RT C言語コントローラ

磁気カードリーダーサンプル

目　次

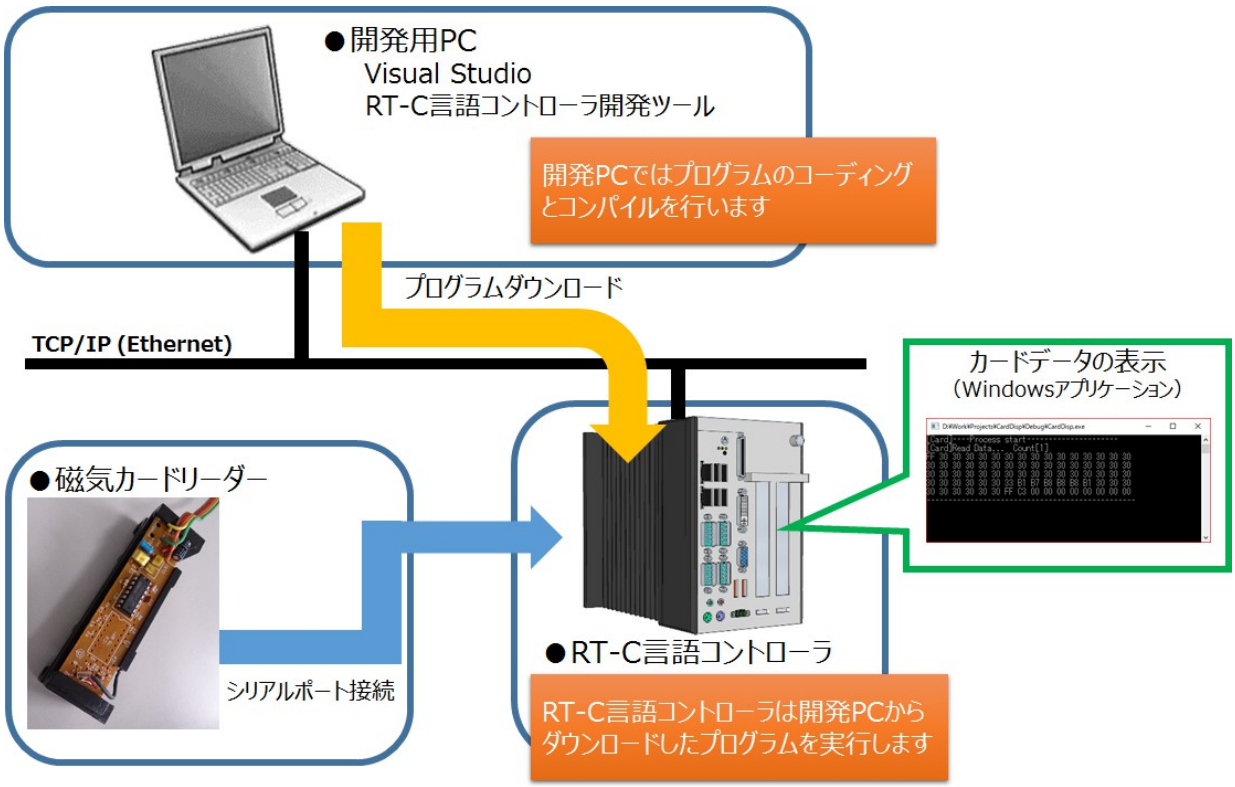
📖 本書では原則として、開発には「Visual Studio 2010」と「INplc-SDK(Express)」を利用した、Windows 10 をベースとする操作手順および画像を使用しています。お使いのソフトウェア・Windows のバージョン等により、記載されている操作手順や画像などが異なる場合がありますので、適時読み替えてご利用ください。

マイクロネット

1. 概要

1.1. システム構成

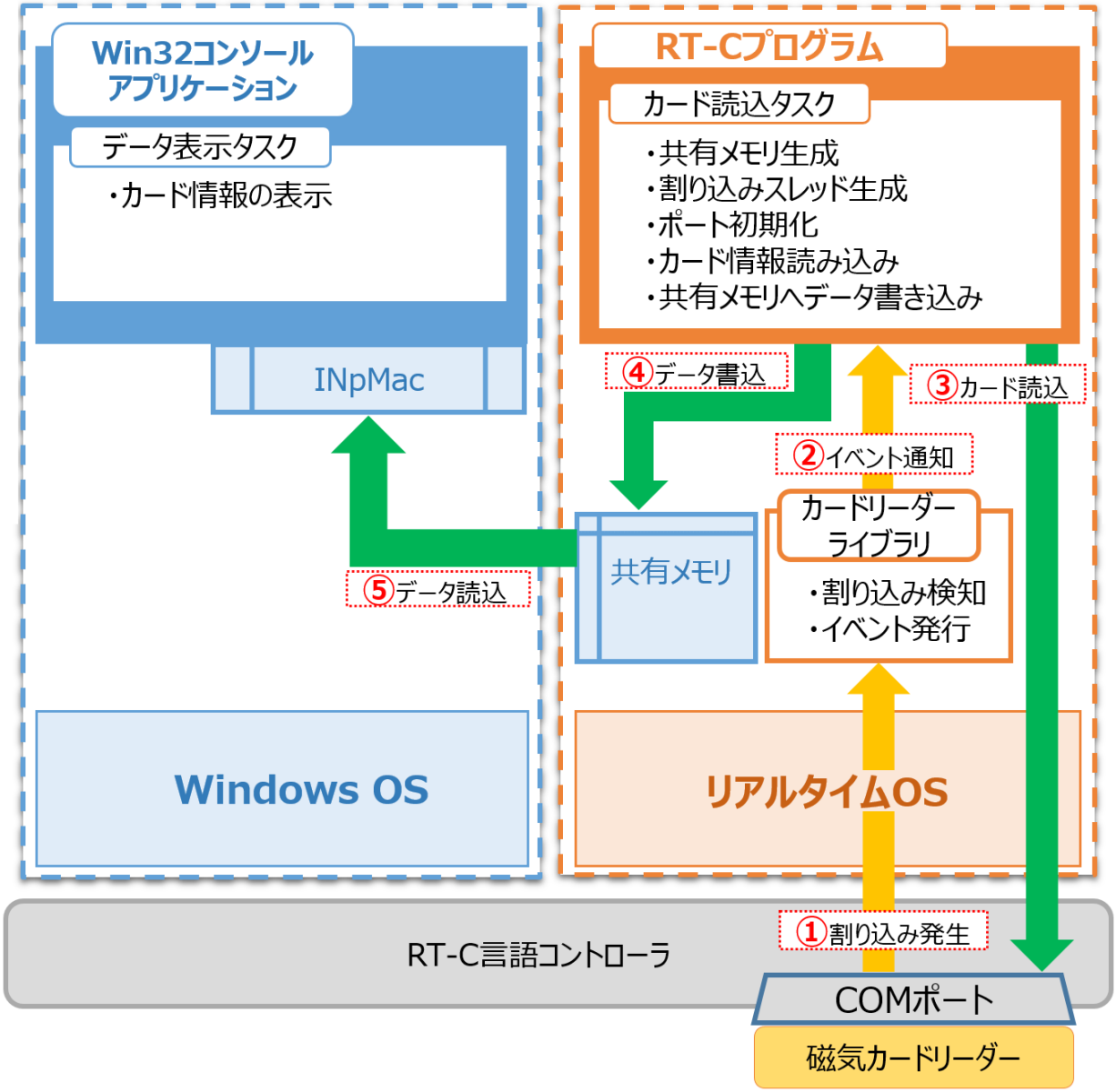
本書で説明するシステムの全体構成を下記に示します。
磁気カードリーダーからカード情報を読み込み、画面に表示する処理を行います。



