電気系学生実験（C課程）

ソフトウェア実験

[簡易OSの開発]

目次

第1章　はじめに

1.1　本実験の目的

1.1.1　主目的

本実験では，簡易オペレーティングシステム (Operating System, 以下 OS と略) の作成を通して，

- 計算機ハードウェアの基本的な動作と，それを制御するソフトウェアを理解し，
- アセンブリ言語によるプログラミングに習熟する

ことを目的とする．

簡易 OS と呼んでいることからもわかるように，本実験で作成する OS の機能は，一般の OS(Windows, Linux, FreeBSD など) のそれに比べ，相当限定されたものになっている．しかしそれでも，送受信処理，キュー制御，割り込み処理，走行レベル設定といった様々な処理が必要となる．またこれらの処理は，数値演算やソートなどのアルゴリズムのプログラムとはずいぶん異なったものになる．さらに，本実験では作成した簡易 OS を CPU ボードの上において動作させる．すなわち，エミュレータではなく，CPU を含めた本物のハードウェアを直接制御することになる．

従って，この簡易 OS の作成には，これまでのプログラム演習の課題とはずいぶん違った知識やセンスが必要となる．最初は戸惑うかも知れない．試行錯誤も必要であろう．それでも，順を追って実験を進めて行くうちに計算機の中心部分を直接操作する感覚に，徐々に喜びを見い出せるようになると信じている．さらに，この実験を終えたあと，OS の仕事とは何か，また計算機がなぜ同時に複数の仕事をできる（ように見える）のか，といった重要事項を実感できると思う．(本実験の意義については，1.4 節も参照せよ．)

1.1.2　もう一つの目的

この実験には,「複数の人間で 1 つのプログラムをつくる」という隠れた目的がある．諸君の多くが就職するであろうソフトウェア会社ではこうした状況が当り前のように見られる．いかに協力して効率的に作業を進められるかは，各班の力量次第である.「黙っていれば誰かがやってくれるだろう」という思想はあらゆる意味で最悪である．また,「自分の担当さえやれば，あとは知らない」という思想も危険である．班になっているのだから，互いに相談，報告，協力すべきである[1)]．

プログラムの得意な者，不得意な者はいるだろう．しかし,「得意な者が全部一人でやってしまい，不得意な者は任せきり」という状況は，なんとも安易で情けない．得意な者は，(会社に入って部下を持った時を想定して) 不得意な者に作業内容を適切に指示してやるべきである．不得意な者は，(会社に入って上司を持った時を想定して) 得意な者に質問するべきである．この指示や質問の際，内容を相手に理解してもらうべく努力することは非常に重要である．例えば，ただ漠然と「わからない」 と質問するのではなく，何がどのようにわからないかを整理した上で質問するべきである．こうしたコミュニケーションの円滑化に関する努力は，今後至る場所で役に立つはずである．なお，全員がわからなければ，教員やティーチングアシスタント (TA) に相談すればよい．

[1)] 経験的には，会話の少ない班ほど再実験 (1.2 節参照) の可能性が高い．

1.2 単位取得条件

本実験の単位 (注: 実際には，ハード実験の単位も総合する) を取得するためには以下の 3 つの条件を共に満たす必要がある．

条件 1　割り込み，システムコール，送受信方式の理解

条件 2　班全員で 1 つの「簡易 OS」，およびその簡易 OS が提供する機能を利用した「エコーバックプログラム」を完成

- 第 3 章で述べる Step 9 までのプログラムが完全に動くことに相当

条件 3　**各自**で選択課題 1 つを完成

- 14 の選択肢が用意されている．
- **ただし，同じ班の者が同じ選択課題に取り組んではならない．**(例外もある．詳しくは 3.14 節参照のこと．)
- 取り組む課題の難易度，およびプログラムのエレガントさなどにより適宜加点する．また，課題をうまく発展させた場合についても加点する．

以上の完成を確認するために，最終試問を行う．その際に提出するレポートなど，最終試問の詳細は 3.15 節を参照のこと．

簡易 OS およびエコーバックプログラムが完成していない場合，その班は**再実験**となる． 条件 2 はクリアしているが，条件 3 をクリアしていない班員がいる場合，その班員だけが再実験の対象となる．1 週間程度の再実験期間内で終らなかった場合は再履修となる．なお，出席状況や実験への取り組み態度によっては，再実験を行わずに当初より再履修扱いとする．

1.3 本テキストの構成について

本テキストは，本編 3 章と付録 4 章で構成されている．本編第 2 章では，本実験の概要について述べる．また，本実験で必要となる基本的な概念もここで説明される．第 3 章では，簡易 OS 作成の作業手順について述べる．従って，**まず第 2 章を通読した後，第 3 章を読みながら実際に実験を進めていくことを勧める**．

付録は A，B，C，D の 4 章からなる．付録 A では主に，ノート PC で作成したプログラムを CPU ボード上で実行するための手順と，アセンブリ言語に関するヒントが述べられている．よくある間違いについても触れているので，うまく動かなくて困った際には参照することを勧める．付録 B では，本 CPU ボードのハードウェア的な詳細 (具体的には，CPU の走行モード，割り込み処理，各内部レジスタの詳細) が述べられている．例えば，割込みマスクレジスタの設定を変えて特定の割込みを許可したい場合や，タイマ 1 コントロールレジスタを 0x0004 に設定する理由を明らかにしたい場合には，本付録を参照することになる．付録 C では，送受信チャンネルを 2 チャンネル化するための方法についての説明が述べられている．付録 D では，作業日報の書き方について述べられている．作業日報は毎回の実験終了時に各班で 1 枚作成して提出してもらうので，本付録を参考にして記入する．

1.4 補足: 本実験の意義

本節では本実験を行うことにどういった意義があるかについて，次の 3 つの観点から補足しておく．実験前に目を通しておくと，意欲が湧く「かも」知れない．

- アセンブラを使って一から簡易 OS を作り上げること，そのものの意義
- 「システムコール」や「ハードウェア割り込みルーチン」など多くのサブルーチンを準備する意義
- 割り込み処理を作成して，簡易 OS に組み込むことの意義

1.4.1　アセンブラで簡易 OS を作成することの意義

前述のように，本実験で最終的にできあがるのは「簡易 OS」とその簡易 OS の機能を使った「エコーバックプログラム」である．このエコーバックプログラムとは，キーボードへ入力したものをディスプレイへ出力するという，なんとも単純なものである． C 言語を知っている諸君の中には,「なんだ，わざわざアセンブラを使って簡易 OS を作成しなくても，scanf と printf を使えばよいではないか.」と思う者がいるかも知れない.

注意してほしいのは，そうした「scanf」や「printf」は，**誰かが作成してくれた**プログラムルーチンという点である．こうしたプログラムルーチンの集合はライブラリと呼ばれる．そしてそのライブラリも，さらに別の誰かが作成してくれたプログラムルーチンを呼び出して使っている．これら最も基本となるプログラムルーチンの集合がカーネル，すなわち OS と呼ばれるものである． 以上から,「scanf と printf を使えばよい」という考え方は，確かに簡単ではあるが他力本願的であると言える．

今回の実験では，諸君は裸のハードウェアだけを渡されて，その上で動作するプログラムとして簡易 OS を作成する．すなわち他力本願をしたくてもできない状況である．頼もしい「scanf」や「printf」は存在しない．むしろ，誰も頼りにせずに，そうした「scanf」や「printf」のような機能を自分自身の手で作成することになる．

諸君の中には,「僕は計算機を使いこなしてるよ，だって C や Fortran でプログラムを組んで自由にソフトウェアを作成できるんだから」と自負している者がいるかも知れない．確かに，ワープロソフトや表計算ソフトといった市販の出来合のソフトを使っている者に比べれば，遥かに計算機を使いこなしていると言えよう．しかし，前述したように，そうした自負も実は OS 会社や天才コンパイラ設計者の手のひらの上だったと言ってよい．これに対して，本実験ではすべて自己責任である．逆に言えば，誰の手も借りずに，自分達だけの力によって計算機ハードウェアを思い通りに使うことができる．なんと素敵なことではないか! [2)] そういう視点に立って，本実験に積極的に取り組み,「計算機を征服」してほしいと願っている．

1.4.2　機能を細分化することの意義

さて，ボタン 1 つで自動的にインスタントラーメンを作ってくれる,「インスタントラーメン専用装置」があったとしよう．もしあなたがインスタントラーメンしか食べないならば，それだけで十分便利に暮らせるだろう．でも，栄養のバランスを考えたり，新しいメニューを思い付くなどして，普通は他の料理も食べたいと思うだろう．そうなると「焼き飯専用装置」や「すき焼き専用装置」を準備しなくてはならない．こうした専用装置を開発する手間は結構大変そうである．おまけに「インスタントラーメン専用装置」にも「すき焼き専用装置」にも，実は同じような「鍋」が内蔵されている．なんと無駄なことであろう! この無駄のため，たちまち台所はいっぱいになってしまう．

今度は，計算機システムを考えてみよう．計算機には沢山の種類の仕事があるのは周知の通りである．この沢山の計算機の仕事の 1 つ 1 つについて，そのための専用装置をゼロから設計・開発していては，恐ろしいほどの費用と時間がかかるだろう．また似た仕事について同じような専用装置を度々作成しなくてはならないのは何とも無駄な話である．

---

2) 「プラモデルを買ってきて，あとは接着剤で組み立てるだけ」よりも,「部品から設計して，自分だけのオリジナルのプラモデルを作る」ほうが面白そうだと思いませんか?

ところで台所には一般に「鍋」と「フライパン」と「包丁」が準備されている程度で,「○○専用装置」はあまり無い．インスタントラーメンを作りたければ「鍋」を使う．焼き飯なら「フライパン」と「包丁」を組み合わせて使う．すき焼きならば「鍋」と「包丁」を使う．このように，単純な機能しかないこれら 3 つの道具を組み合わせて使うと，組み合わせる手間はあるものの，実に様々な料理ができる．新しい料理を思いついても大丈夫だし，無駄が無いので場所をとらない．

もう説明は不要と思うが，計算機システムにおいても同じことが言える．すなわち,「○○専用ソフトウェア」をゼロから作って準備しておくべきではない．それよりも，細分化された基本的かつ汎用的な機能をあらかじめいくつか準備しておいて，必要な機能がほしくなったときにそれらを組み合わせて用いて適切なソフトウェアを作るプロセスのほうが，ずっと効率的であり，発展性がある．実際，世の計算機システムは，まさにこの方針に従っている．

OS は一般にこれら「細分化された基本的かつ汎用的な機能」がぎっしり詰まった道具箱である．ワープロを始めとする多くの応用プログラムは，この OS が提供する基本的機能を存分に活用している．本実験で作成する簡易 OS も，ぎっしりではないが，幾つかの基本的な機能を提供するオリジナルな道具箱である (具体的には「受信機能」「送信機能」「タイマ機能」を提供する)．こうした道具さえできてしまっていれば，それらを組み合わせて利用するのは至って簡単である．実際，諸君はこの簡易 OS 完成後，基本機能を組み合わせてエコーバックプログラムを作成するのだが，その容易さに驚くだろう．また選択課題でも，基本機能を別の形で組み合わせて何らかの応用プログラムを作成するが，基本機能さえ使いこなしていれば，プラモデルを作るように気楽な作業となるあろう．

これら OS の基本機能 (を呼び出して使うこと) は**システムコール**[3] と呼ばれる．本実験を通して理解して頂きたい重要事項である． なお OS の機能は，ユーザが直接利用できるシステムコールだけではない．よりハードウェア側にあって，ユーザからは見えない機能もある．本実験においても登場する**ハードウェア割り込みルーチン**がこの例である．システムコールもハードウェア割り込みルーチンも OS の重要な機能であり，本実験では両者共に作成する．

ところで細分化とは言わばサブルーチン化のことである．従って，細分化すると，どのサブルーチンを呼び出すかの選択処理やサブルーチンに引数を渡す処理などが増えるために，プログラム量も若干増える．呼ぶ側と受け側でレジスタを一致させるなど，神経を使うところも増える．これは細分化された機能を「組み合わせる手間」が生じたためであり，プログラムが若干複雑になる理由の 1 つである．しかし，一度きちんとサブルーチンを作成しておけば何度でも再利用できることを考えると，十分許される程度の手間と言ってよい．

1.4.3 割り込み処理の意義

OS は様々な仕事を同時並行的にこなす必要がある．逆に OS が並行処理できないとなると，極端な場合，計算機がデータ受信状況チェッカになってしまう[4]．郵便受けまで行ったり来たり，手紙が来たかどうかをチェックするばかりで一向に本来の仕事をしない人のようなもので，これでは全く使い物にならない．そこで，割り込みの概念が発明された．割り込みとは，今の作業より重要な作業が入ると，今の作業をその重要な作業が終了するまで中断するという仕組である．メールの例では，郵便受けにメールが来た時にベルが鳴るようにしておけば，鳴るまでは安心して別の仕事ができる．ここではこのベルが割り込みに相当する．このように割り込みを使うことで，OS は擬似的に並行処理を実現できる．

割り込みを使うにはそれ相当の準備が必要であり，その結果，プログラム量は多少増える．たとえば，どんな割り込みが来たらどういう処理をせよといった指定が必要になる (実際には割り込みベクタを操作すること

[3] システムコールなる用語は微妙である．OS の提供する「機能 (サービス)」のことをシステムコールと呼ぶ場合もあれば，その機能を「呼び出す行為」をシステムコールと呼ぶ場合もあるようである．本テキストでも両者を特に使い分けないので，注意頂きたい．

[4] あるマシンに高速ネットワークが接続された状況を考えてみよう．ネットワークの高速性を発揮させるためには，そのマシンの CPU はネットワークの伝送速度で，常にネットワークからのデータの有無をチェックなくてはならない． もし少々遅い CPU だったら... CPU はまさにデータ受信状況チェッカになってしまい，本来の計算の仕事ができなくなってしまう!

になる)．しかし，割り込みは前述のように OS を有効に動作させるための非常に重要な仕組である．本実験は，それが理解できるように構成されている．

第2章　本実験の概要

本章では, これから作成する簡易オペレーティングシステム (OS) の概要について述べる. プログラムを効率良く作成するには, プログラムの全体像 (どのようなプログラムを作成するのか) を予め把握しておくことが重要である. ここでは, 具体的なプログラムの組み方やハードウェアの制御方法の詳細には触れずに, 実験の概略のみを説明する[1)]. 学生の諸君は, 実験を開始する前にこの章を通読し, 簡易 OS の概要を把握しておくことをお勧めする. さらに詳細な内容に関しては, 次章以降を参照してもらいたい. (キーワードとなる事柄に関して, 巻末に索引を用意しているのでそれを利用するとよい.)

2.1　簡易 OS

本実験では, 簡易 OS, つまり, いくつかの限定された機能のみを提供するオペレーティングシステム (システムコールライブラリやハードウェア割り込み処理ルーチン) を作成する. 本節では, 簡易 OS(カーネル) におけるシステムコール, ハードウェア割り込みの役割を概説する.

簡易 OS の機能を用いた 68000CPU ボード[2)] での入出力処理の例 (エコーバックプログラム) を図 2.1 に示す. これは, キーボードから入力された文字をディスプレイ上に表示させる通信プログラムであり, 諸君らが本実験の Step9 で作成するものに相当する.

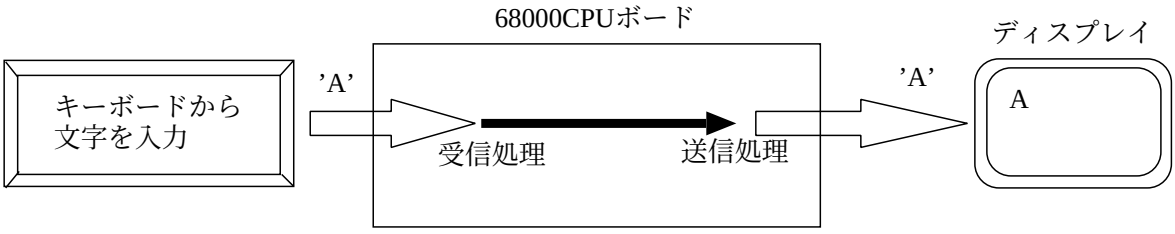


図 2.1: 簡易 OS に動作するユーザプログラムの例：エコーバックプログラム

一般に, ハードウェアの監視, メモリ管理, 割り込み処理などの OS の基本的機能を提供しているソフトウェアをカーネルと呼ぶ. カーネルが提供している機能のうち, 特に, ユーザープログラム上で利用可能なもの (または, それを呼び出すこと) を**システムコール**と呼ぶ. 通常, システムの重要な機能はカーネルが管理するよう設計されているため, ユーザープログラムからそれらの機能を直接利用することはできない. したがって, ユーザープログラムは, **カーネルに仕事を依頼する**という手順をとることにより, 間接的にカーネルの機能の一部を利用している. 一方, カーネルの機能のうち, 内蔵デバイス (ハードウェア) からの割り込み要求に対処するため

[1)] イラストなどを利用して概略のみの説明を行う. そのため, 表現の正確さという意味で, やや厳密さに欠ける部分があるかもしれない. しかし, 実験の目的や概略 (簡易 OS の概要や割り込み処理の手順など) をあらかじめ把握しておくことは, 限られた時間内で実験を終了させるには有効である. 本テキストの第 2 章は実験の大まかなイメージをつかんでもらうために設けた章であり, 足りない部分は付録などを参照して必要に応じて補足してもらいたい.

[2)] 正確には, MC68VZ328 (“$DragonBall^{TM}VZ$”).

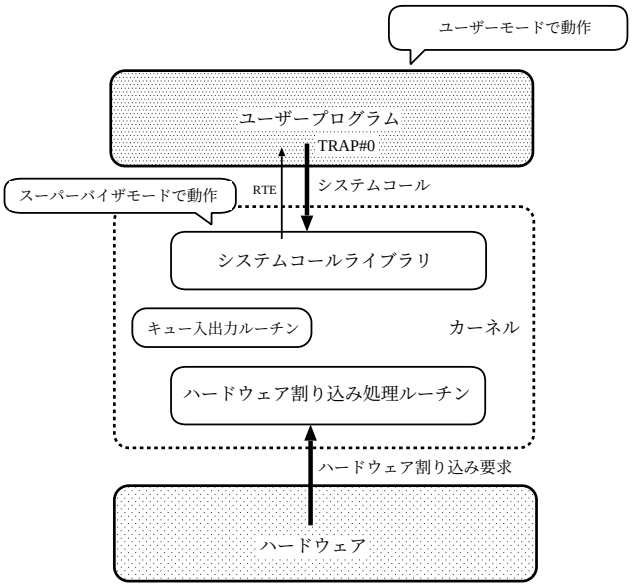


図 2.2: 簡易 OS の概略図

に用意されたものを**ハードウェア割り込み処理**と呼ぶ. これはユーザープログラムでの処理とは関係なく, ハードウェアからの割り込み要求により実行される. カーネル、システムコール、ハードウェア割り込みの対応関係 (簡易 OS の概略) を図 2.2 にまとめる. ハードウェア割り込みおよびシステムコールの詳細については, 2.2 節, 2.3 節において説明する. ここで, ユーザープログラムが実行されているときの CPU の走行モードを**「ユーザーモード」**, カーネルが提供する処理ルーチンが実行されているときの走行モードを**「スーパーバイザモード」**と呼ぶ. CPU の走行モードについての詳細を知りたい場合は**付録 B.1** を参照するとよい.

2.2　68000 におけるハードウェア割り込み機能

2.2.1　ハードウェア割り込み

割り込み処理とは**現在実行中の処理を中断して, より優先度 (走行レベル) の高い別の処理を実行する**ことである (図 2.3). 送受信機やタイマなどハードウェアからの要求により発生する割り込みをハードウェア割り込みと呼び, この機能を用いることでハードウェアの効率的な制御が可能となる. 例えば, キーボードを使って文字を入力する場合を考えてみる. 割り込みの概念がない場合, 計算機はいつ行われるかわからないキーボードからの入力を待ち続けなければならない. 一方, 割り込みの考えを用いる場合は, キーボードから入力があったときのみ, 現在の作業を一時中断して受信作業を行うことができ, 作業効率が大幅に改善される. また, 優先度の高い処理を直ちに実行できる点も割り込み処理の利点である.

本実験では, 「送受信 (UART1) 割り込み」と「タイマ割り込み」の 2 種類の割り込みを利用する. これらの割り込みが発生するタイミングは以下の通りである.

1 送受信割り込み

→「データが送信可能」もしくは「データを受信している」という状態にある限り繰り返し発生する. 図 2.1 の例においては, キーボードからデータが受信された場合, およびディスプレイにデータを送信する場合に利用する.

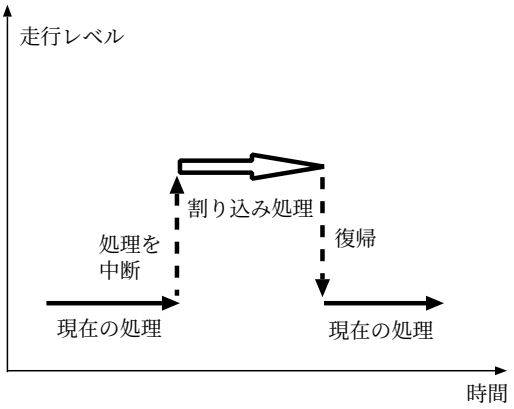


図 2.3: 割り込み処理

2 タイマ割り込み

→タイマ内のカウンタが予め設定していた値に達した場合に発生する.

なお，送信割り込みと受信割り込みは本来独立であるが，本 CPU ボードにおいてはそうした区別は無く,「送受信割り込み」として一体化している．しかしながら，割り込みに応じて何らかの処理を行うためには，どちらの割り込みが起こっているのかを判断する必要がある．本実験では，送受信割り込みが起こった際，ハードウェアの状態を示すレジスタ (デバイスレジスタ) を点検することにより両者を区別する．(この区別する部分は，ハードウエア割り込みインタフェースと呼ばれるサブルーチンに実装される．→**3.8.2 節，3.10.3 節**). 送受信割り込み要求に対する処理に関しては, 本章の **2.6 節** (特に, **2.6.1 節**) において説明しているので, そちらを参照してほしい.

本実験で作成したハードウェア割り込みの詳細については 2.5 節「簡易 OS プログラムの作成」において説明する．68000 の割り込み処理に関してさらに詳細な内容を知りたい場合は, **付録 B.2** 「MC68VZ328 の割り込み処理機能」を参照するとよい. 特に, **付録 B.2.2** 「走行レベルと割り込みレベル」および **B.2.2.3** 「多重割り込みの処理」は重要な項目であるので, 実験を開始する前に一度目を通しておくことをお勧めする.

2.2.2 割り込みベクタ

68000 では, **割り込みの種類に応じて特定の割り込み処理 (例外処理) を呼び出す**ために, **割り込みベクタ**と呼ばれる方式を用いている．割り込み要求が発生したときの割り込み処理ルーチンの呼び出しの過程を図 2.4 にまとめる．割り込みには, 種類ごとに番号 (**割り込みベクタ番号**) が割り当てられている．図 2.4 に示すように, メモリ上の割り込みベクタと呼ばれる領域に “割り込みベクタ番号” と “対応する割り込み処理ルーチンの開始アドレス” との対応表が保存されている[3)] . 割り込みが発生すると, 割り込みベクタ番号に対応するメモリアドレスを参照し, その開始アドレスを取得することで, 割り込み処理を実行する.

割り込みベクタに関して, さらに詳細な内容を知りたい場合は**付録 B.2.4** 「割り込みベクタの設定」を参照するとよい. 特に, **表 B.4**(割り込みベクタの表) には一度目を通しておいてほしい.

[3)] したがって, メモリ上の所定のアドレスにあらかじめ割り込み処理の開始アドレスを書き込んでおく必要がある.

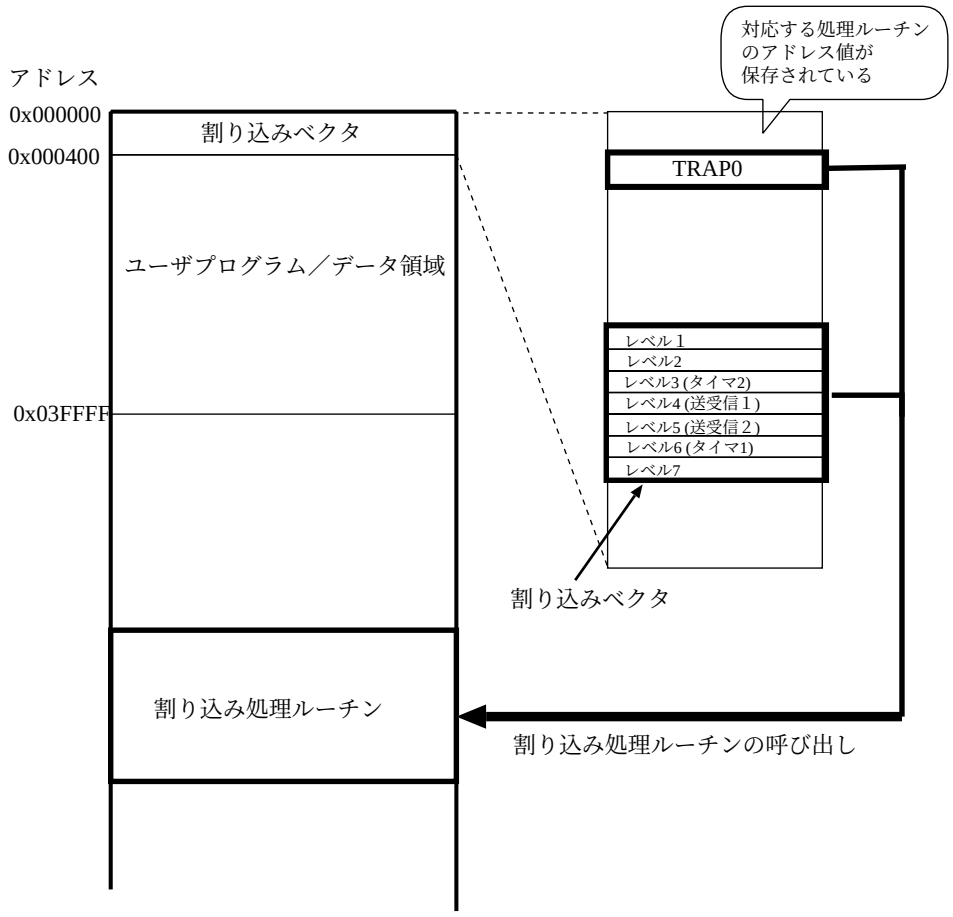


図 2.4: 割り込みベクタによる割り込み処理ルーチンの呼び出し

2.3　システムコール

本実験では, **TRAP 命令**[4] により意図的に割り込み処理を発生させることでシステムコールを実現する. TRAP 命令を用いてシステムコールを呼び出す手順を図 2.5 にまとめる. TRAP#0 命令を用いると TRAP#0 に対応する割り込みベクタを参照し, 割り込み処理ルーチンを起動する. このとき, CPU の走行モードはユーザーモードからスーパーバイザモードへ切り替えられている.

