

テーマ3：組み込みシステムプログラミング

1. 目的

組み込みシステムとは、PC などの汎用計算機とは異なり、家電機器やパソコン周辺機器、ゲーム機器、OA 機器から、自動車や人工衛星まで、「機械や機器に組み込まれて、その制御を行うコンピュータシステム」のことである。最近の大規模システムでは、PC ベースのハードウェアが用いられる場合もあるが、ここでは、組み込みシステムを、ある応用に専用化されたコンピュータシステムと定義する。一般に、組み込みシステムのハードウェアは、制御する対象の機器によって大きく異なる。機械や機器の種類に応じて、各種のセンサやアクチュエータ、通信インタフェースをもっており、周辺デバイスは特殊なものが多い。そのため、ソフトウェア開発の中で、デバイスドライバなど、ハードウェアを直接扱う部分の比率が大きい。ハードウェアを直接扱うソフトウェアの開発は、ハードウェアに関する知識が必要であることに加えて、動作がタイミングに依存する場合も多く、開発が難しいと言われている。本実験では、任天堂ゲームボーイアドバンス(GBA)のハードウェアを利用し、通常の汎用計算機と異なる組み込みシステムのプログラミングの基本を理解することを目的とする。(ゲーム作成が目的ではないが、プログラムに趣向を凝らすことは望ましい)

2. 基礎知識

GBA のプログラミングを行うにあたり、コンピュータアーキテクチャ(計算機工学)におけるメモリアドレス、レジスタ、ポーリング・割り込みに関する知識、アセンブリ言語、および、簡単な C 言語によるプログラミング能力(特にポインタの概念)が必要となる。

2.1. ハードウェア構成

図 1 のように、GBA は、複数のハードウェアの要素で構成されている。実験で関係するのは、メインプロセッサ、内部 RAM、ビデオ(VRAM)、および、キーコントローラである。

2.1.1. メインプロセッサ

メインプロセッサは英国メーカが開発した ARM を利用している。ARM は、iPhone をはじめとする携帯電話などの組み込みシステムで幅広く利用されている。ARM は 32 ビットの RISC プロセッサであり、低消費電力で動作することが特徴である。プロセッサ内部に 16 個の 32 ビットレジスタ(R0 から R15 まで)が内蔵されているが、R15 はプログラムカウンタ(PC)として利用され、R13 と R14 も特殊用途で利用するため、実験のプログラムで利用可能なレジスタは、R0 から R12 までである。ARM 自身はビッグ・エンディアン、リトル・エンディアンの両方をサポートしているバイ・エンディアンのプロセッサであるが、GBA の設定においては、リトル・エンディアンに固定されている。

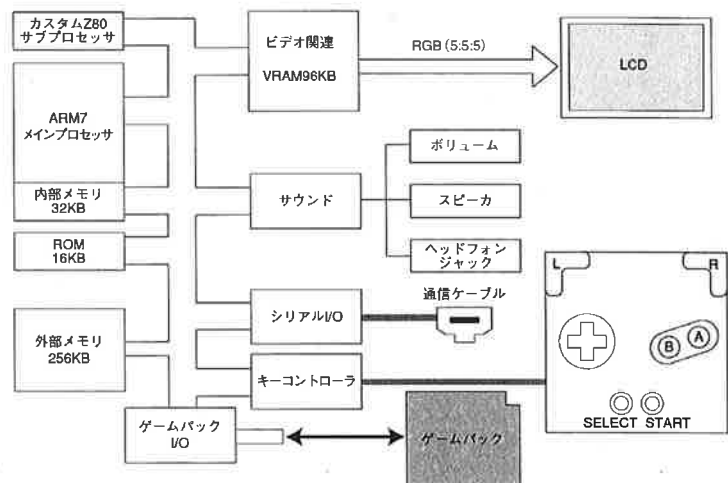


図 1: GBA ハードウェア構成

1 周辺機器を動作させるためのソフトウェア

2.1.2. 外部 RAM

外部 RAM は、実験で作成するプログラムを格納するメモリである。PC 上で作成したプログラムは、外部 RAM ヘダウンロードして、動作させることができる。外部 RAM の領域は、0x0200:0000²から 0x0203:FFFFまでの 256k バイト、バス幅は 16 ビットである。

2.1.3. VRAM

(GBA には 4 種類の表示モードがあるが、実験ではモード 3 を利用する。今は、モードについて意識しなくてよい) VRAM にデータを書き込むと、そのデータが液晶ディスプレイに表示される。VRAM の領域は、0x0600:0000 から 0x0601:7FFFまでの 96k バイト、バス幅は 16 ビットである(この VRAM のメモリは、実際の画面のサイズより大きい領域となる)。液晶ディスプレイは 240×160 ドット(ピクセル、ポイント)で構成され、左上の 0 ドット目が 0x0600:0000 番地からの 2 バイトに相当する。1 ドットは、メモリ上で「0BBBBBGGGGGRRRRR」の 16 ビットで構成され、青、緑、赤のそれぞれに各 5 ビットの階調、すなわち、0 から 31 の階調で表わされる。たとえば、「0000000000000000」(2 進数)の 16 ビットであれば、そのドットは黒色、「0111111111111111」の 16 ビットであれば、そのドットは白色となり、「0111110000000000」の 16 ビットであれば、そのドットは(最も濃い)青色となる。図 2 に VRAM と表示ドットの関係を示す。画面は横が 240 ドットなので、上から 2 列目の左端のドットのアドレスは、0x0600:01E0 となる。VRAM に何もデータを書き込まない初期状態において、どのような色が設定されるかについて、規定はないが、画面は「黒色」となるようである。

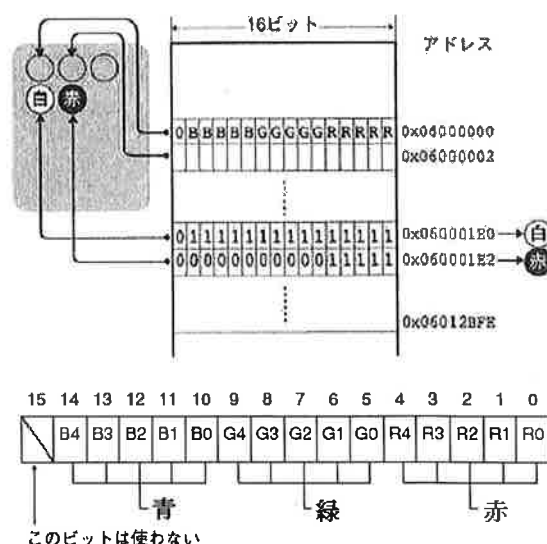


図 2: VRAM と表示ドットの関係

2.1.4. キーコントローラ

GBA のキー(ボタン)は、「START」、「SELECT」、「A」、「B」、十字キーの「↓」、「↑」、「←」、「→」、および、後ろ側の「L」、「R」の 10 個である。これらは、アドレス 0x0400:0130 番地に、図 3 のように割り当てられている。

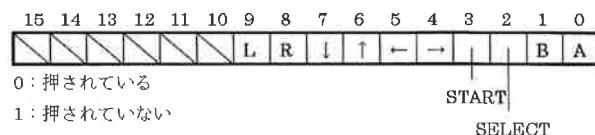


図 3: キー(ボタン)入力状態

キー(ボタン)が押されると、そのビットが 0 となり、押されていないと 1 となる。どのキー(ボタン)も押されていない状態では、このアドレスの第 0 ビットから第 9 ビットのビットが「1111111111」となり、たとえば、「A」ボタンが押されると、「1111111110」となる。第 10 ビットから第 15 ビットの値は、0 か 1 のどちらの値となるかは「不定」である。

2.2. プログラム開発

一般に、電化製品などのマイコン制御チップや家庭用のゲーム機などの組み込みシステムは、それ自体では開発環境を持たないため、PC など別のコンピュータを使ってソフトウェアを開発し、完成したソフトウェアを実機(ターゲット)に送り込んで実行するという手法が採られる。

² ここでは、32 ビットのアドレスを読みやすいように、16 ビットごとにコロンの(:)で区切る。

2.2.1. クロス開発

あるソフトウェアの開発を、そのソフトウェアが動作するシステムとは違うシステム上で開発することを、クロス開発と呼ぶ。ここでは、GBA のプログラムを、GBA 上で開発するのではなく、PC を利用して開発を行う。その際、プログラマが記述したソースコードは、ターゲットとなるシステムで動作するバイナリに変換する必要がある。こうした変換機能を持った特殊なコンパイラやアセンブラを、クロスコンパイラ、あるいはクロスアセンブラという。

2.2.2. 組み込みシステムプログラミングの注意点

組み込みシステム向けのプログラムを行う際、ハードウェアの構成を認識しておく必要がある。通常、ターゲットのメモリ構成は、プログラム開発を行う PC と異なり、メモリの幅が 16 ビットや 8 ビットの場合がある。たとえば、特定のアドレスを 32 ビット(たとえば, unsigned long)としてアクセスしても、メモリ幅が 16 ビットとして割り当てられていると、データが 16 ビット分しか有効でない場合がある(ハードウェアによっては、32 ビットデータを返す場合もある)。

また、特に、入出力を操作するプログラムの場合、時間を認識してプログラムする必要がある。キー(ボタン)入力を取り込むプログラム作成において、ユーザがキー(ボタン)を押してから離すまでの間に、キー(ボタン)入力を取り込まないと、ユーザの操作をプログラムが認識できないことになる。

割り込みや、DMA(Direct Memory Access)の操作についても、プログラムを作成する際に注意すべきである。(今回の実験においては、割り込み、DMA は利用しなくてもよい)

2.2.3. 開発環境

プログラム開発は、Linux 上で行う。Linux には Debian, Fedora, Ubuntu, Red Hat など複数の配布形態(ディストリビューション)があるが、ここでは USB メモリで立ち上がる Ubuntu と呼ぶ Linux ディストリビューションを利用して開発を行う。Ubuntu におけるファイルはすべてメモリ上に構成されるため、PC をシャットダウンすると、作成したファイルなどはすべて消去される。ファイル作成はデータ保存 USB メモリのフォルダ上で行うことが必要である。Ubuntu の具体的操作手順については、第 11 章で解説するのでこれに従うこと。

図 4 に C 言語プログラムからアセンブリ言語、オブジェクトファイル、実行ファイルを経由して、バイナリファイルへの変換の手順を示す。C 言語のプログラムのファイル(FileName.c)は、「コンパイラ」でアセンブリ言語のファイル(FileName.s)に変換(コンパイル)され、「アセンブラ」でオブジェクトファイル(FileName.o)に変換(アセンブル)される。このオブジェクトファイルに、C 言語を実行するために必要なオブジェクトファイル(crt.o)を連結(リンク)して、実行ファイル(FileName.out, あるいは, a.out)を作成する。このことを「リンク」でリンクすると言う。

