

**AN2021-10**

**iMOTIONTM構成可能UARTの使用方法**

**このドキュメントについて範囲と目的**

このアプリケーションノートは、特定のアプリケーションでiMOTIONTM構成可能UARTを使用する方法の例を提供し、構成可能UARTで使用可能な方法について説明します。

**対象とする訪問者**

このアプリケーションノートは、アプリケーションでiMOTIONTM Configurable UARTを使用する方法を理解したいお客様を対象としています。

**目次**

**目次 1**

**1 設定可能なUARTの概要 2**

1.1 はじめに 2

1.2 概要 2

1.3 UARTハードウェアドライバー 3

**2 Buffer Mode (バッファ モード) 4**

2.1 バッファモードの説明 4

2.2 バッファモードのカスタムプロトコルの例 5

2.2.1 バッファモードの初期化 6

2.2.2 受信フレーム構造 7

2.2.3 送信フレーム構造 7

2.2.4 エラー フレーム 8

2.2.5 バッファモードを使用したプロトコルの実装 9

2.2.6 パフォーマンス評価 10

**3 FIFOモード 12**

3.1 FIFOモードの説明 12

3.2 FIFOモードのカスタムプロトコルの例 13

3.2.1 FIFOモードの初期化 13

3.2.2 FIFOモードを使用したプロトコルの実装 13

3.2.3 パフォーマンス評価 16

**4 ガイドラインと 制限事項 17**

4.1 バッファモードとFIFOモード 17

4.2 制約 17

4.3 ガイドライン 18

**5 参考資料 19**

**改訂履歴 20**

|  |  |
| --- | --- |
|  |  |
|  |  |

**1 設定可能なUARTの概要**

**1.1 はじめに**

iMOTIONTM MCE（モーションコントロールエンジン）の最新のソフトウェアリリースは、お客様向けに2種類のUART通信オプションをサポートしています。1つのオプションは、事前定義されたユーザーモードUART通信プロトコルに従うことであり、もう1つのオプションは、構成可能なUART機能を使用してカスタマイズされたUART通信プロトコルを実装することです。

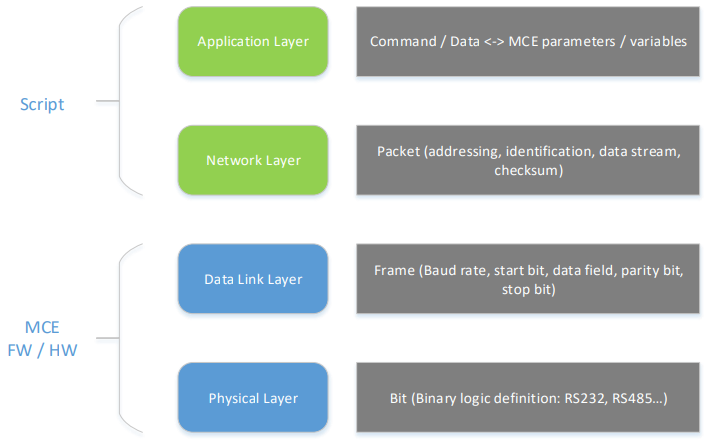
ユーザーモードUART通信プロトコルは、モーター制御アプリケーションにシンプルで信頼性が高く、スケーラブルな通信方法を提供するように設計されています。プロトコルはシンプルなので、MCEとモーターを制御するマスターとして機能するローエンドのマイクロコントローラーでも簡単に実装できます。これは、一部の産業用ファン/ポンプアプリケーションで必要とされる1マスターマルチスレーブネットワークトポロジ（同じネットワーク上に最大15のスレーブノード）をサポートします。各UARTコマンドは1msごとに処理されます。ユーザーモードUART通信プロトコルの詳細については、MCEソフトウェアリファレンスマニュアル[1]のセクション2.3を参照してください。

ユーザーがカスタマイズされたUART通信プロトコルを実装したい場合は、このドキュメントで説明されている構成可能なUART APIを使用して実現できます。構成可能なUART機能は、スクリプトエンジンでサポートされています。スクリプトエンジンはMCEで実行される軽量の仮想マシンであり、ユーザーはモーター制御やPFCを超えたシステムレベルの機能を実装できます。

**1.2 概要**

構成可能なUART機能は、ユーザー定義または業界標準の通信プロトコルを実現できるカスタマイズ可能な通信プロトコルです。設定可能なUART機能には2つの異なる動作モードがあり、通信プロトコルに応じて、一方が他方よりも便利です。1つはバッファモードUARTで、もう1つはFIFOモードUARTです。バッファモードUARTは、ファームウェアでネットワーク層の処理を処理する単純なモードです。対照的に、FIFOモードUARTはネットワーク層の処理を行いませんが、ユーザーはスクリプトコードを使用してネットワーク層を処理できます。図1に示すように、構成可能なUART機能、MCEファームウェア、および関連するハードウェア周辺機器ハンドルの一部を使用する場合、ユーザーがネットワークを実装できるようにしながら物理層とデータリンク層、およびスクリプトを使用してたアプリケーション層。

**図1 通信プロトコル層**



|  |  |
| --- | --- |
|  |  |
|  |  |

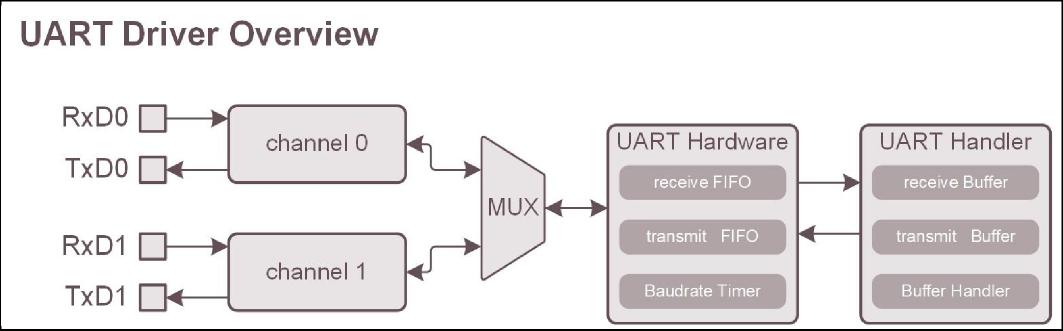
目的のプロトコルを実装するには、スクリプトエンジンとともに構成可能なUART APIを使用する必要があります。表1は、ユーザーが使用できる構成可能なUART APIの完全なリストです。スクリプトエンジンまたは構成可能なUART APIの詳細については、それぞれ[2]および[1]を参照してください。

