基礎ソフト実験Ｉ　レポート２

締切：10月22日（金）正午

問４.データ転送サブルーチンの作成

アドレスレジスタ a1 で指定されるアドレスから始まる連続する 2n バイトのメモリの内容を アドレスレジスタ a2 で指定されるアドレスから始まる領域へと転送するサブルーチンを，68000 アセンブラを用いて作成し，下記の設問(1)から(6)に解答せよ.

簡単なプログラム作成なので，すべて自力で行なうことが望ましい．やむを得ず他人の解答の一部または全部を参考にする場合には，礼儀として誰の解答を参考にしたかを明記すること，そして，下記の設問(1)から(6)において，必ず自分なりの工夫を行なうこと．
「誰の解答を参考にしたかを明記」していなかったり，他人の解答を丸写し（ちょっと語句を変えたぐらいではだめ）している場合は，マイナス点を与え，再提出を命じる．独自性を発揮せよ.

(1)プログラムを作成し，プログラムを明記せよ
- ・繰り返しループ，比較命令，move 命令などを組み合わせること.
- ・サブルーチン内では movem を用いてレジスタの退避，回復を行うこと.
- ・サブルーチンの末尾には，rts を忘れないこと.
- ・余裕があれば，あなたのプログラムが「いろいろな文字列を容易に扱える」ようにするための工夫を行なうこと．例えば，文字列の長さを equ 疑似命令でシンボル化するなど.
- ・余裕があれば，分かりやすいコメントをプログラム中に記述せよ.

(2)あなたが作成したプログラムの説明を行え.
- ・プログラムの目的，レジスタ用途，メモリ内のデータの配置，プログラムの入力と出力，

プログラム作成上工夫したところ，などのうちから適当に説明を行え.

(3)あなたが作成したプログラムの使い方を簡単に説明せよ
あなたが作成したプログラムを他の人が使うものと考えて，プログラムの使い方（文字列や，文字列の長さの指定法，プログラムの実行法など）を簡単に説明せよ

(4)実行結果の報告を行え.

画面のハードコピー，あるいは，実行結果を手で書き写したものを解答すること．
確かに正しく動いていることが分かるような簡単な説明があると望ましい．

(5)転送後に，転送元の領域を 0 で上書きする必要性があるかないか，自分の考えをその理由とともに述べよ．（理由が合理的であれば得点を与える）

(6)余裕があれば，感想，意見，今後可能なら改良すべき点を述べよ(内容に応じて得点を与える)

（注意）アドレスレジスタ a1 で指定される領域と，アドレスレジスタ a2 で指定される領域は重ならないものとしてプログラミングを行ってかまわない．

問５．　メモリ破壊

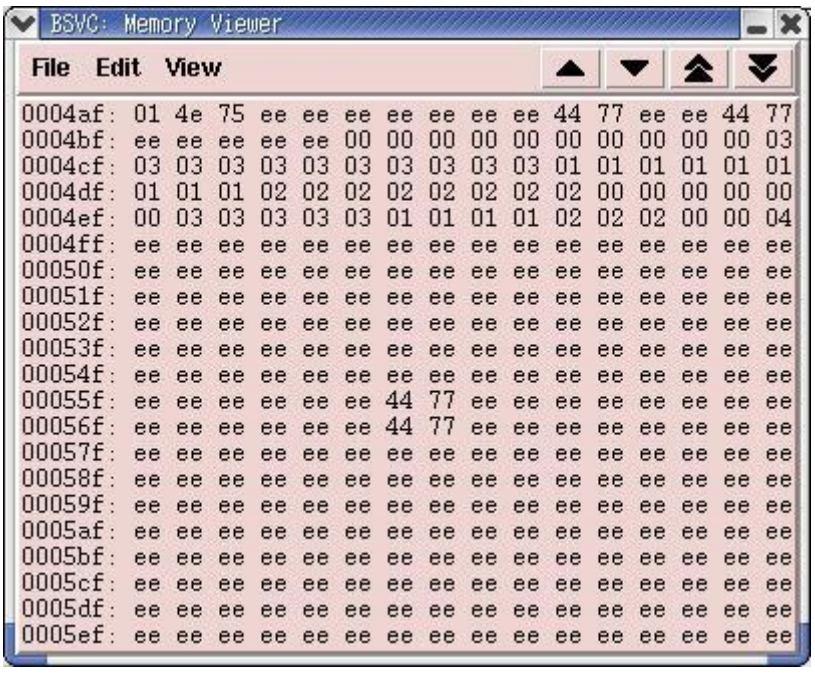
・プログラムの概要

このプログラムは、先のデータ転送プログラム（問題 4）を応用して、メモリ領域をキャンバスに見立てて、数値(0～9,a～f)を使って絵を描画し、その絵を指定された方向(右、左、上、下）に動かすプログラムである。絵となるデータはプログラム中の DATA0 から DATA3 の部分であり、

ee ee ee ee
ee 44 77 ee
ee 44 77 ee
ee ee ee ee

という絵になっている。

このプログラムを実行すると、”ee”で埋め尽くされた領域を、上記の絵が時計回りに渦を巻くように動いていく。この絵の動きを制御するのは、データ領域の CONTROL にある数値(0～4)である。0～3 はそれぞれ順番に右、左、上、下に対応しており、4 は終了を意味している。



