第３章　スクリプト言語 Pawn を用いたプログラミング（補足資料　その２）

本資料では，Pawn の仮想マシンの移植に関する情報を提供しています．

１：仮想マシンが実行できるようにする

２：PC 側アプリケーションから Pawn のバイトコードを受け取って実行する

３：仮想マシンの実行を中断できるようにする

４：ハードウェアアクセス関数（GPIO）を追加する

５：microSD からバイトコードを取得する

６：Pawn4 への対応

１　ステップ０：仮想マシンが実行できるようにする

書籍では LM3S3748 チップを使った付録基板で動作する Pawn の仮想マシンを使用していました．本資料では，LM4F232 評価ボード（http://www.tij.co.jp/ww/mcu/stellaris_cortex_m4f/index.html ）を例に取り，Pawn の仮想マシンの移植手順をご説明いたします．

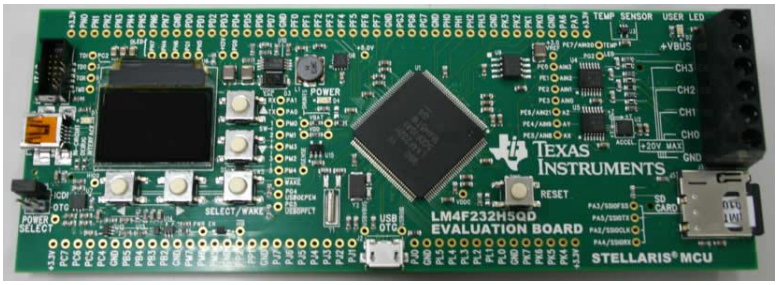

図１　LM4F232 評価ボード外観

Pawn の仮想マシン（PawnVM）を構築する際には，Pawn の開発元である CompuPhase 社が配布しているファイル[1]のうち**表１**のファイルで提供される API を使ったプログラミングを行います．

表１　PawnVM を構成する基本ファイル

amx.c，amx.h，osdefs.h	仮想マシン本体と各種定義
amxcons.c，amxcons.h	仮想マシンが使用するコンソール関連関数
float.c	浮動小数点モジュールを使う場合に必要（）

また，マイコン用に仮想マシンをビルドする際には，cpu の指定を行うとともに，**表 2** のマクロ定義（-D オプションで設定）行います．

表２　ビルド時のマクロ定義

AMX_ANSIONLY	ANSI のみを使用する
AMX_TERMINAL	OS 依存のターミナル関数は使わない（ターミナル関連の関数は別途作成する）
AMX_NODYNALOAD	スタティックリンクで使用する
FLOATPOINT	浮動小数点モジュールを組み込む（必要に応じて）
NDEBUG	assert を無効にしておく
HAVE_STDINT_H，HAVE_ALLOCA_H	ツールチェーン付属の「stdint.h」，「alloca.h」を使用する

さらに，Cortex-M4 コアを搭載したマイコンの浮動小数点演算機能（単精度浮動小数点命令にのみ対応）を使用する際には，**表３**のオプションも付けてビルドする必要があります．

表３　Cortex-M4 用ビルドオプション

-mfloat-abi=softfp -mfpu=fpv4-sp-d16	浮動小数点演算命令を使用する．float を引数とした関数呼び出しの際は汎用レジスタを使用する 倍精度の浮動小数点命令を使用しない

なお,StellarisWare では LM4F232 をターゲットとしたビルドを行う場合に「softfp」は自動的に設定されます.また，Cortex-M4 の FPU(Floating-Point Unit)を利用する際は，浮動小数点演算を実行するよりも前に，FPU を有効にする必要がありますので，注意が必要です.

いきなり多くの機能を盛り込んでしまうと，何が悪かったのかを見極めることが難しくなりますので，最初の段階としては，プログラム中に埋め込まれたバイトコードを実行する PawnVM のみを実装してみたいと思います.

第 0 段階のシステム（pawn_run.c）の主要部分は**リスト１**と**リスト２**になります.

リスト１　仮想マシン用ターミナル関数郡

```
37  /* PawnVM 用文字列出力関数 */
38  int amx_putstr(char* s) {
39    int len = strlen(s);                  UARTPrintf を使った文字列出力
40    UARTprintf("%s",s);                   for ループを使って 1 文字ずつ出力しても可
41    return len;
42  }
43
44  /* PawnVM 用文字出力関数 */
45  int amx_putchar(char c) {
46    MAP_UARTCharPut(UART_BASE , c);       MAP_UARTCharPut を使った文字出力
47    return 1;                                                                  ①
48  }
49
50  /* PawnVM 用コンソールフラッシュ関数 */
51  int amx_fflush(){
52    return 0;                             何もせず抜ける
52  }
54
55  /* PawnVM 用文字入力関数 */
56  int amx_getch(){
57    return  0;                            何もせず抜ける
58  }
59
60  /* native 関数リスト */                                                       ┐
61  const AMX_NATIVE_INFO MY_Natives[] =                                        │
62  {                                                                           ②
63    { "printf"            , n_printf },   C 言語で定義した n_printf を printf と言う名前で  │
64    { NULL , NULL }       // terminator   PawnVM に登録する                    │
65  };                                                                          ┘
```

リスト１では，①の部分で PawnVM が使用するターミナル関数の実装を行っています．今回は表示機能（amx_putstr，amx_putchar）のみに具体的な役割を実装し，その他の関数は何もしない関数にしてあります．表示に関わる関数は**リスト１**の②で登録している n_printf 関数（treminal.h で定義）から呼び出されます．他の組み込み言語と同様に，PawnVM においてもターミナル関連の機能はユーザプログラム中で実装することができますので，移植に伴う作業を最小限に抑えることができます．この段階では，ハードウェアに依存している部分は uart を使った文字出力だけです．そのため，他のマイコンを使う場合も，文字出力を行う関数を置き換えるだけで済みます．StellarisWare には UART を使用した入出力ライブラリー（uartstdio）が準備されていますので，今回は uartstdio を使用して実装しています.