**表1 構成可能なUART API**

|  |  |
| --- | --- |
| **API名** | **簡単な説明** |
| UART\_DriverInit（） | UARTハードウェアドライバーを初期化します。 |
| UART\_DriverDeinit（） | UARTハードウェアドライバーを非初期化します。 |
| UART\_FifoInit（） | UARTハードウェアFIFOを初期化します。 |
| UART\_BufferInit（） | UARTソフトウェアバッファを初期化します。 |
| UART\_GetStatus（） | UART通信ステータスのステータスワードを取得します。 |
| UART\_GetRxDelay（） | 受信フレーム間の遅延時間を返します。 |
| UART\_Control（） | UART制御コマンドを定義する制御ワードに書き込みます。 |
| UART\_RxFifo（） | 受信FIFOから1バイトを返します。 |
| UART\_TxFifo（） | 送信FIFOに1バイトを入れます。 |
| UART\_RxBuffer（） | 指定された場所の受信バッファから1バイトを返します。 |
| UART\_TxBuffer（） | 指定された場所の送信バッファに1バイトを配置します。 |

**1.3 UARTハードウェアドライバー**

図2にUARTドライバーの構造を示します。UART\_DriverInit（）を使用すると、ユーザーはUARTハードウェアに関連する重要なパラメーター（UARTチャネル、ボーレート、データビット、ストップビット、パリティ、txおよびrx信号の反転など）を選択できます。バッファモードまたはFIFOモードのいずれかを使用する前に、ユーザーは最初にハードウェアドライバを初期化する必要があります。UARTドライバの初期化の詳細については、[1]を参照してください。



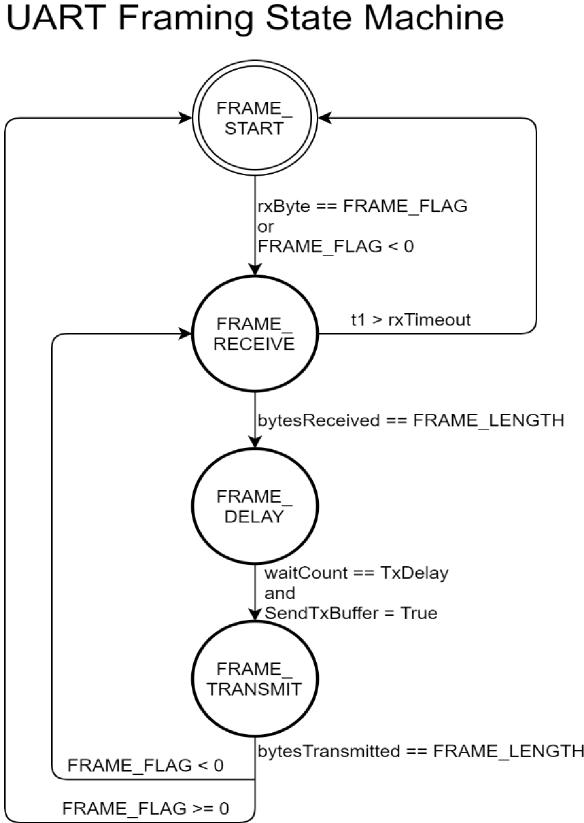
**図2 UARTハードウェアドライバーの概要**

|  |  |
| --- | --- |
|  |  |
|  |  |

**2 Buffer Mode (バッファ モード)**

**2.1 バッファモードの説明**

バッファモードは、MCEファームウェアを利用して、ネットワーク層の物理層、データリンク層、およびタイミング関連部分を処理するUARTモードです。その結果、ユーザーは、ネットワーク層に煩わされることなく、バッファリングされたデータにアクセスして上位層を処理できます。バッファモードの制限の1つは、フレーム内のデータバイト数を固定する必要があることです。バッファモードは初期化によって構成可能であり、実行時にデータバッファとステータス情報へのアクセスを提供します。図3は、バッファモードステートマシンの概要です。



**図3 バッファモードステートマシン**

|  |  |
| --- | --- |
|  |  |
|  |  |

**表2 状態の説明と遷移**

|  |  |  |  |
| --- | --- | --- | --- |
| **保持** | **状態機能** | **移行イベント** | **次のシーケンス状態** |
| FRAME\_START | バイトが受信され、FRAME\_FLAGと比較されます。FRAME\_FLAGに一致しないバイトはすべて無視され、一致するバイトは遷移イベントを示します。 | 受信したバイトは、既知のFRAME\_FLAGと一致します。 | FRAME\_RECEIVE |
| FRAME\_RECEIVE | フレーム長までバイトを受信し、受信遅延タイマーを停止します。すべてのバイトが受信されると、受信遅延タイマーが再び開始されます1。 | 受信したバイトがフレーム長と等しい場合。 | FRAME\_DELAY |
| 最初に受信したバイトから最後に受信したバイトまでの時間がrxTimeoutより大きい場合。 | FRAME\_START |
| FRAME\_DELAY | ステートマシンは、設定された送信遅延の間、この状態のままになります。 | txDelayが満たされ、sendTxBufferがtrueの場合。 | FRAME\_TRANSMIT |
| FRAME\_TRANSMIT | 送信バッファが送信されます。各バイト間の遅延を構成できます2。 | フレームフラグが無効な場合。 | FRAME\_RECEIVE |
| 送信フレームのすべてのバイトが送信され、フレームフラグが有効な場合。 | FRAME\_START |