本実験では, 図 2.5 に示すように, **システムコールインターフェース**を一旦経由して, 該当するシステムコールを呼び出していることに注意してほしい. システムコールインターフェースの目的は, システムコール番号 (引数) に対応したシステムコール処理ルーチンを起動することである. これにより, TRAP#0 を実行するのみで, 複数のシステムコールを呼び出すことが可能になる.

図 2.5 におけるシステムコールインターフェース部の詳細を知りたい場合は, **3.12 節**を参照するとよい.

[4] トラップ命令の機能は, 状態をユーザーモードからスーパーバイザモードへ切り替え, 所定の処理ルーチンを起動することである. トラップ命令の番号は＃ 0 から＃ 15 まであるが, ここでは＃ 0 を用いる. この番号に対応する割り込みベクタに割り込み処理 (システムコール用ルーチン) の開始アドレスを保存しておく.

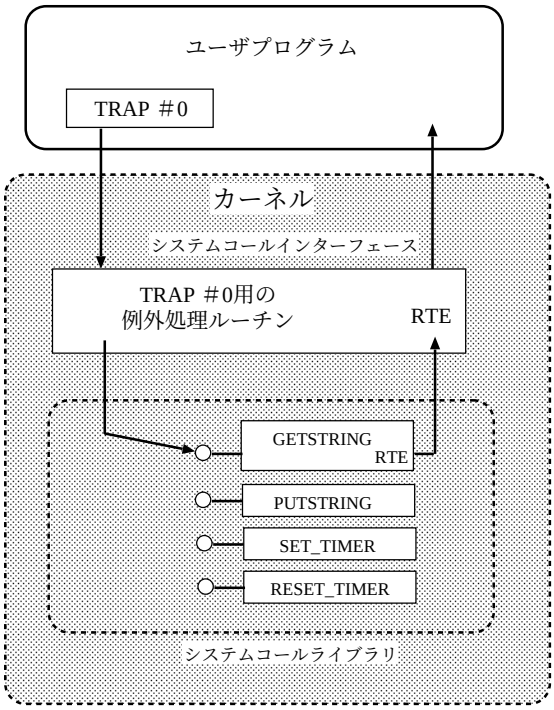


図 2.5: TRAP#0 を用いたシステムコールの概略

2.4 キューの仕組みと役割

本節では, キューの仕組みと役割についてまとめている. キューのプログラム作成の手順を知りたい場合は, **3.4 節「キューの作成 [Step 0]」**を参照するとよい.

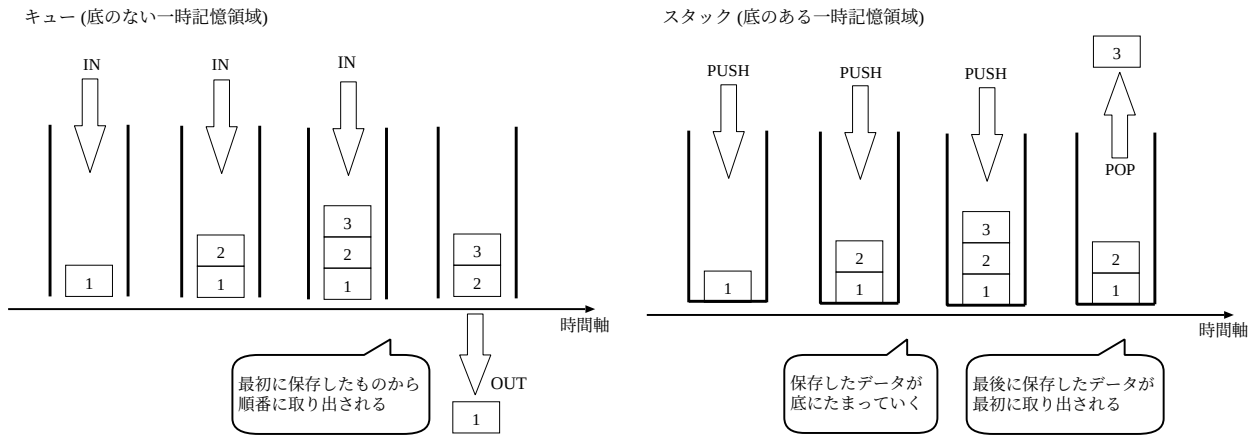
2.4.1 キューとスタックの違い

キューとはデータを一時的に保存しておく領域 (バッファ) のことであり, 図 2.6(a) に示すように, データを到着順に保存し到着順に取り出す仕組みになっている[5)]. スタックとの違いは, 図 2.6(b) に示すように, スタックでは最後に入力されたものが最初に取り出される構造になっていることである[6)].

2.4.2 キューの役割

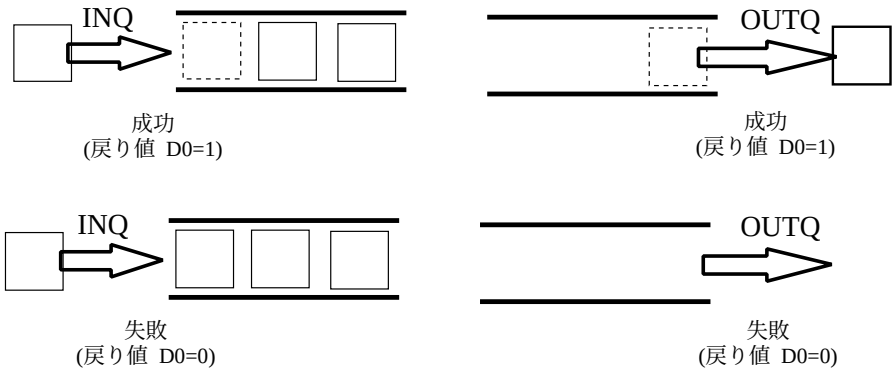
キューは計算機本体の処理と外部 (割り込み) 処理との間の処理速度の差を緩和するバッファとしての役割をもつ. 本実験では, キーボードからの入力データに対するバッファ(受信キュー), およびディスプレイへデータを出力する際のバッファ(送信キュー) として用いる. キューをバッファとして設置しておくことで, メインプログラムはキューとの間でデータの送受信を行うのみでよく, これらの処理はハードウェア割り込みとは独立に適当な時刻に行うことができる.

[5)] この意味で, キューは先入れ先出し (FIFO: First In First Out) と呼ばれる.
[6)] この意味で, スタックは先入れ後出し (FILO: First In Last Out) と呼ばれる.



(a) キュー　　(b) スタック

図 2.6: キューおよびスタックの構造



(a) INQ (入力)　　(b) OUTQ (出力)

図 2.7: キューへの入出力処理

2.4.3　INQ, OUTQ

キューに対してデータの入出力を行うには, 入出力用のプログラム (サブルーチン) が必要となる. 本実験では, INQ(キューへデータを入力するプログラム), OUTQ(キューからデータを出力するプログラム) の 2 つを用意する. キューへのデータの入出力が成功したかどうかは, INQ, OUTQ の戻り値 (内部レジスタ $D0$) を見ることで判断できる[7)]. ここで, キューの管理はカーネルが行っているため, INQ, OUTQ はカーネル内部でのみ用いられるサブルーチンであることに注意してほしい. キューに対する入出力処理の過程を図 2.7 にまとめている.

[7)] 内部レジスタ $D0$ を戻り値とし, これが 1 の場合が成功, 0 の場合が失敗である. キュー内部の構造およびその他の詳細に関しては **3.4 節**を参照のこと.

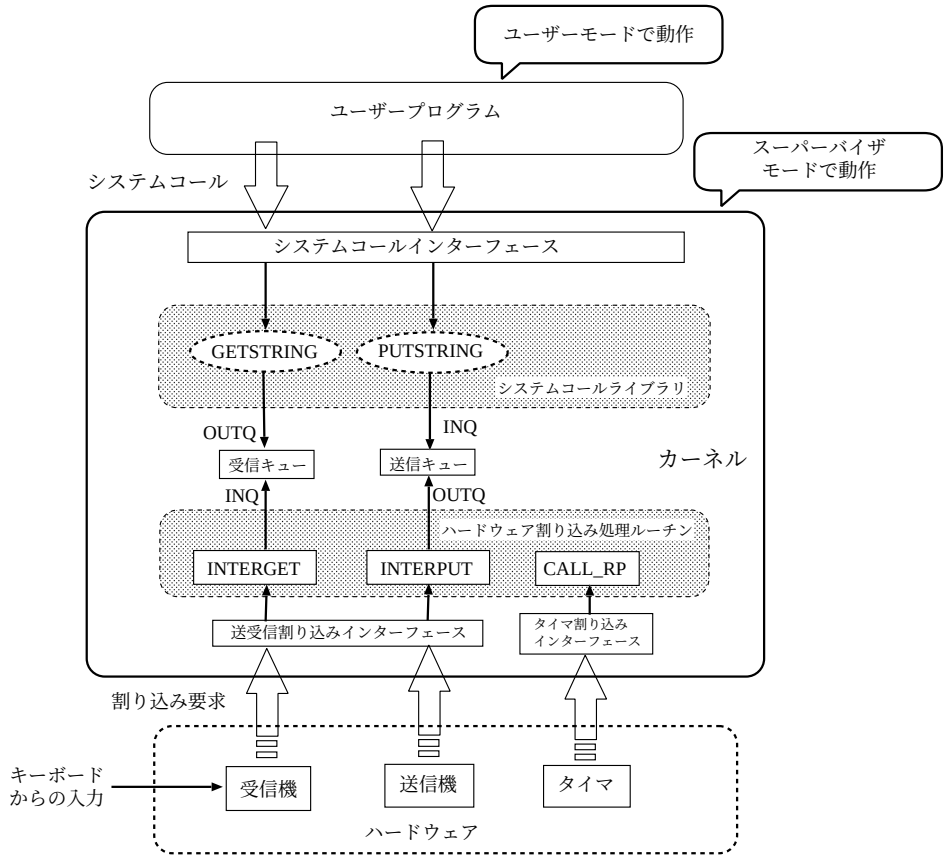


図 2.8: エコーバックプログラムの流れ

2.5 簡易 OS プログラムの作成

本実験で作成するハードウェア割り込み処理ルーチンとシステムコールの一覧を以下に示す．また, これらの簡易 OS の機能を利用したエコーバックプログラムの概略を図 2.8 に示す[8)]．なお, 各デバイスからの割り込みを有効にするには, **割り込みマスクレジスタ (IMR)** において割り込みを許可する必要がある．また, 各デバイスからの割り込みを発生させるには, **各デバイスのコントロールレジスタ**において割り込みを有効にする必要がある．これらデバイスレジスタの設定方法は, **付録 B.3.4** に記載されているので, 必要に応じて参照してほしい．

ハードウェア割り込み処理ルーチン

- INTERGET (受信割り込み処理)：キーボードから入力された文字を受信するための処理
- INTERPUT (送信割り込み処理)：ディスプレイへ文字を送信するための処理
- CALL_RP (タイマ割り込み処理)：特定の時刻に実行される処理

システムコール

- PUTSTRING：送信キューにデータを渡す。

8) 諸君が 3 章の実験手順 Step9 で作成するプログラムがこれに相当する．Step10 では, 簡易 OS の機能を利用した選択課題プログラムを作成する．

- GETSTRING : 受信キューからデータを取り出す。
- RESET_TIMER : タイマのリセット
- SET_TIMER : タイマのセット (タイマを動かす)

エコーバックプログラム (図 2.8) における処理手順

1 キーボードからデータが入力されると受信割り込みが発生し, 受信割り込み処理ルーチン **INTERGET** が呼び出される. INTERGET は入力されたデータを受信キューに移動させる.

2 受信キューに蓄えられたデータは, システムコール **GETSTRING** を呼び出すことでメインプログラム上に読み出される (GETSTRING は受信キューをチェックし, キューにデータが格納されている場合, そのデータをメインプログラム上に読み込む役割をもつ).

3 システムコール **PUTSTRING** を呼び出し, 送信すべきデータを送信キューに転送する. (PUTSTRING はデータを送信キューに転送する役割をもつ).

4 送信キューに保存されたデータは, 送信割り込み処理ルーチン **INTERPUT** によりディスプレイ (出力端末装置) に出力される[9] .

5 3, 4 の処理とは別に, タイマ割り込みを利用することで, 決められた時刻に特定の処理を実行させることができる.

(例: 一定時刻毎に文字をモニタ上に表示させるなど.)

2.6　送受信制御 (UART) 部における処理

本節では, 送受信制御部 (UART[10]) における処理について概略を述べる. なお, プログラムの作成方法などの詳細については, 下記を参照してもらいたい. また, 送信制御部におけるデバイスレジスタの設定方法を確認したい場合は**付録 B.3.4** を参照するとよい.

- 受信制御部：→ 3.10 節
- 送信制御部: → 3.8 節, 3.9 節
- 受信レジスタ (URX1): → 付録 B.3.3.3
- 送信レジスタ (UTX1): → 付録 B.3.3.4

2.6.1　割り込みの発生時の処理

本実験で使用する 68000CPU ボードでは, 受信割り込みと送信割り込みの割り込みベクタ番号が同一である. そのため, **割り込みベクタ番号をもとに送信割り込みと受信割り込みを区別することができない.** したがって, 割り込みが発生した場合には, それが送信と受信のどちらの割り込みであるのかを調べる必要がある. これは, 受信レジスタ (URX1), 送信レジスタ (UTX1) の内容をチェックすることで区別できる (URX1 の第 13 ビット目 (DATA READY ビット) および UTX1 の第 15 ビット目 (FIFO EMPTY ビット) が 0 であるか 1 であるか

[9] 正確には, PUTSTRING において送信割り込みを許可することで, INTERPUT が実行される. 詳しくは, 2.6.3 節において述べる.
[10] Universal Asynchronous Receiver/Transmitter の略.

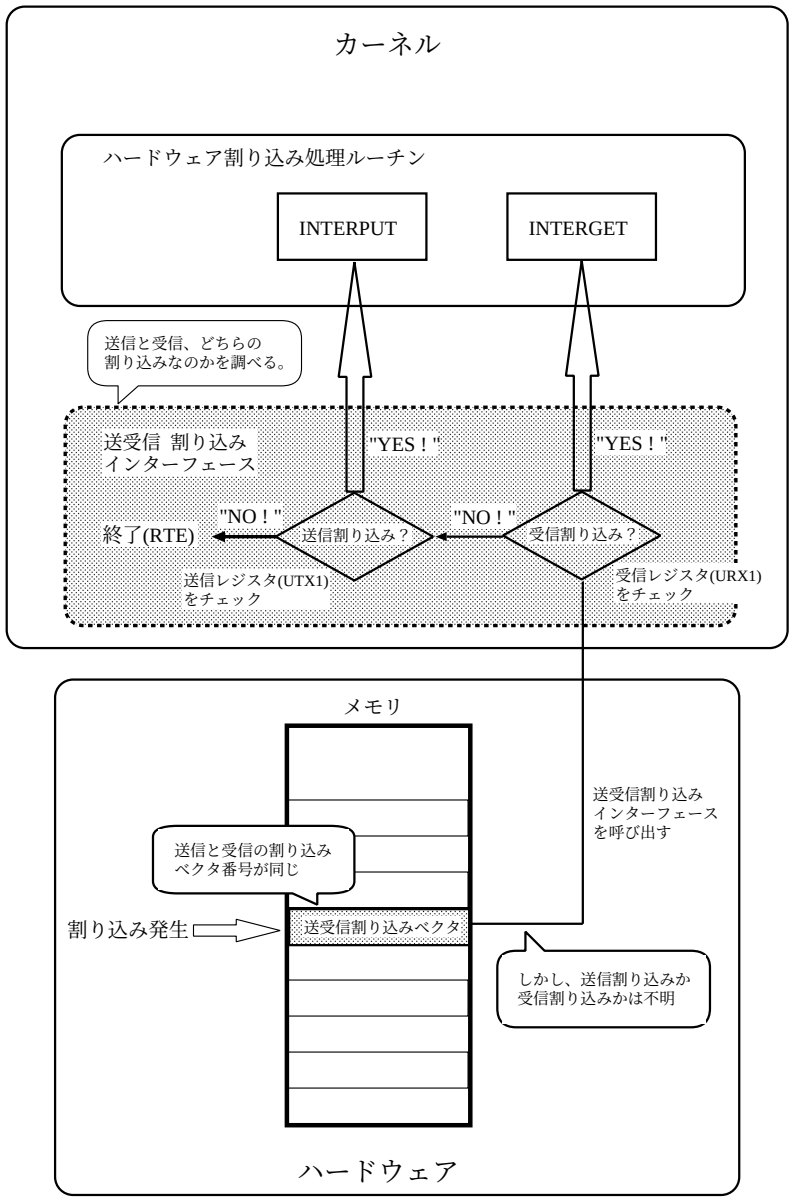


図 2.9: 送受信割り込み発生時の処理

を調べる). 送受信割り込みインターフェース作成の詳細については **3.10.3 節**, **3.8.2 節**を, レジスタの詳細については**付録 B** のレジスタリストを参照のこと.

送信または受信割り込み発生時の処理の手順を図 2.9 に示す. 発生した割り込みが送信であるか受信であるかを調べるためのインターフェースプログラムを用意し, その開始アドレスを割り込みベクタに保存しておく. 割り込みが発生すると, このインターフェースプログラムが呼び出され, 受信割り込みの場合は INTERGET(受信割り込みルーチン) を, 送信割り込みの場合は INTERPUT(送信割り込みルーチン) を実行する.

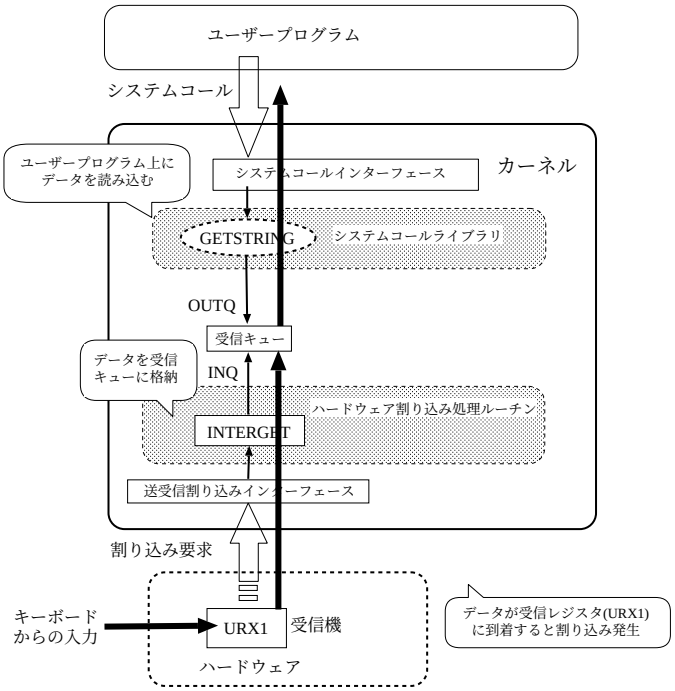


図 2.10: 受信制御部

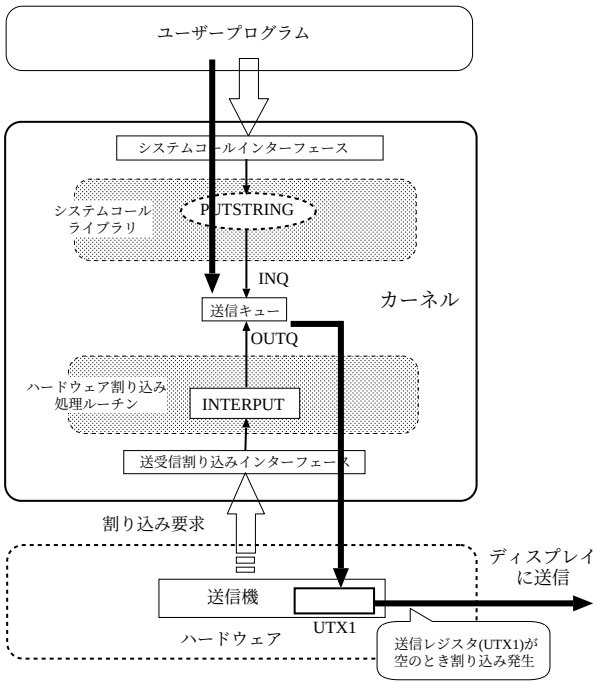


図 2.11: 送信制御部

2.6.2　受信制御部の概略

受信制御部の概要を図 2.10 に示す．キーボードから入力された文字 (データ) は受信レジスタ (URX1) に保存される．**受信割り込みは受信レジスタにデータが格納されている場合に発生する．**したがって, 受信レジスタ

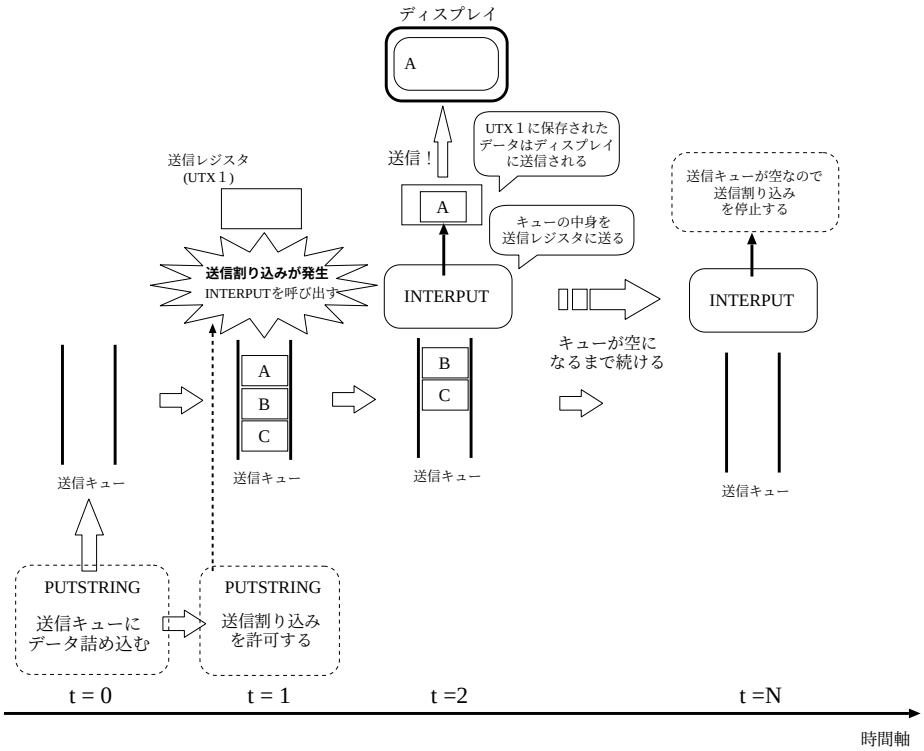


図 2.12: 送信処理の流れ

URX1 にデータが到着するのと同時に, 受信割り込みにより **INTERGET** が呼び出される. このとき, データは受信レジスタの第 0-第 7 ビット (RX DATA) に格納されている. INTERGET は, 受信レジスタからデータを取り出し, 受信キューに保存する (INQ を行う). このとき, 受信レジスタが空になるため, 受信割り込みは停止する. システムコール **GETSTRING** は, 受信キューに保存されているデータをメインプログラム内に読み出す (OUTQ する) 役割をもつ.

2.6.3 送信制御部の概略

送信制御部の概略を図 2.11 に示す. また, 送信時の処理手順を図 2.12 に示す. メインプログラム内でシステムコール PUTSTRING を呼び出すことで, ディスプレイにデータを出力することができる. PUTSTRING は送信キューにデータを保存する役割をもつ. PUTSTRING では, **キューが満杯になるか, または指定されたサイズのデータを全てキューに保存した時点で送信割り込みを許可する**[11] .

送信割り込み処理 **INTERPUT** は送信キューからデータを取り出し, 送信レジスタ (UTX1) に保存する役割をもつ (データを送信レジスタの第 0-第 7 ビット (TX DATA) に保存する). 送信レジスタに保存されたデータは自動的にディスプレイに送信される. **送信割り込みは意図的に割り込みを停止させない限り, 常に発生し続ける**[12] . そのため, キューが空になったら, INTERPUT 内で送信割り込みを停止させる.

[11] PUTSTRING が呼び出された時点では, 送信割り込みは停止されている. 送信割り込みの許可 (または停止) は, USTCNT レジスタにて設定される. USTCNT レジスタの詳細は**付録 B** を参照のこと.
[12] 送信割り込みは送信レジスタにデータがない場合に発生するため.