リスト２　仮想マシン本体

```
67  /* PawnVM 本体 */
68  int amx_proc(){
69    int err;
70    AMX amx;
```

```
71    cell ret;
72    AMX_HEADER hdr;
73    void *prog;
74
75    UARTprintf("Pawn VM Init¥r¥n");
76
77    memcpy(&hdr, prog_data ,  sizeof(AMX_HEADER));    /* ヘッダー情報の取り出し */
78    UARTprintf("¥r¥nstp=%d size=%d¥r¥n¥r¥n",(int)hdr.stp,(int)hdr.size);
79
80    if (hdr.magic != AMX_MAGIC) {
81      UARTprintf("ByteCode Error!!!¥r¥n");            /* バイトコードが不正 */
82      return -1;
83    }
84
85    prog = alloca(hdr.stp);                           /* VM用の作業領域の確保 */
86    memcpy(prog, prog_data ,  hdr.size);              /* プログラムデータのコピー */
87
88    memset(&amx, 0, sizeof(amx));
89    err = amx_Init( &amx, prog );                     /* VMの初期化 */
90    err = amx_Register(&amx, MY_Natives, -1);         /* native関数の登録 */
91    amx_FloatInit(&amx);                              /* Floatモジュールの登録 */
92
93    UARTprintf("Pawn VM Exec¥r¥n");
94    err = amx_Exec(&amx, &ret, AMX_EXEC_MAIN);        /* VMの実行（メイン関数の呼び出し）*/
95    amx_Cleanup(&amx);                                /* VMの破棄 */
96
97    if (err != AMX_ERR_NONE){
98      UARTprintf("Run time error %d¥r¥n", err);       /* 実行時エラーが発生   */
99    }
100   else {
101     UARTprintf("Done!!!!¥r¥n");                     /* PawnVMが正常に終了  */
102   }
103
104   return err;
105 }
```

①
②
③

Pawnのバイトコードは下記の手順で実行されます.

①仮想マシン用の作業領域の準備（**リスト２**の①）

・バイトコード中のヘッダ情報から実行に必要なメモリサイズの情報を読み取る（77行目）

・バイトコードの正当性チェック（80行目～83行目）

・仮想マシン用の作業メモリ領域を確保する（85行目）

・確保したメモリー領域の先頭にバイトコードをコピー（86行目）

②仮想マシンの実行と破棄（**リスト２**の②）

・仮想マシンの生成（89行目）

・拡張機能の組み込み（90,91行目）

・仮想マシンの実行［メイン関数の呼び出し］（94行目）

・実行後の仮想マシンを破棄（95行目）

①の手順でバイトコード読み出しているヘッダー情報には，**表４**に示す情報が格納されています．そのため，ヘッダ部に書き込まれた情報を用いて与えられたバイトコードを実行するために必要なメモリーサイズを特定できるようになっています.

表４　バイトコードヘッダの詳細

メンバー名	バイト数	役割
size	4	バイトコード自体のサイズ
magic	2	署名
file_version	1	ファイルフォーマットのバージョン
amx_version	1	実行時に要求される仮想マシンの最小バージョン
flags	2	バイトコードの状態（デバッグ，スリープ等）を示すフラグ
defsize	2	native 関数テーブル及び public 関数テーブルのサイズ
cod	4	コードセクションのオフセット
dat	4	データセクションのオフセット
hep	4	ヒープ領域の先頭
stp	4	スタックトップ値（トータルのメモリ使用量になる）
cip	4	実行開始アドレス（メイン関数）
publics	4	public 関数テーブルのオフセット
natives	4	native 関数テーブルのオフセット
libraries	4	ライブラリのオフセット
pubvars	4	public 変数テーブルのオフセット
tags	4	public タグのオフセット
nametables	4	シンボルテーブルのオフセット

実行対象となるバイトコードですが，第 0 段階のシステムではプログラム中に埋め込んでいます．pawn_run.c の 34 行目と 35 行目の include 文がバイトコードの埋め込みに対応した部分になります．

```
34  #include "hello_dat.h"          リスト３に対応したバイトコードを使用する場合
35  //#include "float_test_dat.h"   リスト４に対応したバイトコードを使用する場合
```

それでは「スクリプト言語を使う意味が無い！！」とお叱りを受けそうですが，まずは仮想マシン自体が正しく動作するかを確認するため，プログラム中に埋め込んだバイトコードを実行するシステムとしています．

今回のテストでは，**リスト３**（表示機能のテスト），**リスト４**（浮動小数点計算のテスト）のスクリプトをコンパイルして得られたバイトコードを使用しています．

リスト３　テストスクリプト１（hello.p）

```
#pragma dynamic 1000
main(){
  printf "pawn for Stellaris ¥n"
}
```

仮想マシンが使用するスタックサイズを 1000 セルにする

リスト４　テストスクリプト２（float_test.p）

```
pragma dynamic 1000
#include <float>

main(){
    new Float:a;

    for (new i = 0; i <= 90; i=i+10){
       a = floatsin(3.14*i/180.0);
       printf("i = %2d , a=%7.3f ¥n",i,a);
    }
}
```

Float 型の変数を使用するため

変数 a を 0 から 90 まで 10 ずつ変えながら
sin 関数値を計算し
書式を指定して表示する

リスト３及び**リスト４**に先頭の「#pragma dynamic 1000」は，仮想マシンが使用するスタックサイズの指定をする際に使用する記述です．Pawn のコンパイラーはデフォルトでスタックとして 4000 セル(16K バイト)使用するバイトコードを生成します（コンパイラーへ与えるオプションでも指定はできます）．今回は，メモリの無駄使いを避けるため，スタックサイズを 1000 セル（4K バイト）に縮小しています．

プログラム中で取り込んでいる「hello_dat.h」や「float_test_dat.h」は，**リスト５**のシェルスクリプトを使用して対応するバイトコード（amx ファイル）から C 言語の配列へ変換した内容を含むファイルです．

リスト５　バイトコードを C 言語の配列に変換するスクリプト(convert)

```
#!/bin/sh
(echo "const unsigned char prog_data[] = {"; od  -txC -w8 $1 | ¥
sed -e "s/^[0-9]*/¥t/" -e s"/ ¥([0-9a-f][0-9a-f]¥)/0x¥1,/g" -e"¥$d" | ¥
sed -e"¥$s/,$/¥¥n};/") > $2
```