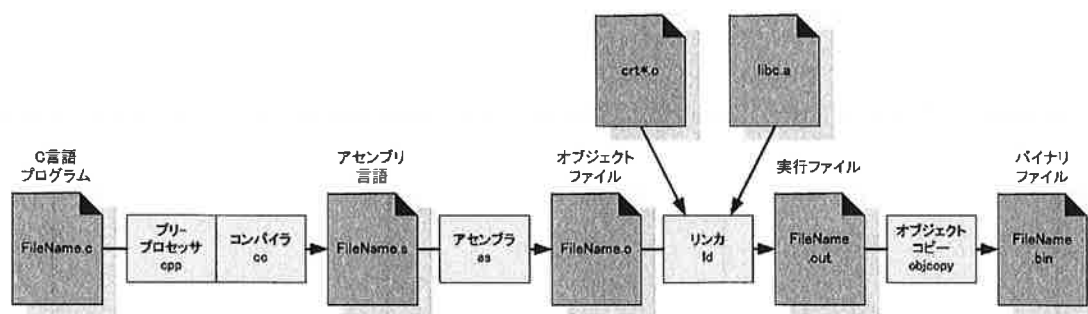


図 4: C 言語プログラムからバイナリファイルへの変換手順

この過程は、パソコンのプログラム作成と同じであるが、ここで利用するコンパイラ、アセンブラなどは、ARM 専用のものを利用する。したがって、この ARM 専用ツールで作成した実行モジュールはパソコンでは実行することができない。PC 上において、PC を動かしているプロセッサ（通常、インテル社製のペンティアム）とは別のプロセッサ（ここでは ARM）用のプログラムを開発することを、「クロス開発」と呼び、その環境（コンパイラやアセンブラなどを）を「クロス開発環境」と呼ぶ。

作成した実行ファイルには、（シンボルやセクション関連の情報などの）情報が付加されているため、このままでは GBA で実行できない。したがって、これらの情報を取り除くオブジェクトコピーという処理を行い、GBA で実行可能なバイナリファイル（FileName. bin）を作成する。ここで作成したバイナリファイルを、GBA へ転送（ダウンロード）して GBA 上で実行する。

プリプロセッサとは、ソフトウェア開発ツールのひとつで、`#define` を定義された値に置き換えるなど、コンパイルの前に前処理を行うプログラムのことであり、通常はコンパイラに組み込まれている。図 5 にアセンブリ言語プログラムからバイナリファイルへの変換手順を示す。アセンブリ言語で作成したプログラムは、アセンブラでオブジェクトファイルに変換し、リンカで実行ファイルを作成する。アセンブラにおいてプリプロセッサ `cpp` を利用すると、アセンブリ言語プログラムのソースコードで、`#define` を利用することが可能となり、読みやすいアセンブリ言語プログラムを書くことができる。

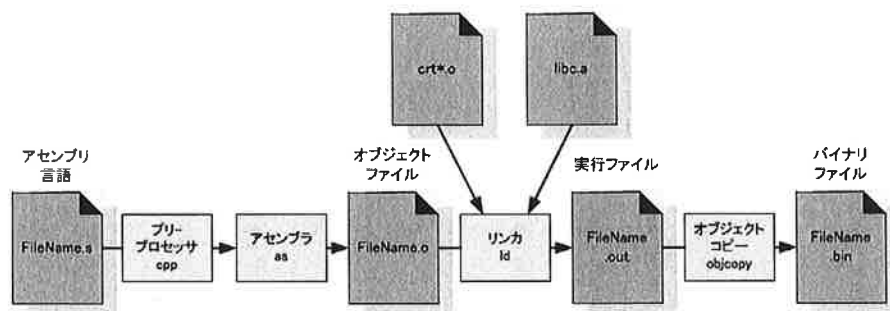


図 5: アセンブリ言語プログラムからバイナリファイルへの変換手順

2.2.4. アセンブリ言語の動作

ARM 用アセンブリ言語も、MIPS やインテル系と基本的な機能は同じで、「ロード」、「ストア」、「ムーブ」、「加算」、「減算」、「ブランチ」などのレジスタおよびメモリのデータ操作である。ただ、命令の記述方式や形式が少し異なる。注意すべき点は、ARM は 32 ビットプロセッサであるため、レジスタは 32 ビット構成であり、「1 ワード」は 32 ビットとなる。しかし、外部メモリや VRAM は 16 ビット幅（ハーフワード）であるため、図 6 に示すように、レジスタの半分のビット数のみ転送可能となる。メモリからレジスタへのロードも同様である。

アセンブリ言語詳細については、付録を参照。

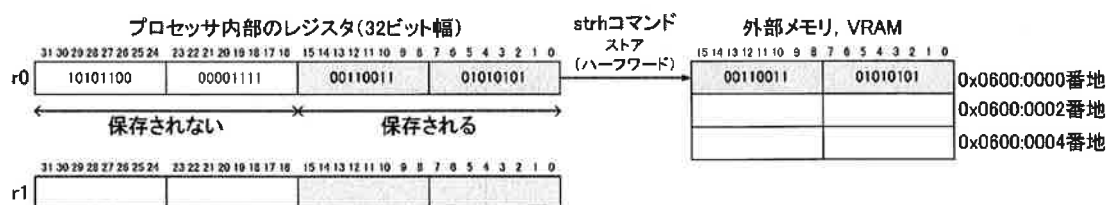


図 6: レジスタから外部メモリ、VRAM へのデータ書き込み

2.2.5. C 言語によるプログラミング

組み込みシステムを C 言語でプログラミングをする場合、通常の PC 用の C 言語プログラムと大きな違いはなく、ポインタがメモリのアドレスを表わすということを把握しておくだけである。例として、

- r3 に 0x0600:0000 (液晶ディスプレイの左上のドットの位置) が格納,
- r4 に 0x0000:7FFF (白色) が格納

されているとする。

r4 の値を、r3 が示すメモリの番地に書き込む、すなわち、液晶ディスプレイの左上のドットを白くする操作は、アセンブリ言語では、

```
strh    r4,    [r3, #0]
```

と記載する。この命令により、VRAM の 0x0600:0000 番地の 16 ビット幅の位置に、0x7FFF が書き込まれる。「#0」の意味は、r3 の値に 0 を加えたアドレスに書き込むということであるが、付録のアセンブリ言語詳細を参照のこと。ここでは、値が 0 であるため、無視してよい。これを C 言語で記載すると、

```
*((unsigned short *) 0x06000000) = 0x7FFF;    // 直接的表現
```

となる。ここで、unsigned short は、符号無しの 16 ビットの定義であり、0x06000000 は型がないので、ここではキャストしている。0x06000000 番地のアドレス、すなわち、ポインタに、0x7FFF のデータを代入するという意味である。ポインタは、メモリのアドレスを示しているということが重要である。この記述のままでも実行可能であるが、プログラムとして見やすく、使いやすく表現すると、次のようになる。

```
typedef volatile unsigned short  hword; // 16ビット変数を hword と定義
#define VRAM 0x06000000    // VRAM の先頭番地を定義
#define BGR(r, g, b) ((b << 10) + (g << 5) + r) // 色の値を作成
/* 中略 */
hword  *ptr;                // ポインタ変数を定義
ptr = (hword*) VRAM;        // VRAM の値を ptr 変数に代入
*(ptr) = BGR(0x1F, 0x1F, 0x1F); // 白色(0x7FFF)データを VRAM に書き込み
```

「<<」演算子は、左へのシフトを意味している。このプログラムは、文字をわかりやすく置き換えているだけで、上記のアセンブリ言語の表現、および、C 言語の直接的表現と同じ内容を実行していることの確認が必要である。

判ってしまえば、何ということはないが、C 言語に慣れていない場合、この部分が最も難しい、と言うか、ややこしい。ここさえ理解できれば、後は、C 言語の文法にしたがってプログラムを作るだけである。

3. 実験1-1

3.1. 液晶ディスプレイに4ドット表示

液晶ディスプレイの画面の左上から、白い点を4ドット表示させるプログラムを、アセンブリ言語で作成し、バイナリファイルに変換し、GBAにダウンロードする。(ドットは非常に小さいので注意)

3.1.1. アセンブリ言語プログラムのソース作成

以下のようなアセンブリ言語のソースコードを作成する。説明のために、プログラムリストの各行に行番号を付けているが、プログラムを書く場合は、必要がない

ファイル名は何でもよいが、今、ここでは、「dots2.S」としておく。一般に、アセンブリ言語のプログラムは、(FileName).s と小文字の s で表現される場合と、(FileName).S の大文字の S で表現される場合がある。両者には違いがあるが、ここでは特に意識しなくてよい。

```
1.  .arm
2.  .text
3.
4.      mov    r1,    #0x04000000
5.      ldr     r2,    =0x0F03
6.      strh    r2,    [ r1 ]
7.
8.      mov    r3,    #0x06000000
9.      ldr     r4,    =0x7FFF
10.     strh    r4,    [ r3 ]
11.     strh    r4,    [ r3, #4 ]
12.     strh    r4,    [ r3, #8 ]
13.     strh    r4,    [ r3, #12 ]
14. hogehoge2:
15.     b hogehoge2
16.
```

図 7: (リスト1) ドット表示プログラム

1行目の「.arm」は、このプログラムがARM用であることをアセンブラに伝えるためのものである。4～6行目は、画面を初期化して描画モードをモード3にセットするための命令で、0x0400:0000番地に0x0F03のデータを書き込んでいるが、内容については意識しなくてよい。8～15行目が、液晶ディスプレイにドットを各ためのプログラムである。15行目の「b hogehoge2」命令の後ろに改行(Enter)を入力するのを忘れないこと。(hogehoge2はラベルと呼ばれ、特定のアドレスを指定する目的で利用される。ラベルの名称は何でもよい)。「b」は無条件ブランチ(ジャンプ)命令であり、このプログラムは15行目を実行すると14行目に無条件でジャンプするため、無限ループに入り、これ以上何も実行しなくなる。

このリスト1のdots2.sソースには、「#define」や「コメント」が含まれていないため、プリプロセッサを通す必要がない。

3.1.2. アセンブラによるオブジェクトファイルの作成

リスト1のプログラムをアセンブルして、オブジェクトファイル「dots2.o」を作成するために、以下のコマンドを実行する。

```
as-arm -o dots2.o dots2.S
```

as-arm は、ARM用のクロスアセンブラを実行するためのコマンドである。「-o」は、出力ファイル名の指定を

示している. このコマンドにより, オブジェクトファイル「dots2.o」が作成される. 最後の dots2.s のサフィックス「.S」は, アセンブリ言語によるプログラム作成時は大文字「.S」とし, C 言語によるプログラム作成時は「.s」とする.

3.1.3. リンカによる実行ファイルの作成

オブジェクトファイル「dots2.o」を実行ファイルとするため, リンクを行うが, プログラムが 0x0200:0000 番地から配置されるように, 以下のコマンドで作成する.

```
ld-arm -Ttext 0x02000000 -o dots2.out dots2.o
```

ld-arm は, ARM 用のリンカである. 「-Ttext」は, プログラムのテキストが配置される位置を設定するためのオプションである. 作成したプログラムは, 外部メモリに配置される. 外部メモリの開始アドレスが 0x0200:0000 であり, この外部メモリの先頭から配置されるために, リンク時にテキストファイルの先頭を指定する. 「-o」は, 出力ファイル名の指定を示している. このコマンドにより, 実行ファイル「dots2.out」が作成される.

3.1.4. オブジェクトコピーによりバイナリファイルの作成

実行可能なバイナリファイルを作成するため, objcopy-arm コマンドにより, オブジェクトコピーを行う. 「-O binary」(O は大文字)は, 作成されるファイルがバイナリ形式で出力するためのオプションである.

```
objcopy-arm -O binary dots2.out dots2.bin
```

このコマンドにより, 実行ファイル「dots2.bin」が作成される.

3.1.5. GBA のセットアップ

1. GBA に AC アダプタを接続する. (電源は OFF のまま)
2. PC と GBA を USB ケーブルで接続する.
3. GBA の電源を入れる. (電源を入れたまま, USB ケーブルを接続したり, 抜いたりしないこと)
4. 電源を入れると, 「GAMEBOY NINTENDO」のロゴが表示され, クリスタルサウンドとともに 2 回点滅した後に, 液晶の画面全体が白色となる. (GBA のボリュームが最小になっているとサウンドが聞こえない)

3.1.6. プログラム (バイナリファイル) のダウンロード, 実行

バイナリファイル「dots2.bin」を GBA にダウンロードし, 実行する.

```
dl-gba dots2.bin
```

これにより, バイナリファイル「dots2.bin」がダウンロードされ, 直ちに, GBA で実行される.