*注意事項: タイミング関連のパラメータの詳細については、[1]を参照してください。*

**2.2 バッファモードのカスタムプロトコルの例**

カスタムプロトコルを実装して、独自のプロトコルを実装する方法を考えてみましょう。カスタムプロトコルの要件は次のとおりです。

**表3**

|  |  |
| --- | --- |
| **要件** | **詳細** |
| ボーレート | 115,200 bps |
| 物理レイヤー | RS-232 |
| UARTフレームビット | 1ストップビット、8データビット、パリティなし |
| フレームあたりのバイト数 | 9 |
| 最大送信遅延 | 20ミリ秒、これはMCEスレーブがデータフレームを受信する際の最大許容遅延です。 |
| コマンド | 少なくとも3つのコマンドをサポートする必要があります。  モーターの始動/速度の設定、モーターの停止、ステータスの取得 |
| フレームチェック | 各フレームのチェックサムを実行する必要があります |

|  |  |
| --- | --- |
|  |  |
|  |  |

**2.2.1 バッファモードの初期化**

まず、スクリプトエンジン設定、UARTドライバーを初期化し、要件を満たすようにバッファーモードを構成する必要があります。コードリスト1の2行目から7行目では、スクリプトバージョンを初期化し、スクリプト開始コマンドを設定し、Task1の実行手順を設定します。最大送信遅延を満たすために、Task1の実行期間を20ミリ秒に設定します。これにより、20ミリ秒間隔でデータフレームを見逃すことはありません。スクリプト設定の詳細については、[2]を参照してください。

UART\_DriverInit（）を使用して、ボーレートを115,200 bpsに設定し、1ストップビットを設定し、8データビットを設定し、パリティを設定せず、UARTチャネルをUART 1に設定し、UART信号の論理反転を無効にします。この後、UART\_BufferInit（）を呼び出して、プロトコルに関していくつかの重要な設定を設定します。

* 最大送信遅延は20ミリ秒ですが、最小送信遅延はありません。この要件を確実に満たすために、すべての遅延（txDelay、txByteDelay）をゼロに設定します。
* RxTimeoutは、受信フレームの最初のバイトと最後のバイトを受信する間の時間です。ボーレートが115,200bpsの場合、1ミリ秒以内に9バイトのフレーム全体を受信すると予想されます。エラーの余地を与えるために、RxTimeoutを3ミリ秒に設定しました。
* フレームあたり9バイトの要件を満たすために、txDataLengthとrxDataLengthを8に設定します。バッファモードは、受信および送信データフレームの開始を示すために、フレームの先頭に追加のバイトを自動的に挿入します。この開始バイトは、それぞれrxFlagとtxFlagによって指定されます。

コードリスト1は、プロトコル要件に基づいてUARTドライバーとバッファーハンドラーを初期化します。

**コードリスト1 ドライバーとバッファーの初期化**

001 /\*\*\*\*\*Script Settings\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

002 /\*Script version value should be 255.255\*/

003 #SET SCRIPT\_USER\_VERSION (1.02)

004 /\*Script execution time for Task1 in 10mS, maximum value

005 65535\*/

006 #SET SCRIPT\_TASK1\_EXECUTION\_PERIOD (2)

007 /\* Start command, Task0: Bit0, Task1: Bit1; if bit is set,

008 script executes after init \*/

009 #SET SCRIPT\_START\_COMMAND (0x3)

010 /\* Script Task1 step, this defines the number of lines to be

011 executed every 10mS\*/

012 #SET SCRIPT\_TASK1\_EXECUTION\_STEP (200)

013 /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

014

015 Script\_Task1\_init()

016 {

017 /\* Driver initialization \*/

018 /\* channel: 1, rxInvert: 0, txInvert: 0, baudrate: 115200,

019 dataBits: 8, parity: 0, stopBits: 1 \*/

020 UART\_DriverInit(1, 0, 0,115200, 8, 0, 1);

021

022 /\* Buffer initialization \*/

023 /\* halfDuplex: 0, rxTimeout: 3, txDelay: 0, txByteDelay:

024 0, rxFlag: 0xA5, txFlag: 0x5A, rxDataLength: 8,

025 txDataLength: 8 \*/

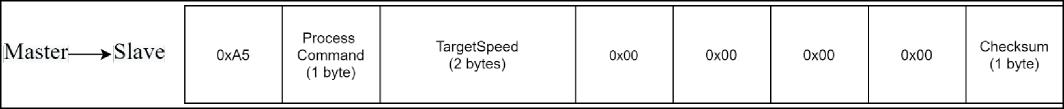
026 UART\_BufferInit(0, 3, 0, 0, 0xA5, 0x5A, 8, 8);

027 }

|  |  |
| --- | --- |
|  |  |
|  |  |

**2.2.2 受信フレーム構造**

次は、要件を満たす受信フレームを構築する必要があります。図4は、フレームの残りの部分を埋めるためのいくつかのnullデータとともに、基本的な要件を満たす受信データフレームの例です。



**図4 受信フレームの例**

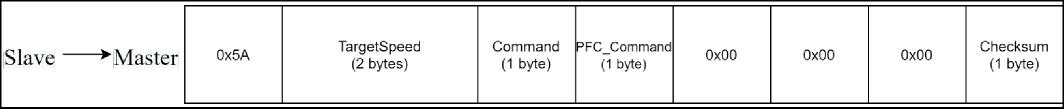
表4に、受信データフレーム構造の詳細を示します。マスターは、この形式のデータフレームをMCEスレーブに送信する責任があります。