リスト５では，バイナリーファイルの内容をダンプする「od」コマンドの出力を，「sed」コマンドへ渡して文字列の置換処理を行い，C 言語の配列の体裁へ変換しています．例えば，MinGW のシェル上で

```
./convert hello.amx hello_dat.h
```

と入力することにより，Pawn のバイトコード(hello.amx)から C 言語のヘッダーファイル(hello_dat.h)への変換が実施されます．変換後のファイルは下記の様な内容となります．

```
const unsigned char prog_data[] = {
  0x97,0x00,0x00,0x00,0xe0,0xf1,0x0a,0x0a,
  0x04,0x00,0x08,0x00,0x60,0x00,0x00,0x00,
      中略
  0x80,0x6c,0x80,0x6c,0x80,0x61,0x80,0x72,
  0x80,0x69,0x80,0x73,0x20,0x0a,0x00
};
```

「hello_dat.h」に書き込まれたバイトコード配列を組み込んだシステムを実行した結果は

PawnVM Step0 Pawn VM Init stp=4236 size=151 Pawn VM Exec pawn for Stellaris Done!!!! PawnVM Stop	Pawn のスクリプトからの表示

となり，めでたく PawnVM が動作したことが確認できます．

PawnVM の出力を確認する際には，TeraTerm 等のターミナルエミュレーターソフトを使用し，ボーレートを 115200bps，データ長を 8 ビット，ストップ・ビットを 1 ビット，パリティは無し，フロー制御は無しの設定で PC と評価ボードを接続しておいてください．

また，本資料で作成するプログラムを LM4F232 評価ボードで実行する際には，LM Flash Programmer の「Configuration」タブの「Quick Set」を「LM4F232 Evaluation Board」に設定して書き込んでください．

一方，pawn_run.c の 34 行目をコメントアウトし，35 行目のコメントを外したプログラムを再ビルドした後に実行した結果は

```
PawnVM Step0
Pawn VM Init

stp=4660 size=338

Pawn VM Exec
i =  0 , a=  0.000
i = 10 , a=  0.174
```

```
    中略
i = 80 , a=  0.985
i = 90 , a=  1.000
Done!!!!

PawnVM Stop
```

となります．浮動小数点計算も動作することが確認できました．**リスト４**のスクリプト中の「floatsin」は C 言語側では「sin」(sinf ではなく)関数が呼び出されているため，ビルド時に「-mfpu=fpv4-sp-d16」を指定し忘れるとシステムが HardFault 状態に陥ってしまいます．

添付ファイルには**リスト５**によって変換したヘッダーファイルを同封していますが，別のスクリプトを試したい場合は，書籍の付属基板を対象にスクリプトを実行した時に生成されるバイトコード（ソースファイルと同じフォルダにあります）を**リスト５**で変換したものを組み込んでください．

仮想マシンを使っていますので，CPU が LM3S3748 であるか LM4F232 であるかにかかわらず実行を行うことができます．ただし，付属基板固有の機能を使ったスクリプトの実行をした場合は，「関数が定義されていない」という実行時エラーが発生します．

仮想マシン自体の動作確認が終わりましたので，バイトコードを「PC 側のアプリケーションから受け取る機能」や「microSD カードから読み出す機能」を追加することでスクリプト言語処理系らしくしてゆきたいと思います．

２　ステップ１：PC 側アプリケーションから Pawn のバイトコードを受け取って実行する

ステップ１では，書籍で使用している簡易開発環境（PawnDev.exe）からバイトコードを受信する機能を追加します．

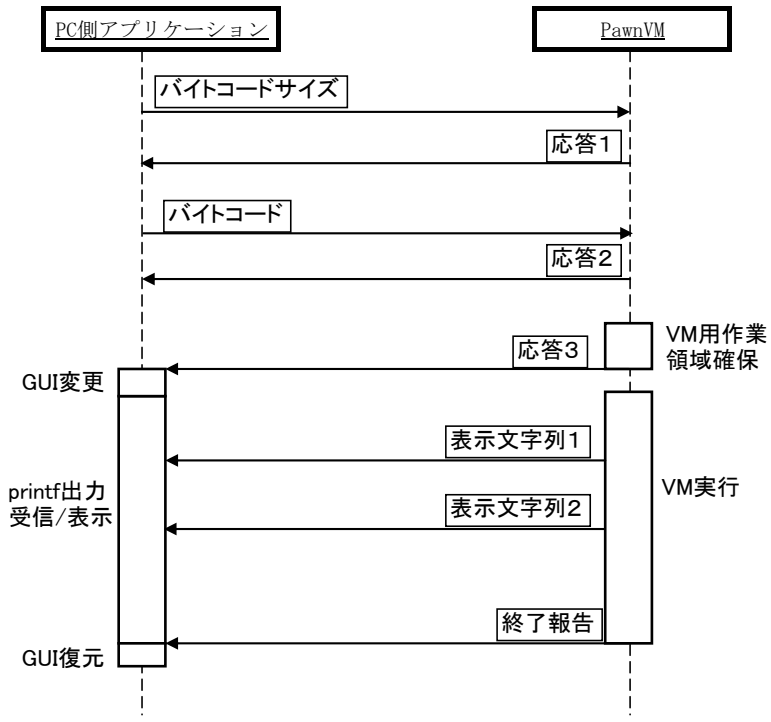

図２　PC 側アプリケーションと PawnVM の通信手順

PC 側アプリケーションでは，「Build」メニューの「run」を選択した場合，Pawn のスクリプトをコンパイルし，

「メニュー」及び「ボタン」の状態を変更した後に，**図２**に示す手順で基板上の PawmVM と通信を行っています.

図２の通信手順を実行する PawnVM 側の処理は**リスト６**及び**リスト７**になります.