・プログラムの実行方法

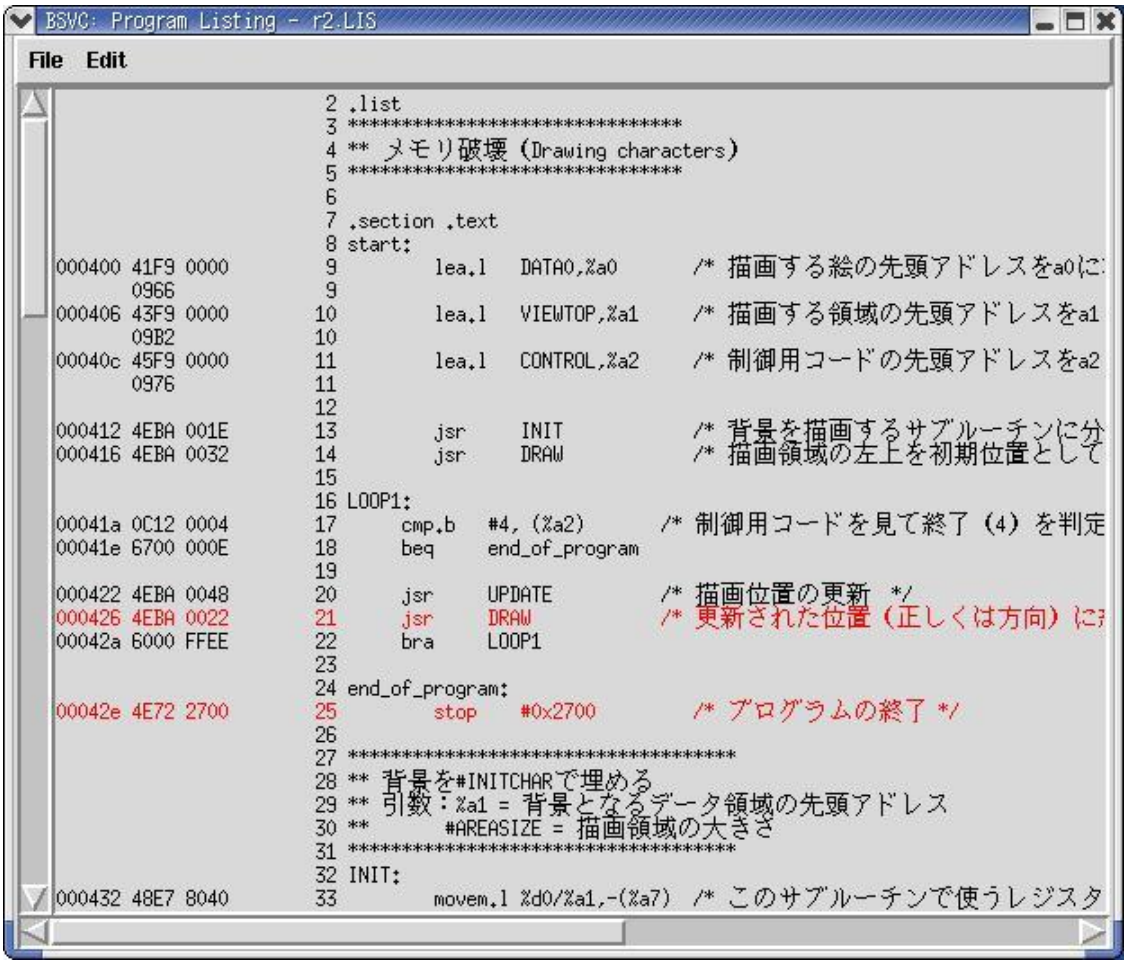
プログラムは 0x400 番地から配置される。また、キャンバスに見立てているメモリ領域は VIEWTOP というラベルから始まる 256 バイトとしている。この領域に“絵”を描画する。プログラムの動作を見やすくするために、必ず以下の処理を行うこと。

（１） エミュレータを起動し、実行プログラムの読み込んだ後、VIEWTOP ラベルのアドレスをListing ウィンドウで調べる(キャンバスの左上隅に相当。前頁の図の例では 0x4ff であったが、この値は変わる場合があるので自分の環境で確認をすること）。

（２） Memory Viewer ウィンドウを開き、メニューの「View」－「From Address…」を選択し、調べた VIEWTOP のアドレスを入力する(要は VIEWTOP のアドレスがウィンドウの一番左側になるように表示を調節する）。