**表4 フレーム構造の詳細を受け取る**

|  |  |  |
| --- | --- | --- |
| **バイト番号** | **名** | **内容** |
| 1 | rxFlag | UART\_BufferInit（）で指定された、受信データフレームの開始を示す最初のバイト。 |
| 2 | プロセスコマンド | このバイトは、MCEスレーブによって実行されるコマンドを指定します。   1. モーターを始動し、速度を設定します 2. モーターを停止します 3. ステータス情報を取得する |
| 3,4 | TargetSpeed | モーターのTargetSpeedを指定する2バイト（リトルエンディアン順）。 |
| 5,6,7,8 | ヌルデータ | これらのバイトは、フレームの残りの部分を埋めるためにゼロで埋められます。 |
| 9 | チェックサム | このバイトは、バイト1〜8のチェックサム値です。チェックサム= -1 \*（byte1 + byte2 + ... byte8） |

**2.2.3 送信フレーム構造**

マスターから受信するデータを決定したら、マスターに送信するフレームを作成する必要があります。図5は、プロトコルを満たすために必要なすべての情報を含む送信データフレームの例です。



**図5 送信フレームの例**

表5に、送信データフレーム構造の詳細を示します。MCEスレーブは、マスターからの正しいコマンドに応答して、この受信フレームを送信します。

|  |  |
| --- | --- |
|  |  |
|  |  |

**表5 フレーム構造の詳細を送信する**

|  |  |  |
| --- | --- | --- |
| **バイト番号** | **名** | **内容** |
| 1 | txFlag | UART\_BufferInit（）で指定された、送信データフレームの開始を示す最初のバイト。 |
| 2, 3 | TargetSpeed | モーターのTargetSpeedを指定するリトルエンディアン順の2バイト。 |
| 4 | コマンド | このバイトは、モーターが停止しているかどうかを指定します  開始状態。  1：開始  0：停止 |
| 5 | PFC\_Command | このバイトは、PFCが停止しているかどうかを指定します  開始状態。  1：開始  0：停止 |
| 6,7,8 | ヌルデータ | これらのバイトは、フレームの残りの部分を埋めるためにゼロで埋められます。 |
| 9 | チェックサム | このバイトは、バイト1〜8のチェックサム値です。チェックサム= -1 \*（byte1 + byte2 + ... byte8） |

**2.2.4 エラー フレーム**

MCEスレーブが無効なチェックサムを受信した場合、エラーフレームを作成する必要があります。図6は、無効なチェックサムを受信したときに送信されるエラーフレームの例です。



**図6 エラーフレームの例**

表6に、エラーフレーム構造の詳細を示します。マスターが誤ったチェックサムでデータフレームを送信した場合、MCEスレーブはそれに応じてこのフレームを送信します。

**表6 エラーフレーム構造の詳細**

|  |  |  |
| --- | --- | --- |
| **バイト番号** | **名** | **内容** |
| 1 | txFlag | UART\_BufferInit（）で指定された、送信データフレームの開始を示す最初のバイト。 |
| 2 | チェックサム | 最後に受信したデータフレームから正しく計算されたチェックサム。 |
| 3,4 | 定数 | エラーを示すためにフレームに配置された2バイトの定数。 |
| 5 | チェックサム | 最後に受信したデータフレームから正しく計算されたチェックサム。 |
| 6,7,8,9 | ヌルデータ | フレームを埋めるためのヌルデータ。 |

|  |  |
| --- | --- |
|  |  |
|  |  |

**2.2.5 バッファモードを使用したプロトコルの実装**

Script\_Task1（）では、UART\_GetStatus（）を使用して、isRxBufferFullビットをポーリングします。このビットをポーリングすると、バッファのサイズを満たした1つのフレームを受信したことがわかります。

次に、チェックサムを計算し、受信したデータフレームのチェックサムと比較します。正しい場合は、コマンドバイトに基づいてコマンドの1つを実行し、チェックサムが正しくない場合は、正しいチェックサムを含むエラーフレームを送信します。

最後に、各バイトのインデックスを指定しながら、UART\_TxBuffer（）を使用して送信データフレームにバイトを挿入します。送信データフレーム全体が構築されたら、UART\_Control（）を呼び出してSendTxBufferビットを設定することで送信を開始できます。

**コードリスト2 バッファモードコードの実装**

001 Script\_Task1()

002 {

003 const int START\_TX\_BYTE = 0x5A;

004 const int LOW\_BYTE\_MASK = 0xFF;

005

006 int checksum\_rx;

007 int checksum\_tx;

008 int status;

009

010 status = UART\_GetStatus();

011

012 /\*UART\_IsRxBufferFull\*/

013 if( status & 0x0100)

014 {

015 checksum\_rx = (-(0xA5 + UART\_RxBuffer( 0 ) + UART\_RxBuffer(1) + UART\_RxBuffer( 2 )) ) & 0xFF;

016

017 if(checksum\_rx == UART\_RxBuffer(3))

018 {

019 if(UART\_RxBuffer(0) == 1) // Set Speed, Start motor, Start PFC

020 {

021 TargetSpeed = UART\_RxBuffer(1) | (UART\_RxBuffer(2)<< 8);

022 Command = 1;

023 PFC\_Command = 1;

024 checksum\_tx = (-

(START\_TX\_BYTE+(TargetSpeed>>8)+(TargetSpeed&LOW\_BYTE\_MASK)+(Command&LOW\_BYTE \_MASK)+(PFC\_Command&LOW\_BYTE\_MASK))) & LOW\_BYTE\_MASK;

025

026 UART\_TxBuffer(0, TargetSpeed>>8 );

027 UART\_TxBuffer(1, TargetSpeed & LOW\_BYTE\_MASK );

028 UART\_TxBuffer(2, Command & LOW\_BYTE\_MASK);

029 UART\_TxBuffer(3, PFC\_Command & LOW\_BYTE\_MASK);

030 UART\_TxBuffer(4, 0x00);

031 UART\_TxBuffer(5, 0x00);

032 UART\_TxBuffer(6, 0x00);

033 UART\_TxBuffer(7, checksum\_tx);

034

035 }

036 if(UART\_RxBuffer(0) == 2)/\*Set speed to min speed, Stop motor, stop

037 PFC\*/

038 {

039 Command = 0;

040 PFC\_Command = 0;

041 TargetSpeed = MinSpd;

|  |  |
| --- | --- |
|  |  |
|  |  |

**コードリスト2 バッファモードコードの実装**

042 checksum\_tx = (-(START\_TX\_BYTE + (TargetSpeed>>8) +(TargetSpeed&LOW\_BYTE\_MASK) + (Command&LOW\_BYTE\_MASK) + (PFC\_Command&LOW\_BYTE\_MASK))) & LOW\_BYTE\_MASK;

