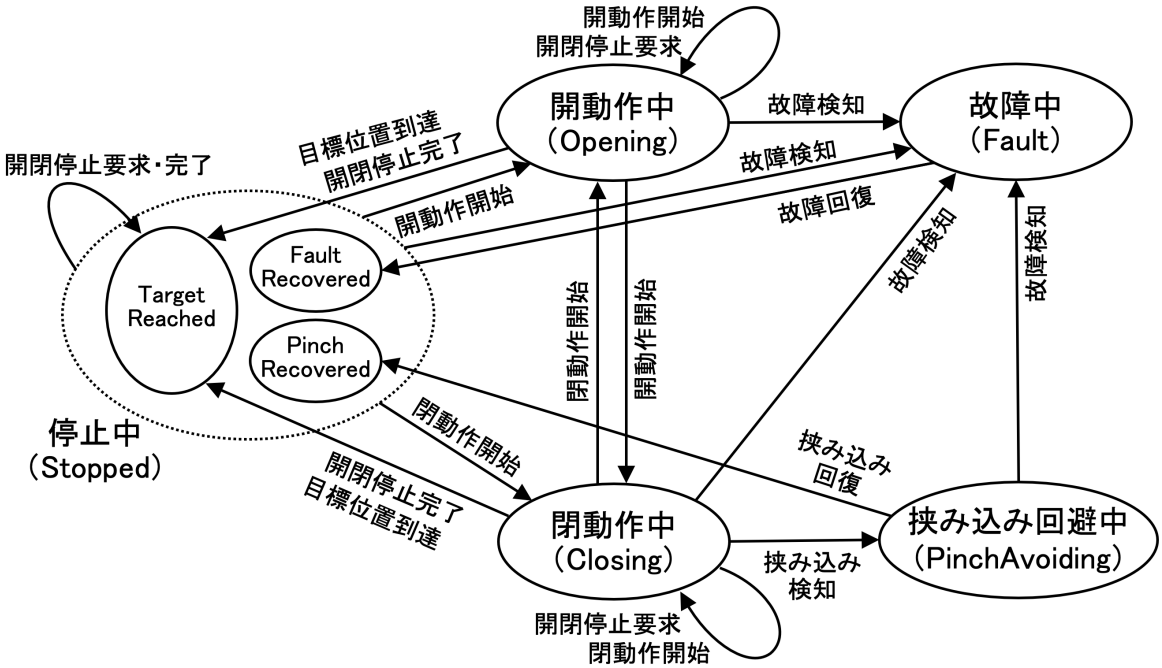

図 7. ウィンドウの拡張状態の状態遷移

commandSource（1ビット）

コマンド源。最後に正常実行した移動コマンドを出したのがビークルコンピュータの場合に0，開閉スイッチの場合に1とする。

position（最大7ビット）

ウィンドウの現在位置（0～100）または位置不明（UNKNOWN）または全閉でない（NONZERO）。

ECU内では，8ビット符号付き整数で表現し，UNKNOWNとNONZEROを負の値で表現しているが，表したい値は103通りしかないため，7ビットで表すことができる（以下の動作ロジックでは，7ビット表現と8ビット表現の間の変換処理は省略している）。また，必要なビット数を減らすために，位置の粒度を大きくしても良い。

switchDisabled（1ビット）
開閉スイッチが無効化されている場合に1とする。

sequenceNum（1ビット）
最後に処理したビークルコンピュータからの移動コマンドのシーケンス番号。

denyRiskPinch（1ビット）
最後に処理したビークルコンピュータからの移動コマンドを，RiskPinchにより拒否した場合に1とする。

denyRiskOpen（1ビット）
最後に処理したビークルコンピュータからの移動コマンドを，RiskOpenにより拒否した場合に1とする。

switchStatus（3ビット）
開閉スイッチの現在状態。表し方は，readSwitchStatus()で取得できる値と同じ。開閉スイッチを汎用スイッチとして使用しない場合は必要ない。

CANメッセージ2は，周期送信に加えて，メッセージ内のposition以外の内容に変化があった場合にも送信する。

5.4. 目標位置の扱い

CANメッセージ2には，ウィンドウ制御ECUからビークルコンピュータに，開閉動作の目標位置を伝えるフィールドを設けていない。これは，ウィンドウ制御ECUから伝えなくても，ビークルコンピュータ側で目標位置を補うことができるためである。

具体的には，ビークルコンピュータからの移動コマンドによって開閉動作している場合には，発行した移動コマンドの目標位置をビークルコンピュータ側で記憶しておけば良い。

開閉スイッチによる指示で開閉動作している場合には，開動作の時は目標位置が100，閉動作の時は目標位置が0とわかる。これは，開閉スイッチによる制御では，目標位置に0と100しか指定しないためである。

挟み込み防止機能により開動作している場合には，実際の目標位置をビークルコンピュータ側で知ることができないため，ビークルコンピュータ側でアプリケーションから見える目標位置をUNKNOWNとする（100とする手もある）。なお，Open SDV API仕様では，異常状態における目標位置については規定しておらず，この方法で仕様に合致している。

6. ウィンドウ制御ECUの動作ロジック

6.1. 内部状態

ウィンドウ制御ECUは，以下の内部状態を持つ。

以下の内部状態の中で，(*)で示すものは，CANメッセージ2に格納するデータであり，positionを除いて，その表し方はCANメッセージ2中の対応するフィールドと同じである。positionは，CANメッセージ2には7ビットで格納するために，特殊値の数値が異なる。ただし，以下の動作ロジックでは，7ビット表現と8ビット表現の間の変換処理は省略している。

extendedStatus (*)
ウィンドウの拡張状態。

commandSource (*)
コマンド源。最後に正常実行した移動コマンドを出したのがビークルコンピュータの場合に0，開閉スイッチの場合に1。

position (*)
ウィンドウの現在位置（0〜100）または位置不明（UNKNOWN）または全閉でない（NONZERO）。

targetPosition
開閉動作の目標位置。extendedStatusがOpening，Closing，PinchAvoidingの場合に有効。

extendedStatusがPinchAvoidingの場合は，ビークルコンピュータ側で管理されている目標位置（アプリケーションから見える目標位置）と一致しない。

switchDisabled (*)
開閉スイッチが無効化されている場合に1とする。

sequenceNum (*)
最後に処理したビークルコンピュータからの移動コマンドのシーケンス番号。

denyRiskPinch (*)
最後に処理したビークルコンピュータからの移動コマンドを，RiskPinchにより拒否した場合に1とする。

denyRiskOpen (*)
最後に処理したビークルコンピュータからの移動コマンドを，RiskOpenにより拒否した場合に1とする。

switchStatus (*)
開閉スイッチの現在状態。

autoOpenFlag
開スイッチが2段階目まで操作された場合にtrueとする。falseに初期化しておく。

autoCloseFlag
閉スイッチが2段階目まで操作された場合にtrueとする。falseに初期化しておく。

開閉動作中にcontrolMotor(Stop)でモーターに停止指示を出した後，停止が完了するまでの間は，extendedStatusをOpeningまたはClosingとする。開閉が停止した時点で呼び出される「開閉停止時の処理」で，extendedStatusをTargetReachedに変化させる。

6.2. 動作ロジック

以下では，CANメッセージ1の受信処理，開閉スイッチの読み込み処理，位置センサーの読み込み処理，開閉停止時の処理，故障検知時の処理，故障からの回復時の処理，挟み込み検知時の処理，CANメッセージ2の送信処理の動作ロジックを示す。これらの処理は，並行に