リスト６　バイトコード受信処理部

```
97  void *prog;
98    unsigned char len_h , len_l , checkSUM;
99    unsigned int sum , i;
100   int  p_len ;
101   unsigned char *prog_data;
102   unsigned char mesBuffer[3];
103   unsigned long data_count;
104
105   /* 3バイト受信するまで待つ */                                                    ┐
106   do{                                                                            │
107     data_count = USBCDCBufferDataAvailable();                                    │
108   }while(data_count < 3);                                                        │
109                                                                                  │
110   USBCDCBufferRead(&len_l, 1);      /* バイトコード長（下位バイト）受信 */        │
111   USBCDCBufferRead(&len_h, 1);      /* バイトコード長（上位バイト）受信 */        │
112   USBCDCBufferRead(&checkSUM, 1);  /* チェックサム受信 */                        │
113                                                                                  │
114   sum = (len_h + len_l) & 0xFF;     /* チェックサムの計算 */                     │
115   if( sum == checkSUM) {                                                         ①
116     mesBuffer[0] = VMAck;           /* Ack の送信 */                             │
117   }                                                                              │
118   else {                                                                         │
119     mesBuffer[0] = VMSizeCSError;  /* チェックサムエラー発生 */                  │
120   }                                                                              │
121   mesBuffer[1] = len_l;                                                          │
122   mesBuffer[2] = len_h;                                                          │
123                                                                                  │
124   /* バイトコード長情報の送信 */                                                 │
125   USBCDCBufferWrite( (unsigned char *)mesBuffer, 3);                             │
126   if(mesBuffer[0] != VMAck ) return VMSizeCSError;  /* エラーが発生した場合は抜ける */ ┘
127
128   p_len = (len_h << 8) | len_l;      /* バイトコード長の計算 */                   ②
129   prog_data = alloca(p_len);        /* バイトコード格納領域の確保 */             ┘
130
131   /* バイトコードの受信 */                                                        ┐
132   data_count=0;                                                                  │
133   do                                                                             │
134   {                                                                              │
135     if(USBCDCBufferDataAvailable() > 0)                                          │
136     {                                                                            │
137       USBCDCBufferRead(&prog_data[data_count], 1);                               │
138       data_count++;                                                              │
139     }                                                                            │
140   }while(data_count < p_len);                                                    │
141                                                                                  │
142   /* チェックサム受信 */                                                       │
143   do{                                                                            │
144     data_count = USBCDCBufferDataAvailable();                                    │
145   }while(data_count < 1);                                                        ③
146   USBCDCBufferRead(&checkSUM, 1);                                                │
147                                                                                  │
148   sum = 0;                                                                       │
149   for(i=0 ; i< p_len ; i++) sum += prog_data[i];  /* チェックサムの計算 */       │
150   if( (sum & 0xFF) == checkSUM) {                                                │
151     mesBuffer[0] = VMAck;           /* Ack の送信 */                             │
152   }                                                                              ┘
```

```
153     else {
154       mesBuffer[0] = VMSizeCSError;  /* チェックサムエラー発生 */
155     }
156
157     /* 状況を報告 */
158     USBCDCBufferWrite( (unsigned char *)mesBuffer, 1);
159     if(mesBuffer[0] != VMAck ) return VMByteCodeCSError;  /* エラーが発生した場合は抜ける */
```

リスト６のバイトコード受信部では

・「バイトコードサイズ」の受信，チェックサム検証と結果の報告（**リスト６**の①）

・バイトコードサイズに誤りが無い場合にバイトコード格納領域の確保（**リスト６**の②）

・「バイトコード」の受信，チェックサム検証と結果の報告（**リスト６**の③）

を行っています.

リスト７　実行準備/終了報告部

```
161     memcpy(&hdr, prog_data ,  sizeof(AMX_HEADER));        /* ヘッダー情報の取り出し */
162
163     prog = alloca(hdr.stp);                /* VM 用の作業領域の確保 */
164     memcpy(prog, prog_data ,  hdr.size);  /* プログラムデータのコピー */
165
166     if (hdr.magic != AMX_MAGIC) {
167       mesBuffer[0] = VMByteCodeError;  /* バイトコードエラー発生 */
168     }
169     else mesBuffer[0] = VMAck;          /* Ack の送信 */
170
171     mesBuffer[1] = ((int)hdr.stp) & 0xFF;         /* 作業領域サイズ（下位）  */
172     mesBuffer[2] = ((int)hdr.stp >> 8) & 0xFF ;  /* 作業領域サイズ（上位）  */
173
174     /* スタックサイズ情報を送信 */
175     USBCDCBufferWrite( (unsigned char *)mesBuffer, 3);
176     if(mesBuffer[0] != VMAck ) return VMByteCodeError; /* エラーが発生した場合は抜ける */
177
178     memset(&amx, 0, sizeof(amx));
179     err = amx_Init( &amx, prog );              /* VM の初期化 */
180     err = amx_Register(&amx, MY_Natives, -1);  /* native 関数の登録 */
181     amx_FloatInit(&amx);                       /* Float モジュールの登録 */
182
183     mwait(50);
184
185     err = amx_Exec(&amx, &ret, AMX_EXEC_MAIN);/* VM の実行 */
186     amx_Cleanup(&amx);                /* VM の破棄 */
187
188     if (err != AMX_ERR_NONE){
189       /* runtime エラー情報の送信 */
190       mesBuffer[0] = VMRTError;        /* runntime error が発生 */
191       mesBuffer[1] = err;              /* error no の送信       */
192       mesBuffer[2] = VMRTError;        /* runntime error が発生 */
193       USBCDCBufferWrite( (unsigned char *)mesBuffer, 3);
194     }
195     else {
196       mesBuffer[0] = VMProgEnd;   /* プログラムが終了したことをホストアプリに伝える */
197       USBCDCBufferWrite( (unsigned char *)mesBuffer, 1);
198     }
```

バイトコードを受信した後の処理は，

・バイトコードの正当性チェックと結果の報告（**リスト７**の①）

・バイトコードの実行（ステップ０と同じ処理）

・終了報告データの送信（**リスト７**の②）

となっています.

バイトコードの正当性チェックの報告を受け取った PC 側アプリは，データ受信待ち状態に入り，終了報告データを受信するまでは，受信した文字列をひたすら「テキストボックス」へ表示する処理を行っています.

バイトコードの実行が終了した際に，PawnVM 側では amx_Exec 関数の戻り値を元に，「正常終了」の場合は 1 バイトの終了報告データを送信し，「実行時エラーによる終了」の場合はエラーコードを含む 3 バイトの終了報告データを送信しています．終了報告データを受信した PC 側アプリは，「メニュー」及び「ボタン」の状態を元に戻す処理を行っています.