1.2. プログラム構成

本サンプルプログラムでは、磁気カードの走行開始時にカードリーダーライブラリが提供する割り込みハンドラ・割り込みスレッドから、カード読込タスクへイベント通知することでカード情報の読込処理を開始します。

本書で作成するシステムのプログラム構成を下記に示します。



■ 処理の流れ

① シリアルポートでの割込み発生を割り込みハンドラが検知します。

② 割込みスレッドから RT-C プログラムへイベント通知します。

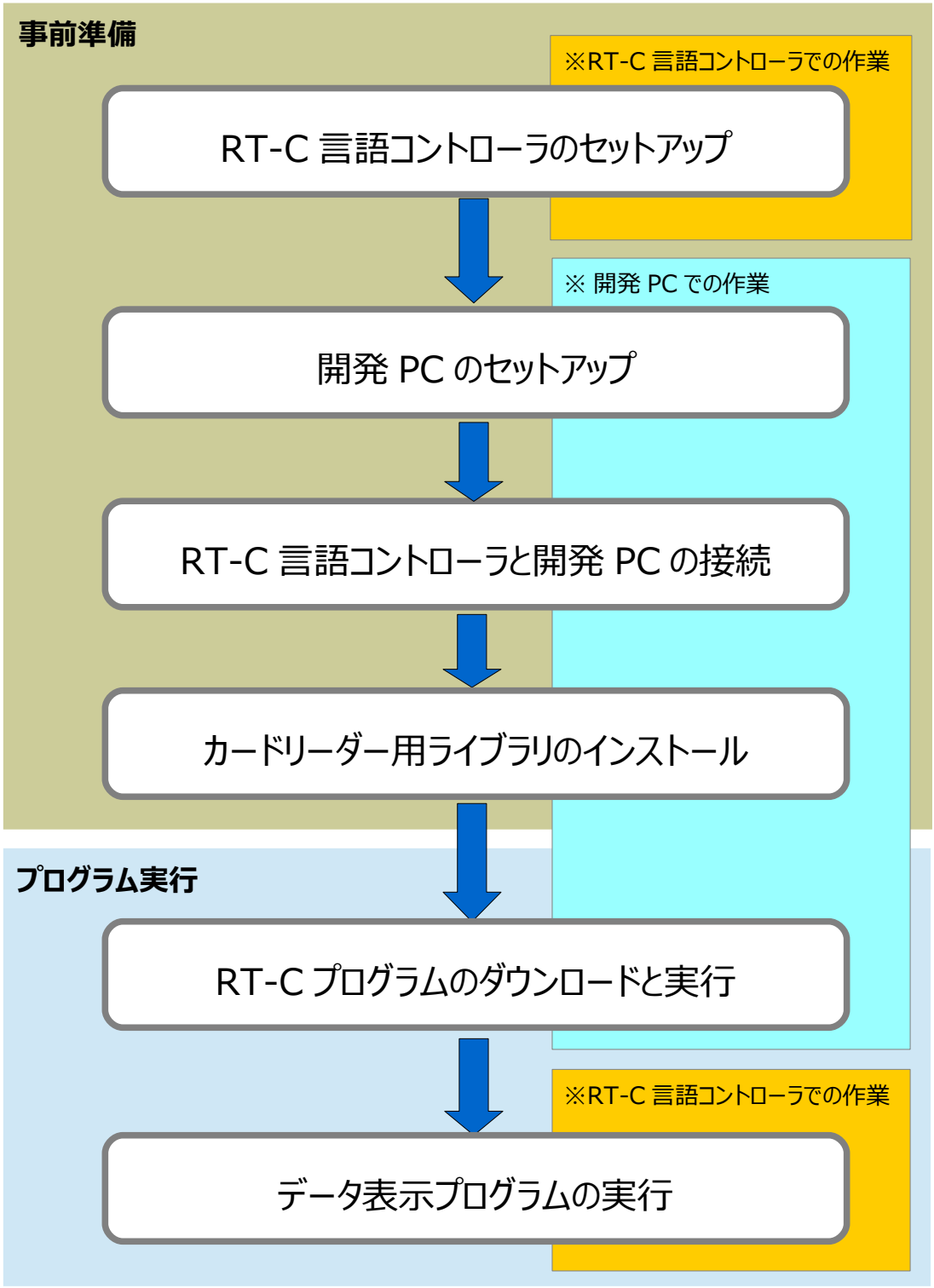
③ イベント通知を受けてカード読込タスクが動作します。
カード読込タスクは、シリアルポートのレジスタへアクセスし、カード情報を取得します。
※カード情報の取得タイミングについては、カードリーダーのタイミングチャートを参照してください。

④ 取得したデータを共有メモリへ書き込みます。

⑤ Windows 側コンソールアプリケーションが共有メモリを参照し、データをコンソール上に出力します。
※Windows 側コンソールアプリケーションは非同期で動作し、共有メモリをポーリング(200ms 周期)します。

2.　サンプルプログラムの実行

早速、サンプルプログラムを動かしてみましょう。
下記の流れでプログラムの動作に必要な事前準備を行った後、プログラムを実行します。

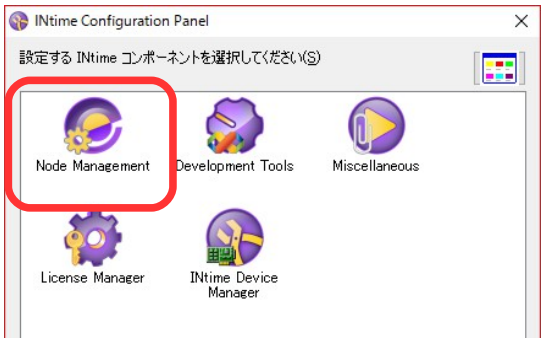


2.1. RT-C 言語コントローラのセットアップ

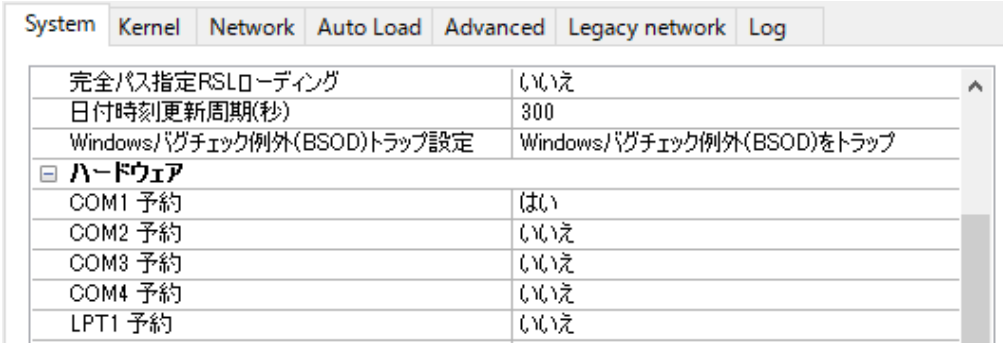
本プログラムの実行に必要な設定、およびツールのインストールを行います。

2.1.1. COM ポートの設定変更

(1) Windows のコントロールパネルをアイコン表示にし、一覧から「INtime Configuration Panel」を選択して設定画面を表示後、「INtime Node Manager」のアイコンをダブルクリックします。