呼び出されることはないと想定している（並行して呼び出される場合には，排他制御の追加が必要である）。

なお，checkStartOpen，checkStartClose，checkStopOpen，checkStopClose，clearAutoFlagsは，1つのECUでの実装と同じものを使用する。

```
CANメッセージ1の受信処理(CANメッセージ1)
{
    // 移動コマンドの処理
    if (CANメッセージ1.sequenceNum != 内部状態.sequenceNum) {
        // 新しい移動コマンドを処理する。
        内部状態.denyRiskPinch = 0;
        内部状態.denyRiskOpen = 0;
        if (内部状態.extendedStatus != Fault
                    && 内部状態.extendedStatus != PinchAvoiding) {
            // 正常動作している場合，移動コマンドを実行する。
            if (CANメッセージ1.targetPosition == COMMAND_STOP) {
                // 開閉停止
                if (内部状態.extendedStatus == Opening
                            || 内部状態.extendedStatus == Closing) {
                    controlMotor(Stop);
                }
                else {
                    内部状態.extendedStatus = TargetReached;
                }
                内部状態.commandSource = 0;
                clearAutoFlags();
            }
            else if (checkStartOpen(CANメッセージ1.targetPosition)) {
                // 開動作
                if (CANメッセージ1.riskOpen == 0) {
                    内部状態.extendedStatus = Opening;
                    内部状態.commandSource = 0;
                    内部状態.targetPosition = CANメッセージ1.targetPosition;
                    controlMotor(Open);
                    clearAutoFlags();
                }
                else {
                    内部状態.denyRiskOpen = 1;
                }
            }
            else if (checkStartClose(CANメッセージ1.targetPosition)) {
                // 閉動作
                if (CANメッセージ1.riskPinch == 0) {
                    内部状態.extendedStatus = Closing;
                    内部状態.commandSource = 0;
                    内部状態.targetPosition = CANメッセージ1.targetPosition;
                    controlMotor(Close);
                    clearAutoFlags();
                }
```

```
                else {
                    内部状態.denyRiskPinch = 1;
                }
            }
            else {
                // すでに目標位置にある場合
                if (内部状態.extendedStatus == Opening
                            || 内部状態.extendedStatus == Closing) {
                    controlMotor(Stop);
                }
                else {
                    内部状態.extendedStatus = TargetReached;
                }
                内部状態.commandSource = 0;
                clearAutoFlags();
            }
        }
        else {
            // 異常状態（故障中，挟み込み回避中）の場合，何もしない。
            // 理由は「異常状態での振る舞い」の節に記載の通り。
        }
        内部状態.sequenceNum = CANメッセージ1.sequenceNum;
    }

    // 開閉スイッチの無効化に関する処理
    if (CANメッセージ1.enableSwitchOnSuccess == 0
            || 内部状態.denyRiskPinch == 1
            || 内部状態.denyRiskOpen == 1) {
        内部状態.switchDisabled = CANメッセージ1.disableSwitch;
        if (内部状態.switchDisabled == 1) {
            clearAutoFlags();
        }
    }
    else {
        内部状態.switchDisabled = 0;
    }
}

開閉スイッチの読み込み処理() // 周期処理
{
    // switchStatusはローカル変数
    switchStatus = readSwitchStatus();

    if (内部状態.extendedStatus != Fault
                && 内部状態.extendedStatus != PinchAvoiding
                && 内部状態.switchDisabled == 0) {
        switch (switchStatus) {
        case NoAction:
            if ((内部状態.extendedStatus == Opening
                            || 内部状態.extendedStatus == Closing)
```

```
                        && 内部状態.switchStatus != NoAction
                        && 内部状態.switchStatus != Fault
                        && !(内部状態.autoOpenFlag)
                        && !(内部状態.autoCloseFlag)) {
                    内部状態.commandSource = 1;
                    controlMotor(Stop);
                }
                break;
            case ManualOpen:
            case AutoOpen:
                内部状態.extendedStatus = Opening;
                内部状態.commandSource = 1;
                内部状態.targetPosition = 100;
                controlMotor(Open);
                内部状態.autoCloseFlag = false;
                if (switchStatus == AutoOpen) {
                    内部状態.autoOpenFlag = true;
                }
                break;
            case ManualClose:
            case AutoClose:
                内部状態.extendedStatus = Closing;
                内部状態.commandSource = 1;
                内部状態.targetPosition = 0;
                controlMotor(Close);
                内部状態.autoOpenFlag = false;
                if (switchStatus == AutoClose) {
                    内部状態.autoCloseFlag = true;
                }
                break;
            case Fault:
                if ((内部状態.extendedStatus == Opening
                            || 内部状態.extendedStatus == Closing)
                        && 内部状態.switchStatus != Fault) {
                    内部状態.commandSource = 1;
                    controlMotor(Stop);
                    clearAutoFlags();
                }
                break;
            }
        }
        else {
            // 異常状態（故障中，挟み込み回避中）と，開閉スイッチが無効化
            // されている場合には，何もしない
        }
        内部状態.switchStatus = switchStatus;
    }

    位置センサーの読み込み処理() // 周期処理
    {
```

```
    // positionはローカル変数
    position = readPosition();
    switch (内部状態.extendedStatus) {
    case TargetReached:
    case FaultRecovered:
    case PinchRecovered:
        // 何もしない（停止中に現在位置が変化した場合の例外処理は，ビー
        // クルコンピュータ側で行う）
        break;
    case Opening:
        if (checkStopOpen(position)) {
            // 開動作中に目標位置に到達
            controlMotor(Stop);
        }
        break;
    case Closing:
        if (checkStopClose(position)) {
            // 閉動作中に目標位置に到達
            controlMotor(Stop);
        }
        break;
    case Fault:
        // 何もしない
        break;
    case PinchAvoiding:
        if (position != UNKNOWN && position != NONZERO
                    && position >= 内部状態.targetPosition - STOP_OVERRUN) {
            // 挟み込み回避中に目標位置に到達
            controlMotor(Stop);
        }
        break;
    }
    内部状態.position = position;
}

開閉停止時の処理()
{
    switch (内部状態.extendedStatus) {
    case Opening:
    case Closing:
        内部状態.extendedStatus = TargetReached;
        clearAutoFlags();
        break;
    case PinchAvoiding:
        内部状態.extendedStatus = PinchRecovered;
        break;
    }
}

故障検知時の処理()
```

```
{
    内部状態.extendedStatus = Fault;
    clearAutoFlags();
    controlMotor(Stop);             // 可能なら停止させる
}

故障からの回復時の処理()
{
    内部状態.extendedStatus = FaultRecovered;
    controlMotor(Stop);             // 念のため停止させる
}

挟み込み検知時の処理()
{
    内部状態.extendedStatus = PinchAvoiding;
    内部状態.targetPosition = 挟み込み回避のための開動作の目標位置;
    clearAutoFlags();
    controlMotor(Open);
}

CANメッセージ2の送信処理()    // 周期＋イベント処理
{
    内部状態からCANメッセージ2を構成し，CANに送出する
}
```

7. ビークルコンピュータの動作ロジック

7.1. サービスコールからリターンするタイミング

ウィンドウを操作するサービスコール（startMove，stopMove，lock，unlock）は，ウィンドウ制御ECUによって処理された後にリターンすることを基本とする。これは，次の理由による。

- ウィンドウが開閉動作中にstartMoveが呼び出された場合，startMoveによりウィンドウが開動作するか閉動作するかがウィンドウ制御ECU側でしか決定できず，リスク制御により移動コマンドを拒否するかも，ウィンドウ制御ECU側でしか判定できない。そのため，ウィンドウ制御ECUでの判定後に，サービスコールからリターンする必要がある。
- lockにより開閉スイッチが無効化される場合（lockの操作優先度が，開閉スイッチの操作優先度と同じかそれより高い場合），開閉スイッチの無効化がウィンドウ制御ECUに処理されたことを確認してからlockからリターンしないと，lockから正常リターンした後に，開閉スイッチによりウィンドウが動作開始する可能性がある。
- ウィンドウ制御ECUのダウン時やCAN通信の故障時に，サービスコールが正常実行されなかったにも関わらず，E_OKが返ってしまう可能性がある（これを，故障時の例外的な振る舞いとして，許容することも考えられる）。

【補足説明】

startMoveをstartOpenとstartCloseに分ければ，リスク制御により移動コマンドを拒否するかを，ビークルコンピュータ側で判定できるようになる。ただし，アプリケーションからの

使いやすさは，悪化すると思われる。

7.2. コマンドの伝え方とコマンドキュー

startMove/stopMoveによる移動コマンドは，CANメッセージ1のtargetPositionなどのフィールドにより，ウィンドウ制御ECUに伝える。lock/unlockによるロックコマンドは，ロック状態の変化により開閉スイッチの無効化に影響を与える場合のみ，CANメッセージ1のdisableSwitchを用いてウィンドウ制御ECUに伝える。startMove/stopMoveが正常実行された場合に，開閉スイッチの無効化を解除する必要があることは，CANメッセージ1のenableSwitchOnSuccessを用いてウィンドウ制御ECUに伝える。

移動コマンドをウィンドウ制御ECUに伝える際に，移動コマンドに付与するシーケンス番号を1ビットとしているため，ある移動コマンドが処理されるまで，次の移動コマンドの発行を待たせる必要がある。一般には，複数の移動コマンドが待つことになるため，コマンドキューを用意し，待っているコマンドを管理する（図　8）。並行して呼び出されたサービスコール間の実行順序は任意で良いため，コマンドキューで順序を管理する必要はないが，簡単のためにキューで実現する。

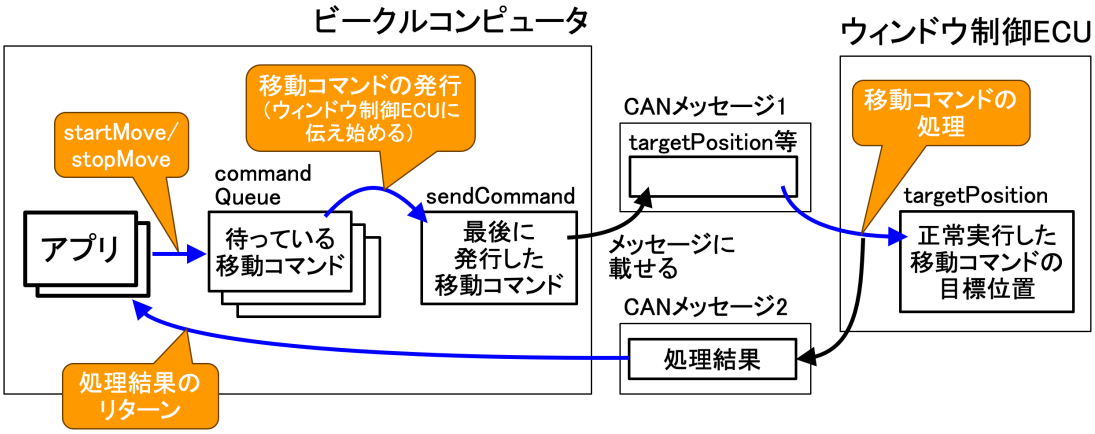

図 8. 移動コマンドの処理の流れ

開閉スイッチ無効化制御コマンドについても，1つの指示が実行されたことを確認してから，次の指示を伝える必要がある。これは，複数の指示を連続して伝えると，図　9のような状況で，どの指示の実行が完了したか区別できないためである。

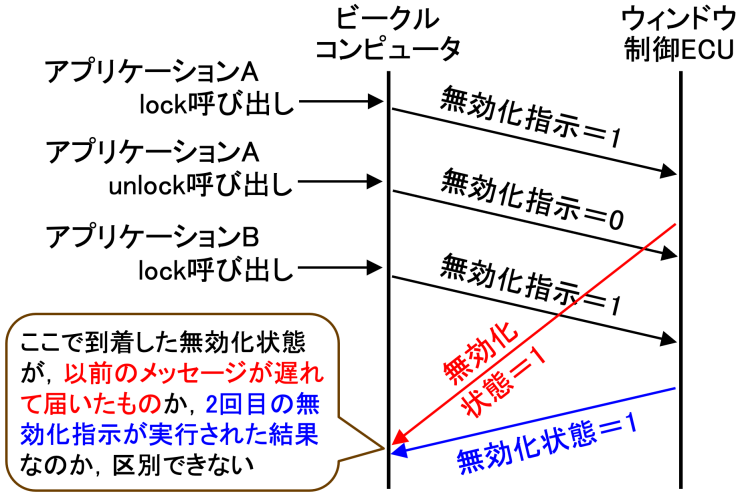

図 9. 開閉スイッチ無効化の確認の必要性

そのため，lockとunlockについても，1つのロックコマンドが処理されるまで，次のロックコマンドの発行を待たせる必要がある。一般には，複数のロックコマンドが待つことになるため，ロックコマンドについてもコマンドキューで管理する。開閉スイッチの無効化状態に影響しないlock/unlockについても，まずはコマンドキューに入れることとし，それを省略する方法については最適化手法として後述する。

CANメッセージ1では，移動コマンドと開閉スイッチ無効化制御コマンドを，同時に伝えることができる（両コマンドを同時に伝えた場合，両方がアトミックに実行される）。また，新しい移動コマンドを発行できる状態と，新しい開閉スイッチ無効化制御コマンドを発行できる状態は，原理的には独立である。

そのため，移動コマンドとロックコマンドが独立に実行できるものであれば，両コマンドを同時/独立に発行することができる。しかし実際には，ロックの取得により移動コマンドがエラーになる場合や，移動コマンドでロックが解除される場合があるなど，独立には実行できない。

そこで，まずは，新しい移動コマンドを発行でき，かつ，新しい開閉スイッチ無効化制御コマンドを発行できる場合（この状態をコマンド発行可能状態と呼ぶ）に，新しいコマンド（移動コマンドまたは開閉スイッチ無効化制御コマンド）を1つだけ発行することとする。両コマンドを同時/独立に発行する手法については，最適化手法として後述する。

7.3. 状態を変化させないサービスコール

サービスコールはウィンドウ制御ECUによって処理された後にリターンするという原則の例外として，状態を変化させないサービスコールがある。

並行して呼び出されたサービスコール間の実行順序は任意で良いという仕様から，状態を変化させないサービスコールの処理は，ウィンドウ制御ECUによって処理されていないコマンドが残っていても，その時点の状態に基づいて実行して良い。