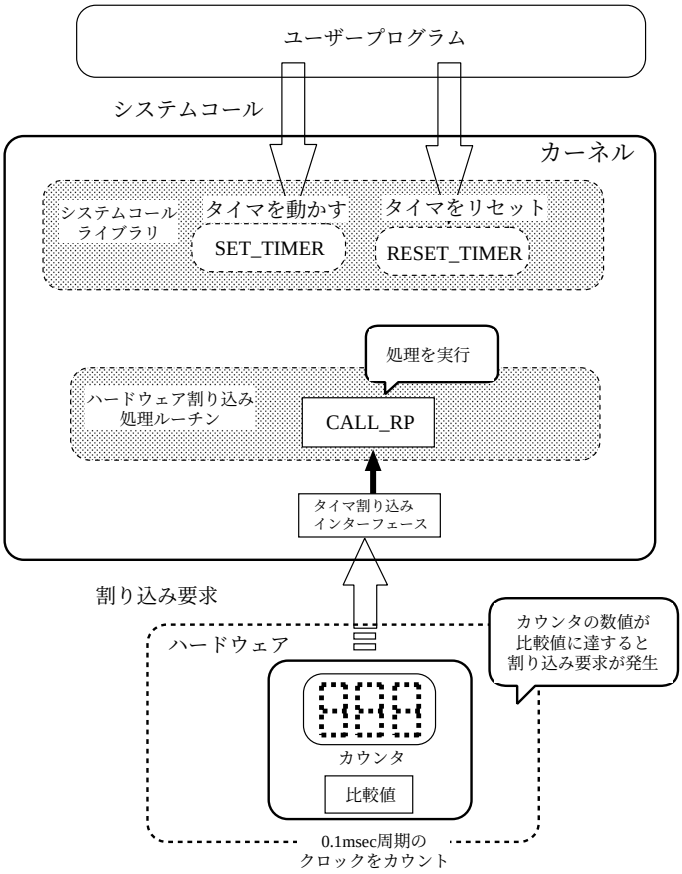


図 2.13: タイマ

2.7　タイマの役割

本実験に用いる CPU ボードは, 独立に設定可能なタイマを 2 つ備えている[13]．タイマとは, その名前の通り時間を計測する装置であり, ある一定時刻に達するとタイマ割り込みを発生させることができる．この機能を利用することで, 一定時間毎に特定の処理を実行させることができる．タイマによるハードウェア割り込みの手順を図 2.13 にまとめる．

タイマは, 内部カウンタ[14] と比較値保存用レジスタ[15] からなる．カウンタは基準信号 (0.1msec 周期の信号[16]) のクロックをカウントし, カウンタの数値が比較値と一致すると, カウンタがリセットされ, 割り込み (カウント一致) 信号が出力される．これによりタイマ割り込みが発生する[17]．比較値を適切な値に設定することにより, タイマ割り込みが発生するタイミングを調節することができる．タイマ割り込み処理ルーチンはタイマ割り込みインターフェースを経由して呼び出されることに注意してもらいたい．

タイマ割り込みインターフェースの詳細およびタイマ作成の手順については, **3.11 節**を参照するとよい．

---

[13] 本実験で使用するタイマは 1 つのみである．
[14] 内部レジスタ TCMP1 のこと
[15] 内部レジスタ TCN1 のこと
[16] 本実験では, システムクロック (33MHz) の 1/16 の周波数 (約 2MHz) の信号を 1/207 に分周することで, 0.1msec 周期の基準信号を生成している．分周比はプリスケーラレジスタ (TPRER1) において設定される．TPRER の詳細は**付録 B** に記載されている．
[17] あらかじめ, タイマ 1 制御レジスタ (TCTL1) において, タイマ割り込みを許可しておく必要がある．

第3章　プログラムの作成手順

本実験では，小さなプログラムから始めて，動作チェックをしながら順次プログラムを書き加えていくという段階的な手順で，簡易 OS を完成させる．そして，最後に，その簡易 OS が提供する機能を使って，エコーバックプログラムを作成する．以下，本章では標準的な作成手順を説明する．

3.1　作成するサブルーチン一覧

具体的な作業に入る前に，本実験で作成する各サブルーチンの大まかな機能と使われ方を再掲しておく (前章も参照のこと).

1. **システムコール**
   ユーザプログラムにおいてユーザが「意図的」に呼び出して使うサブルーチンであり，OS が提供する「道具」のうち，ユーザが直接利用できるものである．送信機能，受信機能，タイマ機能に細分化されている．市販の OS はこうしたシステムコールを数多く提供している．なお，いずれのシステムコールも 3. で後述するシステムコールインタフェースを一旦経由して呼び出される．
   - PUTSTRING : 文字列の送信 → 3.9.1 節
   - GETSTRING : 文字列の受信 → 3.10.1 節
   - RESET_TIMER : タイマのリセット → 3.11.1 節
   - SET_TIMER: タイマ動作開始 → 3.11.1 節

2. **ハードウエア割り込み処理ルーチン**
   ハードウエアが発生する割り込みに応じて言わば「無意識的」に呼び出されるルーチン． ただし，割り込みによって直ちに呼び出されるのではなく，3. で後述するハードウエア割り込みインタフェースを一度経由して呼び出される．
   - INTERPUT : 送信割り込み時の処理 → 3.8.1 節
   - INTERGET : 受信割り込み時の処理 → 3.10.2 節
   - CALL_RP : タイマ割り込み時の処理 → 3.11.1 節

3. **インタフェース関係**
   前述の諸処理を呼び出す前に通過するルーチン．引数の受け渡しや，呼び出すサブルーチンの選択等が行われる．
   - システムコールインタフェース → 3.12.2 節
   - 送受信用ハードウェア割り込みインタフェース → 3.8.2 節，3.10.3 節
   - タイマ用ハードウェア割り込みインタフェース → 3.11.2 節

4. **初期化ルーチン**
   メモリやスタック領域の確保や，タイマデバイスや送受信デバイスの設定，割り込みに関する設定を行う．

表 3.1: 作業の流れ

手順	初期化担当	タイマ担当	受信担当	送信担当
1	Step 1(3.5 節)		Step 0(キュー) 作成 (3.4 節)	
2	Step 2(3.6 節)			
3	Step 3(3.7 節)			
4	Step 4(3.8 節)			
	送信割り込み用のハードウェア割り込みインタフェース作成	補助	補助	INTERPUT 作成
5	Step 5(3.9 節)			
	補助	補助	補助	PUTSTRING 作成
6	Step 6(3.10 節)			
	受信割り込み用のハードウェア割り込みインタフェース作成	補助	INTERGET, GETSTRING 作成	補助
7	Step 7(3.11 節)			
	タイマ割り込みに関するハードウエア割り込みインタフェース作成	RESET_TIMER, SET_TIMER, CALL_RP 作成	補助	補助
8	Step 8(3.12 節)			
	システムコールインタフェース作成	補助	補助	補助
9	Step 9 エコーバックプログラムを作成 (3.13 節)			
10	Step 10 選択課題を各自で作成 (3.14 節)			

- キュー初期化ルーチン → 3.4.1 節
- 内蔵デバイスレジスタ初期化ルーチン → 3.5 節 他

5. キュー関係

送受信処理で利用されるキューに関する汎用ルーチン．送受信関係のシステムコールやハードウエア割り込み処理ルーチンから呼び出される．従って，スーパバイザモードで実行される．

- INQ → 3.4.2 節
- OUTQ → 3.4.2 節

3.2 分担および作業の進め方

本実験では，各班で以下のように 4 分担して作業を進める．

- 初期化 (初期化ルーチン，システムコールインタフェース，ハードウェア割り込みインタフェース作成の責任者)
- タイマ (初期化ルーチン，RESET_TIMER, SET_TIMER, CALL_RP 作成の責任者)
- 受信 (キュー，INQ, OUTQ, GETSTRING, INTERGET 作成 の責任者)

- 送信 (キュー，INQ, OUTQ, PUTSTRING, INTERPUT 作成 の責任者)

班の人数は必ずしも 4 人とは限らない．3 人の班の場合は比較的負担の軽いタイマを誰かが重複担当する，5 人の班の場合は初期化を 2 人で担当する，というように臨機応変に割り当てる．

分担を決めた後，表 3.1 に従って作業を進める．具体的な作業内容が書いてある担当が**責任者**である．たとえば Step 5 では送信担当が PUTSTRING 作成の責任者 (取りまとめ役) である．責任者以外の覧には「補助」と書いてあるが，責任者同様に実際的な作業を担当する (「補助」＝「取りまとめではない」という程度の意味)．

Step 4 に入る段階で，キューが完全に動作している必要がある．このため，Step 1–3 の段階では，半分の人員をキュー作成に充て，同時並行して作業を進める．

各担当者は，自分が書いたプログラムソースコードに最低限度のコメントを記入する．また，**レビュー担当者**は，班員が書いたソースコードを確認して，誰がソースコードを読んでも分かるように，より詳しいコメントを追記する．なお，追記したコメントには自分の名前を記載しておくこと（最終試問の際にプログラムのソースコードの提出が求められ，その際にコメントが書かれているかどうか評価される）．レビュー担当の割り当ては，次の通り．

- 初期化担当者は，タイマ担当者のソースコードをレビュー
- タイマ担当者は，初期化担当者のソースコードをレビュー
- 受信担当者は，送信担当者のソースコードをレビュー
- 送信担当者は，受信担当者のソースコードをレビュー

3.3 作業日報

本実験では，各回の作業内容を作業日報として各班で 1 枚提出してもらう．作業日報には，班員各自の担当内容の進捗状況と次回の作業予定，問題点，レビュー報告などを記入する．作業日報の詳細については，付録 D を参照のこと．

3.4　キューの完成 [Step 0]

3.4.1　キューの初期化ルーチンの作成

前章でも述べたように，キューとは循環的に使用できるメモリ領域のことである．キューは表 3.2 の変数を使って制御される．

前章でも述べたように，キューは送信用と受信用にそれぞれ 1 つ準備される．以下では受信キューを第 0 番キューとし，送信キューを第 1 番キューとする．キューの大きさは，送受信いずれも 256 バイト (1 バイト ×256 文字分) とする．

キューの初期化ルーチンでは，以下の処理を送信/受信キューの両方について行う．

- キューのデータ領域，および前述の変数用の領域を確保 (.bss セクションにおいて，ディレクティブ.ds を利用)
- データ領域の先頭アドレスを top に代入
- データ領域の先頭アドレスを out に代入
- データ領域の先頭アドレスを in に代入
- データ領域の末尾アドレスを bottom に代入
- データ数 s を 0 にセット

3.4.2　キューへの入力 (INQ), 出力 (OUTQ) ルーチンの作成

次にキューに対してデータを 1 つ入力するルーチン INQ，ならびにデータを一つ取り出すルーチン OUTQ を作成する．図 3.1 に示すように，データをキューに入れるときは，in が示す番地に入れ in の値を一つ進め，s の値を 1 増やす．データを取り出すときは，out が示す番地からデータを一つ取り出し，out の値を一つ進め，s の値を 1 減らす．in，out の値を一つ進める場合，その値が bottom ならば top に設定する．s=256 のとき INQ は失敗し，s=0 のとき OUTQ は失敗する（復帰値 0 を返す）．

INQ(no, data)

【機能】

- 番号 no のキューにデータを入れる．

【入力】

- キュー番号 no → %D0.L に代入しておく
- 書き込む 8 bit データ data → %D1.B に代入しておく

【戻り値】

表 3.2: キュー制御用の変数

top	キューの先頭の番地 (固定的)
bottom	キューの末尾の番地 (固定的)
out	次に取り出すべきデータのある番地
in	次にデータを入れるべき番地
s	キューに溜っているデータの数

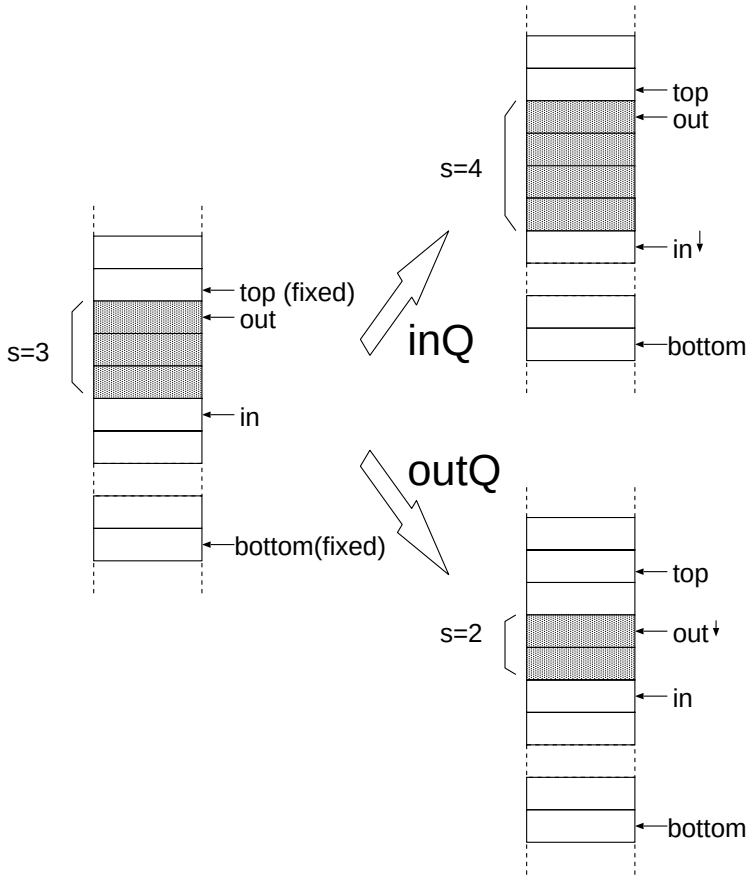


図 3.1: キューの構造ならびに INQ, OUTQ によるデータ入出力

- 失敗 0/ 成功 1 →%D0.L に格納されるようにする

【処理内容】

(1) 現走行レベルの退避

(2) 割り込み禁止（= 走行レベルを 7 に）

(3) s == 256 ならば %D0 を 0（失敗：queue full）に設定し，(7) へ

(4) m[in] = data

(5) if (in == bottom) in=top
else in++

(6) s++, %D0 を 1（成功）に設定

(7) 旧走行レベルの回復

OUTQ(no, data)

【機能】

- 番号 no のキューからデータを一つ取り出す.

【入力】

- キュー番号 no → %D0.L

【戻り値】

- 失敗 0/ 成功 1 →%D0.L

- 取り出した 8 bit データ data →%D1.B

【処理内容】
(1) 現走行レベルの退避
(2) 割り込み禁止（=走行レベルを 7 に）
(3) s == 0 ならば %D0 を 0（失敗：queue empty）に設定し，(7) へ
(4) data = m[out]
(5) if (out == bottom) out=top
　　else out++
(6) s– –, %D0 を 1（成功）に設定
(7) 旧走行レベルの回復

【注意】
- 走行レベルの退避 = ステータスレジスタ (%SR) をスタックに退避 (→ 付録 B.2.2)
- 走行レベルの設定 = ステータスレジスタ (%SR) の値を設定 (→ 付録 B.2.2)

以上の記述において，m[p] は p 番地のデータ（8bit）を示している．また，処理内容の部分の top，bottom，in，out，s は番号 no のキューに対応する変数を示すものとする．また，INQ, OUTQ のいずれにおいても走行レベルを 7 にして割り込み禁止しているが，これはキューが破壊されることを防ぐためである．

3.4.3　キューの正常動作の確認

キューの正常動作を確認しないまま簡易 OS を作成していくと，後で OS 全体の動作が不安定になるなど問題が生じることが多い．(特に，実験の後半になってプログラムがうまく動かなくなることが多い．こんな場合，キューの間違いに気づくまでとても時間がかかってしまう．) そこで，作成したキューの確実な正常動作を**エミュレータ環境** (すなわち，基礎実験 1(アセンブラ演習) と同じ実験環境) で以下の手順により確認しておく．

1. 送信キュー/受信キューのメモリ領域が正常であるかどうか確認する．
   - キューが確保されているメモリアドレスを「∗.LIS ファイル」を見て調べればよい
2. 以下の確認手順 3–6 の準備として，送信キュー/受信キューとは別に，次の 4 つの領域を確保する．
   - INQ で読み込むためのデータの領域：適当なデータを書き込んでおく．アドレスレジスタ %A0 にその先頭アドレスを代入しておく．
   - INQ の戻り値 (0 or 1) を格納する領域：%A1 にその先頭アドレスを代入しておく．
   - OUTQ で読み込んだデータを格納する領域：%A2 にその先頭アドレスを代入しておく．
   - OUTQ の戻り値 (0 or 1) を格納する領域：%A3 にその先頭アドレスを代入しておく．

   (注意) 以上 4 つの領域のサイズは，必ずキューサイズ以上 (300 程度?) にしておくこと．(手順 4 ではキューサイズ (256) 以上のデータを入出力して，キューを溢れさせる実験を行うため．
3. INQ でキューの途中までデータを詰めた後，それらが OUTQ で正しく取り出されることを確認する．
   (a) INQ を 128 回行い，キューにデータを入力し (図 3.2)，以下の項目を確認する
   - キュー内にデータが正常に格納されている．(エミュレータのメモリ内容の表示機能を利用)
   - 戻り値は全て%D0=1（成功）である．

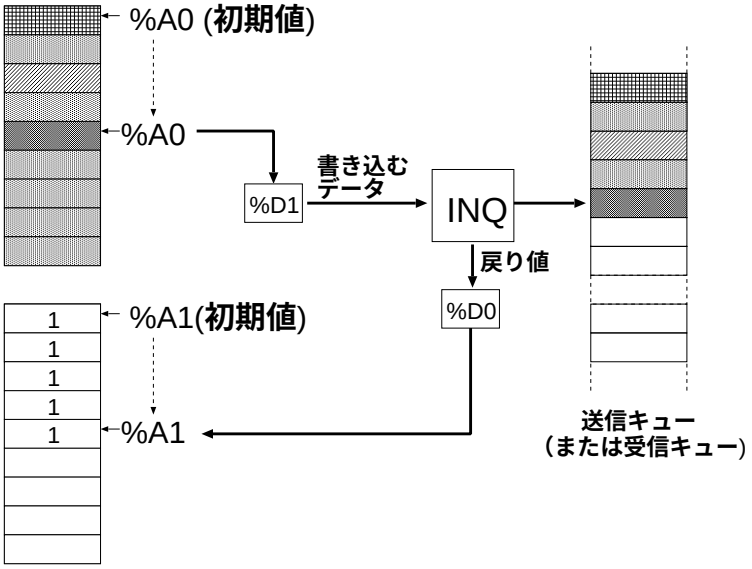


図 3.2: INQ のテスト

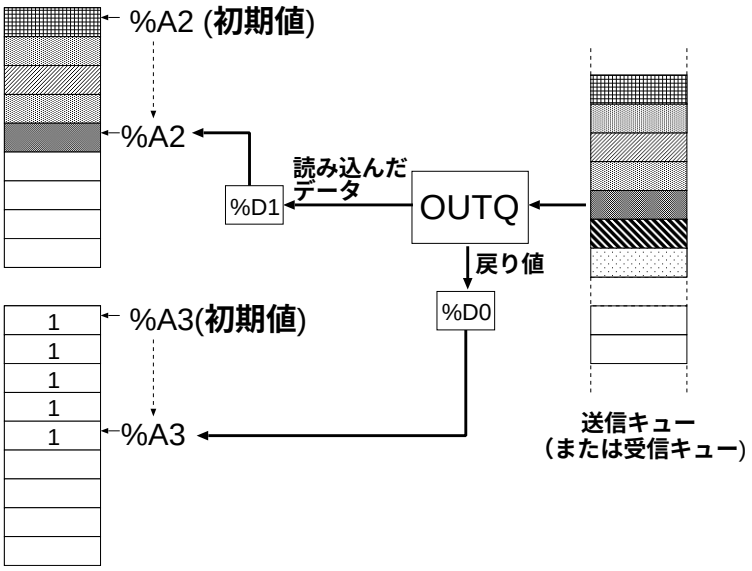


図 3.3: OUTQ のテスト

(注意 1) INQ で書き込むべきデータ data (= %D1.B) は

```
move.b (%A0)+, %D1
```

により与えることとする.

(注意 2) INQ 実行後，その戻り値 (0 or 1) を

```
move.l %D0, (%A1)+
```

により格納しておく．これにより実行後にどういう出力が得られたかがエミュレータにより一括して確認できる.

(b) OUTQ を 129 回行い，キューからデータを取り出し (図 3.3)，以下の項目を確認する

- 128 回目までは入力時と同じ順番でデータが出力される．
- この間，戻り値は全て%D0=1（成功）である．
- 129 回目の OUTQ が失敗し，戻り値は%D0=0 となる．

(注意 1) OUTQ で読み込んだデータ data (= %D1.B) は

```
move.b %D1, (%A2)+
```

としてメモリに格納しおく．

(注意 2) 同時に，OUTQ の戻り値 (0 or 1) についても

```
move.l %D0, (%A3)+
```

により格納しておく．

4. キューが一杯になるまで INQ を行い，その後，OUTQ によるデータ出力が正しく行われることを確認する．

   (a) INQ を 257 回を行い，キューにデータを入力し，以下の項目を確認する．
   - 256 回目まで INQ が成功する．
   - 戻り値は全て%D0=1（成功）である．
   - 257 回目の INQ が失敗し，戻り値が%D0=0(失敗) となる．
   - キューには 256 回目までのデータが保存されている．

   (b) OUTQ を 257 回行い，キュー内のデータを出力し，以下の項目を確認する．
   - 256 回目まで出力される．
   - 戻り値は全て%D0=1（成功）である．
   - 入力時と同じ順番で 256 個のデータが出力される．
   - 257 回目で OUTQ に失敗する．戻り値は%D0=0(失敗) である．

5. 以上第 3 項および第 4 項を両方のキュー (no=0,1) について確認しておく．

6. キューのバッファ領域外へのデータの書込みがないかを確認する．
   - キューのメモリ領域の前後に予め適当なデータを書き込んでおく．
   - 第 3，4，5 項のチェックを再び行い，これら予め書き込んでおいたデータが変更されていないことを確認する．

以上ではキューサイズを 256 として記述している．しかし，動作テストの際に限り，チェックのしやすさを考えて，キューサイズを 10 程度にしても構わない．ただし，動作確認が終了したら，キューサイズを 256 に戻しておくように．(INQ の処理内容 (3) の個所も忘れずに．)

3.5 初期化ルーチンの作成 [Step 1]

初期化ルーチンを作成する．このルーチンでは以下のような処理を行う．

1. 各種レジスタの定義 (各レジスタのアドレスに扱い易い名前を割り当てる)
2. スタック領域の確保
3. 割り込み関係の初期化
4. 送受信 (UART: Universal Asynchronous Receiver/Transmitter) 関係の初期化
5. タイマ関係の初期化

以下が初期化ルーチンのプログラムである．ただし，割り込みベクタの設定など幾つか項目が抜けていたり，キューの初期化ルーチンが無かったり，後に変更される点もある．従って**これで簡易 OS プログラムに必要な初期化がすべて完了しているわけではない**.

```
*****************************************************************
** 各種レジスタ定義
*****************************************************************

***************
** レジスタ群の先頭
***************
.equ REGBASE,   0xFFF000          | DMAP を使用.
.equ IOBASE,    0x00d00000

***************
** 割り込み関係のレジスタ
***************
.equ IVR,       REGBASE+0x300     | 割り込みベクタレジスタ
.equ IMR,       REGBASE+0x304     | 割り込みマスクレジスタ
.equ ISR,       REGBASE+0x30c     | 割り込みステータスレジスタ
.equ IPR,       REGBASE+0x310     | 割り込みペンディングレジスタ

***************
** タイマ関係のレジスタ
***************
.equ TCTL1,     REGBASE+0x600     | タイマ１コントロールレジスタ
.equ TPRER1,    REGBASE+0x602     | タイマ１プリスケーラレジスタ
.equ TCMP1,     REGBASE+0x604     | タイマ１コンペアレジスタ
.equ TCN1,      REGBASE+0x608     | タイマ１カウンタレジスタ
.equ TSTAT1,    REGBASE+0x60a     | タイマ１ステータスレジスタ

***************
** UART1（送受信）関係のレジスタ
***************
.equ USTCNT1,   REGBASE+0x900     | UART1 ステータス/コントロールレジスタ
.equ UBAUD1,    REGBASE+0x902     | UART1 ボーコントロールレジスタ
.equ URX1,      REGBASE+0x904     | UART1 受信レジスタ
.equ UTX1,      REGBASE+0x906     | UART1 送信レジスタ

***************
** LED
***************
.equ LED7,      IOBASE+0x000002f  | ボード搭載の LED 用レジスタ
.equ LED6,      IOBASE+0x000002d  | 使用法については付録 A.4.3.1
.equ LED5,      IOBASE+0x000002b
.equ LED4,      IOBASE+0x0000029
.equ LED3,      IOBASE+0x000003f
.equ LED2,      IOBASE+0x000003d
.equ LED1,      IOBASE+0x000003b
.equ LED0,      IOBASE+0x0000039
```

```
****************************************************************
** スタック領域の確保
****************************************************************
.section .bss
.even
SYS_STK:
        .ds.b   0x4000  | システムスタック領域
        .even
SYS_STK_TOP:            | システムスタック領域の最後尾


****************************************************************
** 初期化
** 内部デバイスレジスタには特定の値が設定されている.
** その理由を知るには，付録 B にある各レジスタの仕様を参照すること.
****************************************************************
.section .text
.even
boot:
        * スーパーバイザ & 各種設定を行っている最中の割込禁止
        move.w #0x2700,%SR
        lea.l  SYS_STK_TOP, %SP | Set SSP

        ****************
        ** 割り込みコントローラの初期化
        ****************
        move.b #0x40, IVR       | ユーザ割り込みベクタ番号を
                                | 0x40+level に設定.
        move.l #0x00ffffff,IMR  | 全割り込みマスク

        ****************
        ** 送受信 (UART1) 関係の初期化 (割り込みレベルは 4 に固定されている)
        ****************
        move.w #0x0000, USTCNT1 | リセット
        move.w #0xe100, USTCNT1 | 送受信可能，パリティなし，1 stop，8 bit，
                                | 送受割り込み禁止
        move.w #0x0126, UBAUD1  | baud rate = 38400 bps
                #0x0038                         230400
        ****************
        ** タイマ関係の初期化 (割り込みレベルは 6 に固定されている)
        ****************
        move.w #0x0004, TCTL1   | restart，割り込み不可，
                                | システムクロックの 1/16 を単位として計時，
                                | タイマ使用停止
        bra MAIN

****************************************************************
% 現段階での初期化ルーチンの正常動作を確認するため，最後に 'a' を
% 送信レジスタ UTX1 に書き込む. 'a' が出力されれば，OK.
****************************************************************
.section .text
.even
MAIN:
        move.w #0x0800+'a',UTX1 | 0x0800 を足す理由については，
                                | 付録参照
LOOP:
        bra LOOP
```

3.6 受信割り込みのテスト [Step 2]