043

044 UART\_TxBuffer(0,TargetSpeed>>8 );

045 UART\_TxBuffer(1,TargetSpeed & LOW\_BYTE\_MASK );

046 UART\_TxBuffer(2,Command & LOW\_BYTE\_MASK);

047 UART\_TxBuffer(3,PFC\_Command & LOW\_BYTE\_MASK);

048 UART\_TxBuffer(4, 0x00);

049 UART\_TxBuffer(5, 0x00);

050 UART\_TxBuffer(6, 0x00);

051 UART\_TxBuffer(7,checksum\_tx);

052 }

053 if(UART\_RxBuffer(0) == 3) // Get status information

054 {

055 checksum\_tx = (-(START\_TX\_BYTE + (TargetSpeed>>8) +

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 056  057  058  059  060 | (TargetSpeed & LOW\_BYTE\_MASK) + (Command & LOW\_BYTE\_MASK) + (PFC\_Command &  LOW\_BYTE\_MASK))) & LOW\_BYTE\_MASK;  UART\_TxBuffer(0,TargetSpeed>>8 );  UART\_TxBuffer(1,TargetSpeed & LOW\_BYTE\_MASK ); UART\_TxBuffer(2,Command & LOW\_BYTE\_MASK);  UART\_TxBuffer(3,PFC\_Command & LOW\_BYTE\_MASK ); | | | |
| 061 |  |  | UART\_TxBuffer(4, 0x00); |  |
| 062 |  |  | UART\_TxBuffer(5, 0x00); |  |
| 063 |  |  | UART\_TxBuffer(6, 0x00); |  |
| 064 |  |  | UART\_TxBuffer(7,checksum\_tx); |  |
| 065 |  |  | } |  |
| 066 |  |  | } |  |
| 067 |  |  | else |  |
| 068 |  |  | { // incorrect checksum received, send correct | checksum (チェックサム) |
| 069 |  |  | UART\_TxBuffer(0,checksum\_tx); |  |
| 070 |  |  | UART\_TxBuffer(1,0xEE); |  |
| 071 |  |  | UART\_TxBuffer(2,0xEE); |  |
| 072 |  |  | UART\_TxBuffer(3,checksum\_tx); |  |
| 073 |  |  | UART\_TxBuffer(4, 0x00); |  |
| 074 |  |  | UART\_TxBuffer(5, 0x00); |  |
| 075 |  |  | UART\_TxBuffer(6, 0x00); |  |
| 076 |  |  | UART\_TxBuffer(7, 0x00); |  |
| 077 |  |  | } |  |
| 078 |  |  |  |  |
| 079 |  |  | /\* UART\_SendTxBuffer | UART\_ClrRxBufferFlag \*/ |  |
| 080 |  |  | UART\_Control( 0x0500); |  |
| 081 |  | } |  |  |
| 082 | } |  |  |  |

**2.2.6 パフォーマンス評価**

モーターとPFCの両方が実行されているときに10kHzのインバーター周波数と33kHzのPFC周波数が与えられた場合、次のメトリックが取得されました。

* データフレームを受信すると、スクリプトが実行されてデータフレームの送信が開始されるまでに1ミリ秒もかかりません。
* 150ミリ秒の間隔で長期間データフレームを送信すると、MCEスレーブは20ミリ秒以内に問題なく正しく応答できました。

|  |  |
| --- | --- |
|  |  |
|  |  |

* CPU負荷の平均は69％で、CPU負荷のピークは72％でした。

o Script\_Task1は、CPU帯域幅の残りを取得するため、CPU負荷には影響しません。

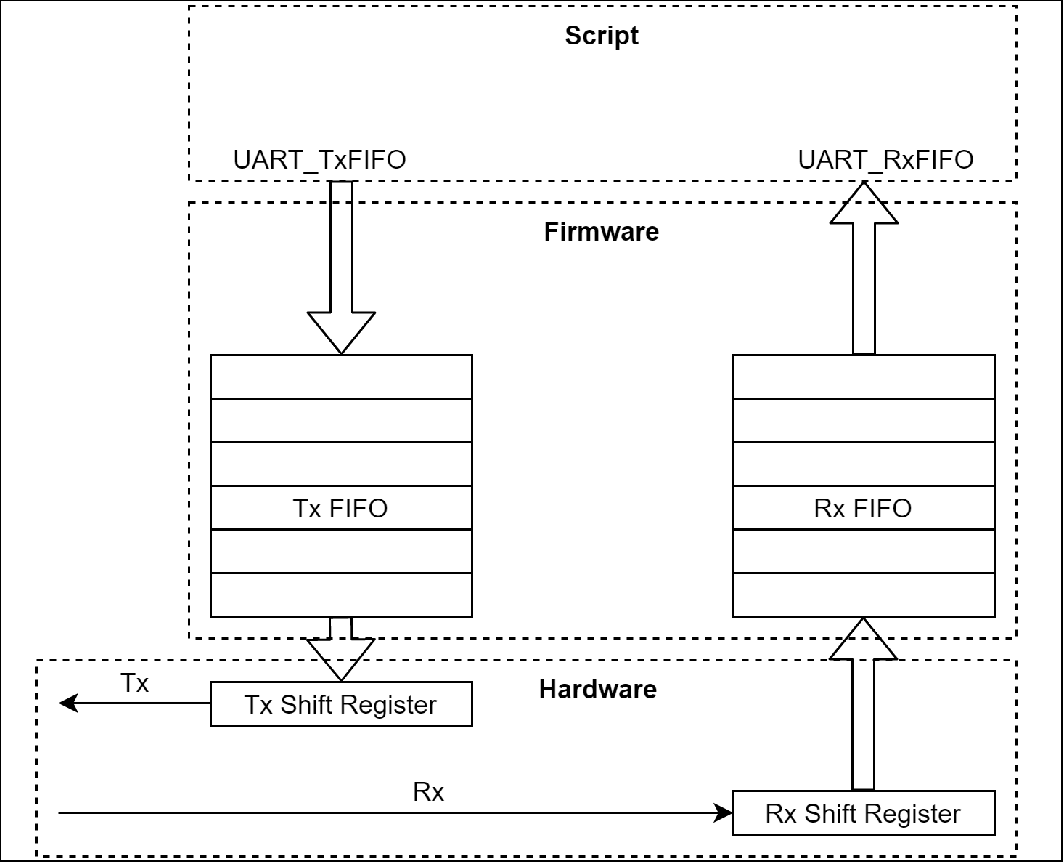
* 3つのRAM変数と2つの定数が必要でした（ただし、定数がなくても実行できます）。