なお，このシステムでは書籍のシステムでも使用している CDC ドライバー（LM4F232 用にハードウェア設定を変更）を用いて PC との通信を行っています.

バイトコードを外部から受け取る処理を追加しましたので，ビルドし直すことなくスクリプトの修正を行うことができるようになり，やっとスクリプト言語処理系らしくなりました.

３　ステップ２：仮想マシンの実行を中断できるようにする

ステップ１のシステムでは，「無限ループを含む Pawn のスクリプトを実行してしまった場合」に PC 側アプリは終了報告データ永遠に待つことになり，PC 側アプリの再起動と PawnVM の再起動を余儀なくされます．それでは不便ですので，ステップ２では外部から PawnVM を強制終了する機能を追加してみたいと思います.

ステップ１のプログラム（pawn_run.c）へ**リスト８**の修正（赤字部分）を加えることで，仮想マシンを強制終了する機能が追加されます.

リスト８　強制終了機能

```
51  AMX g_amx;                                          /* 変数 amx をローバル変数 g_amx へ変更 */
52  static volatile tBoolean g_PAWNVMBreak = false;     /* 終了要求発生フラグ */
53  int AMXAPI prun_Monitor(AMX *amx);                  /* 強制終了 hook 関数プロトタイプ */
54
55  /* 終了要求監視関数 */                                                              ┐
56  void USBCDCOptFunc()                                                               │
57  {                                                                                  │
58    unsigned char cmd;                                                               │
59    unsigned long data_count;                                                        │
60                                                                                     │
61    data_count = USBCDCBufferRead(&cmd, 1);                                          │
62    if(data_count)                                                                   │
63    {                                                                               ②
64      /* pwanVM 実行中に終了要求があった場合はを hook 関数を登録 */                    │
65      if( cmd == '¥v' ){                                                           │
66        if(g_PAWNVMBreak == false) {                                               │
67          g_PAWNVMBreak = true;                                                    │
68          amx_SetDebugHook(&g_amx, prun_Monitor);     /* Hook 関数の登録 */        │
69        }                                                                          │
70      }                                                                            │
71    }                                                                              │
72  }                                                                                ┘

        中略

115 /* pwanVM を強制終了させる hook 関数 */                                          ┐
```

```
116 int AMXAPI prun_Monitor(AMX *amx)                                    ┐
117 {                                                                    ①
118   return AMX_ERR_EXIT;                                              │
119 }                                                                    ┘
120
121 /* PawnVM 本体 */
122 int amx_proc(){
123   int err;
124   cell ret;                              /* amx_proc 内で定義していた amx は削除 */

        中略

207   memset(&g_amx, 0, sizeof(g_amx));
208   err = amx_Init( &g_amx, prog );              /* VM の初期化 */
209   err = amx_Register(&g_amx, MY_Natives, -1);  /* native 関数の登録 */
210   amx_FloatInit(&g_amx);                       /* Float モジュールの登録 */
211
212   g_PAWNVMBreak = false;
213   USBCDCIntRxExtRegister( USBCDCOptFunc );     /* 終了要求監視関数の登録 */
214
215   mwait(50);
216
217   err = amx_Exec(&g_amx, &ret, AMX_EXEC_MAIN);  /* VM の実行 */
218   amx_Cleanup(&g_amx);                           /* VM の破棄 */
219
220   if (err != AMX_ERR_NONE){
221     if(g_PAWNVMBreak)                                                ┐
222     {                                                                │
223       mesBuffer[0] = VMAbort;    /* ホストアプリからの要請で仮想マシンが Break した事を伝える */  ③
224       USBCDCBufferWrite( (unsigned char *)mesBuffer, 1);             │
225     }                                                                ┘
226     else{
227     /* runtime エラー情報の送信 */
228       mesBuffer[0] = VMRTError;    /* runntime error が発生 */
229       mesBuffer[1] = err;          /* error no の送信       */
230       mesBuffer[2] = VMRTError;    /* runntime error が発生 */
231       USBCDCBufferWrite( (unsigned char *)mesBuffer, 3);
232     }
233   }
234   else {
235     mesBuffer[0] = VMProgEnd;      /* プログラムが終了したことをホストアプリに伝える */
236     USBCDCBufferWrite( (unsigned char *)mesBuffer, 1);
237   }
238
239   USBCDCIntRxExtUnregister();       /* 終了要求監視関数の取り外し */
```

強制終了機能は，Pawn の Debughook 機能を使って実現しています．本来はバイトコード実行中にデバッグを行うための機能ですが，仮想マシンへのポインタを引数に持ち，「AMX_ERR_EXIT」を返す関数（**リスト８**の①）を amx_SetDebugHook 関数を用いて実行中の仮想マシンに登録することにより（**リスト８**の 68 行目），仮想マシンを強制終了させることができます．また，amx_SetDebugHook 関数を使用するにあたって，関数 amx_proc 内で定義していた仮想マシンを指し示す変数 amx をグローバル変数 g_amx へ変更しています．

amx_SetDebugHook を PC 側アプリケーションの要求により発動する仕組みですが，仮想マシンを起動する直前に終了要求監視関数（リスト８の②）を CDC ドライバーの受信割り込み処理へ追加し（**リスト８**の 213 行目），Pawn スクリプト実行中に PC 側アプリにおいて「VM Stop」ボタンが押された時に送られるメッセージを捕捉し，Hook 関数の登録を行っています．また，バイトコードの実行が終了した後は，監視の必要がありませんので，終了要

求監視関数を取り外しておきます（**リスト８**の 239 行目）．PC 側アプリの要求によって実行を停止した場合は，強制終了に対応する終了報告データの送信を行っています（**リスト８**の③）．

４　ステップ３：ハードウェアアクセス関数を追加する

ステップ３では，いよいよハードウェアアクセス関数（GPIO 関係）を追加します．本資料では，書籍版とは趣向を変えて StellarisWare で用意されている API をそのまま呼び出す形の拡張機能を実装してみたいと思います．