これに該当するケースとして，サービスコールがエラーになる場合がある。具体的には，E_OBJECT_LOCKED，E_OBJECT_FAULT，E_OBJECT_STATUSなどのエラーチェックは，サービスコールが呼び出された時点や，コマンドキューに入っている間に行なっても良い。ただし，そのコマンドを発行する時にも，必ずエラーチェックを行うことが必要である。

同じ理由により，getStatusは即座に実行することができる。

7.4. 現在状態の取り込み時のロックの強制解除

ビークルコンピュータは，ウィンドウ制御の現在状態を内部状態に取り込む時に，必要に応じてロックの強制解除を行う。ここでは，ウィンドウ制御ECUから伝えられた拡張情報とコマンド源の情報を用いて，ロックの強制解除を行う条件について検討する。

ロックを強制解除するのは，ロックを取得していないアプリケーションが，ロックを取得しているよりも高い操作優先度で移動コマンドを実行した時と，異常状態に遷移した時である。そのため，すべての状態遷移がビークルコンピュータに伝わるのであれば，これらの時だけにロックを強制解除すれば良い。一方，一部の状態遷移が伝わらない場合には，伝わらなかった状態遷移を補う形で，ロックの強制解除を行う必要がある。

例えば，主状態が停止中で，ロックが取得されている時に，ウィンドウの故障が検知され，

その後すぐに故障から回復した場合を考える。この時，すべての状態遷移が伝わっていれば，故障が検知された時にロックを強制解除すれば良い。一方，「故障検知」の状態遷移が伝わらず，「故障から回復」の状態遷移が伝わった場合には，その前に「故障検知」の状態遷移をしていたはずなので，ロックを強制解除する必要がある。

同様に，「挟み込みから回復」の状態遷移が伝わった場合も，その前に「挟み込み検知」の状態遷移をしていたはずなので，ロックを強制解除する。

「目標位置に到達」の状態遷移が伝わった場合も，その前に「開動作中」または「閉動作中」に状態遷移していたはずなので，動作を指示したのがロックを取得していないアプリケーションであった場合には，ロックを強制解除することが必要である。

ただし，ビークルコンピュータからの移動コマンドによる「開動作中」または「閉動作中」への状態遷移は，移動コマンドの処理結果という形で必ず伝えられ，抜けることはない。そのため，移動コマンドの処理結果を取り込む際にロックの強制解除を行っておけば，ウィンドウ制御の現在状態を取り込む時には，ロックの強制解除を行う必要がない。また，ビークルコンピュータからの移動コマンドによる「目標位置に到達」の状態遷移の際にも，ロックの強制解除を行う必要がない。

なお，コマンド源の情報は，移動コマンドがビークルコンピュータからのものか，開閉スイッチによる制御からのものかを区別するために用いる。

7.5. CANメッセージ2受信処理の流れ

ビークルコンピュータは，CANメッセージ2を受信すると，以下の処理を行う。

1. 新たに処理された移動コマンドの処理結果の取り込み

   移動コマンドが正常実行された場合には，（必要な場合には）ロックの強制解除を行う，目標位置を現在位置に一致させるなど，内部状態を移動コマンドの実行後の状態に更新する。また，移動コマンドを出したサービスコールからリターンする。

   移動コマンドがリスク制御により拒否された場合には，移動コマンドを出したサービスコールからエラーリターンする。

2. 新たに処理された開閉スイッチ無効化制御コマンドの処理結果の取り込み

   開閉スイッチ無効化制御コマンドが実行された場合には，内部状態をロックコマンドの実行後の状態に更新し，ロックコマンドを出したサービスコールからリターンする。

3. ウィンドウ制御の現在状態の取り込み

   CANメッセージ2でウィンドウ制御ECUから伝えられるウィンドウの拡張状態またはコマンド源が変化した場合には，新しい状態を内部状態に取り込む。また，イベントの生成と，（必要な場合には）ロックの強制解除を行う。

   移動コマンドが新たに正常実行された場合にも，これらの処理を行う。これは，ビークルコンピュータから同じコマンドが連続して発行された場合には，ウィンドウの拡張状態とコマンド源が変化しないにも関わらず，これらの処理が必要になるためである。

   さらに，上の条件に合致せず，現在位置のみが変化した場合の処理も行う。

4. コマンドの発行

   コマンドが処理された結果，コマンド発行可能状態となった場合には，新たなコマンドを発行する。コマンドキューの先頭から順に，発行できるコマンドを探し，発行可能な最初のコマンドを発行する。発行可能なコマンドを探す過程で，エラーになるコマンドや，ウィンドウ制御ECUにコマンドを発行することなく実行できるコマンドがあれば，それらを処理する。

5. 開閉スイッチの状態の取り込み

   開閉スイッチの状態を，内部状態に取り込む。

7.6. コマンドとコマンドキューの操作

コマンドの管理するためのデータ構造として，以下のデータ型を用意する。

```
type CommandType = struct {
    targetPosition : UInt8 ([0,100] | COMMAND_STOP(127)
                    | COMMAND_LOCK(126) | COMMAND_UNLOCK(125)
                    | COMMAND_NULL(124)),
    application : ApplicationIdType,
    priority : PriorityType,
    riskPinchCode : ReturnType,
    riskOpenCode : ReturnType,
    // この他に，コマンドを出したサービスコールに対して処理結果を送り，
    // サービスコールからリターンさせるために必要な情報を格納する。
};
```

各フィールドの意味は次の通り。

targetPosition
ウィンドウの目標位置またはコマンドの種類。以下のいずれかで表す。

- 0〜100の数値（開閉開始（startMove）。数値は目標位置を表す）
- COMMAND_STOP（開閉停止（stopMove））
- COMMAND_LOCK（ロック取得（lock））
- COMMAND_UNLOCK（ロック解除（unlock））
- COMMAND_NULL（コマンドが無効）

application
コマンドを出したアプリケーションID。

priority
コマンドの操作優先度。

riskPinchCode
移動コマンドが，RiskPinchに対応できている場合にE_OK，対応できていない場合には，リスク制御で拒否された場合に返すべきエラーコード。移動コマンドに対してのみ有効。

riskOpenCode
移動コマンドが，RiskOpenに対応できている場合にE_OK，対応できていない場合には，リスク制御で拒否された場合に返すべきエラーコード。移動コマンドに対してのみ有効。

以下の動作ロジックでは，コマンドを管理するコマンドキューに対する操作を，以下のように記述する。

foreach *command* in *commandQueue* { 処理 }
*commandQueue*で指定されるコマンドキュー中のコマンドを，順に*command*に代入し，処理を実行する。

***commandQueue*.enqueue(*command*)**
*command*で指定されるコマンドを，*commandQueue*で指定されるコマンドキューに入れる。

***commandQueue*.delete(*command*)**
*command*で指定されるコマンドを，*commandQueue*で指定されるコマンドキューから削除する。

***commandQueue*.empty()**
*commandQueue*で指定されるコマンドキューが空の場合にtrueを返す。

7.7. 内部状態

ビークルコンピュータは，以下の内部状態を持つ。getStatusは，(*)で示す内部状態をそのまま返す（ただし，ロック状態の3つのフィールドは，以下ではフラットな内部状態と扱っているが，getStatusでは1つの構造体に入れて返す必要がある）。

mainStatus (*)
ウィンドウの主状態。CANメッセージ2を受信したタイミングで，ウィンドウの拡張状態から生成する。

extendedStatus
ウィンドウの拡張状態。前回受信したCANメッセージ2中のextendedStatus。

commandSource
コマンド源。最後に正常実行した移動コマンドを出したのがビークルコンピュータの場合に0，開閉スイッチの場合に1。前回受信したCANメッセージ2中のcommandSource。

position (*)
ウィンドウの現在位置。前回受信したCANメッセージ2中のposition。

targetPosition (*)
ウィンドウの目標位置（ウィンドウ制御ECUに最後に正常実行された移動コマンドによって設定された値，または，目標位置到達時の位置）。

currentApplication
ウィンドウ制御ECUに最後に正常実行されたビークルコンピュータからの移動コマンドを出したアプリケーションID。

sequenceNum
ウィンドウ制御ECUに最後に処理されたビークルコンピュータからの移動コマンドのシーケンス番号。前回受信したCANメッセージ2中のsequenceNum。

sendCommand
最後に発行したコマンド。

sendSequenceNum
sendCommandの移動コマンドのシーケンス番号。sendCommandの移動コマンドがウィンドウ制御ECUに処理された後は，sequenceNumと一致する。

sendEnableSwitchOnSuccess
sendCommandの移動コマンドが正常実行されたら，開閉スイッチの無効化を解除する場合に，1とする。

commandQueue
実行を待っているコマンドを管理するキュー。

locked (*)
ロックされているか否か。ロックされている場合にtrueとなる。

lockApplication (*)
ロックしたアプリケーションID。lockedがtrueの場合に有効。

lockPriority (*)
ロックを取得した操作優先度。lockedがtrueの場合に有効。

sendDisableSwitch
ウィンドウ制御ECUに対して，開閉スイッチの無効化を指示する場合に1とする。

switchDisabled
ウィンドウ制御ECUが，開閉スイッチを無効化している場合に1。前回受信したCANメッセージ2中のswitchDisabled。

switchStatus
開閉スイッチの現在状態。前回受信したCANメッセージ2中のswitchStatus。

7.8. 動作ロジック

以下では，startMove，stopMove，lock，unlockの各サービスコールの処理ルーチン，CANメッセージ2の受信処理，CANメッセージ1の送信処理の動作ロジックを示す。これらの処理は，並行に呼び出されることはないと想定している（並行して呼び出される場合には，排他制御の追加が必要である）。