(2) INtime Node Manager 画面の「System」タブの「ハードウェア」項目にある「COM1 予約」を【はい】にしてください。



(3) 「保存」を押下し、「閉じる」で設定を終了します。

(4) COM1 ポートにカードリーダーを接続してください。

2.1.2. INpMac(実行環境)のインストール

INpMac のインストーラ(INpMacSetup.EXE)を実行し、RT-C 言語コントローラに INpMac 実行環境(Runtime)をインストールしてください。

※インストーラは、「INplc-SDK」CD の下記フォルダにあります。

[INplc-SDK CD-ROM]:¥Util¥INpMac¥INpMacSetup.EXE

2.1.3. データ表示プログラムの実行ファイル配置

カード情報を Windows 画面に表示するデータ表示プログラム(Win32 コンソールアプリケーション)の実行ファイルを RT-C 言語コントローラの任意の場所にコピーしてください。

※実行ファイルは、RTC_CARD_Sample.zip の下記フォルダにあります。

[zip]:¥SampleProgram¥CardDisp¥Release¥CardDisp.exe

2.2. 開発用 PC のセットアップ

お手持ちの PC へ RT-C 言語コントローラの開発環境を構築します。

2.2.1. Visual Studio のインストール

Visual Studio をインストールしてください。
インストール後、Visual Studio を起動して開発環境の初期設定を行ってください。
※ Blend for Visual Studio は使用できません。

2.2.2. RT-C 言語コントローラ開発ツールのインストール

コントローラ付属の開発ツール「INplc-SDK」をインストールしてください。
※ インストール方法は、「INplc-SDK」CD 内にある「INplc-SDK セットアップ手順書」を参照してください。

2.2.3. INpMac(開発環境)のインストール

INpMac のインストールを行います。
INpMac のインストーラ(INpMacSetup.EXE)を実行し、開発用 PC へ INpMac 開発環境(Develop)をインストールしてください。
※インストーラは、「INplc-SDK」CD の下記フォルダにあります。
[INplc-SDK CD-ROM]:¥Util¥INpMac¥INpMacSetup.EXE

2.2.4. カードリーダーライブラリの配置

本プログラム専用のカードリーダーライブラリを開発用 PC 内に配置します。
RTC_CARD_Sample.zip 内にある Setup.exe を実行してください。必要なファイルが開発用 PC へ配置されます。

2.2.5. RT-C プロジェクトファイルの配置

RT-C プロジェクトファイル(RTC_CardReader.zwe)を開発用 PC の任意の場所にコピーしてください。
※RT-C プロジェクトファイルは、RTC_CARD_Sample.zip の下記フォルダにあります。
[zip]:¥SampleProgram¥RTC_CardReader.zwe

2.2.6. 開発用 PC と RT-C 言語コントローラの接続

「RT-C ユーザーズマニュアル」を参照し、開発用 PC と RT-C 言語コントローラを接続してください。

2.3. カードリーダーライブラリのインストール

開発用 PC と RT-C 言語コントローラが接続された状態でカードリーダーライブラリ(RTC_CARD.fwpack)を RT-C 言語コントローラにインストールします。

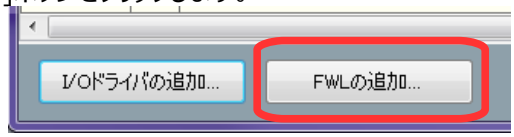
(1) 開発用 PC のスタートメニューの「すべてのプログラム」から[Micronet]>[INplc]>[INplc Configuration Tool]をクリックします。

(2) INplc コントローラ選択ダイアログが表示されるので、一覧から RT-C 言語コントローラを選択して［接続］ボタンをクリックします。

※ 環境によっては、このダイアログが表示されない場合があります。

(3) INplc Configuration Tool の[IO・FWL 起動設定] アイコンをダブルクリックします。

(4) 画面下部の[FWL の追加]ボタンをクリックします。



(5) fwpack の選択ダイアログが表示されるので、RT-C の LIB フォルダにある「**RTC_CARD.fwpack**」を選択して［開く］ボタンをクリックします。

※ RT-C の LIB フォルダ： 32bit OS【C:¥Program Filse¥Micronet¥RT-C¥LIB¥】
64bit OS【C:¥Program Filse (x86)¥Micronet¥RT-C¥LIB¥】

(6) ライブラリのインストールが完了するとメッセージが表示されるので［OK］ボタンをクリックします。

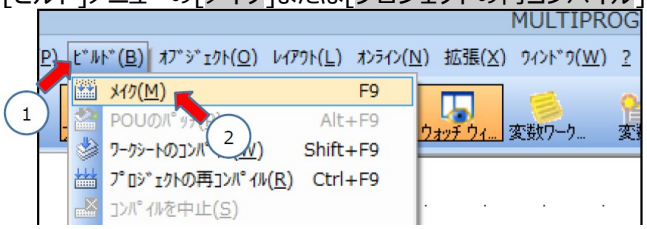
(7) 一覧に「**RT-C Card Reader API Library**」が追加されるので、［Use］列のチェックを ON にしてください。

(8) ［保存］ボタンをクリックして、表示されるメッセージに従い、コントローラを再起動してください。

2.4. RT-Cプログラムのダウンロードと実行

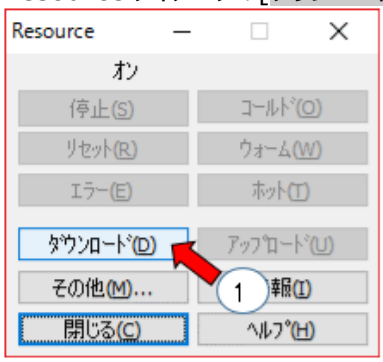
(1) 任意の場所にコピーしたRT-Cプロジェクトファイル「RTC_CardReader.zwe」を解凍します。
「RTC_CardReader.zwe」ダブルクリックすることでRT-C言語コントローラ開発ツール「MULTIPROG」が起動するので、表示されるメッセージに従いプロジェクトを解凍してください。