ここでは，**表４**に示す GPIO 関係の機能を PawnVM に追加します．

表４　ステップ３で追加する機能

関数名	引数/戻り値	動作
mwait(pause_ms)	pause_ms：ミリ秒単位の休止時間 戻り値：無し	指定された時間だけ処理を止める
void SysCtlPeripheralEnable (　ulPeripheral)	ulPeripheral：有効にする周辺回路のアドレス 　SYSCTL_PERIPH_GPIOA 等 戻り値：無し	指定した周辺回路を有効にする
void GPIOPinTypeGPIOOutput (　nsigned long ulPort, 　unsigned char ucPins, 　unsigned long ulStrength)	ulPort：ベースアドレス ucPins：ピンマスク ulStrength：ドライブ強度 　下記の何れかの値，省略可 　GPIO_STRENGTH_2MA　　：出力強度＝2mA（デフォルト） 　GPIO_STRENGTH_4MA　　：出力強度＝4mA 　GPIO_STRENGTH_8MA　　：出力強度＝8mA 　GPIO_STRENGTH_8MA_SC　：出力強度＝8mA slew 制御付き 戻り値：無し	指定したピンをディジタル出力として使用する
void GPIOPinTypeGPIOIutput (　unsigned long ulPort, 　unsigned char ucPins, 　unsigned long ulPinType)	ulPort：ポートのベースアドレス ucPins：ピンマスク ulPinType：ピンのタイプ 　下記の何れかの値,省略可 　GPIO_PIN_TYPE_STD　　：　標準 　GPIO_PIN_TYPE_PULLUP　：　プルアップ(デフォルト) 　GPIO_PIN_TYPE_PULLDOWN　：　プルダウン 戻り値：無し	指定したピンをディジタル入力として使用する
void GPIOPinWrite (　unsigned long ulPort, 　unsigned char ucPins, 　unsigned char ucVal)	ulPort：ポートのベースアドレス ucPins：ピンマスク ucVal　：設定値 戻り値：無し	指定したピンの状態を設定する
long GPIOPinRead (　unsigned long ulPort, 　unsigned char ucPins)	ulPort：ポートのベースアドレス ucPins：ピンマスク 戻り値：ピンの状態	指定したピンの状態を読み出す

表４の拡張機能では，Pawn のスクリプトと C 言語の関数の間での引数の受け渡し方法が全て「値渡し」ですので，Pawn から受け取った引数を所定の型へキャストを行い，C 言語の関数へ受け渡す働きのグルー関数(Glue Function)を定義します．グルー関数の例は**リスト９**（指定したピンを GPIO 出力として設定する）となります．

リスト９　指定したピンを GPIO 出力とする拡張機能

```
125 /* 指定したピンを GPIO 出力として設定 */
126 static cell AMX_NATIVE_CALL n_PinTypeGPIOOutput(AMX *amx,const cell *params)
127 {
128   unsigned long ulPort     = (unsigned long)params[1];                    第 1 引数のキャスト
```

```
129    unsigned char ucPins     = (unsigned char)params[2];          第２引数のキャスト
130    unsigned long ulStrength = (unsigned long)params[3];          第３引数のキャスト
131
132    MAP_GPIODirModeSet( ulPort , ucPins , GPIO_DIR_MODE_OUT);     GPIO 出力とする
133    MAP_GPIOPadConfigSet( ulPort , ucPins , ulStrength , GPIO_PIN_TYPE_STD );   ドライブ強度を設定
134
135    return 0;                                                     戻り値はなし
136  }
```

リスト９のグルー関数では，Pawn のスクリプトから渡された 3 つの引数をキャストし，StellarisWare で提供されている「MAP_GPIODirModeSet」と「MAP_GPIOPadConfigSet」へデータを引き渡し，指定されたピンをドライブ強度の指定付きで GPIO 出力とする設定を行っています.

新しく作成したグルー関数郡を PawnVM へ登録するためには関数リストを作成する必要がありまので，これまで printf しか登録していなかった native 関数リストへ新たに作成した６個のグルー関数を追記します(**リスト１０**).

リスト１０　native 関数リスト

```
172  /* native 関数リスト */
173  const AMX_NATIVE_INFO MY_Natives[] =
174  {
175    { "printf"                 , n_printf },
176    { "mwait"                  , n_mwait },
177    { "SysCtlPeripheralEnable" , n_SysCtlPeripheralEnable },
178    { "PinTypeGPIOOutput"      , n_PinTypeGPIOOutput },
179    { "PinTypeGPIOInput"       , n_PinTypeGPIOInput },
180    { "GPIOWrite"              , n_GPIOWrite },
181    { "GPIORead"               , n_GPIORead },
182    { NULL , NULL }        // terminator
183  };
```

さらに，新しく作成した native 関数を Pawn のコンパイラへ認識させるためのヘッダーファイル（**リスト１１**）を作成してハードウェアアクセス関数（GPIO 関係）の追加作業が完成です．このヘッダーファイルは「host_app¥include」フォルダーへ配置します.

リスト１１　新たに追加した native 関数用のヘッダーファイル（LM4F232.inc）

```
1    #if defined _lm4f232_included
2      #endinput
3    #endif
4    #define _lm4f232_included
5    #pragma library LM4F232
6
7    #define SYSCTL_PERIPH_GPIOG 0x20000040                  driverlib¥sysctrl.h
8    #define SYSCTL_PERIPH_GPIOM 0xf000080B                  から引用
9
10   #define GPIO_PORTG_BASE 0x40026000                      inc¥hw_memmap.h
11   #define GPIO_PORTM_BASE 0x40063000                      から引用
12
13   #define GPIO_PIN_0 0x00000001                           driverlib¥gpio.h
14   #define GPIO_PIN_1 0x00000002                           から引用
15   #define GPIO_PIN_2 0x00000004
16   #define GPIO_PIN_3 0x00000008
17   #define GPIO_PIN_4 0x00000010
18   #define GPIO_PIN_5 0x00000020
19   #define GPIO_PIN_6 0x00000040
20   #define GPIO_PIN_7 0x00000080
21
22   #define GPIO_PIN_TYPE_STD     0x00000008
23   #define GPIO_PIN_TYPE_STD_WPU 0x0000000A
24   #define GPIO_PIN_TYPE_STD_WPD 0x0000000C
```

```
25
26  #define GPIO_STRENGTH_2MA     0x00000001
27  #define GPIO_STRENGTH_4MA     0x00000002
28  #define GPIO_STRENGTH_8MA     0x00000004
29  #define GPIO_STRENGTH_8MA_SC 0x0000000C
30
31  native mwait( pause_ms );                                                    native 関数の
32  native SysCtlPeripheralEnable( ulPeripheral );                               プロトタイプ宣言
33  native PinTypeGPIOOutput( ulPort , ucPins , ulStrength = GPIO_STRENGTH_2MA );
34  native PinTypeGPIOInput( ulPort , ucPins , ulPinType = GPIO_PIN_TYPE_STD_WPU );  デフォルト値を指定
35  native GPIOWrite( ulPort , ucPins , ucVal );                                 した引数は省略可
36  native GPIORead( ulPort , ucPins );
```