また，これらから呼び出される関数として，revokeLock，canIssueCommand，

issueMoveCommand，checkLock，executeLock，checkUnlock，executeUnlockの動作ロジックを示す。

なお，targetPositionWhenStopped，checkMove，checkRiskは，1つのECUでの実装と同じものを使用する。

以下の動作ロジック中の「〜〜イベント（〜〜）を生成」という記述の意味は，1つのECUでの実装と同じである。

```
// ロックの強制解除
//
// OtherLockReleasedイベントの生成が必要な場合には，lockReleasedをtrue
// にする。
revokeLock(lockReleased)
{
    OwnLockRevokedイベント（内部状態.lockApplication）を生成
    if (lockReleased)
        OtherLockReleasedイベント（内部状態.lockApplication）を生成
    }
    内部状態.locked = false;
    内部状態.sendDisableSwitch = 0;
}

// コマンド発行可能状態かの判断
canIssueCommand()
{
    return 内部状態.sendSequenceNum == 内部状態.sequenceNum
            && 内部状態.sendDisableSwitch == 内部状態.switchDisabled;
}

// 移動コマンドの発行
issueMoveCommand(command, lockRevoke)
{
    内部状態.sendCommand = command;
    内部状態.sendSequenceNum = 1 - 内部状態.sendSequenceNum;
    if (lockRevoke && 内部状態.switchDisabled == 1) {
        内部状態.sendEnableSwitchOnSuccess = 1;
    }
}

// startMoveサービスコール
startMove(position, priority)
{
    // command, retval, lockRevokeはローカル変数
    retval = checkMove(呼び出したアプリケーションID, priority, &lockRevoke);
    if (retval != E_OK) {
        return retval;
    }

    // コマンドのデータ構造の作成
```

```
    commandのメモリ領域を確保する
    command.targetPosition = position;
    command.application = 呼び出したアプリケーションID;
    command.priority = priority;
    command.riskPinchCode = checkRisk(RiskPinch);
    command.riskOpenCode = checkRisk(RiskOpen);

    if (canIssueCommand()) {
        issueMoveCommand(command, lockRevoke);
        コマンドが処理されるのを待ち，処理結果をretvalに格納
    }
    else {
        内部状態.commandQueue.enqueue(command);
        コマンドが処理されるのを待ち，処理結果をretvalに格納
    }
    return retval;
}

// stopMoveサービスコール
stopMove(priority)
{
    // command, retval, lockRevokeはローカル変数
    retval = checkMove(呼び出したアプリケーションID, priority, &lockRevoke);
    if (retval != E_OK) {
        return retval;
    }

    // コマンドのデータ構造の作成
    commandのメモリ領域を確保する
    command.targetPosition = COMMAND_STOP;
    command.application = 呼び出したアプリケーションID;
    command.priority = priority;
    command.riskPinchCode = E_OK;
    command.riskOpenCode = E_OK;

    if (canIssueCommand()) {
        issueMoveCommand(command, lockRevoke);
        コマンドが処理されるのを待ち，処理結果をretvalに格納
    }
    else {
        内部状態.commandQueue.enqueue(command);
        コマンドが処理されるのを待ち，処理結果をretvalに格納
    }
    return retval;
}

// lockのエラーチェック
//
// 開閉スイッチの無効化が必要な場合に，*p_disableSwitchをtrueにする。
checkLock(application, priority, p_disableSwitch)
```

```
{
    *p_disableSwitch = false;
    if (内部状態.locked) {
        if (priority >= 内部状態.lockPriority) {
            return E_OBJECT_LOCKED;
        }
    }
    if (priority <= 開閉スイッチの操作優先度 && 内部状態.switchDisabled == 0) {
        *p_disableSwitch = true;
    }
    switch (内部状態.mainStatus) {
    case Fault:
        return E_OBJECT_FAULT;
    case PinchAvoiding:
        return E_OBJECT_STATUS;
    }
    return E_OK;
}

// ロック取得処理
executeLock(application, priority)
{
    if (内部状態.locked && 内部状態.lockApplication != application) {
        OwnLockRevokedイベント（内部状態.lockApplication）を生成
    }
    内部状態.locked = true;
    内部状態.lockApplication = application;
    内部状態.lockPriority = priority;
}

// lockサービスコール
lock(priority)
{
    // command, retval, disableSwitchはローカル変数
    retval = checkLock(呼び出したアプリケーションID, priority, &disableSwitch);
    if (retval != E_OK) {
        return retval;
    }

    // コマンドのデータ構造の作成
    commandのメモリ領域を確保する
    command.targetPosition = COMMAND_LOCK;
    command.application = 呼び出したアプリケーションID;
    command.priority = priority;

    if (canIssueCommand()) {
        if (!disableSwitch) {
            executeLock(呼び出したアプリケーションID, priority);
            retval = E_OK;
        }
```

```
        else {
            内部状態.sendCommand = command;
            内部状態.sendDisableSwitch = 1;
            コマンドが処理されるのを待ち，処理結果をretvalに格納
        }
    }
    else {
        内部状態.commandQueue.enqueue(command);
        コマンドが処理されるのを待ち，処理結果をretvalに格納
    }
    return retval;
}

// unlockのエラーチェック
//
// 開閉スイッチの無効化の解除が必要な場合に，*p_enableSwitchをtrueにする。
checkUnlock(application, p_enableSwitch)
{
    *p_enableSwitch = false;
    if (!(内部状態.locked) || 内部状態.lockApplication != application) {
        return E_OBJECT_STATUS;
    }
    if (内部状態.lockPriority <= 開閉スイッチの操作優先度
                                    && 内部状態.switchDisabled == 1) {
        *p_enableSwitch = true;
    }
    return E_OK;
}

// ロック解除処理
executeUnlock(application)
{
    OtherLockReleasedイベント（application）を生成
    内部状態.locked = false;
}

// unlockサービスコール
unlock()
{
    // command, retval, enableSwitchはローカル変数
    retval = checkUnlock(呼び出したアプリケーションID, &enableSwitch);
    if (retval != E_OK) {
        return retval;
    }

    // コマンドのデータ構造の作成
    commandのメモリ領域を確保する
    command.targetPosition = COMMAND_UNLOCK;
    command.application = 呼び出したアプリケーションID;
    command.priority = PRIORITY_NULL;
```

```
    if (canIssueCommand()) {
        if (!enableSwitch) {
            executeUnlock(呼び出したアプリケーションID);
            retval = E_OK;
        }
        else {
            内部状態.sendCommand = command;
            内部状態.sendDisableSwitch = 0;
            コマンドが処理されるのを待ち，処理結果をretvalに格納
        }
    }
    else {
        内部状態.commandQueue.enqueue(command);
        コマンドが処理されるのを待ち，処理結果をretvalに格納
    }
    return retval;
}

CANメッセージ2の受信処理(CANメッセージ2)
{
    // moveCommandExecutedはローカル変数
    moveCommandExecuted = false;

    if (CANメッセージ2.sequenceNum != 内部状態.sequenceNum) {
        // 発行している移動コマンドが新たに処理された。
        assert(vcstat.sendCommand.targetPosition != COMMAND_LOCK
               && vcstat.sendCommand.targetPosition != COMMAND_UNLOCK);
        // RiskPinchとRiskOpenの両方で拒否されることはないため，どちら
        // を先にチェックしても良い。
        if (CANメッセージ2.denyRiskPinch == 1) {
            内部状態.sendCommandを出したサービスコールに，
                内部状態.sendCommand.riskPinchCodeを送り，リターンさせる
        }
        else if (CANメッセージ2.denyRiskOpen == 1) {
            内部状態.sendCommandを出したサービスコールに，
                内部状態.sendCommand.riskOpenCodeを送り，リターンさせる
        }
        else {
            // 移動コマンドの処理結果の取り込み
            moveCommandExecuted = true;
            内部状態.currentApplication = 内部状態.sendCommand.application;
            if (内部状態.locked && 内部状態.lockApplication
                                        != 内部状態.currentApplication) {
                revokeLock(true);
            }
            if (内部状態.sendCommand.targetPosition != COMMAND_STOP) {
                内部状態.targetPosition = 内部状態.sendCommand.targetPosition;
            }
            else {
```

```
                内部状態.targetPosition
                        = targetPositionWhenStopped(CANメッセージ2.position);
            }
            if (内部状態.sendEnableSwitchOnSuccess == 1) {
                // 以下のassertが成り立たなくなるのは，ウィンドウ制御
                // ECUが，移動コマンドをアトミックに実行しなかった場合。
                assert(CANメッセージ2.switchDisabled == 0);
                内部状態.sendEnableSwitchOnSuccess = 0;
            }
            ControlledBySelfイベント（内部状態.currentApplication）を生成
            内部状態.sendCommandを出したサービスコールに，E_OKを送り，リターンさせる
        }
        内部状態.sequenceNum = CANメッセージ2.sequenceNum;
    }

    if (CANメッセージ2.switchDisabled != 内部状態.switchDisabled) {
        // 開閉スイッチ無効化制御コマンドが新たに処理された。
        switch (内部状態.sendCommand.targetPosition) {
        case COMMAND_LOCK:
            assert(CANメッセージ2.switchDisabled == 1);
            executeLock(内部状態.sendCommand.application,
                        内部状態.sendCommand.priority);
            内部状態.sendCommandを出したサービスコールに，E_OKを送り，リターンさせる
            内部状態.sendCommand.targetPosition = COMMAND_NULL;
            break;
        case COMMAND_UNLOCK:
            assert(CANメッセージ2.switchDisabled == 0);
            executeUnlock(内部状態.sendCommand.application);
            内部状態.sendCommandを出したサービスコールに，E_OKを送り，リターンさせる
            内部状態.sendCommand.targetPosition = COMMAND_NULL;
            break;
        default:
            // 移動コマンドや異常状態への遷移によって，開閉スイッチの
            // 無効化が解除された。
            assert(CANメッセージ2.switchDisabled == 0);
            break;
        }
        内部状態.switchDisabled = CANメッセージ2.switchDisabled;
    }

    if (moveCommandExecuted
            || CANメッセージ2.extendedStatus != 内部状態.extendedStatus
            || CANメッセージ2.commandSource != 内部状態.commandSource) {
        // ウィンドウ制御ECUの状態が変化
        switch (CANメッセージ2.extendedStatus) {
        case TargetReached:
            内部状態.mainStatus = Stopped;
            if (CANメッセージ2.commandSource == 0) {
                TargetReached（内部状態.currentApplication）イベントを生成
            }
```

```
            else {
                if (内部状態.locked) {
                    revokeLock(true);
                }
                TargetReached（開閉スイッチによる制御のID）イベントを生成
            }
            内部状態.targetPosition
                    = targetPositionWhenStopped(CANメッセージ2.position);
            break;
        case FaultRecovered:
            内部状態.mainStatus = Stopped;
            if (内部状態.locked) {
                revokeLock(false);
            }
            OtherLockReleasedイベント（NULL）を生成
            FaultRecoveredイベントを生成
            内部状態.targetPosition
                    = targetPositionWhenStopped(CANメッセージ2.position);
            break;
        case PinchRecovered:
            内部状態.mainStatus = Stopped;
            if (内部状態.locked) {
                revokeLock(false);
            }
            OtherLockReleasedイベント（NULL）を生成
            PinchRecoveredイベントを生成
            内部状態.targetPosition
                    = targetPositionWhenStopped(CANメッセージ2.position);
            break;

        case Opening:
            内部状態.mainStatus = Opening;
            if (CANメッセージ2.commandSource == 0) {
                // ロックの強制解除と内部状態.targetPositionの更新は,
                // 移動コマンドの処理結果の取り込みで行われているため,
                // ここで行う必要はない。
                ControlledByOtherイベント（内部状態.currentApplication）を生成
            }
            else {
                if (内部状態.locked) {
                    revokeLock(true);
                }
                ControlledByOtherイベント（開閉スイッチによる制御のID）を生成
                内部状態.targetPosition = 100;
            }
            break;

        case Closing:
            内部状態.mainStatus = Closing;
            if (CANメッセージ2.commandSource == 0) {
```

```
            // ロックの強制解除と内部状態.targetPositionの更新は,
            // 移動コマンドの処理結果の取り込みで行われているため,
            // ここで行う必要はない。
            ControlledByOtherイベント（内部状態.currentApplication）を生成
        }
        else {
            if (内部状態.locked) {
                revokeLock(true);
            }
            ControlledByOtherイベント（開閉スイッチによる制御のID）を生成
            内部状態.targetPosition = 0;
        }
        break;

    case Fault:
        内部状態.mainStatus = Fault;
        if (内部状態.locked) {
            revokeLock(false);
        }
        FaultDetectedイベントを生成
        内部状態.targetPosition = UNKNOWN;
        break;

    case PinchAvoiding:
        内部状態.mainStatus = PinchAvoiding;
        if (内部状態.locked) {
            revokeLock(false);
        }
        PinchDetectedイベントを生成
        内部状態.targetPosition = UNKNOWN;
        break;
    }
    内部状態.extendedStatus = CANメッセージ2.extendedStatus;
    内部状態.commandSource = CANメッセージ2.commandSource;
    内部状態.position = CANメッセージ2.position;
}
else if (CANメッセージ2.position != 内部状態.position) {
    // 現在位置のみが変化
    if ((内部状態.extendedStatus == TargetReached
                    || 内部状態.extendedStatus == FaultRecovered
                    || 内部状態.extendedStatus == PinchRecovered)
                && 内部状態.position != UNKNOWN
                && 内部状態.position != NONZERO) {
        // 停止中に現在位置が動いた場合は，一瞬故障したものと扱う。
        if (内部状態.locked) {
            revokeLock(false);
        }
        // FaultDetectedイベントの生成は省くことができる。
        OtherLockReleasedイベント（NULL）を生成
        FaultRecoveredイベントを生成
```

```
            内部状態.targetPosition
                    = targetPositionWhenStopped(CANメッセージ2.position);
            // 内部状態.extendedStatusは，アプリケーションからは参照で
            // きない状態なので，ここで更新する必要はない。
        }
        内部状態.position = CANメッセージ2.position;
    }

    // command, retval, lockRevoke, disableSwitch, enableSwitchは
    // ローカル変数
    if (canIssueCommand()) {
        // コマンドの発行
        foreach command in 内部状態.commandQueue {
            switch (command.targetPosition) {
            default:        // startMove
            case COMMAND_STOP:
                retval = checkMove(command.application,
                                   command.priority, &lockRevoke);
                if (retval != E_OK) {
                    commandを出したサービスコールに，retvalを送り，リターンさせる
                    内部状態.commandQueue.delete(command);
                }
                else {
                    issueMoveCommand(command, lockRevoke);
                    内部状態.commandQueue.delete(command);
                    // foreachループを抜ける。
                    goto exit;
                }
                break;
            case COMMAND_LOCK:
                retval = checkLock(command.application,
                                   command.priority, &disableSwitch);
                if (retval != E_OK) {
                    commandを出したサービスコールに，retvalを送り，リターンさせる
                    内部状態.commandQueue.delete(command);
                }
                else if (!disableSwitch) {
                    executeLock(command.application, command.priority);
                    commandを出したサービスコールに，E_OKを送り，リターンさせる
                    内部状態.commandQueue.delete(command);
                }
                else {
                    // 開閉スイッチの無効化を指示
                    内部状態.sendCommand = command;
                    内部状態.sendDisableSwitch = 1;
                    内部状態.commandQueue.delete(command);
                    // foreachループを抜ける。
                    goto exit;
                }
                break;
```

```
            case COMMAND_UNLOCK:
                retval = checkUnlock(command.application, &enableSwitch);
                if (retval != E_OK) {
                    commandを出したサービスコールに，retvalを送り，リターンさせる
                    内部状態.commandQueue.delete(command);
                }
                else if (!enableSwitch) {
                    executeUnlock(command.application);
                    commandを出したサービスコールに，E_OKを送り，リターンさせる
                    内部状態.commandQueue.delete(command);
                }
                else {
                    // 開閉スイッチの無効化解除を指示
                    内部状態.sendCommand = command;
                    内部状態.sendDisableSwitch = 0;
                    内部状態.commandQueue.delete(command);
                    // foreachループを抜ける。
                    goto exit;
                }
                break;
            }
        }
      exit:
    }

    if (CANメッセージ2.switchStatus != 内部状態.switchStatus) {
        // TODO: 開閉スイッチの状態変化に伴うイベントの生成（未仕様化）
        内部状態.switchStatus = CANメッセージ2.switchStatus;
    }
}

CANメッセージ1の送信処理()    // 周期＋イベント処理
{
    CANメッセージ1.targetPosition = 内部状態.sendCommand.targetPosition;
    CANメッセージ1.sequenceNum = 内部状態.sendSequenceNum;
    CANメッセージ1.riskPinch
            = (内部状態.sendCommand.riskPinchCode == E_OK) ? 0 : 1;
    CANメッセージ1.riskOpen
            = (内部状態.sendCommand.riskOpenCode == E_OK) ? 0 : 1;
    CANメッセージ1.enableSwitchOnSuccess = 内部状態.sendEnableSwitchOnSuccess;
    CANメッセージ1.disableSwitch = 内部状態.sendDisableSwitch;
}
```