(2) 解凍したプロジェクトをビルドします。
[ビルド]メニューの[メイク]または[プロジェクトの再コンパイル]を選択します。

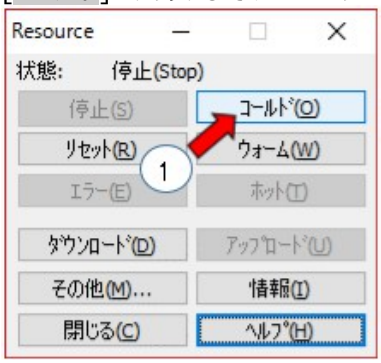


(3) プロジェクトコントロールを起動します。
[オンライン]メニューの[プロジェクトコントロール]を選択します。

(4) RT-C言語コントローラへプログラムをダウンロードします。
Resourceダイアログの[ダウンロード]をクリックしてください。



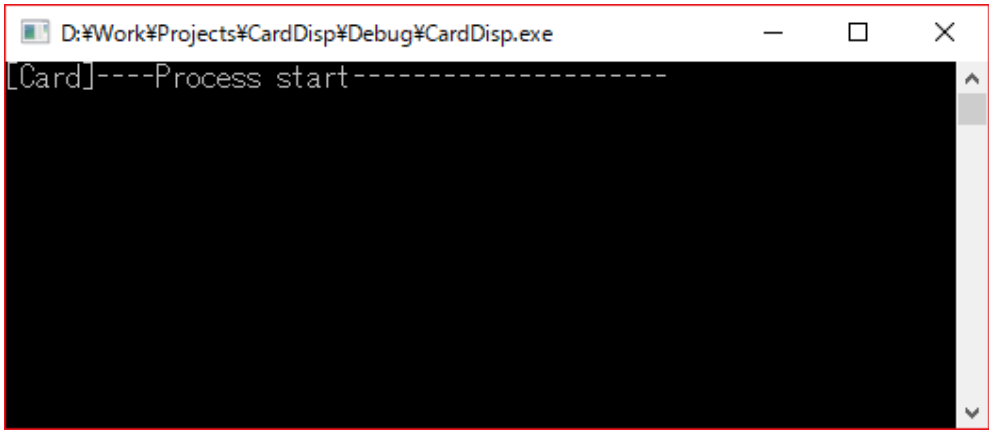
(5) プログラムを実行します。
[コールド]をクリックしてください。プログラムがRT-C言語コントローラ上で実行されます。



2.5. データ表示プログラムの実行

読み取ったデータをコンソールに表示するため、データ表示プログラムを実行します。
RT-C 言語コントローラ上に配置した CardDisp.exe をダブルクリックしてください。

正常に起動すると下記のようにコンソールウィンドウが表示されます。



以上で、プログラムの実行手順は終了です。

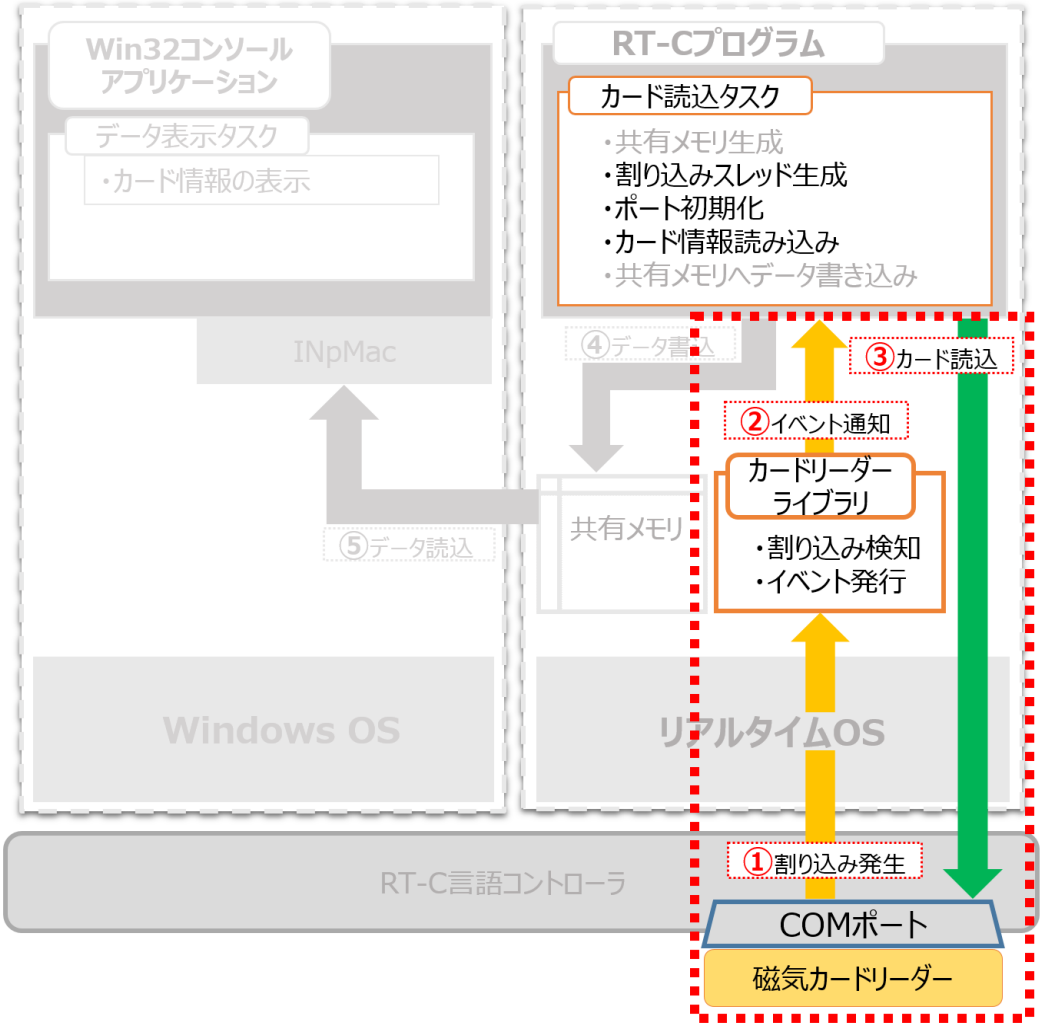
早速、磁気カードをカードリーダーに通してみましょう。
カード情報が読み込まれるとコンソールに下記のように出力されます。

※上記は、JIS-2 型磁気カードの一般的なデータフォーマットです。
※本プログラムでは、データのエラーチェック(パリティチェック等)は行っていません。

カード情報が表示されない場合、P.22「トラブルシューティング」を参照してください。

3.　磁気カードリーダーについて

本プログラムで使用する磁気カードリーダーの仕様、および下図に示すデータ取得処理について説明します。



3.1. 磁気カードリーダー仕様

メーカー	NECトーキン(株)	型式	MCS-1303P-1
カードデータ種	JIS-Ⅱ型	詳細	記録密度：210BPI ビット構成：8BIT/キャラクタ データ量：72 キャラクタ
電源	DC 5 V	詳細	DC 3～5 V
インターフェース	No	信号名称	信号内容
	1	RDD	読出データ *1
	2	RCP	データサンプリング用パルス *1
	3	CLS	カード走行信号 *1
	4	Vcc	電源電圧
	5	GND	グランド