本節以降では，いよいよ割り込み (→2.2.1 節) に対応したプログラムを作成する．ある割り込み A に対応したプログラムでは，一般的に初期化部において

1. 割り込み A 発生の際にジャンプすべきルーチンの先頭アドレスを，割り込み A に対応するベクタに設定する (→ 2.2.2 節，付録 B.2.4)
2. その割り込み A の内部デバイスレジスタ (→ 付録 B.3) を設定して割り込み A を「有効」にする
3. 割り込みマスクレジスタ (IMR → 付録 B.3.1.2) を制御し，割り込み A を「許可」する
4. 走行レベル (→ 付録 B.2.2) を 0 にする．

といった準備が必要となる．

本節では受信割り込みの割り込み処理をテスト的に作成する．この受信割り込み処理を行うために，初期化部において以下のように設定する (それぞれ上述の 4 つの準備に対応)．[1]．

1. 受信割り込みの際にジャンプすべきルーチンの先頭アドレスを，送受信割り込みのベクタ (=UART1 割り込みのベクタ=レベル 4 割り込みベクタ) に設定
2. 受信の割り込みが有効かつ送信の割り込みが無効になるように UART1 ステータス／コントロールレジスタ (USTCNT1→ 付録 B.3.3.1) 設定
3. 送受信割り込みを許可するよう割り込みマスクレジスタ (IMR→ 付録 B.3.1.2) を設定
4. 走行レベルを 0 にする

テストプログラムでは，以上の初期化終了後，ステータスレジスタの設定により，スーパバイザモード (→ 第 2.1 節，付録 B.1.1) のまま，無限ループ (中身のない空ループ) になるようにする．

受信割り込みの際にジャンプするルーチンも準備しなくてはならない．ここでは，そのルーチンを

> 受信レジスタ URX1 にある 受信データを レジスタ %D0 へ move し，それを送信レジスタ UTX1 へ move する (※)

としておく．受信レジスタ URX1 については付録 B.3.3.3 や図 2.10 を参照せよ．送信レジスタ UTX1 については付録 B.3.3.4 や図 2.11 を参照せよ．いずれも本実験において重要な役割を為すレジスタである．

本ハードウエアでは，受信データがあると受信割り込みが発生する (実際には「送受信」割り込み．2.2.1 節，2.6.1 参照．本ページの脚注も参照)．従って，プログラムロード実行後に (ノート PC からの) キー入力の受信があると，上記ルーチンが起動されて画面にエコーバックされることが確認できればよい．

Tips

「※」印の文章は，注意して読んでほしい．URX1 の内容を%D0 経由で「そのまま」UTX1 にコピーするわけではない！move の際，受信データが 8 bit であるのに対し，URX1 と UTX1 はそれぞれ 16 bit であり，上位 8 ビット分のヘッダが必要である点に注意せよ (→ 付録 B.3.3.3,B.3.3.4)．

Tips

以上で「エコーバック専用装置」はできたが，OS ができたわけではない．これからが本番である (→1.4.2 節)．

---

[1] 前章 2.2.1 節でも述べたように，本ハードウエアでは，送信，受信の割り込みベクタは同じであり，従って，それらのどちらが実際に発生したかは，受信レジスタ URX1 と送信レジスタ UTX1 の内容をそれぞれチェックする他ない (Step 4 でこの区別をする部分を作成する)．ただし，USTCNT1 の設定により送信割り込みがマスクできていれば，ここで生じる送受信関係の割り込みは必ず受信割り込みである．

3.7　送信割り込みのテスト [Step 3]

送信割り込みの割り込み処理を

UTX1 に 8bit データ'a' を書き込む

とする．書き込む際にはヘッダ (UTX1 の上位 8 bit) の付与を忘れずに．

テストプログラムでは，前節同様，まず初期化部においてベクタを設定し，さらに送信以外の割り込みがかからないように UART1 ステータス／コントロールレジスタ (USTCNT1→ 付録 B.3.3.1) および割り込みマスクレジスタ (IMR→ 付録 B.3.1.2) を設定しておく．

以上の初期化の後，スーパバイザモードのまま走行レベルを 0 とした後，無限ループ (中身のない空ループ) になるようにする．

本ハードウエアでは，送信可能な状態である限り (＝送信中でない限り) 送信割り込みが発生し続ける (→2.2.1 節，2.6.3 節，付録 B.3.3)．従って，上の処理が何度も読み出されれば（すなわち，文字 “a” が何度も繰り返し出力される），送信割り込みが正常に起こっていると言える．

Tips

そろそろプログラムも複雑になってきて，うまく動かずデバックが必要になる頃でしょう．よくある間違いとデバックの方法については，A.7 節に説明があります．デバッグには A.4.3.1 節の LED を使うという手もあるでしょう．

3.8 送信割り込みルーチンの作成 [Step 4]

Step 3 では送信割り込みのテスト用ルーチンを作ったが，本ステップでは正式な送信割り込み用ルーチン INTERPUT を作成する．なお，本ステップ以降では，キューおよびその制御プログラム INQ, OUTQ を利用するので，Step 0 が完了している必要がある．

3.8.1 送信割り込みルーチン INTERPUT の作成

送信割り込みに関するルーチン INTERPUT の仕様は以下の通り．

INTERPUT(ch)

【機能】

- チャネル ch の送信キューからデータを一つ取り出し，実際に送信する (=UTX1 に書き込む)．
- チャネル ch が 0 以外の場合は，何も実行しない．

【入力】

- チャネル ch → %D1.L

【戻り値】

- なし

【処理内容】

(1) 割り込み禁止（走行レベル を 7 に）．

(2) ch ≠ 0 ならば，何もせずに復帰．

(3) OUTQ(1,data) を実行する (=送信キューから 8bit データを１つ取り出し data に代入)．

(4) OUTQ の戻り値が 0(失敗) ならば，送信割り込みをマスク (USTCNT1 を操作) して復帰．

(5) data を送信レジスタ UTX1 に代入して送信．(上位 8 ビット分のヘッダ付与を忘れずに)

【注意】

- INTERPUT は送信割り込みに応じて呼び出されるルーチンだが，**この送信割り込みと，その禁止 (マスク) および許可 (アンマスク) について，ここでもう一度振り返ってみよう．**
  送信割り込みは，マスクをしない限り，ずっと出続けると言ってよい．送信するデータもないのにずっと出続け，その度に INTERPUT が呼び出されるというのは何とも無駄である．従って，送信すべきデータを準備し終った段階でアンマスクし，送信すべきデータがなくなった (すなわちキューが空になった) 段階でマスクをし，送信割り込みの発生をストップするわけである．
  割り込みを禁止 (マスク) するのは，INTERPUT の (4) のところである．一方，割り込みを許可 (アンマスク) するのは何時か? 3.8.3 節で行う INTERPUT のテストの際は，キューにデータを貯めた直後に 1 回割り込みを許可する．許可されると同時に INTERPUT が呼び出され，キューが空になるまで許可の状態が続くわけである．
- OUTQ はキューから取り出した値を格納する場所 data として，レジスタ %D1.B を用いる点に注意．
- 本 CPU ボードには送受信が 2 チャンネル用意され，UART1，UART2 という名称で区別されている．本実験では UART1 しか利用しないので，チャンネルを表す引数 ch は常に 0 (=UART1 に対応) である．一見無駄であるが，本実験に続く後半のソフト実験との整合性を考慮して，このようにしている[2)]．

[2)] チャンネルを 2 つ使用できるように機能を拡張する場合，このように ch を持たせておく方が，比較的容易にプログラムの書き換えができる．その意味でも，ここでは引数 ch を持つようにプログラムを書いておくことを推奨する．2 チャンネル化については，付録 C に掲載しているので興味があれば参考にされたい．

- 付録 B.2.3 を参照し，使用するレジスタの保存と復元を行うこと．

3.8.2　送信割り込み用のハードウェア割り込みインタフェースの作成

ルーチン INTERPUT は送信割り込みに応じて呼び出され，送信キューのデータを 1 つ実際に送信する．ところが，前述のように，本ハードウエアでは，送受信の割り込みベクタは同じである． 従って，INTERPUT を呼び出す前に，送受信のどちらが実際に発生したかをチェックする処理が必要となる．

そこで，以下の機能を持つハードウェア割り込みインタフェースを作成し，これが UART1 の割り込みによって呼び出されるようにベクタを設定しておく．

送受信割り込み用のハードウェア割り込みインタフェース

【処理内容】

(1) 今起こっている UART1 の割り込みが，送信割り込みであるかを，送信レジスタ UTX1 の第 15 ビット目を用いてチェックする．

(2) 送信割り込みであった場合，ch=%D1.L=0 として INTERPUT を呼び出す．

【注意】

- **送信レジスタ UTX1 の内容は，一度読むと (例えば move.w UTX1, %D1 とすると) 更新されてしまう．**従って，現在の UTX1 の内容を何度も参照する場合，一旦別のレジスタにコピーしておくべきである．
- 呼び出し前のルーチンに戻るには，rte を利用する．
- ch=0 としている理由については，前 3.8.1 節【注意】参照のこと．
- なぜ 15 ビット目を見るのかは，→ 付録 B.3.3.4 UTX の節

3.8.3　INTERPUT の動作テスト

初期化モジュールの後 (すべての割り込みをマスクしておく) に，送信キューに a （コード 0x61）を連続 16 文字，b（コード 0x62）を連続 16 文字，$\cdots$ と合計 256 文字書き込み，さらに out を top に，s を 256 に設定するよう記述する．その後，走行レベルを 0 にし，送信割り込みを許可し (USTCNT1 を操作)，無限ループになるようにする．

プログラムロード実行後，キューに入れた文字がすべて出力されることを確認する．このように動作するのはなぜかを考えてみよう．無限ループの直前で送信割り込みが許可される．本ハードウエアでは，送信割り込みは（マスクしていない限り）ほぼ常に発生しているので，直ちに INTERPUT が起動されて，OUTQ から１文字を出す．INTERPUT が終了しても，発生しつづける送信割り込みにより，繰り返し INTERPUT は呼び出される．この繰り返しが止まるのは，INTERPUT 中の処理 (4) で，送信割り込みがマスクされたとき，すなわちキューが空になったときである．要するに，キュー中の文字が全部出続けるまで，INTERPUT は繰り返し呼び出されることになる．

3.9 送信制御部の完成 [Step 5]

3.9.1 PUTSTRING の作成

データを送信キューに格納し，送信割り込みを開始するルーチン PUTSTRING を作成する．

PUTSTRING(ch, p, size)

【機能】

- チャネル ch 用の送信キューに， p 番地から始まる size バイト分のデータを格納する．
- その後，送信割り込みを許可し，戻り値として%D0 に書き込みサイズを代入したあと，走行レベルを元に戻す．
- 復帰値 (%D0) はキューに書き込んだデータのバイト数である．
- 送信キューが一杯になるとそれ以上は書き込まない．つまり，指定サイズ以下の個数で書き込めるだけのデータをキューに書き込む．
- 本実験では，チャネル ch が 0 以外の場合は何も実行しないように実装する．

【入力】

- チャネル ch → %D1.L
- データ読み込み先の先頭アドレス p → %D2.L
- 送信するデータ数 size → %D3.L

【戻り値】

- 実際に送信したデータ数 sz → %D0.L

【処理内容】

(1) ch ≠ 0 ならば，(11) へ．(=なにもせず復帰)

(2) sz ⟵ 0 , i ⟵ p

(3) size = 0 ならば (10) へ

(4) sz = size ならば (9) へ

(5) INQ(1, p[i]) を実行し，送信キューへ i 番地のデータを書き込む．

(6) INQ の復帰値が 0 (失敗 / queue full) なら (9) へ

(7) sz++, i++

(8) (4) へ

(9) USTCNT1 を操作して送信割り込み許可 (アンマスク)

(10) %D0 ⟵ sz

【注意】

- 基本的には 3.8.3 節の処理内容を思い出せばよい．PUTSTRING はキューに文字をためて (処理 (1)-(8))，送信割り込みを許可するだけである (処理 (9))．あとは，キューが空になるまで，ポンプ (INTERPUT) がキューの内容を吸い出してくれるわけである．
- もう一度言おう．PUTSTRING がキュー (タンク) にデータ (水) を貯めて，その後に割り込み処理を許可 (栓を開ける) する．送信割り込み処理は許可されている限り出続けるので，INTERPUT が『繰り返し』呼び出されることになる．INTERPUT の機能により，その度にデータ (水) が 1 つづつ取り出される．キュー (タンク) の中身が空になったことに INTERPUT が気づくと，INTERPUT は割り込みをマスクする (栓をする)．これでこの『繰り返し』が終る．

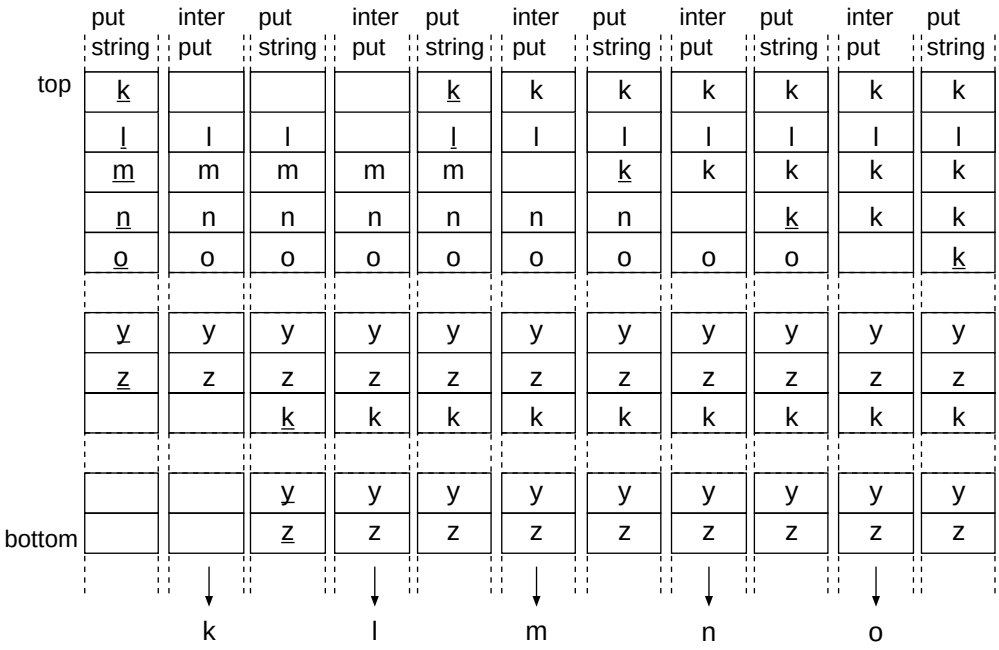


図 3.4: PUTSTING と INTERPUT が交互に実行された場合のキューの様子 (簡単のためキューの長さは 32 としている)

3.9.2　送信制御部の動作テスト

前節で作成した INTERPUT と，本節の PUTSTRING が正しく動作しているかどうかを以下の手順で確認する．

まず，あらかじめ .data セクション (初期値のあるデータ領域を定義するセクション) に，

```
TDATA1:  .ascii "0123456789ABCDEF"
TDATA2:  .ascii "klmnopqrstuvwxyz"
```

と宣言しておく．

テストプログラムは，初期化モジュールを呼び出した後 (USTCNT1 を操作して送信割り込みもマスクしておく) に，スーパーバイザモードのまま，走行レベルを 0 にし，

PUTSTRING(0,#TDATA1, 16)

を実行したのち，数秒の空ループ処理を行う．その後，

PUTSTRING(0,#TDATA2,16)

を繰り返し実行するようにする．

プログラムを実行すると，まず画面に「0123456789ABCDEF」が表示される．これは空ループ処理の途中途中で INTERPUT が 16 回起動され，キューの中に入っている文字列を先頭から順に 1 文字ずつ送信するからである． 次に「klmnopqrstuvwxyz」を数回表示した後，TDATA2 の文字列のプレフィックス (「k」 や 「kl」) が繰り返し表示される．

なぜこのようにプレフィックスが表示されるか考えてみよう．このテストプログラムの後半では，高速に PUTSTRING が繰り返し呼び出される．そして PUTSTRING が呼び出される合間合間に送信割り込み処理，

すなわち INTERPUT が起動される．例えば，PUTSTRING→INTERPUT→PUTSTRING→INTERPUT→... のように交互に呼び出されたとしよう (図 3.4)．このテストでは PUTSTRING は一度に 16 文字も送信キューに書き込む．これに対し INTERPUT は一回に 1 文字しか出せない．従って，PUTSTRING と INTERPUT が交互に出る状況では，INTERPUT による送信処理が追い付かなくなってきて，キューには未送信文字がたまってくる．そうしていずれキューの空領域が 16 文字以下になってしまう．そうなると PUTSTRING は「k」から「z」までの 16 文字すべてを書き込めずに終ってしまう (キュー**が満杯**になると INQ が失敗するので)．さらにキューが完全に満員になると，1 文字分ずつしか空きがでなくなるので,「k」しかキューに入らず，結果的に「k」 ばかりがキューに溜ってくる．

3.10　受信制御部の完成 [Step 6]

受信割り込みに関するルーチンとして，INTERGET，GETSTRING を作成する．INTERGET は，受信割り込みを利用して，受信したデータを次々に受信キューに書き込む．GETSTRING はこの受信キューからデータを取り出す．

3.10.1　GETSTRING の作成

GETSTRING(ch, p, size)

【機能】

- チャネル ch の受信キューから size バイトのデータを取り出し，p 番地以降にコピーする．
- 復帰値 (%D0) は読み出したデータのサイズである．
- 入力キューが空になるとそれ以上は読み出さない．つまり，size 以下の個数の読み出せるだけのデータを取り出す．
- 本実験では、チャネル 0 以外の場合は，何も実行しないように実装する．

【入力】

- チャネル ch → %D1.L
- データ書き込み先の先頭アドレス p → %D2.L
- 取り出すデータ数 size → %D3.L

【戻り値】

- 実際に取り出したデータ数 sz → %D0.L

【処理内容】

(1) ch ≠ 0 ならば何も実行せず復帰．

(2) sz ⟵ 0 , i ⟵ p

(3) sz = size ならば (9) へ

(4) OUTQ(0, data) により，受信キューから 8bit データ読み込み

(5) OUTQ の復帰値（%D0 の値）が 0 (失敗) なら (9) へ

(6) i 番地に data をコピー．

(7) sz++, i++

(8) (3) へ

(9) %D0 ⟵ sz

【注意】

- GETSTRING は，メモリ上 (具体的には 受信キュー上) のデータを別のメモリ上の場所に移動させるだけである．

3.10.2　受信割り込みルーチン INTERGET の作成

INTERGET(ch, data)

【機能】

- 受信データを受信キューに格納する．
- チャネル ch が，0 以外の場合は，何も実行しない．

【入力】

- チャネル ch → %D1.L
- 受信データ data → %D2.B

【戻り値】
- なし

【処理内容】
(1) ch ≠ 0 ならば何も実行せず復帰.
(2) INQ(0, data)

【注意】
- 本ルーチンは受信割り込みに応じて呼び出される．ただし，送信割り込み用ルーチン INTERPUT がハードウェア割り込みインタフェース (3.8.2 節) を介して間接的に呼び出されるのと同様に，受信割り込み用ルーチン INTERGET もハードウェア割り込みインタフェース (次節 3.10.3 にて作成) を介して間接的に呼び出される．
- 受信データ data はレジスタ %D2.B に入っているので，INTERGET はレジスタの値をメモリ上 (受信キュー上) のある場所に移動させるだけである．
- INQ を呼び出す際は，ch=%D0.L，data=%D1.B と，レジスタがずれる点に注意.

3.10.3 受信割り込み用のハードウェア割り込みインタフェースの作成

3.8.2 節では，INTERPUT を読み出す送信割り込み用のハード割り込みインタフェースを作成したが，本節では，INTERGET を読み出す受信割込み用のハードウェア割り込みインタフェースを作成する． 具体的には，受信割り込みに応じて，受信レジスタの内容およびチャンネル情報を引数として，INTERGET を呼び出すように，3.8.2 節で作成したハード割り込みインタフェースに次の処理ステップを加える，

送受信割り込み用のハードウェア割り込みインタフェース (3.8.2 節のつづき)

【処理内容】
(3) 受信レジスタ URX1 を %D3.W にコピー
(4) %D3.W の下位 8bit(データ部分) を %D2.B にコピー
(5) 今起こっている UART1 の割り込みが，受信割り込みであるかを，%D3.W の 第 13 ビット目を用いてチェックする．
(6) 受信割り込みであった場合，チャンネル ch = %D1.L =0, データ data = %D2.B として INTERGET を呼び出す.

【注意】
- 3.8.2 節の【注意】において，送信レジスタ UTX1 の内容が 1 回読むと消えてしまうことを述べた．受信レジスタ URX1 についても同様のことが言える．すなわち **URX1 の内容は，一度読むと (例えば move.w URX1, %D1 とすると) 更新されてしまう**．従って，URX1 の内容を何度も参照する場合，一旦別のレジスタにコピーしておくべきであり，実際，上の (3) で %D3 にコピーしているのはこのためである．
- 呼び出し前のルーチンに戻るには，rte を利用する．

3.10.4 受信制御部の動作テスト

bss セクション (初期値の無いデータ領域を定義するセクション) に，

```
WORK: .ds.b 256
```

と宣言しておく．

テストプログラムでは，初期化モジュールを呼び出した後に，スーパバイザモードのまま，走行レベルを 0 にし，

```
データ入力のための 10 秒程度の時間待ち (空ループ)
GETSTRING(0, #WORK, 256)　　(GETSTRING 復帰値を n とする)
PUTSTRING(0, #WORK, n)
```

を繰り返し実行する．

プログラムロード実行後，10 秒で数文字キー入力したり，数 10 秒間キー入力をしなかったり，キーを押しっぱなしにしたりする．表示される文字により，GETSTRING の復帰値が正しいか，キューが空あるいは一杯のときの処理が正しいかどうか確認する．

プログラム上では，空ループの最中にキー入力に応じた受信割り込みが度々発生し，INTERGET により受信キューに文字が貯められていくことになる．次に，GETSTRING により受信キューから WORK にコピーされる．最後に PUTSTRING により WORK から送信キューにコピーされ，結局ユーザーが入力した文字が画面に表示される．

3.11 タイマ制御部の完成 [Step 7]

3.11.1 タイマ制御ルーチンの作成

タイマ管理ルーチンとして以下の 3 つのルーチンを作成する．

RESET_TIMER()

【機能】

- タイマ割り込みを不可にし，タイマも停止する．

【入力】

- なし

【戻り値】

- なし

【処理内容】

(1) タイマ 1 コントロールレジスタ TCTL1 を設定 (restart, 割り込み不可，システムクロックの 1/16 を単位として計時，タイマ使用停止)．

SET_TIMER(t,p)

【機能】

- タイマ割り込み時に呼び出すべきルーチンを設定する．
- タイマ割り込み周期 t を設定し，t * 0.1 msec 秒毎に割り込みが発生するようにする．
- タイマ使用を許可し，タイマ割り込みを許可する．(=タイマをスタートさせる)

【入力】

- タイマ割り込み発生周期 t → %D1.w (型に注意)
- 割り込み時に起動するルーチンの先頭アドレス p → %D2.L

【戻り値】

- なし

【処理内容】

(1) 割り込み時に起動するルーチン先頭アドレス p を，大域変数 task_p に代入する．
(2) タイマ 1 プリスケーラレジスタ TPRER1 を設定し，0.1 msec 進むとカウンタが 1 増えるようにする．
(3) タイマ割り込み発生周期 t を，タイマ 1 コンペアレジスタ TCMP1 に代入する．
(4) タイマ 1 コントロールレジスタ TCTL1 を設定 (restart, 割り込み許可 (enable the compare interrupt)，システムクロックの 1/16 を単位として計時，タイマ使用許可) し，タイマをスタートさせる

【注意】

- タイマ割り込み (compare interrupt) は，0.1msec 毎に 1 つ増えるカウンタが，タイマ 1 コンペアレジスタに入れていた値に一致した時に発生する．
- カウンタ周波数 = (システムクロック/16) / (TPRER1 の値+1)
- システムクロックは 33.161216MHz
- 1 カウンタあたりの時間 = カウンタ周波数の逆数 → これを 0.1 msec にしたい．
- 従って TPRER1 は .... (自分で求めよ)．
- あらかじめ task_p を，`bss` セクションで定義しておく必要がある．

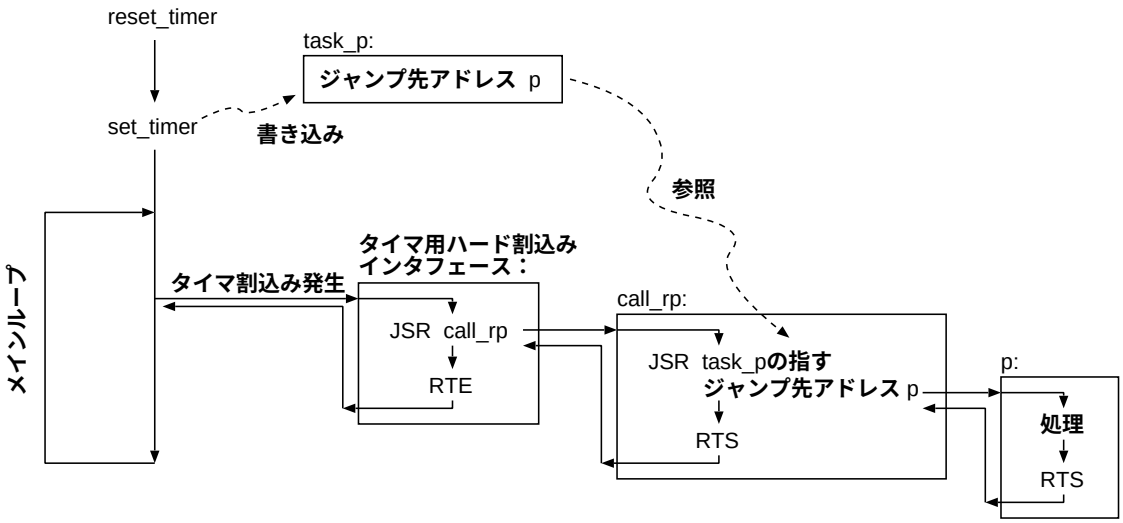


図 3.5: タイマ割り込み処理の流れ

CALL_RP()

【機能】

- タイマ割り込み時に処理すべきルーチンを呼び出す.