8. ビークルコンピュータの動作ロジックの最適化

8.1. 最適化の必要性

前章の設計では，すべてのサービスコールをウィンドウ制御ECUに伝え，処理された後にリターンすることとしているが，この方法には，多数のサービスコールが並行して呼び出され

ると，CAN通信がボトルネックになり，サービスコールの実行が遅延するという問題がある。ここでは，この問題に対応するための最適化手法について検討する。

ここで検討する最適化においては，並行して呼び出されたサービスコール間の実行順序は任意で良いという仕様を活用する。

以下では，ビークルコンピュータからウィンドウ制御ECUに伝えているが，ウィンドウ制御ECUによって処理されていないコマンド（移動コマンドまたは開閉スイッチ無効化制御コマンド）を，処理待ちのコマンドと呼ぶ（図 10）。なお，コマンド発行可能状態は，処理待ちのコマンドがない状態と言い換えることもできる。

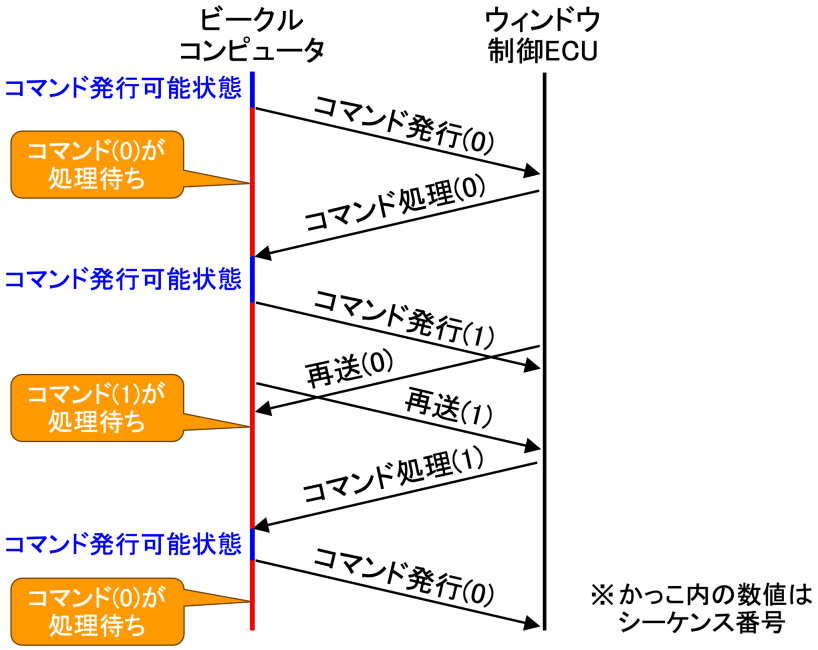

図 10. 処理待ちのコマンド

8.2. オーバライドされる移動コマンドのスキップ

startMove/stopMoveについては，サービスコールの後勝ちの原則と，並行して呼び出されたサービスコール間の実行順序は任意で良いことから，呼び出されているstartMove/stopMoveの内の任意の1つを実行すれば，それを最後に実行したものとみなすことで，他はすべて実行する必要がない（ただし，ロックの強制解除とControlledBySelfの生成は必要）。実行する必要がない移動コマンドを処理することを，移動コマンドをスキップすると言う。

ここで注意が必要なのは，ビークルコンピュータ側でコマンドを発行する時点で検出できないエラー（具体的には，リスク制御によるエラー，ウィンドウ制御ECUのダウンやCAN通信の故障によるエラー）があることである。そのため，実行を待っている移動コマンドの内の1つをウィンドウ制御ECUに発行した時点で，他の移動コマンドをスキップする処理を行うと，発行したコマンドがエラーになった場合に，どの移動コマンドも実行しなかったことになってしまう。

そのため，移動コマンドのスキップ処理は，発行した移動コマンドが正常実行されたことを確認した後，その移動コマンドによるイベントを生成する前に行う必要がある。

このことから，実行を待っている移動コマンドの中で，エラーになる可能性が低いもの（具体的には，すべてのリスクに対応できているstartMoveと，任意のstopMove）を発行する

と，移動コマンドを早くスキップできる可能性が高まる。

もう1つ注意が必要なのが，移動コマンドによりロックが強制解除される場合である。開閉スイッチの無効化を解除しない移動コマンドが処理されても，開閉スイッチの無効化を解除する移動コマンドをスキップすることはできない。そのため，開閉スイッチの無効化を解除する移動コマンドを発行すると，多くの移動コマンドをスキップできる可能性がある。

以上では，移動コマンドを早く/多くスキップできるという観点で発行するコマンドを選択する方法を示したが，別の考え方として，操作優先度の高い移動コマンドを発行する方法も有力である。操作優先度の高いものは，リスク制御が必要なく，開閉スイッチの無効化を解除する可能性も高いため，移動コマンドを早く/多くスキップすることにも役立つ。この方法は，ロジックもシンプルであるため，以下の動作ロジックでは，この方法を採用している。

移動コマンドAが正常実行された時点で行うスキップ処理では，移動コマンドAによってオーバライドされたとみなせる移動コマンドをスキップする。具体的には，移動コマンドAが正常実行された時点で，移動コマンドキュー中の移動コマンドBをスキップできないのは，移動コマンドAが開閉スイッチの無効化を解除せず，移動コマンドBが開閉スイッチの無効化を解除する場合である。これを言い換えると，開閉スイッチが無効化されており，移動コマンドAがロックを取得しているアプリケーションによって出されたものであり，移動コマンドBが他のアプリケーションによって出されたものである場合である。