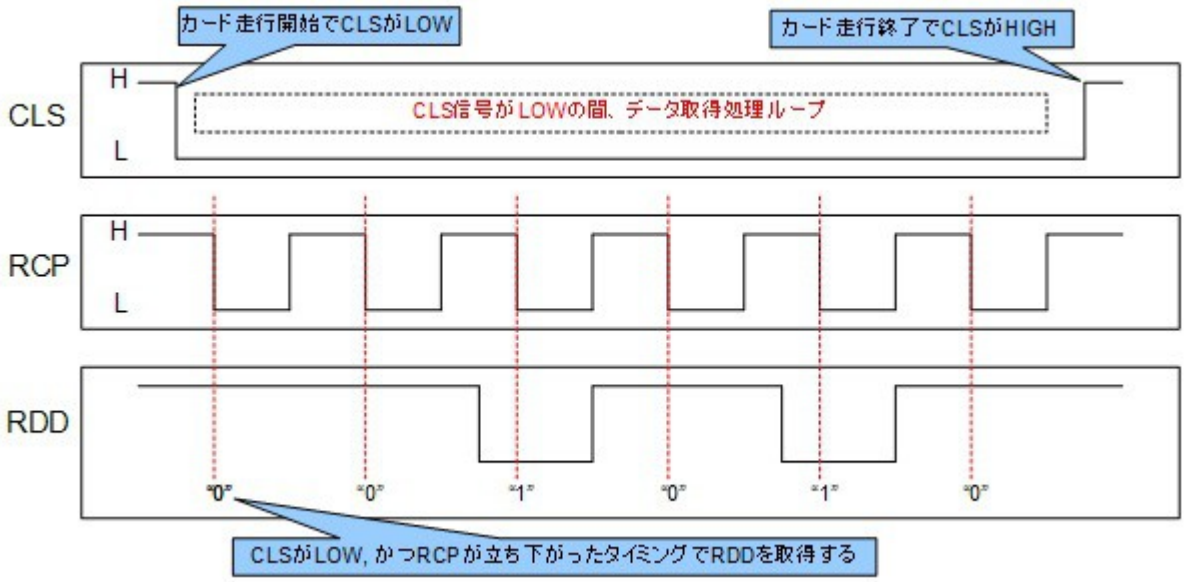
*1 出力は、負論理となります。

3.2. 出力信号

No	信号名称	内容	備考
1	CLS (Card Loading Signal)	カード走行信号	カード走行中に出力がLowとなります
2	RCP (Reading Clock Pulse)	サンプリングパルス	1bit分のデータを読み取ったタイミングでLowとなります
3	RDD (Reading Decode Data)	読み出しデータ	1bit分のデータを表します(High:0, Low:1)

3.3. タイミングチャート

磁気カードリーダーから出力される信号のタイミングチャートを下記に示します。



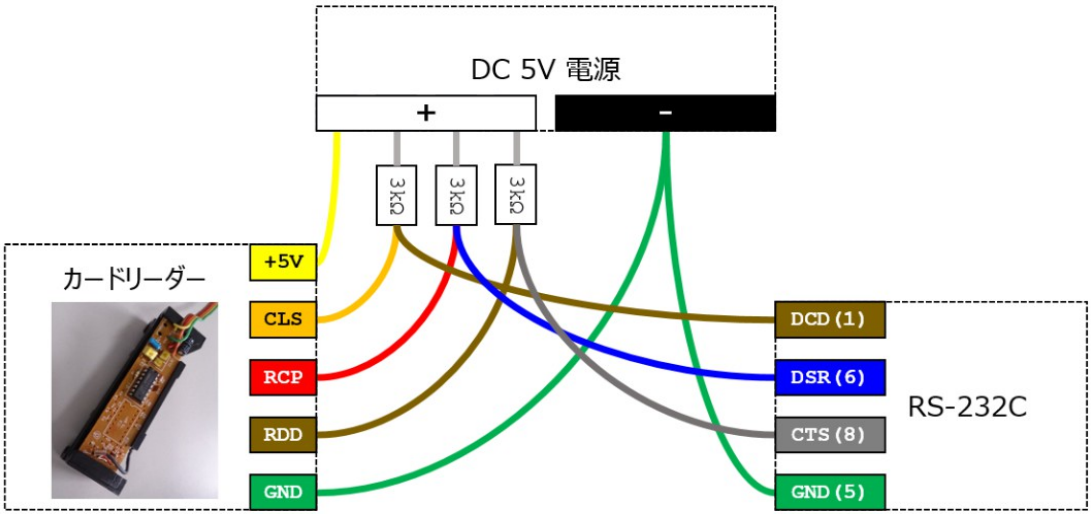
3.4. 信号割付

本プログラムでは、シリアルポートの制御ピンを利用して、カードリーダーからの出力信号を取得します。
シリアルポートに対するカードリーダー信号の割り付けを下記に示します。（使用する Pin のみ記載）

RS232C(D-Sub 9ピン)				カードリーダー	
Pin No.	**信号名**	**I/O**	**内容**	**信号名**	**内容**
1	DCD	In	キャリア検出	CLS	カード走行信号
5	GND	-	グランド		
6	DSR	In	データ・セット・レディ	RCP	サンプリング周期信号
8	CTS	In	送信可	RDD	データ読み出し信号

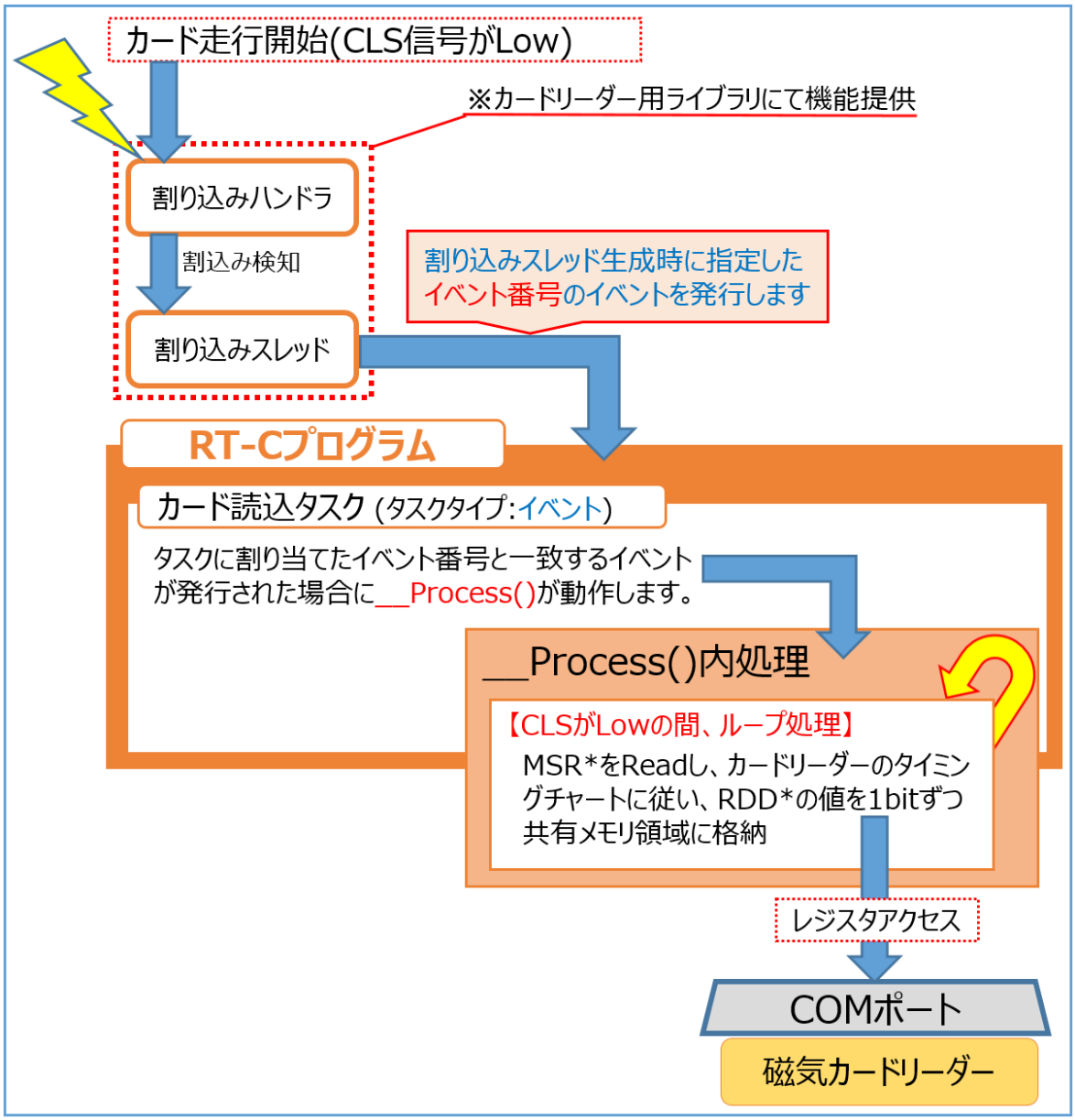
※DCD/DSR/CTS はいずれもモデムステータスレジスタにより信号状態の読み取りが可能です。また、信号変化時に割込みを有効にすることが出来ます。本プログラムでは、このモデムステータス割り込みを利用します。