（３） プログラム領域も同時に観察できるように、ウィンドウ右上の↑ボタンを数回押して、表示領域を調節する（前頁の図を参照）。

ここで注意することは、エミュレータ上でいきなり”RUN”を実行してしまうと、一瞬でプログラムが終了するため、絵がどのように動くのかを観察することができない。そのため、プログラム中の「LOOP1：」の行から４行下の「jsr DRAW」の行に break point を設定する（当然、stop 命令の箇所にも break point を設定）。上記の準備ができたら、”RUN”を繰り返し押しながら、メモリ領域（キャンバスに見立てた領域）を見ましょう（前頁図を参照）。これにより、絵を描画し終わったところ（実際には絵となるデータを転送し終わったところ）でプログラムを毎回停止させることができ、絵が指定の方向に描画されていくのが確認できる。（”step”実行でプログラムの動きを逐次追うことが望ましい）



【課題内容】

(1) 実際に，実習室でこのプログラムを打ち込んで動作させ，ステップ実行によってどのようにメモリ領域が変化して行くのかを報告せよ．画面のハードコピー，あるいは，実行結果を手で書き写したものを解答すること．確かに正しく動いていることが分かるような簡単な説明があると望ましい．

(2) プログラム中のデータの部分に「* CONTROL」で始まるコメントされた行がある。この行頭の「＊」を外し、代わりにその前の行の CONTROL をコメントアウトして実行すると、本問題のタイトルである「メモリ破壊」が起こる。どのようなことが起こったか、またなぜそのようなことが起こったのかを説明せよ。

(3) 上記のメモリ破壊を防ぐためには、描画領域（転送して良いメモリ領域）の境界条件を考慮する必要がある。改良すべき点について述べよ（内容に応じて得点を与える）。

問５のソースプログラム

****

```
** メモリ破壊 (Drawing characters)
*******************************

.section .text
start:
        lea.l   DATA0,%a0       /* 描画する絵の先頭アドレスを a0 に格納 */
        lea.l   VIEWTOP,%a1     /* 描画する領域の先頭アドレスを a1 に格納 */
        lea.l   CONTROL,%a2     /* 制御用コードの先頭アドレスを a2 に格納 */

        jsr     INIT            /* 背景を描画するサブルーチンに分岐 */
        jsr     DRAW            /* 描画領域の左上を初期位置として絵を描画 */

LOOP1:
        cmp.b   #4, (%a2)       /* 制御用コードを見て終了 (4) を判定  */
        beq     end_of_program

        jsr     UPDATE          /* 描画位置の更新  */
        jsr     DRAW            /* 更新された位置（正しくは方向）に絵を描画  */
        bra     LOOP1

end_of_program:
        stop    #0x2700         /* プログラムの終了 */

************************************
** 背景を#INITCHAR で埋める
** 引数 : %a1 = 背景となるデータ領域の先頭アドレス
** #AREASIZE = 描画領域の大きさ
************************************
INIT:
        movem.l %d0/%a1,-(%a7)  /* このサブルーチンで使うレジスタを退避 */
        move.w  #AREASIZE,%d0
LOOP2:
        move.b  #INITCHAR,(%a1)+
        subq.w  #1,%d0
        bne     LOOP2
```

```
        movem.l (%a7)+,%d0/%a1   /* レジスタの回復 */
        rts
************************************
** DRAW
** 引数：
**      %a0=動かす絵の先頭アドレス
**      %a1=絵を描画する領域の先頭アドレス（描画位置）
************************************
DRAW:
        movem.l %d0-%d1/%a0-%a1,-(%a7) /* レジスタの退避  */

        moveq.l #LENGTHX,%d0
        moveq.l #LENGTHY,%d1


LOOP3:
        move.b  (%a0)+,(%a1)+
        subq.w  #1,%d0
        bne     LOOP3

        adda.w  #16-LENGTHX,%a1
        moveq.l #LENGTHX,%d0
        subq.w  #1,%d1
        bne     LOOP3
        movem.l (%a7)+, %d0-%d1/%a0-%a1 /* レジスタの回復 */
        rts

************************************
** 制御コードによる絵の位置情報（アドレス）の更新
** 引数
**   %a2=現在の制御コードが格納してあるアドレス
** 戻り値
**   %a1=絵を描画する領域の先頭アドレス（位置）
************************************
UPDATE:
```

```
        cmpi.b  #0,(%a2)                /* 右に動かす */
        beq     RIGHT
        cmpi.b  #1,(%a2)                /* 左に動かす */
        beq     LEFT
        cmpi.b  #2,(%a2)                /* 上に動かす */
        beq     UP
        cmpi.b  #3,(%a2)                /* 下に動かす */
        beq     DOWN

RIGHT:
        adda.w  #1,%a1
        bra     FINISH
LEFT:
        suba.w  #1,%a1
        bra     FINISH
UP:
        suba.w  #0x10,%a1
        bra     FINISH
DOWN:
        adda.w  #0x10,%a1
        bra     FINISH

FINISH:
        adda.w  #1,%a2
        rts

.section .data
**********************************
** Data Area
**********************************
        .equ    LENGTHX, 4      /* 描画する絵の横幅（バイト単位） */
        .equ    LENGTHY, 4      /* 描画する絵の縦幅（バイト単位） */
        .equ    INITCHAR, 0xee   /* 背景に使う”文字(0x00-0xff) */
        .equ    AREASIZE, 0x100 /* 描画に使用する領域の大きさ */

DATA0:  .dc.b   INITCHAR,INITCHAR,INITCHAR,INITCHAR     /* 描画する絵の一部 */
```

```
DATA1:   .dc.b    INITCHAR,0x44,0x77,INITCHAR              /* 描画する絵の一部 */
DATA2:   .dc.b    INITCHAR,0x44,0x77,INITCHAR              /* 描画する絵の一部 */
DATA3:   .dc.b    INITCHAR,INITCHAR,INITCHAR,INITCHAR      /* 描画する絵の一部 */

CONTROL:.dc.b
0,0,0,0,0,0,0,0,0,0,3,3,3,3,3,3,3,3,3,3,3,1,1,1,1,1,1,1,1,1,2,2,2,2,2,2,2,2,0,0,
0,0,0,0,3,3,3,3,3,1,1,1,1,2,2,2,0,0,4
*CONTROL:.dc.b 0,0,0,0,0,0,0,0,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,4        /* 絵の制御用
コード */


VIEWTOP:          .ds.b    AREASIZE                    /* 領域の確保 */

.end
```