なお，lock/unlockについては，呼び出したアプリケーションや操作優先度が絡むために，オーバライドの関係が単純ではない。オーバライドされるサービスコールのスキップより，後述のロックコマンドの一括処理の方が効果的と思われる。

8.3. 開閉スイッチの無効化状態を変更しないロックコマンド

開閉スイッチの無効化状態を変更しないロックコマンドは，ウィンドウ制御ECUに伝えずに実行することができる。ここでは，この性質を活用した最適化手法について検討する。

ここで注意を要するのは，開閉スイッチの無効化状態を変更しないロックコマンドであっても，処理待ちのコマンドがある場合には，すぐに実行して良いとは限らないことである。例えば，移動コマンドが処理待ちの間に，開閉スイッチの無効化状態を変更しないlockをすぐに実行すると，処理待ちの移動コマンドがエラーになるべきであるにも関わらず，実行されてしまうという問題を生じる（図 11）。これは，発行した移動コマンドを取り消すことができないためである。

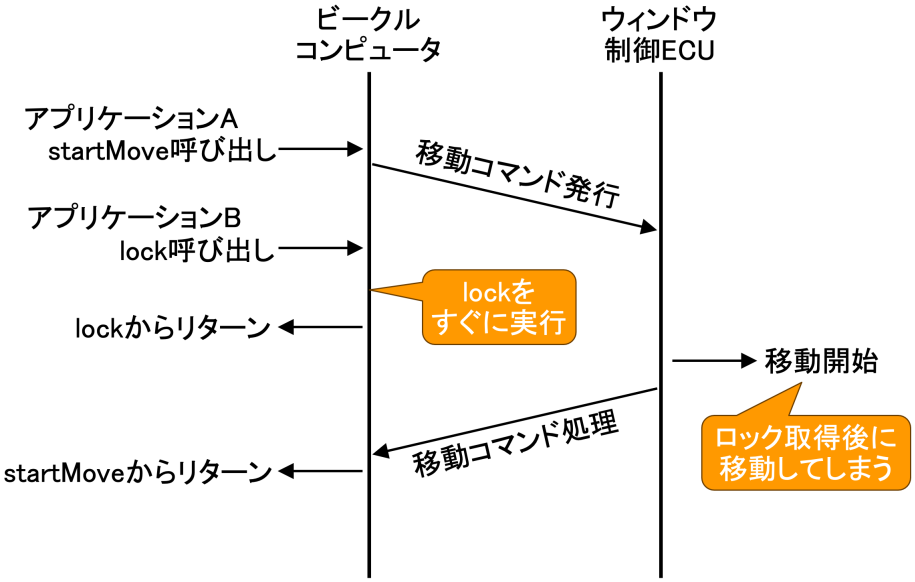

図 11. 開閉スイッチの無効化状態を変更しないlockの即時実行

そこで，開閉スイッチの無効化状態を変更しないlockについては，処理待ちのコマンドがある場合にはすぐには実行せず，コマンド発行可能状態でのみ，すぐに実行することとする。

開閉スイッチの無効化状態を変更しないunlockについては，それを実行することで，処理待ちのコマンドを処理してはならない状況になることはない。そのため，処理待ちのコマンドの有無に関わらず，すぐに実行することとする。

【補足説明】

上では，開閉スイッチの無効化状態を変更しないlockについては，処理待ちのコマンドがある場合にはすぐには実行しないこととしたが，実際には，処理待ちのコマンドの種類によっては，すぐに実行して問題ない場合もある。以下では，参考のために，処理待ちのコマンドの種類毎に，すぐに実行できるかを検討する。

- 移動コマンドが処理待ちの間に，開閉スイッチの無効化状態を変更しないlockをすぐに実行すると，図　11のような問題を生じる場合がある。ただし，移動コマンドを出したアプリケーション（図　11のアプリケーションA）と，lockを呼び出したアプリケーション（図 11のアプリケーションB）が同じ場合には，処理待ちの移動コマンドが実行されて良いため，問題を生じない。
- lock（これをlock1と呼ぶ）が処理待ちの間に，開閉スイッチの無効化状態を変更しないlock（これをlock2と呼ぶ）をすぐに実行した場合は，lock1の操作優先度はlock2より高い（lock1の操作優先度は開閉スイッチの操作優先度と同じかそれより高く，lock2の操作優先度は開閉スイッチの操作優先度より低い）ため，lock1はlock2によるロックをオーバライドすることになり，問題は生じない。
- unlockが処理待ちの間に，開閉スイッチの無効化状態を変更しないlockをすぐに実行すると，lockの実行によりロックしたアプリケーションが変わり，処理待ちのunlockがエラーになるべきであるにも関わらず，実行されてしまうという問題を生じる。

これを利用して，ビークルコンピュータ側の動作ロジックをさらに最適化する余地がある。

8.4. ロックコマンドの一括処理

開閉スイッチの無効化状態は，無効化と無効化解除の2つの状態しか取らないため，無効化状

態が変化した場合，変化前の状態で開閉スイッチの無効化状態を変更する必要があったロックコマンドは，変化後の状態では，無効化状態を変更する必要がない。そのため，無効化状態が変化した場合，その変化を起こしたロックコマンドだけでなく，コマンドキュー中のロックコマンドの多くが，すぐに実行可能となる。

そこで，開閉スイッチ無効化制御コマンドが実行され，新しい指示が発行可能になった時に，コマンドキュー中のすべてのロックコマンドの実行を試みる。コマンドキュー中のロックコマンドの多くを，ウィンドウ制御ECUに伝えずに実行できると期待できる。

8.5. コマンドの同時/独立発行

前述の通り，CANメッセージ1では，移動コマンドと開閉スイッチ無効化制御コマンドを，同時/独立に伝えることができる。これを活用するには，どのようなコマンドが同時/独立に発行できるかを検討する必要がある。

両コマンドを同時/独立に発行すると不都合が起こる例を，図　12に示す。この例では，まず，アプリケーションAがlockを呼び出し，ビークルコンピュータからウィンドウ制御ECUに開閉スイッチの無効化が指示される。その処理結果がウィンドウ制御ECUから返ってくる前に，アプリケーションBがstartMoveを呼び出し，移動コマンドと開閉スイッチの無効化指示が，同時に発行されたとする。その後，開閉スイッチの無効化指示だけが処理された結果がウィンドウ制御ECUから返ってくると，lockからリターンするが，その後でウィンドウ制御ECUで移動コマンドが実行され，ウィンドウが移動開始してしまう。この不都合の原因は，ロックによりエラーになるべき移動コマンドを，ロックと同時に発行したことである。

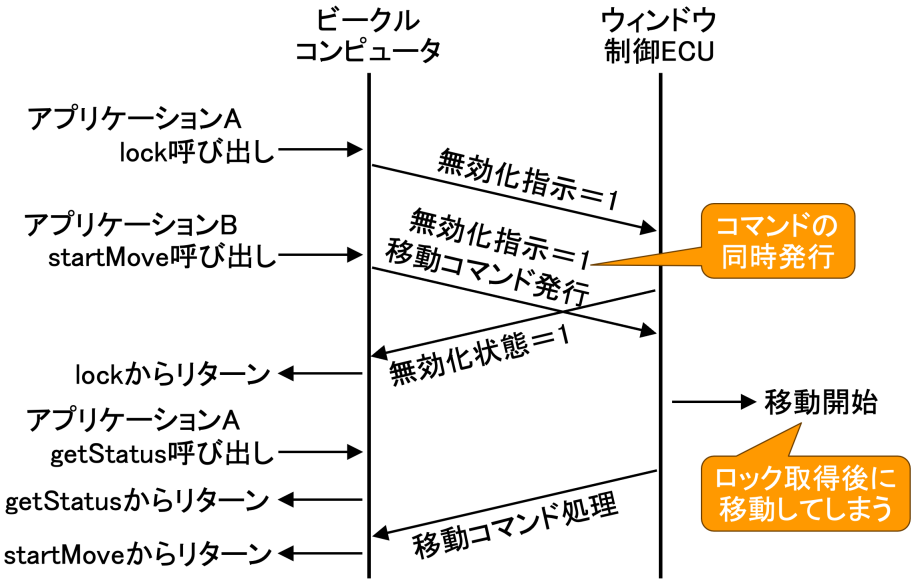

図 12. 移動コマンドと開閉スイッチ無効化制御コマンドの同時/独立発行

そのため，あるコマンドが処理待ちの間に，別種のコマンドを新たに発行して良いかを個別に検討する。

1. 移動コマンドが処理待ちの間のlockの発行

   移動コマンドの実行によりロックが解除される場合はあるが，それによってlockがエラーになる状況はないため，移動コマンドが処理待ちの間に，lockに伴う開閉スイッチの無効化指示を発行しても問題ない。

   ただし，図　11に示した通り，移動コマンドが処理待ちの間に，開閉スイッチの無効化状

態を変更しないlockを実行してはならないことに注意が必要である。この場合，開閉スイッチ無効化制御コマンドを発行する必要はないため，lockの実行を単に待たせる必要がある。

ここで，処理待ちの移動コマンドが，正常実行により開閉スイッチの無効化を解除する（CANメッセージ1のsendEnableSwitchOnSuccessを1にしている）場合の振る舞いが気になる。まず，正常実行により開閉スイッチの無効化を解除する移動コマンドが発行されるのは，その時点で開閉スイッチが無効化されている（すなわち，ロックが取得されている）場合に限られる。この状態でエラーにならないlockの呼び出しは，ロックが取得されているよりも高い操作優先度での呼び出しであり，開閉スイッチの無効化状態を変更しない。これは，図　11に示したのと同じ状況であり，lockの実行を単に待たせる必要がある。

8.3節での検討結果とあわせると，現時点の状態でエラーにならないlockは，下の表のように処理する。

処理待ちの ロックコマンド	処理待ちの 移動コマンド	lockによる 開閉スイッチの 無効化指示の要否	lockの処理方法
なし	なし	不要	すぐに実行
なし	なし	必要	開閉スイッチの 無効化指示を発行
なし	あり	不要	実行/発行を待たせる
なし	あり	必要	開閉スイッチの 無効化指示を発行
あり	—	不要	実行/発行を待たせる
あり	—	必要	実行/発行を待たせる

2. 移動コマンドが処理待ちの間のunlockの発行

   移動コマンドの実行によりロックが解除され，それによってunlockがエラーになる場合がある。しかし，移動コマンドの実行によりロックが解除された場合には，開閉スイッチの無効化は解除された状態となるため，unlockにより発行される開閉スイッチの無効化解除指示は，間違いにはならない。

   そこで，移動コマンドが処理待ちの時でも，unlockによる開閉スイッチの無効化解除指示を発行する。移動コマンドによりロックが解除された場合には，unlockをエラーリターンさせる必要がある。具体的には，CANメッセージ2受信処理内で，新たに処理された開閉スイッチ無効化制御コマンドの処理結果を取り込む時に，ロックコマンドを出したサービスコールをエラーリターンさせる必要がある場合がある。