3.5. カードリーダーとシリアルポートの配線図



3.6. データ取得処理

カードリーダーの出力信号を取得し、磁気カード情報を得るためには、カード走行が開始されたタイミングで即座にデータ読み込み処理を動作させる必要があります。
本サンプルプログラムではシリアルポートの MSR(モデムステータスレジスタ)の状態変化をトリガーとするモデムステータス割込みの検知によりデータ取得を開始します。下記は、データ取得処理のイメージです。



*MSR:　モデムステータスレジスタ。カードリーダーの CLS, RCP, RDD の信号状態を取得します。

*RDD:　RCP(サンプリング周期信号)の立ち下がりで取得することで磁気カード情報を 1bit ずつ取得します。
詳細は、P.11「タイミングチャート」を参照してください。

実際にカードデータ取得を行っている箇所のソースコードを下記に示します。

[ソースコード] ¥¥SampleProgram¥CardReader¥CardReader¥CardReader.cs

```
public void __Process()
{
    // 割り込みスレッドからのイベント通知により本プロセスが駆動します
    Byte byState;              // モデムステータスレジスタ
    if (Err != 0) return;      // エラーが発生している場合は、即終了

    // 割り込み要因のチェック (割り込み要因がなくなるまで処理ループ)
    while ((pcibus.InBYTE(IO_INTRR) & IIR_PENDING) == 0x00)
    {
        // CLS(カード走行信号)がLowの間、データ読み込み処理をループ
        byState = 0x00;    // 初回は必ずループ処理に入るため、0x00をセット
        while ((byState & CLS) == 0x00)
        {
            byState = pcibus.InBYTE(IO_MSTS);   // モデムステータスレジスタ読込
            Set_CardData( byState ); // モデムステータスレジスタをチェックし、データを格納
        }
    }
}
```

上記の__Process()部からコールしている Set_CardData()です。

```
private void Set_CardData(Byte State)
{
    uint uiIndex;                       // データ格納用配列インデックス
    // CLS(カード走行信号)チェックし、カード走行開始と走行終了を判断します
    if ((State & CLSx) == CLSx)         // CLS信号状態変化検知
    {
        if ((State & CLS) == 0x00) // カード走行開始(CLS信号:Low)の場合
        {
            RDD_Cnt = 0;                // データ格納回数をクリア
            for (int i = 0; i < 100; i++)
            { pbData[i] = 0; }          // データ保存先メモリのクリア
        } else {                        // カード走行終了(CLS信号:High)の場合
            (*pReadCnt)++;              // 読み取り回数をインクリメント
        }
    }
    // RCP立下りのタイミングでRDDを読み込みます
    if ((State & (RCP | RCPx)) == RCPx)  // RCPの立下りを検知(RCP変化あり、かつRCP:Low)
    {
        // RDD(カードデータ)を1bitずつ格納する
        if ((State & RDD) == 0x00)      // RDD信号:Lowの場合、データビットを[1]に設定
        {
          pbData[RDD_Cnt / 8] |= (Byte)(0x01 << (int)(RDD_Cnt % 8));
        }
        else                            // RDD信号:Highの場合、データビットを[0]に設定
        {
          pbData[RDD_Cnt / 8] &= (Byte)~(0x01 << (int)(RDD_Cnt % 8));
        }
        // データ格納回数をインクリメント ※カード情報の先頭は0xFFであるため、カウント開始条件に「RDDがLow」を付加する
        if (((RDD_Cnt == 0) && ((State & RDD) == 0x00))
            || (RDD_Cnt > 0))
        {
            RDD_Cnt++;
        }
    }
}
```

3.7. 割り込み処理について

データ取得処理にて説明した割り込みを行うための初期化コードを記載します。
[ソースコード] ¥¥SampleProgram¥CardReader¥CardReader¥CardReader.cs

(1) カードリーダーライブラリの API をコールし、割り込みスレッドを生成します。
割り込みスレッド生成時に指定するイベント番号は、RT-C プロジェクトで作成するイベントタスクに割り当てるイベント番号と紐づける必要があります。本サンプルでは、イベント 0 を指定しています。
下記関数は、__Init()部からコールしています。

```
private void Init_Interrupt()
{
    UInt16 hINTRR;

    // 割り込みハンドラ・スレッドのクローズ(失敗は無視します)
    // 割り込みスレッド生成時に指定する割込みレベルと同じ値を指定してください
    CARD.RTC_CloseInterruptThread(IRQ4_LEVEL);

    // 割り込みハンドラ・スレッド生成
    // イベント番号:0, 割り込みレベル:IRQ4
    hINTRR = CARD.RTC_CreateInterruptThread(0, IRQ4_LEVEL);
}
```

(2) シリアルポートの初期設定を行います。
レジスタ構成、ビット割付については、付録を参照してください。

```
private void Init_Device()
{   // Comポートの初期化を行います
    // UARTの停止
    pcibus.OutBYTE(IO_MCTL, 0x00);   // モデムコントロール: ER RS off /割り込み無効
    Delay();
    // ボーレートの設定
    pcibus.OutBYTE(IO_LCTL, 0x80);   // DLAB=1 (ボーレート設定モードON)
    Delay();
    pcibus.OutHWORD(IO_BASE, 12);    // ボーレート: 12 = 9600 Kbps
    Delay();
    // 通信モードの設定
    pcibus.OutBYTE(IO_LCTL, 0x03);   // 通信設定:パリティ=無, StopBit=1bit, データ長=8bit
    Delay();
    // FIFOの設定
    pcibus.OutBYTE(IO_INTRR, 0x06); // FIFO: 未使用
    Delay();
    // 送受信可能に設定
    pcibus.OutBYTE(IO_MCTL, 0x0b);   // モデムコントロール: ER RS ON / 割り込み有効
    Delay();
    // 割り込み許可設定
    pcibus.OutBYTE(IO_INTMSK, 0x08);// 割り込みマスク: モデムステータス割り込み有効
    Delay();

    // 割り込み要因をクリア
    while ((pcibus.InBYTE(IO_INTRR) & 0x01) == 0x00) // 割り込み要因がなくなるまで処理ループ
    {   // モデムステータスレジスタのReadにより割込み要因をクリアします
        pcibus.InBYTE(IO_MSTS);
    }
}
```