3.1.7. プログラム実行結果の確認

液晶ディスプレイの画面が黒くなり, 左上に白いドットが4つ表示されていることを確認する. (ドットは非常に小さいため, よく見ないとわからないので注意)

プログラムを再ダウンロードする場合は, GBA の電源を切り, 再び, 入れ直す.

3.1.8. バイナリファイルの内容確認

バイナリファイル「dots2.bin」の内容を確認する. ファイルは, テキスト形式で書かれているので, 通常のエディタでは表示できない. オブジェクトダンプコマンド「od」を利用し, ファイル内容を表示する.

```
od -t x4 dots2.bin
```

オプション「-t x4」は, ファイル内容を, 16 進数で 4 バイトずつ表示するためのものである. 出力結果の左側の

列は、ファイルの先頭からのアドレス(番地)を 16 進数で示している。全部で 48 バイト(12 ワード)の ARM の命令が機械語表示される。先頭の命令は「e3a01301」となっているはずである。

3.1.9. 逆アセンブルによる内容確認

命令の 16 進数表現「e3a01301」では、わかりにくいので、逆アセンブルを行い、どのような命令の内容になっているかを確認する。

```
das-arm dots2.bin > dots2.txt
```

このコマンドを実行すると、「dots2.bin」の内容を、バイナリファイルをアセンブリ言語として解釈し、「dots2.txt」ファイルに保存する。この「dots2.txt」のファイルは、「dots2.s」と同じディレクトリ(フォルダ)内に作成される。このコマンドを実行した後に、エディタで「dots2.txt」を開き、内容を確認する。この逆アセンブルの結果のファイルは、レポート作成時に必要となるので保存しておくこと。(逆アセンブルを実施するのはこの課題のみ)

3.2. ドットの色変更

リスト1では、液晶ディスプレイに白いドットが表示されたが、ドットの色を「水色」とするプログラム(たとえば dots3.S)を作成し、実行結果を確認する。プログラム作成の際、先の課題で作成したファイル名と異なる名前とすることがある。リスト 1 のプログラム中の白色のデータを、水色のデータに変更すると、ドットが水色となって表示される。

3.3. ドットの表示位置変更

次に、表示位置を右下とするプログラム(たとえば dots4.S)を作成し、実行結果を確認する。リスト1のプログラムでは、ドット書込み位置が、左上の 0x0600:0000 と設定しているが、画面サイズを考え、右下の画面の位置のメモリ番地を計算し、白い点 4 ドットを表示させる。

3.4. リスト 1 のプログラムの C 言語化

アセンブリ言語で記載されたリスト1のプログラムの内容すべてを C 言語で書き直す。ファイル名は何でもよいが、ここでは、「dots5.c」とする。特に、リスト1の 4~6 行目(描画モードをモード 3 にセットするための命令)において、ポインタの使い方に注意を払う。

3.4.1. C 言語プログラムのコンパイル

C 言語で作成したプログラム「dots5.c」をコンパイルし、アセンブリ言語とする。

```
cc-arm -S dots5.c
```

C 言語プログラム「dots5.c」をコンパイルすることで、アセンブリ言語プログラム「dots5.s」が作成される。「-S」オプションは、アセンブリ言語ファイル作成のためのオプションである。このオプションをつけずに、コンパイルすると、直接、実行ファイルが作成される。しかし、このままでは、プログラムのメモリ配置などを考慮していないため、この実行ファイルは、そのままでは GBA で実行することができない。dots5.c をコンパイルし dots5.s を作成した後は、これまでと同様にアセンブル、リンクを行い dots5.bin を作成し、GBA にダウンロードして動作させ、その動作が仕様に合っているかを確認する。

C 言語のプログラムを記述する際は、★必ずインデント(字下げ)をすること★。インデントされていないソース

コードに対しての質問は受け付けない。

4. 実験1-2

4.1. キー（ボタン）入力

実験1で作成したアセンブリ言語のプログラムを利用して、キー（ボタン）を押したら、液晶ディスプレイの画面のドットの色が変わるようなプログラムを作成する。

4.1.1. 仕様決定

プログラムの仕様を決定する。「キー（ボタン）を押したら色が変わる」プログラムであれば、どのような仕様でもよい。たとえば、簡単な仕様は

- 初期状態で、白いドットが表示される
- 「START」ボタンを押すと、ドットの色が赤に変わる
- そのキー（ボタン）を離すと、ドットの色が白にもどる

といったものである。「L」ボタンを押すと赤、「R」ボタンを押すと緑色に変わり、ボタンを離しても色は変化しない、などといった仕様も考えられる。

4.1.2. プログラム作成（アセンブリ言語）

決定した仕様に合ったプログラムをアセンブリ言語で作成する。GBA にダウンロードして動作させ、その動作が仕様に合っているかを確認する。

4.1.3. キー（ボタン）判定

たとえば、「A」ボタンが押されたことを判別するプログラムは、図 8 のように書くことができる。

```
1.  hogehoge3:
2.      ldr        r8,        =0x04000130
3.      ldrh       r9,        [ r8 ]
4.      ldr        r10,       =0x0001
5.
6.      and        r11, r10, r9
7.      cmp        r11, r10
8.      bne        hogehoge4
9.
10.     // 「A」(ボタン)が押されていない場合の処理
11.     // 中略 (この部分が各自記載する)
12.     b           hogehoge3
13.
14.  hogehoge4:
15.     // 「A」(ボタン)が押された場合の処理
16.     // 中略 (この部分が各自記載する)
17.     b           hogehoge3
18.
```

図 8: (リスト 2) キー判定プログラム

第 3 行目の `ldr` 命令で、「A」を含めすべてのキー（ボタン）入力状態の値を読み込み、第 6 行目の `and` 命令で「A」(ボタン)の状態のみを取り出す。第 7 行目の `cmp` 命令で、「A」(ボタン)が押されているかどうかの判定を行う。第 8 行目 `bne` (ブランチ・ノット・イコール)において、「A」(ボタン)が押されていない場合は「A」(ボタン)の状態と 1 が等しければ、そのまま次の行である「A」(ボタン)を押されていない場合の処理を実行する。「A」(ボタン)

が押されていれば(「A」(ボタン)の状態と 1 が等しくなければ),「hoge4」のラベルへジャンプし,「A」(ボタン)が押された場合の処理を行う。ラベルは任意の名称でよい。

4.2. プログラム作成 (C 言語)

キー(ボタン)入力を判定するために作成した仕様に合ったプログラムを C 言語で作成する。GBA にダウンロードして動作させ,その動作が仕様に合っているかを確認する。

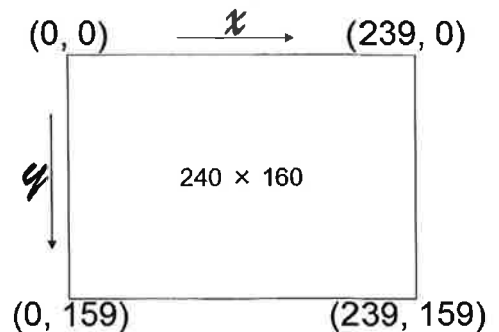
5. 実験2-1

5.1. 画面の塗りつぶし

画面すべてを水色で塗りつぶすプログラムをC言語で作成する。

5.2. 点の描画 (その 1)

右の図のような X-Y 座標系 (画面の左上が(0,0), 右下が(239,159)の 240×160 の画面)を考え, 変数 `x1 = 120; y1 = 80; color1` を赤色として, 関数 `draw_point(x1, y1, color1)` を呼び出すと, 画面の中央に赤色の点を描画するプログラムを C 言語で作成する. 背景画面は水色とする.



点の描画を行うプログラムを作成する際には, 11.3.5 節に例として記載している `draw_point` の関数(メソッド)を完成させ利用すること. 関数 `draw_point(x, y, color)` とは, GBA の画面の左上を(0, 0), 右下を(239, 159)とする座標系において, (x, y) の位置に指定した色の点を描画する関数である.

5.3. 点の描画 (その 2)

座標(X, Y)に赤色の点を描画するプログラムをC言語で作成する. 背景画面は水色とする. それぞれ点の座標の値は以下で計算する.

- $X = \{(\text{今日の日付}) + (\text{班の番号})\} \% 59 + 60$
- $Y = (\text{今日の日付}) \times (\text{班の番号}) \% 39 + 81$

(注1) 今日の日付: 6月9日の場合は"609"とする. (0609 としてはいけない)

(注2) % は割った余りを表わす.

描画された点のおよその位置を確認する.

5.4. 点の移動

前節で座標(X, Y)に描画した赤色の点を, 以下に記述するように, 十字キーに合わせて移動させるプログラムをC言語で作成する. 背景画面は水色とする. 点が移動する速さは目で確認できる程度とする.

- ↑: 上
- ↓: 下
- ←: 左
- →: 右

6. 実験2-2

6.1. 四角形の描画

座標A(X_a , Y_a), 座標B(X_b , Y_b)を対角線の頂点とする赤色で塗りつぶされた四角形を描画するプログラムをC言語で作成する. 背景画面は水色とする. それぞれの点の座標の値は以下で計算する.

- $X_a = \{(\text{今日の日付}) + (\text{班の番号})\} \% 59$
- $Y_a = (\text{今日の日付}) \times (\text{班の番号}) \% 39 + 41$
- $X_b = (\text{班に所属する学生でIDの下4桁が最も小さい人のIDの下4桁}) \% 59 + 181$
- $Y_b = (\text{班に所属する学生でIDの下4桁が最も大きい人のIDの下4桁}) \% 39 + 81$

描画した図形の見え目が指定通りの四角形になっているかを確認する. 一人の班の場合は, 最も小さい人のIDを自分のIDとし, 最も大きい人のIDは自分のID+今日の日付とする.

6.2. 円の描画

座標E(X_e , Y_e)を中心とし半径Rの赤色で塗りつぶされた円を描画するプログラムをC言語で作成する. 背景画面は水色とする. それぞれの点の座標の値は以下で計算する.

- $X_e = (\text{今日の日付}) \times (\text{班の番号}) \% 19 + 101$
- $Y_e = (\text{班に所属する学生でIDが最も小さい人のIDの下4桁}) \% 19 + 61$
- $R = (\text{班に所属する学生でIDが最も大きい人のIDの下4桁}) \% 9 + 29$

描画した図形の見え目が指定通りのほぼ円形になっているかを確認する. この課題では円を塗りつぶすが, 塗りつぶさない円の描画は高度なアルゴリズムが必要になる.

6.3. 線分の描画 (その1)

座標A(X_a , Y_a)から座標B(X_b , Y_b)へ赤色の線分を描画するプログラムをC言語で作成する. 背景画面は水色とする. (座標A, 座標Bは, 四角形の描画の際と同じである)

描画した図形の見え目がほぼまっすぐな線分になっているかを確認する. 線分描画アルゴリズムの一例を後述するが, どのようなアルゴリズムを利用してもよい.

6.4. 線分の描画 (その2)

座標C(X_c , Y_c)から座標D(X_d , Y_d)へ赤色の線分を描画するプログラムをC言語で作成する. 背景は水色とする. それぞれの点の座標の値は以下で計算する.