3. lockが処理待ちの間の移動コマンドの発行

   図　12に示した通り，開閉スイッチの無効化指示が処理待ちの間に，移動コマンドの発行すると，問題を生じる場合がある。具体的には，lockを呼び出したアプリケーション（図

12のアプリケーションA）と，移動コマンドを出したアプリケーション（図　12のアプリケーションB）が異なる場合に問題が生じる。

そこで，開閉スイッチの無効化指示が処理待ちの時は，無効化指示を出したのと同じアプリケーションが移動コマンドを出した場合には移動コマンドを発行し，そうでない場合は移動コマンドの発行を待たせる。

4. unlockが処理待ちの間の移動コマンドの発行

   unlockの実行により，移動コマンドがエラーになる状況はないため，unlockが処理待ちの間に，移動コマンドを発行しても問題ない。そこで，開閉スイッチの無効化解除指示が処理待ちであっても，移動コマンドを発行する。

コマンドの同時/独立発行を実装するために，発行中のコマンドを保持する変数を，移動コマンド用とロックコマンド用で別に用意する。また，コマンドキューについても，移動コマンド用とロックコマンド用に分離する。

コマンドのデータ構造については，移動コマンドとロックコマンドでは持っている情報が異なるため，別のデータ構造とした方が効率的であるが，以下の動作ロジックでは，簡単のために，1つのデータ構造で両方のコマンドを管理するようにしている。

8.6. 内部状態

最適化した動作ロジックにおけるビークルコンピュータの内部状態は，最適化前に対して，sendCommandとcommandQueueを削除し，以下の4つを追加したものとなる。

sendMoveCommand
: 最後に発行した移動コマンド。コマンドのデータ構造中のpriorityは使用しない。

sendLockCommand
: 最後に発行した開閉スイッチ無効化制御コマンドを出したロックコマンド。コマンドのデータ構造中のriskPinchCode，riskOpenCodeは使用しない。

moveCommandQueue
: 実行を待っている移動コマンドを管理するキュー。

lockCommandQueue
: 実行を待っているロックコマンドを管理するキュー。

8.7. 動作ロジック

以下では，startMove，stopMove，lock，unlockの各サービスコールの処理ルーチン，CANメッセージ2の受信処理の動作ロジックを示す。これらの処理は，並行に呼び出されることはないと想定している（並行して呼び出される場合には，排他制御の追加が必要である）。

また，これらから呼び出される関数として，noPendingMove，noPendingDisableSwitch，canIssueMove，issueMoveCommandの動作ロジックを示す。

なお，targetPositionWhenStopped，checkMove，checkRiskは，1つのECUでの実装と

同じものを，revokeLock，checkLock，executeLock，checkUnlock，executeUnlock，CANメッセージ1の送信処理は，最適化前と同じものを使用する。