|  |  |
| --- | --- |
|  |  |
|  |  |

**3 FIFOモード**

**3.1 FIFOモードの説明**

FIFOモードは、構成可能UARTの別の構成です。これは、転送されたデータワードのシーケンスが尊重されることを保証する先入れ先出しの原則に基づく単純なプロトコルです。バッファモードとは異なり、FIFOモードにはステートマシンがありませんが、FIFOハードウェアの単純なファームウェアラッパーです。受信したデータはすべてハードウェアバッファによってキャプチャされ、先入れ先出し法で取得できます。送信FIFOにロードされたデータは、すぐに送信を開始します。したがって、FIFOモードはフレーム内の可変数のデータバイトをサポートします。図7は、FIFOモードでのデータフローと、データフローの一部の処理を担当するレイヤーを表す図です。



**図7 FIFOモード図**

FIFOモードの利点は、柔軟性がはるかに高く、基盤となるファームウェアのオーバーヘッドがそれほど多くないことです。欠点は、バッファモードほど簡単に使用できないことです。FIFOモードでは、データは先入れ先出し方式でのみ送受信でき、ネットワーク層に関連付けられたタイミング要件は、ユーザーによるスクリプトを使用して実装する必要があります。

|  |  |
| --- | --- |
|  |  |
|  |  |

**3.2 FIFOモードのカスタムプロトコルの例**

セクション2.2で説明したのと同じプロトコルを実装しましょう。

**3.2.1 FIFOモードの初期化**

UARTドライバーの初期化やスクリプト設定については何も変更する必要はありません。受信データフレームと送信データフレームのサイズをそれぞれ設定して、FIFOを初期化するだけです。これを行うには、UART\_FifoInit（）を使用してrxFifoSizeとtxFifoSizeを9に設定します。これにより、受信および送信のFIFOサイズがそれぞれ9バイトに設定されます。

**コードリスト3**

001 /\*\*\*\*\*Script Settings\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

002 /\*Script version value should be 255.255\*/

003 #SET SCRIPT\_USER\_VERSION (1.02)

004 /\*Script execution time for Task1 in 10mS, maximum value

005 65535\*/

006 #SET SCRIPT\_TASK1\_EXECUTION\_PERIOD (10)

007 /\* Start command, Task0: Bit0, Task1: Bit1; if bit is set,

008 script executes after init \*/

009 #SET SCRIPT\_START\_COMMAND (0x3)

010 /\* Script Task1 step, this defines the number of lines to be

011 executed every 10mS\*/

012 #SET SCRIPT\_TASK1\_EXECUTION\_STEP (200)

013 /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

014

015 Script\_Task1\_init()

016 {

017 UART\_DriverInit(1 /\*channel\*/, 0 /\*rxInvert\*/, 0

018 /\*txInvert\*/, 115200 /\*baudrate\*/, 8 /\*dataBits\*/, 0

019 /\*parity\*/, 1 /\*stopBits\*/);

020 UART\_FifoInit(9 /\*rxFifoSize\*/, 9 /\*txFifoSize\*/);

021 }

FIFOモードの初期化以外は、プロトコルについて何も変更する必要はありません。受信、送信、エラーデータフレームの構造については、2.2.2、2.2.3、2.2.4を参照してください。スクリプトコードの実装に直接進むことができます。

**3.2.2 FIFOモードを使用したプロトコルの実装**

Script\_Task1（）では、UART\_GetStatus（）を使用して、isRxFifoFullビットをポーリングします。このビットをポーリングすると、FIFOのサイズを満たす1つのフレームを受信したことがわかります。

データフレームを受信したら、UART\_RxFifo（）を使用して、フレームのすべてのバイトを先入れ先出しの順序でバイトごとに格納します。その後、UART\_Control（）を使用してClrRxFIFOビットを設定することにより、受信FIFOをクリアします。

次に、チェックサムを計算し、受信したデータフレームのチェックサムと比較します。正しい場合は、コマンドバイトに基づいてコマンドの1つを実行し、正しくない場合は、正しいチェックサムを含むエラーフレームを送信します。

最後に、UART\_TxFIFO（）を使用して、送信フレームをバイトごとに送信します。FIFOの最初のバイトは、回線を介して送信される最初のバイトであることに注意してください。