- $X_c = (\text{班に所属する学生でIDが最も大きい人のIDの下4桁}) \% 59 + 61$
- $Y_c = (\text{今日の日付}) \times (\text{班の番号}) \% 39 + 121$
- $X_d = (\text{班に所属する学生でIDが最も小さい人のIDの下4桁}) \% 59 + 121$
- $Y_d = \{(\text{今日の日付}) + (\text{班の番号})\} \% 39$

描画した図形の見え目がほぼまっすぐな線分になっているかを確認する.

7. 実験3-3

プログラムを新規に設計、制作し、動作を確認する。作成には C 言語を使用し、どのような構成や内容でもよい。グループ内のメンバーが、次の4つの役割の主担当を担う。各主担当が1人でその役割を実施するのではなく、全員で話し合いながら、作業を進める。

- 仕様主担当(リーダー)：プログラムの仕様を検討、仕様書作成
- コード主担当：プログラムのコーディング
- テスト主担当：プログラムのテスト手順検討および実施

リーダーは各主担当が実施する内容をすべて把握しておく必要がある。グループが3名構成の場合は、それぞれの役割をひとつずつ担当する。グループが2人構成の場合、1人が仕様主担当、もう1人がテスト担当を担い、プログラムの作成は2人で分担する。また、仕様主担当がリーダーを兼ねる。

7.1. 自由プログラム作成

7.1.1. プログラム仕様決定 (全員)

どのようなプログラムとするか、リーダーが中心となって、全員で話し合い、仕様の概要を決める。たとえば、「シューティングゲームで、キーを押すとカーソルが移動し、A ボタンを押すと敵を倒す」などといった程度の内容を決める。プログラムの仕様は自由に決定してよいが、「お絵かき」や「ペイント」といった矢印キーでカーソルを上下左右に移動させて画面に絵を書くようなプログラム以外の仕様とすること。

仕様の概要は、自由プログラム作成課題の実施日までに、グループで相談しておき、実施日の実験開始前に発表を行う。

7.1.2. 仕様書作成 (仕様主担当)

話し合った仕様をもとに、仕様書を作成する。具体的には、初期状態、「キー」の割当て、「画面」の色の変化、操作手順などを決めていく。たとえば、

- 初期状態で全画面が黒く表示される
- 「START」を押すと敵が表示され、上下左右にランダムに移動する
- 矢印キーでカーソルが上下左右に移動する
- 敵の位置にカーソルを重ねて、「A」ボタンを押すと敵を倒す
- 敵を三体倒すと、次のステージへ移行し、敵の動きが早くなる
- これに合わせてカーソルも早く移動できるようになる
- 隠しコマンドで、「B」ボタンを押すと敵の動きが遅くなり、倒しやすくなる
- 第2ステージをクリアすると、期状態にもどり、全画面が黒い初期状態となる
- 例外：「A」、「B」、「START」以外のボタンを押しても、何も変化しない

などのように、プログラムの「使い方マニュアル」を記述する。

7.1.3. プログラム設計 (全員)

仕様に従って、プログラムを設計する。ここで行うのはソースコードの記述ではなく、関数名、(グローバル、ローカルなどの)変数名、複数の関数がある場合はそれらの関係を決める。

サンプルプログラムは、「sample」フォルダに置いてあるので、プログラム作成時の参考としてよい。

7.1.4. コード作成 (コード担当)

C 言語を利用して、設計したプログラムを記述する。この段階で、作成したプログラムにコメントや解説を付ける必要はない。(コメントは、レポート作成時に、各自がソースコードに追記する。)

7.1.5. テスト手順作成 (テスト担当)

作成した仕様書をもとに、テストを行う項目を整理した具体的なテスト手順を作成する。

● テスト手順1

1. 初期状態 → 全画面が黒く表示
2. 初期状態で「START」ボタンを押す → 敵(9ドット正方形の青色)が表示される
3. 初期状態で「START」以外のボタンを押す → 何も変化しない

● テスト手順2

1. 敵が上下左右移動する → どのボタンを押しても上下左右に移動する
2. 直径 5 ドットの正方形(塗りつぶしなし)カーソルが中央に表示される
3. 上下左右の矢印キーで、カーソルが移動する
4. カーソルは敵より高速で移動する
5. 敵の上にカーソルが重なると、カーソルを優先して表示

● テスト手順3

1. 敵の上にカーソルが重なった状態で「B」ボタンを押すと敵が消滅する
2. 1 体目の敵が消滅すると、2 体目の敵(黄色)が表示
3. 2 体目の敵が消滅すると、3 体目の敵(赤色)が表示
4. 3 体目の敵が消滅すると、第 1 ステージクリア

● テスト手順4

1. 第 1 ステージクリアをクリアすると、第 2 ステージへ移行
2. 第 2 ステージでは、敵およびカーソルの移動が早くなる
3. 第 3 ステージをクリアするとゲーム終了で、初期状態へ戻る
4. 第 2 ステージ移行では、隠しコマンド(「B」ボタンを押す)が利用可能
5. 「B」ボタンを押している間は敵は移動しない(カーソルは移動可能)

など、必要なテスト項目を列举する。

7.1.6. 動作確認 (全員)

テスト担当が中心になって、作成したプログラムを GBA にダウンロードし、動作させ、作成したテスト手順にそって操作を行い、決められた通りに動作するかを確認する。作成したプログラムがテスト手順通りに動かなければ、プログラムを修正し、再度、動作を確認する。

8. 発表

自由プログラムの動作確認を行った後に、各班は発表を行う。説明する項目は以下とする。発表者はこちらから指定するので、班の全員がプログラムの仕様、動作、テスト項目を理解しておく。

- 自由プログラムの仕様
- プログラムの動作
- ソースコードの簡単な解説

- テストした項目
- プログラム上の工夫した点

自由プログラムの工夫度、発表内容でレポート採点時の基本点とする。発表する班以外のメンバは質問を行う。

9. 報告(レポート)

9.1. レポート作成

作成した仕様書、プログラム、テスト手順はグループ内のメンバ全員で共有しておくこと。しかし、次の各報告事項については、グループで実施するのではなく、グループで行った仕様書・プログラム作成、および、テスト手順に沿ったテスト結果をもとに、リーダーも含め、各個人ごとに個別に検討し、ひとりひとりでレポートとしてまとめ提出せよ。参考にした文献は必ず参考文献として記載すること。レポートは、MS Word で作成し、e-class を利用して、提出期限までにアップロードせよ（詳細については後述する）。再提出が必要な場合は、e-class のステータスに表示されるので、それに従うこと。再提出など、必要な情報は e-class で提供するので、個別に確認に来る必要はない。

9.1.1. 報告事項 1

「液晶ディスプレイに4ドット表示」のアセンブリ言語プログラムのリスト1の8～15行目の内容を、それぞれの行ごとに詳しく説明せよ。

9.1.2. 報告事項 2

「液晶ディスプレイに4ドット表示」の逆アセンブルの表示結果と、リスト1の違いをリストアップし、その違いについて詳しく解説を行うこと。特に、逆アセンブル結果の最後の 2 行が、元のアセンブリ言語のプログラムと異なっている理由を考え、詳細に説明せよ。

9.1.3. 報告事項 4

「ドットの表示位置変更」において、なぜ「mov」命令が利用できなかったかについて、実例を用いて詳細に説明せよ。

9.1.4. 報告事項 5

「液晶ディスプレイに4ドット表示プログラムの C 言語化」で作成した C 言語プログラムを記述し、プログラムのすべての行について、行ごとに説明せよ。

9.1.5. 報告事項 6

「キー(ボタン)入力プログラム(C 言語)」で作成した仕様を、論理立てて記述する。作成した C 言語のプログラムのソースコード、および、その解説を記述せよ。

9.1.6. 報告事項 7

「点の移動(その 1)」、「四角形の描画」、「円の描画」、「線分の描画(その 1)」、「線分の描画(その 2)」で作成したプログラム(アルゴリズム)すべてに関して、要点と動作をそれぞれ解説せよ。特に、線分の描画に用いたアルゴリズムについて詳しく記載すると共に、線分の描画その 1 とその 2 の違いを明確にせよ。加えて、線分の描画その 1 のプログラムの流れをフローチャートで表現せよ。フローチャートを記載するのは線分の描画その 1

のみでよい(点, 四角, 線分の描画その 2 に関して, フローチャートを記載する必要はない。

9.1.7. 報告事項 8

「線分の描画」で利用したアルゴリズム以外の線分描画アルゴリズムを調べ説明せよ。また、本実験で利用した線分描画アルゴリズムと、調べた結果の線分描画アルゴリズムとを、線分の精度、描画速度等に関して、比較せよ。

9.1.8. 報告事項 9

「自由プログラム」で作成した仕様を、論理立てて記述する。作成したプログラムの流れ、各部の役割をソースコードの解説として記述せよ。ソースコードそのものはレポートの付録とすること。その付録のソースコードにおいて、関数、定義した各変数、if 文、for 文、while 文、case 文、switch 文、その他必要と思われるところには、コメントを付けて提出すること。また、動作結果が、テスト手順と合っているか項目についての動作を記述せよ。

9.1.9. 報告事項 10

この実験を行った際の問題点と、その解決方法、および、考察を、ソースコードと対応させてまとめよ。

9.1.10. 報告事項 11

「自由プログラム」で作成したプログラムの仕様書、(コメントが入っていない)ソースコードのファイル、テスト手順を記述したファイル、バイナリファイルの4つのファイルを、すべて zip 形式としてまとめて、グループのリーダーが e-class の「仕様書等の提出へ」アップロードする。リーダーも自分のレポートは、「各グループ」のページにアップロードせよ。

9.1.11. 報告事項 12

この実験におけるすべての課題を行った際の問題点と、その解決方法、および、考察を、ソースコードと対応させてまとめよ。

10. 実験実施計画

情報システム演習実験は3週で実施する予定である。以下に実験実施計画の目安を示す。

- ・ 第1週：（既定課題）実験1-1, および, 実験1-2
実験1-1, および, 実験1-2を終了した時点で, 第1週を終了してもよいし, 第2週を前もって実施してもよい。
- ・ 第2週：（既定課題）実験2-1, および, 実験2-2
この週の実験終了までに実験2-2の最後までの実施を完了することが望まれる。第3週は自由課題を実施するため, 既定課題は第2週で終わりとする。
- ・ 第3週：（自由課題）実験3
自由プログラミングを実施する。どのようなプログラミングを作成するかは, 第3週の実験を開始するまでに, 必ずグループ内で事前に相談のこと。
第3週の実験は, 3つの講時(3 講時 13:10~14:40, 4 講時 14:55~16:25, 5 講時 16:40~18:10)からなり, 3 講時と 4 講時で自由課題のプログラミングを完成させ, 5 講時で発表を行う。発表では, プログラムの動作の紹介と, 作成したプログラム(コード)の概要の説明を求める。

自宅等の Windows パソコンで利用可能な本実験のプログラム開発環境を提供する。この開発環境を利用してプログラムを作成し, パソコン上で動作するエミュレーション環境(VBA: Visual Boy Advance)で作成したプログラムを実行し確認することが可能である。授業で利用するのは実機のハードウェア(Game Boy Advance)であるが, エミュレーション環境の VBA はパソコン上で動作するため, 実機よりもかなり(100 倍以上)高速な動きとなるので注意が必要である。

11. 採点基準

情報システム演習実験の採点は合計 100 点満点, 1テーマ 25 点満点とする。本テーマは, 実験1, 2の規定課題が 10 点, 実験3の自由課題が 5 点, レポートが 10 点の合計 25 点とする。