問６．キューの実装

キュー入出力サブルーチンプログラム QueueIn, QueueOut を作成せよ．ここでは，以下数ページにわたって，「キューの説明」，「作成すべきプログラム QueueIn, QueueOut の仕様」，「プログラム作成上のヒント」などを説明する．本設問では，これらの情報をもとに，実際のプログラム作成とそのテスト実行を行ってもらいたい．

必ず設問 6 まで読んでから作業に入ること．また，設問 6 を挑戦する人はその仕様が説明パートの最後の方に記載してあるので，参照すること．

（注意）条件分岐やジャンプ命令とは切り離して，サブルーチンの形式を順守せよ．守ってない場合は大きく減点する．

設問１　以下に説明する QueueInおよびQueueOutのサブルーチンを，68000アセンブラを用いて作成せよ．

- プログラムリストのコピー（プリンタなどで印字したものでよい）を添付すること．
- プログラムの説明（プログラムの目的，レジスタ用途，メモリ内のデータの配置，要所におけるスタックの動作，プログラムの入力と出力，などのうちから適当に）も行うこと．
- プログラムの使い方の説明も行なうこと．
- 「プログラム作成上工夫したところ」を必ず説明すること

設問 2　QueueIn と QueueOut を呼び出すような簡単なプログラムを，上記設問 1 のプログラムに追加し，**全体として実行可能なプログラムを作成せよ．**

設問 3　設問 3a または 設問 3b に解答せよ

設問 3a　プログラムを実際に動作させ実行結果（画面のコピー，実行の様子の記述）を報告せよ．

設問 3b　もし，あなたのプログラムがうまく動作しなかった場合，（１）うまく動作しない理由，（２）どうすれば動くようになるかを考えよ．これら２点について十分に検討し，論理的かつ詳細に報告せよ．

設問 4　キューが一杯の時，キューが空の時において，あなたのプログラムがどういう動作を行ったかを報告せよ．そして，それが正しい動作であるかどうかを判断し報告せよ（必要ならプログラムの修正を行え）．

設問 5　感想，意見，今回作成したプログラムの改良すべき点を述べよ.

設問 6　余裕のある人は，この問題を解け（出来に応じて，評価得点を追加する）.
キューは，通信などを行う際のバッファとして使われることがある．例えば，２つのチャンネルを持つ通信回線を考えた場合，１つのチャンネルにつき送信用キューと受信用キューが必要となれば，合計で４つのキューを切り替えて（選択して）使うことになる．設問 6 では，先に作成したキュー（単純なキュー）のプログラムを拡張し，４つのキュー（キュー#0,#1,#2,#3）を任意に選択することが出来る機能を追加したプログラムを作成する（拡張したキュー）．どのキューを選択するかは，呼び出し側でレジスタを用いて指定する．実装方法の詳細については，ヒント（８）を参考にせよ．但し，拡張したキューが完成した場合，設問１から設問５はすべて拡張したキューについてレポートせよ(単純なキューについての説明をする必要はない）.