|  |  |
| --- | --- |
|  |  |
|  |  |

**コードリスト4 FIFOモードコードの実装**

|  |
| --- |
| 001 Script\_Task1()  002 {  003 const int UART\_STATUS\_RX\_FIFO\_FULL = 0x0002;  004 const int UART\_CONTROL\_CLEAR\_RX\_FIFO = 0x0002;  005 const int UART\_CONTROL\_CLEAR\_TX\_FIFO = 0x0008;  006 const int START\_RX\_BYTE = 0xA5;  007 const int START\_TX\_BYTE = 0x5A;  008 const int LOW\_BYTE\_MASK = 0xFF;  009  010 int rx\_status;  011 int start;  012 int cmd;  013 int speed\_l;  014 int speed\_h;  015 int checksum\_pc;  016 int checksum\_calc;  017 int temp;  018  019  020 rx\_status = UART\_GetStatus();  021 if( rx\_status & UART\_STATUS\_RX\_FIFO\_FULL)  022 {  023 start = UART\_RxFifo(); // start byte  024 cmd = UART\_RxFifo(); // cmd byte  025 speed\_l = UART\_RxFifo();// speed low byte  026 speed\_h = UART\_RxFifo();// speed high byte  027 temp = UART\_RxFifo(); // null data  028 temp = UART\_RxFifo(); // null data  029 temp = UART\_RxFifo(); // null data  030 temp = UART\_RxFifo(); // null data  031 checksum\_pc = UART\_RxFifo();  032 UART\_Control(UART\_CONTROL\_CLEAR\_RX\_FIFO);  033  034 checksum\_calc = (-(start + cmd + speed\_l + speed\_h)) &  LOW\_BYTE\_MASK;  035 i  036 {  037 if(cmd == 1) // Set Speed, Start motor, Start PFC  038 {  039 TargetSpeed = speed\_l | (speed\_h << 8);  040 Command = 1;  041 PFC\_Command = 1;  042 checksum\_calc = -(START\_TX\_BYTE + (TargetSpeed>>8) + (TargetSpeed&LOW\_BYTE\_MASK) + (Command&LOW\_BYTE\_MASK) + (PFC\_Command&LOW\_BYTE\_MASK)) & LOW\_BYTE\_MASK;  043  044 UART\_TxFifo(START\_TX\_BYTE);  045 UART\_TxFifo( TargetSpeed>>8 );  046 UART\_TxFifo( TargetSpeed & LOW\_BYTE\_MASK );  047 UART\_TxFifo(Command & LOW\_BYTE\_MASK);  048 UART\_TxFifo(PFC\_Command & LOW\_BYTE\_MASK);  049 UART\_TxFifo(0x00);  050 UART\_TxFifo(0x00);  051 UART\_TxFifo(0x00);  052 UART\_TxFifo(checksum\_calc);  053 }  054 if(cmd == 2)/\*Set speed to min speed, Stop motor, stop |

|  |  |
| --- | --- |
|  |  |
|  |  |

**コードリスト4 FIFOモードコードの実装**

|  |
| --- |
| 055 PFC\*/  056 {  057 Command = 0;  058 PFC\_Command = 0;  059 TargetSpeed = MinSpd;  060 checksum\_calc = -(START\_TX\_BYTE + (TargetSpeed>>8) + (TargetSpeed&LOW\_BYTE\_MASK) + (Command&LOW\_BYTE\_MASK) + (PFC\_Command&LOW\_BYTE\_MASK)) & LOW\_BYTE\_MASK;  061  062 UART\_TxFifo(START\_TX\_BYTE);  063 UART\_TxFifo( TargetSpeed>>8 );  064 UART\_TxFifo( TargetSpeed & LOW\_BYTE\_MASK );  065 UART\_TxFifo(Command & LOW\_BYTE\_MASK);  066 UART\_TxFifo(PFC\_Command & LOW\_BYTE\_MASK);  067 UART\_TxFifo(0x00);  068 UART\_TxFifo(0x00);  069 UART\_TxFifo(0x00);  070 UART\_TxFifo(checksum\_calc);  071 }  072 if(cmd == 3) // Get status information  073 {  074 checksum\_calc = -(START\_TX\_BYTE + (TargetSpeed>>8) + (TargetSpeed&LOW\_BYTE\_MASK) + (Command&LOW\_BYTE\_MASK) + (PFC\_Command&LOW\_BYTE\_MASK)) & LOW\_BYTE\_MASK;  075  076 UART\_TxFifo(START\_TX\_BYTE);  077 UART\_TxFifo( TargetSpeed>>8 );  078 UART\_TxFifo( TargetSpeed & LOW\_BYTE\_MASK );  079 UART\_TxFifo(Command & LOW\_BYTE\_MASK);  080 UART\_TxFifo( PFC\_Command & LOW\_BYTE\_MASK );  081 UART\_TxFifo(0x00);  082 UART\_TxFifo(0x00);  083 UART\_TxFifo(0x00);  084 UART\_TxFifo(checksum\_calc);  085 }  086 }  087 else  088 { // incorrect checksum received, send correct checksum  089 UART\_TxFifo(START\_TX\_BYTE);  090 UART\_TxFifo(checksum\_calc);  091 UART\_TxFifo(0xEE);  092 UART\_TxFifo(0xEE);  093 UART\_TxFifo(checksum\_calc);  094 UART\_TxFifo(0x00);  095 UART\_TxFifo(0x00);  096 UART\_TxFifo(0x00);  097 UART\_TxFifo(0x00);  098 }  099 }  100 } |

|  |  |
| --- | --- |
|  |  |
|  |  |

**3.2.3 パフォーマンス評価**

モーターとPFCの両方が実行されているときに10 kHzのインバーター周波数と33 kHzのPFC周波数が与えられた場合、次のメトリックが取得されました。

* データフレームを受信すると、スクリプトの実行とデータフレームの送信に1ミリ秒もかかりません。
* 150ミリ秒の間隔で長期間データフレームを送信すると、MCEスレーブは20ミリ秒以内に問題なく正しく応答できました。
* CPU負荷の平均は69％で、CPU負荷のピークは72％でした。
* Script\_Task1は、CPU帯域幅の残りを取得するため、CPU負荷には影響しません。
* スクリプトには、8つのRAM変数と6つの定数が必要でした。
* クラスBソフトウェアを有効にすると、CPU負荷の平均は75％、CPU負荷のピークは77％でした。
* 平均CPU負荷値は68％から75％に変動します。

|  |  |
| --- | --- |
|  |  |
|  |  |

**4 ガイドラインと 制限事項**

**4.1 バッファモードとFIFOモード**

特定のアプリケーションでは、カスタムプロトコルの実装にユーザーがバッファモードとFIFOモードのどちらを使用するかを決定するのが難しい場合があります。以下の比較表は、この決定を支援するために使用する必要があります。