12. 付録

12.1. 実験開始時の PC 操作手順

12.1.1. Windows の立ち上げ

ノート PC の電源をオンにして、Windows を立ち上げる。

12.1.2. インターネットの接続

画面右下の無線 LAN のアイコン  をクリックし、DO-NET1x の「接続」をクリックし、無線 LAN に接続する。次に、ブラウザを立ち上げて「Agree」をクリックして、認証を行い、インターネットに接続する。インターネット接続は、次の Oracle VM Virtual Box 立ち上げの前に行っておく必要がある。

12.1.3. ソフトウェア開発用 Linux (Ubuntu) の立ち上げ

画面左下の Windows のスタートボタンから「Oracle VM Virtual Box」を起動する。

Oracle VM Virtual Box マネージャのメニューの「⇒起動」をクリックする。

しばらくすると、Ubuntu が立ち上がる。画面上部に2つのメッセージバーが表示されるが、⊗をクリックして、消去する。右上の□をクリックし全画面表示しておく。

12.1.4. データ保存 USB メモリの挿入

データ保存 USB メモリを PC の USB ポートに挿入する。Virtural Box のメニューの「デバイス」の「USB」の「Sony Storage Media」(USB メモリの種類によっては別の表示の可能性もある)を選択し、USB メモリをマウントする。しばらくすると、自動的にデータ保存 USB メモリのフォルダが開き、左メニューバー上に USB メモリの形をしたアイコンが表示される。実験開始の初日の時点で、USB メモリにフォルダやファイルが保存されている場合は、教員あるいは TA に申し出て、消去すること。(System Volume Information というアイコンの場合は消去しなくてもよい)

ファイルやフォルダを消去するには、それらを左下のゴミ箱に入れ、右クリックで「ゴミ箱を空にする」をクリックすれば消去できる。

12.1.5. ISD-LAB フォルダ

USB のフォルダが全画面表示されていれば、ウインドウ左上の⊗⊖ⓂのⓂをクリックすると、デスクトップが表示されウインドウ表示となる。デスクトップ上の「ISD-LAB へのリンク」をクリックすると、ウインドウがオープンする。「sample」、「work」の2つのフォルダ(ディレクトリ)がある。「sample」には、GBA 用のサンプルプログラムがあるので、参考にしてよい。「work」は作業フォルダであるが、このフォルダはデータ保存 USB メモリにコピーして使用する。

12.1.6. フォルダをコピー (重要)

さきほどの「ISD-LAB」フォルダ内の「work」フォルダを、今オープンしたデータ保存 USB メモリに左クリックでドラッグしながらコピーする。「work」フォルダの中にいくつかのファイルがあるが、これらは C 言語のプログラムをコンパイルする際に必要となるので、消去してはいけない。

12.1.7. ファイル作成

「work」のウインドウ内で、右クリックをし、「新しいドキュメント」→「空のドキュメント」を選択すると、「無題のド

キュメント」テキストファイルが作成される。そこで、ファイル名、たとえば、「dots2.S」と入力する。ウインドウに自動的にファイル「dots2.S」が作成される。この「dots2.S」をクリックすると、自動的にテキストエディタ「gedit」が立ち上がり、「dots2.S」の内容を記述できる。ここでプログラムを作成する。全画面表示されていれば、ウインドウ左上の⊗⊖⊙の⊙をクリックすると、デスクトップが表示されウインドウ表示となる。**ファイルを作成する場合、必ず USB 上の「work」のフォルダ内に作成する。** USB 外にファイルを作成した場合、PC を再起動するとファイルが削除される可能性があるので注意が必要である。

課題ごとのソースコードのファイルはレポート作成時に必要となるので、すべて保存しておく。

12.1.8. コマンド入力

「work」のウインドウ内で、右クリックをし、「端末の中に開く」を選択すると、「シェル」(コマンドプロンプト)のウインドウがオープンする。「user@Ubuntu: [ディレクトリ名/work]\$」というプロンプトが表示されるので、そこで、コマンドを入力する。「work」のウインドウ以外で右クリックすると、作成したファイルが見えず、コンパイルできない場合がある。プロンプトの表示の最後が[ディレクトリ名/work]になっていることを確認する。

12.2. GBA の USB ケーブルの PC 接続手順

12.2.1. GBA のセットアップ

1. GBA に AC アダプタを接続する。(電源は OFF のまま)
2. GBA に USB ケーブルを接続し、電源を入れる。電源を入れると、「GAMEBOY NINTENDO」のロゴが表示され、クリスタルサウンドとともに2回点滅した後に、液晶の画面全体が白色となる。(GBA のボリュームが最小になっているとサウンドが聞こえない)
3. GBA の USB ケーブルを PC に接続する。
4. メニューの「デバイス」の「USB」の「OPTIMIZE PRODUCT GBA BOOT CABLE USB」を選択する。
この操作を行わないと、Ubuntu から GBA が認識できず、作成したプログラムをダウンロードできない。

12.3. 実験終了時の PC 操作手順

12.3.1. インターネットの利用

画面左のメニューバーの「Firefox」のアイコンをクリックすると、Web ブラウザが起動する。ブラウザメニューの「同志社大学 SSO」をクリックし、UserID と Password を入力し、「Log in」する。ログイン情報は保存しても保存しなくてもよい。

「Weg Single Sign-On」画面の Office 365 アイコンをクリックし、Outlook を立ち上げる。



12.3.2. ファイルバックアップと送付

その日の作業が完了すると、エディタのウインドウをすべてクローズした後に、work フォルダの内容を圧縮して、班全員にメールで送付する。work フォルダを右クリックし、「圧縮」を選択し、その中の「.tar.gz」を選択し、さらにその中の「.zip」(一番下)を選択し、「作成」を押す。そうすると、同じフォルダ上に work.zip というフォルダが作成される。work.zip のファイルを添付したメールを班のメンバ全員のメールアドレスに送付する。

データ保存 USB メモリを抜く場合は、**必ず、マウント解除する。** USB メモリのアイコンを「右クリック」し「取り出

し」を選択すると、マウントが解除される。**実験最終日には、データ保存 USB メモリ内のデータをすべて消去**しておく。

持参した USB メモリを利用してデータを保存する場合は、USB を PC に挿入し、Oracle VM Virtual Box マネージャのメニューの「デバイス」から挿入した USB メモリをマウントする必要がある。また、取り外す場合は、マウントを解除する。

12.3.3. Linux (Ubuntu) シャットダウン

画面右上の電源ボタンをクリックし、「シャットダウン...」を選択する。確認のウインドウが表示されるので、「シャットダウン」をクリックする。

12.3.4. Windows シャットダウン

「Oracle VM Virtual Box マネージャ」のメニューバーの「ファイル」から「終了」を選択する。最後に Windows をシャットダウンする。

12.3.5. 文字コード (Linux と Windows の違い)

一般に Linux の日本語漢字コードは UTF-8 であり、Windows はシフト JIS (SJIS) であるため、互換性がない。Linux 上で作成した日本語ファイルを Windows で読む場合、秀丸エディタを利用するか、MS Word を利用してファイルをオープンする必要がある。

12.3.6. その他

プログラム作成中に、用意されている必要なファイルや、システム関連ファイルを消去しないように注意が必要である。ファイルを消去したために、コマンドが動作しない、システムが異常となるなどの状態に陥った場合は、Ubuntu を再起動する。

二進数、十六進数を計算できる電卓は、左上の「アプリケーション」→「アクセサリ」の「電卓」を利用することができる。電卓の「表示」の「プログラミング」により基数を切り替えることが可能である。

12.4. ARM 命令セット

12.4.1. 参考ドキュメント

ARM 命令セット(アセンブリ言語記述方法)は、「ARM 命令セット概要」、および、必要であれば「ARM 命令セット詳細」を参考にする。

12.4.2. 利用命令

ARMのどのような命令を利用しても問題ないが、おそらく、「AND」(アンド)、「ADD」(加算)、「SUB」(減算)、「B」(無条件ジャンプ)、「CMP」(比較)、「BEQ」(条件ジャンプ)、「BNE」(条件ジャンプ)、「MOV」(移動)、「LDR」(ロード)、「LDRH」(ロードハーフワード)、「STRH」(ストアハーフワード)の命令の利用で十分だと思われる。

12.4.3. 命令サンプル

- レジスタ r1 の値を、レジスタ r2 にコピーする場合

```
mov r2, r1
```

- 8ビットのデータ 0x12 を、レジスタ r1 に代入する場合

```
mov r1, #0x12
```

(mov を利用できるのは、8 ビット幅のデータのみ。0x12000000 は mov 命令利用可能だが、0x10200000 は不可。8 ビット幅より大きいデータの場合は ldr を利用)

- 32ビットのデータ 0x12345678 を、レジスタ r1 に代入する場合

```
ldr r1, =0x12345678
```

- レジスタ r1 の値に、1 を加えて、計算結果を r2 に入れる場合

```
add r2, r1, #1
```

- レジスタ r1 の値に、レジスタ r2 の値を加えて、計算結果を r3 に入れる場合

```
add r3, r1, r2
```

- レジスタ r1 の値と、レジスタ r2 の値の論理積 (and) の計算結果を r3 に入れる場合

```
and r3, r1, r2
```

- 16ビット幅のデータをメモリからリードする際、レジスタ r1 にアドレス、レジスタ r2 にリードしたデータを入れるとする場合

```
ldrh r2, [r1]
```

- 16ビット幅のデータを、メモリにライトする際、レジスタ r2 にライトするデータが入っており、レジスタ r1 にアドレスが入っているとする場合

```
strh r2, [r1]
```

- 16ビット幅のデータを、メモリにライトする際、レジスタ r2 にライトするデータが入っており、レジスタ r1 にアドレスが入っているとする。レジスタ r1 の値に 4 を加えた値のアドレスにデータをライトする場合

```
strh r2, [r1, #4]
```

- レジスタ r1 の値から 1 を引き、その結果をレジスタ r1 に再び入れる。その際、r1 が 0 かどうかを判別し、0 ならばラベル hogehoge1 へジャンプし、0 でなければ次の命令を実行する場合

```
subs r1, r1, #1
```

```
bne hogehoge1
```

(ARM の命令では、特に **CMP** などの比較命令を利用しなくても、減算命令のみで、比較結果を利用することができる)

- レジスタ **r1** の値と、レジスタ **r2** の値を比較し、等しければラベル **hogehoge2** ヘジャンプし、等しければ次の命令を実行する場合

```
cmp  r1,  r2
beq  hogehoge2
```

- レジスタ **r1** の値と、レジスタ **r2** の値を比較し、等しければラベル **hogehoge3** ヘジャンプし、等しければ次の命令を実行する場合

```
cmp  r1,  r2
bne  hogehoge3
```

- 命令を実行した際に、無条件にラベル **hogehoge4** ヘジャンプする場合

```
b  hogehoge4
```

12.5. C 言語についての簡単な説明

12.5.1. C 言語プログラムの構造

C 言語プログラムにおいて、実行文は関数（Java で言うところのメソッド）と呼ばれる単位の中に記述する必要がある。

C 言語プログラムの場合、通常、一番初めに実行する関数を main として定義する。しかし、この実験においては C 言語実行のための前処理を行わないため、プログラムは記述した順番に前（上）から実行される。main よりも前（上）に関数を記述していればその関数が実行されるので、複数の関数を定義する場合、main を最も前（上）に書いておくことが望ましい。

typedef や #define は main よりも前（上）に定義しておく。typedef の文の最後にはセミコロンが必要であるが、#define の文の最後にはセミコロンが不要である。