■ 付録

シリアルポートのレジスタ構成およびサンプルプログラムで主に使用しているレジスタのビット構成を示します。

【レジスタ構成】

Address	R/W	名称	略称	説明
BASE + 0	R	レシーブデータ	RBR	受信データを読み出す(DLAB=0)
BASE + 0	W	トランスミットデータ	THR	送信データを書き込む(DLAB=0)
BASE + 0	R/W	ボーレート・LSB	(RBR)	通信スピードの下位情報を書き込む(DLAB=1)
BASE + 1	R/W	ボーレート・MSB	(THR)	通信スピードの上位情報を書き込む(DLAB=1)
BASE + 1	R/W	インタラプトイネーブル	IER	IRQ 要因を書き込む(DLAB=0)
BASE + 2	R	インタラプト ID	IIR	現在の割込み要因を読み出す
BASE + 2	W	FIFO コントロール	FCR	内臓 FIFO の動作を書き込む
BASE + 3	R/W	ラインコントロール	LCR	送受信データのフォーマットを書き込む(bit7 が DLAB)
BASE + 4	R/W	モデムコントロール	MCR	9Pin の制御ピンからの割込みフラグを読み出す
BASE + 5	R	ラインステータス	LSR	通信状態を読み出す
BASE + 6	R	モデムステータス	MSR	9Pin の制御ピンを読み出す
BASE + 7	R/W	スクラッチ	SCR	制御に関係のないレジスタ

※ COM1：ベースアドレスは 0x03F8、IRQ4（リアルタイム OS デコード値：0x48）

【インタラプトイネーブル】

IRQ 発生要因を指定します。

bit	7	6	5	4	3	2	1	0
	rsv	rsv	rsv	rsv	modem	Rx line	Tx data	Rx data
R/W	R=0	R=0	R=0	R=0	R/W	R/W	R/W	R/W

[bit3=1] モデムステータス割り込み

[bit2=1] ラインステータス(Rx)割り込み

[bit1=1] 送信データエンプティ割り込み

[bit0=1@16450 互換モード] データ受信割り込み

[bit0=1@16550 互換モード] Rx FIFO トリガーレベル&タイムアウト割り込み

【インタラプト ID】

割り込み発生時の発生要因が示されます。

bit	7	6	5	4	3	2	1	0
	FIFO	FIFO	rsv	rsv	ID	ID	ID	pending
R/W	R	R	R=0	R=0	R	R	R	R

[bit7,6=11b] 16550 互換モード

[bit7,6=00b] 16450 互換モード

[bit3,2,1] 割り込みがある場合に更新(複数割り込み時は高優先度を示す)

(011b) Rx line status /優先度 1st

(010b) Rx data @16450 互換モード /優先度 2nd

(010b) Rx FIFO trigger @16550 互換モード /優先度 2nd

(110b) Rx FIFO time-out /優先度 3rd

(001b) Tx data empty @16450 互換モード /優先度 4th

(001b) Tx FIFO empty @16550 互換モード /優先度 4th

(000b) modem status /優先度 5th

[bit0=0] 割り込み有り(初期値 1b)

【モデムステータス】

制御ピンの状態が示されます。

bit	7	6	5	4	3	2	1	0
	DCD	RI	DSR	CTS	DCDx	RIx	DSRx	CTSx
R/W	R	R	R=0	R=0	R	R	R	R

[bit7=1] 現在 DCD ピンは、アクティブ(アサート)な状態を受けている

[bit6=1] 現在 RI ピンは、アクティブ(アサート)な状態を受けている

[bit5=1] 現在 DSR ピンは、アクティブ(アサート)な状態を受けている

[bit4=1] 現在 CTS ピンは、アクティブ(アサート)な状態を受けている

[bit3=1] DCD ピンが変化された

[bit2=1] RI ピンがインアクティブ(ネゲート)への変化を受けた

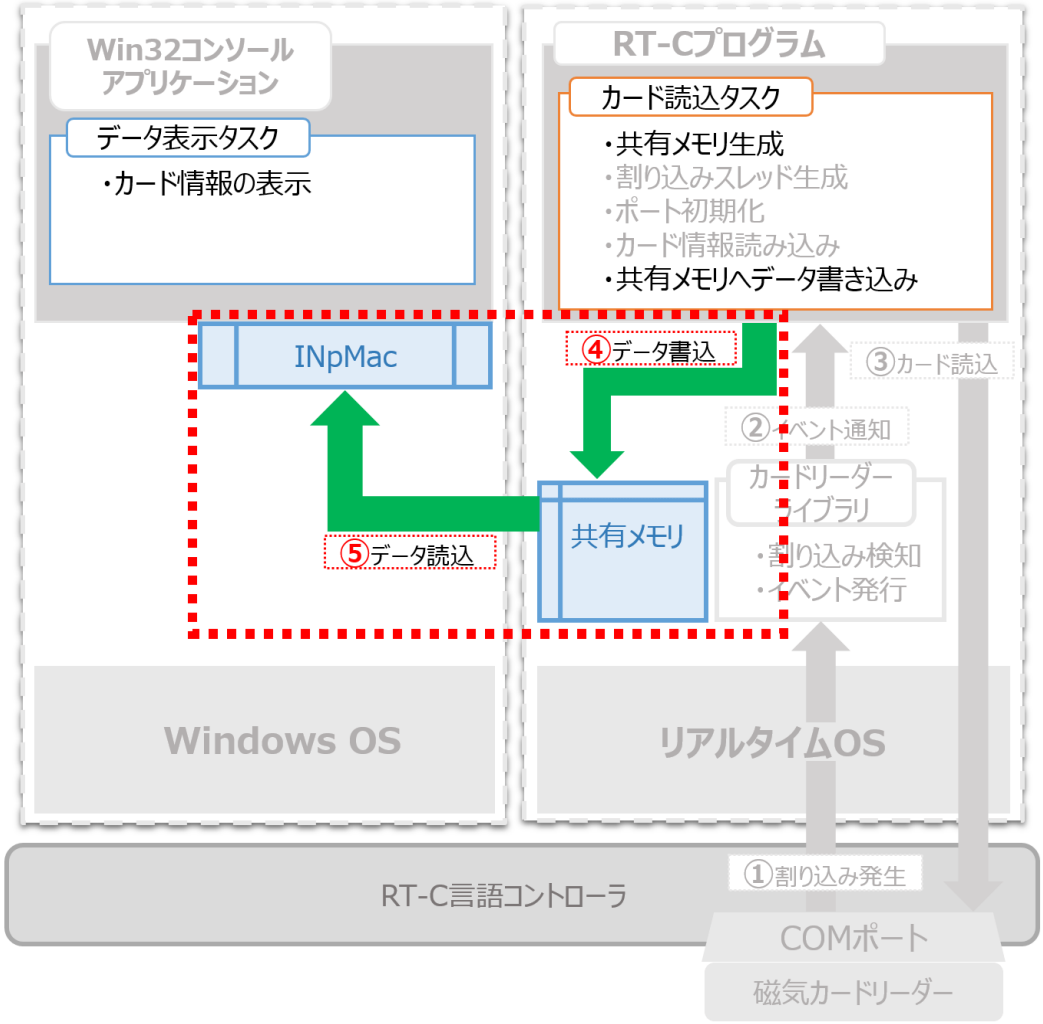
[bit1=1] DSR ピンが変化された

[bit0=1] CTS ピンが変化された

※bit3～0 のフラグが、モデムステータス割り込みの要因となります

4. Windows アプリケーションとのデータ連携について

本サンプルでは、リアルタイム OS と WindowOS 間のデータ連携として、共有メモリを利用しています。
本項では、下図のように共有メモリを介したデータ連携について説明します。