**表7 バッファとFIFOモードの比較**

|  |  |  |
| --- | --- | --- |
| **特長** | **バッファ** | **FIFO** |
| サポートされる最大フレームサイズ | 9 bytes | 31 bytes |
| ネットワーク層の一部を実装しますか？ | 有り | 番号無し |
| ランダムアクセス？1 | 有り | 番号無し |
| サポートされる最大ボーレート | 115,200 bps | 230,400 bps |
| 半二重をサポートしますか？ | 有り | 有り |
| 追加のCPU負荷2 | 有り | 番号無し |

1. *ランダムアクセス：データ構造内の特定のバイトを選択する機能です。FIFOモードはランダムアクセスデータ構造ではありませんが、バッファモードはランダムアクセスデータ構造です。*
2. *追加のCPU負荷：各モードに関連付けられている基盤となるファームウェアが原因で発生します。バッファモードにはファームウェアにステートマシンが含まれていますが、FIFOモードには含まれていません。*

セクション2.2.6および3.2.3で説明したように、FIFOモードでは、スクリプトコードの実装で使用するRAMを増やす必要があります。これは、受信FIFOのデータにランダムにアクセスできなかったためです。データを変数に配置し、後でスクリプトで使用する必要があります。一方、バッファモードでは、ユーザーはUART\_TxBuffer（）APIを使用してバッファからデータにランダムにアクセスし、必要なバイトのインデックスを指定できます。

**4.2 制約**

バッファモードとFIFOモードの制限については、表8と表9を参照してください。

**表8 バッファモードの制限**

|  |  |
| --- | --- |
| **制限** | **詳細** |
| 最大フレームサイズ | バッファモード構成の最大フレームサイズは8バイトです（rxFlag / txFlagは含まれません）。 |
| 最大ボーレート | バッファモードでサポートされる最大ボーレートは、115,200ボー/秒です。 |
| クラスBの問題 | クラスBの安全性テストが有効になっている場合、スタックオーバーフローテストが失敗したため、バッファモードUARTは最終的にフェイルセーフモードになります。クラスB安全性テストを有効にしながらカスタマイズされたUARTプロトコルが必要な場合は、代わりにFIFOモードを使用することをお勧めします。 |
| タスク0 | バッファモードも使用している場合は、タスク0を使用することをお勧めしません。これは、バッファモードステートマシンに干渉する可能性があるためです。タスク0を使用する必要がある場合は、タイムクリティカルな操作にのみ使用する必要があります。 |
| CPU制限 | CPU制限が90％に近づくと、CPU帯域幅が不足するため、バッファモードに問題が発生する可能性があります。 |

|  |  |
| --- | --- |
|  |  |
|  |  |

**表9 FIFOモードの制限**

|  |  |
| --- | --- |
| **制限** | **詳細** |
| 最大フレーム長 | FIFOモードでサポートされる最大フレーム長は31バイトです。 |
| 最大ボーレート | FIFOモードによる最大ボーレートサポートは、毎秒230,400ボーです。 |
| タスク0 | タスク0はCPUの負荷に影響を与える可能性があり、CPUの負荷が通信プロトコルに影響を与える可能性があります。タスク0を使用する必要がある場合は、タイムクリティカルな操作にのみ使用する必要があります。 |
| クラスBの問題 | スクリプトの複雑さ、モーター周波数、およびPFC周波数によっては、クラスBソフトウェアがフェイルセーフモードに入るときに問題が発生する場合があります。これは、過剰なスタック消費が原因で、クラスBソフトウェアがフェイルセーフモードになります。 |

**4.3 ガイドライン**

構成可能なUARTが目的のプロトコルを実装できるかどうかを判断するための一般的なガイドラインを次に示します。

1. FIFOモードでサポートされる最大データフレームは31バイトで、構成可能UARTでサポートされる最大データフレームです。

2. アプリケーションがモーター、PFC、クラスB安全性テスト、および構成可能UARTを必要とする場合、ユーザーはCPU負荷の問題とスクリプトの複雑さの問題にうんざりしている必要があります。

1. PFCとインバーターの周波数によっては、CPUの制限にぶつかる場合があります。CPU負荷が95％を超えると、ドライブはフェイルセーフモードに入る可能性があります。
2. スクリプトの複雑さによっては、ドライブがフェイルセーフモードに入る場合もあります。これは主にスタックの制約によるものであり、ユーザーがスクリプトコードを作成する方法によって、必要以上にスタックが消費される可能性があります。スクリプトコードでスタック消費を減らす方法については、[2]を参照してください。

3. 半二重通信を実装することは可能ですが、バッファモードでのみ可能です。

a. バッファモードは、halfDuplexパラメータを設定することにより、BufferInit（）APIを介してこれを直接サポートできます。

4. 構成可能なUARTは、固定長パケットでのパケットを一度に受信することをネイティブにサポートします。

1. つまり、ユーザーは、パケット全体を検出するために、FIFOまたはバッファーがいっぱいになったときにポーリングするだけで済みます。
2. 構成可能なUARTは、フレーム全体を検出するために固定長のパケットのみを使用しますが、他のプロトコルは、パケットを検出するために異なるスキームを使用する場合があります。

5. FIFOモードは、可変長の送信フレームサイズをサポートします。

6. 設定可能なUARTには、送信の完了を示すAPIがありません。

|  |  |
| --- | --- |
|  |  |
| **5 参考資料**   1. iMOTIONTMモーターコントロールエンジンソフトウェアリファレンスマニュアル（REV1.3）。 2. iMOTIONTMスクリプト言語（REV 1.1）の使用方法。 |  |

|  |  |
| --- | --- |
|  |  |
|  |  |

**改訂履歴**

|  |  |  |
| --- | --- | --- |
| **ドキュメントバージョン** | **リリース日** | **変更の説明** |
|  |  |  |
|  |  |  |
|  |  |  |