リスト１１は LM4F232 評価ボード上のスイッチ類と LED を使うための最小限の構成になっています．作成した拡張機能を使用したサンプル・スクリプトは**リスト１２**となります．このスクリプトでは評価ボード上の▲(UP)スイッチ(PM0 に接続)が押されるまで，評価ボード上の LED（PG2 に接続）の状態を 100ms おきにトグルしています．StellarisWare 用意されている API をそのまま呼び出す形の拡張機能を作成しましたので，C 言語で書いたプログラムとほとんど同じ見た目のスクリプトになります．

リスト１２　GPIO テストスクリプト（gpio_test.p）

```
#include <LM4F232>                                                  LM4F232 評価ボードの GPIO を使用する
#pragma dynamic 1000

main()
{
  SysCtlPeripheralEnable( SYSCTL_PERIPH_GPIOG );                    GPIOG を有効にする
  PinTypeGPIOOutput( GPIO_PORTG_BASE , GPIO_PIN_2 );                PG2 を GPIO 出力（ドライブ強度 2mA）に設定

  SysCtlPeripheralEnable( SYSCTL_PERIPH_GPIOM );                    GPIOM を有効にする
  PinTypeGPIOInput( GPIO_PORTM_BASE , GPIO_PIN_0 );                 PM0 を GPIO 入力（プルアップ）に設定

  GPIOWrite( GPIO_PORTG_BASE , GPIO_PIN_2 , GPIO_PIN_2 );           PG2 へ 1 を書き込む
  new up = GPIORead( GPIO_PORTM_BASE , GPIO_PIN_0 );                PM0 の状態を読み込む

  new led_stat;
  while( up == GPIO_PIN_0 ) {                                       PM0 が押されていない間
    mwait(100);                                                       100ms 休止する
    led_stat =( GPIORead( GPIO_PORTG_BASE , GPIO_PIN_2 ) );           PG2 の状態を読み込む
    GPIOWrite( GPIO_PORTG_BASE , GPIO_PIN_2 , ~led_stat  );           PG2 の状態を反転する
    up = GPIORead( GPIO_PORTM_BASE , GPIO_PIN_0 );                    PM0 の状態を読み込む
  }                                                                 を繰り返す
}
```

５　ステップ４：microSD からバイトコードを取得する

ここまでは，PC 側アプリから転送されたバイトコードを実行するシステムを作成して来ましたが，ステップ４ではバイトコードを評価ボード上の microSD カードから読み出すシステムを作成してみます．

ステップ０で作成したシステムをベースに，SD カードからの読み込み機能及びステップ 3 で作成したグルー関数群を追加したシステムとしています．

リスト１３　microSD カードから Pawn バイトコードを読み出す機能の追加

```
150   AMX_HEADER hdr;
151   void *prog;
152   unsigned char *prog_data; /* バイトコード格納用配列 */
153
```

```
154     FATFS fatfs;        /* FAT ファイルシステム */
155     FIL   fp;           /* ファイルオブジェクト */
156     unsigned short bytesRead;
157
158     /* microSD カードをマウントする */                                    ┐
159     if( f_mount(0 , &fatfs) != FR_OK ){                                  │
160       UARTprintf("mount error¥r¥n");                                     │
161       return 0;                                                          │
162     }                                                                    ┘
163                                                                          ①
164     /* バイトコードファイルをオープンする */                              ┐
165     if( f_open( &fp , "boot.amx" , FA_READ | FA_OPEN_EXISTING ) != FR_OK ){ │
166       UARTprintf("boot.amx not found¥r¥n");                              │
167       return 0;                                                          │
168     }                                                                    ┘
169
170     UARTprintf("Pawn VM Init¥r¥n");
171
172     /* バイトコードのヘッダ部を読み出す */                                ┐
173     f_read( &fp , &hdr , sizeof(AMX_HEADER) , &bytesRead );              ②
174     UARTprintf("¥r¥nstp=%d size=%d¥r¥n¥r¥n",(int)hdr.stp,(int)hdr.size); ┘
175
176     if (hdr.magic != AMX_MAGIC) {
177       UARTprintf("ByteCode Error!!!¥r¥n");    /* バイトコードが不正 */
178       return -1;
179     }
180
181     UARTprintf("code size = %d¥r¥n",hdr.size);
182     prog_data = alloca(hdr.size);             /* バイトコード格納用配列の確保 */
183
184     f_lseek( &fp , 0 );                                  /* 読み出し位置を先頭へ戻す */  ┐
185     f_read( &fp , prog_data , hdr.size , &bytesRead );   /* バイトコードを読み出す */    ③
186     f_close(&fp);                                        /* ファイルをクローズする */    ┘
187
188     prog = alloca(hdr.stp);                  /* VM 用のスタック領域の確保 */
189     memcpy(prog, prog_data ,  hdr.size);     /* プログラムデータのコピー */
190
191     memset(&amx, 0, sizeof(amx));
192     err = amx_Init( &amx, prog );           /* VM の初期化 */
```

StellarisWare 付属の FatFs ライブラリーを利用することで，microSD カードから Pawn のバイトコードを読み込む機能の追加は**リスト１３**の赤字の部分の修正で済んでしまいます．ステップ０のシステムに対して追加した処理としては，

・バイトコードファイル（boot.amx）を開く（**リスト１３**の①）

・バイトコードのヘッダー情報をファイルから読み出す（**リスト１３**の②）

・バイトコードをファイルから読み出す（**リスト１３**の③）

となります．

実行したいバイトコードを「boot.amx」というファイル名で microSD カードへ保存し，評価ボードのカード・スロットへ差し込んだ状態で，電源を供給すると Pawn プログラムの実行が開始されます．

６　Pawn4 への対応

現在最新の Pawn 言語のバージョンは 4.0.4548（書籍及び本資料ではバージョン 3.3.3785 を使用）となっていますので，ここでは Pawn の最新版を使う場合の修正点をまとめておきます．

Pawn のバージョン４系列は,それまでのバージョンとの互換性がありませんので,マイコン上で動作する PawnVM と PC 側で動作させている Pawn のコンパイラの双方とも入れ替える必要があります.