実際は，キューの実装法は，ここに説明した通りに作る必要はない．例えば，ここで説明している「GET_FLG, PUT_FLG」を使わなくても，キューを実装することは可能である．もし，他の実装法を思い付いたら，ぜひ挑戦してもらいたい（但し，「キュー」として正しく動くこと）．そして，レポートとして報告してもらいたい．独自の工夫をしていれば，高く評価し，高得点を与える.

[キューとは]

キューは，データを一時的に保管しておくような領域（バッファ）の１種で，データを到着順に詰め込み，そして，詰め込んだ順に取り出すことができます．つまり１，２，３の順でデータを入れたら１，２，３の順でデータを取り出せます．スタックの場合には，３，２，１の順でデータを取り出すので，スタックとキューはデータの取り出し順が逆になります．そのことから キューは，First In First Out とも呼ばれます（対して，スタックは First In Last Out と呼ばれます）.

[キューの役割]

キューは，しばしば，外部とコンピュータ本体との間のバッファとして使われます.
すなわち,
（１）外部の処理速度と CPU の処理速度のずれの調整,
（２）CPU が外部の見張りに能力を奪われることなく，外部との通信を可能にする,
などの機能を果たします.

例えば，ネットワークで多量のデータをやり取りする場合，送られてくるデータの速度

に CPU 側が追い付かないようなことが起こり得ます．そこで，外部から送られてきたデータをいったんキューへ転送し，そのあとキューから取り込むようにすれば，CPU 側に時間の余裕ができます．あるいは，外部から送られてくるデータの速度が CPU 側よりも遅いことも起こり得ます．そのとき，CPU が常にデータの到着を見張るのでは，CPU の能力が無駄になってしまいます．そこで，データ到着時に，いったん，キューにデータを置き，ハードウエア割り込みの機能を利用するようにします．そうすれば，CPU が常にデータの到着を見張る必要がなくなります．このように，外部とコンピュータ本体の間に置かれたキューを利用して，コンピュータと外部との時間のずれを吸収できるわけです．

[キューのデータ操作方式]

キューへのデータの書き込みと読み出しは，互いに非同期に行われます．すなわち，書き込みのタイミングと読み出しのタイミングは，互いに独立です．

・キーボード・バッファとしてのキュー

キーボードからの入力は，いったんキーボード・バッファに格納されます．従って，コンピュータが他の処理（例えばディスクアクセスなど）を行っていたとしても，キーボードからの入力は抜け落ちることなく受け付けられます．適当な時に（キーボードからの入力とは独立に），キーボード・バッファからのデータが取り出しを行えます．

・プリンタ・バッファしてのキュー

ユーザが，プリンタに対して印刷を行うと，普通，印刷データは，いったんプリンタ・バッファに格納されるため，見かけ上，印刷の処理はすぐに終わったかのように見えます．一方，プリンタ印字速度の速さにあわせて，プリンタ・バッファからデータが取り出され，印刷が行われます．プリンタは，プリンタ・バッファが一杯にならない限り，複数の印刷データを受け取ることも可能です．

[キューの構造]

キューの実装では，キューを管理するためのいくつかのデータ（ポインタやフラグなど）を考えること．キューの実現法はいろいろと考えられるが，以下に，実現法の一例をまとめている．

・キューのデータ領域

今回作成するキューは，データ領域の大きさを２５６バイトとして実装すること．すなわち，１つのキューの中には同時に２５６個のバイトデータを格納できるように実装すること．

・キューデータ領域の先頭のメモリアドレス（BF_START）

ここには，キューのデータ領域の先頭メモリアドレスを格納する.

・キューデータ領域の末尾のメモリアドレス（BF_END）

ここには，キューのデータ領域の末尾メモリアドレスを格納する.

キューのデータ領域は，BF_START から BF_END までである.

・キューに書き込むべきデータアドレスを管理するポインタ（PUT_PTR）

PUT_PTR には，キューのデータ領域のうち書き込むべきメモリアドレスを格納する. つまり，PUT_PTR を使って，キューのデータ領域への書き込みを行う. プログラムは次の通り.

```
movea.l PUT_PTR,%A0
move.b %D0,(%A0)
```

なお，書き込みが終わったあとは，PUT_PTR に１足す必要がある.

・キューから読み出すべきデータアドレスを管理するポインタ（GET_PTR）

GET_PTR は，キューのデータ領域から読み出すべきアドレスを覚えているポインタである. つまり，キューのデータ領域からの読み出しは，次のようなプログラムで行う.

```
movea.l GET_PTR,%A0
move.b (%A0),%D0
```

読み出しが終わったあとは，GET_PTR に１足す必要がある.