【入力】

- なし

【戻り値】

- なし

【処理内容】

(1) 大域変数 task_p の指すアドレスへジャンプする

3.11.2 タイマ用のハードウェア割り込みインタフェースの作成

送受信割り込み用のハードウェア割り込みインタフェースは作成済であるが，本節ではタイマ用のハードウェア割り込みインタフェースを作成する．なお，**タイマ割り込みのベクタを操作し，タイマ割り込みの際に，このハードウェア割り込みインタフェースにジャンプするように設定しておくことを忘れずに**.

このインターフェースでは，前述の CALL_RP() を呼び出すようにしておく．前述のように，CALL_RP は，task_p が指すアドレスにジャンプするように設計されている．従ってこのハードウェア割り込みインタフェースを作成することで，時間になると，SET_TIMER でセットしたアドレスに処理を移す仕組みが完成する (図 3.5 参照).

タイマ用ハードウェア割り込みインタフェース

【処理内容】

(1) タイマ 1 ステータスレジスタ TSTAT1 の第 0 ビットが 1 となっているかどうかをチェックする．0 ならば rte で復帰.

(2) タイマ 1 ステータスレジスタ TSTAT1 を 0 クリアしておく．(TSTAT1 のリセット)

(3) CALL_RP を呼び出す.

【注意】

- タイマ割り込みは 2 種類あって，本プログラムで使用している compare interrupt 以外に，capture interrupt がある．本実験ではカウンタが設定数に達した時に発生する compare interrupt だけを利用するために，上記 (1) でそのチェックを行っている．
- 呼び出し前のルーチンに戻るには，rte を利用する．

3.12　システムコールインタフェースの完成 [Step 8]

3.12.1　システムコールの概要

1.4.2 節や 2.3 でも触れているように，OS はシステムコールと呼ばれる細分化された機能の提供する．本実験では，これまで作成した以下の 4 つのルーチンをシステムコールとして提供する簡易 OS を作成する．

PUTSTRING(ch, p, size)
チャネル ch に， p 番地から size バイトのデータを出力する．

GETSTRING(ch, p, size)
チャネル ch からの入力を， size バイトだけ p 番地以降に読み込む．

RESET_TIMER()
タイマ割り込みを停止させる．

SET_TIMER(t, p)
タイマ割り込みの周期を t*0.1 msec に，タイマ割り込みで呼び出されるプログラムの開始番地を p に設定する．

以下では便宜上，各システムコールに次の番号を割り当てる．

番号	システムコール
1	GETSTRING
2	PUTSTRING
3	RESET_TIMER
4	SET_TIMER

3.12.2　システムコールインターフェースの作成

各システムコールは，ユーザプログラムにおいて trap #0 割り込みにより起動される．例えば，文字列受信に関するシステムコール GETSTRING を呼び出す場合は次のように行う．

```
move.l 1,    %D0   | GETSTRING
move.l #0,   %D1   | ch    = 0
move.l #BUF, %D2   | p     = #BUF
move.l #256, %D3   | size = 256
trap #0
```

レジスタ%D0 には，システムコール番号を格納している．レジスタ%D1 から%D3 には，そのシステムコール (この場合 GETSTRING) の引数を指定している．以上の例では，チャンネル 0 の受信キューから，BUF で指す領域に 256 個のデータを読み込むことになる．

以上の手順によると，どのシステムコールを呼ぶ場合も，同じ trap #0 の割り込みを使うことになる．従って，どのシステムコールを呼ぶ場合でも，ひとまずは trap #0 のベクタで指定された同じ処理にジャンプすることになり，そこで次にどこのシステムコールにジャンプするべきかを判断することになる (→2.3 図 2.5)．この処理部分をシステムコールインタフェースと呼ぶ．

システムコールインタフェース
【機能】

- 呼び出すべきシステムコールを，%D0 (システムコール番号 1-4 を格納) を用いて判別する．
- 目的のシステムコールを呼び出す

【入力】
- システムコール番号 → %D0.L
- システムコールの引数 → %D1 以降

【戻り値】
- システムコール呼び出しの結果 → %D0.L

【処理内容】
(1) システムコール番号 %D0 を 実行先アドレス (%D0=1 ならば GETSTRING の先頭アドレス) に変換する．
(2) システムコールを呼び出す (=実行先アドレスにジャンプ)．

【注意】
- 本来ならば，処理 (1) と (2) の間に「引数処理」が行われるべきである．
- 例えば GETSTRING を trap #0 で呼び出す際に，%D3 → ch，%D1 → size としていれば，%D1 と%D3 の内容を交換する引数処理を行う必要がある．
- しかし，あらかじめレジスタの対応関係に気をつけておくことで，この引数処理を省略することができる．(本テキストに従えば，自然と省略できるようになっている．)
- スタック待避/復旧すべきレジスタに気をつけること．(%D0 を退避／復帰してしまうと...!?)
- 処理 (1) の組み方は様々考えられる．拡張性が高くなる (=新しいシステムコールが増えても，変更が少ない) よう工夫することが望ましい．
- 呼び出し前のルーチンに戻るには，rte を利用する．

3.13　ユーザプログラムの完成 [Step 9]

3.13.1　エコーバックプログラムの作成

本ステップでは，これまでに作成したシステムコールをすべて利用して，エコーバック (+タイマ処理) プログラムを作成する．具体的には，初期化ルーチンの最後尾に以下のユーザプログラムにジャンプするようにする．

```
***************
** システムコール番号
***************
.equ SYSCALL_NUM_GETSTRING,     1
.equ SYSCALL_NUM_PUTSTRING,     2
.equ SYSCALL_NUM_RESET_TIMER,   3
.equ SYSCALL_NUM_SET_TIMER,     4

****************************************************************
*** プログラム領域
****************************************************************
.section .text
.even
MAIN:
** 走行モードとレベルの設定 (「ユーザモード」への移行処理)
move.w #0x0000, %SR  | USER MODE, LEVEL 0
lea.l USR_STK_TOP,%SP  | user stack の設定

** システムコールによる RESET_TIMER の起動
move.l #SYSCALL_NUM_RESET_TIMER,%D0
trap   #0

** システムコールによる SET_TIMER の起動
move.l #SYSCALL_NUM_SET_TIMER, %D0
move.w #50000, %D1
move.l #TT,    %D2
trap #0

*****************************
* sys_GETSTRING, sys_PUTSTRING のテスト
* ターミナルの入力をエコーバックする
*****************************
LOOP:
move.l #SYSCALL_NUM_GETSTRING, %D0
move.l #0,   %D1         | ch    = 0
move.l #BUF, %D2         | p     = #BUF
move.l #256, %D3         | size = 256
trap #0

move.l %D0, %D3          | size = %D0 (length of given string)
move.l #SYSCALL_NUM_PUTSTRING, %D0
move.l #0,  %D1          | ch = 0
move.l #BUF,%D2          | p  = #BUF
trap #0

bra LOOP

*****************************
*   タイマのテスト
* '******' を表示し改行する.
* 5回実行すると，RESET_TIMER をする.
*****************************
TT:
movem.l %D0-%D7/%A0-%A6,-(%SP)
cmpi.w #5,TTC            | TTC カウンタで 5 回実行したかどうか数える
beq TTKILL               | 5 回実行したら，タイマを止める

move.l #SYSCALL_NUM_PUTSTRING,%D0
move.l #0,    %D1        | ch = 0
move.l #TMSG, %D2        | p  = #TMSG
move.l #8,    %D3        | size = 8
trap #0
```

```
addi.w #1,TTC              | TTC カウンタを 1 つ増やして
bra TTEND                  | そのまま戻る
TTKILL:
move.l #SYSCALL_NUM_RESET_TIMER,%D0
trap #0
TTEND:
movem.l (%SP)+,%D0-%D7/%A0-%A6
rts

*****************************************************************
*** 初期値のあるデータ領域
*****************************************************************
.section .data
TMSG:
.ascii  "******\r\n"       | \r: 行頭へ (キャリッジリターン)
.even                      | \n: 次の行へ (ラインフィード)
TTC:
.dc.w  0
.even

*****************************************************************
*** 初期値の無いデータ領域
*****************************************************************
.section .bss
BUF:
.ds.b 256             | BUF[256]
.even

USR_STK:
.ds.b 0x4000             | ユーザスタック領域
.even
USR_STK_TOP:            | ユーザスタック領域の最後尾
```

3.13.2 エコーバックプログラムの正常動作の確認

以上の，プログラムを作成した後，以下の動作を確認する．

- キーを打つとそれが画面にエコーバックされる．
- 5 秒毎に****** が表示され改行が行なわれる．ただし，この表示は 5 回で終了する．

なお，エコーバックの確認のとき，でたらめにキー入力するよりは，例えば “a” と “b” を正確に交互に入力するほうが，プログラムミスを見つけやすい．

3.14　選択課題プログラムの完成 [Step 10]

次の 14 種の課題の中から各自 1 つを選び，実際にプログラムを作成する．**ただし，班員全員が異なった選択課題を作成することを条件とする (課題 14 については内容が違えば複数人が選択してもよいものとする)．**課題によって難易度は異なる．どれでも 1 つ組めばよいが，より難易度が高いもののほうが点数は高くなる． なお，エレガント (効率的，汎用的，プログラムが見易いなど) に実装するなど，工夫が見られる場合についてはさらに加点する．また，以下の課題を出発点として，それをうまく発展させた場合についても加点する．なお，CPU ボード上の LED を使っても良い (付録 A.4.3.1 参照).

1. タイマ割り込みによる文字の表示 · · · 難易度 1
   - タイマ割り込みにより，一定時間毎に文字を表示させる．
   - 表示させる文字は複数用意する．
2. 質疑応答 (1)· · · 難易度 1
   - 質問に対して Y,N で回答し，その結果をもとに何かの結果を表示させる（例えば，性格診断など）.
3. キャラクタアニメーション · · · 難易度 1∼
   - 文字列を変化させながら表示を繰り返すことで，動いているように見せる．
   - 文字列は 1 行でもよいし，複数行でもよいし，画面全体でもよい．
4. 入力文字の統計を表示 · · · 難易度 2
   - 一定時間内にランダムに a-z の文字を入力させる．
   - a-z がそれぞれ何回入力されたかをカウントしておく．
   - 最後に最も入力数の多かった文字を表示する．
5. 質疑応答 (2)· · · 難易度 2∼
   - 質疑応答 (1) に次の 2 点を加える
   - 質問には制限時間を設ける．
   - また回答に要した時間により，最終的な結果を変化させる．
6. 時計 (1)· · · 難易度 2∼
   - プログラムを実行してからの秒数を表示させる．
7. 時計 (2)· · · 難易度 2∼
   - プログラムを実行してからの時間を”mm:ss”という形で表示.
8. ストップウォッチ · · · 難易度 2
   - キー操作により，スタート，ストップ，リセットを行う．
9. タイプ練習 (1)· · · 難易度 2∼
   - 文字列（単語列）を画面に表示する．
   - ユーザにその文字列を入力させる．
   - 間違えた入力は画面出力しない．

- 文字列の長さにより，制限時間を変える．

10. タイプ練習 (2)··· 難易度 3〜
    - タイプ練習 (1) に次の 2 点を加える
    - 最初にレベル選択を行う．
    - 選択したレベルにより文字表示の速度が変化する．
11. タイプ練習 (3)··· 難易度 3〜
    - タイプ練習 (1) に次の 2 点を加える
    - 正解数，誤り数を集計し表示させる．
    - タイプスピード (打/分) を計算し表示させる．
12. スロットマシン ··· 難易度 3〜
    - キー入力するたびに，画面にランダムな複数桁の数字を表示させる．
    - 当たり/はずれの判定を行う．
    - 真にランダム，すなわち何が次に出るかプログラマすら予測できないようにすること．(テーブルとして乱数を持つような実装は不可)
13. 四則演算が可能な電卓 ··· 難易度 3〜
14. オリジナルプログラム ··· 難易度 1〜
    - 作成したシステムコールを利用して (すべて利用する必要はない)，各自自由にプログラムを作成する．
    - 創造性を発揮して，以上の課題例に無いような面白いプログラムに期待する．

3.15　最終試問

本実験の最終回に班ごとの試問を行う．各自教員の前でプログラムを動かしてもらう．レポートを参照しながら，口頭で説明して頂く．最終試問の結果により，本実験に関する合否が確定する (1.2 節も参照のこと)

動作確認するもの

- 簡易 OS 本体＋ Step 9 のユーザプログラム (=エコーバックプログラム)
- 各個人の選択課題のプログラム (Step 10)

班としての提出物

- 簡易 OS 本体＋ユーザプログラム (Step9) のプリントアウト．仕様説明等，充分にコメントを記載しておくこと．
- 簡易 OS 本体，ユーザプログラム (Step9)，選択課題 (Step10, 班員全員分) のプログラム．(プログラムは USB メモリを教員が渡すのでそれにコピーして提出する．)

各個人の提出物

- Step 10 の選択課題プログラムのプリントアウト，およびその内容/工夫した点を説明したレポート．

Step 10 のレポートにおいて，プログラム作成の際に工夫した点や発展させた点をきちんとアピールすること．こうした工夫は加点の対象となる．

各個人への試問

- 担当部分に関連する試問

自分が担当した作業内容（初期化，タイマ，受信，送信）について，何点か試問を行うので，担当部分に関しては特に詳しく内容を理解しておくこと．

付 録A　プログラムの作成と実行

A.1　プログラム開発環境

A.1.1　CPU ボード

A.1.1.1　システムの構成

本実験では，68000 互換 の CPU MC68VZ328 (Dragonball-VZ) を搭載した CPU ボードを使用する．

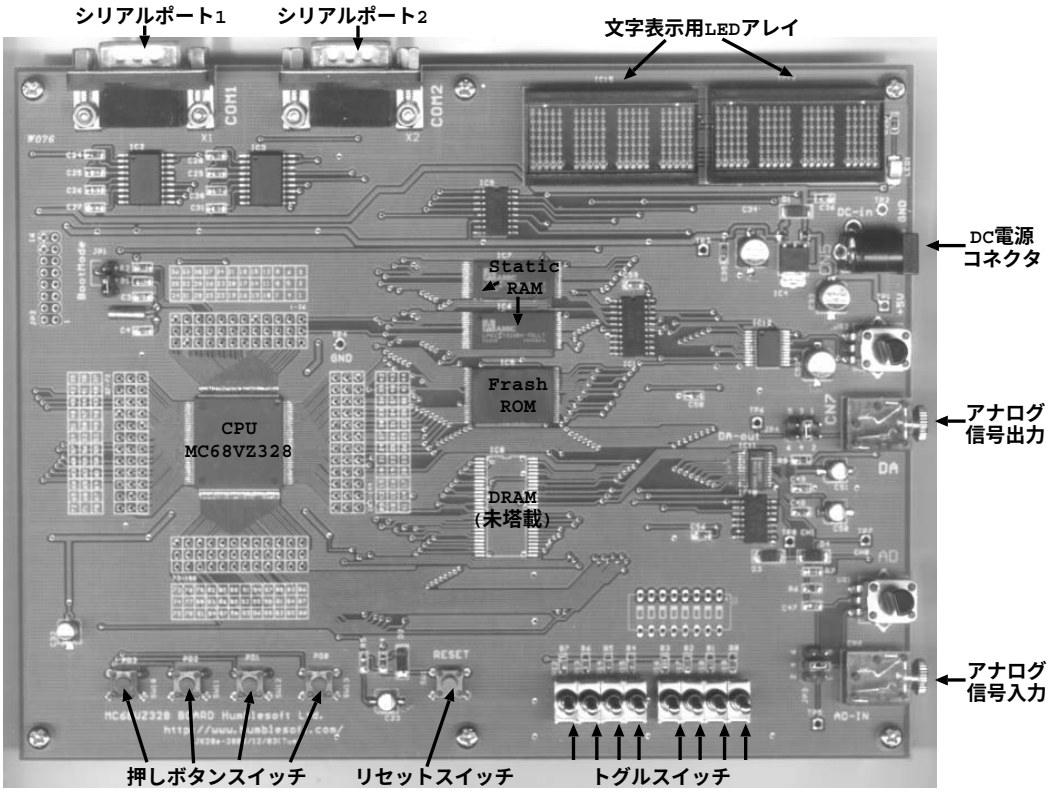


図 A.1: 本実験で使用する CPU ボード

表 A.1: CPU ボードの仕様

CPU	Motorola MC68VZ328 “Dragonball-VZ”
システムクロック	33MHz (33,161,216Hz)
メインメモリ	256KByte (Static RAM)
ブート (起動用)ROM	512KByte (Flash ROM)
シリアルポート	2 ポート (RS-232C) (通信条件: 1 stop bit, 8bit character, no parity, 38400bps)
押しボタンスイッチ	4 個
トグルスイッチ	8 個
LED アレイ	8 文字

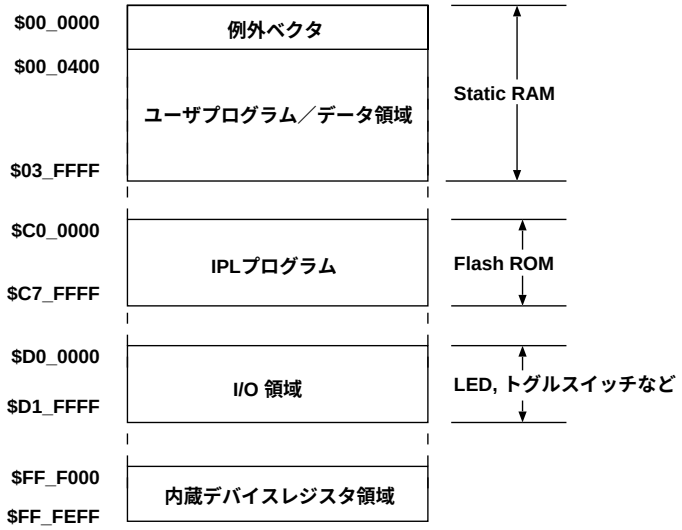


図 A.2: MC68VZ328 CPU ボードメモリマップ

A.1.1.2　準備

CPU ボードは各人に 1〜2 台が割り当てられる．本実験では，それに加えて 1 人 1 台プログラム開発用のノート PC が与えられる．実験開始時には次のように機器を準備する．

1. CPU ボードの COM1 ポートとノート PC 背面のシリアルポートを RS-232C クロスケーブルで接続する．
2. ノート PC の電源を入れ，kterm 上で通信端末ソフト (m68k-term) を起動する．
3. まず CPU ボードの DC 電源コネクタに AC アダプタを接続し，それから AC アダプタを机上のテーブルタップに接続する．

なお，CPU ボードの電源を切る際には，電気的なショックを避けるため必ず AC アダプタの方を先にテーブルタップから外すこと．

A.1.1.3　CPU ボードの起動

CPU ボードをリセットするには，リセットスイッチを押す．CPU ボードの電源投入時，またはリセット時には，ブート ROM に格納されている IPL プログラム (Initial Program Loader）が起動し，次のような処理が自動的に行われる．

1. LED に “`IPLReady`” と表示し，起動メッセージを COM1 ポートから送信する．これはそのままノート PC 上の m68k-term の画面に表示される．
2. ここで，ノート PC から 68000 の実行ファイルを転送すると，CPU ボードはそれを COM1 ポートを通して受信し，メインメモリに格納する．
3. プログラムの格納が終わったら LED に “`go ...  ?`” と表示して COM1 から 1 文字受信するまで待ち，受信確認後 LED に “`Run ....`” と表示し，0x400 番地へジャンプする．

従って，0x400 番地から始まるプログラムを作成して外部から送り込んでやれば，CPU ボードは完全にそのプログラムの制御下に入る．その際の CPU ボードのメモリマップは図 A.2 のようになる．

A.1.2 ノート PC の起動と終了

A.1.2.1 起動

ノート PC の電源投入後，OS 選択画面 (Windows XP or Red Hat Linux) が起動するので，Red Hat Linux を選択する．しばらくすると，ログインを求めるプロンプトを表示する．すべての学生は基本的に「student」というアカウントでログインすることになる．初期パスワードについては教員の指示に従うこと．

ログイン後，下のアイコン列左端の「赤い帽子」アイコンをクリック,「システムツール」→「ターミナル」で，kterm が起動される．

A.1.2.2 終了 (シャットダウン)

ノート PC の電源をいきなり切ってはならない．下のアイコン列左端の「赤い帽子」アイコンから，ログアウト → シャットダウンを選択，実行する．しばらくすると自動的に電源オフとなる．

※なお，PC や実験機材は物理的，電気的なショック (例: 鞄に引っ掛けて机から落としたり，鉛筆で回路を突っつくなど) を与えると壊れることがあるので大切に扱うこと．故意に破損した場合は，弁償頂くことになる (CPU ボードは 1 枚 20 万円以上)．

A.2 プログラムの作成

本実験で使用する CPU ボードはプログラム開発の機能を有しないため，ここでは PC 上でプログラム開発を行い，出来上がったプログラムを CPU ボードに転送して実行，という手順をとる．

A.2.1 ソースファイルの編集

まず，アセンブリ言語 (機械語) で記述したソースファイルをテキストエディタ (emacs) を用いて入力・編集する．ファイルの拡張子は `.s` とする．

```
% emacs step1.s
```

A.2.2 実行形式ファイルの作成 (アセンブラ m68k-as の使い方)

ソースファイル`.s` が完成したら，アセンブラにかけて 68000 の実行ファイル (拡張子は `.abs`) を生成する．CPU はアセンブリ言語で書かれた人が読める形のソースファイルを直接実行することはできないため，CPU ボードでプログラムを動かすためには，まずアセンブラを用いてプログラムを 68000 の実行形式に変換しなければならない．本実験では，アセンブラとして`m68k-as` コマンドを使用する．

ソースファイル `step1.s` から実行形式 `step1.abs` を得るには，次のようにする．なお，`-t 400` は開始番地を指定するオプションであり，プログラムを CPU ボードで起動するために必要である．

```
% m68k-as -t 400 step1.s
```

`m68k-as` でソースファイルをアセンブルすると，まずリスティングファイル (`step1.LIS`) が作成される．`.LIS` ファイルには，機械語コードとそのアドレス，ラベルの値などが整理されて詳細に記されており，これによって，アセンブル時のエラーの箇所や種類，プログラムやデータ領域の実行時の絶対アドレスなどを知ることができる．もしエラーがあれば`.abs` ファイルは出力されないので，A.2.1 へ戻り完全にエラーが無くなるまでプログラムの訂正とアセンブルを繰り返す必要がある．

そして最終的にエラーが無くなれば step1.abs ファイルが生成される．.abs ファイルは実際には CPU で直接実行可能なバイナリーデータではなく，バイナリーデータをファイル転送に適した 16 進テキストで表現した S レコード形式になっているが，本実験で用いる CPU ボードの IPL は，この S レコード形式の .abs ファイルを読み込み，内部でバイナリーデータに展開する機能を持っているため，CPU ボードにはこの.abs ファイルを転送すればよい．

A.3　プログラムの転送と実行 (通信端末ソフト m68k-term の使い方)

本実験では端末エミュレータとしてm68k-term を使用する．m68k-term は端末モードとコマンドラインモードの 2 つのモードを持つ．端末モードでは，PC へのキー入力がそのまま CPU ボードへ転送され，CPU ボードから送信されてきた文字がそのまま端末画面に表示されるようになるため，PC を通して間接的に CPU ボードを操作することができる．コマンドラインモードでは，“kermit@...>” のようなプロンプトが表示され，プログラムの転送 (xm) や，接続 (c)，m68k-term の終了 (q) といったコマンドを使用できる．2 つのモード間の切替は，端末モードからコマンドラインモードへの移行はC-\ c (コントロールキーを押しながらバックスラッシュ ‘\’ を同時押下した後，‘c’ を押下)，コマンドラインモードから端末モードへの移行は接続 (c) コマンドにより行う．

m68k-term を用いたプログラム転送と実行の手順は次のようになる．

1. まず最初に，PC 上で通信端末ソフト (m68k-term) を起動させておく．初期状態は端末モードとなる．

   ```
   % m68k-term
   ```

   (注：最後に「&」をつけてバックグラウンド実行してはならない)

2. 次に CPU ボードの電源を静かに ON にし，起動メッセージが表示されることを確認する．電源が既に ON の場合はリセットスイッチを押すか，一旦 OFF にして再起動する．

3. C-\ c でコマンドラインモードに入り，.abs ファイルを CPU ボードへ送信する．例えば step1.abs を CPU ボードに転送 (transmit) する場合，次のようにする．

   ```
   > xm step1.abs
   ```

4. プログラムの転送が終わったら，接続 (connect) コマンドで端末モードに戻る．

   ```
   > c
   ```

5. すると，“... hit any key.” というメッセージが表示されるので，任意のキーを打って，ユーザプログラムの実行を開始する．

6. m68k-term を終了 (quit) するためには，C-\ c で一旦コマンドモードに戻り，

   ```
   > q
   ```

   とする．

以下，実行例を示す．ただし，“[RET]” はリターンキーを表す．

```
% m68k-term
```

```
Connecting to /dev/ttyS0, speed 38400.
 Escape character is Ctrl-\ (ASCII 28, FS): enabled
Type the escape character followed by C to get back,
or followed by ? to see other options.
----------------------------------------------------
MC68VZ328 Board IPL Program. by Humblesoft.
Ver. 1.01-01 2004/07/05(Mon) (38400bps, 33MHz)
waiting for S-file

C-\ c
(Back at HWSW-PCxx)
----------------------------------------------------
C-Kermit 8.0.206, 24 Oct 2002, for Red Hat Linux 8.0
 Copyright (C) 1985, 2002,
  Trustees of Columbia University in the City of New York.
Type ? or HELP for help.

kermit@HWSW-PCxx> xm step1.abs

..............

kermit@HWSW-PCxx> c

Connecting to /dev/cuaa0, speed 38400.
 Escape character is Ctrl-\ (ASCII 28, FS): enabled
Type the escape character followed by C to get back,
or followed by ? to see other options.
----------------------------------------------------
....................................................

*** Load Success ***
Load  Address:00000400-0000076c
Start Address:00000400
if you want to start your program, hit any key

[RET]

*** your program start. ***
a

C-\ c
(Back at HWSW-PCxx)
----------------------------------------------------

kermit@HWSW-PCxx> q

Closing /dev/ttyS0...OK
%
```