最新版（4.0.4548）を使う場合は，まず CompuPhase 社のサイト（http://www.compuphase.com/pawn/pawn.htm）から Pawn のソースコード[2]またはインストーラ[3]を入手します.

６－１　PawnVM のバージョンアップ

解凍先（またはインストール先）のフォルダー中の「source¥amx」の中に仮想マシン及び各種拡張機能のソースファイルが格納されていますので，**表１**のファイルを新しいファイルへ差し替えます．さらにバージョン４から gcc 向けの部分が独立したファイルになりましたので,「amxexec_gcc.c」を追加します．また，Float モジュールを組み込む際に使用している「float.c」の 332 行目がコメントブロックの入れ子になっていますので，先頭の/を削除しておいてください.

本資料や書籍で使用している Stellaris 固有の機能に関しては，ビルド時のマクロ定義によって処理を切り替えるように作成してありますので，**表２**のマクロ定義に加えて PAWN4 をマクロ定義した上でビルドを行ってください．以上でバージョン４の仮想マシンは完成です.

マクロ定義 PAWN4 を追加した理由ですが，バージョン４より仮想マシンから C 言語の関数へ参照渡しを行う際の API が変更になったことへ対応するためです.

具体的には，バージョン３までは，仮想マシンから渡された params[1]と C 言語上の cell 型のポインタ変数（*cptr）との間でアドレス変換を行う際には,

```
amx_GetAddr( amx , param[1] , &cptr);
```

と記述していましたが，バージョン４では,

```
cptr = amx_Address( amx , params[1] );
```

と記述します．この変更に対応するために Stellaris 固有の機能を実装している部分では，次の様にマクロ定義 PAWN4 を使って，バージョン４とそれ以外の場合で API 関数を使い分けるように記述しています.

```
#if defined PAWN4
  cptr = amx_Address( amx , params[1] );
#else
  amx_GetAddr( amx , params[1] , &cptr );
#endif
```

６－２　PC 側アプリのバージョンアップ

PC 上で動作するコンパイラの入れ替えを行うのですが，バージョン４以前はコンパイラー本体は DLL ファイル(libpawnc.dll)での提供であったのに対して，バージョン４では exe ファイル（pawncc.exe）での提供に切り替変わっています．そのため PC 側アプリを修正する必要があります.

書籍ウェブ・ページからダウンロードした pawn_cq.zip の中には PC 側アプリケーションのソースファイルも同封していますので，下記の修正を加えることで「pawncc.exe」を使った Pawn スクリプトのコンパイルが可能になります．修正箇所としては，PawnDevForm.cs 中の「Run」メニューに対するイベントハンドラー（miRun_Click）

において DLL 内の関数 pc_compile を呼び出している部分になります.

修正前

```
string[] argv = new string[5];
argv[0] = "";
argv[1] = "-i ." + Path.DirectorySeparatorChar + "include";
argv[2] = "-D" + sourcePath;
argv[3] = "-epawn_err";
argv[4] = sourcePath + Path.DirectorySeparatorChar + sourceName;

string error_file = sourcePath + Path.DirectorySeparatorChar + "pawn_err";
System.IO.File.Delete(error_file);
pawn_compile(argv);
```

修正後

```
string error_file = sourcePath + Path.DirectorySeparatorChar + "pawn_err";
System.IO.File.Delete(error_file);

Process proc;
ProcessStartInfo psi = new ProcessStartInfo();

// 外部アプリ(pawncc.exe)とオプションの指定
psi.FileName = System.AppDomain.CurrentDomain.BaseDirectory + Path.DirectorySeparatorChar + "pawncc.exe";
psi.Arguments = "";
psi.Arguments += " " + "-D" + sourcePath;
psi.Arguments += " " + "-epawn_err";
psi.Arguments += " " + sourcePath + Path.DirectorySeparatorChar + sourceName;

psi.CreateNoWindow = true;          // 別途ウィンドウを開かない
psi.RedirectStandardOutput = true; // 標準出力をリダイレクト
psi.RedirectStandardError = true;  // 標準エラーをリダイレクト
psi.UseShellExecute = false;        // シェル機能は使わない
psi.ErrorDialog = true;             // プロセスを起動できなかった時にダイアログを出す

proc = Process.Start(psi);          // プロセスを起動

proc.WaitForExit();                 // プロセス(pawncc.exe)が終了するまで待つ

proc.Close();                       // 使用したリソースの後片付け
proc.Dispose();
```

修正を最小限にするために，Pawn のスクリプト中にエラーがあった場合にファイルに書き出しています.

また，同ファイルの先頭部分で DLL をインポートしている箇所は削除（またはコメントアウト）してください.

```
[DllImport("libpawnc.dll")]
extern static int pc_compile(int argc, string[] argv);
public static void pawn_compile(string[] args)
{
    pc_compile(args.Length, args);
}
```

“C#”でもマクロ定義が使用できますので，#if PAWN4・・・#else ・・・#endif の形でバージョン定義によって使い分ける形式でも構いません.

リンク集

[1] Pawn 3.3.3785 ソースファイル：http://www.compuphase.com/pawn/pawn-3.3.3875.zip

[2] Pawn 4.0.4548 ソースファイル：http://www.compuphase.com/pawn/pawn-4.0.4548.zip

[3] Pawn 4.0.4548 インストーラ：　http://www.compuphase.com/pawn/pawn-4.0.4548.exe