特設

第3章 PIOアセンブリ・コードが手軽に作れる

実機がなくてもデバッグできる簡易PIOエミュレータ

森岡 澄夫

ラズベリー・パイPico（以降，Pico）のPIO（プログラマブルI/O）は，カスタマイズした高速インターフェースを作れるという，他のマイコン・ボードにない画期的な機能を提供しています．しかし，若干クセのあるアセンブリ言語でのプログラミングが必要な上，デバッグ環境が整っておらず，開発がやりにくい問題がありました．

開発のきっかけ

● PIOはインターフェース作成に高いポテンシャルを持つ

ラズベリー・パイを含む従来のマイコン・ボードは，マイコンの動作クロックに近い高速インターフェースを作ることが困難です．例えば，マイコン・チップ内のUARTの数が足りなくなれば，処理を自作するしかありません．ペリフェラルにない独自インターフェースが必要になるときも同様です．

しかし，ソフトウェアでGPIOを制御するやり方では，高帯域を出せません．また，ラズベリー・パイのように，リアルタイムOSではないLinuxを使っている場合には，GPIOのスイッチング速度だけでなく，タイミング精度もかなり悪く，実用には耐えません．

PicoのPIO[3]は，これを解決できる画期的な機能です．高速（入出力信号の周波数が62.5MHzくらいまで）かつ，高いタイミング精度（クロック単位）のI/O処理を，FPGA（Field Programmable Gate Array）のような回路設計をすることなく，プログラミングだけで実現できます．実際に，Pico SDKのPIO設計サンプルには，UARTだけでなくSPIやI²Cも含まれているほど，実用度の高い機能です．

PIOは，本誌2021年8月号のPico特集で多くのページを割いて解説されています[6][7][8]［開発元発行のマニュアルは文献(3)］．ここでは詳細の説明は割愛しますが，Picoに載っているArmコアとは別のミニI/Oプロセッサ（ステート・マシンと呼ばれる）が計8個あって，並列動作できるようになっています．

● PIO開発環境が抱えていた問題

利用価値の高いPIOですが，筆者が実際に試してみたところ[7]，次の2点がネックでした．

1. アセンブリ言語での開発になる．命令セットはあまり一般的な設計ではない独特なもので，命令動作の細部まで頭に入れていないと分かりにくいバグを発生させてしまう．
2. ステート・マシンを1ステップずつ実行させてモニタリングできるようなデバッガが提供されていない．GPIOピンにオシロスコープをつないで波形観測しながら開発することになる．しかしステート・マシン内部のレジスタ値などを観測しにくく，ちょっとした間違いであってもなかなか特定できない．

特に後者は問題で，オシロスコープとにらめっこしながらのデバッグの面倒さは，PIOを使ってみる気を削ぐものでした．

● ステート・マシン1つのデバッグに特化したエミュレータ

そこで今回，「PIOを使おうと思い立ったら，すぐに開発できるようにする」ことをコンセプトに，ステート・マシン実行を可視化するエミュレータを作ることにしました．

図1に作業の全体的な流れを，**表1**に本エミュレータでの制限事項を示します．詳細は後で説明しますが，エミュレータからさまざまなC言語API関数を提供します．それらの関数を使ってアセンブリ・コードを書き［文献(8)で紹介されているMicroPythonによるコード記述の仕方と類似］，任意のCコンパイラでコンパイルします．また，GPIO信号や割り込み信号などステート・マシンへの入力を，別途CSVファイルにあらかじめ書いておきます．

コンパイルで得られたバイナリを実行するとエミュレーションが行われます．その結果，各クロック・サイクルで実行した命令，レジスタ値，GPIO出力値を記録したCSVファイルと，Pico C/C++SDKでコンパイル可能なアセンブラ・ソースファイルが生成されま