A.4　プログラムのデバッグ

出来上がったプログラムが即，正常に動作することは極めてまれである．多くの場合，プログラムが完動するようになるまでには，様々な条件下で動作試験を行ってはプログラムの誤り (bug) を発見し，修正していく地道な作業の繰り返しが必要となる．

A.4.1　プリントアウト

デバッグを始めるにあたり，まずプログラムをプリントアウトしたものを用意しておくとよい．プログラムを互いにチェックし合えば，自分では気付かなかった間違いをより速く発見できる．役割分担に固執せず，班員皆で情報を共有し，協力して行うことが成功の秘訣である．印刷の方法は付録 A.5 の「6 印刷」に示す．

A.4.2　エミュレータ m68k-emu の使い方

プログラム中の問題のある個所を特定するためには，プログラムをエミュレータ上で動作させてみることが有効である．エミュレータを使用すると，ノート PC 上の仮想的な CPU 上でプログラムを実行させながら，CPU のレジスタやメモリ中の値を観察することができる．このエミュレータは非常に強力であり，trap 命令や bus error 等の CPU 内部の割り込み動作まで忠実に再現される．ただし，残念ながら本実験で使用するエミュレータ (BSVC) には，CPU ボードに搭載されている CPU MC68VZ328 (Dragonball-VZ) に固有の内部デバイスが実装されていないため，シリアル入出力やタイマ等の動作を再現することはできない．

エミュレータを使用する際には，まず次のようにしてエミュレータを起動して “File/Load Program” コマンドで.abs ファイルを読み込み，プログラムカウンタ PC の値を簡易 OS の開始番地 0x400 に設定する．

```
% m68k-emu
```

内部デバイスからのハードウェア割り込みを起こすことはできないが，適当なタイミングで PC の値を変更することによって，割り込み処理ルーチン内の動作チェックも可能である．ルーチンやデータのアドレスは “Window/Program Listing” で表示されるウィンドウ，またはアセンブル時に生成される.LIS ファイルによって知ることができる．

A.4.3　CPU ボード上でのデバッグ

デバイスの動作まで含めた総合的なデバッグのためには，やはり実際の CPU ボードでの動作チェックが必要となる．

- 1 文字を表示する行をプログラム中に挿入してみて，もし実行時に文字が表示されなければ，その地点より以前に何らかの問題があってそこまで到達できなかったものと考えられる．
- 様々な例外について，割り込みルーチン内でそれぞれ異なる 1 文字を表示するようにしておけば，どの割り込みが入ったかを見分けることができる．

文字表示には送信レジスタ (UTX1) や次節で説明する LED を使用するとよい．

A.4.3.1 LED の使い方

本実験で使用する CPU ボードはボード上に LED アレイを搭載しており，8 個の文字の表示が可能である．LED に関する設定は起動時に IPL において初期化済みであり，プログラム中では単に文字コードを決まったアドレスに書き込むだけで文字を表示させることができるように設定されている．

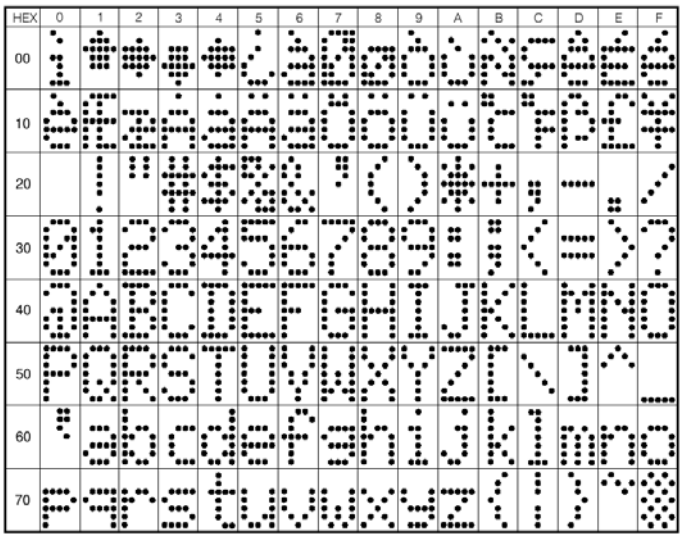
LED の各桁には，ハードウェアの都合上連続しないアドレスが割り当てられている．これらについては，3.5 節 (29 頁) の step1 の初期化ルーチンにおいて左から順に LED7〜LED0 の 8 つのラベルが定義されている．プログラム中ではそのラベルを用いて各桁に文字コードを書き込めばよい．

例えば次のようにすると，LED には”12345678” と表示される．

```
move.b  #'1', LED7      /* 文字 '1' を LED の 8 桁目に表示 */
move.b  #'2', LED6      /* 文字 '2' を LED の 7 桁目に表示 */
move.b  #'3', LED5      /* 文字 '3' を LED の 6 桁目に表示 */
move.b  #'4', LED4      /* 文字 '4' を LED の 5 桁目に表示 */
move.b  #'5', LED3      /* 文字 '5' を LED の 4 桁目に表示 */
move.b  #'6', LED2      /* 文字 '6' を LED の 3 桁目に表示 */
move.b  #'7', LED1      /* 文字 '7' を LED の 2 桁目に表示 */
move.b  #'8', LED0      /* 文字 '8' を LED の 1 桁目に表示 */
```

本機で用いる LED の文字コード表を表 A.2 に示す．このうち，0x20〜0x7E の範囲内は ASCII コードと同一であるが，0x00〜0x1F に記号が追加されたものになっている．

表 A.2: LED の文字コード



(PD4435 data sheet[2] より一部改変)

A.4.3.2　押しボタンスイッチの使い方

本実験で使用する CPU ボードは 4 つの押しボタンスイッチを搭載しており，CPU はその状態をメモリアドレス 0xFFF419 の下位 4bit として参照することができる．押しボタンの各ビットの値は通常 1 であり，ボタンが押された瞬間に 0 となる．例えば，プログラムの実行が速すぎる時などに，プログラム中で次のようなサブルーチンを呼び出すようにすれば，左端の押しボタンスイッチが押されるまでプログラムを一時停止させておくことができる．

使用例：

```
.equ PUSHSW, 0xFFF419

pause:  btst.b #3, PUSHSW    /* 押しボタンの bit 3(左端) をテスト */
        bne pause             /* 1 の間ループ */
        rts
```

A.4.3.3　トグルスイッチの使い方

本実験で使用する CPU ボードは 8 つのトグルスイッチを搭載している．これは，2 進数での数値入力や，1 ビットのフラグの値の入力などに利用できる．8 つのトグルスイッチはメモリ空間中のアドレス 0xD00041 にある 1 バイト (8bit) 上にマッピングされており，1 つのスイッチの on/off の状態が，1bit の 1/0 の値に対応している．CPU からはこのアドレスを読み出すことによって，8 つのスイッチの状態をまとめて取得することができる．

使用例：

```
.equ TOGGLESW, 0xD00041
       move.b TOGGLESW, %D0    /* 1 バイトの値を読み取って， */
       move.b %D0, LED0        /* それを LED に表示 */
```

A.5 主な UNIX 命令一覧

1. UNIX 標準コマンドによるファイル操作.

% cp *file*1 *file*2	···	*file*1 を *file*2 にコピー
% rm *file*	···	*file* を消去
% mv *file*1 *file*2	···	ファイル *file*1 の名前を *file*2 に変更
% ls	···	現在のディレクトリに存在するファイルの一覧表示.
% ls -l	···	詳細表示.

2. USB メモリの mount, umount 操作.

% mount /mnt/usbfm	···	USB メモリを mount する
	···	この後は，umount するまで，usbfm はファイルデバイスとして利用可能
% umount /mnt/usbfm	···	USB メモリを unmount する

なお，/etc/fstab に/mnt/usbfm の mount 設定がなされておらず，マウントポイント (/mnt/usbfm) が作られていない場合には，mount, umount は root になって操作を行う必要がある．特に，mount 操作は，次のように行う.

#mkdir /mnt/usbfm ⋱ ただし，/mnt/usbfm が無い時だけ

#mount -t vfat /dev/sda1 /mnt/usbfm/ ⋱ ただし，USB メモリの device 名が/dev/sda1 の場合

3. mtools を用いた USB の操作.

以下，USB メモリを USB，ハードディスクを HD と略記する.

% mcopy *file*1 /mnt/usbfm/*file*2	···	HD 上の *file*1 を USB 上に *file*2 としてコピー
% mcopy /mnt/usbfm/*file*1 *file*2	···	USB 上の *file*1 を HD 上に *file*2 としてコピー
% mdel /mnt/usbfm/*file*	···	USB 上の *file* を削除
% mren /mnt/usbfm/*file*1 *file*2	···	USB 上のファイル *file*1 の名前を *file*2 に変更
% mdir /mnt/usbfm/	···	USB に格納されているファイルの一覧表示

4. ファイルの編集.

% emacs *file.s* &	···	*file.s* を作成，編集.

C-\	日本語入力モード切替 (a↔ あ)
SPC	変換
C-o, C-i	文節拡大，縮小
C-f, C-b	文節移動
RET	確定

5. プログラムの実行.

% m68k-as -t 400 *file.s*	···	*file.s* をアセンブル
% m68k-emu	···	エミュレーターで実行
% m68k-term	···	通信端末ソフトを起動

C- c	コマンドラインモードへ切替
> xm *file.abs*	ファイルを転送
> c	端末モードへ戻る
> q	終了

6. 印刷.

% a2ps *file.s* | lpr -P*printer*	···	*file.s* を *printer* で印刷 *printer* = HWSW-P3 〜 HWSW-P4 (-P の後にスペースを空けてはならない)

A.6 アセンブリ言語の書式

本節では GNU アセンブラ (gas) のアセンブリ言語の書式について述べる．

A.6.1 シンボルの命名規則

シンボル (=ラベル，定数) は文字列で表わされる．はじめの１文字目は，アルファベット，‘.’（ピリオド）および ‘_’（アンダーライン）のうちのいずれかでなければならない．2 文字目からは ‘$’（ダラー）と数字も使うことができる．大文字と小文字は区別される．

A.6.2 ステートメント

`m68k-as` アセンブリ言語は，インストラクション (機械語命令コード)，ディレクティブ (擬似命令)，マクロ，コメント (註釈)，の４種類のステートメントから構成される．この実験ではマクロは使用しないこと．

A.6.2.1 ステートメントの書式

１つのステートメントは１行に対応し，コメント以外のステートメントでは１行を次のようなフォーマットで記述する．

LABEL OPERATION OPERAND COMMENT

各フィールドの書き方は次の通り．

A.6.2.1.1 ラベルフィールド

ラベルは記入された位置のプログラムやデータの絶対アドレスに付ける名前である．最後は必ず ‘:’ (コロン) で終わる．ラベル本体とコロンの間にスペースを入れてはならない．ラベルだけの行も可能である．ここで定義されたラベル (‘:’ を除いた部分) は，同じプログラム中のオペランドフィールドにおいて，アドレス値を表す数値定数として参照することができる．

A.6.2.1.2 オペレーションフィールド

このフィールドには命令のニーモニックや後述の疑似命令（ディレクティブ）を記述する．２カラム目から始めるか，あるいは１つ以上のブランク，タブでラベルフィールドと区切る必要がある．

A.6.2.1.3 オペランドフィールド

ここでは，オペレーションフィールドに記入した命令やディレクティブに与えるオペランド (引数) を指定する．命令によってはオペランドを持たない場合もある．このフィールドは１つ以上のブランクあるいはタブによってオペレーションフィールドと区別される．なお，複数のオペランドを取る命令の第一オペランドと第二オペランドの間は ‘,’ (カンマ) で区切る．

A.6.2.1.4　コメントフィールド

各行の行末にはコメント (註釈) を付けることができる．記号 '|' の後ろ側，もしくは記号 "/*" と "*/" で囲まれた部分がコメントフィールドとして認識される．後者については，行末に限らずどの場所に書いてもよい．しかしネスティング ("/*" と "*/" で囲まれた部分をさらに "/*" と "*/" で囲むこと) はできない．

コメントフィールドでは日本語の 2 バイト文字も許される．一般にアセンブリ言語で記述されたプログラムは難解になりがちであるため，他の人のみならず数日後の自分のためにも，その行で何をするのか，なぜそうしたのか，といった意図を明確にしておく必要がある．例えば，サブルーチンの先頭には，そのサブルーチンの引数と返り値の仕様やサブルーチン内部で使用するレジスタの役割等を記述するべきである．

A.6.2.2　コメントステートメント

ラインの最初の非空白文字が '*'(アスタリスク) もしくは '|' である行全体もコメントとして扱われる．空行もコメントとして処理される．

A.6.3　ディレクティブ

アセンブリ言語には，68000CPU の命令以外に，アセンブラを制御するための擬似命令(ディレクティブ) が多数用意されている．ディレクティブは，'.'(ピリオド) で始まる．ここでは，GNU アセンブラ (gas) 上で今回使用すると思われるディレクティブについてのみ説明する．

A.6.3.1　.section

プログラムをデータ領域とコード領域などのモジュールに分けて記述するために .section という命令を用いる．

シンタックス：

```
.section  type
```

本実験では，オペランド *type* として，.text， .data， .bss の 3 種類が用いられる．それぞれの意味は以下の通り．

.text	:	プログラム領域
.data	:	初期値のあるデータ領域
.bss	:	初期値のないデータ領域

プログラムのほとんどすべてのパートは，以上 3 つのいずれかのセクションに属する．セクションタイプの宣言は次の.section が出現するまで有効である．

A.6.3.2　.include

他のファイルをソースファイル中に取り込む．プログラムを複数ファイルに分割すると編集しやすいかもしれない．ただし本実験では簡易 OS 部分を最終的に 1 つのファイルにまとめて提出すること．

```
.include  "filename"
```

A.6.3.3 .end

メインモジュールを含むファイルの最後に記述する．複数のライブラリモジュールをリンクする際には，この .end ディレクティブが含まれているモジュールが，メインプログラムとみなされる．

オペランドを指定した場合，それは実行開始番地と見なされる．本実験では，アセンブル時のオプション(-t 400) により実行開始番地を指定するので，オペランドと共に .end を利用することはない．

A.6.3.4 .equ

このディレクティブを用いて，様々な値に新しい名前を付けることができる．例えば，内蔵デバイス用レジスタのアドレス等の定数値に名前を付ければ，プログラムをわかりやすくすることができる．

シンタックス：

.equ *symbol*, *expression*

説明：

symbol このステートメントによって定義されるシンボル.

expression 割り当てる値.

A.6.3.5 .align / .even

適当なバイト数の空きデータ領域を生成することによって，次のメモリ領域の先頭アドレスをオペランドで指定された数値の整数倍に合わせる．

シンタックス：

.align *operand*

.even は .align の特別な場合である．アドレスを偶数にするときは， .even を使う方がオペランドが不要であり，簡潔である．本実験では .even を採用している．

A.6.3.6 .ds

アセンブル時に指定された大きさのデータ領域を確保する．確保されたデータの初期値は不定である．通常 .bss セクションに記述し，初期化を必要としないデータ領域の確保に用いる．

シンタックス：

{*label*} .ds{.*qualifier*} *operand*

説明：

label このラベルは確保されたデータ領域の先頭アドレスを指す.

qualifier データ 1 つのサイズの指定．バイトデータでは b，ワードデータでは w，
ロングワードデータでは l を指定する.

operand 確保するデータの個数

A.6.3.7　.dc

アセンブル時にデータ領域を確保し，そこに予め初期値を格納しておきたい場合に用いられる．通常 .data セクションに記述し，数値データのための領域確保に用いる．

シンタックス：

{*label*}　.dc{.*qualifier*}　*operand*1 {, *operand*2, ···}

説明：

label　　このラベルはデータの先頭アドレスを指す．

qualifier　.ds の場合と同様に，b，w，l のいずれかを指定する．

operand　　データの初期値

A.6.3.8　.ascii / .asciz

ディレクティブ.ascii　や.asciz　は文字列のデータ領域の確保を容易に記述できるように用意されている．後者が文字列の最後に 0x00 を付与するのに対し，前者は何も付与しない．

シンタックス：(.asciz についても同じ)

{*label*}　.ascii　’’*operand*1’’ {, ’’*operand*2’’, ···}

説明：

label　　このラベルはデータの先頭アドレスを指す．

operand　’’ 文字列’’

以下の例に示すように，ディレクティブ.dc を用いて.ascii と等価なステートメントを書こうとすると煩雑になる．なお，　.dc.b "ABCDf" や　.dc.b ’ABCDf’ はエラーとなる．

生成されるバイト列	ディレクティブステートメント
0A	.dc.b　10
00 FF	.dc　0xFF
00 FF	.dc.w　0xFF
00 0A 00 05 00 07	.dc.w　10,5,7
41 42 43 44 66	.dc.b　’A’,’B’,’C’,’D’,’f’
41 42 43 44 66	.ascii　"ABCDf"
41 42 43 44 66 00	.asciz　"ABCDf"

A.6.3.9　.org

プログラムの先頭の絶対アドレスを指定するディレクティブであるが，本実験ではアセンブル時のオプション (-t 400) により実行開始番地を指定するため，　.org 命令を使用してはならない．

A.7 プログラム作成上の注意事項

次のような間違いは構文的には正しいため，アセンブル時にはエラーとして検出されない．しかし，これらはプログラムの暴走の原因となることが多いため，自分で入念にチェックすること．

A.7.1 定数値の前の # を忘れていないか？.

即値（イミディエイト）モードの # の付け忘れは非常に多い．

```
×   move.b  1,%D0  /* 1 番地の値を%D0 に代入．*/
→   move.b  #1,%D0  /* 1 という値を%D0 に代入．*/
```

A.7.2 データのサイズを表すサフィックス .b .w .l は正しいか？

全てのニーモニックに添字（サフィックス）を明示することが望ましい．サイズを間違えると連続したデータの値が破壊されて正しく動作しなかったり，アドレスエラーの例外が生じたりすることがある．

```
×   movem   %D0-%D7/%A0-%A6,-(%SP) /* 省略時は .w なので，上位ワードが保存されない．*/
→   movem.l %D0-%D7/%A0-%A6,-(%SP)
```

A.7.3 ワード，ロングデータが奇数番地に配置されていないか？

奇数番地を .w または .l でアクセスするとアドレスエラーが生じる．

```
→    .even
LABEL: .dc.w    1  /* LABEL の値は偶数となる．  */
```

A.7.4 サブルーチンの引数の仕様と呼び出し側のレジスタ設定が一致しているか？

サブルーチンやシステムコールを呼ぶ際に与える引数をどのレジスタに割り当てるかを班で話し合って決めておく．

A.7.5 ユーザモードで特権命令を使用していないか？

特権命令を含んだサブルーチン (inQ, outQ など) を呼んでいる場合は気付きにくい．

```
move.w  #0x0,%SR  /* ユーザモードへ移行．*/
...
   ×   move.w  #0x2700,%SR  /* SR の操作．特権違反．（アドレスエラー）*/
```

A.7.6 不要な割り込みが入っていないか？

上記のような様々な原因で各種の例外が発生するため，一応例外ベクタの全てのエントリを設定しておいた方が安全である．

```
NONE:   rte  /* なにもしない．*/
```

さらに，step 数の早い段階では，使用しない割り込みを禁止（マスク）しておくこと．

A.7.7　レジスタ退避に誤りがないか？

レジスタ退避は単純なミスですが，気づきにくいものです．退避すべきレジスタ，退避する必要が無い，あるいは退避させてはいけないレジスタをしっかり考える．また，退避したレジスタの復帰忘れはないか細かくチェックすべきである．

A.7.8　プログラムの最後

プログラムの最後には改行が必要である．

付 録B　MC68VZ328の機能説明

米 Motorola 社が開発した MC68VZ328 (“*Dragonball™VZ*”) は 1 つの LSI の中に，68000 互換の CPU と，いくつかの周辺デバイス（チップ選択回路，割り込みコントローラ，RS-232C または赤外線による送受信インターフェース，タイマ，カラー TFT 液晶ディスプレイコントローラ等）を内蔵した多機能 1 チップマイコンである．PDA(電子手帳) を始め，電子玩具，携帯ゲーム機，MP3 プレーヤー，情報家電，測深機，ナビゲーションシステム，携帯電話などのモバイル端末装置への組み込み用途に広く用いられている．

本章では，68000CPU ファミリに共通する割り込み処理のための機能と，MC68VZ328 固有の内蔵デバイスについて概説し，割り込み処理を駆動するために必要な設定の例を挙げる．

B.1　CPUの走行モード

B.1.1　スーパーバイザモードとユーザモード

ユーザプログラムの不具合による影響をある程度抑えてシステムの信頼性を高めるため，68000CPU は，通常のユーザープログラムを実行するユーザモードと OS などのシステムプログラムを実行する特権状態 (監督者(スーパーバイザ)モード) の 2 つの走行モードを備えている．システム全体の動作に影響を与えるいくつかの処理の実行や，ハードウェアデバイスを取り扱う特殊なメモリ領域へのアクセスは，CPU がスーパーバイザモードにある場合に限って許されるようになっている (表 B.1).

表 B.1: ユーザーモードとスーパーバイザモード

	スーパバイザモード	ユーザーモード
%SR の bit 13	1	0
特権命令	実行可能	実行不可
内部デバイスレジスタへのアクセス	可	不可
%SP	スーパーバイザスタックポインタ	ユーザースタックポインタ
本実験での用途	初期化，キュー，割り込み処理	メインプログラム (Step9 以降)

ステータスレジスタ %SR の bit 13 が 1 の状態は，スーパーバイザモードと呼ばれる．OS 等のシステムプログラムや割り込み処理ルーチンは全てスーパーバイザモードで動作する．CPU の起動時，すなわち本実験での初期化プログラム実行時もスーパーバイザモードである．

%SR の bit 13 が 0 の状態は，ユーザモードと呼ばれる．通常，ユーザプログラムはユーザモードで動作する．本実験では簡易 OS の完成後，メインルーチンのテストプログラムをユーザモードで動作させる．

B.1.2　特権命令

68000 の特権命令を表 B.2 に示す．これらの命令はスーパーバイザモードでのみ使用できる．もしユーザモードで特権命令を使用しようとした場合は，特権違反の例外 (Privilege Violation) が発生する．

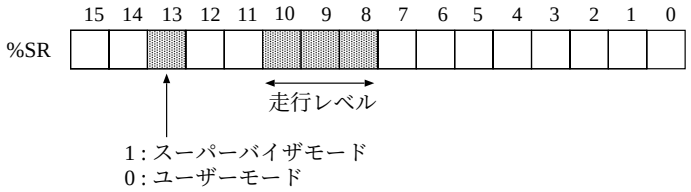


図 B.1: ステータスレジスタ %SR

表 B.2: 68000 の特権命令

処理内容	特権命令
%SR の操作	`move.w <ea>,%SR`
	`andi.w #<data>,%SR`
	`eori.w #<data>,%SR`
	`ori.w #<data>,%SR`
%USP の操作	`move.l <ea>,%USP`
	`move.l %USP,<ea>`
割り込み処理からの復帰	`rte`
外部デバイスのリセット	`reset`
プロセッサの停止	`stop #<data>`

B.1.3 スーバーバイザスタックとユーザスタック

68000 にはスーパーバイザモード用とユーザモード用に 1 つずつ，計 2 つのスタックポインタが内蔵されており，ステータスレジスタ %SR の bit 13 の値に応じて自動的に切り換わるようになっている[1)]．スーパーバイザスタックは，主に OS において割り込み処理時の実行環境の保存などに用いられる．一方，ユーザスタックは，サブルーチンコール時の戻り番地の保存やサブルーチンへ引数を渡す際に利用される．

2 つのスタックのためのメモリ領域を確保することはプログラマの責任である．自分のプログラムが使用するスタックの最大使用量を見積って充分な大きさのメモリ領域を確保する必要がある．データ領域の中にスタックのための領域を余裕を持って確保しておき，プログラム開始直後にスタック領域の末尾のアドレスをスタックポインタ %SP にセットすること．

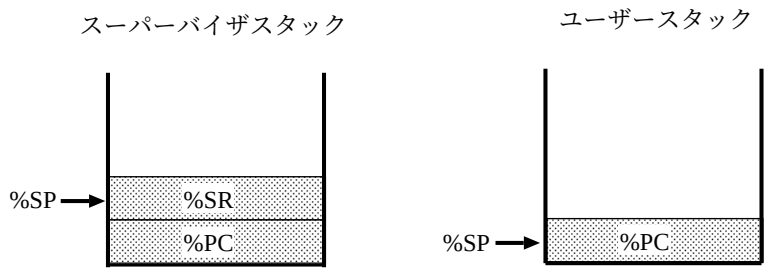


図 B.2: スタック領域

1) アセンブリ言語ではどちらも通常は %SP と記述するが，スーパーバイザモードではユーザモードスタックポインタを %USP という名前で参照することができる．

B.2 MC68VZ328 の割り込み処理機能

B.2.1 割り込み発生時の CPU の動作

68000 CPU には割り込み処理の機能が組み込まれており，ハードウェアやソフトウェアからの割り込み信号を受理すると実行中のプログラムを中断し，最後に実行した命令と次の命令を実行するまでの間に，暗黙のうちに次のような処理を実行するようになっている．

1. スーパーバイザモードへ移行.
2. 割り込みレベルとベクタ番号を取得，走行レベルを変更.
3. %SR と%PC の値をシステムスタックに退避.
4. 割り込みベクタに書かれているアドレスへジャンプ.

その後 CPU は，割り込み処理ルーチンを，スーパーバイザモードのまま割り込みレベルと同じ走行レベルで実行する．割り込み処理の最後で rte 命令が実行されると，CPU はシステムスタックからステータスレジスタ %SR とプログラムカウンタ %PC を復帰し，以前の走行レベルに戻って，中断していた処理の実行を何事もなかったかのように再開する．

B.2.2 走行レベルと割り込みレベル

B.2.2.1 CPU の走行レベル

ステータスレジスタ %SR の bit 8–10 は 0〜7 の値をとる CPU の現在の走行レベルを表す 3 ビットの値を保持する．CPU は，ユーザプログラム実行中には通常最も低いレベルであるレベル 0 で走行する．ハードウェアから割り込みが入ると，そのデバイスによって決められたレベルに自動的に遷移する．

B.2.2.2 ハードウェア割り込みのレベル

各種デバイスからの割り込みには，割り込みレベルと呼ばれる優先度が設定されている．本実験で使用する CPU の内部デバイスからの割り込みのレベルを表 B.3 に示す.