プログラムを記述する際は、**必ずインデント（字下げ）をすること**。インデントしなくてもプログラムは動作するが、プログラムの記述において自分自身がミスを起こす場合が多いため、必ずインデントしてプログラムを記述する。

if-else 文、for 文、while 文、switch 文などは、Java と同様に利用可能である。これらの文を利用する場合もインデントに注意すること。Java で事前定義されている true、false は、C 言語では定義されていないので、C 言語のプログラムとして利用するのであれば、#define TRUE 1 のように定義しておく必要がある。一般に、#define で特定の値を定義する場合はすべて大文字を使うのが慣例である。

```
/* typedef はここで (volatile unsigned short を hword と) 定義 */
typedef volatile unsigned short hword;

/* define はここで定義 (VRAM という文字列が 0x06000000 で置き換えられる) */
#define VRAM 0x06000000

/* main 関数で始める */
int main (void) {

    /* インデント（字下げ）する */
    /* 利用する変数を定義する */
    hword *ptr;      // ptr をポインタとして定義 「*」はポインタ の意味
    hword color;

    /* プログラムの実行文を順番に記述する */
    ptr = (hword*) 0x04000000 ;
    *ptr = 0x0F03;    // 0x04000000 番地にデータ 0xF03 を書き込み

    color = 0x7FFF ; // color 変数に 0x7FFF (白色データ) を入れる
    ptr = (hword*) VRAM;

    /* 画面に点を描画 */
    *ptr = color;     // 0x06000000 番地にデータ 0x7FFF を書き込み

    /* 通常なら return 0 で終了するが、本実験のプログラムにおいては
       プログラムが終了後に暴走しないように無限ループを挿入しておく */
    while (1);
    return 0;

} // main 関数の終わりのカッコ
```

12.5.2. Java との違い

Java はアプリケーションプログラム作成に向けた言語であり、ハードウェアを直接制御するプログラムを記述できない。一方、C 言語は、ハードウェアの制御や、コンピュータシステムの記述に適した言語で、クラスの定義がなく、プログラミング言語としての仕様は Java よりも単純である。

型 (一部)

型	Java	C 言語
int	符号付き 32 ビット整数 -2147483648~-2147483647	CPU に依存。32 ビット CPU の場合、同左となるが、16 ビット CPU の場合、符号付き 16 ビット整数となる
unsigned int	定義なし	CPU に依存。32 ビット CPU の場合、符号なし 32 ビット整数となる 0~0xFFFFFFFF
short	符号付き 16 ビット整数	同左
unsigned short	定義なし	符号なし 16 ビット整数 0~0xFFFF
char	符号なし 16 ビット Unicode を表現	符号なし 8 ビット
typedef	定義なし	型宣言の名前の置き換え typedef short u16; と定義すると、 u16 mode; は short mode; と同じ意味となる

定数 (一部)

名称	Java	C 言語
true	boolean の「真」	0 以外で、通常は、“1”を利用
false	boolean の「偽」	“0”を利用する

定数の定義

Java	C 言語
public static final int KEY = 1234;	#define KEY 1234 int i = KEY;
	#define はプログラム中の KEY という文字列を 1234 に置き換えるだけ。

プログラムの開始

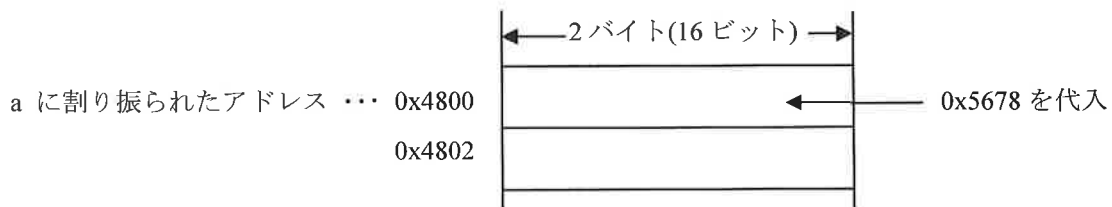
Java	C 言語
public static void main (String[] args) {	int main (int argc, char* argv[]) {
main クラスを定義	main の宣言 int は、main 関数の戻り値の定義であり、今回は無視してよい () 内の引数は無くても問題なし

12.5.3. 「ポインタ」について

C 言語を利用する上で、Java と異なる大きな点にひとつがポインタである。簡単に言うと、ポインタとは、メモリのアドレス（番地）だと思ってよい。

```
unsigned short a;    // unsigned short は符号なし 16 ビットとする（注）  
a = 0x5678;
```

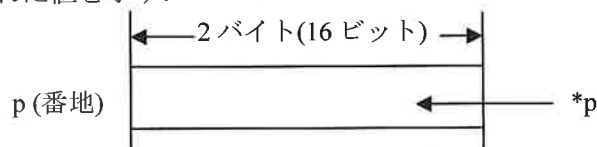
上記のように記述すると、下図のように、「メモリのデータエリアのある番地（0x4800 番地として説明するが、実際はどのような値か不明）に、変数 a としての、領域を確保し、その領域に 0x5678 を格納する」ということである。



このとき、unsigned short a の a は、変数 a の値である 0x5678 を示す。

```
unsigned short *p;    // unsigned short は符号なし 16 ビットとする
```

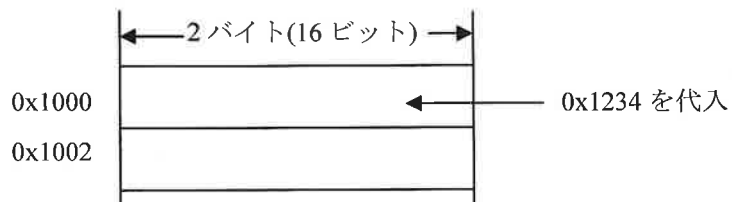
上記のように「*」をつけて定義すると、p をポインタとして宣言することになる。このとき、p は番地を示し、*p はその番地に格納された値を示す。



この例の場合、

```
unsigned short *p;    // unsigned short は符号なし 16 ビットとする  
  
p = 0x1000;           // p のアドレスを設定  
*p = 0x1234;          // アドレス 0x1000 番地に、0x1234 のデータを格納
```

と記述することができる。すなわち、*p のポインタを宣言した場合、0x1000 番地に 0x1234 のデータを書込むことを意味する。



しかし、ここでは、0x1000 は通常でポインタ型として定義されておらず、C 言語のルールにより、型が異なる値を代入する場合、型変換（キャスト）を行う必要がある。（Java でもキャストを利用して、型変換ができる）

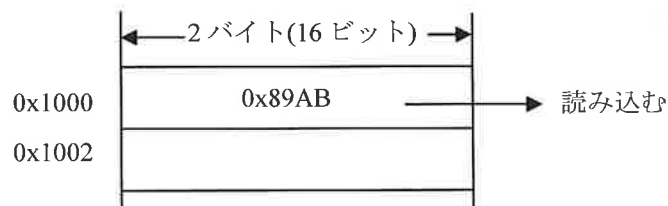
```
unsigned short *p;    // unsigned short は符号なし 16 ビットとする  
  
p = (short *) 0x1000; // p のアドレスを設定  
*p = 0x1234;         // アドレス 0x1000 に、0x1234 のデータを格納
```

変数として定義せずに、0x1000 番地に、0x1234 のデータを格納するだけであれば、

```
*(unsigned short *) 0x1000 = 0x1234; //0x1000 番地に 0x1234 格納
```

と記述できる。

また、以下のように、0x2000 番地に 0x89AB が格納されており、その値を読み込んで、変数 b に代入する場合は、



```
b = *(unsigned short *) 0x2000 ; // 0x2000 番地に変数 b 格納
```

と記述することができる。この命令により、b に 0x89AB が代入される。あるいは、p をポインタとして定義し、

```
unsigned short *p; // unsigned short は符号なし 16 ビットとする  
  
p = 0x2000; // p のアドレスを設定  
b = *p;     // アドレス p 番地のデータを変数 b に格納
```

と記述できる。

(注) 一般的に unsigned short を 16 ビットとしたが、プロセッサによっては異なる場合もある。

12.5.4. 最低限のコーディングルール（プログラム記述時の取り決め）

1. while, for, if 文などの内容は、タブでインデントして（文の書き出しを下げて）書く。
2. while, for, if 文などの内容が 1 文であっても、かならず、{ } でくくる。
3. プログラムの 1 行に複数の変数、あるいは、複数の処理を「,」で区切って記述しない。

(注) このルールでプログラムが書かれていない場合、TA はプログラムを確認しない。

12.5.5. 関数について

Java 言語のメソッドは、C 言語では関数（ファンクション）と呼ばれる。main() のみのプログラム

であれば問題ないが、main 関数以外の関数を作成する場合は注意が必要である。通常の C 言語のプログラムは、main 関数がどこにあっても、main から実行されるように、コンパイル時の処理が行われる。(C RunTime Startup と呼ばれる crt.o コードが自動的に追加される) 本実験では、開発環境によりプログラムが自動的に操作(追加, 削除)されるのを排除し、作成したソースコードがそのまま実行されるように手順を設定している。

そのため、作成したプログラム内で複数の関数を記述した場合、作成されるバイナリファイルは、記述したソースコード順に忠実に配置される。すなわち、ファイルの中で、main 関数よりも前に関数を記述すると、その関数が main よりも前に実行される。(必ずしも main 関数が最初に実行されるわけではない)

一方、C 言語の特性から、main 関数 の中で、main 関数 よりも後に、別の関数を記述する場合、main 関数の記述よりも前に、プロトタイプ宣言が必要となる。以下に例を示す。

```
typedef volatile unsigned short hword;
#define BGR(r, g, b)      ((b << 10) + (g << 5) + r)

void draw_point (hword, hword, hword); /* 関数のプロトタイプ宣言 */

/* main 関数の記述 (main 関数を最初に記述) */
int main (void) {
    hword x1;
    hword y1;
    hword color1;
    /* (中略) */
    draw_point (x1, y1, color1); // 関数を呼ぶ
    /* (中略) */
} // main 関数の終わりのカッコ

// draw_point の関数の記述 (その他の関数を main 関数よりも後に記述)
void draw_point (hword x, hword y, hword color) {
    / *ここで指定された(x, y)の位置に、指定された色の点を書く */
} // draw 関数の終わりのカッコ
```

12.5.6. Volatile (ヴォラタイル)

volatile は型修飾子の一つ (const も型修飾子)。Kernighan&Ritchie : 「プログラミング言語 C」では、「volatile の目的は、黙っていると処理系で行われる最適化を抑止することにある。例えば、メモリ・マップ方式の入出力をもつマシンでは、ステータス・レジスタに対するポインタは、ポインタによる見かけ上、冗長な参照をコンパイラが除去するのを防ぐのに、volatile へのポインタと宣言することが可能である。」と書いている。

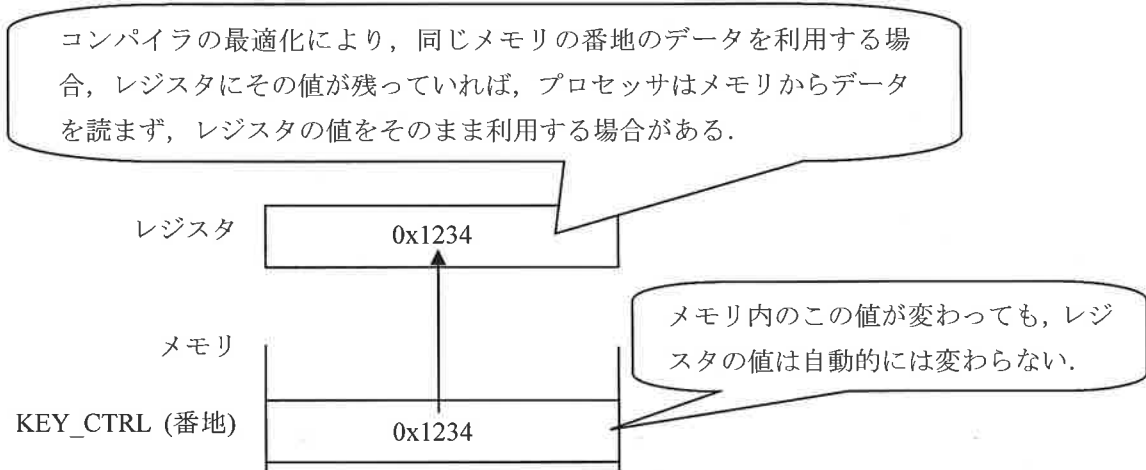
一般的に、プログラムをコンパイルする際に、コンパイラは自動的に最適化を行う。たとえば、

```
int i=3;
```