書き込み時には PUT_PTR を１足し，読み出し時には GET_PTR を１足すが，PUT_PTR, GET_PTRが，データ領域の最後BF_ENDまで来たら，BF_STARTに戻すようにすること. 例えば，キューにデータを書き込むと，最初は BF_START に書き込まれ，書き込むべきアドレス PUT_PTR は１ずつ増えていき，いつかは，最後の BF_END に到達する. ここでの考え方は，「PUT_PTR が BF_END へ到達する頃には、BF_START 付近のデータはすでに読み出し済みであり，再度，BF_START 付近を使っても(BF_START 付近のデータを新しいデータで上書きして消しても）よいだろう」というものである.

つまり，PUT_PTR が最後の BF_END まで到達したら，次は１足すのでなく，最初の BF_START に戻す. このことは，GET_PTR についても同様である.

データ領域の大きさは 256 バイトあるから，書き込み済みでまだ読み出されていないデータを最大 256 バイトまで格納することができる. すでに，256 バイト格納されていてこれ以上のデータを書き込めないときには「書き込み禁止」を，データ領域が空になっていて読み出すべきデータが無いときには「読み出し禁止」を行うことが必要である. 例えば，キューへの書き込みだけを 256 回連続して行うと，キューのデータ領域が一杯になり，PUT_PTR が，GET_PTR に追い付く. このような場合には，次回の書き込みを禁止せねばな

らない．一方，キューからの読み出しだけを連続して行うと，やがてはキューのデータ領域が空になり，GET_PTR が，PUT_PTR に追い付く．このような場合には，次回の読み出しを禁止せねばならない．つまり，キューのデータ領域が一杯にならない限り書き込みが行え，キューのデータ領域が空にならない限り読み出しを行える．

キューの書き込み禁止，読み出し禁止を行う方法は，いくつか考えることができる．最も簡単な方法は，キューのデータ領域が現在空なのかそうでないのか（読み出し用），また一杯なのかそうでないのか（書き込み用）のフラグを別途設けることである．以下，それぞれを，読み出し許可フラグ（GET_FLG），書き込み許可フラグ（PUT_FLG)と呼ぶことにする．

・読み出し許可フラグ（GET_FLG)

これは，「読み出し禁止」か「読み出し許可」かを表すフラグである．キューのデータ領域が空になった場合，GET_FLG の値を「読み出し禁止」に設定する．キューの読み出しの実行前には，必ずこのフラグを調べ，「読み出し禁止」になっているときには，読み出しは行わない．

GET_FLG の値	意味
0x00	読み出し禁止（buffer empty）
0xFF	読み出し許可

・書き込み許可フラグ（PUT_FLG)

これは，「書き込み禁止」か「書き込み許可」かを表すフラグである．キューのデータ領域が一杯になった場合，PUT_FLG の値を「書き込み禁止」に設定する．キューへの書き込みの実行前には，必ずこのフラグを調べ，「書き込み禁止」になっているときには，書き込みは行わない．

PUT_FLG の値	意味
0x00	書込み禁止（buffer full）
0xFF	書き込み許可

[キューの実現]

キューのプログラムは，「キューの初期化」，「キューへの書き込み PUT_BUF」，「キューからの読み出し GET_BUF」という３つのサブルーチンから構成される．以下に各サブルーチンの機能をまとめる．

・キューの初期化

キューの使用前に，初期化が必要である．キューの初期化は，普通，始めに１回だけ行う．

キューの初期化では，PUT_PTR，GET_PTR，PUT_FLG，GET_FLG に適当な値を設定する．

１．PUT_PTR

PUT_PTR にキューのデータ領域の先頭アドレス BF_START を格納し，キューへの書き込み PUT_BUF が最初に呼び出されたときには，キューのデータ領域の先頭にデータが格納されるようにする.

２．GET_PTR

GET_PTR にキューのデータ領域の先頭アドレス BF_START を格納し，キューからの読み出し GET_BUF が最初に呼び出されたときには，キューのデータ領域の先頭からデータを取り出すようにする.

３．PUT_FLG

PUT_FLG へ 0xＦＦを格納し，書き込み許可に設定する．最初に PUT_BUF が呼び出されるときに，キューへの書き込みが出来るようにする.

４．GET_FLG

GET_FLG へ 0xＯＯを格納し，読み出し禁止に設定する（最初キューは空だから）．最初に GET_BUF が呼び出されるときに，キューへの読み出しが出来ないようにする.

・キューへのデータ書き込み PUT_BUF

１．PUT_FLG のチェック

PUT_FLG が，書き込み許可か，書き込み禁止かを調べ，書き込み禁止なら何もしない．サブルーチン呼び出し側へ，すでに書き込み禁止になっていることを知らせること.

２．PUT_PTR でポイントされる場所へのデータ書き込み

PUT_FLG が書き込み許可の場合には，PUT_PTR を使ってデータを書き込む.