4.1. RT-Cプログラム側

リアルタイム OS 側の RT-C プログラムにて、共有メモリ領域を確保し、磁気カードリーダーから取得したデータを書き込みます。共有メモリの生成には、RTC_API ライブラリで提供されている RtMemory(共有メモリ操作クラス)を利用します。

(1) __Init()部からコールしている「共有メモリ初期化処理」で共有メモリ領域の確保をします。
本サンプルでは、確保した共有メモリの先頭 4byte にカード読み取り回数、5byte 目以降にカードデータを格納しています。

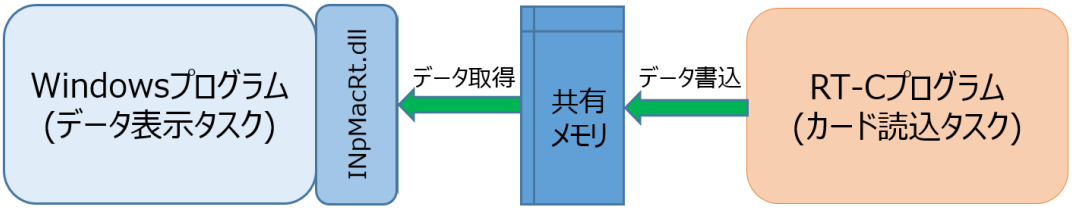
```
private void Init_Memory()
{
    void* pMem;
    IecString80 MEM_NAME = new IecString80();
    MEM_NAME.ctor();
    MEM_NAME.s.Init("CardReader");      // 共有メモリのカタログ名
    // カードデータ保存先メモリを確保
    cmemData = new RtMemory(4096
                            , (Byte)RTC_APILib.HandleSelection.ROOT_PROCESS
                            , MEM_NAME);
    if (cmemData.GetLastErr() != 0)
    {   // すでにメモリ領域がある場合は参照する
        IecString80 sProcName = new IecString80();
        sProcName.ctor();
        sProcName.s.Init("");

        // 指定名称"CardReader"のメモリ領域を参照します
        cmemData = new RtMemory(sProcName, MEM_NAME);
        if (cmemData.GetLastErr() != 0)
        {
            Err = 1;     // メモリ参照エラー
            RT.printf("RtMemory Refer Error\n");
            return;
        }
    }
    // メモリ領域の先頭アドレス取得
    pMem = cmemData.GetMemPtr();
    if (pMem == null)
    {
        RT.printf("memGet Error\n");
        Err = 1;     // メモリ確保エラー
        return;
    }
    pReadCnt = (UInt32*)pMem;          // メモリの先頭4byteにカード読み取り回数を格納
    pbData = (Byte*)&pReadCnt[1];      // メモリの5byte目以降にカードデータを格納
    *pReadCnt = 0;                     // カード読み取り回数を初期化
}
```

(2) 取得したカード情報を共有メモリ領域へ書き込みます。
書込みは、共有メモリ生成時に取得したアドレスポインタを使用します。
P.14「Set_CardData()」を参照してください。

※RtMemory(共有メモリ操作クラス)の詳細については、「RT-C ユーザーズマニュアル」を参照してください。

4.2. Widowsプログラム側

リアルタイム OS 側で生成された共有メモリ領域を Windows アプリケーションから参照するために、共有メモリ I/F の「INpMac」を利用します。



※INpMac を利用してメモリ参照する場合、リアルタイム OS 側でカタログされていることが前提となります。
※INpMac 機能の詳細については、「INpMac_UsersManual.pdf」を参照してください。

INpMac 機能を利用した「データ読み込み処理」を下記に記載します。

[ソースコード] ¥¥SampleProgram¥CardDisp¥CardDisp¥CardDisp.cpp

```
bool ReadData()
{
    int      iSize = 0;
    BYTE     *byMemData;      // 共有メモリ用

    byMemData = (BYTE *)malloc(100);   // メモリ確保

    // データの読込み
    iSize = InpMacRt_ReadMemory((LPCTSTR)"",(LPCTSTR)cRtMemName
                                                    ,0,byMemData, 100 );

    if(iSize > 0)  // 正常に読込できた場合
    {
        // カード読み取り回数チェック
        if(readCnt != *byMemData)
        {
            // カード読み取り回数を更新
            readCnt = *byMemData;
            // データをコンソールに出力
            DispData( byMemData );
        }
    }
    free(byMemData);
    return true;
}
```

INpMacRt_ReadMemory() を使用します。
名前 (cRtMemName) はリアルタイム OS 側でカタログした名称と一致させる必要があります。

カード読み取り回数が更新されていた場合のみコンソール出力を行います

カード読み取り回数が更新されていた場合にコールされるコンソール出力処理 DispData()を記載します。

```
void DispData(BYTE *byData)
{
    int      iCnt;                  //ループカウンタ
    char     offset = 4;            //データ部格納領域のオフセット(4byte目以降)
    // カードデータをコンソールに出力します
    printf("[Card]Read Data...  Count[%d]", readCnt);
    for (iCnt = 0; iCnt < 80; iCnt++)
    {
        if (iCnt % 16 == 0){ printf("\n"); }
        printf("%02X ", *(byData + offset + iCnt));
    }
    printf("\n-------------------------------------------------\n", readCnt);
    return;
}
```

データ表示プログラムのメイン関数を記載します。

本サンプルでは、200msec 周期で共有メモリのチェックを行っています。

```
#include "stdafx.h"

// 共有メモリカタログ名
char     cRtMemName[] = "CardReader"; // C#プログラムで確保する共有メモリのカタログ名称とあわせてください。
// データ読み取り回数保持
int      readCnt;

int _tmain(int argc, _TCHAR* argv[])
{
    bool bRet = true;
    readCnt = 0;
    printf("[Card]----Process start----------------------\n");

    while(bRet)                    // 共有メモリをポーリングします
    {
        bRet = ReadData();         // 共有メモリ読み込み＆コンソール出力
        Sleep(200);                // 200msecスリープ
    }
    printf("[Card]----Process end----------------------------\n");
    return 0;
}
```

5. トラブルシューティング

本教材のサンプルプログラム実行時、正常に動作しない場合は、以下を参照し設定を見直してください。

現象	原因	対応
カード情報が Windows 画面に表示されない	磁気カードリーダー用の電源が接続されていない	電源を接続してください
	磁気カードリーダーとシリアルケーブルのコネクタ配線が外れている	配線図を参照して正しく接続してください
	シリアルケーブルの接続先が間違っている	お使いのコントローラに複数の COM ポートがある場合は、他の COM ポートに接続し直してからプログラムを再実行してください
	COM1 ポートの割り込み制御をリアルタイム OS 側に移していない	P.5「COM ポートの設定変更」を参照して、設定を変更してください
	COM1 ポートで使用される IRQ が異なっている	サンプルプログラムでは IRQ4 が割り当てられていることを想定しています

その他、製品 Web ページにて「よくあるご質問(FAQ)」や「サンプルプログラム」を紹介しておりますので、本書やマニュアルと合わせてご活用ください。

■ 製品Webページ： http://mnc.co.jp/RT_Controller/RT_C/

RT C言語コントローラ

磁気カードリーダーサンプル

Rev. 2016/12/19

製品に関するお問い合わせはコチラまで

E-MAIL：**bcd@mnc.co.jp**

株式会社マイクロネット http://mnc.co.jp/