表 B.3: MC68VZ328 のハードウェア割り込みのレベル

デバイス	レベル
送受信割り込み 1 (UART1)	レベル 4
タイマ割り込み 1 (タイマ 1)	レベル 6

B.2.2.3 多重割り込みの処理

CPU が低いレベルで走行中にそれよりも高いレベルの割り込みが入ると，低いレベルの処理が中断されて高いレベルの方の割り込み処理が起動される．このとき CPU の走行レベルは高い方のレベルへ遷移する．そして割り込み処理が終了すれば，元の低い走行レベルへ戻り，中断されていた低いレベルの処理が再開される (図 B.3).

割り込み処理中に多重に割り込みが入った場合も同様に，走行中のレベルよりも新しい割り込みのレベルの方が高ければ，新しい割り込み処理の方が先に実行される．例えば, 受信割り込み処理 (レベル 4) の途中でタ

イマ割り込み (レベル 6) が発生した場合, CPU は受信割り込み処理を一旦中断しタイマ割り込みを実行する. 受信割り込み処理はタイマ割り込み終了後に再開される (図 B.4).

もし新しい割り込みのレベルが走行レベルよりも低かった場合には，新しい割り込みの処理は後回しにされ，現在実行中の割り込み処理が終了した後に，新しい割り込み処理が行われる.

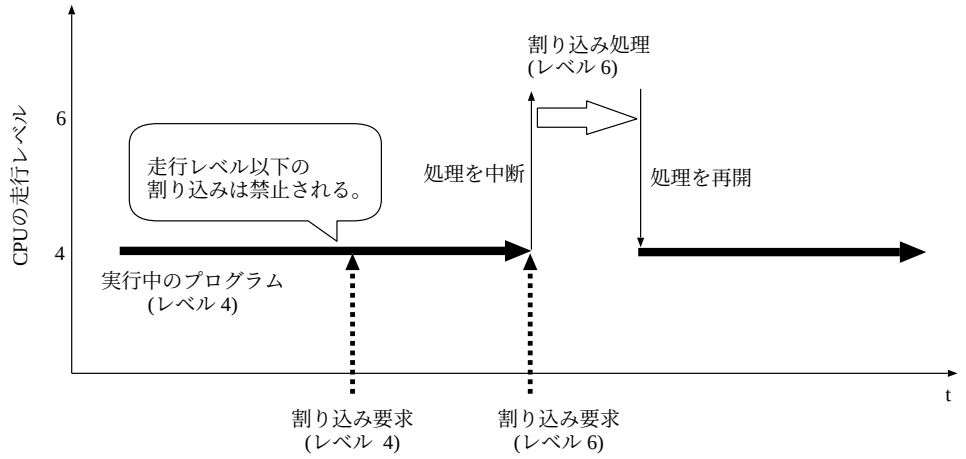


図 B.3: 割り込み処理の例

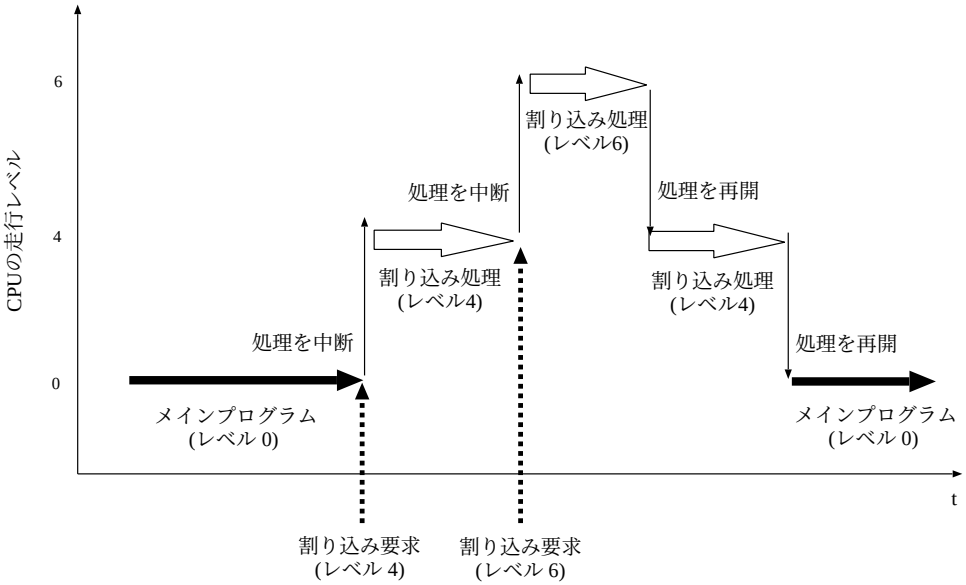


図 B.4: 割り込み処理中に他の割り込みが入った場合

B.2.2.4 割り込みの禁止

割り込み処理は現在の走行レベルより優先度 (レベル) が高い場合のみ実行される．例えば，CPU の現在の走行レベルが 4 である場合, レベル 4 以下の割り込みは全て禁止され, レベル 5 以上の場合のみ受け入れられる (図 B.3)．したがって, 全ての割り込み処理が実行されないようにするためには，走行レベルを 7 にすればよい．

起動時の CPU の走行レベルも 7 である．初期化ルーチンでは，インタラプトコントローラや各種デバイス，後述の割り込みベクタの設定等が全て完了するまで，全てのデバイスからの割り込みを禁止しておかなければならない．全ての初期化設定が終わった後で初めて走行レベルを 0 にし，割り込みを許可する．

```
move.w #0x2700,%SR     /* 割り込み禁止．*/
move.w #0x2000,%SR     /* 割り込み許可．（スーパーバイザモードの場合）*/
```

B.2.3 割り込み処理ルーチンの作成

割り込み処理ルーチン[2] は，プログラム中で初期化ルーチンやメインルーチンよりも後の位置に配置し，一般的に次のような形で準備する．

```
interrupt:                              /* この割り込み処理の開始アドレスのラベル */
    movem.l %D0-%D7/%A0-%A6,-(%SP)      /*  使用するレジスタをスタックに保存．*/
    ...
    /* ここで割り込みの原因となった事象に対処する処理を行う．  */
    ...
    movem.l (%SP)+, %D0-%D7/%A0-%A6     /* スタックからレジスタの値を復帰．*/
    rte                                 /* 割り込み処理終了．  */
```

B.2.3.1 レジスタの保存と復元

割り込み処理ルーチンの干渉を受けずにユーザプログラムの処理を正しく行うためには，ユーザプログラムで使用中のレジスタの値が割り込み処理によって変更されないようにしなければならない．そのため割り込み処理ルーチン内では，予め変更する可能性のあるレジスタを全て最初に保存し，処理後に元の値に戻す必要がある．レジスタの保存・復元処理は自動的には行われないため，プログラマの責任で割り込み処理ルーチンの中に明示的に記述しておかなければならない．通常，複数のレジスタを一度に保存・復元するために，movem 命令が用いられる．ただし，システムコール等で値をユーザプログラムへ返す必要があるレジスタに限っては，保存・復元しないようにする場合もある．

B.2.3.2 rte 命令

rte(ReTurn from Exception) 命令は，%SR と %PC をシステムスタックから掘り出 ($\overset{pop}{}$) して割り込み処理から復帰する命令である．割り込み処理ルーチンからの復帰では，rts(ReTurn from Subroutine) 命令ではなく，rte 命令を使用する．

---

[2] 割り込みハンドラとも呼ばれる．

B.2.4 割り込みベクタの設定

割り込みベクタあるいは例外ベクタは，割り込み処理ルーチンの開始アドレスを格納するための配列である．ハードウェアからの割り込みや各種の例外 (後述) が発生すると，CPU は対応するベクタを参照し，割り込み処理ルーチンを自動的に駆動するようになっている．

68000 では，例外ベクタはメモリ空間の中で 0 番地から始まる領域を占めている．1 つのアドレス値に 4 バイトを要するため，各項目のアドレスは ベクタ番号 * 4 で求められる．

MC68VZ328 では例外ベクタが拡張されており，ベクタ番号 64 番以降のユーザ割り込みベクタ領域に，1 レベルに 1 ベクタが対応した 7 つのユーザ割り込みエントリを持つことができる．この実験では各レベルに対するエントリにベクタ番号 64 以降を割り当てるものとする．そうした場合の MC68VZ328 の例外ベクタ全体を表 B.4 に示す．

表 B.4: MC68VX328 の例外ベクタ

ベクタ番号	アドレス (16 進)	対応する例外
—	0x000	リセット：スーパバイザスタックポインタの初期化
—	0x004	リセット：プログラムカウンタの初期化
2	0x008	バスエラー (Bus Error)
3	0x00C	アドレスエラー (Address Error)
4	0x010	不当命令 (Illegal Instruction)
5	0x014	0 による除算 (Divide by 0)
6	0x018	CHK 命令
7	0x01C	TRAPV 命令
8	0x020	特権違反 (Privilege Violation)
9	0x024	トレース (Trace)
10	0x028	ライン 1010 エミュレータ (Line 1010)
11	0x02C	ライン 1111 エミュレータ (Line 1111)
12〜14	0x030〜0x038	（未定義，予約済み）
15	0x03C	非初期化割り込み (Uninitialized)
16〜23	0x038〜0x05C	（未定義，予約済み）
24	0x060	スプリアス割り込み (Spurious)
25〜31	0x064〜0x07C	オートベクタ割り込み
32〜47	0x080〜0x0BC	TRAP 命令ベクタ (Trap 0 〜 15)
48〜63	0x0C0〜0x0FC	（未定義，予約済み）
64	0x100	(未使用)
65	0x104	レベル 1 ユーザ割り込み
66	0x108	レベル 2 ユーザ割り込み
67	0x10C	レベル 3 ユーザ割り込み
68	0x110	レベル 4 ユーザ割り込み (UART1)
69	0x114	レベル 5 ユーザ割り込み
70	0x118	レベル 6 ユーザ割り込み (タイマ 1)
71	0x11C	レベル 7 ユーザ割り込み

初期化ルーチンの中で割り込み処理ルーチンを割り込みベクタに登録するには，割り込み処理ルーチンの開始アドレスを move 命令などを使って対応するベクタに書き込めばよい．

```
move.l #uart1_interrupt, 0x110 /* level 4, (64+4)*4 */
```

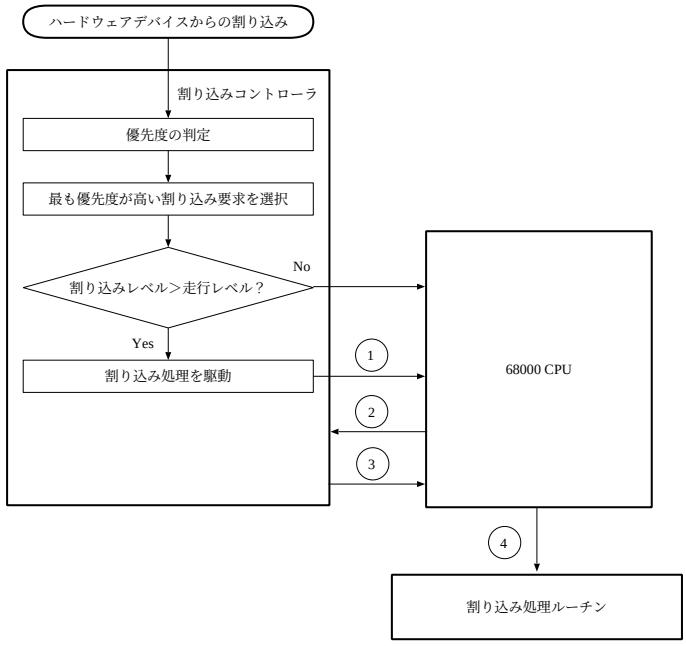
例外とは，CPU が直面する様々な異常事態やシステムコールのことである．MC68VZ328 において発生する主な例外とその原因を表 B.5 に示す.

表 B.5: MC68VZ328 で発生する主な例外事象

例外	例外発生の原因
バスエラー (2)	メモリの割り当てられていない範囲をアクセスした場合や，チップからの応答がないか，応答が非常に遅い場合に発生する．
アドレスエラー (3)	68000 は 16bit のマイクロプロセッサであるため，構造的にワードやロングのデータを奇数番地に配置することができない．ワードやロングのサイズで奇数番地をアクセスした場合には，アドレスエラーの例外が発生する．即値（イミディエイト）モードの # を書き忘れたときに生じることが多い．アドレスレジスタ間接アドレシングの場合は分かりにくい場合がある． × `move.w 1,%D0 /* 1 番地にワード単位でアクセス．*/` ○ `move.w #1,%D0 /* 1 という値を%D0 へ代入．*/` ？ `move.w #1,(%A0) /* %A0 が奇数なら×．*/` プログラムやデータが偶数番地に配置されるようにするためには，`.even` ディレクティブを用いるとよい．
不当命令 (4)	現在のプログラムカウンタ %PC の指すメモリ中の値が，命令コードとして解釈できない場合に発生する．これが発生した場合には，プログラムの流れの制御の誤り等で %PC がデータ領域に突入し，暴走している可能性が高い．
0 による除算 (5)	`divs`, `divu` 命令で分母が 0 であった場合に発生する．
特権違反 (8)	ユーザーモード (%SR のビット 13 = 0) でありながら特権命令を使用した場合に発生する．
TRAP 命令ベクタ (32〜47)	`trap` 命令はプログラムの中から割り込みを発生させる命令である．例えば，CPU がプログラム中の `trap #0` という命令を実行すると，ベクタ番号 32 の割り込みが発生し，対応した割り込み処理がスーパーバイザモードで起動される．ユーザプログラムから OS の機能を呼び出すために利用されることが多い． 本実験では `trap #0` 命令を使用してシステムコールを実現する．
ユーザ割り込み (64〜)	タイマや送受信コントローラなどのハードウェアデバイスが，内部デバイスレジスタによって設定された所定の状態にあるときに発生する割り込みである．具体的に割り込み信号を発生するために必要な各デバイスの設定については B.3 節を参照.

B.2.5 割り込みコントローラの動作

MC68VZ328 は，ハードウェアデバイスからの割り込みを処理し，CPU に伝えるための割り込みコントローラを内蔵している．ハードウェアデバイスからの割り込み信号は，図 B.5 のように処理される．



1. 割り込みコントローラはチップ内とチップ外の周辺機器からの割り込みイベントを収集する．次にそれらの割り込みを優先度順に並べ，それらのうちで最も優先度の高いものが処理中の割り込みよりも優先度が高い場合，その処理を CPU に要求する．
2. CPU はこの割り込み要求に応答し，現在実行中の命令の処理が完了した後に割り込みを承認する返答を送る．
3. 割り込みコントローラはこの割り込み承認サイクルを認識し，割り込み要求に対応するベクタ番号を CPU バスに置く．
4. CPU はこのベクタ番号を読み取り，例外ベクタテーブル中の割り込み処理ルーチンのアドレスを引き，そこから実行を始める．

図 B.5: 割り込み処理のフローチャート

ステップ 2, 4 は CPU が処理し，ステップ 1, 3 は割り込みコントローラが自動的に処理する．

ステップ 2 では，CPU のステータスレジスタ (SR) によって割り込みを隠す (マスクする) ことができる．これにより，現在どの割り込みレベルの割り込みを許可するかを決定することができる．

ステップ 4 では，CPU は割り込みベクタ番号を読み，それを 4 倍してベクタのアドレスに変換し，そのアドレスから 4 バイトのプログラムアドレスを読み，そこへジャンプする．このアドレスが割り込み処理ルーチンの最初の命令のアドレスになる．

割り込みの優先度は割り込みレベルに基づいている．同じ割り込みレベルに複数の割り込みがある場合は，割り込み処理ルーチンの中でソフトウェアによって順序を決定する必要がある．

B.3 MC68VZ328 の内部デバイスレジスタ

MC68VZ328 では，内蔵する周辺機器を制御するための内部デバイスレジスタ，プログラムやデータを保持するためのメモリ (RAM, ROM)，など様々なデバイスに専用のメモリ空間を割り当てることができる．そのうちデバイスレジスタは，0xFFF000–0xFFFDFF の範囲の領域に割り当てられている．レジスタエリアにデータを書き込むと，書き込まれたデータ中の 1 つ 1 つの 0/1 のビットが電気的なスイッチとして働き，各デバイスの機能を部分的に ON/OFF する．これを用いて，デバイスに数値パラメータを与えて設定したり，外部からのデータを取り込んだりすることなどが可能である．

本実験で作成する簡易 OS では MC68VZ328 の内部デバイスのうち次のようなものを使用する．

- 割り込みコントローラ
- タイマ
- 送受信コントローラ (UART)

本節はこれらのデバイスの機能と，それを制御する内部レジスタの仕様について述べている．その説明の中で，内部レジスタは次のような形で表現される．

略号 **内部デバイスレジスタ名** **アドレス**

	BIT 7	6	5	4	3	2	1	BIT 0
	0	0	1	1	1	1	0	1
TYPE	r	r	r	rw	rw	rw	0	1

ビット割り当ては本実験で設定が必要な部分のみ説明している．説明のないビットについては割り当て表に従い 0 か 1 を設定する．設定値は，指示に従って各ビットの値を定め，その 0/1 の列を 2 進数とみなして数値に直すことによって求められる．例えばこの場合の設定値は 0x3D となる．割り当て表の TYPE は，r はレジスタのそのビットが読み込み専用，rw はそのビットが読み書き可であることを表す．実際のデータサイズはレジスタによってそれぞれ異なり，バイト (.b) とワード (.w) とロングワード (.l) のものがある．

内部レジスタへ値を設定するためには，move 命令等を用いる．

```
move.w #設定値, デバイスレジスタのアドレス
```

レジスタには固有の名前 (略号) が付いているので，コーディング時にはそれをラベルとして使用すればよい．

なお，B.3.4 にこの実験での推奨値を表している．必ずしも示されている通りにする必要はないが，参考にするとよいだろう．

B.3.1 割り込みコントローラ

ここでは割り込みコントローラとそれに関連する信号について説明する．MC68VZ328 の割り込みコントローラは全ての内部割り込みと外部割り込みをサポートする．割り込みには 7 つのレベルが存在する．レベル 7 が最も高い優先度で，レベル 1 が最低である．本実験で使用する割り込みは以下の通りである．

- タイマーユニット 1(レベル 6)
- UART ユニット 1(レベル 4)

また，本実験では用いないが他にも次のような割り込み要因がある．

- タイマーユニット 2(レベル 1 から 6 に設定可)
- UART ユニット 2(レベル 1 から 6 に設定可)

割り込み処理を制御するには以下のレジスタを設定する必要がある．

B.3.1.1 割り込みベクタレジスタ (Interrupt Vector Register/IVR)

割り込みベクタレジスタ (IVR) は割り込みベクタナンバーの上位 5 ビットを設定するために使われる．割り込みが発生すると割り込みレベルに従ってベクタ番号 65 から 71 の例外処理ベクタの格納するアドレスが呼ばれるが，このレジスタの値を変更するとこの例外ベクタ番号を変更する事ができる．このレジスタはシステムのスタートアップ時に設定されなければならない．もし IVR が設定される前に割り込みが発生すると，未初期化割り込みとして割り込みベクタナンバー 0x0F が CPU に返される．これは割り込みベクタ 0x3C となる．

このレジスタのビット割り当ては以下の通りであり，その設定は表 B.6 に記述されている．本実験では IVR の値を 0x40 にするが，この場合の割り込みベクタの基点は 0x100 (0x40×4) に設定される．これはユーザ割り込みベクタの開始位置である．

IVR **割り込みベクタレジスタ** **0xFFF300**

	BIT 7	6	5	4	3	2	1	BIT 0
	VECTOR							
TYPE	rw	rw	rw	rw	rw	0	0	0

表 B.6: 割り込みベクタナンバ

名称	説明
VECTOR Bits 7-3	**ベクタナンバ**: 割り込みベクタナンバの上位 5 ビット

B.3.1.2 割り込みマスクレジスタ (Interrupt Mask Register/IMR)

割り込みマスクレジスタ (IMR) は特定の割り込みをマスクするために使われる．ある割り込みをマスクするためには，レジスタの対応するビットをセットする．各割り込み原因に対して 1 つの制御ビットが存在する．ある割り込みがマスクされたとき，割り込みコントローラは CPU への割り込み要求を生成しないが，その状態は割り込みペンディングレジスタで観測できる．リセット時には IMR の全てのビットは 1 であり，全ての割り込みはマスクされている．

IMR **割り込みマスクレジスタ** **0xFFF304**

	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
TYPE									rw	rw	rw	rw	rw	rw	rw	rw

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	0	0	1	1	1	1	1	1	1	1	1	1	1	MUART1	MTMR1	1
TYPE			rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

表 B.7: 割り込みマスクレジスタ

名称	説明
MUART1 Bit 2	**Mask UART1 Interrupt**: 0 = UART1 割り込みを許可．1 = 禁止.
MTMR1 Bit 1	**Mask Timer 1 Interrupt**: 0 = タイマ 1 割り込みを許可．1 = マスク.

B.3.1.3 割り込みステータスレジスタ (Interrupt Status Register/ISR)

割り込み処理中において，割り込みハンドラは割り込みステータスレジスタ (ISR) を調べることにより割り込みの要因を決定する．このレジスタのあるビットがセットされているとき，それらは対応する割り込みが CPU に送られたことを表す．同レベルの割り込み要因が複数ある場合，ソフトウェア割り込みハンドラは実行するアプリケーションに依存して割り込み間の順位付けを変更できる.

ISR **割り込みステータスレジスタ** **0xFFF30C**

	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
TYPE									rw	rw	rw	rw	rw	rw	rw	rw

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	0	0	0	0	0	0	0	0	0	0	0	0	0	UART1	TMR1	0
TYPE			rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

表 B.8: 割り込みステータスレジスタ

名称	説明
UART1 Bit 2	**UART1 Interrupt Request**: 0 = UART1 処理要求がない．1 = 処理要求あり．
TMR1 Bit 1	**Timer 1 Interrupt Status**: 0 = タイマ 1 割り込みがない．1 = 割り込みあり．

B.3.1.4 割り込みペンディングレジスタ (Interrupt Pending Register/IPR)

割り込みペンディングレジスタ (IPR) は読み取り専用であり，どの割り込みが未処理かを表す．ある割り込み源が割り込みを要求したがその割り込みが割り込みマスクレジスタによってマスクされていたとき，IPR の対応するビットがセットされ，割り込みステータスレジスタ (ISR) のビットはセットされない．マスクされていない割り込みで未処理のものがある場合は，両方のレジスタのビットがセットされる．

IPR **割り込みペンディングレジスタ** **0xFFF310**

	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
TYPE									rw	rw	rw	rw	rw	rw	rw	rw

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	0	0	0	0	0	0	0	0	0	0	0	0	0	UART1	TMR1	0
TYPE			rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

表 B.9: 割り込みペンディングレジスタ

名称	説明
UART1 Bit 2	**UART1 Interrupt Request**: 0 = UART1 処理要求がない．1 = 処理要求あり．
TMR1 Bit 1	**Timer 1 Interrupt Status**: 0 = タイマ 1 割り込みがない．1 = 割り込みあり．

B.3.2 タイマの使い方

ここでは MC68VZ328 の汎用タイマモジュールの操作法の詳細を述べる．汎用タイマモジュールは 2 つの 16 ビット汎用タイマ，プリスケーラ，比較値レジスタから構成される．

タイマの値はプリスケーラによって定められた周波数で値が 1 ずつ増やされる．タイマはその値が設定された比較値に達したときに割り込みを発生させることができる．

各タイマは 8 ビットのプリスケーラを持つ．プリスケーラへの入力はシステムクロック SYSCLK から得られた周波数またはそれを 16 で割ったものであり，プリスケーラは入力された周波数を TPRER1 レジスタで設定された値で割って出力する．2 つのタイマは縦につなぐことで 1 つの 32 ビットタイマとして使うこともできる．

汎用タイマを構成する 2 つの 16 ビットタイマは同一のものである．図 B.6 は汎用タイマのブロック図である．

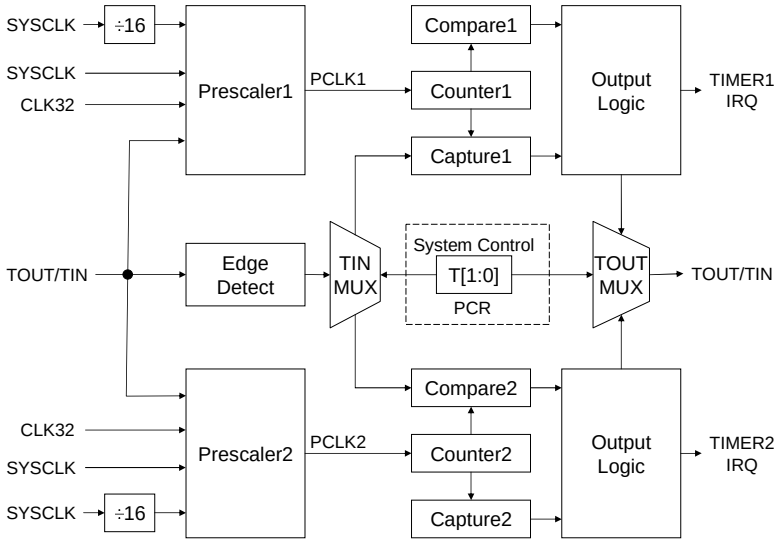


図 B.6: 汎用タイマブロック図

B.3.2.1 タイマ 1 コントロールレジスタ (Timer Control Register 1/TCTL1)

タイマ 1 コントロールレジスタ (TCTL1) は汎用タイマ 1 の全体の動作を制御するために用いられる．設定法は表 B.10 にある．TCTL レジスタは次のものを制御する．

- コンペアイベント割り込みの使用
- プリスケーラへの入力クロック
- 汎用タイマの使用

TCTL1　　タイマ 1 コントロールレジスタ　　0xFFF600

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	0	0	0	0	0	0	0	0	0	0	0	IRQEN	CLKSOURCE			TEN
TYPE								rw	rw	rw	rw	rw	rw	rw	rw	rw

B.3.2.2 タイマ 1 プリスケーラレジスタ (Timer Prescaler Register 1/TPRER1)