以下の動作ロジック中の「〜〜イベント（〜〜）を生成」という記述の意味は，1つのECUでの実装と同じである。

```
// 処理待ちの移動コマンドがないことの判定
noPendingMove()
{
    return 内部状態.sendSequenceNum == 内部状態.sequenceNum;
}

// 処理待ちの開閉スイッチ無効化制御コマンドがないことの判定
noPendingDisableSwitch()
{
    return 内部状態.sendDisableSwitch == 内部状態.switchDisabled;
}

// 移動コマンドを発行できる状態かの判断
//
// 移動コマンドを発行できる状態の場合にtrueを返す。noPendingMove()が
// trueの時に呼び出すことを想定している。
// 移動コマンドを発行できるのは，処理待ちの移動コマンドがなく，かつ，
// 以下のいずれかの条件が成り立つ場合である。
// ・開閉スイッチ無効化制御コマンドが処理待ちでない
// ・開閉スイッチの無効化指示が処理待ちで，無効化指示を出したのと移動
//   コマンドを出したのが同じアプリケーション
// ・開閉スイッチの無効化解除指示が処理待ち
canIssueMove(command)
{
    return noPendingDisableSwitch()
            || 内部状態.sendDisableSwitch == 0
            || 内部状態.sendLockCommand.application == command.application;
}

// 移動コマンドの発行
issueMoveCommand(command, lockRevoke)
{
    内部状態.sendMoveCommand = command;
    内部状態.sendSequenceNum = 1 - 内部状態.sendSequenceNum;
    if (lockRevoke && 内部状態.switchDisabled == 1) {
        内部状態.sendEnableSwitchOnSuccess = 1;
    }
}

// startMoveサービスコール
startMove(position, priority)
{
    // command, retval, lockRevokeはローカル変数
    retval = checkMove(呼び出したアプリケーションID, priority, &lockRevoke);
```

```
    if (retval != E_OK) {
        return retval;
    }

    // コマンドのデータ構造の作成
    commandのメモリ領域を確保する
    command.targetPosition = position;
    command.application = 呼び出したアプリケーションID;
    command.priority = priority;
    command.riskPinchCode = checkRisk(RiskPinch);
    command.riskOpenCode = checkRisk(RiskOpen);

    if (noPendingMove() && canIssueMove(command)) {
        issueMoveCommand(command, lockRevoke);
        コマンドが処理されるのを待ち，処理結果をretvalに格納
    }
    else {
        内部状態.moveCommandQueue.enqueue(command);
        コマンドが処理されるのを待ち，処理結果をretvalに格納
    }
    return retval;
}

// stopMoveサービスコール
stopMove(priority)
{
    // command, retval, lockRevokeはローカル変数
    retval = checkMove(呼び出したアプリケーションID, priority, &lockRevoke);
    if (retval != E_OK) {
        return retval;
    }

    // コマンドのデータ構造の作成
    commandのメモリ領域を確保する
    command.targetPosition = COMMAND_STOP;
    command.application = 呼び出したアプリケーションID;
    command.priority = priority;
    command.riskPinchCode = E_OK;
    command.riskOpenCode = E_OK;

    if (noPendingMove() && canIssueMove(command)) {
        issueMoveCommand(command, lockRevoke);
        コマンドが処理されるのを待ち，処理結果をretvalに格納
    }
    else {
        内部状態.moveCommandQueue.enqueue(command);
        コマンドが処理されるのを待ち，処理結果をretvalに格納
    }
    return retval;
}
```

```
// lockサービスコール
lock(priority)
{
    // command, retval, disableSwitchはローカル変数
    retval = checkLock(呼び出したアプリケーションID, priority, &disableSwitch);
    if (retval != E_OK) {
        return retval;
    }

    // コマンドのデータ構造の作成
    commandのメモリ領域を確保する
    command.targetPosition = COMMAND_LOCK;
    command.application = 呼び出したアプリケーションID;
    command.priority = priority;

    if (noPendingDisableSwitch()) {
        if (!disableSwitch && noPendingMove()) {
            executeLock(呼び出したアプリケーションID, priority);
            retval = E_OK;
        }
        else if (disableSwitch) {
            内部状態.sendLockCommand = command;
            内部状態.sendDisableSwitch = 1;
            コマンドが処理されるのを待ち，処理結果をretvalに格納
        }
        else {
            内部状態.lockCommandQueue.enqueue(command);
            コマンドが処理されるのを待ち，処理結果をretvalに格納
        }
    }
    else {
        内部状態.lockCommandQueue.enqueue(command);
        コマンドが処理されるのを待ち，処理結果をretvalに格納
    }
    return retval;
}

// unlockサービスコール
unlock()
{
    // command, retval, enableSwitchはローカル変数
    retval = checkUnlock(呼び出したアプリケーションID, &enableSwitch);
    if (retval != E_OK) {
        return retval;
    }

    // コマンドのデータ構造の作成
    commandのメモリ領域を確保する
    command.targetPosition = COMMAND_UNLOCK;
```

```
    command.application = 呼び出したアプリケーションID;
    command.priority = PRIORITY_NULL;

    if (!enableSwitch) {
        executeUnlock(呼び出したアプリケーションID);
        retval = E_OK;
    }
    else if (noPendingDisableSwitch()) {
        内部状態.sendLockCommand = command;
        内部状態.sendDisableSwitch = 0;
        コマンドが処理されるのを待ち，処理結果をretvalに格納
    }
    else {
        内部状態.lockCommandQueue.enqueue(command);
        コマンドが処理されるのを待ち，処理結果をretvalに格納
    }
    return retval;
}

CANメッセージ2の受信処理(CANメッセージ2)
{
    // moveCommandExecutedはローカル変数
    moveCommandExecuted = false;

    if (CANメッセージ2.sequenceNum != 内部状態.sequenceNum) {
        // 発行している移動コマンドが新たに処理された。
        // RiskPinchとRiskOpenの両方で拒否されることはないため，どちら
        // を先にチェックしても良い。
        if (CANメッセージ2.denyRiskPinch == 1) {
            内部状態.sendMoveCommandを出したサービスコールに，
                内部状態.sendMoveCommand.riskPinchCodeを送り，リターンさせる
        }
        else if (CANメッセージ2.denyRiskOpen == 1) {
            内部状態.sendMoveCommandを出したサービスコールに，
                内部状態.sendMoveCommand.riskOpenCodeを送り，リターンさせる
        }
        else {
            moveCommandExecuted = true;
            内部状態.currentApplication = 内部状態.sendMoveCommand.application;

            // オーバライドされたとみなせる移動コマンドのスキップ
            foreach command in 内部状態.moveCommandQueue {
                // 移動コマンドをスキップできないのは，次の3つの条件を
                // 満たす場合。
                // (1) 開閉スイッチが無効化されている。
                // (2) 移動コマンドAがロックを取得しているアプリケーショ
                //     ンによって出されたものである。
                // (3) 移動コマンドBがロックを取得していないアプリケー
                //     ションによって出されたものである。
                if (!(内部状態.switchDisabled == 1
```

```
                && 内部状態.lockApplication == 内部状態.currentApplication
                && 内部状態.lockApplication != command.application)) {
                // commandがスキップ可能
                if (内部状態.locked && 内部状態.lockApplication
                                        != command.application) {
                    revokeLock(true);
                }
                ControlledBySelfイベント（command.application）を生成
                commandを出したサービスコールに，E_OKを送り，リターンさせる
                内部状態.moveCommandQueue.delete(command);
            }
        }

        // 移動コマンドの処理結果の取り込み
        if (内部状態.locked && 内部状態.lockApplication
                                != 内部状態.currentApplication) {
            revokeLock(true);
        }
        if (内部状態.sendMoveCommand.targetPosition != COMMAND_STOP) {
            内部状態.targetPosition = 内部状態.sendMoveCommand.targetPosition;
        }
        else {
            内部状態.targetPosition
                    = targetPositionWhenStopped(CANメッセージ2.position);
        }
        if (内部状態.sendEnableSwitchOnSuccess == 1) {
            // 以下のassertが成り立たなくなるのは，ウィンドウ制御
            // ECUが，移動コマンドをアトミックに実行しなかった場合。
            assert(CANメッセージ2.switchDisabled == 0);
            内部状態.sendEnableSwitchOnSuccess = 0;
        }
        ControlledBySelfイベント（内部状態.currentApplication）を生成
        内部状態.sendMoveCommandを出したサービスコールに，
                                        E_OKを送り，リターンさせる
    }
    内部状態.sequenceNum = CANメッセージ2.sequenceNum;
}

// retval, disableSwitch, enableSwitchはローカル変数
if (CANメッセージ2.switchDisabled != 内部状態.switchDisabled) {
    // 開閉スイッチ無効化制御コマンドが新たに処理された。
    switch (内部状態.sendLockCommand.targetPosition) {
    case COMMAND_LOCK:
        assert(CANメッセージ2.switchDisabled == 1);
        retval = checkLock(内部状態.sendLockCommand.application,
                                内部状態.sendLockCommand.priority,
                                &disableSwitch);
        if (retval == E_OK) {
            executeLock(内部状態.sendLockCommand.application,
                                内部状態.sendLockCommand.priority);
```

```
            }
            内部状態.sendLockCommandを出したサービスコールに，
                                        retvalを送り，リターンさせる
            内部状態.sendLockCommand.targetPosition = COMMAND_NULL;
            break;
        case COMMAND_UNLOCK:
            assert(CANメッセージ2.switchDisabled == 0);
            retval = checkUnlock(内部状態.sendLockCommand.application,
                                        &enableSwitch);
            if (retval == E_OK) {
                executeUnlock(内部状態.sendLockCommand.application);
            }
            内部状態.sendLockCommandを出したサービスコールに，
                                        retvalを送り，リターンさせる
            内部状態.sendLockCommand.targetPosition = COMMAND_NULL;
            break;
        default:
            // 移動コマンドや異常状態への遷移によって，開閉スイッチの
            // 無効化が解除された。
            assert(CANメッセージ2.switchDisabled == 0);
            内部状態.sendDisableSwitch = 0;
            break;
        }
        内部状態.switchDisabled = CANメッセージ2.switchDisabled;
    }

    if (moveCommandExecuted
            || CANメッセージ2.extendedStatus != 内部状態.extendedStatus
            || CANメッセージ2.commandSource != 内部状態.commandSource) {
        // ウィンドウ制御ECUの状態が変化
        switch (CANメッセージ2.extendedStatus) {
        case TargetReached:
            内部状態.mainStatus = Stopped;
            if (CANメッセージ2.commandSource == 0) {
                TargetReached（内部状態.currentApplication）イベントを生成
            }
            else {
                if (内部状態.locked) {
                    revokeLock(true);
                }
                TargetReached（開閉スイッチによる制御のID）イベントを生成
            }
            内部状態.targetPosition
                    = targetPositionWhenStopped(CANメッセージ2.position);
            break;
        case FaultRecovered:
            内部状態.mainStatus = Stopped;
            if (内部状態.locked) {
                revokeLock(false);
            }
```

```
            OtherLockReleasedイベント（NULL）を生成
            FaultRecoveredイベントを生成
            内部状態.targetPosition
                    = targetPositionWhenStopped(CANメッセージ2.position);
            break;
        case PinchRecovered:
            内部状態.mainStatus = Stopped;
            if (内部状態.locked) {
                revokeLock(false);
            }
            OtherLockReleasedイベント（NULL）を生成
            PinchRecoveredイベントを生成
            内部状態.targetPosition
                    = targetPositionWhenStopped(CANメッセージ2.position);
            break;

        case Opening:
            内部状態.mainStatus = Opening;
            if (CANメッセージ2.commandSource == 0) {
                // ロックの強制解除と内部状態.targetPositionの更新は,
                // 移動コマンドの処理結果の取り込みで行われているため,
                // ここで行う必要はない。
                ControlledByOtherイベント（内部状態.currentApplication）を生成
            }
            else {
                if (内部状態.locked) {
                    revokeLock(true);
                }
                ControlledByOtherイベント（開閉スイッチによる制御のID）を生成
                内部状態.targetPosition = 100;
            }
            break;

        case Closing:
            内部状態.mainStatus = Closing;
            if (CANメッセージ2.commandSource == 0) {
                // ロックの強制解除と内部状態.targetPositionの更新は,
                // 移動コマンドの処理結果の取り込みで行われているため,
                // ここで行う必要はない。
                ControlledByOtherイベント（内部状態.currentApplication）を生成
            }
            else {
                if (内部状態.locked) {
                    revokeLock(true);
                }
                ControlledByOtherイベント（開閉スイッチによる制御のID）を生成
                内部状態.targetPosition = 0;
            }
            break;
```

```
        case Fault:
            内部状態.mainStatus = Fault;
            if (内部状態.locked) {
                revokeLock(false);
            }
            FaultDetectedイベントを生成
            内部状態.targetPosition = UNKNOWN;
            break;

        case PinchAvoiding:
            内部状態.mainStatus = PinchAvoiding;
            if (内部状態.locked) {
                revokeLock(false);
            }
            PinchDetectedイベントを生成
            内部状態.targetPosition = UNKNOWN;
            break;
        }
        内部状態.extendedStatus = CANメッセージ2.extendedStatus;
        内部状態.commandSource = CANメッセージ2.commandSource;
        内部状態.position = CANメッセージ2.position;
    }
    else if (CANメッセージ2.position != 内部状態.position) {
        // 現在位置のみが変化
        if ((内部状態.extendedStatus == TargetReached
                        || 内部状態.extendedStatus == FaultRecovered
                        || 内部状態.extendedStatus == PinchRecovered)
                    && 内部状態.position != UNKNOWN
                    && 内部状態.position != NONZERO) {
            // 停止中に現在位置が動いた場合は，一瞬故障したものと扱う。
            if (内部状態.locked) {
                revokeLock(false);
            }
            // FaultDetectedイベントの生成は省くことができる。
            OtherLockReleasedイベント（NULL）を生成
            FaultRecoveredイベントを生成
            内部状態.targetPosition
                    = targetPositionWhenStopped(CANメッセージ2.position);
            // 内部状態.extendedStatusは，アプリケーションからは参照で
            // きない状態なので，ここで更新する必要はない。
        }
        内部状態.position = CANメッセージ2.position;
    }

    // command, retval, lockRevoke, selectedCommand, disableSwitch,
    // enableSwitchはローカル変数
    if (noPendingDisableSwitch()) {
        // ロックコマンドキュー中のロックコマンドの中で，開閉スイッチ
        // の無効化状態を変化させないものをすぐに実行し，実行できない
        // ものの中で，発行すべきロックコマンドを決定する。
```

```
        selectedCommand = NULL;
        foreach command in 内部状態.lockCommandQueue {
            switch (command.targetPosition) {
            case COMMAND_LOCK:
                retval = checkLock(command.application,
                                        command.priority, &disableSwitch);
                if (retval != E_OK) {
                    commandを出したサービスコールに，retvalを送り，リターンさせる
                    内部状態.lockCommandQueue.delete(command);
                }
                else if (!disableSwitch && noPendingMove()) {
                    executeLock(command.application, command.priority);
                    commandを出したサービスコールに，E_OKを送り，リターンさせる
                    内部状態.lockCommandQueue.delete(command);
                }
                else if (disableSwitch) {
                    if (selectedCommand == NULL
                            || command.priority < selectedCommand.priority) {
                        selectedCommand = command;
                    }
                }
                break;
            case COMMAND_UNLOCK:
                retval = checkUnlock(command.application, &enableSwitch);
                if (retval != E_OK) {
                    commandを出したサービスコールに，retvalを送り，リターンさせる
                    内部状態.lockCommandQueue.delete(command);
                }
                else if (!enableSwitch) {
                    executeUnlock(command.application);
                    commandを出したサービスコールに，E_OKを送り，リターンさせる
                    内部状態.lockCommandQueue.delete(command);
                }
                else {
                    if (selectedCommand == NULL) {
                        selectedCommand = command;
                    }
                }
                break;
            }
        }
        // ロックコマンドの発行
        if (selectedCommand != NULL) {
            内部状態.sendLockCommand = selectedCommand;
            switch (selectedCommand.targetPosition) {
            case COMMAND_LOCK:
                内部状態.sendDisableSwitch = 1;
                break;
            case COMMAND_UNLOCK:
                内部状態.sendDisableSwitch = 0;
```

```
                break;
            }
            内部状態.lockCommandQueue.delete(selectedCommand);
        }
    }

    // command, retval, lockRevoke, selectedCommand,
    // selectedCommandLockRevokeはローカル変数
    if (noPendingMove()) {
        // 発行すべき移動コマンドの決定
        selectedCommand = NULL;
        foreach command in 内部状態.moveCommandQueue {
            retval = checkMove(command.application,
                               command.priority, &lockRevoke);
            if (retval != E_OK) {
                commandを出したサービスコールに，retvalを送り，リターンさせる
                内部状態.moveCommandQueue.delete(command);
            }
            else if (noPendingMove() && canIssueMove(command)) {
                if (selectedCommand == NULL
                        || command.priority < selectedCommand.priority) {
                    selectedCommand = command;
                    selectedCommandLockRevoke = lockRevoke;
                }
            }
        }
        // 移動コマンドの発行
        if (selectedCommand != NULL) {
            issueMoveCommand(selectedCommand, selectedCommandLockRevoke);
            内部状態.moveCommandQueue.delete(selectedCommand);
        }
    }

    if (CANメッセージ2.switchStatus != 内部状態.switchStatus) {
        // TODO: 開閉スイッチの状態変化に伴うイベントの生成（未仕様化）
        内部状態.switchStatus = CANメッセージ2.switchStatus;
    }
}
```

付録 A: バージョン履歴

2025年11月19日　最初のリリース版（リリース日は21日）

2025年12月25日　仕様変更に対応した版

- 1つのECUでの実装を追加。章構成を変更
- Open SDV API仕様の変更に対応
- 考慮不足による不具合等を修正

- コマンドキューを2つに分割して最適化を実施

2026年1月7日　説明会に向けて改訂した版

- ネーミングコンベンションの変更に対応
- 設計の説明を改善。ドキュメントの構成を変更
- 動作開始/停止条件の判定を関数に

2026年1月18日　説明会でのコメントを受けて改訂した版

- TargetReachedに関連情報を追加するなどの仕様変更に対応
- CANメッセージ2の主状態と状態変化の理由を，拡張状態に統合
- 最適化前後のビークルコンピュータの動作ロジックを示すように変更
- コマンドの同時/独立発行について整理・実装
- オーバライドされるコマンドのスキップを実装
- 設計の説明を改善/追加

2026年1月28日　一応の完成版

- ロックの強制解除条件に関する不具合を修正
- ロックコマンドの処理方法を改善
- 移動コマンドの同時発行条件を拡大
- 設計の説明を改善/追加

2026年3月27日　公表版

- revokeLockの仕様変更
- 設計の説明を改善