としても、プログラムの中でこの変数 i を利用しなければ、コンパイラはこの i は不要なものとして、削除する場合がある。

```
key = *(hword *) KEY_CTRL;
```

として、キー (ボタン) の値を読み込んだ場合でも、コンパイラはこの KEY がキー (ボタン) 入力のデータを読み込んでいると判断できず、メモリ内の同じアドレスのデータは、プログラムが書き換えない限り変化しないと判断する可能性がある。すなわち、最初にこの命令を実行したときに、このアドレスが指し示すデータをレジスタに入れ、再び、同じ命令が実行されても、このアドレスが指し示すデータを直接読まずに、レジスタの値を再利用する場合がある。通常、メモリからデータを読むより、レジスタの値をそのまま利用したほうが高速であるため、コンパイラがこのような最適化を行う。しかし、キー (ボタン) の値を読む場合、レジスタの値をそのまま利用すると、キー (ボタン) の状態が変わったことをプログラムが認識できなくなる。



そこで、変数に volatile をつけて宣言すると、コンパイラは、最適化のためにレジスタに保存されている値を利用せず、アドレスが指し示すデータを読みに行く。

12.5.7. 線分描画について

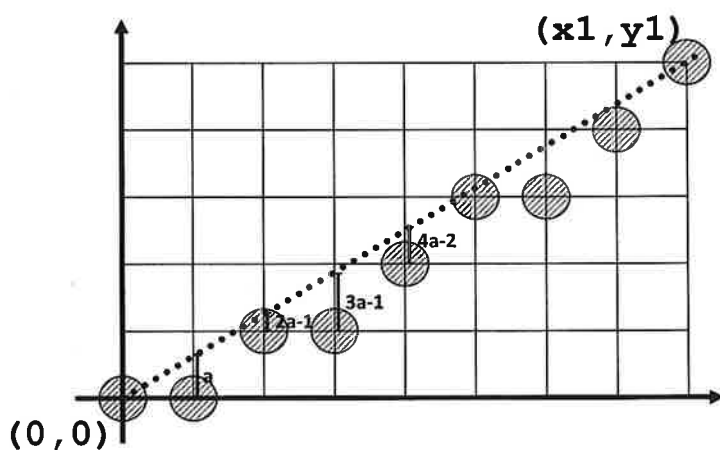
画面上に線分を描画するにはなんらかのアルゴリズムを考える必要がある。以下に線分描画アル

ゴリズムの一例を示す。これ以外にも線分描画を行うアルゴリズムはいくつかあり、本実験における課題の実施には、どのようなアルゴリズムを利用してもよい。

アルゴリズムの説明を簡単にするため、ここでは原点から傾き a の線分を考える。(実際の課題は原点ではなく画面上の特定の点である)

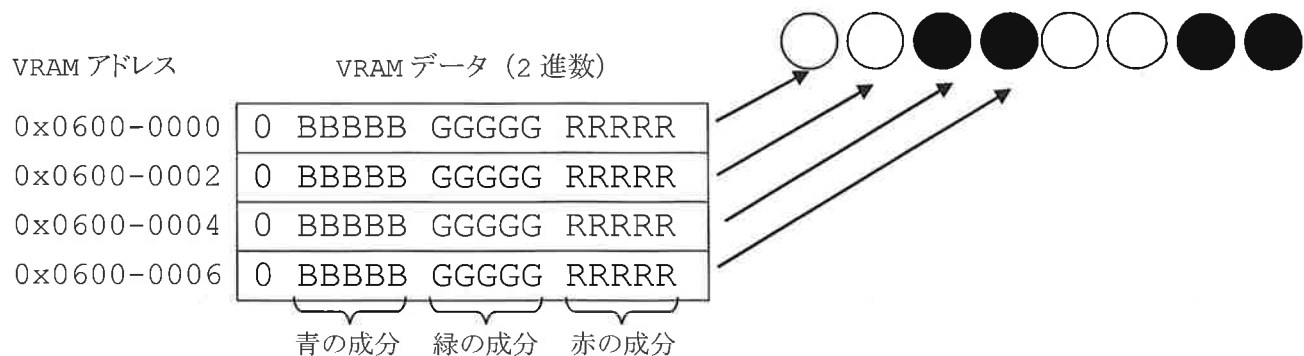
- 線分の式を $y = ax$ ($0 \leq x \leq x_1$) とする。また、 a は $1/2$ 以下とする。
- ここで傾き $a = y_1 / x_1$ となる。
- 表示画面では、整数の座標にしか点を描画できない。
- x 軸方向に+1 ごと、 y のどの値に点を描画するか判定を行う。
- ここでは、直線の直下の y の整数値の位置に点を描画するものとする。
- $x=0$ の場合を検討する。
- $(0,0)$ は始点なので、この位置に点を描画する。
- 次に、 $x=1$ の場合を検討する。
- $(1,0)$ と $(1,1)$ のどちらに点を描画するかを考える。
- 直線の直下の $(1,0)$ に描画する。
- 次に、 $x=2$ の場合を検討する。
- $(2,1)$ と $(2,2)$ のどちらに点を描画するかを考える。
- 直線の直下の $(2,1)$ に描画する。
- . . .
- $x=x_1$ まで繰り返す

注意：このままでは小数点、割り算を利用する必要が生じるので、整数および足し算で実現できるよう工夫する必要がある。



12.6. BGR(r, g, b) 宣言についての説明

GBA は 3 万 2 千色を表現することが可能で、液晶画面の 1 ドットを 15 ビットで表わす。



	VRAM データ (2 進数)	16 進数	BGR(r, g, b) 表現
黒色	0 00000 00000 00000	0x0000	BGR(0x00, 0x00, 0x00)
赤色	0 00000 00000 11111	0x001F	BGR(0x1F, 0x00, 0x00)
緑色	0 00000 11111 00000	0x03E0	BGR(0x00, 0x1F, 0x00)
青色	0 11111 00000 00000	0x7C00	BGR(0x00, 0x00, 0x1F)
黄色	0 00000 11111 11111	0x03FF	BGR(0x1F, 0x1F, 0x00)
シアン(水色)	0 11111 11111 00000	0x7FE0	BGR(0x00, 0x1F, 0x1F)
マゼンタ	0 11111 00000 11111	0x7C1F	BGR(0x1F, 0x00, 0x1F)
白色	0 11111 11111 11111	0x7FFF	BGR(0x1F, 0x1F, 0x1F)

```
#define BGR(r, g, b) ((b << 10) + (g<<5) + r)
```

このコードは、プログラム中の「BGR(r, g, b)」の文字列を「((b << 10) + (g<<5) + r)」の文字列に置き換える。ここで、「<<」演算子は、左へ特定ビット数のシフトを意味する。

たとえば、BGR(0x07, 0x0A, 0x0F)とすると 0x04 を左へ 10 ビットシフトし、0x08 を左へ 5 ビットシフトし、0x0C はシフトせず、これら 3 つの値を足すという意味になる。

設定値 (16 進数)	設定値 (2 進数)	シフト後の値 (2 進数)	命令
0x07	0 00000 00000 <u>00111</u>	0 <u>00111</u> 00000 00000	左へ 10 ビットシフト
0x0A	0 00000 00000 <u>01010</u>	0 00000 <u>01010</u> 00000	左へ 5 ビットシフト
0x0F	0 00000 00000 <u>01111</u>	0 00000 00000 <u>01111</u>	シフト無し
		0 <u>00111</u> <u>01010</u> <u>01111</u>	足し算した合計

12.7. FAQ

12.7.1. コンパイル時のエラーメッセージ

Q01	コンパイル時のエラーメッセージについて教えてください。
A01	・ エラーメッセージの数字の意味：ファイル名の直後の数字が、エラーの原因となっているソースコード内の行数を示しています。たとえば、dots2.c:5:6 error: ...

	<p>の場合、5 行目にエラーの原因があります。(注：文の最後を書くべきの ; が抜けている場合は、次の行が示される場合がありますので、示された行の前後も含めて確認して下さい)</p> <ul style="list-style-type: none"> エラーメッセージがたくさん表示されてよくわからない場合は、まず初めに表示されているエラーメッセージを解決して下さい。 error: No such file or directory 「そのようなファイルあるいはディレクトリは存在しません」：コンパイルしようとしているファイル名が間違っているか、作成されていません。 error: 'i' undeclared 「変数 i が未宣言」：使用された変数名が宣言されていないか、綴りが間違っている可能性があります。 error: parse error at end of input 「入力の最後での構文解析エラー」：文の最後の } がどこかで抜けている可能性があります。 error: too few arguments to function 'draw_point' 「draw_point の関数で引数が不足」：関数呼び出しの際の引数の数が呼び出す側が、関数の定義と合っていません。 error: stray '¥201' in program：プログラム中に全角のスペースが入っています。スペースに限らず、プログラム中の全角文字はすべてエラーになります (コメント内/* . . . */ の全角文字の使用は問題ありません) error: 'for' loop initial declarations are only allowed in C99mode 「C99 モード以外では、for 文内で宣言することは許可されていません」：ここで利用するコンパイラは C99 モードではないため、Java のように for 文の中で変数を宣言できません。たとえば、for (int i = 0; i < 10; i++) としていれば、int i; の宣言を for の外に出して下さい。
--	---

12.7.2. コンパイル時の警告メッセージ

Q02	コンパイル時の警告メッセージについて教えてください。
A02	<ul style="list-style-type: none"> 警告メッセージは無視してもコンパイルが終了してアセンブリ言語プログラムを出力しますが、現時点で、無視してよい警告かどうかは必ず確認して下さい。無視してよくない警告を無視すると、プログラムが正常に動作しない場合があります。 warning: assignment makes pointer from integer without a cast 「キャストなしに整数をポインタを割り当てています」：C 言語において、= の両側の変数は同じ型である必要があるのですが、たとえば、左辺がポインタ型、右辺が整数型の場合、警告が表示されます。当面は無視して問題ありません。 warning: unused variable 'i' 「変数 i が未使用」：作成したプログラム内で、「i」を定義したのに、使っていないという警告です。定義文を削除すれば警告は出なくなります。無視して問題ありません。 warning: no newline at end of file 「ファイルの最後の改行がない」：プログラムの最後の行に改行を入れて下さい。無視して問題ありません。

12.7.3. コンパイル・アセンブル・リンク

Q03	<p>コンパイル (cc-arm 実行) した際に、</p> <pre>error: 'r' undeclared (first use in this function) note: each undeclared identifier is reported only once for each function it appears in . . .</pre> <p>というエラーが発生します。</p>
A03	#define BGR(r, g, b)