３．PUT_PTR の更新

書き込みが終わったら， PUT_PTR を更新して，次の書き込みに備えること．いま書き込んだ場所（PUT_PTR）が，キューのデータ領域の最後(BF_END)であるかどうかを調べ，BF_END ならば，PUT_PTR を，キューのデータ領域の先頭(BF_START)に設定する.
そうでなければ，単に，PUT_PTR に１足す.

４．PUT_FLG の設定

キューが一杯になったならば，PUT_FLG を書き込み禁止に設定する必要がある．
PUT_PTR，GET_PTR を比較し，一致していれば，キューが一杯になったことを意味する．
PUT_FLG を書き込み禁止に設定する．

５．GET_FLG の設定（以後の読み出しの許可）
もはや，キューは空でないから，GET_FLG を読み出し許可に設定する．

・キューからのデータ読み出し GET_BUF

基本的な考え方は，「キューへのデータ書き込み PUT_BUF」と同じである．

１．GET_FLG のチェック
GET_FLG が読み出し許可か，読み出し禁止かを調べ，読み出し禁止ならば，何もしない．サブルーチン呼び出し側へ，すでに読み出し禁止になっていることを知らせること．

２．GET_PTR を使ったデータ読み出し
GET_FLG が読み出し許可の場合には，GET_PTR を使ってデータを書き込む．

３．GET_PTR の更新
読み出しが終わったら，GET_PTR を更新して，次の読み出しに備えること．
いま読み出した場所（GET_PTR）が，キューのデータ領域の最後(BF_END)であるかどうかを調べ，BF_END ならば，GET_PTR を，キューのデータ領域の先頭(BF_START)に設定する．そうでなければ，単に，GET_PTR に１足す．

４．GET_FLG の設定
キューが空になったならば，GET_FLG を読み出し禁止に設定する必要がある．GET_PTR，PUT_PTR の比較を行い，もし一致していればキューが空になったことを意味する(つまり，GET_PTR が，PUT_PTR に追い付いた）．GET_FLG を読み出し禁止に設定する．

５．PUT_FLG の設定（以後の書き込みの許可）
もはや，キューは一杯でないから，PUT_FLG を書き込み許可に設定する．

[キューによる QueueIn および QueueOut の実装]

今まで説明してきたサブルーチン GET_BUF,PUT_BUF を使って，「レジスタの待避機能」を付け加えた，新たなサブルーチンQueueInおよびQueueOutを実装すること．サブルーチン QueueIn および QueueOut は，以下の入力と出力を持つように実装せよ．QueueIn, QueueOutには，データのアドレス p という入力がある．これら入力については，

・サブルーチンQueueIn, QueueOut を呼び出す側で，データのアドレス p をアドレスレジスタ A0 にセットして，呼び出す．

と仮定してプログラムを作成せよ（サブルーチンに入力を渡す方式としては，システムスタックを介して入力を渡す場合と，レジスタを介して入力を渡す場合の２つが考えられる．

もちろん，システムスタックを介して入力を渡す方式の方が，使いやすいサブルーチンを作成できる．今回は，課題を簡単にするために，あえてレジスタを介して入力を渡す方式とした．皆に同じ機能を持ったQueueIn, QueueOut を作成してもらいたいので，レジスタを介して入力を渡す方式を守ってもらいたい ）。

また，キューが一杯の時にQueueInを呼び出した場合や，キューが空の時にQueueOutを呼び出した場合は，キュー操作が失敗したことを呼び出し側のプログラムに通知しなければならない．そこで，

・サブルーチンQueueInおよびQueueOutでは，キュー操作が失敗した場合はデータレジスタ D0 に 0 を，成功した場合は 0 以外の値（例えば１）をセットする。

ことにより，呼び出し側のプログラムに対してキュー操作の成功／失敗を知らせるようにせよ．

【 QueueIn(p) 】

・機能：キューに p 番地のデータを入れる．

・処理内容：

(1)サブルーチン内部で使用するレジスタ群の現在の値をシステムスタックに退避．

(2)キューへの書き込み PUT__BUF の処理を実行し， p 番地のデータを格納する．

(3) サブルーチン内部で使用したレジスタ群の元の値をシステムスタックから取り出す．

(4) D0 を 「成功（０以外の値)」または 「失敗（０)」に設定

【 QueueOut(p) 】

・機能：キューからデータを一つ取り出し，p 番地に copy．

・処理内容：

(1)サブルーチン内部で使用するレジスタ群の現在の値をシステムスタックに退避．

(2)キューからの読み出し GET__BUF の処理を実行し，p 番地にデータを格納する．

(3) サブルーチン内部で使用したレジスタ群の元の値をシステムスタックから取り出す.