タイマ 1 プリスケーラレジスタ (TPRER1) はプリスケーラ 1 の分周比の設定に用いられる．レジスタの設定は表 B.11 の通り．

表 B.10: タイマ 1 コントロールレジスタ

名称	説明
IRQEN Bit 4	**Interrupt Request Enable**: 0 = 比較割り込み使用禁止．1 = 比較割り込み許可．
CLKSOURCE Bits 3-1	**Clock Source**: プリスケーラへの入力クロックの設定．000 = カウンタを止める．001 = 入力は SYSCLK．010 = 入力は SYSCLK/16．011 = 入力は TIN．1xx = 入力は CLK32．
TEN Bit 0	**Timer Enable**: 0 = タイマは禁止．1 = 許可．

TPRER1 **タイマ 1 プリスケーラレジスタ** **0xFFF602**

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	0	0	0	0	0	0	0	0	PRESCALER							
TYPE	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

表 B.11: タイマ 1 プリスケーラレジスタ

名称	説明
PRESCALER Bits 7-0	**Prescaler**: プリスケーラの出力周波数の設定．入力周波数はここで設定される値で割られる．0x00 = 1 で割る，..., 0xFF = 256 で割る．

B.3.2.3 タイマ 1 コンペアレジスタ (Timer Compare Register 1/TCMP1)

タイマ 1 コンペアレジスタ (TCMP1) はタイマ 1 のカウンタ値と比較するための値を格納する．コンペアイベントはタイマのカウンタ値がこの値と一致したときに発生する．システムリセット時にはこのレジスタの値は 0xFFFF にセットされる．

TCMP1 **タイマ 1 コンペアレジスタ** **0xFFF604**

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	COMPARE															
TYPE	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

B.3.2.4 タイマ 1 カウンタレジスタ (Time Counter Register 1/TCN1)

このレジスタは読み取り専用であり，現在のカウンタ値を持っている．このレジスタはいつでも読むことができ，カウンタの値には影響を与えない．

TCN1 **タイマ 1 カウンタレジスタ** **0xFFF608**

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	COUNT															
TYPE	r	r	r	r	r	r	r	r	r	r	r	r	r	r	r	r

B.3.2.5 タイマ 1 ステータスレジスタ (Timer Status Register 1/TSTAT1)

このレジスタはタイマの状態を表す．キャプチャイベントが発生したときは CAPT ビットがセットされる．比較イベントが発生したとき，つまりカウンタ値がコンペアレジスタ値と一致した場合，COMP ビットがセットされる．これらのビットをクリアするには 0 を書き込めばいい．ビットをクリアする場合はまずビットを読み込み値が 1 である場合だけにしなければならない．こうすることで割り込みを見逃すことがなくなる．なぜなら，もしこのように行わないと，ステータスを読み込む操作とビットをクリアする操作の間に割り込みが起きた場合にそれを見逃してしまうからである．レジスタの設定は表 B.12 の通り．

TSTAT1 **タイマ 1 ステータスレジスタ** **0xFFF60A**

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	0	0	0	0	0	0	0	0	0	0	0	0	0	0	CAPT	COMP
TYPE	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

表 B.12: タイマ 1 ステータスレジスタ

名称	説明
CAPT Bit 1	**Capture Event**: 0 = キャプチャイベントが起こっていない．1 = イベントが起こった
COMP Bit 0	**Compare Event**: 0 = コンペアイベントが起こっていない．1 = イベントが起こった

B.3.3 送受信制御 (UART)

ここでは UART (Universal Asynchronous Receiver/Transmitter) を解説する．MC68VZ328 の UART のポートは外部のシリアルデバイスと通信を行うために用いられる．

UART ではデータは標準的な “start-stop” フォーマットを用いて運ばれる．UART モジュールは start-stop 非同期通信での標準的な操作を全て行うことができる．シリアルデータは内部のボーレート生成器を用いて標準的なビットレートで送受信される．図 B.7 は UART モジュールの高レベルブロック図である．

送信機は 1 キャラクタ (バイト) を CPU バスから受け取り，それをシリアルに変換して送信する．送信データがない間は送信機は連続的にアイドル状態を出力する．送信するキャラクタがある場合，スタート，ストップ，パリティビットがキャラクタに追加され，指定されたビットレートでシフトされ下位ビットから順に送信される．

送信機は新たなデータが送信可能になったときにマスク可能な割り込みを起こす．本実験では FIFO EMPTY 割り込みを用いる．この割り込みは FIFO 中にデータがない場合に発生する．送信機は FIFO が完全に空になるまで次の割り込みを発生しない．

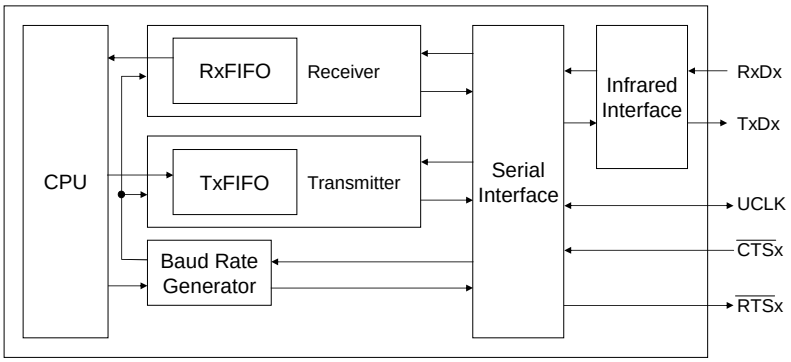


図 B.7: UART1 ブロック図

$\overline{\text{CTS1}}$ はハードウェアフロー制御で用いられるが本実験では使用しない．

UART の受信機はシリアルデータを受け取り，キャラクタに変換する．受信機は 2 つのモード (非同期，同期) で動作する．本実験で用いる非同期モードではスタートビットを見つけ，後に続くデータビットを抽出する．スタートビットが認識されたとき，残りのビットはシフトされ FIFO に読み込まれる．

パリティが許可されているとき，パリティビットがチェックされその状況は URX レジスタに報告される．フレームエラー，ブレイク，オーバーランなども同様にチェックされ報告される．4 つのステータスビットは URX レジスタの上位バイト (ビット 11-8) にあるが，これらはレジスタの下位の受信キャラクタと共に 16 ビットのワード値として読まれた場合のみ有効である．

本実験では DATA READY 割り込みを利用できる．この割り込みはデータを受信したときに発生する．URX レジスタの OLD DATA ビットは，FIFO にデータがあるが長時間読まれていないことを表す．

B.3.3.1 UART1 ステータス/コントロールレジスタ (USTCNT1)

UART1 ステータス/コントロールレジスタ (USTCNT1) は UART1 モジュールの全体の動作を制御するために用いられる．レジスタのビット割り当ては以下の通りである．レジスタの設定は表 B.13 にある．

USTCNT1 **UART1 ステータス/コントロールレジスタ** **0xFFF900**

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	U EN	RX EN	TX EN	CL KM	P EN	O DD	ST OP	8/7	OD EN	CT SD	RX FE	RX HE	RX RE	TX EE	TX HE	TX AE
TYPE	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

B.3.3.2 UART1 ボーコントロールレジスタ (UART 1 Baud Control Register/UBAUD1)

UART1 ボーコントロールレジスタ (UBAUD1) はボーレート生成器，整数プリスケーラ，UCLK 信号の動作を制御するために用いられる．ビット割り当ては以下の通りである．レジスタの設定は表 B.14 にある．

表 B.13: UART1 ステータス/コントロールレジスタ

名称	説明
UEN Bit 15	**UART1 Enable**: 0 = UART1 モジュールの使用を禁止．1 = 許可.
RXEN Bit 14	**Receiver Enable**: 0 = 受信は禁止され受信 FIFO は空になる．1 = 受信を許可.
TXEN Bit 13	**Transmitter Enable**: 0 = 送信は禁止され送信 FIFO は空になる．1 = 送信を許可.
CLKM Bit 12	**Clock Mode Selection**: 0 = 非同期．1 = 同期.
PEN Bit 11	**Parity Enable**: 0 = パリティチェックを使用しない．1 = 使用する.
ODD Bit 10	**ODD Parity**: 0 = 偶数パリティ．1 = 奇数パリティ．注: PEN = 0 の場合はこのビットは無視される.
STOP Bit 9	**Stop Bit Transmision**: 0 = ストップビット 1 ビット．1 = ストップビット 2 ビット.
8/7 Bit 8	**8- or 7-Bit**: キャラクタ長の設定．7 ビットに設定された場合，送信機は最上位ビットを無視し，受信機は最上位ビットを強制的に 0 にする．0 = 7 ビット文字の送受信．1 = 8 ビット文字の送受信.
ODEN Bit 7	**Old Data Enable**: URX レジスタ中の OLD DATA ビットがセットされていたときに割り込みを発生させるかを設定する．0 = OLD DATA 割り込みを禁止．1 = 許可.
CTSD Bit 6	$\overline{\text{CTS1}}$ **Enable**: $\overline{\text{CTS1}}$ ピンの状態が変化したときに割り込みを起こすかを設定する．0 = $\overline{\text{CTS1}}$ 割り込みを禁止．1 = 許可.
RXFE Bit 5	**Receiver Full Enable**: 受信 FIFO がいっぱいになったときに割り込みを起こすかを設定する．0 = RX FULL 割り込みを禁止．1 = 許可.
RXHE Bit 4	**Receiver Half Enable**: 受信 FIFO が半分以上埋まったときに割り込みを起こすかを設定する．0 = RX HALF 割り込みを禁止．1 = 許可.
RXRE Bit 3	**Receiver Ready Enable**: 受信 FIFO にデータが 1 バイト以上あるときに割り込みを起こすかを設定する．0 = RX 割り込みを禁止．1 = 許可.
TXEE Bit 2	**Transmitter Empty Enable**: 送信 FIFO が空のときに割り込みを起こすかを設定する．0 = TX EMPTY 割り込みを禁止．1 = 許可.
TXHE Bit 1	**Transmitter Half Empty Enable**: 送信 FIFO 中のデータ数が半分未満になったときに割り込みを起こすかを設定する．0 = TX HALF 割り込みを禁止．1 = 許可.
TXAE Bit 0	**Transmitter Available for New Data**: 送信 FIFO に空きがあるときに割り込みを起こすかを設定する．0 = TX AVAIL 割り込みを禁止．1 = 許可.

UBAUD1 **UART1 ボーコントロールレジスタ** **0xFFF902**

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	0	0	UCLK DIR	0	BAUD SRC	DIVIDE			0	0	PRESCALER					
TYPE			rw		rw	rw	rw	rw			rw	rw	rw	rw	rw	rw

表 B.14: UART1 ボーコントロールレジスタ

名称	説明
DIVIDE Bits 10-8	**Divide**: ボーレート生成器で生成されたクロック周波数を割る数を設定する．000 = 1 で割る,001 = 2 で割る,010 = 4 で割る,. . .,111 = 128 で割る．
PRESCALER Bits 5-0	**Prescaler**: プリスケーラの値を設定する．周波数を割る数は 65 - (PRESCALER)．

システムクロックが 33.16MHz の場合に一般的なボーレートを得るには表 B.15 のように DIVIDE と PRESCALER を設定すればいい．

表 B.15: ボーレートの設定

ボーレート	DIVIDE	PRESCALER
230400	0	0x38
115200	1	0x38
57600	2	0x38
28800	3	0x38
14400	4	0x38
38400	1	0x26
19200	2	0x26
9600	3	0x26
4800	4	0x26
2400	5	0x26
1200	6	0x26
600	7	0x26

B.3.3.3 UART1 受信レジスタ (UART 1 Receiver Register/URX1)

UART1 受信レジスタ (URX1) は受信 FIFO とデータの状態を示している．FIFO ステータスビットは FIFO の現在の状態を反映している．電源投入時にはこれらのビットはランダムな値になっている．受信割り込みを許可する前に，USTCNT レジスタの UEN ビットと RXEN ビットはセットされなくてはならない．UART1 受信レジスタを読み込むことで FIFO ステータスビットは初期化され，その後では受信割り込みを許可することができる．なお，キャラクタ状態ビットはキャラクタビットと共に 16 ビットのワード値として読まれた場合のみ有効である．ビット割り当ては以下の通りである．レジスタの設定は表 B.16 にある．

URX1 **UART1 受信レジスタ** **0xFFF904**

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	0	0	DATA READY	0	0	0	0	0	RX DATA							
TYPE	r	r	r	r	r	r	r	r	r	r	r	r	r	r	r	r

表 B.16: UART1 受信レジスタ

名称	説明
DATA READY Bit 13	**Data Ready (FIFO Status)**: 0 = 受信 FIFO にデータがない．1 = データがある．
RX DATA Bits 7-0	**Rx Data (Character Data)**: 受信 FIFO の最初のデータを表す．DATA READY ビットが 0 の場合は意味を成さない．

B.3.3.4 UART1 送信レジスタ (UART 1 Transmitter Register/UTX1)

UART1 送信レジスタ (UTX1) は送信機がどのように動作するかを制御あるいはどのような状態にあるかを調べるために用いられる．ビット割り当ては以下の通りである．レジスタの設定は表 B.17 にある．

UTX1 **UART1 送信レジスタ** **0xFFF906**

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	FIFO EMPTY	0	0	0	NO CTS1	BUSY	0	0	TX DATA							
TYPE	r	r	r	rw	rw	rw	rw	rw	w	w	w	w	w	w	w	w

表 B.17: UART1 送信レジスタ

名称	説明
FIFO EMPTY Bit 15	**FIFO EMPTY (FIFO Status)**: 送信 FIFO が空であるかを示す．0 = 送信 FIFO が空ではない．1 = 空である．
NOCTS1 Bit 11	**Ignore $\overline{\text{CTS1}}$ (TX Control)** : フロー制御の設定．0 = フロー制御を行う ($\overline{\text{CTS1}}$ が 0 になるまで送信を行わない)．1 = フロー制御を行わない．
BUSY Bit 10	**Busy (TX Control)**: 送信機が動作中かを表す．0 = 送信機はデータを送信中ではない．1 = データ送信中．
TX DATA Bits 7-0	**Tx Data (Character) (Write-Only)**: ここに書き込まれたデータが送信される．このビットは書き込み専用である．

注意：
UTX1に2回に分けて書き込むとターミナルに文字が表示されない(UTX1は書き込みがある度にリセットがかかるため)

B.3.4 内部デバイスレジスタの設定例

B.3.4.1 割り込みコントローラ

レジスタ名	アドレス	サイズ	説明	推奨値	コメント
IVR	0xFFF300	B	Interruput Vector Register 割り込みベクタレジスタ	0x40	0x100〜
IMR	0xFFF304	L	Interrupt Mask Register 割り込みマスクレジスタ	0xFF3FFF 0xFF3FFB 0xFF3FF9	禁止 UART1 許可 UART1+Timer1 許可
ISR	0xFFF30c	L	Interrupt Status Register 割り込みステータスレジスタ		
IPR	0xFFF310	L	Interrupt Pending Register 割り込みペンディングレジスタ		

B.3.4.2 タイマ 1

レジスタ名	アドレス	サイズ	説明	推奨値	コメント
TCTL1	0xFFF600	W	Timer1 Control Register タイマ 1 コントロールレジスタ	0x0004 0x0015	停止 割込許可
TPRER1	0xFFF602	W	Timer1 Prescaler Register タイマ 1 プリスケーラレジスタ	206	0.1msec 周期
TCMP1	0xFFF604	W	Timer1 Compare Register タイマ 1 コンペアレジスタ	50000	5 秒間隔の場合
TCN1	0xFFF608	W	Timer1 Counter Register タイマ 1 カウンタレジスタ		
TSTAT1	0xFFF60a	W	Timer1 Status Register タイマ 1 ステータスレジスタ		

B.3.4.3 送受信コントローラ 1(UART1)

レジスタ名	アドレス	サイズ	説明	推奨値	コメント
USTCNT1	0xFFF900	W	UART1 Status/Control Register UART1 ステータス/コントロールレジスタ	0x0000 0xE100 0xE108 0xE10C	reset 送受信可 +受信割込許可 +送受信割込許可
UBAUD1	0xFFF902	W	UART1 Baud Control Register UART1 ボーコントロールレジスタ	0x0126	38400 bps
URX1	0xFFF904	W	UART1 Receiver Register UART1 受信レジスタ		下位バイト= ASCII コード
UTX1	0xFFF906	W	UART1 Transmitter Register UART1 送信レジスタ	0x08XX	XX=ASCII コード

本実験では，シリアルポート COM1 の通信条件を，1 stop bit, 8bit character, no parity, 38400bps，に設定する必要があるが，これは Step1 の初期化ルーチン (29 頁〜) での USTCNT1 と UBAUD1 の設定をそのまま用いればよい．また，フロー制御は行わない．

プログラムからデータを送受信するためには，UTX1 と URX1 の下位バイトに文字の ASCII コードを読み書きする．Shift-JIS コードを使用して日本語を表示させることも可能である．その場合は上位バイトと下位バイトの 2 回に分けて書き込む必要があるが，これは interput によって自動的に行われる．

送信割り込みを使用する際には，USTCNT1 の TXEE ビット (bit 2) で送信バッファが空になったとき割り込みが発生するように設定する．一方，受信割り込みを使用する際には，RXRE ビット (bit 3) で受信バッファにデータが 1 バイト以上入っているときに割り込みが発生するように設定する．よってハードウェアによる送受信バッファは 1 文字分のみを使用することになる．

付 録C　送受信チャンネルの2チャンネル化

本実験で使用する CPU ボードにはシリアルポート (RS-232C) が 2 チャンネル搭載されている．それぞれのチャンネルで送受信を行うことが可能であり，UART1，UART2 という名称で区別されている．本実験ではシリアルポート 1 での送受信すなわち UART1 のみを用いて簡易 OS の開発を行うが，本実験に引き続いて行われる後半のソフト実験では，応用課題のひとつにポートを 2 つ利用することが挙げられるため，ここでは 2 チャンネル化について簡単に説明する．

C.1　割り込みベクタの設定

UART2 の割り込みはレベル 5（変更可）のユーザ割り込みに設定する．割り込みベクタ番号とアドレスについては，表 B.4 を参照のこと．

C.2　内部デバイスレジスタの設定

C.2.1　割り込みコントローラ

割り込みマスクレジスタ (IMR)，割り込みステータスレジスタ (ISR)，割り込みペンディングレジスタ (IPR) はそれぞれが，Bit 2 が UART1 に対応し，Bit 12 が UART2 に対応している．従って，UART2 を利用する際には，Bit 12 の値を操作すればよい．

表 C.1: 割り込みマスクレジスタ

名称	説明
MUART2 Bit 12	**Mask UART2 Interrupt**: 0 = UART2 割り込みを許可．1 = 禁止.

表 C.2: 割り込みステータスレジスタ

名称	説明
UART2 Bit 12	**UART2 Interrupt Request**: 0 = UART2 処理要求がない．1 = 処理要求あり.

表 C.3: 割り込みペンディングレジスタ

名称	説明
UART2 Bit 12	**UART2 Interrupt Request**: 0 = UART2 処理要求がない．1 = 処理要求あり．

C.2.2 送受信制御 (UART)

UART2 は UART1 と同様のレジスタ構成を持つ．従って，UART2 のステータス/コントロールレジスタ (USTCNT2)，ボーコントロールレジスタ (UBAUD2)，受信レジスタ (URX2)，送信レジスタ (UTX2) の構成については，付録 B.3.3 を参考にするとよい．ただし，UART2 レジスタのアドレスは，0xFFF910 から始まるので注意されたい．

C.2.3 内部デバイスレジスタの設定例

以下に，UART2 の使用を想定した内部デバイスレジスタの設定例を示す．

C.2.3.1 割り込みコントローラ

レジスタ名	アドレス	サイズ	説明	推奨値	コメント
IVR	0xFFF300	B	Interruput Vector Register 割り込みベクタレジスタ	0x40	0x100〜
IMR	0xFFF304	L	Interrupt Mask Register 割り込みマスクレジスタ	0xFF3FFF 0xFF3FFB 0xFF3FF9 0xFF2FF9	禁止 UART1 許可 +Timer1 許可 +UART2 許可
ISR	0xFFF30c	L	Interrupt Status Register 割り込みステータスレジスタ		
IPR	0xFFF310	L	Interrupt Pending Register 割り込みペンディングレジスタ		

C.2.3.2　送受信コントローラ 2(UART2)

レジスタ名	アドレス	サイズ	説明	推奨値	コメント
USTCNT2	0xFFF910	W	UART2 Status/Control Register UART2 ステータス/コントロール レジスタ	0x0000 0xE100 0xE108 0xE10C	reset 送受信可 +受信割込許可 +送受信割込許可
UBAUD2	0xFFF912	W	UART2 Baud Control Register UART2 ボーコントロールレジスタ	0x0126	38400 bps
URX2	0xFFF914	W	UART2 Receiver Register UART2 受信レジスタ		下位バイト= ASCII コード
UTX2	0xFFF916	W	UART2 Transmitter Register UART2 送信レジスタ	0x08XX	XX=ASCII コード

C.3　その他の変更点

後半のソフト実験では，本実験で作成したプログラムを 2 チャンネルに対応したプログラムに変更する必要がある．具体的には，キューの使用数，INTERPUT/INTERGET，PUTSTRING/GETSTRING がこれに相当するが，本実験でプログラムを作成する際に，Step 4 から Step 8 で作成した各ルーチンにチャンネルを表す引数 ch を持たせるように作成していれば，2 チャンネル化は比較的容易に達成できる．

C.3.1　キューの追加

UART1 のみを使用するときは，送受信用にキューをひとつずつ計 2 つ利用していたが，チャンネルがひとつ増えるのでそれに応じてキューも送受信用に 2 つ追加する．INQ，OUTQ の第 1 引数 no には，キューの番号を指定するように設計されているので，UART2 用に新たに追加したキューについても，この引数で番号を指定することが可能である．

C.3.2　送受信割り込みルーチン

送受信割り込みルーチン INTERPUT，INTERGET にも,（テキスト通りにプログラムを作成していれば）チャンネルを指定するための引数 ch がある．これまでの INTERPUT，INTERGET の機能では,「チャンネル ch が 0 以外の場合は何もしない」となっていたが，2 チャンネル使用する場合は,「チャンネル ch が 0 または 1 以外の場合は何もしない」に変更し，ch = 1 の場合の送信キューからのデータの取り出し，あるいは受信キューへのデータの書き込みを行う処理を追加すればよい．

C.3.3　送受信割り込み用のハードウェア割り込みインタフェース

UART1 のときと同様に，UART2 用にもハードウェア割り込みインタフェースを作成する．UART2 用に作成した割り込みインタフェースが UART2 の割り込みによって呼び出されるように割り込みベクタの設定をしておく必要がある．送受信のどちらが発生したかは，UTX2 の第 15 ビット目あるいは，URX2 の第 13 ビット目を用いてチェックし，割り込み内容に応じて，ch = %D1.L = 1 として INTERPUT あるいは INTERGET を呼び出す．送受信レジスタに関する注意点については，3.8.2 と 3.10.3 を参照のこと．

C.3.4 送受信制御部

送受信制御部 PUTSTRING，GETSTRING にもチャンネルを指定するための引数 ch があるので，チャンネル ch が 1 であったときようの処理を追加すればよい．チャンネルによる処理内容の違いは，利用するキューの番号が違うことと，INTERPUT では USTCNT2 を操作して，送信割り込みを許可する点である．

付 録D　作業日報の書き方

D.1　記入内容

作業日報は各班で 1 枚，毎回の実験後に提出してもらう．作業日報に各内容は以下の通り．

グループ

- 班の名前を記入（例：1 班）．

日時

- 実験日と時間帯を記入（例：○月×日（△）　○○時○○分〜○○時○○分）．

出席者/欠席者

- その日の出席者と欠席者を記入．

作業日報作成者

- 作業日報作成者を記入．毎回同じ者が作成するのではなく，交代で作成すること．

進捗状況

- 担当者，担当内容，作業時間，作業状況について記入．
- 誰がどの作業を担当し，その作業が予定通りに進んだかどうかなど，詳しく書くこと．

次回予定

- 担当者，担当内容，作業目標について記入．
- 次回の作業予定について，今回の作業状況等を踏まえて，作業目標を記入すること．

問題点等

- 今回の作業でうまく行かなかった点，疑わしき点などを記入．
- できるだけ詳しく書いておくこと．次回，教員や TA が問題点についてアドバイスしやすくなるので．

D.2　作業日報記入例

図 D.1 に作業日報の記入例を示す．

作業日報（ソフトウェア実験1）

グループ	0 班	確認	
日 時	10 月 21日 （ 火 ）	14 時 50 分 ～	17 時 30 分
出席者	○○ ○○, △△ △△ □□ □□, ☆☆ ☆☆	欠席者	
作業日報作成者	○○ ○○		

進捗状況	担当者	担当内容	作業時間	作業状況
	○○ ○○ △△ △△	・初期化ルーチンの作成 初期化ルーチンを作成し，正常動作していることを確認した．	2時間30分	完了
	□□ □□ ☆☆ ☆☆	・キューの作成 キューの初期化ルーチンとキューへの入力（INQ），出力（OUTQ）ルーチンを作成した．キューの正常動作確認で手順2までは動作が確認された．手順3で，129回目のOUTQが失敗しない！	2時間30分	動作確認中

次回予定	担当者	担当内容	作業目標
	○○ ○○ △△ △△	・受信割り込みのテスト	受信割り込みについて理解を深めて，実際に割り込みテストプログラムを作成する．
	□□ □□ ☆☆ ☆☆	・キューの作成（続き）	キューの正常動作確認手順3がうまく行かない原因を突き止めて，プログラムを修正する．できるだけ次回のうちに，キューの正常動作確認をすべて終わらせる．

問題点等
キューの正常動作確認手順3で，INQは128回成功するが，OUTQを129回すると，129回目までもOUTQが成功したことになってしまう．原因を突き止めようと，いろいろ工夫したが今回は解決に至らなかった．何かいいデバッグ方法があれば教えてもらいたい．

図 D.1: 作業日報の記入例

関連図書

[1] MOTOROLA : MC68VZ328 Integrated Processor User's Manual MC68VZ328UM/D, Rev.0, 02/2000

[2] OSRAM Opto Semiconductors Inc.: PD4435 data sheet, July 5, 2001-14