	は定義文（プログラム中のこの文字列を置き換える）ですので、BGR と括弧の間にスペースがあつてはいけません。スペースが入っていないか確認してください。
Q04	コンパイル（cc-arm 実行）した際に、 error: syntax error before '{' token error: parse error before "hword" error: initializer element is not constant warning: data definition has no type or storage class などのエラーがいっぱい表示されます。
A04	作成したプログラムの main の後ろに() がついているかどうか確認してください。
Q05	リンク（ld-arm 実行）した際に ld-arm: warning: cannot find entry symbol _start; defaulting to 02000000 というウォーニング（警告）がでます。
A05	プログラムの実行を 0x020000000 番地から実行するように作成していますが、そこに _start というシンボルがセットされていないという警告です。アセンブリ言語から、コマンドにより直接実行プログラムを作成しているため、このような警告がでますが、プログラムの実行には問題ありませんので、現時点で無視して問題ありません。
Q06	リンク（ld-arm 実行）した際に undefined reference to '__aeabi_idiv' というエラーが表示されます。
A06	ここで利用するコンパイラは割り算をサポートしていませんので、プログラムにおいてはそのままでは割り算を使えません。割り算を使いたい場合は、自分で割り算の関数を作成して下さい。
Q07	リンク（ld-arm 実行）した際に undefined reference to 'memcpy' というエラーが表示されます。
A07	関数内の変数（自動変数）として配列を定義すると、高速実行のためのコンパイラが memcpy というライブラリを読み出そうとします。しかし、実際にはライブラリがないためのエラーが発生します。配列の定義文の先頭に static と記述すると、自動変数ではなく、静的変数として確保されますので、エラーがでなくなります。詳細な仕組みについては C 言語の授業で説明します。
Q08	ダウンロード（dl-gba 実行）した際に Error: file open failure! あるいは Error: file read error! というエラーが表示されます。
A08	バイナリファイル（.bin）が正常に作成されていません。アセンブリ言語のファイル（.S）の中身が空の場合、as-arm などそれぞれのコマンドを実行しても、各コマンドにおいてエラーは表示されません。ファイルが作成されているかを確認してください。

12.7.4. プログラム実行

Q09	アセンブリ言語で作成したプログラムを実行しても、GBA の画面がまったく変わりません。
A09	プログラム内で、以下の手順で、画面モードを初期設定する必要があります。 <pre> mov r1, #0x04000000 ldr r2, =0xF03 strh r2, [r1] </pre> これらは GBA の画面の初期設定であり、今は内容を理解する必要はありません。

Q10	C言語で作成したプログラムを実行しても、GBA の画面がまったく変わりません。
A10	プログラム内で、以下の手順で、画面モードを初期設定する必要があります。 <code>*(hword *)0x04000000 = 0x0F03;</code> これらは GBA の画面の初期設定であり、今は内容を理解する必要がありません。
Q11	for 文でループさせてドットを複数表示しようとしているのに、ドットは1点しか表示されません。
A11	for 文の中にポインタの初期設定が入っていないか、確認してください。たとえば、 <pre>for (i = 0; i < 240; i = i + 1) { ptr = (hword*) VRAM; *ptr = BGR(0x1F, 0x1F, 0x1F); ptr = ptr + 1; }</pre> となっている場合、常に同じ位置にドットが書かれます。
Q12	プログラム実行時に、キー入力 that 2回目以降、受け付けられません。
A12	キーを判定する文、たとえば、 <code>if (key & 0x03FF)</code> の前で、 <code>key</code> のデータが読み込まれていることを確認してください。たとえば、 <code>key = *(hword *) KEY_CTRL;</code> あるいは、while ループの中で <code>key</code> のデータを読み込まれていることを確認して下さい。while ループがない、あるいは、while ループの外で <code>key</code> のデータが読み込まれていると、 <code>key</code> のデータは1度しか読み込まれず、2回目以降の動作が行われません。
Q13	プログラムは正しいのに、キー入力が受け付けられません。
A13	キー入力のためのポインタ宣言に <code>volatile</code> が含まれているか確認してください。 <code>volatile</code> の宣言がないと、プログラムが正しくても動作しない場合があります（動作する場合もあります）

12.7.5. PC 操作

Q14	PC で入力するコマンドが長いので、入力するのに時間がかかります。よい方法がありますか？
A14	Linux にはヒストリ機能というのがあり、過去に入力したコマンドを覚えています。キーボードの上向き矢印を入力すると、過去に入力したコマンドを再入力できます。また、コマンドの一部を編集して入力することも可能です。下向き矢印で先のコマンドにも行けます。「history」というコマンドを入力すると、過去に入力したコマンドの一覧が表示されます。「！」と、コマンドの番号を入力すると、そのコマンドが再実行される機能もあります。
Q15	データ保存 USB メモリを取り出そうとしたのですが、「ドライブを停止できませんでした」というエラーが表示されます。
A15	USB メモリ内の開いているファイル、フォルダをすべて閉じてから、再び、マウント解除してください。
Q16	デスクトップ上の USB メモリのアイコン <code>sdc1</code> をクリックすると、「デバイスをマウントできませんでした」と表示されます。
A16	エラー表示を閉じた後に、もう一度、トライしてみてください。何度かトライしてダメな場合は、USB メモリを抜き、再び、挿入してください。そして、10 秒ほど待った後に、アイコンをクリックしてください。
Q17	データ保存 USB メモリにデータが書き込めません。

A17	USB メモリのマウントを解除し、プロパティのリードオンリーを解除した後に、再び、オープンしてください。
-----	--

Q18	dl-gba を実行してファイルを GBA にダウンロードしようとする、 Status = Negotiation error! と表示され、正常にダウンロードできません。
A18	GBA の電源を入れ直し、画面全体が白色になった後に dl-gba を実行してください。それでも同じエラーが出る場合には、PC と GBA を接続するケーブルが正常につながっていません。GBA の電源をオフにし、PC、GBA のコネクタを挿し直した後に、GBA の電源を入れて下さい。特に、GBA 側のコネクタをきっちり挿入してください。

Q19	dl-gba を実行してファイルを GBA にダウンロードしようとしても、正常にダウンロードできません。エラーメッセージも表示されません。(ダウンロードの際に、 Detail = send_file: File data transmission error というエラーメッセージが表示される場合もあります)
A19	Ubuntu の USB ドライバがおかしくなっている場合が考えられます。まず、GBA の電源を切り、PC と GBA を接続しているケーブルの <u>PC 側の USB ケーブルを抜いてください</u> (GBA 側のケーブルは抜かないでください)。その後、もう一度 USB を挿入し、GBA の電源を入れて下さい。

Q20	PC においてインターネットが利用できません。
A20	画面右上の無線 LAN のアイコンに「!」が付いていないか確認してください。「!」が付いている場合には、無線 LAN のアイコンをクリックし、「利用可能」リストから「DO-NET」を選択してください。無線 LAN アイコンをクリックした時に「無線は無効になっています」と表示されている場合は、ハードウェアの無線 LAN 機能が無効になっています。キーボードの上の中央にある無線 LAN アイコンがオレンジ色になっているので、指で触れて青色になったことを確認してください。

Q21	自分の USB メモリにファイルを保存してもよいでしょうか
A21	構いません。ただ、授業の最後には、班のメンバ全員に対してその日作成したすべてのファイルをメールで送信してください。

12.7.4. GBA 操作

Q22	GBA の電源を入れても、画面がクリアされず文字が表示され、ダウンロード可能状態になりません。
A22	PC との接続ケーブルが正常に挿入されているか確認してください。また、電源をオンする際には、GBA の他のキーを押さないようにしてください。

12.7.5. 課題に関して

Q23	プログラムで小数は使えないのですか。
A23	この課題では、ハードウェアにおいて直接プログラムを実行しています。プロセッサが機能として小数の演算器を持っていれば小数が使えますが、ARM には小数の機能がありませんので、ここでも使えません。 小数を使いたい場合は、自分で桁数をずらして、整数演算としてください。
Q24	プログラムで割り算が使えないのですが

A24	この課題では、ハードウェアにおいて直接プログラムを実行しています。プロセッサが機能として割り算命令を持っていれば割り算は使えますが、ARM には割り算の機能がありませんので、ここでも使えません。(分母が定数の割り算のみ使用できます) 割り算を使いたい場合は、自分で割り算の関数を作成してください。引き算を繰り返すことで割り算を実現できます。
-----	--

Q25	プログラムで乱数を使いたいのですが。
A25	この課題では、ハードウェアにおいて直接プログラムを実行しています。つまり、高度なプログラム環境で用意されている乱数などはありません。 乱数を作成するためには、いくつかの手法がありますが、そのうちの1つとして、ゲームをする人(プレーヤ)の動作を利用する方法があります。プログラムの開始時に関数の中でカウントするループを動かします。プレーヤが START ボタンを押したら、そのループを抜けるようにし、抜けるタイミングでカウント値を利用することで乱数を作れます。

Q26	線分の描画がわかりません。
A26	傾きが 45 度でない線分の描画を行うアルゴリズムは一般的に難しいです。整数の座標系の上で直線を描画するためには、工夫が必要です。また、割り算も使えませんので、自分で工夫してください。

Q27	円の塗りつぶし描画がわかりません。
A27	塗りつぶしでない輪郭だけの円の描画は簡単ではありませんが、塗りつぶしの場合は比較的簡単です。円の方程式を思い出して、考えてみてください。

12.8. レポート作成、および、提出方法（重要）

レポート作成には、Office365（www.office.com）の Word を使用し、以下の手順を守ってファイルを作成すること。

- Office365 をブラウザから起動し、左のタブから Word を立ち上げ、「新規作成」で「新しい空白の文書」をクリックすると、「ドキュメント.docx」が開く。
- 文字を入力する前に、変更履歴の記録を行う。**
 - 図 1 の例のように、メニューバーの「校閲」を選択し、「変更履歴の記録」の「すべてのユーザー」をクリックする。
 - メニューのところに「チェック/コメント」と表示されているか確認する。
 - このまま入力すると、文字に色と下線がつくが、気にせずこのまま書き続ける。
- メニューバーの「ファイル」の「名前を付けて保存」を選択し、ファイル名は、「1116????????_同志社花子.docx」のように、学生 ID_氏名.docx として、自分のパソコンに保存する。
- この「学生 ID_氏名.docx」のファイルを e-class にアップロードする。
- 提出したレポートは、e-class の「マイレポート」から確認可能である。期限をすぎた場合は、自動的に提出できなくなる。期限内であれば、修正したドキュメントを再アップロード可能である。レポートに関する採点結果は公表しない。
- レポートの Word ファイルを修正する場合は、Office365 の Word から（OneDrive に保存されている）以前のファイルを開くか、自分のパソコンに保存した「学生 ID_氏名.docx」を開いて修正する。その際には、上記に示した **変更履歴の記録を行う** を忘れないこと。
 - Office365 の Word から自分のパソコンに保存したファイルを開くには、Office365 の左のタブから「ホーム」を選択し、「クイックアクセス」の「アップロード」から行うことができる。



図 1: 変更履歴の設定の確認

注記：自分のパソコンにインストール済みの Word を利用する場合は、レポートのファイルを開き、図 2 のようにメニューバーの「校閲」を選択し、「変更履歴の記録▼」の▼から「変更履歴の記録」を選択する。（再修正する場合もこの設定を行う）

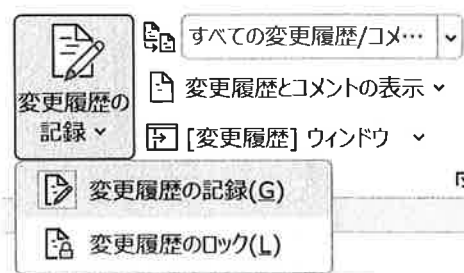


図 2: 変更履歴の設定