(4) D0 を 「成功（０以外の値)」または 「失敗（０)」に設定

ヒント

(1)キューをダイナミックに管理するために，キューのデータ領域及びキューを管理するためのデータは，DS と DC などの擬似命令で構成すべきである．つまり，GET_PTR, PUT_PTR,GET_FLG, PUT_FLG, GET_FLG, PUT_FLG などのキューを管理するためのデータは，DC または DS 命令を用いてメモリ上に，一種の変数として確保せよ．もし１つのキューを扱うのであれば，例えば，次のように宣言できる（宣言の方法は１通りではない）。

```
.equ B_SIZE 256
BF_START:
        ds.b B_SIZE-1
BF_END:
        ds.b 1
PUT_PTR:
        ds.l 1
GET_PTR:
        ds.l 1
PUT_FLG:
        ds.b 1
GET_FLG:
        ds.b 1
```

(2)ラベルを用いて，プログラムを分かりやすくすること.
プログラム中には，アドレス等を直接数値で書くのではなく，代わりにラベルを用いると分かりやすい．ラベル名は，なるべく意味を持った名前をつけるようにせよ．ラベルは文頭におく．ラベルの前にスペースやタブを書いているとうまくいかないことが多い.

(3)「オペランドに制限がある」ことに注意せよ
例えば，「cmp (a2)+,d2」は動くが，「cmp d2,(a2)+」は動かない．アセンブラ命令には，各々指定可能なオペランドの種類に制限がある．例えば，cmp 命令には，ディスティネーションにデータレジスタしか書けないという制限がある．詳しくは資料を参考にせよ.

(4)サブルーチン内で使用するレジスタの退避／回復は，システムスタックに対する push

およびpopを行えばよい．push は move レジスタ，(SP)+で，pop は move -(SP),レジスタで行なうことができる（６８０００では， push, pop というアセンブラ命令はなく，moveを用いる）。

(5)複数のレジスタの退避／回復には，MOVEM命令が便利である．
複数のレジスタは，レジスタリストという形式で1つのオペランドに書くことができる．例えば，D1,D2,D3,A0,A1,A2,A3という7つのレジスタを表現したい時は，D1-D3/A0-A3と書けば，これで1つのオペランドになる．

(6)サブルーチンから呼び出し側のプログラムに戻る時は，RTS命令を使うとよい．

(7)コメントにおけるif文(条件分岐)を実現するには，CMP,CMPIなどの比較命令と，Bcc関係の条件つきブランチ命令を組合わせる．飛び先のアドレスは，条件つきブランチ命令のオペランドに飛び先のラベルを書けばよい．

(8)設問6について，キューの個数を複数（４つ）に拡張するためには，宣言に工夫が必要である．複数のキューを扱うとき，上記のヒント(１)の宣言ではうまくいかない．もし，ヒント（１）の宣言をそのまま使うことにすると，４つのキューが必要なので，４つのキューの各々に上記と同様の宣言を繰り返すとともに，初期化，読み出し，書き込みのサブルーチンも４つ作らねばならない（プログラムの作成の手間が大きい）．
　複数のキューをうまく扱えるようにする方法を考えることが，設問6の目的である．例えば次のようなことが考えられる．

１．ディスプレースメント付きアドレスレジスタ間接モードの利用．
　　各キューへのポインタを簡潔に記述できるようになる．つまり，何番のキューのどの要素を操作したいかが決まれば，各キューの先頭アドレスに適切な数字を足すことで，操作したい要素のアドレスを求めることができる．これは，C 言語でいう構造体だと考えることができる．
２．４つのキューの各先頭アドレスを EQU でシンボル化（プログラムの先頭部分で宣言）
３．キューの各要素のオフセットを EQU でシンボル化（プログラムの先頭部分で宣言）

また，設問6を行う場合，QueueInとQueueOutの仕様は，以下のように変更せよ．
【 QueueIn(no,p) 】
・機能：キュー #no に p 番地のデータを入れる．
・キュー#noはレジスタd1を，データのアドレスpはレジスタa0を介して渡す．
・処理内容：

(1)サブルーチン内部で使用するレジスタ群の現在の値をシステムスタックに退避.

(2) キュー#no に対して，キューへの書き込み PUT__BUF の処理を実行し， p 番地のデータを格納する.

(3) サブルーチン内部で使用したレジスタ群の元の値をシステムスタックから取り出す.

(4) D0 を 「成功（0 以外の値)」または 「失敗（0)」に設定

【 QueueOut(no,p) 】

・機能：キュー #no からデータを一つ取り出し，p 番地に copy.

・キュー#no はレジスタ d1 を，データのアドレス p はレジスタ a0 を介して渡す.

・処理内容：

(1)サブルーチン内部で使用するレジスタ群の現在の値をシステムスタックに退避.

(2) キュー #no に対して，キューからの読み出し GET__BUF の処理を実行し， p 番地にデータを格納する.

(3) サブルーチン内部で使用したレジスタ群の元の値をシステムスタックから取り出す.

(4) D0 を 「成功（0 以外の値)」または 「失敗（0)」に設定